Augmenting Eclipse with Dynamic Information

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Marcel Härry

Mai 2010

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz

David Röthlisberger

Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Marcel Härry
mhy@students.unibe.ch
http://scg.unibe.ch/research/senseo

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Abstract

Traditional IDEs such as Eclipse provide a broad range of supportive tools and views to manage and maintain software projects. However, they provide developers mainly with static views on the source code neglecting any information about runtime behavior. As object-oriented programs heavily rely on polymorphism and late-binding, it is difficult to understand such programs only based on their static structure. Developers therefore tend to gather runtime information with debuggers or profilers to reason about dynamic information. Information gathered using such procedures is volatile and cannot be exploited to support developers navigating the source space to analyze and comprehend the software system or to accomplish other typical software maintenance tasks.

In this thesis we present an approach to augment static source perspectives of Eclipse with dynamic information such as precise runtime type information or memory and object allocation statistics. Dynamic information can leverage the understanding for the behavior and structure of a system. We rely on dynamic data gathering based on aspects to analyze running Java systems.

To integrate dynamic information into Eclipse we implemented a plugin extending the Eclipse Java Development Toolkit (JDT) called *Senseo*. This plugin augments existing IDE tools of Eclipse and several standard views of JDT such as the Package Explorer with dynamic information. Besides these enrichments, *Senseo* provides several visualizations such as an overview of the collaboration within the software system. We comprehensively report on the efficiency of our approach to gather dynamic information. To evaluate our approach we conducted a controlled experiment with 30 professional developers. The results show that the availability of dynamic information in the Eclipse IDE yields for typical software maintenance tasks a significant 17.5% decrease of time spent while significantly increasing the correctness of the solutions by 33.5%.

Acknowledgements

The presented work was completed with the support of many people. I would like to express my gratitude to everyone that supported me during the time I was working on this thesis. Everyone of you helped me to complete this thesis.

David Röthlisberger supervised me during this thesis and his ideas, input and motivation kept me on track. This work would not have been possible without him. I enjoyed the various discussions we had many times over lunch as well as appreciated the constructive critic he gave on my thesis.

Prof. Oscar Nierstrasz for letting me write my thesis in the Software Composition Group (SCG). His inspirational lectures revealed many interesting aspects of software engineering and kept me interested in this field.

All the SCG staff and students for their advice and contributions to this work and the great time we had. Having lunch together or working at the SCG student's pool was always fun and interesting.

Special thanks goes to everyone that participated in the controlled experiment and shared some of their spare time to validate our approach and giving further input and ideas.

Also all my friends that did not see me a lot in the last months, but still supported me in finishing my work and taking the load off of me. I appreciate your unconditional support and this thesis would not have been finished without you!

My parents and my brother for all their support in the last years and enabling me to study at the university. I would not be anywhere near here without your ongoing support!

Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As an introduction we briefly outline the shortcomings of static perspectives on a software system in IDEs, such as the limited support to analyze runtime behavior, and argue for the integration of dynamic information into the IDE. We then briefly present *Senseo*, an Eclipse plugin addressing the discussed shortcomings. Finally, we outline the various contributions included in this thesis and describe its structure.

1.1 Shortcomings of Traditional IDE Perspectives

Traditional IDEs such as Eclipse provide multiple perspectives on a software system. These perspectives aim to support developers in various activities during software development and maintenance. However, such perspectives are mainly based on information obtained through static analysis, which processes source code to interpret the structure and behavior of the studied software system. Static analysis algorithms such as the ACP algorithm [65] which infers type information of Java systems, try to approximate the behavior of the software system based on the acquired information [38]. However, object-oriented programs heavily rely on language features like inheritance, interface or abstract types, late-binding, or polymorphism. This means that the behavior of object-oriented systems can only be completely determined at runtime [29]. For example, views based on static source code analysis cannot disclose precise information about all types bound to a certain variable at runtime. While searching for implementors of a certain interface type, static views reveal all candidate implementors, but at runtime actually only one specific implementation is used. Such inexact search results mislead developers in their assumption about a source artifact and what parts of the software system are relevant for the task at hand. We conclude that views based on static source code analysis are *imprecise*.

As it is crucial for program comprehension to disclose any aspect of a software system [8], such perspectives are not sufficient to understand the behavior of an object-oriented program. Furthermore, we state that features of object-oriented languages such as polymorphism necessitate the access to dynamic information to precisely reveal all characteristics of an object-oriented software system. Other researchers also report that having dynamic information available is of great value for typical software maintenance tasks [40].

Furthermore, in most software systems conceptually related code is scattered over many different source artifacts, *e.g.*, classes and methods. It is therefore difficult to understand how an application is implemented purely by navigating and browsing the source code [58]. Additionally, as static perspectives cannot reveal precise information about message sends between software artifacts, it is impossible for developers to precisely reason about how different parts of the software system collaborate.

To support developers during development and maintenance of software systems, IDEs also provide means to efficiently navigate the source code space or to study source artifacts, for instance, to obtain precise information about runtime types. Analyzing the existing static perspectives revealed that several activities developers usually perform during typical software maintenance are not sufficiently supported. The following list summarizes the identified activities:

- Understanding higher-level concepts such as application layers, models, or separation of concerns.
- Identifying collaboration patterns, that is, how various source artifacts communicate with each other at runtime.
- Gaining an overview of control flow and execution complexity, for instance to quickly locate performance bottlenecks.
- 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.
- Understanding execution paths and runtime types of an object-oriented system employing complex hierarchies including abstract classes and interfaces.

As discussed static perspectives provide *imprecise* information during such activities, therefore developers usually resort to debuggers to circumvent these limitations. By setting appropriate breakpoints and stepping through the execution developers try to predict execution flow or runtime types. However, such workarounds are very cumbersome and lead to further problems: Information gathered by debuggers is *volatile*, that is, it cannot be aggregated or embedded into the IDE. Furthermore, information revealed by debuggers only represents a current situation unrelated to any other execution or to the next cycle of the loop we currently study. Such an isolated view impedes studying and comparing the current situation to other execution paths of the same feature or to other executions of a certain code artifact. Moreover, as debuggers do not reveal any additional information about collaborating parts of the software system, the inspection carried out by debuggers remains very isolated and disconnected from other parts of the execution. These limitations of debuggers complicate the inspection of scattered or tangled code and result in an *incomplete* view on the software system.

Such an *incomplete* view impedes analysis of how features or concrete source artifacts behave at runtime. For instance, performance issues of source artifacts are often only revealed in certain circumstances or in combination with other specific source artifacts. But as it remains unclear which source artifacts are used at runtime as well as which source artifacts collaborate with each other, any reasoning about performance issues in the IDE is very cumbersome and difficult.

Summary. As static perspectives provide an *imprecise* and *incomplete* view on a software system studying the following aspects of an object-oriented software system is barely supported without embedding dynamic information:

- **Runtime types** - Inheritance or interface and abstract types impede any prediction about which types are actually used at runtime.
- **Execution flow** - As various aspects of program execution are not certain until execution, any prediction of execution flow remains uncertain.
- **Performance aspects** - As only system execution reveals which code is exercised at runtime, analyzing performance bottlenecks is difficult without dynamic information.
- **Scattered code** - Determining collaboration within a system encompassing scattered code based on *imprecise* and *incomplete* information is cumbersome.

Problem statement. *Various characteristics of object-oriented software systems such as inheritance and polymorphism can only be determined at runtime. IDEs purely relying on static source code analysis cannot reveal important aspects of an object-oriented software system and provide therefore imprecise and incomplete perspectives on the system. Such static perspectives badly support developers during software maintenance.*

To address the stated problem we need to integrate solutions to analyze and visualize dynamic information in IDEs to support developers working on typical software maintenance tasks.

In this thesis we focus on supporting developers maintaining software systems in Java, a widely used statically-typed language. As we further discuss in Chapter 2, IDEs for Java such as Eclipse on which we focus our integration barely embed dynamic information up to now.

1.2 Senseo - Augmenting Eclipse with Dynamic Information

Previously, we outlined various activities that are not sufficiently supported by static perspectives of traditional IDEs, as they neglect dynamic information such as runtime types. We aim to seamlessly integrate enrichments and visualizations of different kinds dynamic information in the IDE to efficiently support developers during software maintenance. To achieve a solution for the discussed limitations we need to address different aspects of gathering, aggregating and integrating dynamic information. The following research questions give an overview on the different challenges such a solution pose:

Research Questions

- *What kind of dynamic information is useful for software maintenance?*
- *How can we gather, process, aggregate, and store dynamic information efficiently?*
- *How can we represent and integrate dynamic information into IDEs?*
- *How can we validate our approach?*

Our solution is embodied in *Senseo*, an Eclipse plugin augmenting different views of Eclipse with dynamic information. *Senseo* enriches the standard Java Development Tools (JDT)[1] with different visualizations of the gathered information.

We state our thesis as follows:

Thesis

> Integrating dynamic information into the IDE addresses shortcomings of its static source code perspectives and supports developers to more efficiently and correctly perform typical software maintenance tasks.

Senseo is our working prototype to validate our approach. To gather dynamic information *Senseo* relies on *MAJOR* [64], a tool to weave aspects in the standard Java class library (JDK). Using these aspects we are able to define what kind of dynamic information is gathered. To gather, process and store dynamic information efficiently, *MAJOR* is directly integrated into *Senseo*. This integration embeds the process of information gathering seamlessly into the work processes developers usually perform within Eclipse. While running in the instrumented Java VM, *MAJOR* collects several kinds of dynamic information of the studied software system and transmits these to the *Senseo* plugin. Subsequently, the received information is parsed and imported in our data storage. We are able to export the gathered data from our storage system as well as to load such exported data. This feature enables developers to archive, share and compare gathered data from multiple executions.

The integration of *MAJOR* answers our second research question and enables us to address the third question, namely to represent and integrate dynamic information into IDEs. Based on the processed and aggregated information we are able to integrate enrichments that range from visualizations on a coarse-grained level within the source code view up to higher level views providing an overview of the whole software system. The integrated visualizations disclose hotspots of the analyzed software system in the source code view as well as within the various navigation trees that come with Eclipse. Such visualizations are for example colored icons disclosing the exact number of invocations of a method, class or package.

Information about the argument types of a method or its callers and callees are disclosed by tooltips, small windows that pop up when the mouse hovers over a source element (a method name, a variable, etc.). Additionally, these visualizations are interactive and provide assistance to navigate the source space, hence developers can navigate towards source artifacts with our enrichments by clicking on an element of interest. The *CCRC*, a visualization of the gathered data tree, the Calling Context Tree (CCT), discloses the call stack and the context of the gathered dynamic information. Furthermore, we provide a *Collaboration Overview* which discloses how the different parts of the software system such as packages, classes or methods collaborate to each other.

[1] `http://eclipse.org/jdt/`

1.3 Contributions

Senseo enriches Eclipse with dynamic information to ease maintenance of object-oriented systems written in Java. The process of integrating dynamic information into IDEs includes several aspects such as *data gathering*, *storage and processing* and *enrichments and visualizations* as well as we aimed to *validate* our approach. We outline these four contributions in the following paragraphs.

Data Gathering. In this thesis we examine different challenges related to the process of gathering dynamic information such as choosing the appropriate technique to seamlessly integrate gathering of dynamic information into the IDE. Furthermore, we outline different approaches to solve these challenges. Our choice is *MAJOR*[2] which enables us to cover all methods executed in the JVM and allows us to select what kind of information should be collected. We subsume its architecture and advantages compared to other techniques and validate by means of a benchmark its performance. *Senseo* integrates *MAJOR* into Eclipse and enables developers to dynamically analyze their applications within Eclipse.

Storage and Processing. The gathered dynamic information has to be stored, processed and aggregated to be integrated into the IDE. *Senseo* implements its own storage system which enables us to store the gathered dynamic information of multiple projects and executions. It is possible to query the stored information on a package, class and method level. Additionally, this storage system provides different kinds of aggregated information. For instance, we group all methods of the examined software system based on their number of invocations into different clusters to easily disclose hotspots and bottlenecks. Furthermore, it is possible to store and load the gathered information to aggregate it over multiple executions and to exchange it with other developers.

Enrichments and Visualization. In this thesis we contribute several approaches of enrichments and visualizations of dynamic information in the IDE. We reveal how *Senseo* integrates aggregated information into the different views of the IDE and how these enrichments support developers during typical software maintenance tasks, as outlined in Section 1.1. Examples of such enrichments are tooltips showing dynamic information on a source code level. All contributed enrichments improve the navigation of the source code space and the search for various elements, such as precise information about all callers of the currently studied element. Additionally, we contribute several high level overviews of the software system such as a *Collaboration Overview*.

Validation. Besides a use case based validation, we validate our approach with a controlled experiment. 30 professional developers solved five typical software maintenance tasks concerned with comprehending and maintaining a representative software system. We measured time and answer correctness for both an experimental group using our Eclipse enhancements and a control group using the traditional Eclipse IDE. With this

[2]`http://www.inf.usi.ch/projects/ferrari/MAJOR.html`

experiment we show that integrating dynamic information into Eclipse yields a significant 17.5% decrease of time spent while significantly increasing the correctness of the solutions by 33.5%.

1.4 Structure of the Thesis

Our thesis has the following structure:

Chapter 2 outlines the state of the art in dynamically analyzing and visualizing software systems in IDEs. Furthermore, we present different techniques to gather dynamic information. We then briefly look at other controlled experiments in software engineering.

In **Chapter 3** we motivate our approach to integrate dynamic information into the IDE. We further examine what kind of dynamic information is required to be gathered to address the outlined problems.

Chapter 4 addresses the issue of of dynamic *information gathering* and briefly outlines some of the implementation details. This includes an overview of *MAJOR*, its inner data structure and optimizations performed to reduce the gathering overhead in terms of analyzing time and memory consumption.

In **Chapter 5** we describe the general architecture of *Senseo*: How *Senseo* integrates *MAJOR* and how it *stores and processes* the gathered information. We then outline the different enrichments and visualizations we integrated into the Eclipse IDE.

Chapter 6 discusses the controlled experiment we conducted: Experiment setup, subject selection and the statistical evaluation are outlined. Additionally, we discuss the threats to validity and conclude that the results support our claim that *Senseo* supports developers during software maintenance tasks correctness and speed. We are even able to show a significant increase in both correctness and speed in conducting typical software maintenance tasks.

Chapter 7 critically discusses the various aspects of our work on *Senseo*. We further discuss details of various parts of *Senseo*'s architecture such as the information gathering.

In **Chapter 8** we argue for the inclusion of dynamic information into IDEs to support developers in their daily work on software maintenance, such as reasoning about different parts of a software system or understanding of the underlying code. Additionally, we outline further work to extend the current integration and visualization of dynamic information in Eclipse.

Attached to this thesis is a user's guide to *Senseo* and various resources gathered during the controlled experiment.

Chapter 2

State of the Art

In this chapter we examine the state of the art in dynamically analyzing and visualizing software systems. First, we describe traditional IDEs and outline related work in improving and enriching their source code perspectives. Second, we look at existing tools to gather dynamic information and describe their features. Third, we present several tools or IDE enhancements visualizing static or dynamic information to improve software comprehension, maintenance, or navigation and study their contributions to improve the understanding of software systems. Not all of them are based on dynamic analysis, some employ different techniques, such as relating source artifacts to each other based on navigation patterns. Ultimately, we briefly outline different controlled experiments in software engineering.

2.1 Traditional IDEs

Static source code analysis is widely integrated in mainstream IDEs such as Eclipse or NetBeans. Besides enabling developers to edit the source code of a software system, IDEs often provide developers with other means to study and maintain a software system such as syntax highlighting or debuggers. Modern IDEs integrate a broad range of different perspectives and supportive tools. For instance, they integrate means to navigate the hierarchical structure of the source code. Such an example is the Package Explorer in Eclipse in which source code artifacts are hierarchical organized (Package → Class → Method). Another example is the class browser available in Smalltalk environments. This browser contains a series of panes horizontally aligned side by side at the top of a text editor window; these panes present the class hierarchy of the Smalltalk system. Such a browser does not integrate any runtime behavior, it is therefore for example not possible to determine which parts of the software system are used while executing a certain feature.

Eclipse integrates several search facilities, for example developers can search for implementors of a certain interface or can look up references of a method within the software system. As these search facilities are purely based on static source code analysis they return for example every class implementing a certain interface. Developers can therefore barely determine which of the found classes are used at runtime and are hence relevant to study the software system. For the same reason, all possible references to a method are disclosed, which makes any studying of runtime collaboration uncertain and imprecise. A more sophisticated tool is the Call Hierarchy View which discloses callers and callees of a certain method. It is possible to restrict the search scope, to reveal collaborations restricted to certain packages. However, as this lookup is still based on static source code analysis it can never represent the actual collaboration occurring at runtime or restrict callers and callees lookup to certain feature executions.

Eclipse also integrates various visualizations and enrichments to ease navigation within the software system and to enrich its views. Such an example are tooltips, which embody the documentation of a method in small pop-ups that appear when hovering with the mouse over a reference to that method. Eclipse also uses a broad range of icons and annotations to display compiler errors or warnings, or search results within the source code view. Compiler warnings are for example annotated by yellow icons containing an exclamation mark within the various navigation trees or by yellow annotations in the overview ruler on the right side of the source code view. *Senseo* reuses some of these techniques to enrich Eclipse's views with dynamic information.

With static analysis, especially with static type inference [46], it is possible to gain insights into the types that variables assume at runtime. Deriving static types by type inference and type reconstruction has a wide range of usages. However, most approaches do not scale very well and trade precision by performance [62]. Roel Wuyts et al. introduce with RoelTyper [67] a fast type reconstructor for Smalltalk. It type-checks instance variables of classes based on heuristics. Pluquet et al. [47] argue that such an imprecise but faster approach is required to include feedback about the possibly used types into development environments. They present the type reconstruction algorithm used by the RoelTyper and tested the heuristic in different Smalltalk environments. Pluquet et al. found out they are able to find nearly instantly in 75% of the cases the correct type. Such a fast method allows direct usage within IDEs. However, this approach implements type inference based on static source code analysis and provides therefore only an approximation. This means that it still guesses which types might occur at runtime and does not reveal the actual types used at runtime.

Knowledge about the underlying structure of the code can be exploited during refactoring tasks or other code transformations. Additionally, to ease code completion, knowledge about the structure of the software system is used to provide developers with a list of candidate messages that could be sent to a variable of certain type. Robbes and Lanza [52], for example, show that additional ordering of candidate messages based on the change history can improve the expected results for code completion.

2.2 Dynamic Analysis

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 [23, 66]. 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.

The concept of data and execution flow analysis has been widely studied in the field of static and dynamic analysis. For instance, Eisenbarth et al. [24] proposed the use of static and dynamic feature analysis to better support program comprehension. In later work they also motivate the use of concept analysis and dynamic information of execution traces to reason about features within a software system [25]. Their approach helps developers to easily identify the different components of a certain feature within a software system to quicker understand the internal structure and relations. That is for example, identifying layers of a software system such as the persistent layer or identifying how these layers communication with each other.

Issues of dynamic analysis are portability and performance. Furthermore, dynamic analysis generates a huge amount of data which has to be processed, aggregated and stored for further use. Developers usually resort to debuggers to dynamically analyze software systems and to, for example, determine runtime types or to predict the behavior of certain source artifacts. Unfortunately, information extracted during a debugging session is volatile, that is, it disappears at the end of the session. Furthermore, debuggers are often separated from the usual workflow within IDEs and provide only a relation between the current (halted) context of an execution and the source code. Contrary to *Senseo* debuggers are therefore cumbersome to gain a bigger picture on the underlying software system nor they are capable of aggregating dynamic information over several runs.

Tools dynamically analyzing software systems rely mainly on techniques enabling system profiling capabilities. Using these techniques the tools can reveal execution trace or execution time of a certain method. Profiling capabilities have been integrated in IDEs, such as the NetBeans Profiler[1] and Eclipse's Tracing and Profiling Project (TPTP)[2].
In general, profilers give information about a program's execution performance, but similar to debuggers, profilers also focus on specific runs of a system and hence give limited general insights into overall complexity of source artifacts. For instance profilers do not aggregate the gathered information over several executions and can reveal therefore no information about how a certain source artifact complexity is affected by the execution of different software features.
Profilers suffer therefore from similar drawbacks as debuggers: the collected dynamic information is not integrated in the static source views in the IDE. Hence, developers use such tools only occasionally instead of continuously benefiting from dynamic information that is directly available in the source code views.

Conceptually, there is not much difference between *Senseo* and a debugger or profiler. However, the main difference is, that *Senseo* presents and continuously updates the dynamic information **directly within** the source code views, without extra actions to be taken by the developer. Additionally, *Senseo* records and aggregates dynamic information over several runs and provides an easy way to compare such gathered data by easily selecting which gathered information is used to generate the enrichments and visualizations.

In the following paragraphs we outline different known techniques to gather dynamic information about Java based software systems:

[1] http://profiler.netbeans.org/
[2] http://www.eclipse.org/tptp/performance/

JFluid. The NetBeans profiler uses the JFluid technology [16] to which we refer simply as JFluid. It exploits dynamic bytecode instrumentation and code hot swapping to collect dynamic metrics [17]. Contrary to *MAJOR*, which is used by *Senseo*, 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 [15]. JFluid relies therefore on a customized JVM which ties JFluid to that special JVM and makes portability harder. NetBeans accesses the gathered CCT through a socket or shared memory. The gathered data is only used to provide profiling information which makes JFluid a pure profiling tool.

***J.** Dufour et al. [20] present a variety of dynamic metrics for Java programs. They introduce a tool called *J [22] for metrics measurement. *J relies on the Java Virtual Machine Profiler Interface (JVMPI) [63]. JVMPI requires that profiler agents are written in native code which hampers portability, whereas *MAJOR* implements everything directly in Java using standard aspect weaving techniques. Moreover, JVMPI is known to impose high performance overhead, and for this reason instrumentation of the studied software system is slow and impractical.

PROSE provides aspect support within the JVM [48] and aims to ease the collection of certain dynamic metrics based on direct access to JVM internals. PROSE combines bytecode instrumentation and aspect support at the just-in-time compiler level. However, contrary to *MAJOR* it does not support aspect weaving in the standard Java class library. This makes it impractical to gain insights in used JDK libraries. In addition, this limitation makes it difficult to reason about *border regions*, regions where execution flows enter application code or where application code calls JDK code.

As a technique to reduce the amount of gathered information, sampling-based profiling techniques can be used. They are common for feedback-directed optimizations in dynamic compilers [3]. Such techniques significantly reduce the overhead of metrics collection. However, sampling produces incomplete and possibly inaccurate information, whereas the integration of dynamic information into an IDE requires complete and precise metrics for all executed methods.

2.3 IDE Enhancements and Visualizations

A common feature in modern IDEs is to provide different views on the source artifacts of a software system. Such a view is for example a visualization of the class hierarchies in the studied software system presenting various information about the existing classes and interfaces such as which classes implement a certain interface. Further research tries to take more sources into account and provide polymetric views of a system. For instance, Girba et al. [27] proposed to use the history of an observed hierarchy as an additional source. They use polymetric views to disclose the evolution of class hierarchies.

In the following paragraphs we discuss different tools and concepts integrating and visualizing static and/or dynamic information.

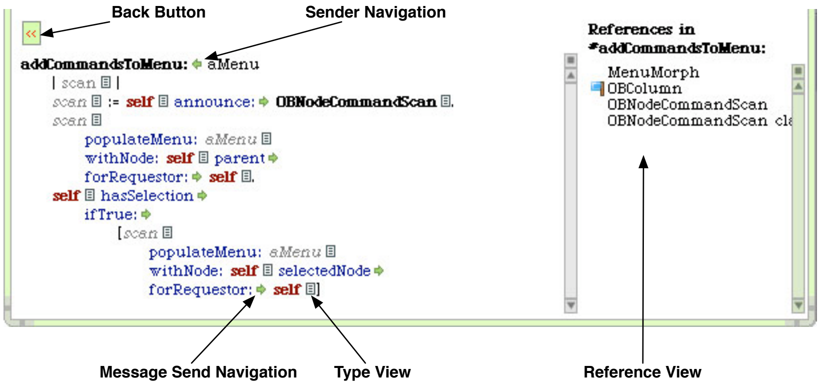
Class Blueprints. Lanza and Ducasse [34] propose class blueprints, a visualization of the internal structure of classes. These blueprints provide several layers representing different aspects of an object: For instance, one layer represents the initialization of the object and another one represents the accessor layer, where setter and getters can be found. These layers form a template on which concrete classes can be mapped. Within these layers, various methods or attributes of a class are represented by colored boxes of different shape. Blueprints ease the understanding of a class and help developers to quickly grasp the inner structure of a class. Class Blueprints are only based on static source code analysis, they therefore do not take into account any runtime information. Furthermore, contrary to *Senseo* they are implemented in an external view and not integrated into the IDE. They are not directly usable while navigating the source code space and require developers to perform a context switch (switching to another window) to use these blueprints. Such a required context switch makes class blueprints not as valuable as information directly embedded in the IDE.
Ducasse et al. [19] adopted a similar approach to represent the different roles of packages in package surface blueprints. They group packages based on references to other packages. This reveals how packages are connected to each other.

JIVE: Visualize Java in Action. Reiss [51] developed a tool to visualize the behavior of Java programs in real-time. This work introduces a dynamic Java visualizer that provides a view of a program in action with low overhead. Hence, it can be used at any time by programmers to understand what their program is doing while it is doing it. The process to gather dynamic information is ongoing, but can be turned on and off during the execution. It visualizes the gathered data in a box display where each box represents a certain class or thread. Each box contains differently colored rectangles representing the various statistics gathered. Such boxes are visualized in an external tool which limits its usage, as it does not integrate dynamic information into the IDE and the source code views. While *Senseo* supports developers to navigate the source code space, *JIVE*, for example, does not link the gathered dynamic information with the static source code.

VizzAnalyzer. Löwe et al. [40] merge information from both static and dynamic analysis to generate visualizations in a dedicated tool. For static source code analysis and even transformation of Java sources, they developed a software called *Recorder*[3]. This tool provides the basis for further dynamic analysis which is performed by using a tool called *VizzAnalyzer* [60] which maps information from a debugger interface to graphical views. These graphical views are presented in an external tool and are updated live from the debugger interface. It is possible to apply filters and aggregation on the gathered information. On the dynamic side of their analysis they can reason about *Calls*, *Accesses* (on attributes), *Knows* (representing relations, such as binding another object in a variable) and *Instance-Of*. Due to the visualization in an external tool, *VizzAnalyzer* is limited in supporting developers directly within the IDE. Contrary to *Senseo* developers therefore have to change the current view on the software system and interrupt their workflow to benefit from *VizzAnalyzer*'s visualizations.

[3]`http://recorder.sf.net`

Hermion. Röthlisberger et al. [56] with Hermion integrate dynamic information into the IDE and provide mechanisms to query such information. Hermion integrates navigation features into the source code to enable navigation to callers and callees. Furthermore, Hermion indicates runtime types of variables and provides a reference view to disclose collaboration with other classes. Compared to *Senseo*, Hermion's feature-centric per-



```
addCommandsToMenu: aMenu
    | scan |
    scan := self announce: OBNodeCommandScan.
    scan
        populateMenu: aMenu
        withNode: self parent
        forRequestor: self.
    self hasSelection
        ifTrue:
            [scan
                populateMenu: aMenu
                withNode: self selectedNode
                forRequestor: self]
```

```
References in
#addCommandsToMenu:
  MenuMorph
  OBColumn
  OBNodeCommandScan
  OBNodeCommandScan cl
```

Figure 2.1: Enriched source code view in Hermion

spective is more targeted to reveal collaboration within features or disclose methods used in different features. It is for example limited in disclosing hotspots within a software system, as it does not reveal information about the number of objects allocated within a certain method.

Feature Views. Greevy et al. [28] propose to analyze software evolution through feature views. Features are usually assembled by different source artifacts of a software system. These artifacts are a valuable source of information for a reverse engineer. By capturing the execution trace it is possible to map a feature execution with its source artifacts. Such execution traces are often huge and difficult to interpret. Greevy et al. visualize such traces in *feature views*, by compacting them into simple sets of source artifacts that participate in a features runtime behavior. The visualizations are generated using Mondrian [41] which is integrated in the *Moose*[4] reengineering environment [43]. Furthermore, they are able to compare such feature traces and their views over time, to evaluate how a feature changes over time.
However, their approach mainly compacts and visualizes feature traces; any integration into an IDE is missing. Moreover, the view on a feature is limited to its execution trace and does not reveal any further dynamic information such as runtime types of variables. Feature Views do also not reveal other interesting aspects of a software system such as exact number of allocated objects within a method or within a complete feature trace.
Röthlisberger [57] integrated these visualization into the Squeak IDE in a so called *feature browser*. The various contributions of different parts of the software system to a certain feature are linked with the corresponding source artifacts. Such an integration enables developers to navigate, browse and modify these feature artifacts within a single environment. Additionally, this environment contains a *feature tree* which visualizes the recorded method call tree of a feature. An empirical study with twelve graduate computer

[4]http://www.moosetechnology.org/

science students showed that such a feature centric environment has a positive effect on program comprehension and in particular on the efficiency in discovering the exact locations of software defects and in correcting them efficiently.

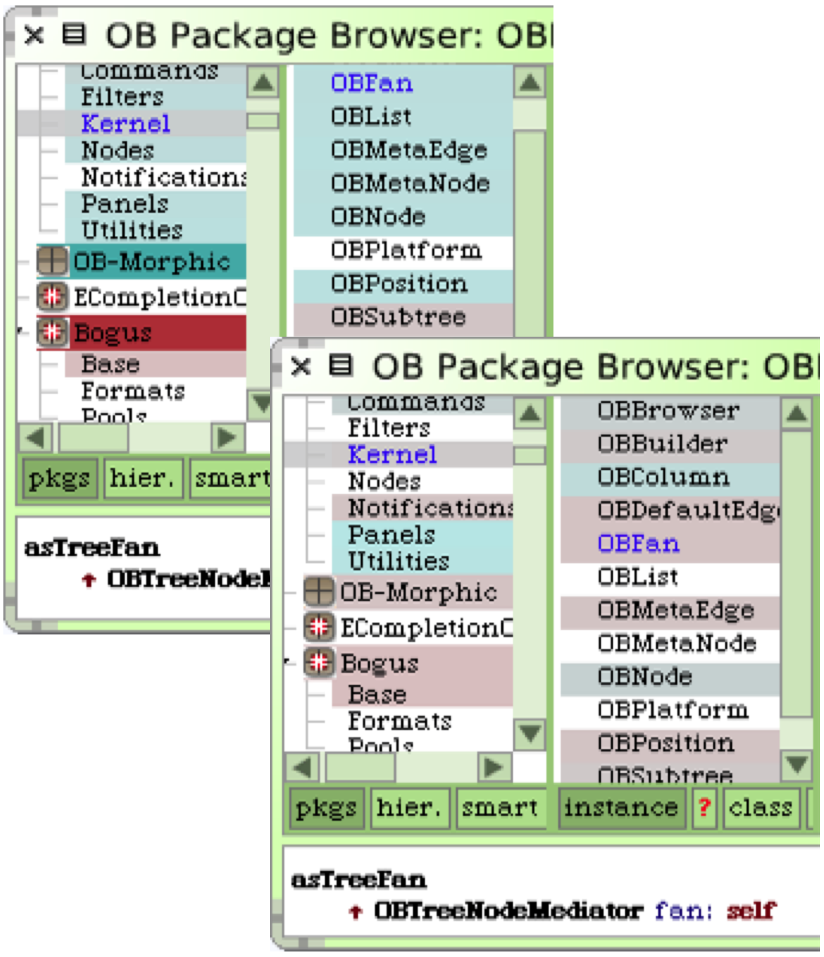
Fluid source code views [14] are another approach to disclose related source artifacts within the source code space. They present related code (for instance an invoked method) directly in the current source code view in an additional widget. Such a view recognizes the separated but linked nature of source 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.

Ferret [11] recognizes the conceptual relation between static and dynamic aspects of software systems 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. Contrary to *Senseo*, Ferret does not aim at giving an overview of the system or enriching the static IDE perspectives with dynamic information.

Mylyn (formerly Mylar) [30] computes a degree-of-interest value for each source artifact based on the historical selection or modification of the artifact. The background color of the artifacts highlights their relative degree-of-interest in the context of the current task; interesting entities are assigned a "hot" color.
In Mylyn the information used to compute the interest value is relatively simple: selecting and editing an artifact increases the interest; if no further event occurs the interest decreases over time. Mylyn therefore only uses the recorded analysis of a developer's IDE usage to disclose candidate source artifacts and does not take any static or dynamic source code analysis into account.

HeatMaps. Röthlisberger et al. [59] propose HeatMaps as a simple and uniform mechanism to represent complex information in an easily understandable way in any IDE. Heat is computed by two different means, namely in time-based HeatMaps or metrics-based manner. Figure 2.2 shows such HeatMaps integrated in the Smalltalk Squeak IDE. The top left one highlights the number of versions of source artifacts and the bottom right one the recently browsed artifacts.

NavTracks exploit the navigation history of software developers to form associations between related source files (e.g., class files) [61]. Based on these relations NavTracks reveals related entities to developers. This approach works at the granularity of files, hence does not take into account specific methods or classes. As it is therefore based on a single data source, namely the recency of browsing in the navigation history, such a recommendation list helps little to obtain an overview of the whole system; the developer just sees a list of artifacts possibly related to a specific artifact, but does not see all interesting entities in a "big picture" view. These recommendations are always relative to



```
asTreeFan
    ↑ OBTreeNodel
```

```
asTreeFan
    ↑ OBTreeNodeMediator fan: self
```

Figure 2.2: Two HeatMaps in the Smalltalk Squeak IDE

a selected artifact, that is, dependent on what the developer has currently selected, thus it is not easy to identify all artifacts related for a given task.

FEAT [54] applies a concern graph to visualize scattered but conceptually related code elements together in order to identify and navigate elements relevant for a particular concern. Recent versions of FEAT are able to automatically infer the source entities related to particular concerns [53]. Robillard et al. define a concern as “anything a stakeholder may want to consider as a conceptual unit, including features, nonfunctional requirements, and design idioms” [55]. Usually the source code implementing a concern is not encapsulated in a single source entity but is instead scattered and tangled throughout a system [55]. To determine the entities participating in a concern, FEAT analyzes system investigation activities performed by the developer in the IDE [53]. The resulting concern graph presented in the FEAT Eclipse plugin supports developers performing maintenance tasks involving identified concerns [53].
As the authors report, FEAT’s concern identification algorithm is heavily dependent on how organized the analyzed investigation activities are [53, 55]. Disorganized investigation sessions yield vague, incomplete and often useless concern graphs [53]. Thus the

FEAT approach is not very robust and not properly usable when only having available development sessions from developers unfamiliar with the system under study. Furthermore, the FEAT approach requires developers to manually validate the proposed concerns by rejecting false positives, that is, concerns wrongly identified.

CodeCrawler. Lanza et al. [35] contribute CodeCrawler, a stand-alone tool to analyze statics and dynamics of programs. It is mainly targeted at visualizing object-oriented software based on polymetric views. The principle of such views is to represent source code entities as nodes and their relationships as edges between the nodes. It has been integrated into Moose[5] [18], a platform for software and data analysis, and there is also an Eclipse Integration called *X-Ray*[6].
CodeCity is a visualization tool based on the analyses performed by Moose. It provides static source analysis on top of Moose. The visualization is a navigable 3D city where classes are represented as buildings within the city. Packages are visualized as districts of the city in which the buildings remain. CodeCity maps static metrics, such as number of methods or attributes in a class to a certain building. Moreover, it visualizes the nesting level of classes within packages by placing them in darker regions of a district.

CodeMap. Erni [26] provides with CodeMap a visualization of the source code using the same visual language as atlas cartography [33]. CodeMap generates maps from the vocabulary of a software system. Features belonging to certain terms are grouped together on islands with the idea of grouping code belonging to the same domain. CodeMap aids the developer with a mental model of the software project. A current implementation [32] integrates the approach within an Eclipse Plugin[7] and combines it with various other static and dynamic source code analysis such as test coverage or code ownership. It integrates supportive tools for navigation such as displaying search results or visualizing control or data flows on the map.

All these different tools and visualizations contribute differently to the various fields of software maintenance, reverse engineering, or program comprehension, and address similar topics as *Senseo*. In Table 2.1 we summarize the main differences of these tools to *Senseo* and highlight *Senseo*'s advantages compared to them.

2.4 Controlled Experiments in Software Engineering

Other researchers also conducted controlled experiments to validate tools supporting software maintenance tasks. Cornelissen et al. [10] 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. [49] evaluated with 25 students the benefits of Dynamic Object Process

[5]http://www.moosetechnology.org/
[6]http://xray.inf.usi.ch/
[7]http://scg.unibe.ch/research/softwarecartography

Tool	**Main differences compared to *Senseo***
Class Blueprints [34]	Presents blueprints in an external tool which are purely. based on static source code analysis.
JIVE [51]	Visualizes dynamic information in an external tool and does not integrate the information into the IDE.
VizzAnalyzer [40]	Visualizes dynamic information in an external tool and is therefore not integrated into the IDE.
Hermion [56]	Only enriches source code views with dynamic information, no visualization revealing the bigger picture on a software system is integrated.
Feature Views [28]	Limited to compact and visualize feature traces. No direct integration into the IDE or no information about runtime types is disclosed.
Fluid source [14] code views	Only linking static source code views, may give wrong results in case of polymorphism.
Ferret [11]	Provides only a possibility to query the source code space taking dynamic information into account.
Mylyn [30]	Discloses source artifact based on the historical selection or modification of the artifact. No dynamic information is used.
HeatMaps [59]	Visually discloses source artifacts based development information such as historical navigation.
NavTracks [61]	Discloses related code based on the navigation history of the developer and does not take into account any runtime information.
FEAT [54]	Organizes scattered code based on concerns. Organization is completely based on the way developers investigated the source code.
CodeCrawler [35]	Mainly targeted to visualize the structure of object-oriented software systems based on polymetric views.
CodeMap [26]	Provides a mental model based on static source code analysis.

Table 2.1: Comparing tools to *Senseo*

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.

2.5 Summary

Static source code analysis is the main technique used by many different integrations of visualizations into traditional IDEs. The perspective on a software system remains therefore mainly a static one.
We described several existing approaches to gather dynamic information. We showed that most of them are used for to gather profiling information about a software system. Moreover, we noted that most of them either provide incomplete information, are not easy portable, or suffer from severe performance issues.
We then presented various tools providing static or dynamic information about a software system. Furthermore, we looked at various IDE enhancements or visualizations. We showed how such tools or enrichments support developers in reasoning about runtime behavior. Approaches outlined in other works also visualize software dynamics, but are usually not directly integrated into an IDE but rather provided by separate tool. We compared these tools to *Senseo* in Table 2.1.
At the end we outlined other controlled experiments conducted in research to validate tools supporting software maintenance tasks.

Chapter 3

Motivation

The narrow focus on static source code analysis in IDEs results in particular in a static view on a software system. However, features of object-oriented languages such as abstract-types or late-binding impede any software comprehension purely based on static source code analysis. Such purely static views on source code do not tell us how the program behaves at runtime nor does it reveal any information about runtime types. Furthermore, static analysis such as static type inference, is a computationally expensive task and provides often imprecise results [50]. The analysis is imprecise in the sense that too much information can be taken into account – not all the possible cases actually appear at runtime – or that we cannot reason about exact number of invocation. The following examples illustrate common problems of static source code analysis in the context of object-oriented languages.

3.1 Reasoning About Complex and Extensible Frameworks

Huge and extensible software systems such as Eclipse are often based on a well documented architecture. The different parts of the framework forming such an architecture like the one of Eclipse are generally exposed through interface types to developers implementing other parts or plugins. Eclipse comes therefore with a huge hierarchy of interfaces to model the structure of different aspects of the Eclipse framework. Such an interface is for example `IResource` which is heavily used to represent the state of a resource maintained by a workspace in Eclipse such as a file containing source code. `IResource` is a very abstract representation for any possible structure to give a unique way to handle different structures such as a view representing a certain visualization of a software system in the same manner within Eclipse. This makes it a widely used interface type in different parts of Eclipse like for example within the core of Eclipse[1]. Digging deeper into Eclipse plugins such as JDT suddenly reveals the main problems when reasoning about abstraction in interfaces: JDT encompasses interfaces and classes modeling Java source code artifacts such as classes, methods, fields, or local variables.

[1] `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/class-use/IResource.html`

IResource is used a lot within JDT for the already known use cases, but it is also used to represent these JDT specific types. For instance, JDT models Java element containers, such as folders, files or compilation units with its own interface hierarchy (*IFolder*, *IFile*, ...) which all implement IResource. However, these resources are passed to the core framework as IResource-types and are also passed back with these types. This means that within the JDT these resources have to be casted back into the representing Java elements, which is illustrated by the following code sample taken from the JavaModelManager:

```
public static IJavaElement create(IResource resource, IJavaProject project) {
  /*...*/
  int type = resource.getType();
  switch (type) {
    case IResource.PROJECT :
      return JavaCore.create((IProject) resource);
    case IResource.FILE :
          return create((IFile) resource, project);
    /* ... */
        default :
          return null;
  }
}
```

While we statically examine JDT and reason about its structure, we encounter IResource types in many different parts such as in variables, in message sends, or as return values. However, we are not able to tell whether a certain return value represents a Java element or any other resource represented by IResource within the Eclipse framework.
We can conclude that the introduction of the JDT plugin establishes new use cases for IResource. But these JDT specific types are not mentioned in the Eclipse core documentation which is the main reference for IResource. Furthermore, by purely looking at the source code we are not able to determine which implementations of IResource are used within JDT and in which parts.
While looking at JDT we come across an example similar to IResource namely IJavaElement. Contrary to IResource, IJavaElement is only used within the JDT plugin. However, its behavior is similar to that of IResource. Figure 3.1 shows an extract of the JDT interfaces and classes representing static artifacts of a class. Clients of this representation usually refer to interface types such as IJavaElement or IJavaProject, as the following code snippet found in JavadocHover illustrates:

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

This example reveals that for some elements the resulting javaProject is wrong or undefined. We have the impression that the if conditions are not comprehensive or not correct at all. Or the developer who implemented this source artifact was aware that only certain clients of IJavaElement are contained in the elements collected during the execution of this method. However, this assumption is contrary to the documented method signatures we find in the generated documentation which lists simply IJavaElement as a client of this method. The developer could have excluded the remaining clients of

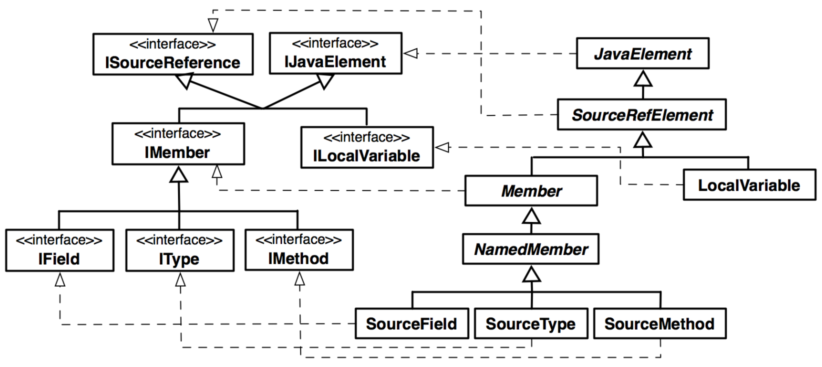


Figure 3.1: JDT interface and class hierarchies representing Java source elements (extract).

`IJavaElement` by intent, as he might have been aware of implementation or performance problems. Another possibility is that the method `getJavaProject` is wrongly implemented for some element types. Thus we have two questions about this code:

1. Which implementations of `getJavaProject` are invoked?
2. Which types are stored in variable `element`; are all relevant cases covered with the `if` statements?

3.2 Understanding Abstract Class and Interface Hierarchies

For the first question, we search for all declarations of method `getJavaProject`. Using static source code analysis, Eclipse reveals all implementors of `IJavaElement`. However, Eclipse lists also many more implementations of such a message which might be totally unrelated to `IJavaElement`. Additionally, JDT itself declares more than 20 methods with this name, most of which are not related to the representation of source code elements. We have to skim through this list to find out which declarations are defined in subtypes of `IJavaElement`. After having found these declarations, we still cannot be sure which are actually invoked in this code as we rely only on static source code analysis.

3.3 Runtime Types

Even when manually narrowing down the list of declarations of *getJavaProject* to those actually defined in subtypes of `IJavaElement`, we still have a lot of remaining candidates. Moreover, we can neither be sure whether those are really invoked by this source code nor how often and with which concrete receiver types.

To address the second question of Section 3.1, we first search for all classes implementing `IJavaElement` in the list of references to this interface. This yields a list with more than 2000 elements; all are false positives as `IJavaElement` is not supposed to be implemented by clients. We thus search for all sub-interfaces of `IJavaElement` to see whether those have implementing classes. After locating two direct sub-interfaces (`IMember` and `ILocalVariable`), each of which has more than 1000 references in JDT, we give up searching for references to indirect sub-interfaces, such as `IField` or `IType`. It is not possible to statically find all concrete implementing classes of `IJavaElement`, in particular not those actually used in this code.

Thus we resort to using the 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 message send, we would have to debug many more scenarios, which is very time-consuming as this code is executed many times for each system run.

3.4 Invocation Count

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. However, such information is crucial to assess the performance impact of particular code statements. Speaking about such code statements: While running the debugger we further realize that `getJavaProject` is very often executed, as our breakpoint within this method frequently halts the execution. However, as debuggers usually do not provide a way to efficiently count the exact number of invocations we are required to look for another tool . Such a tool is for example a profiler with which we can reason about the invocation count. This means that we have to rely on two different tools built for different purposes that both provide just volatile information.

3.5 Assessing Runtime Complexity

Using the profiler shows that executing `getJavaProject` for some types of source elements is remarkably slow. However, developers addressing this efficiency problem need to know for which types the implementation of `getJavaProject` is slow and why, for instance, whether the inefficient implementations create many objects or repeatedly execute code. However, by using the profiler we are not able to reveal which types are responsible for the performance bottlenecks. This leaves us in the situation that we end up pinpointing the receiver using a debugger and a profiler together to validate our assumptions. Furthermore, we have to aggregate information about runtime complexity over multiple runs to pinpoint specific method executions being slow. Revealing all methods invoked by polymorphic message sends using a debugger is often time-consuming, in particular if a code statement is repeatedly executed with different input parameters as in the example above. This makes any procedure to examine such a situation rather tedious and hardly efficient.

We therefore need a way to reveal such hotspots within their calling context spanning multiple executions. The execution complexity of a method is often heavily dependent on the passed parameters or on the receiver to which the message triggering the method invocation is sent. A visualization combining source code with dynamic information would enable us to quickly determine which types suffer from poor performance.

It is much more convenient for a developer if the IDE itself could collect and show runtime information aggregated over several runs together with the static structure, *i.e.*, augmenting Eclipse's source code viewer to show precisely which methods are invoked at runtime and how often, optionally even displaying runtime types for receiver, arguments, and return values.

3.6 Understanding Execution Flow

To gain knowledge about the architecture of a software system or a framework it is necessary to understand the execution flow of such a system and how the different parts collaborate. The importance of execution path information becomes clear when inspecting Java applications employing abstract classes or interfaces. As we have mentioned earlier such systems often follow a black box approach. Hence, the source code usually refers to these abstract types, while at runtime concrete types are used. However, locating the concrete class whose methods are executed at runtime when the source code just refers to interface types, can be extremely cumbersome with static navigation, since there may be a large number of concrete implementations of a declared type.

Therefore it is crucial for a developer to have information about the execution flow and collaborating parts at hand. Additionally, a developer is interested in how these parts communicate with each other. This asks once more for a visualization revealing execution flow and the collaborating parts of a source artifact to developers. Additionally, as executions might involve interface or abstract types it is important to reveal the actual runtime types as well as their calling context.

3.7 Persistent Integration into IDEs

As outlined in Section 3.4 information gathered by profilers and especially by debuggers is volatile, bound to specific executions and therefore disappears at the end of the session. Furthermore, it is not possible to persistently store such information for much longer than immediate usage. Besides that both tools do not feed aggregated information back to the IDE, thus developers use them only occasionally instead of benefiting from runtime information directly in the static IDE views. It is therefore crucial to integrate the gathered information in a non-volatile, persistent manner, so that it is possible to aggregate and prepare it for further usage.

3.8 Summary

In this chapter we have outlined different examples for the typical software maintenance tasks we outlined in Section 1.1. We showed that plain static views of traditional IDEs

limit developers in each of these tasks. Additionally, we can conclude that existing dynamic analysis is often limited to tracing mechanisms and focusing on capturing a call tree of message sends. Furthermore, we showed that dynamic behavior is not linked with the static structure of a program. However, it is important to have runtime information available to developers. Moreover, it should be possible to persist such information for future analysis or comparison.
We propose to integrate the gathered information directly within the IDE to support developers in efficiently performing typical software maintenance tasks. To support many different software maintenance tasks we aim to integrate the gathered information at different levels: At a fine-grained method level to illustrate complexity, for instance, based on receivers of a message send. But also at a more coarse-grained level, for instance, for entire packages or classes, to quickly identify candidate locations for highly invoked artifacts.

Additionally, we need to give an overview of the different parts of the entire software system collaborating with each other. Such an overview provides a big picture view of the examined software system and highlights points of interest.

To summarize our proposal we conclude that we need to:

- gather dynamic information.
- aggregate and store such dynamic information.
- integrate the aggregated information on a fine- and coarse-grained level into the IDE.
- provide higher level overviews.
- provide means to navigate the source code space through these enrichments and visualizations.

By meeting these requirements developers are provided in the IDE with yet missing but required information about the behavior of a software system at runtime. Moreover, we claim that direct access within an IDE to dynamic information such as message sends and dynamic types eases software maintenance by making available dynamic information and by enabling efficient browsing and navigation.

Chapter 4

Gathering Dynamic Information

Gathering dynamic information is the basis to reason about runtime behavior and therefore required for any of our further work. In the following sections we will introduce the topic of dynamic analysis and present our solution to efficiently gather dynamic information.

4.1 Dynamic Analysis

Dynamic software analysis is commonly referred to as the analysis of the properties of a program at runtime [4]. It yields precise information about the runtime behavior of a system. In contrast to static analysis, dynamic analysis derives properties that hold for one or more executions [4]. Gathering dynamic information usually requires the examination of the running program through program instrumentation [37] using different kind of techniques. Denker et al. [12] identify the following techniques to gather runtime information: *Source code modification*, *logging services*, *bytecode modification*, *method wrappers*, *debuggers*, or *instrumentation*. Studying these techniques we encounter different kinds of limitations or drawbacks. For instance, such a technique might require adjustments to the execution environment and is therefore not fully suitable to seamlessly integrate information gathering into the IDE. We analyze these techniques regarding the gathering of dynamic information of Java based software systems:

- *Source code modification* or *logging services* are not appropriate, as we aim to integrate the process of data gathering seamlessly into the IDE and do not want to introduce source code changes that might affect the runtime behavior.
- *Bytecode modification* requires profound knowledge of the bytecode instructions used by the virtual machines. This makes the instrumentation not easily portable to different implementations of the Java Virtual Machine.
- *Method wrappers* are easy to integrate in dynamic languages like Python or Smalltalk. However, they are not really feasible in Java due to its limited capabilities of intercession.

- *Debuggers* can be used to gather dynamic information, however, as they are usually not meant for instrumentation, they do not provide an efficient interface to continuously gather runtime information.

Of the presented techniques by Denker et al., *instrumenting* the virtual machine, the runtime environment in which the system runs, is best suited for our goal. This technique does not require any modification of the source code, which makes it suitable to seamlessly integrate the process of information gathering into the IDE and the instrumentation is also a flexible way to gather runtime information. In Section 2.2 we presented various frameworks, that provide different ways to instrument the Java Virtual Machine (JVM). We summarize their drawbacks as follows: *PROSE* does not cover the standard JVM libraries them self, thus it will not provide a full coverage of a software system. *JFluid* uses a hard-coded, low-level instrumentation to collect gross time for a single code region, which doest not satisfy our requirement of a very flexible way to gather dynamic information. Another presented tool is **J* which relies on the Java Virtual Machine Profiler Interface (JVMPI) [63] which is known to cause high performance overhead and requires profiler agents to be written in native code.
As a suitable solution for our requirements we selected *MAJOR* [64], an AspectJ-based [31] weaving tool enabling comprehensive aspect weaving into every class loaded in a Java Virtual Machine, including the standard Java class library, vendor specific classes, and dynamically generated classes. *MAJOR* is based on the standard AspectJ compiler and weaver; it uses advanced bytecode instrumentation techniques to ensure portability [5]. Moreover, using aspects eases customization and extension and makes it fully portable to different implementations of the JVM while introducing only a moderate overhead. Using aspects we can easily adapt the gathering of dynamic information to our needs and can describe further data to be gathered. The code *MAJOR* has woven (instrumented) is executed immediately after the JVM bootstrapping, which enables us to gain nearly full coverage of all executed code within the complete runtime. Only a small part of the initialization until *MAJOR* is loaded cannot be covered. However, we can ignore this issue as for nearly all applications this part is completely irrelevant and only prepares the environment to load *MAJOR*.

The following paragraph discusses how *MAJOR* internally organizes the gathered data to get an accurate and compact representation, and how the collected information is transfered to the IDE.

Aspect-based Calling Context Tree Construction The woven aspects use the collected data to continuously build a tree of calling context profiles (CCT) [1], containing different dynamic metrics such as amount of invocations of a method. The CCT represents a generic data structure that is able to hold various metrics for each executed calling context. It provides a compact and yet informative representation of every calling context. Figure 4.1 illustrates a code snippet together with the corresponding CCT (showing only method invocation counts as metric). Each CCT node stores dynamic metrics and refers to an identifier of the target method for which the metrics have been collected. It also has links to the parent and child nodes for navigation in the CCT. Our CCT representation is designed for extensibility so that additional metrics can be easily integrated.

MAJOR sends periodically a snapshot representing the current CCT to the interested

Code Sample

```
void f() {
  for(int i=1;i<=10;++i) {
    h();
    g(i);
  }
}
void g(int i) {
  for(int j=1;j<=i;++j) {
    h();
  }
}
void h() { return; }
```

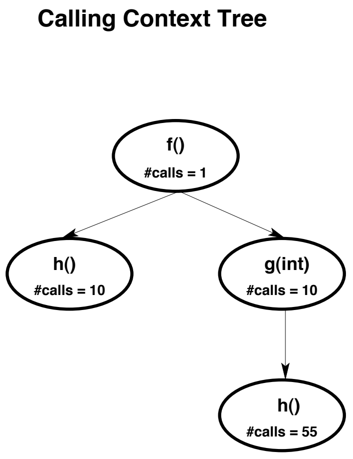


Figure 4.1: Sample code and its corresponding CCT

observer over a socket based communication channel.

4.2 Categories of Dynamic Information

In Chapter 3 we outlined different characteristics of object-oriented software systems which can only be revealed completely at runtime. Each of these characteristics can be detected or described with one or several kinds of dynamic information. These kinds of dynamic information can be stripped down into different categories, such as *execution flow* (which methods are called in which order), runtime types (*e.g.*, argument types) or further interesting metrics. Depending which specific software maintenance tasks we would like to support with our enrichments and visualizations we need to gather different kinds of dynamic metrics.
We can determine the required categories of dynamic information by looking at the different examples we presented in Chapter 3: Runtime complexity can for instance be expressed in terms of number of objects created, number of methods invoked, or amount of memory allocated in a source code artifact. Such a static artifact is for instance a method like *getJavaProject* which we discussed in Section 3.1. Another example is message sending: In object-oriented programs, understanding how objects send messages to each other at runtime is crucial. However, static analysis cannot always reveal the actually invoked method as the statically defined type might represent an interface or abstract type. Due to inheritance respectively late-binding the method might be implemented in the super- or subtypes of that statically defined type. Additionally, precise receiver types, which could possibly be a sub-type of the class defining the method, can also be reified. Based on these examples we define the following categories:

Execution Flow. To reason about collaboration within a software system, knowledge about which methods call which other methods is essential. Such information can reveal the execution flow of a software system and is a first step towards identifying collaborating parts within a software systems. By storing the gathered dynamic information in the tree-like structure CCT we also map the calling context of a method and can derive from that the execution flow.

Number of invocations This dynamic metric helps developers to quickly identify hotspots in the source code, *i.e.*, very frequently invoked methods or classes containing such methods. Furthermore, methods never invoked at runtime become visible, which is useful when removing dead code or extending the test coverage of the application's test suite. Related to this metric is the number of invocations of other methods triggered from that particular method.

Method Invocation Variants If we know which methods are invoked in which order, we also like to reason about the actual variants of such an invocation (*e.g.*, precise argument or receiver types). Each part of that configuration can result in different ways the execution continues. For instance, depending on the receiver type a different implementation of the examined method is invoked. We expect the following three pieces of information about a method invocation to be crucial for the integration of dynamic information into the IDE:

- *Receiver types.* Often sub-types of the type implementing the method receive the message send at runtime. Knowing receiver types and their frequency thus further increases program understanding. Such knowledge can, for example, reveal that a certain implementation of a method is not used in the execution of an examined feature.
- *Argument types.* Information about actual argument types and their frequency increases the understanding of a method, *i.e.*, how it is used at runtime. Precise information about argument types eases the understanding of the task a method is performing as the processing of the task is often dependent on the parameters passed. Moreover, a notion of the frequency of certain argument types gives more insights into how often certain paths of the method are executed and thus gives hints what parts of a method could benefit the most from optimization.
- *Return types.* As return values pass results from one method to another, knowing their types and their frequency helps developers to better understand communication between different methods. For instance, a client of a method often does not know what precise type it will be assigned or whether a `null` return value is also to be expected. Thus, showing precise return types including relative frequency, improves the understanding of a method from the client's point of view.

Number of created objects. By reading static source code, a developer usually cannot tell how many objects are created at runtime in a class, in a method or by a line of source code. It is unclear whether a source artifact creates one or one thousand objects — or none at all. This dynamic metric, however, is useful to assess the costs imposed by the execution of a source artifact, to locate inefficient code, or to discover potential problems, for instance inefficient algorithms creating enormous numbers of objects.

Allocated memory. Different objects vary in memory size. Having many but very tiny objects might not be an issue, whereas creating a few but very huge objects could be a sign of an efficiency problem. Hence, we also provide a dynamic metric revealing memory usage of various source artifacts, such as classes or methods. This metric can be combined with the number of created objects metric to reveal which types of objects consume most memory and are thus candidates for optimization.

Dynamic Bytecode Metric. The aggregation of this metric relates to the complexity of the corresponding computation. 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 [9, 21]. 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 [6].

Having these categories of dynamic information available provides us with sufficient information to support developers by means of integrating or visualizing the gathered information within the IDE.

4.3 Performance Evaluation

Gathering dynamic information suffers often from excessive overhead and a huge amount of collected data. Many approaches use naive and non-optimized techniques such as using a standard hash-table implementation to to store the collected data. Hence, already for medium-sized applications, serialization of such non-optimized data structures introduces long latencies to transmit the gathered data. Furthermore, the transferred data can grow up to several gigabytes of serialized dynamic information.

In order to validate that the used technique to gather dynamic information 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[1] [7]. For our measurements, we use *MAJOR*[2] version 0.6 with AspectJ[3] version 1.6.5 and the SunJDK 1.6.0_13 Hotspot Server Virtual Machine. We execute the benchmarks on a quad core machine running CentOS Enterprise Linux 5.3 (Intel Xeon, 2.4GHz, 16GB RAM).

Figure 4.2 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 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 is taken for computing the overhead. For each run of each benchmark, the CCT and

[1]http://dacapobench.org/
[2]http://www.inf.usi.ch/projects/ferrari/
[3]http://www.eclipse.org/aspectj/

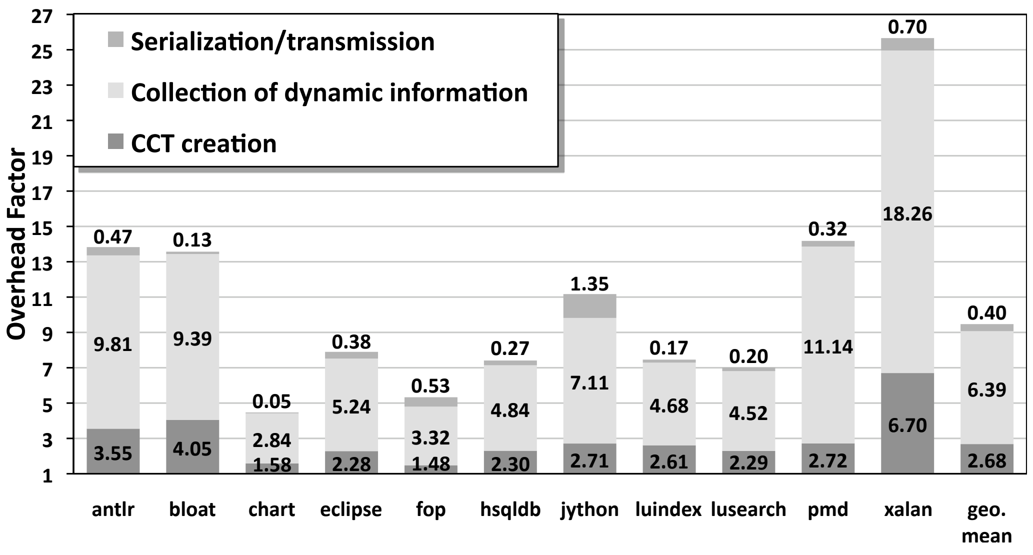


Figure 4.2: *MAJOR* overhead for the DaCapo benchmarks

the gathered dynamic information are serialized and transmitted once upon benchmark completion.

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.

We conclude that *MAJOR* 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.

4.4 Summary

We described the different concepts and techniques for dynamic analysis and introduced *MAJOR* as the most suitable solution. Additionally, we explained why the chosen technique is the best option for our requirements. We then outlined the different required categories of dynamic information to get a solid basis for enrichments and visualization to support developers in typical software maintenance tasks. As gathering dynamic information often suffers from poor performance, we benchmarked *MAJOR* and could show that its overhead of factor 9.47 still allows us to cope even with large-sized applications.

Chapter 5

Senseo - Integrating Dynamic Information in Eclipse

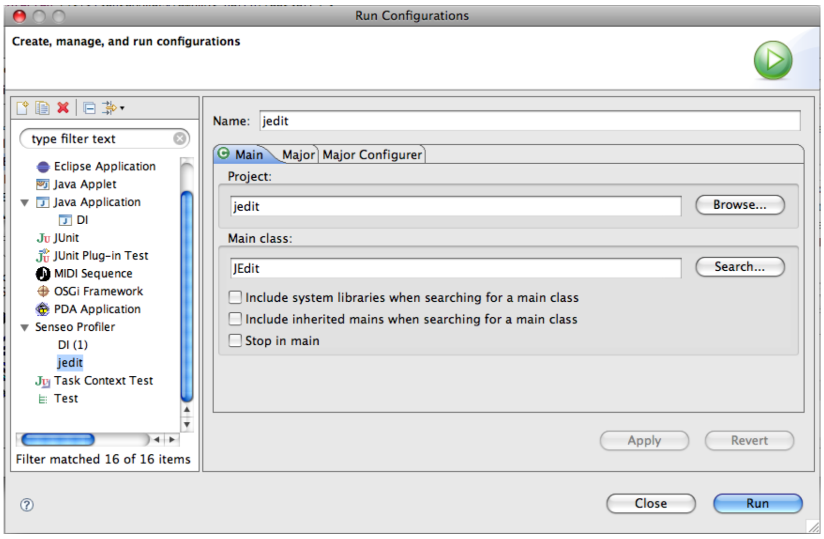
In Chapter 3 we outlined different problems we aim to solve and motivated the integration of dynamic information into the IDE. Moreover, we discussed the requirements to be able to provide software developers with dynamic information useful for typical software maintenance tasks.
To prototype this approach we implemented *Senseo*, an Eclipse plugin that enables developers to dynamically analyze Java applications. *Senseo* addresses the different points we stated in Section 3.8.

Senseo enriches the source views of Eclipse using the different categories of dynamic information as discussed in Section 4.2. *Senseo* is an approach to augment IDEs with dynamic metrics towards the goal of supporting the understanding of runtime behavior of applications, such as execution flow or execution complexity of source artifacts.
First, we present the basic architecture of information gathering in *Senseo*. Second, we illustrate several practical integrations and visualizations of the gathered dynamic metrics embedded in Eclipse.

5.1 Information Gathering

Senseo takes care of collecting the dynamic information and storing it in an aggregated format to ease the computation of dynamic metrics. The actual gathering of dynamic information is provided by a calling context profiling aspect run by *MAJOR*. Hence the application to be analyzed is executed in a separate application VM where *MAJOR* weaves the data gathering aspect into every loaded class, while the Eclipse IDE runs in a standard VM to avoid perturbations. While the subject system is still running, *MAJOR* periodically transfers the gathered dynamic data from the application VM to Eclipse using a socket. We do not have to halt the application to obtain its dynamic data. For efficiency reasons, we collect data during several seconds in the instrumented application VM before we send it in one piece over the socket.
To analyze the application dynamically within the IDE, developers have to execute it with

Senseo. This is done by an additional *Run Configuration* called *Senseo Profiler* which *Senseo* adds to the already existing Eclipse *Run Configurations*. Similar to the *Java Run Configuration* known to all Java developers using Eclipse, one can choose the *Main*-class to be executed and run the selected Java application. Figure 5.1 illustrates such a *Run Configuration*. *Senseo* then packs the application in a JAR to fit the technical requirements of *MAJOR*. It further opens a socket to listen for the to be transmitted dynamic information and then starts the packed application with *MAJOR*, *i.e.*, the instrumented JVM. During the execution, the application can be used as usual.


Run Configurations
Create, manage, and run configurations

type filter text
- Eclipse Application
- Java Applet
- Java Application
  - DI
- JUnit
- JUnit Plug-in Test
- MIDI Sequence
- OSGi Framework
- PDA Application
- Senseo Profiler
  - DI (1)
  - jedit
- Task Context Test
- Test

Filter matched 16 of 16 items

Name: jedit
Main | Major | Major Configurer
Project: jedit [Browse...]
Main class: JEdit [Search...]
☐ Include system libraries when searching for a main class
☐ Include inherited mains when searching for a main class
☐ Stop in main

[Apply] [Revert]
[Close] [Run]

Figure 5.1: Run Configuration to launch an Eclipse Project in the instrumented JVM.

5.2 Information Processing

Senseo receives the transfered CCT over a socket and analyzes and processes the received CCT. This means it decompresses the received information and extracts the collected dynamic metrics. Additionally, it aggregates these metrics and stores them in its own storage system which is optimized for fast access from the IDE. To finish the process of parsing the received CCT, the filled caches holding data such as the clusters grouping methods based on their invocation count, are cleared and counters are reset. Finally, *Senseo* notifies subscribed listeners such as opened editors or other visualizations about the changed data basis. Figure 5.2 gives an overview of the setup of our approach.

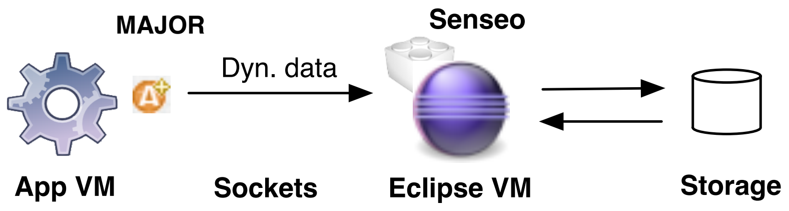
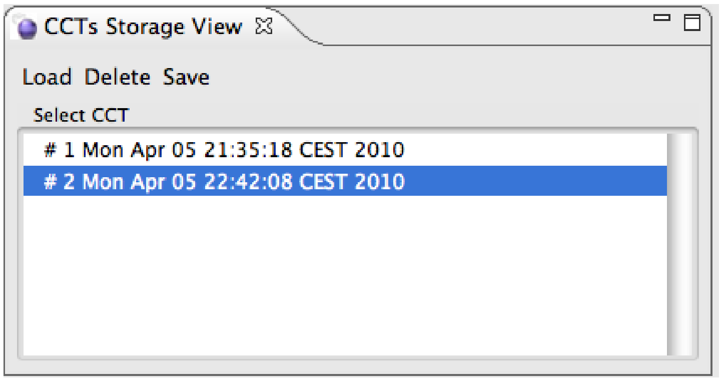


Figure 5.2: Setup to gather dynamic information.



1 Mon Apr 05 21:35:18 CEST 2010
2 Mon Apr 05 22:42:08 CEST 2010

Figure 5.3: List and selection of available CCTs.

Senseo separately stores each execution, that is, each CCT built over the lifetime of an execution. Usually *Senseo* displays the last collected CCT in the Storage View (Figure 5.5, (B)), however, it is possible to select other CCTs and thus display dynamic data aggregated over many application runs. Figure 5.3 shows the Storage View in which developers can select the currently active CCT, store collected CCTs to disk, load them from exported files as well as remove them from the list of currently loaded CCTs.

The storage system provides access to the dynamic information, similar to how Java applications are organized: A storage instance contains packages which contain classes which contain methods. Additionally, the storage system provides an easy way to query the gathered information as well as the aggregated information such the class with the most invoked methods. *Senseo* caches any statistical information such as the maximum number of invocations within a class until further dynamic information is received; this guarantees efficient access to such data after a primary initialization. Aggregated statistical information can for example contain the allocated amount of bytes for the complete class or package.

Figure 5.4 shows the class diagram and reveals how the storage system is organized and how the gathered dynamic information is stored within *Senseo*. The various gathered categories of dynamic information are stored at a method level in a *SenseoMethod-Edge* connecting the sending and receiving method. *SenseoMethod*, *SenseoClass* and *SenseoPackage* aggregate this dynamic information for all elements they contain (such as all methods in a class) and provide means to query the aggregated data, for example by using *getMetricCount(String metric)*.

5.3 Enrichments

We integrate several dynamic metrics directly in the static source code in order to provide developers with insights into runtime behavior while studying code. We discuss these visualizations in the following sections.

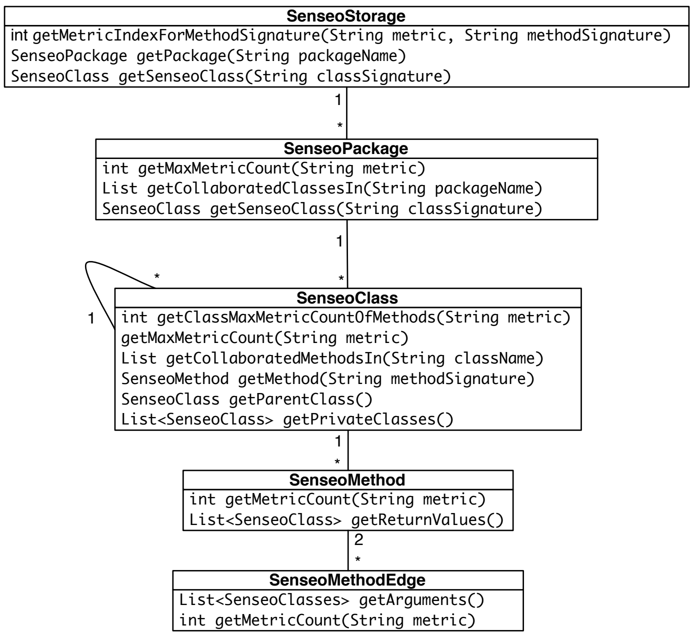
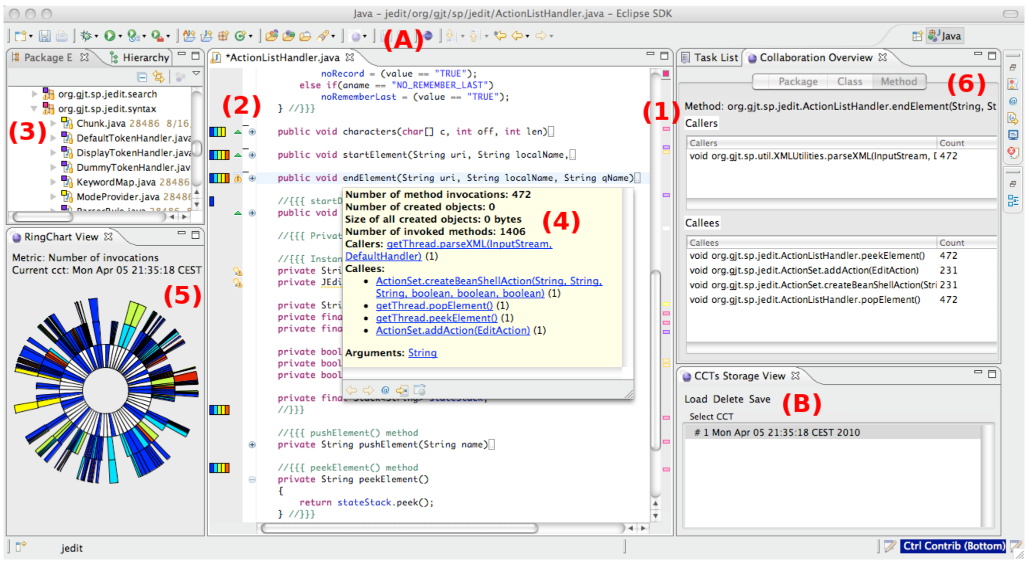

SenseoStorage
int getMetricIndexForMethodSignature(String metric, String methodSignature)
SenseoPackage getPackage(String packageName)
SenseoClass getSenseoClass(String classSignature)
1
*
SenseoPackage
int getMaxMetricCount(String metric)
List getCollaboratedClassesIn(String packageName)
SenseoClass getSenseoClass(String classSignature)
1
*
*
1
SenseoClass
int getClassMaxMetricCountOfMethods(String metric)
getMaxMetricCount(String metric)
List getCollaboratedMethodsIn(String className)
SenseoMethod getMethod(String methodSignature)
SenseoClass getParentClass()
List<SenseoClass> getPrivateClasses()
1
*
SenseoMethod
int getMetricCount(String metric)
List<SenseoClass> getReturnValues()
2
*
SenseoMethodEdge
List<SenseoClasses> getArguments()
int getMetricCount(String metric)

Figure 5.4: Class diagram of the storage system.



Metric: Number of invocations
Current cct: Mon Apr 05 21:35:18 CEST

```
            noRecord = (value == "TRUE");
        else if(aname == "NO_REMEMBER_LAST")
            noRememberLast = (value == "TRUE");
    } //}}}

    public void characters(char[] c, int off, int len)

    public void startElement(String uri, String localName,

    public void endElement(String uri, String localName, String qName)

    //{{{ startD
    public void

    //{{{ Privat

    //{{{ Instan
    private Stri
    private JEdi

    private Stri
    private fina
    private fina

    private bool
    private bool
    private bool

    private fina
    //}}}

    //{{{ pushElement() method
    private String pushElement(String name)

    //{{{ peekElement() method
    private String peekElement()
    {
        return stateStack.peek();
    } //}}}
```

Number of method invocations: 472
Number of created objects: 0
Size of all created objects: 0 bytes
Number of invoked methods: 1406
Callers: getThread.parseXML(InputStream, DefaultHandler) (1)
Callees:
- ActionSet.createBeanShellAction(String, String, String, boolean, boolean, boolean) (1)
- getThread.popElement() (1)
- getThread.peekElement() (1)
- ActionSet.addAction(EditAction) (1)

Arguments: String

Method: org.gjt.sp.jedit.ActionListHandler.endElement(String, St

Callers

Callers	Count
void org.gjt.sp.util.XMLUtilities.parseXML(InputStream, [	472

Callees

Callees	Count
void org.gjt.sp.jedit.ActionListHandler.peekElement()	472
void org.gjt.sp.jedit.ActionSet.addAction(EditAction)	231
void org.gjt.sp.jedit.ActionSet.createBeanShellAction(Stri	231
void org.gjt.sp.jedit.ActionListHandler.popElement()	472

1 Mon Apr 05 21:35:18 CEST 2010

Figure 5.5: Overview of *Senseo* and all its techniques to integrate dynamic information into Eclipse

5.3.1 Dynamic Metrics Selection

As we gather four different types of dynamic metrics we provide a possibility to choose which metric is used for the different visualizations. Besides the number of invocations of methods, we also provide metrics such as the number of objects a method creates or the amount of memory it allocates. Furthermore, to the already mentioned three dynamic metrics we provide a dynamic metric which represents the number of message

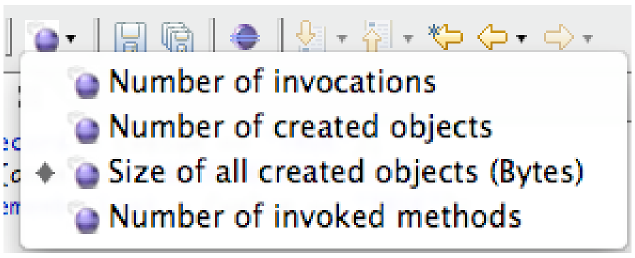

Number of invocations
Number of created objects
◆ Size of all created objects (Bytes)
Number of invoked methods

Figure 5.6: Selection of dynamic metric to be used for categorization.

sends to a certain method and which is derived from the number of method invocations. We placed the selection in the main menu bar of Eclipse (Figure 5.5, (A)). Selecting a metric immediately switches the source for all the different visualizations and updates them amongst all open views. Figure 5.6 shows the expanded menu entry containing all available dynamic metrics as well as the current selected one (Size of all created objects).

5.3.2 HeatMaps

Based on the work of Röthlisberger et al. [59] we use HeatMaps to separate the values of each dynamic metric into different clusters and highlight based on these clusters different hotspots within the application. We reuse the concept of HeatMaps in all parts of the system to visualize such clusters.

Ruler columns are one of the visualizations where HeatMaps play a central role. There are two kind of rulers next to the source editor: (i) the standard ruler on the left (Figure 5.5, (1)) showing local information and (ii) the overview ruler on the right (Figure 5.5, (2)) giving an overview of the entire file opened in the editor. In the traditional Eclipse IDE these rulers denote annotations for errors or warnings in the source file. Ruler (i) only shows the annotations for the currently visible part of the file, while the overview ruler (ii) displays all available annotations for the entire file. Clicking on such an annotation in (ii) brings the developer to the annotated line in the source file, for instance to a line containing an error.

We extended these two rulers to also display dynamic metrics. For every executed method in a Java source file the overview ruler presents, for instance, how often it has been executed on average per system run using three different icons colored in a hot/cold scheme: *blue* means only a few, *yellow* several, and *red* many invocations [59]. Clicking on such an annotation icon causes a jump to the declaration of the method in the file. The ruler on the left side provides more detailed information: It shows on a scale from 1 to 6 the frequency of invocation of a particular method compared to all other invoked methods as shown in Figure 5.7. A completely filled bar denotes methods that have been invoked the most in an application.
To associate the continuous distribution of metric values to one of the discrete scales, we use the *k-means* clustering algorithm [39].
The dynamic metrics in these two rulers allow developers to quickly identify hotspots

in their code while looking at one opened file and thus they help developers to locate candidate methods for optimization or further investigation. Such hotspots can differ from the currently selected dynamic metric and depend on the runtime behavior a developer is interested in. The applied heat metaphor allows different methods to be compared in terms of number of invocations.

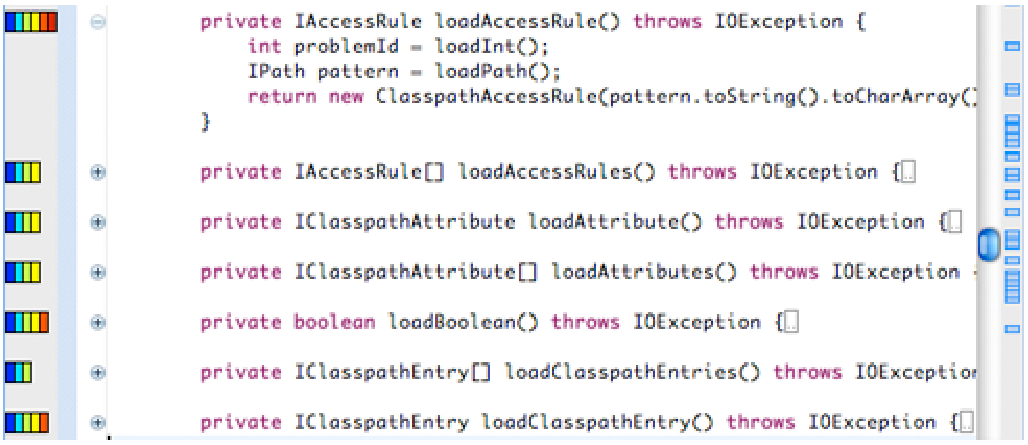

```
private IAccessRule loadAccessRule() throws IOException {
    int problemId = loadInt();
    IPath pattern = loadPath();
    return new ClasspathAccessRule(pattern.toString().toCharArray(
}

private IAccessRule[] loadAccessRules() throws IOException {

private IClasspathAttribute loadAttribute() throws IOException {

private IClasspathAttribute[] loadAttributes() throws IOException

private boolean loadBoolean() throws IOException {

private IClasspathEntry[] loadClasspathEntries() throws IOExceptio

private IClasspathEntry loadClasspathEntry() throws IOException {
```

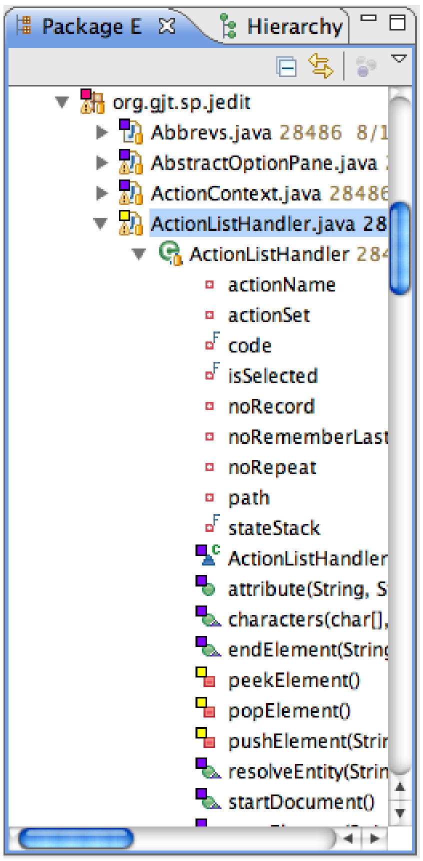
Figure 5.7: Rulers left and right of the editor view showing dynamic metrics.

To see fine-grained values of the dynamic metrics, the annotations in the two columns are also enriched with tooltips. Developers hovering over a heat bar in the left column or over the annotation icon in the right bar get a tooltip displaying precise metric values, for instance exact total numbers of invocations or even number of invocations from specific methods or receiver types.

Package Explorer is the primary tool in Eclipse to locate packages and classes of an application. *Senseo* augments the package explorer (Figure 5.8) with dynamic information to guide the developer at a high level to the source artifacts of interest, for instance to classes creating many objects. For that purpose, we annotate packages and classes in the explorer tree with icons denoting the degree with which they contribute to the selected dynamic metric such as amount of allocated memory (Figure 5.5, (3)). A class for instance aggregates the metric value of all its methods, a package the value of all its classes. The same metric values as in the overview ruler are used to map them to three different package explorer icons: *blue*, *yellow*, and *red*, representing a heat coloring scheme. These annotations are further populated to other views within Eclipse such as the the *Class Hierarchy* (Figure 5.9), which provides a view on the hierarchy of the current active class.

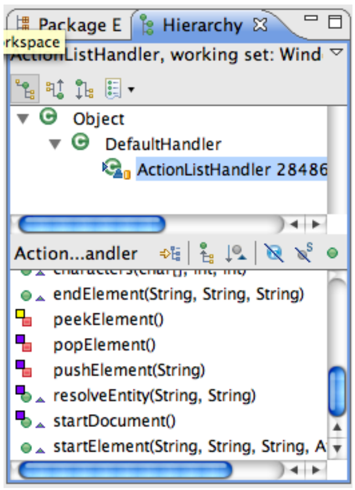
5.3.3 Tooltip

As a technique to complement source code without impeding its readability we opted to use *tooltips* (Figure 5.5, (4)), small windows that pop up when the mouse hovers over a source element (a method name, a variable, *etc.*). Tooltips are interactive, which means the developer can for instance open the class of a receiver type by clicking on it. This supports the developer in navigating through the source code space.



```
Package E   Hierarchy
org.gjt.sp.jedit
  Abbrevs.java 28486  8/1
  AbstractOptionPane.java
  ActionContext.java 28486
  ActionListHandler.java 28
    ActionListHandler 284
      actionName
      actionSet
      code
      isSelected
      noRecord
      noRememberLast
      noRepeat
      path
      stateStack
      ActionListHandler
      attribute(String, S
      characters(char[],
      endElement(String
      peekElement()
      popElement()
      pushElement(Strir
      resolveEntity(Strin
      startDocument()
```

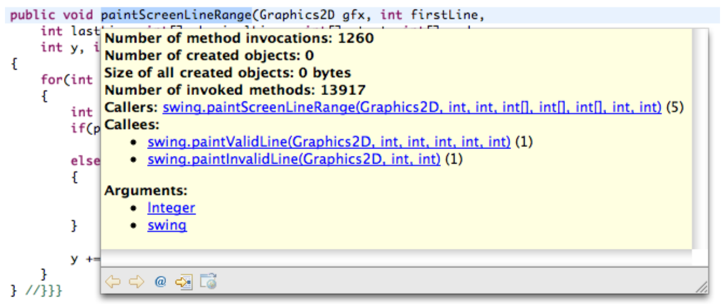
Figure 5.8: Package Explorer enriched with decorators.



```
Package E   Hierarchy
rkspace
ActionListHandler, working set: Wind
Object
  DefaultHandler
    ActionListHandler 28486
Action...andler
endElement(String, String, String)
peekElement()
popElement()
pushElement(String)
resolveEntity(String, String)
startDocument()
startElement(String, String, String, A
```

Figure 5.9: Hierarchy View enriched with decorators.

Method header. The tooltip that appears on mouse over the method name in a method header shows (i) all callers invoking that particular method, (ii) all callees, that is, all methods invoked by this method, and optionally (iii) all argument and return value types. We also show how often a particular invocation occurred. For instance for a callee, we

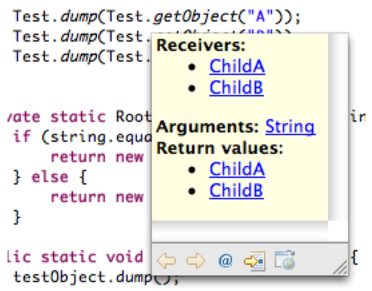


```
public void paintScreenLineRange(Graphics2D gfx, int firstLine,
    int last
    int y, i
{
    for(int
    {
        int
        if(p

        else
        {

        }

        y +=
    }
} //}}}
```

Number of method invocations: 1260
Number of created objects: 0
Size of all created objects: 0 bytes
Number of invoked methods: 13917
Callers: swing.paintScreenLineRange(Graphics2D, int, int, int[], int[], int[], int, int) (5)
Callees:
- swing.paintValidLine(Graphics2D, int, int, int, int, int) (1)
- swing.paintInvalidLine(Graphics2D, int, int) (1)

Arguments:
- Integer
- swing

Figure 5.10: Tooltip appearing for a method name in its declaration.



```
Test.dump(Test.getObject("A"));
Test.dump(Test.
Test.dump(Test.

vate static Root                              in
if (string.equa
    return new
} else {
    return new
}

lic static void                               {
testObject.dump();
```

Receivers:
- ChildA
- ChildB

Arguments: String
Return values:
- ChildA
- ChildB

Figure 5.11: Tooltip for a message send occurring in a method.

display the qualified name of the method containing the call site and the number of invocations from this callee. Optionally, we also display the type of object to which the message triggering the invocation of the current method was sent, if this is a sub-type of the class implementing the current method. For a callee we provide similar information: The class implementing the invoked method, the name of the message, and how often a particular method was invoked. Additionally, we can show concrete receiver types of the message send, if they are not the same as the class implementing the called method. Figure 5.10 shows a concrete method name tooltip for method `paintScreenLineRange`.

In a method header, we can optionally show information about argument and return types, if developers have chosen to gather such data. Tooltips presenting this information appear when the mouse is over the declared arguments of a method or the defined return type.

Method body. We also augment source elements in the method body with tooltips. Each message send defined in the method we map to information stored in our storage system, we provide the dynamic callee information similarly as for the method name, namely concretely invoked methods, optionally along with argument or return types that occurred in this method for that particular message send at runtime, as shown in Figure 5.11. For the return statements (*return*) the tooltip shows the return types the method answered. For arguments passed to the method we show the argument types with which the method was invoked.

5.3.4 Calling Context Ring Chart (CCRC)

The CCRC [42] offers a condensed visualization of a Calling Context Tree (CCT) which is stored in the storage system. The CCRC uses the CCT selected in the Storage View to generate the visualization. If multiple CCTs are selected the latest CCT is displayed. It 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 proportionally to the aggregated metric contribution of the corresponding CCT subtree.

Senseo integrates a CCRC view of the CCT (Figure 5.5, (5)) which is interlinked with the static source view. The ring 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 [42] visualizes only a single, aggregated dynamic metric, by sizing each ring segment according to the aggregated metric contribution of the corresponding CCT subtree, the new CCRC version integrated in *Senseo* supports visualization by coloring each ring segment according to the calling context's "hotness" with respect to the selected metric, as seen in Figure 5.12. 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 red.

5.3.5 Collaboration Overview

In a separate view next to the source code editor (Figure 5.5, (6)), *Senseo* presents all dynamic collaborators for the currently selected artifact. For instance, if a method has been selected, the Collaboration Overview shows the collaborators at the package, class, or method level (Figure 5.13); that is, it lists all packages or classes invoking methods of the package or class in which the selected method is declared (callers). Figure 5.14 shows the collaboration on a class level. Furthermore, the Collaboration Overview shows all packages or classes with which the package or class declaring the method is actively communicating (callees). For the method itself, the Collaboration Overview lists all direct callers and callees. This overview is navigable; clicking on the selected method or class opens a window in the editor pointing to the selected resource. Additionally, we can double-click on a certain class or package to get a more specific overview (Figure 5.15) of the selected item on a class or package level.

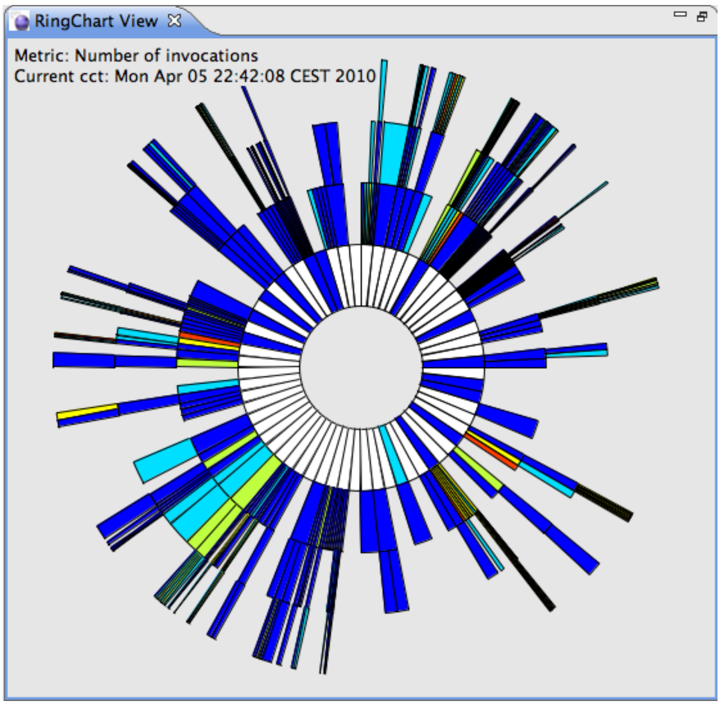
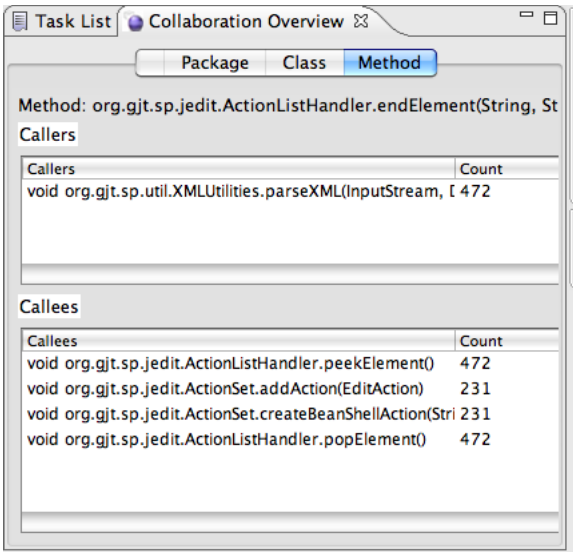


Figure 5.12: *CCRC* colored based on the number of invocations.



Method: org.gjt.sp.jedit.ActionListHandler.endElement(String, St

Callers

Callers	Count
void org.gjt.sp.util.XMLUtilities.parseXML(InputStream, [	472

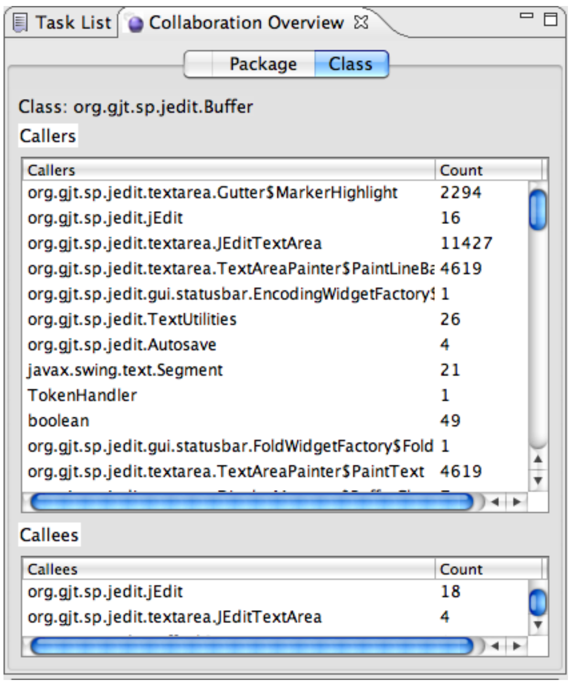
Callees

Callees	Count
void org.gjt.sp.jedit.ActionListHandler.peekElement()	472
void org.gjt.sp.jedit.ActionSet.addAction(EditAction)	231
void org.gjt.sp.jedit.ActionSet.createBeanShellAction(Stri	231
void org.gjt.sp.jedit.ActionListHandler.popElement()	472

Figure 5.13: Collaboration Overview showing callers and callees of a method.

5.4 Summary

We introduced *Senseo* as a prototype approach to gather, aggregate and visualize dynamic information in the IDE. We showed how *Senseo* provides an effective way to gather dynamic information. Moreover, it provides means to store, aggregate and query such information and enables developers to aggregate data gathered over multiple executions.



Class: org.gjt.sp.jedit.Buffer

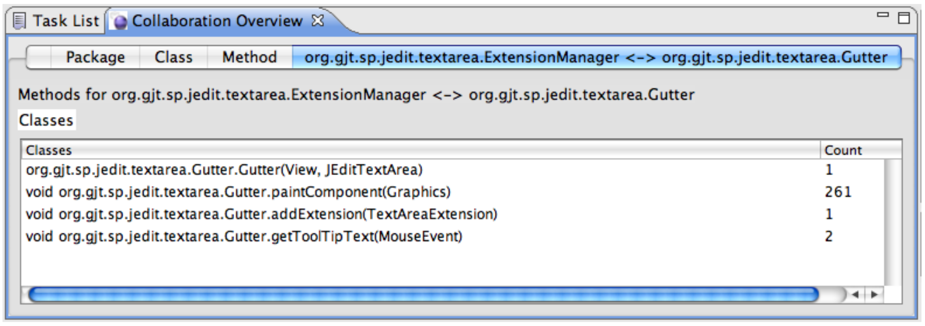
Callers

Callers	Count
org.gjt.sp.jedit.textarea.Gutter$MarkerHighlight	2294
org.gjt.sp.jedit.jEdit	16
org.gjt.sp.jedit.textarea.JEditTextArea	11427
org.gjt.sp.jedit.textarea.TextAreaPainter$PaintLineBa	4619
org.gjt.sp.jedit.gui.statusbar.EncodingWidgetFactory$	1
org.gjt.sp.jedit.TextUtilities	26
org.gjt.sp.jedit.Autosave	4
javax.swing.text.Segment	21
TokenHandler	1
boolean	49
org.gjt.sp.jedit.gui.statusbar.FoldWidgetFactory$Fold	1
org.gjt.sp.jedit.textarea.TextAreaPainter$PaintText	4619

Callees

Callees	Count
org.gjt.sp.jedit.jEdit	18
org.gjt.sp.jedit.textarea.JEditTextArea	4

Figure 5.14: Collaboration Overview showing collaboration on a class level.


Task List
Collaboration Overview
Package
Class
Method
org.gjt.sp.jedit.textarea.ExtensionManager <-> org.gjt.sp.jedit.textarea.Gutter

Methods for org.gjt.sp.jedit.textarea.ExtensionManager <-> org.gjt.sp.jedit.textarea.Gutter

Classes

Classes	Count
org.gjt.sp.jedit.textarea.Gutter.Gutter(View, JEditTextArea)	1
void org.gjt.sp.jedit.textarea.Gutter.paintComponent(Graphics)	261
void org.gjt.sp.jedit.textarea.Gutter.addExtension(TextAreaExtension)	1
void org.gjt.sp.jedit.textarea.Gutter.getToolTipText(MouseEvent)	2

Figure 5.15: Collaboration Overview showing collaboration between two classes.

We outlined the different key elements of *Senseo* and showed how these features such as ruler columns, tooltips or Collaboration Overview support developers during typical software maintenance tasks.

Chapter 6

Validation

First, we show how *Senseo* supports developers in addressing different software maintenance use cases by solving the problems outlined in Section 3.1. However, these use cases do not fully validate our approach. Consequently, we motivate conducting a controlled experiment to validate our approach and describe:

1. the experimental design,
2. the hypotheses,
3. the subject system,
4. the experimental procedure,
5. the statistical evaluation of our experiment,
6. possible threats to validity.

To conclude this chapter we discuss observations made during the experiment and put them into relation with the results. We further discuss developer feedback encompassing both informal and qualitative feedback.

6.1 Solving the Use Cases

In Section 3.1 we raised two questions about a typical code example from the Eclipse JDT. This example can be seen as typical for many other object-oriented applications written in Java. Such applications contain code that is similar to our example (`IJavaElement`) and also similarly difficult to understand purely with information about the static application structure. Using *Senseo* developers are able to reason about which implementations of `getJavaProject` are invoked and can reveal which types are stored in variable `element`.

First, to determine the `getJavaProject` methods invoked in the given code example, developers hold the mouse over the call site written in source code to get a tooltip mentioning all distinct methods that have been invoked at runtime at this call site, along with the number of invocations. This tooltip saves us from browsing the statically generated list of more than 20 declarations of this method by showing us precisely the actually invoked methods. Second, to find out which types of objects have been stored in `element`, we can look at the message send to `getElementType` whose statically defined receiver is the variable `element`. The tooltip also shows the runtime receiver types of a message send, which are all types stored in `element` in this case. It turns out that the types of `element` are `SourceField`, `LocalVariable` but also `SourceMethod`, thus the `if` statements in this code have to be extended to also cover `SourceMethod` elements. We were unable to statically elicit this information.

To assess the efficiency of the various invoked `getJavaProject` methods, we navigate to the declaration of each such method. The dynamic metrics in the ruler columns reveal how complex an invocation of this method is, for instance how many objects an invocation creates on average, even depending on the receiver type. Thanks to these metrics we find out that if the receiver of `getJavaProject` is of type `LocalVariable`, the code searches iteratively in the chain of parents of this local variable for a defined Java-project. We can optimize this by searching directly in the enclosing type of the local variable.

Addressing these two questions reveals that *Senseo* can help developers to reason more efficiently about typical tasks in software maintenance. Additionally, we can see that developers are also able to find information which would not be retrievable by relying purely on static source code analysis. However, these use cases are very limited and give only small feedback about how our approach is useful to developers. To validate our approach in a more reliable way, we designed and conducted a controlled experiment to validate our approach with professional software developers.

6.2 Experimental Design

We conducted a controlled experiment with 30 professional Java developers to measure the expected impact of *Senseo* on performing typical software maintenance tasks on object-oriented systems. We now describe the experimental design, the subjects, the evaluation procedure, the final results (including qualitative feedback) as well as threats to validity.
This experiment aims at quantitatively evaluating the impact of the *Senseo* plugin and the dynamic information it integrates into 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. [10] which evaluated a trace visualizing tool called *EXTRAVIS*.

6.2.1 Hypothesis

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

Consequently, we formulate these two alternative hypotheses:

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

6.2.2 Subjects

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. 15 subjects has been working at least one or two years generally with Java, while 12 people has been working for more than 3 years mainly as Java Developers. All participants are familiar with the Eclipse IDE either for Java Development or other languages, such as Python or Ruby.

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 an average 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. The four areas in detail:

- **Java:** 23 subjects have much or expert experience with Java, 5 subjects know and use Java, while the remaining 2 subjects have only little experience with Java.

Table 6.1: 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

- **Eclipse:** 18 subjects are working daily with Eclipse and rated themselves as experts or claimed to have much experience, while 11 know Eclipse but rarely use it and only 1 person has little experience with Eclipse.
- **Unfamiliar code:** 3 people claimed to be experts in working with unfamiliar code. 19 subjects have much experience with reverse engineering code of other people, while 8 people have only little experience.
- **Dynamic analysis:** One person rated herself as an expert in the field of dynamic analysis, while 11 subjects have much experience. Also 11 subjects are rarely employing dynamic analysis techniques and 7 persons claimed to have only little experience.

Additionally, we asked the subjects about their experience in developing or maintaining open source projects to verify how many people are familiar with looking at their own or other open source projects. 3 considered themselves to be experts in maintaining such projects, while 3 subjects claimed to be experienced. The majority (17) has no (9) or only little (8) experience, while 7 subjects rarely contribute to open source projects.

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 6.1.

In Appendix B we added the subject questionnaire as well as the results of each question.

6.2.3 Subject System and Tasks

As a subject system we have chosen *jEdit*[1], 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

[1]http://www.jedit.org/

Table 6.2: The nine activities by Pacione et al.

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

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. [44]. They identified nine principal activities for reverse engineering and software maintenance tasks covering both static and dynamic analysis. Table 6.2 gives an overview of these activities. Based on these activities Pacione et al. 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 6.3 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, so 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. Beforehand, the experimenters solved all tasks themselves to prepare an answer model according to which the subjects' answers were corrected. 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 co-authors of this article) answered all prepared questions. We compared and combined both solutions

Table 6.3: The five software maintenance tasks

Task	*Activities*	*Description*
1.1	A 1, 9	Locating a feature in code 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
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 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 two features on a fine-grained method level to locate a defect in a 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 and analyzing its performance
5.2	A 5, 6, 7, 8	Comparing this algorithm to another, similar algorithm in terms of efficiency

to form an answer model which we then used to grade the subjects' answers.

We also ran a pre-test by giving the questions to two students. In this pre-test we evaluated if the allocated time slot of two hours is reasonable to complete all questions and whether our answer model is sound. The two experimenters graded the solutions of the two students independently and the ratings only differed in one point for one single question of a student, otherwise the ratings were the same. This pre-test also helped us to formulate the questions slightly differently for better clarity. Furthermore, we simplified three questions as they were too complex and time-consuming in the beginning.

6.2.4 Experimental Procedure

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 information about the *Senseo* plugin. Furthermore, we provided the *Senseo* subjects with a short description and a screenshot highlighting and explaining the core features of the *Senseo* plugin. This documentation served as a reference during the experiment and contained similar explanations as the Users Guide to *Senseo* (see Appendix A).

We only answered clarification questions during the introduction to give the same amount of information to all subjects. After the introduction we allowed the subjects to play with *Senseo* for 20 minutes to get a feeling for it as none of the subjects had ever used *Senseo* before.

Afterwards, we started the experiment. We supervised all subjects during the entire experiment and recorded the time they took to answer each question. Appendix C contains a copy of the questions given to the subjects. 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. 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. 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.

6.2.5 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 according to the overlap with the answer model, which forms a set of expected answer elements (usually the names of certain source artifacts).

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.

6.3 Results

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

Table 6.4: 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

availability of dynamic information in the IDE is most useful. Finally, we evaluate the qualitative feedback we gathered by means of a debriefing questionnaire.

In the appendix we added detailed results of the experiment tasks (Section C.2, Section C.3) as well as the questionnaire given to the subjects after the experiment (Appendix D).

In the obtained experiment results we could not find any outliers resulting from extraordinary conditions. 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. Even though we removed one task, the remaining four tasks still cover all of the nine maintenance activities defined by Pacione (compare Table 6.2).

6.3.1 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 6.1.

To statistically verify whether the *Senseo* plugin 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 6.4), 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 the *Senseo* plugin as the p-value is with 0.0016 considerably lower than α=0.05 (see Table 6.4).

From the observations of subjects during the experiment, from their informal feedback during the debriefing interviews and particularly from the formal questionnaires (discussed in detail 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 views (presented in 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 Overview accurately presents all source artifacts that are related or collaborating with 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 presumably be even better if participants had used the *Senseo* plugin in their daily work before doing the experiment.

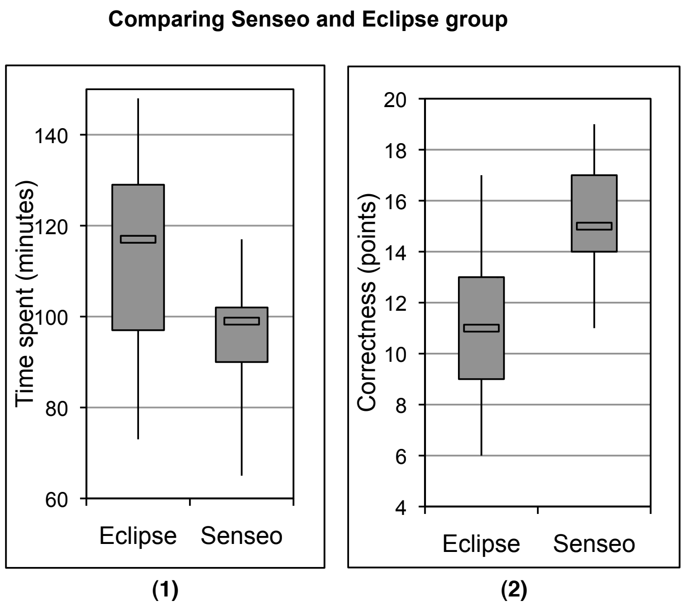


Figure 6.1: Box plots comparing time spent and correctness between control and experimental group.

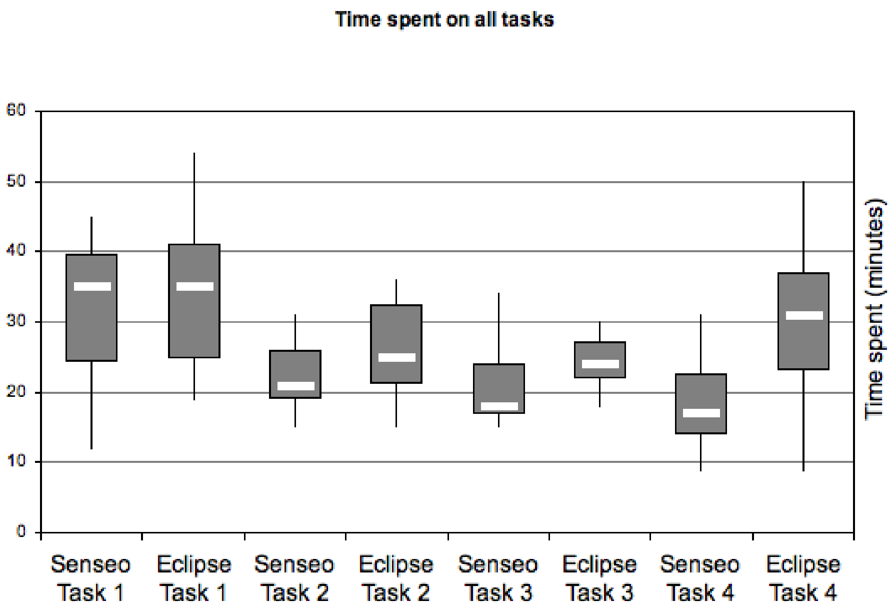


Figure 6.2: Box plots comparing time spent between control and experimental group on all tasks.

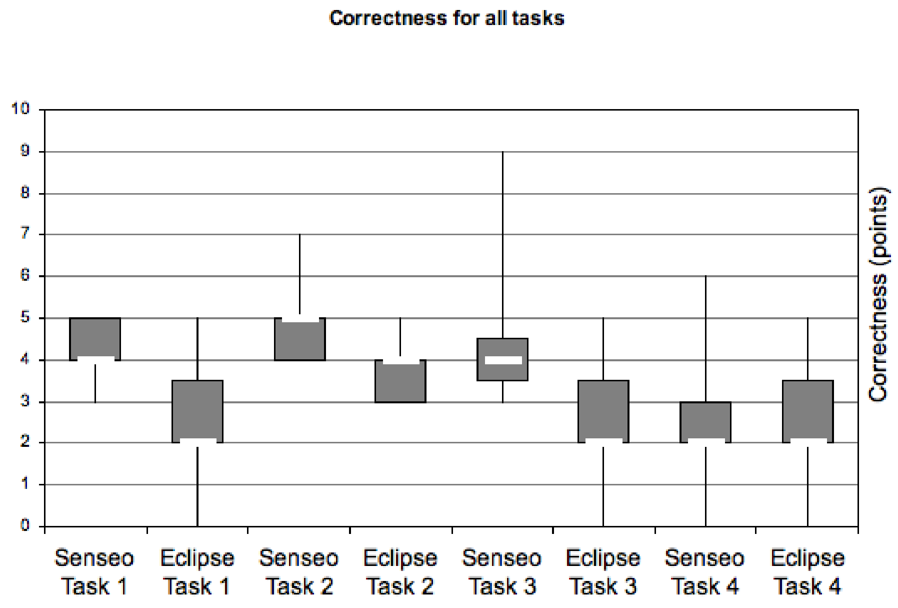


Figure 6.3: Box plots comparing correctness between control and experimental group on all tasks.

6.3.2 Correctness Results

The *Senseo* group's answers for the four maintenance questions are 33.5% more correct, which is also shown in the box plot in Figure 6.1. Additionally, Figure 6.2 and Figure 6.3 show box plots comparing time spent and correctness of the *Senseo* and the control group on each task.

To test the null hypothesis $H2_0$ which suggests that there is no effect of the availability of the *Senseo* plugin 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 6.4). As the t-test gives a p-value of 0.0001 which is clearly below α=0.05, we reject the null hypothesis and accept the alternative hypothesis H2, which means that having available the *Senseo* plugin during software maintenance activities supports developers to more correctly solve maintenance tasks.

The availability of the *Senseo* plugin increases the correctness of answers probably due to the following reasons :

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

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 6.5: Task individual performance concerning time required and correctness.

address a task.

6.3.3 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 6.5 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 the *Senseo* plugin both in terms of time required to solve them and the correctness of the solution. However, for task 4 the benefit of the *Senseo* plugin is less pronounced.

Furthermore, we examined more in detail which tools are useful for what kind of tasks by analyzing the results of the subjects using *Senseo* and the questionnaire they completed at the end. Figure 6.4 presents an overview of how useful a tool was to complete a task. Usefulness has been categorized in five categories, ranging from 0 (tool have not been used at all) to 5 (tool was a main contribution to solve the task). We counted only people using information from the specific tool. If they looked at it but believed that it did not provide any value, we counted the usefulness as 0. The *CCRC* has not been widely used by the subjects due to several reasons we outline in Section 6.4. Concerning the other tools, we can clearly see that the Collaboration Overview and the tooltips have been used to reveal exact caller and callee information as well as to reason about the collaboration within the software system. HeatMaps as well as the annotations and the decorators were mainly useful while navigating the source space to detect hotspots such as methods that have been frequently invoked. The Collaboration Overview was for all subjects at least a starting point to find the appropriate methods or source artifacts, while the tooltips were useful to navigate and study the software system on a fine-grained level.

Coming back to the kind of tasks introduced in Section 6.2 that we wanted to support with *Senseo*, we can conclude that the *Senseo* plugin successfully aided developers performing such tasks. The experimental task 1 refers to task type 1, task 2 to type 2 and 3, and task 3 to type 4, while for task 4 we consider lower level information as more relevant, such as information on a method body level, which is currently not supported by *Senseo*.

6.4 Feedback

In the experiment we collected also qualitative feedback by means of a questionnaire to evaluate the impact of particular features of the *Senseo* plugin on specific kinds of

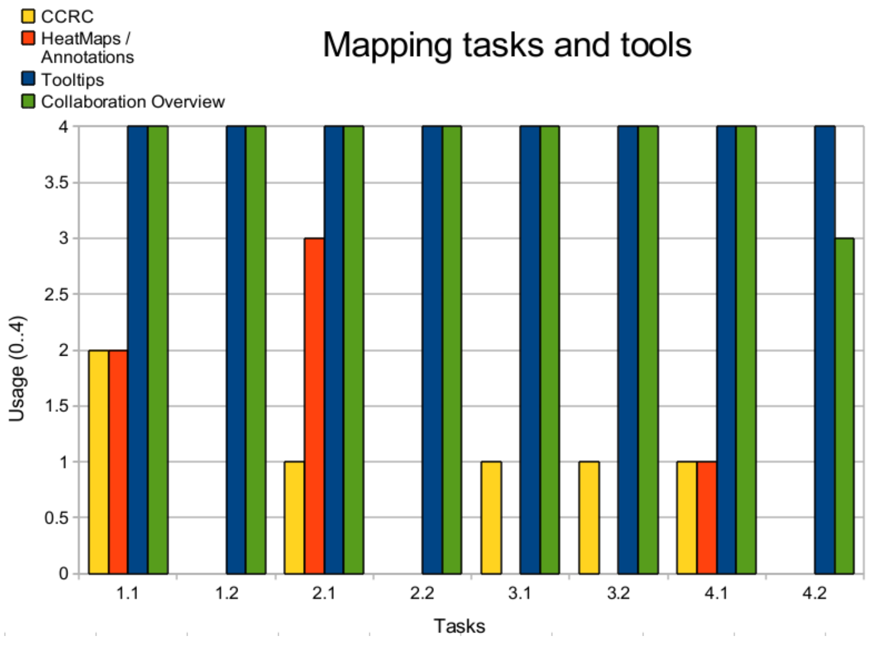


Figure 6.4: Mapping how much tools have been used in tasks.

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 6.6: Percentage of subjects using specific dynamic information in particular tasks

maintenance tasks. This evaluation yields answers to the question which *Senseo* feature and which kind of dynamic information is actually relevant or useful for what kind of software maintenance tasks. In the following sections we discuss the obtained feedback and outline several conclusions we drew.

6.4.1 Qualitative Feedback

In Table 6.6 we list for each task the percentage of subjects that used a specific kind of dynamic information integrated by the *Senseo* plugin (evaluation of the question "Did you use dynamic information X in task Y?"), and Table 6.7 presents how useful subjects rated each *Senseo* technique on 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 solution process is very well supported by the availability of dynamic information in IDEs: (i) tasks requiring

Table 6.7: 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 Overview	3.7

developers to understand how different source artifacts collaborate with or 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 the *Senseo* plugin while for task 4 (low level defect correction) dynamic information was less useful.

Appendix D contains for further study all the questions of the experiment questionnaire as well as the detailed results of each question.

6.4.2 Informal Feedback

Besides the qualitative feedback the subjects also provided informal feedback during discussions after the experiment. Some feedback was concerned with feature requests for further visualizations or improvements which we discuss in Section 8.1. Mainly, the subjects underlined the value of dynamic information in IDEs and mentioned that they would be very happy to have such kind of information available during their daily work. Some subjects mentioned that in the very first moment they have been misled by HeatMaps, especially in ruler columns, as the HeatMaps have not been appropriate for the current task the subjects were solving, but were still attracting developer's attention. Furthermore, the subjects stressed the usefulness of the Collaboration Overview and had various suggestions for further improvement. However, it was also noted that sometimes the Collaboration Overview was overloaded with collaborating classes or packages and hence a filtering mechanism would be useful.
Some subjects mentioned also that they would be very interested to see how *Senseo* performs with larger software systems than *jEdit* and how the workflow of collecting dynamic information would fit to systems like J2EE applications. Some skepticism was raised how *Senseo* would scale with such big applications and how useful for instance a Collaboration Overview would be in this case.

6.4.3 Observations

During the experiment we observed the subjects while solving the different tasks and took notes about certain usage behaviors we could study. One of the main observations

was that the *CCRC* was not heavily used and developers lost interest in it the more they proceeded within the experiment. Another observation revealed that the Collaboration Overview was heavily used to navigate through the source code space and study the collaboration within different parts of the software system.
The HeatMaps in views like the package tree were used to quickly determine hotspots within the expanded tree. HeatMaps in the ruler column were used to gain an overview of a displayed class or spot an interesting class. We can conclude that HeatMaps were considered to be a reasonable means to get a quick overview of the system and to navigate towards a certain source artifact.
Subjects very familiar with Eclipse and its daily usage are inclined to use the traditional tools that rely on static source code analysis. They often used *Senseo* either to verify the results of the traditional tools or to combine the different (static and dynamic analysis based) tools to navigate and search faster the source code space for a possible solution. This includes the usage of the Collaboration Overview or even sometimes the *CCRC*. Also the tooltip has been widely used to verify their assumptions and seems to provide an efficient way to study methods.

6.4.4 Feedback Conclusion

From the results evaluating the different *Senseo* concepts, we conclude that developers particularly benefit from the availability of the collaboration views and runtime type information in source code. Also considered to be 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 is 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.

6.5 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.

6.5.1 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 the *Senseo* plugin, 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 anonymized their answers), and we 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 6.2.

6.5.2 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 6.1).

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 [44] 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.

Lastly, we discuss the objection that the control group using the standard Eclipse IDE could have performed better if additional plugins, for instance *JMetrics*[2] would have been available. We did not prevent control group subjects from installing additional tools or plugins into Eclipse. We asked them at the beginning whether they could name a specific plugin to be installed and even allowed them to install additional plugins as they see a need during the experiment. However, none of the subjects opted to install such an additional aid nor could he or she after the completed experiment name a plugin that would have been specifically useful for the maintenance tasks we designed.

[2] `http://sourceforge.net/projects/jmetrics/`

6.5.3 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 [44] of representative maintenance tasks. Furthermore, we asked open questions to the subjects to better model industrial reality than would be possible with multiple choice questions.

The literature recommends avoiding experimental groups consisting of only students [45]. We therefore selected subjects who all have professional experience in industry as software developers as mentioned in Section 6.2. 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 6.2 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.

6.6 Summary

In this chapter we showed how *Senseo* can help to reason about a certain software system while solving common software maintenance tasks. We conclude that the integration of dynamic information is very helpful and supportive for developers. To emphasize this conclusion and to measure the value of *Senseo* in a practical manner, we performed a controlled experiment with 30 professional developers. This experiment reveals that the participants spent 17.5% less time on the maintenance tasks while at the same time providing 33.5% more correct answers. Additionally, we discussed the feedback we received through a qualitative questionnaire and informal discussions with the subjects after the experiment. We could show that this feedback coincides with our own observations we did during the experiment.

Chapter 7

Discussion

Integrating dynamic information into IDEs features several problems such as gathering, storing and aggregating dynamic information or visualizing the aggregated information. In this section we critically discuss different aspects of our work, *i.e. Senseo*. For the key parts of *Senseo* we go more into detail and discuss where *Senseo* is a powerful tool and where it lacks certain features. To conclude we discuss how *Senseo* supports the different activities outlined at the beginning and assess the conducted validation.

IDE Integration. While integrating the different enrichments and visualizations of *Senseo* we had to address several problems. Some of these problems could be solved, others are not yet fully addressed and should be addressed in future work.

- **Data Gathering** - *Senseo* integrates the process of data gathering as an additional Eclipse *Run Configuration* similar to the one launching Java applications, which makes it almost fully transparent to developers to use in their usual workflows. However, *MAJOR* requires the application to execute within a specially instrumented JVM, which makes the integration not as transparent as we wished but which was the most appropriate solution for a seamless integration. Furthermore, the integration of *MAJOR* has certain limitations:
  - *MAJOR* requires us to instrument the used JVM upfront. This is a necessary task if we want to cover the complete JDK code and not only gather dynamic information of application specific code. As this is a one-time task, its costs are affordable and as we integrated the process to instrument the JVM into Eclipse it is also performed without difficulty. We could have integrated the task into the installation process of the *Senseo* plugin which would completely hide the instrumentation from a developer's view. However, we opted to integrate this task into the run configuration to enable developers to choose which JVM they want to instrument.
  - Before its execution, the application is packed into a JAR-archive to execute it with *MAJOR*. This eases our integration with *MAJOR* but restricts developers as they are required to follow the convention to organize their source code in a folder called *src/*, in order to be able to build a JAR-archive. We opted for

this solution to much better decouple *MAJOR* from *Senseo*. Additionally, like this it is much easier to exchange *MAJOR* with a different technique to gather dynamic information.

Performance. As dynamic analysis deals with a huge amount of data, it is important to process this information in an efficient manner; especially if we would like to integrate the aggregated data continuously in the IDE. The first implementation of *MAJOR*, provided a straightforward and naive way to collect and transfer dynamic information, thus suffered from an excessive overhead. *MAJOR* used a naive, non-optimized aspect for collecting dynamic information and always transmitted the complete CCT to the Eclipse plugin using Java's standard serialization mechanism. Even for medium-sized applications, serialization introduced long latencies and generated several hundred megabytes of data. As a case study, we ran Eclipse itself as a target application and analyzed the usage of the JDT core. Without *Senseo*, starting and terminating Eclipse took 53 seconds; executing the same scenario with *Senseo* took 188 seconds, i.e., the overhead of *Senseo* was 255%. The transmitted CCT grew over 200 MB thus loading and storing of the received CCT was problematic.

The recent version of *MAJOR* performs much better and addresses the discussed overhead: The aspects to gather dynamic information are optimized, particularly the code that collects runtime type information. Wherever runtime type information can be statically inferred, the new aspect avoids expensive access to dynamic context information through AspectJ's reflection API. For instance, if all formal method arguments are of primitive or final type, the actual argument types cannot vary at runtime and therefore do not need to be collected. For instance, there is no need to collect runtime argument types in a method `foo` with the formal arguments `foo(int i, String s, double d)`, since `int` and `double` are primitive types and `String` is a final type.

Additionally, we use an optimized serialization mechanism that transmits the CCT in an incremental way, sending only those nodes whose dynamic information has changed since the previous transmission. In addition, the data structures that store dynamic information are optimized as well as since they are accessed frequently. Thanks to the principle of locality [13] the updated parts of CCTs are close together and typically only a small subset of the CCT nodes is transmitted. Thus, we can frequently update the dynamic information in the Eclipse plugin, *e.g.* once per second. The 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. Hence, we can serialize all data with Java's efficient, low-level `DataOutputStream` API instead of using the expensive `ObjectOutputStream` class, which gives us further speed improvements. Repeating the previous case study showed that starting Eclipse took 98 seconds and the transmitted CCT grew only up to 45 MB, which gives us a time overhead of less than 100% as well as an impressively small amount of collected data, *i.e.* only a fifth of the previous size. If we consider that the new optimized *MAJOR* collects even more dynamic metrics, we can conclude that the overhead has actually been reduced even more.

Taking all these improvements into account, we can conclude that *Senseo* relies on an efficient way to gather dynamic information.

- **Integration of dynamic information.** Enrichments such as annotations, HeatMaps or the decorators have been enthuastically accepted and adopted by the subjects during the controlled experiment. Generally the possibility to switch the dynamic metric on which the categorization is based empowers developers to quickly identify hotspots within the software system. The obtained feedback in the validation showed that developers appreciate enrichments within navigation trees and the source code view. However, such enrichments to embed dynamic information into the IDE could also be used in other static source code views such as enriching the call hierarchy view with execution flow information, or it could be combined with other existing tools that are integrated into Eclipse such as the Findbugs Eclipse plugin[1]. There are many other views and tools within Eclipse which could benefit from dynamic information. However, for *Senseo* we have chosen to enrich the tools and views most used by developers during their development work, for instance, the source code editor or the package explorer.

- **Enrichments and visualizations.** Each of the various enrichments and visualizations contributes differently to ease the analysis of software systems. Means to navigate in these enrichments or overviews such as the possibility to jump to the source artifact of the displayed method in the collaboration overview have also been largely accepted and adopted by the subjects. However, they pointed out some limitations, for example concerning annotations and decorators:

  – Annotations and decorators disclosing dynamic information can overlap with enrichments based on static source code analysis. For instance, annotations in the overview column might interfere with existing annotations for search results or compiler warnings. On the one hand, this could be addressed by choosing a different color schema for these enrichments. On the other hand, the annotations or decorators are placed in a very small area and share a small space with a lot of other information contributed by other tools. This means that collisions are likely and often tools override each other's enrichments. Furthermore, as we aim to reveal the hotness of the enriched artifacts such as classes in the package explorer, a color schema similar to HeatMaps is much more intuitive due to its gradient from cold (blue) to hot (red). Therefore, we claim that such a schema is much more likely than any other color schema we could have chosen to not collide with existing annotations and decorations.

  – Clustering methods into their specific groups is only based on a single dynamic metric. This means that the values of the collected dynamic metrics are not combined to calculate different clusters for HeatMap-based enrichments. We considered a combination of multiple metrics to be useful and discussed various ideas how to enable developers to combine different metrics. However, we also raised many concerns about the value of such combinations as nearly none of the possible combinations provides any meaningful value. Further questions such as how the values should be connected and interpreted are raised. For instance, it was unclear to us how we could combine the

[1]http://code.google.com/p/findbugs/

number of object allocations with the number of method invocations. Would an invocation count as much as an allocated object? What would it mean to a developer if a method has often been invoked and has allocated many objects, while another method has only been invoked once but has allocated the same number of objects? How would we group such combinations? Due to these open questions and as we could not find any meaningful combination, we have chosen to not provide any possibility to combine the different metrics.

  - Eclipse does currently not provide any possibility to show multiple tooltips in one pop-up window, which means that our introduced tooltips containing dynamic information are overwriting the existing Javadoc tooltips. Several subjects complained that the absence of the Javadoc tooltips conceals useful information and that it would be much more useful if both (Javadoc and our dynamic information) were displayed in the tooltip. Unfortunately, such a feature would have required us to change the core of the Eclipse framework.

- **Enabling Software Navigation** is a crucial feature of IDEs; especially while performing software maintenance tasks it is important to quickly navigate the software system. We integrated for example hypertext-like links in the displayed information in tooltips to navigate to the related definition. In the Collaboration Overview or the *CCRC* it is possible to easily navigate a selected element by clicking on it while holding the ctrl-key. While the navigation within tooltips and within the Collaboration Overview is easy and has been considered as useful by many subjects, the navigation within the *CCRC* has often been criticized as cumbersome. This limitation is related to the issue that the *CCRC* does not scale well for huge CCTs. In Section 8.1 we outline some ideas to address these drawbacks.

We can summarize that in general the integration of dynamic information in Eclipse is successful. However, each part of the integration has certain limitations. Some limitations are given by the used tools such as *MAJOR* and its upfront instrumentation while other limitations are due to a prototype based implementation not yet covering all possible aspects. The current enrichments proved to be useful to navigate the source space and to conduct typical software maintenance tasks. However, further work is required to improve the existing implementation.

Addressing Shortcomings of traditional IDEs. In Section 1.1 we outlined different activities developers usually perform during software maintenance and which are badly supported in traditional IDEs. *Senseo* is our approach to better support developers in these activities. In the following points we discuss how the different parts of *Senseo* support the outlined activities, but also discuss various limitations of our approach:

- **Understanding execution paths and runtime types of an object-oriented system employing complex hierarchies including abstract classes and interfaces**: Visualizations such as the *CCRC* or enrichments such as tooltips use different means to display information about runtime types or execution paths either on a fine-grained level (tooltips) or on a coarse-grained level (*CCRC*). Tooltips reveal information about actual runtime types and callers or callees of methods, while the *CCRC* reveals the calling context of the currently studied method. Furthermore, as *Senseo* is able to reveal the used implementors of abstract classes or interfaces, we are able to quickly examine how complex hierarchies including abstract classes and interfaces are used.

- **Understanding higher-level concepts such as application layers, models, or separation of concerns**: Decorators in the Package Explorer and HeatMaps help us to understand how and how heavily different parts of the software system have been involved during the execution.
- **Identifying collaboration patterns, that is, how various source artifacts communicate with each other at runtime.** The Collaboration Overview allows us to easily identify which source artifacts communicate with each other at runtime and how frequently this communication occur. Furthermore, the collaboration is not only disclosed on a method level; we can also reason about collaboration between classes or packages.
- **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**: Using HeatMaps, decorators, or the *CCRC* we can easily spot points of interest and focus on them. Furthermore, the Collaboration Overview helps us to identify tightly coupled methods or classes and packages to refactor these quality issues. Additionally, tooltips reveal detailed information about the costs of a method invocation and concrete information about the involved runtime types.
- **Gaining an overview of control flow and execution complexity, for instance to quickly locate performance bottlenecks**. Tooltips provide information about the control flow such as exact information about callees and senders of message sends. Furthermore, the *CCRC* helps us to gain an overview on the execution complexity of the currently studied context.

We conclude that *Senseo* is able to support all our outlined software maintenance activities in a sufficient manner. However, as previously outlined *Senseo* currently provides no solution to restrict data gathering to only a specific feature execution and therefore all enrichments and visualizations are always based on the entire gathered CCT. This means that developers still have to filter out information manually to gain an overview only on a specific feature execution. Developers studying a central layer used by several features are therefore overwhelmed with information from different feature executions, which impedes studying only a specific feature. Furthermore, as subjects noted in the feedback of the experiment the *CCRC* is barely usable to navigate huge CCTs.
Besides these limitations, *MAJOR* is currently not able to analyze intra-procedural control flow, hence *Senseo* provides only a high-level view on execution paths between methods. We could address this limitation by guessing which branch of a control-flow statement has been executed and by looking at all the callees of a method. This procedure, however, is very imprecise; the only proper solution is to improve *MAJOR* to also cover intra-procedural flow or to employ another technique to gather dynamic information capable of addressing this drawback. This means that *Senseo*'s view is limited when it comes to examining complex and large methods.

Validation. First, we validated *Senseo* only on a use case basis. The analyzed use cases matched the software maintenance tasks we discussed in Section 3.1, which we also encountered while developing with the Eclipse framework. As described in Chapter 6, we aimed to verify our approach by conducting a quantitative experiment with 30 professional software developers. The results of the experiment show that 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%. Additionally, the general feedback showed that the integration of dynamic information into the IDE is appreciated by developers and considered as supportive. However, our validation cannot be seen as a field study as all subjects' experience with *Senseo* is limited to the experiment itself. This means that we measured only the impact of *Senseo* on the subjects' efficiency and correctness within the experimental setup and not during their usual work as software developers. To validate *Senseo*'s impact and usefulness on typical activities a developer encounters during her daily development work, evaluation over a longer time period and with a different experimental setup is required. However, due to time constraints we were not able to conduct such a field study. But from the results of our experiment, we conclude that *Senseo* supports developers in conducting typical software maintenance tasks as its availability increased correctness and efficiency.

Summary. In this chapter we discussed different aspects of *Senseo* and its validation. We showed that *Senseo* adequately and seamlessly integrates all important features into Eclipse and addresses the previously discussed limitations of traditional IDEs. However, *Senseo* still has certain drawbacks such as *MAJOR*'s inability to reify intra-procedural control flow. Furthermore, we discussed the advantages as well as the various limitations of the implemented enrichments and visualizations such as that tooltips are hiding important static information. To conclude we showed that we successfully validated *Senseo* in an experimental setup but that further validation is required to measure *Senseo*'s impact and usefulness on developers' daily work concerned with developing and maintaining software systems.

Chapter 8

Conclusions

In this last chapter we conclude our work by summarizing the most important contributions we made during our studies on this topic and by discussing some perspectives for future work in the area of integrating dynamic information in IDEs as well as ideas to improve our existing approach *Senseo*.

In this work we presented the thesis that integrating dynamic information in the IDE supports developers during typical software maintenance tasks. We identified the shortcomings of traditional IDEs relying purely on static source code analysis and showed how the availability of dynamic information helps developers during software maintenance tasks by providing information about the runtime behavior of the software system. Especially in object-oriented software systems, dynamic information increases knowledge about the system which cannot be or is only partially supplied by static source code analysis with expensive computations. In Chapter 3 we motivated the integration of dynamic information into the IDE to address the different shortcomings of traditional IDEs. Furthermore, we identified different requirements to be covered by our work to successfully integrate dynamic information into the IDE. Such corner-points are for example that we need to embed the information directly in the IDE within the familiar tools, that we need to provide means to navigate the source code space based on the dynamic information or that we need to provide higher level overviews. Based on the identified shortcomings and these requirements we elaborated different categories of dynamic information to be gathered. We discussed different techniques to gather dynamic information and presented *MAJOR* as our choice.
For this thesis we targeted the Java language and Eclipse, the most widely used IDE for Java development. We implemented our approach in *Senseo*, an Eclipse plugin providing a framework to gather dynamic information directly within the IDE, processing the received dynamic metrics and integrating the aggregated data into the IDE. *Senseo* consists of the following parts:

- **Data Gathering** - We are able to run the software system to be examined directly from within the IDE with an instrumented JVM.
- **Data Processing / Storage** - *Senseo* receives, process and store the gathered dynamic information. Furthermore, *Senseo* provides means to query aggregated information as well as storing and loading the gathered data.

- **Enrichments** - *Senseo* provides various enrichments to the Eclipse IDE: For instance, it augments the source code view with HeatMaps denoting hotspots within the current source code artifact or it decorates various navigation trees with colored icons to disclose points of interest within the whole software system. Additionally, it enriches tooltips to provide aggregated information about what kind of dynamic information has been gathered for a certain method.
- **Visualizations** - By integrating a *CCRC* view we visualize the calling context tree (CCT) and reveal information about the execution flow. Furthermore, we integrated the concept of HeatMaps within this visualization to group the different parts of the system as well as to disclose hotspots.
- **Collaboration Overview** - To disclose collaboration patterns and to reveal the coupling of the different parts of the examined software system. *Senseo*'s Collaboration Overview provides information about collaborating methods, classes or packages in tables.
- **Navigation** - All the different enrichments, visualizations and overviews encompass means to navigate the source space. For instance, tooltips embed hyperlinks to navigate to the listed elements or the *CCRC* opens the source editor at the selected method by holding *CRTL* and clicking on the node representing the method.

To validate our approach we conducted a controlled experiment with 30 professional software developers to measure the impact of *Senseo* on different typical software maintenance tasks. The result shows that the visibility of dynamic information in the Eclipse IDE yields a significant 17.5% decrease of time spent while significantly increasing the correctness of the solutions by 33.5%.

In Chapter 7 we critically discussed our approach and our contributions. We showed how *Senseo* addresses the identified shortcomings of traditional IDEs and outlined the different shortcomings of the current implementation of *Senseo*.

We can conclude that *Senseo* successfully integrates dynamic information in the static views of the Eclipse IDE and supports Java developers in performing different kinds of software maintenance tasks. Furthermore, it eases the comprehension of the runtime behavior of a software system and discloses hotspots or bottlenecks during execution.

8.1 Future Work

We believe that *Senseo* is a solid framework to form a foundation for further research on integrating dynamic information into IDEs. In the following paragraphs we outline some ideas on future work on *Senseo*:

Intra-Procedural Control Flow. While *Senseo* currently purely focuses on the inter-procedural control flow represented by the Calling Context Tree, further research could also capture the intra-procedural control flow. We think that this offers additional support for program comprehension and optimization. Moreover, *Senseo* is an adequate framework to extend existing perspectives of IDEs or to enable the integration of further dynamic analysis such as memory leak and data race detectors. Thanks to our aspect-based gathering technique, the development and integration of such advanced features based on additional dynamic information can be completed in straightforward manner.

Improving Information Gathering. Currently the gathering of dynamic information is a continuous process: *MAJOR* subsequently transmits updated parts of the CCT to *Senseo*. This process can be improved in various ways such as integrating mechanisms to enable or disable the process of information gathering at runtime or by enabling developers to select within the IDE which parts of an application should be covered by instrumentation. Such selection could happen before starting the instrumentation or even at runtime. This would enable developers to limit data gathering to specific software features and source artifacts. Hence, developers could examine certain source artifacts in the context of a specific feature and no other dynamic information, *e.g.* from other features, would be displayed and therefore developers would not be misled by unrelated information. Additionally, it would be interesting to be able to selectively swap aspects at runtime to select and hence change the kind of data which is collected. This would for example enable developers to collect information about the number of created objects in a first execution, while in a second execution a developer would gather the number of invoked bytecodes.
Senseo currently requires the application to be started in an instrumented VM which does not provide access to the usual debugging mechanism often used to further inspect a running application. Hence, if a developer wants to use the debugger to halt the execution at certain breakpoints he has to switch to the common debugger where no dynamic information is collected. We think that a better integration of our instrumentation into the existing Java Run Configurations, combined with the feature to selectively turn on and off the data gathering would improve the overall usage of *Senseo*.

Data Processing. *Senseo* uses its own storage system to store and process the gathered information. We organize the gathered data on a per project basis hierarchically in packages, classes and methods. Such a storage system introduces an additional overhead to process and aggregate the gathered dynamic information. Furthermore, a dedicated storage system requires us to parse the entire CCT transmitted over the socket. Furthermore, we need to implement a way to serialize our data structure to persistently store the contained information on disk as well as implementing our own way to query the storage system. Using a back-end optimized for storing and querying information organized in graphs such as Neo4J[1] would ease the integration of features like persistency and would provide an optimized way to query such a storage system [36]. This would tighten *Senseo* more to a graph based representation of the gathered data. However, as the execution flow within a system can always be modeled as a graph, such a representation is apparent.

[1] `http://neo4j.org/`

Visualizations. *Senseo* forms a solid basis to enrich Eclipse with dynamic information. However, the implemented visualizations are very limited and offer various possibilities for further improvements. Besides the integration of more visualizations or more dynamic metrics, we see also a potential to improve existing visualizations:

- **Collaboration Overview** - Our validation revealed that the Collaboration Overview has been perceived as a very powerful tool. However, the visualization of the aggregated information is only textual, which impedes navigation. Especially a larger and widely used source artifact contains many collaborating parts and reveals therefore very quickly the limitation of such a textual representation. We think that an approach similar to the *CCRC* is more appropriate to visualize the aggregated data and to improve navigation in such a visualization. We imagine packages to be arranged on an plane connected by edges revealing the collaboration. Additionally, the thickness of these edges could be weighted based on how often a certain collaboration occurs. Developers could then click on a package to study the collaboration of the classes within this package. Furthermore, classes not contained by this package would reveal collaboration between packages. This approach could further go into methods and provide a graphical representation of the collaboration on a finer grained level.

- ***CCRC*** has been evaluated to be only helpful in specific use cases such as quickly spotting hotspots in the CCT. One of its issues is that the ring chart quickly grows huge and is therefore hard to navigate. Having data about more than one feature further overloads the ring chart and complicates its navigation. However, we see potential in a graphical representation of the CCT, as it can be very helpful to track execution flow and to detect hotspots in a larger system. We imagine that the visualization could be enriched with further information. Additionally, we imagine a better interaction with source code navigation such as automatic zooming towards the currently active method. Another problem is for example that a currently active method can be contained multiple times in the tree. We imagine that all occurrences of the active node could be better highlighted in the ring chart. For instance, the tree could be separated into several views, each zoomed to one of the occurrences of the active method in the tree.

Integration. We integrated dynamic information into different parts of the JDT such as the source code view or the package tree. Moreover, we augmented traditional static source code views such as tooltips with the aggregated information. However, as traditional IDEs already feature a broad range of different views on information gained through static source code analysis such as a type hierarchy or source code search results, we imagine further integration with these views. Augmenting these static perspectives with the same dynamic information we integrated in *Senseo* or, for instance, embedding which other classes are collaborating with the currently studied class the type hierarchy. Furthermore, we imagine search results to be annotated with dynamic information or enriching other tools such as CodeMap are with our aggregated data. Hence, *Senseo* would provide an interface to query the gathered dynamic information through a public interface.

Language Independence *Senseo* is currently tightened to the language Java and Eclipse. The discussed and addressed use cases, however, are not restricted to Java but are valid for any kind of object-oriented languages. Eclipse already supports a broad range of object-oriented languages that use the same kind of tools for static source code analysis. Hence Eclipse could provide a solid framework to work with different languages and support also the integration of dynamic information for other languages besides Java. As gathering dynamic information is decoupled from the other parts of *Senseo*, integrating support for other languages would be done by providing tools to gather dynamic information in these languages.

Appendix A

User’s Guide to *Senseo*

This chapter guides you through the installation of *Senseo* and its usage. Furthermore, we describe the different features and enrichments of *Senseo*.

A.1 Requirements

To be able to use *Senseo* with all its features a standard Eclipse installation in version *3.4.1* bundled with JDT is required. Furthermore, you preferably should have a current version of the Java SDK installed on your system. We propose that you unpack and install the version to be used in your Home-Directory in *~/java/current*.

Note to OS X users: Due to limitations in Apple’s bundling of the Java Virtual Machine, *Senseo* is not able to gather dynamic information with the standard Apple JVM. However, there are free (as in free software) implementations of the Java Virtual Machine available for Mac OS X such as the *Soyalatte JVM*. Soyalatte is known to work with *Senseo*. We recommend an installation in *~/java/current*.

A.2 Installation

Senseo can be installed from an Eclipse update site. You need to add the following URL as an Eclipse Update Site in the Eclipse Update Manager:

Update-Site: `http://scg.unibe.ch/download/senseo/eclipse/`

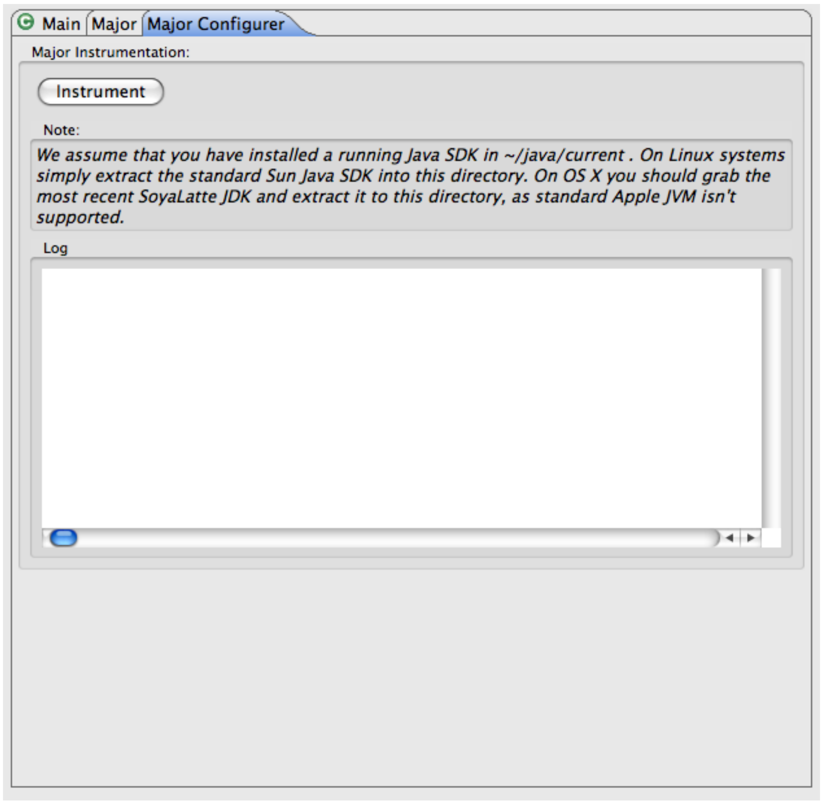
You are then able to install the *Senseo* plugin within your Eclipse installation. After successfully restarting Eclipse, *Senseo* is enabled and immediately available.

A.3 Gathering Data

To gather dynamic information you can create a new Run Configuration (Figure 5.1) in Eclipse for your current project similar as you would normally launch your application within Eclipse. You need to choose a *Senseo* Profiler Run Configuration and select the *Main*-class of your project, to specify the entry-point to your application. Afterwards, you can launch your application. First, *Senseo* prepares your application to be instrumented by *MAJOR* and then launches the packed JAR-file with *MAJOR*. To stop instrumentation you can simply terminate your application. Please note that you need to run an initial instrumentation for the first execution (see Paragraph Initial Instrumentation for further information).

Note: Currently we require your project to follow the convention that all necessary source code is located in a folder called *src/* within your project. We use this convention to generate a *JAR*-File to be passed to *MAJOR*.

Initial Instrumentation. *MAJOR* requires an initial instrumentation of the JDK with which your application is executed. This means that prior to your very first execution of an application in *Senseo* you need to open the *Senseo* Runtime Configuration and select the *Major Setup* tab (Figure A.1). In this view, clicking on the button *Instrument* starts the instrumentation of the JDK in your Home-Directory in *~/java/current*. This can take up to several minutes and on slower machines even up to half an hour. However, this initial instrumentation is only required once prior to the first usage of *Senseo*.



We assume that you have installed a running Java SDK in ~/java/current . On Linux systems simply extract the standard Sun Java SDK into this directory. On OS X you should grab the most recent SoyaLatte JDK and extract it to this directory, as standard Apple JVM isn't supported.

Figure A.1: Instrumentation of JDK

A.4 Visualizations / Enrichments

Senseo features different kinds of enrichments and visualizations to integrate the gathered dynamic information in Eclipse. The following sections explain each of these enrichments in detail and explain their functionalities.

A.4.1 Tree Metrics

In the various trees of Eclipse *Senseo* shows for each artifact (package, class, etc.) either a red, yellow, or purple icon to represent the metric value of a particular artifact. If the current selected metric is for instance *number of invocations*, then a red colored artifact means that the method has often been invoked. For packages or classes, the metric value is aggregated over all methods defined in a package or class. Figure 5.8 and Figure 5.9 give two examples how the different trees are enriched with the current selected metric. For instance, in the package tree we can distinguish between three metric values: high (red), average (yellow), and low (purple). To change the metric (*e.g.* from *number of invocations* to *number of created objects*), you can use the switch (depicted with (A) in Figure 5.5).

A.4.2 Ruler Annotations

Figure 5.7 shows an example of rulers enriched with colored annotations within the source code view. *Senseo* provides enrichments on both sides of the source code view displaying HeatMaps and annotations. The hotness of these enrichments are based on the currently selected metric.

Left Ruler Column Metrics. The left ruler column provides local information for the currently displayed methods of a class. It shows a color gradient from blue to red with six distinct color values. The color values have the same meaning as in the package tree (the more red, the more active is the current method for the current metric). The metric shown in the ruler column is also changeable with switch (A) in Figure 5.5. A white color in the column means that this method has not been invoked.

Right Overview Ruler Column Metrics. This overview ruler provides an overview of the entire source file, that is, of all classes and methods defined in that file, also methods currently not visible in the source code view. The colors encode the same meaning as in the package tree or in the left ruler column. In the overview ruler, *Senseo* just shows three distinct color values as in the package tree.

A.4.3 Tooltip Metrics

The tooltip, appearing when mouse-overing a method declaration (Figure 5.10) or invocation (Figure 5.11), shows more detailed information about how a particular method was used at runtime. For instance, the tooltip shows how often a method was invoked (*callers count*), how many objects it created, the size of all created objects (in bytes), or the number of methods this method invoked (callees). Furthermore, the tooltip contains a list of actual callers of this method, a list of actual callees, a list of argument types, and a list of return types. With *type* we refer to the class of the object that has been actually used as argument or return value, not the statically declared type. Note that tooltips are interactive. You can for instance click on a return type to navigate to that particular class. A tooltip encompassing dynamic information only appears if there is dynamic information available about a method or a method invocation. Otherwise, if a particular method has not been invoked or has not been analyzed by *Senseo*, we show the traditional (Javadoc) tooltip.

A.4.4 CCRC

The *CCRC* presents the call stack with the method in the center as the root (by default the *main* method). All direct callees of a method are arranged in a ring around that particular method, the indirect callees appear in the next ring, and so on. The colors used in the *CCRC* are the same as used in the left ruler column (gradient from red to blue). You can navigate in the *CCRC*, for instance zoom in or out, click on a method to select all occurrences of this method in the *CCRC* (all occurrences are colored in gray with a red border), double-click in a method to focus on this method, that is, turn it into the new center of the *CCRC*, or you can press *Ctrl* and click on a method to view its source. The *CCRC* is particularly useful to detect performance bottlenecks, for instance, methods that trigger the invocation of many nested methods (that is, a method which has many subsequent rings).

A.4.5 Collaboration Overview

The Collaboration Overview shows you either for a selected package, a selected class, or a selected method with which other packages, classes, or methods it collaborates. A collaborator is either a sender (the other artifact invokes methods in this package or class) or a callee (this package, class, or method invokes methods in this other package or class). In this view you can easily locate all dependencies of an artifact.

Appendix B

Subject Expertise Questionnaire

The following questions were used to categorize the subjects based on their experience to create two groups; one using *Senseo* and another one – the control group – having only standard Eclipse available. Each question is followed by the evaluated answers.

1. For how many years have you worked as a professional software developer in industry up to now?

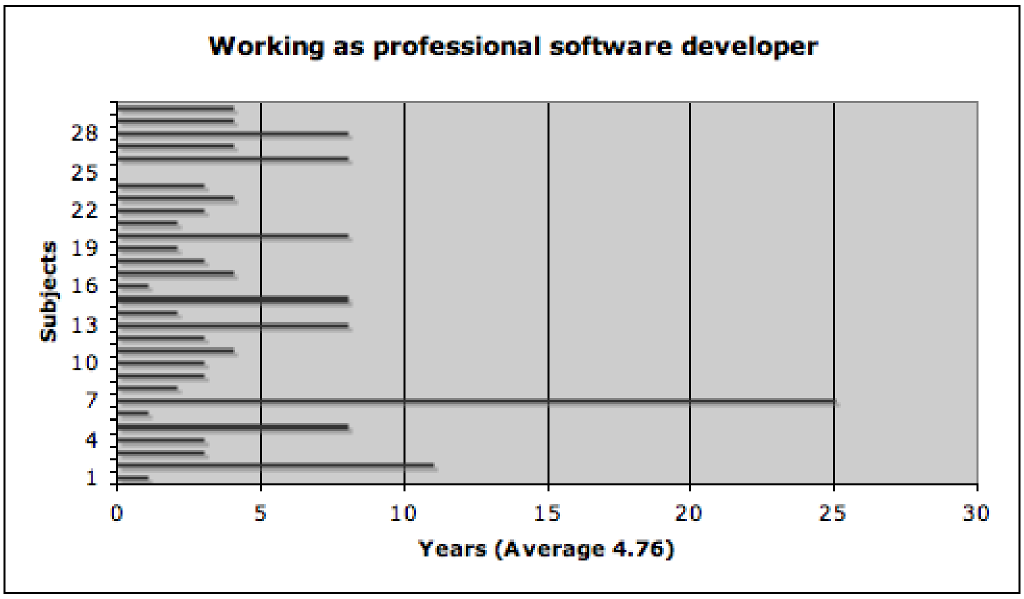


Figure B.1: Question 1: Working as professional software developer

2. For how many of those years have you mainly worked as Java developer?

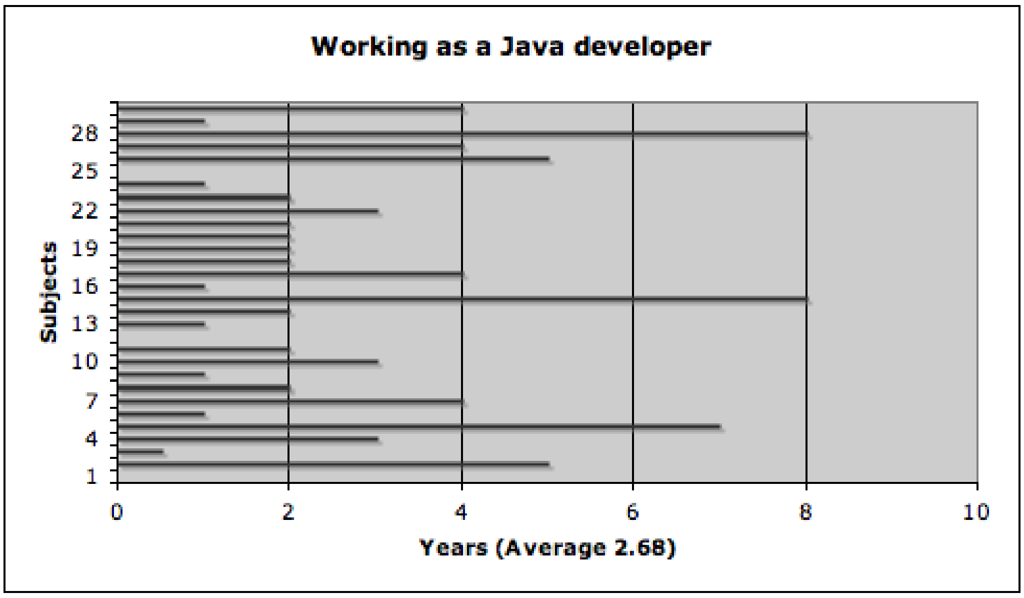


Figure B.2: Question 2: Working as a Java developer

3. How many years ago did you learn Java?

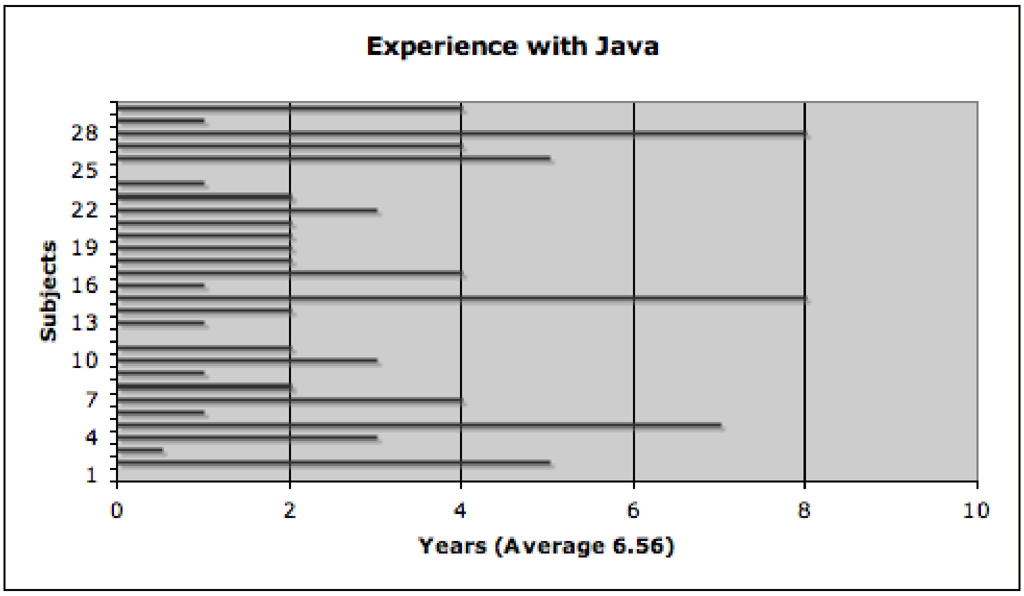


Figure B.3: Question 2: Experience with Java

4. Which degree or education have you completed?

1. Computer Scientist/Software Engineer or MSc/Licentiate University/ETH/Technical College **53.3%**

2. BSc University/ETH/Fachhochschule **40%**

3. Practical Education as Computer Scientist or Software Engineer **6.7%**
4. Other

5. Please rate your specific knowledge in the following areas on a scale from 0 (no experience or knowledge) to 4 (expert in this area).

	0 - No experience	1	2	3	4 - Expert
Java	3.3%	3.3%	16.7%	56.7%	20%
Eclipse	0%	3.3%	33.3%	50%	13.3%
Working with unknown code written by other people (reverse engineering, software analysis, program comprehension)	0%	10.3%	13.8%	69%	6.9%
Employing of dynamic analysis techniques and tools (Debugger, Profiler, Visualizations)	0%	23.3%	36.7%	36.7%	3.3%
Developing or maintaining open source projects (*e.g.* a SourceForge project)	30%	26.7%	23.3%	10%	10%

Table B.1: Experience in the different areas

6. Have you ever used the text editor *jEdit*?

- Yes: 50%
- No: 50%

7. Have you even looked at, maintained or re-used the source code of *jEdit*?

- Yes: 20%
- No: 80%

Appendix C

Experiment Tasks

In the following sections we include the task sheet we handed out to the subjects, followed by the detailed evaluation of the subjects. The first table (Table C.1) represents the time spent on each task per subject. The second table (Table C.2) shows their correctness per task.

C.1 Tasks

Thank you for participating in this experiment!
Please answer the following questions about the *jEdit* application. In total you have 90 minutes to answer all questions. Please refrain from answering questions in a rush, rather skip the last question(s) if there is not enough time left.
For each question we will record the time it takes you to answer it and we will also rate the correctness of your answer. At max you can achieve three points for each question if your answer is fully correct. Please do not go back to a previous question which you have already answered to not perturb the recorded answering time for this previous question.
The questions are concerned with typical maintenance tasks you would perform in an application you have not developed yourself and about which you have no or only limited knowledge. You are not asked to actually perform these maintenance tasks, but to just acquire the necessary knowledge and insights into the system to be able to perform these tasks. The system used in the experiment is *jEdit*, a fully-fledged text editor written in Java.
We ask you to work on five different tasks, each consists of two subtasks. These tasks are ordered in a top-down approach, which means that you first gather general knowledge about *jEdit* and later go down on a more detailed level to study the inner structure and behavior of *jEdit*.

Before beginning with the tasks, we provide you with a short written description and manual for the tools provided to you (*Senseo* and its features, tools). You can use this description as a reference during the experiment. No other tools are permitted during the experiment, in particular no manuals or other documentation about *jEdit*.
At the end of the experiment we kindly ask you to answer a short questionnaire asking

you for your experience with the experiment and the tools used therein.
The experiment and the allocated 90 minutes start when we ask you to turn this page, that is, when you tell us that you are ready to start. ;)

Task 1 Question 1.1
In *jEdit* you find the feature *indent selected lines* in the menu at *Edit ->Indent ->Indent selected lines*. Please study where and how this feature is implemented in the code on a coarse-grained level: Which packages, classes implement the feature? In which part or layer of the application is this feature mostly implemented? In the UI, model, text processing, or somewhere else?

Question 1.2
Compare the package in which the *indent selected lines* feature is mainly implemented with the package org.gjt.sp.jedi.guit in terms of collaboration between them. Do these packages collaborate with each other and if so, how and where (which classes communicate)?

Task 2 We want to assess the quality of *jEdit*, for instance whether its classes are organized in the right packages or whether we should move classes to other packages. For this we need to analyze how certain classes are coupled to each other and how they communicate to the rest of the system. If a class of package C heavily communicates with classes of package B, it should maybe moved to package B. Or if a class is heavily used by package B (high fan-in from B) but has low fan-out to other packages, it should probably also belong to package B. Fan-in of a class is defined as the number of methods or constructors of this class that are invoked from other classes. Fan-out of a class is the number of methods or constructors of other classes this class is invoking. In the following we ask you to compare the quality of different artifacts of *jEdit*.

Question 2.1
Compare the quality of class org.gjt.sp.jedit.buffer.FoldHandler, org.gjt.sp.jedit.msg.ViewUpdate and org.gjt.sp.jedit.Mode to each other in terms of fan-in, fan-out. Please give a list saying which class has the highest quality if we consider low fan-in and low fan-out as a sign for tight cohesion, that is, good inner quality. Which class is second, which third?

Question 2.2
Which classes of package org.gjt.sp.jedit.msg are heavily coupled to package org.gjt.sp.jedit? Are there classes in org.gjt.sp.jedit.msg heavily coupled to other packages than org.gjt.sp.jedit?

Task 3 Question 3.1
Analyze class org.gjt.sp.jedit.buffer.FoldHandler. What objects (*i.e.* instances of which classes) send messages to instances of FoldHandler (callers)? To which objects (that is, instances of which classes) do instances of FoldHandler send messages (callees)?

Question 3.2
What kind of methods or constructors of FoldHandler and its subclasses are executed in which order by clients (maybe there is common entry point used by all clients and

they subsequently invoke the same methods in the FoldHolder hierarchy in the same order)? Which are the invocation sites (classes and methods) of these methods and constructors?

Task 4 We look again at the *indent selected lines* (Edit ->Indent ->Indent selected lines) feature. This feature works on selected lines of text, but after executing (*e.g.* indenting two lines of text), the selection is often lost, no text is selected anymore sometimes. We want to correct this problem by keeping the selection after having indented a text.

Question 4.1
Compare this feature to the implementation of other related features such as *remove trailing whitespaces* or *shift indent left*. What is implemented differently in these features than in *indent selected lines*?

Question 4.2
We really want *indent selected lines* to keep the selected text. How can you achieve this? What will you change in the code (which methods to adapt, what statements to add, change or remove)? We do not expect you to provide us with working code, just suggestions about what kind of statements you would add, change or remove to make it work.

Task 5 Feature *remove whitespace* (*Edit ->Indent ->Remove trailing whitespaces*) has been identified by users to be very slow; the lead developer of *jEdit* thinks class org.gjt.sp.jedit.indent.WhitespaceRule is the bottleneck. Can you confirm this assumption? Answer the following two questions to decide whether we should optimize this class.

Question 5.1
What are the methods and the control structures in class WhitespaceRule that get executed the most? Are there repeated patterns of code that gets executed when this rule is applied to a document?

Question 5.2
Compare class WhitespaceRule to other classes in the same hierarchy of Rules (classes implementing IndentRule). Is it really used more often or more heavily (*e.g.* does it create more objects or execute more complex code) than the other classes? Explain why or why not.

C.2 Time Results

The following table shows the amount of time spent (in minutes) of each subject per task.

Subject	*Senseo*	1.1	1.2	2.1	2.2	3.1	3.2	4.1	4.2	5.1	5.2	Total
Subject 1	1	17	4	7	8	5	13	7	4	12	4	65
Subject 2	0	17	8	5	12	7	12	4	8	16	7	73
Subject 3	1	10	18	14	11	11	12	14	9	10	9	99
Subject 4	1	11	13	12	5	8	7	12	10			78
Subject 5	0	18	19	9	12	6	12	19	4	10	5	99
Subject 6	0	13	6	12	13	12	15	9	13	15	13	93
Subject 7	1	23	13	12	9	12	16	12	5	8	9	102
Subject 8	0	29	11	25	11	21	9	20	21			147
Subject 9	0	27	18	12	10	6	16	25	18			132
Subject 10	1	8	5	16	13	12	22	13	3	19	8	92
Subject 11	0	15	16	14	21	8	15	18	12	16	18	119
Subject 12	0	12	13	20	13	8	16	10	22			114
Subject 13	1	19	17	10	10	10	10	11	4			91
Subject 14	0	22	20	19	9	10	12	16	21			129
Subject 15	1	29	5	10	9	4	12	11	10	8	4	90
Subject 16	0	27	8	7	8	24	6	19	18	4	4	117
Subject 17	1	26	17	9	22	9	8	8	1	3	3	100
Subject 18	0	7	13	23	9	10	13	20	17	9	10	112
Subject 19	1	18	7	14	10	8	17	9	4	18	6	87
Subject 20	1	21	18	11	9	13	14	6	8	13	6	100
Subject 21	1	22	18	15	12	8	9	12	11	4	5	107
Subject 22	0	16	9	15	7	11	14	20	4	7	5	96
Subject 23	1	19	12	12	12	11	6	15	15			102
Subject 24	0	16	24	17	7	7	19	13	18	5	5	121
Subject 25	1	25	19	8	11	12	11	12	19	4	4	117
Subject 26	0	15	16	25	11	10	11	7	2	14	5	97
Subject 27	0	27	27	6	9	21	8	21	29	18	11	148
Subject 28	1	19	17	14	14	3	15	16	3	10	11	101
Subject 29	1	19	16	8	13	5	12	8	9	13	2	90
Subject 30	0	14	28	16	11	17	10	27	2	8	6	125

Table C.1: Time per task per subject

C.3 Correctness Results

The following table shows the correctness of each subject per task.

Subject	*Senseo*	1.1	1.2	2.1	2.2	3.1	3.2	4.1	4.2	5.1	5.2	Total
Subject 1	1	1	2	2	2	4	2	1	1	2	2	15
Subject 2	0	1	2	2	1	2	2	1	2	1	1	13
Subject 3	1	2	3	3	2	2	2	1	1	0	0	16
Subject 4	1	3	2	4	3	2	2	2	1	0	0	19
Subject 5	0	1	0	2	2	1	1	1	1	2	1	9
Subject 6	0	2	2	2	2	2	2	1	1	0	0	14
Subject 7	1	2	2	3	2	1	2	0	0	3	2	12
Subject 8	0	2	1	3	1	1	1	0	0	0	0	9
Subject 9	0	0	0	2	1	2	1	0	0	0	0	6
Subject 10	1	2	2	3	2	2	2	2	2	0	0	17
Subject 11	0	2	0	2	2	2	1	1	1	0	0	11
Subject 12	0	2	1	2	2	2	2	0	0	0	0	11
Subject 13	1	2	1	2	2	2	2	0	0	0	0	11
Subject 14	0	2	2	3	2	2	2	0	0	0	0	13
Subject 15	1	2	2	3	2	3	2	0	0	2	2	14
Subject 16	0	3	2	2	1	2	2	1	0	1	1	13
Subject 17	1	3	2	2	2	3	1	1	1	1	1	15
Subject 18	0	1	1	2	1	2	2	1	1	1	1	11
Subject 19	1	2	2	3	1	2	3	2	1	2	3	16
Subject 20	1	3	1	3	2	2	2	2	2	2	0	17
Subject 21	1	2	1	3	2	2	1	1	1	1	1	13
Subject 22	0	2	0	2	1	2	2	1	1	1	2	11
Subject 23	1	3	2	2	3	1	2	2	1	0	0	16
Subject 24	0	2	0	3	1	2	2	1	0	1	1	11
Subject 25	1	2	2	2	2	2	2	1	1	1	1	14
Subject 26	0	1	0	3	2	2	2	1	1	1	1	12
Subject 27	0	2	2	3	2	2	2	3	1	0	0	17
Subject 28	1	2	2	3	2	2	1	1	2	1	2	15
Subject 29	1	2	3	2	1	2	2	2	3	1	2	17
Subject 30	0	1	1	3	1	2	0	1	0	0	0	9

Table C.2: Correctness per task per subject

Appendix D

Feedback Questionnaire

To finish the experiment each subject was asked to answer the following questionnaire. Each question is followed by the evaluated answers. The first two questions were answered by subjects of both groups, while the remaining questions were only relevant for subjects of the *Senseo* group.

1. How did you feel about the time pressure?

Subjects selected on of the following statements, which we rated from 5 (high time pressure) down to 1 (too much time):

- Time pressure was very high; impossible to cope with all tasks.
- Serious time pressure, but could cope with most task.
- Felt no time pressure
- Could have done slightly more task in the time provided
- There was too much time, could have performed many more tasks in these 90 minutes

Results for question 1:

- Arithmetic mean: 2.80 (n = 30)
- Standard aberration: 0.42 (n = 30)
- Arithmetic mean *Senseo* group: 2.67 (n = 15)
- Arithmetic mean control group: 2.93 (n = 15)

2. Please rate the difficulty of the different tasks:

Subjects rated the tasks on a range from 5 (impossible) to trivial (1)

	Arithmetic mean (n = 30)	STD (n = 30)	Arithmetic mean *Senseo* group (n = 15)	Arithmetic mean control group (n=15)
Task 1.1	3.2	0.69	3.0	3.4
Task 1.2	2.8	0.41	2.7	2.9
Task 2.1	3.6	0.25	3.1	4.1
Task 2.2	2.4	0.31	2.5	2.3
Task 3.1	3.2	0.56	3.2	3.2
Task 3.2	3.5	0.41	3.1	3.9
Task 4.1	3.1	0.27	3.0	3.2
Task 4.2	3.2	0.19	2.9	3.5
Task 5.1	2.8	0.86	2.7	2.9
Task 5.2	2.9	0.72	2.9	2.9
Overall:	3.07		2.9	3.2

Table D.1: Question 2 results

The following questions are only answered by subjects from the Senseo *group.*

3. Please specify for each task which dynamic information provided by *Senseo* you used.

Table D.2: Question 3 results

	Run-time types (Tooltip)	Number of invocations	Number of created objects	Number of exec. bytecodes	*CCRC*	Dynamic Collaborators
Task 1	33%	53%	33%	27%	7%	53%
Task 2	47%	67%	47%	33%	7%	80%
Task 3	47%	40%	27%	20%	0%	73%
Task 4	20%	27%	13%	7%	0%	33%
Task 5	27%	40%	27%	13%	7%	27%

(Results in the table, n = 15)

4. How useful do you rate dynamic information respectively features provided by *Senseo* in a scale from 4 (very useful) to 0 (not useful at all)?

Dynamic information / *Senseo* feature	Arithmetic Mean (n = 15):	STD (n = 15)
Tooltip showing runtime types	3.6	0.45
Ruler column with dynamic metrics	3.2	0.38
Overview ruler column with dynamic metrics	3	0.41
Package tree with dynamic metrics	2.4	0.29
CCRC	2.1	0.94
Collaboration View	3.7	0.11

Table D.3: Question 4 results

5. Concerning the overall performance of *Senseo*, do you think it provided you with added value in terms of:

	Yes (n = 15)	No
High-level understanding (*e.g.* package-level collaboration)	87%	13%
Low-level understanding (method structure, collaboration, control flow, etc.)	73%	27%
Understanding dependencies between different artifacts	80%	20%
Feature localization, feature understanding (identifying and understanding artifacts implementing a specific feature)	67%	33%
General program understanding	73%	27%
Performance assessment or optimization	53%	47%
Software quality assessment	53%	47%

Table D.4: Question 5 results

6. After using *Senseo*, do you think it provides you with added value for your daily work on maintaining or developing software systems?

- 87%
- 13%

Bibliography

[1] Glenn Ammons, Thomas Ball, and James 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] Erik Arisholm, Hans Gallis, Tore Dyba, and Dag I.K. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[3] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[4] Thomas Ball. The concept of dynamic analysis. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC'99)*, number 1687 in LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.

[5] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.

[6] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009. `http://dx.doi.org/10.1002/spe.890`.

[7] 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, October 2006. ACM Press.

[8] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[9] Brian F. Cooper, Han B. Lee, and Benjamin 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, April 1998.

[10] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart 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.

[11] Brian de Alwis and Gail C. Murphy. Answering conceptual queries with ferret. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 21–30, New York, NY, USA, 2008. ACM.

[12] Marcus Denker, Orla Greevy, and Michele Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.

[13] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[14] Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid source code views. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 260–263, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Mikhail Dmitriev. Design of JFluid: a profiling technology and tool based on dynamic bytecode instrumentation. Technical report, Mountain View, CA, USA, 2003.

[16] Mikhail 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. ACM Press, 2004.

[17] Mikhail Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, 2004.

[18] Stéphane Ducasse and Michele Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.

[19] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

[20] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. *SIGPLAN Not.*, 38(11):149–168, 2003.

[21] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, November 2003.

[22] Bruno Dufour, Laurie Hendren, and Clark 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.

[23] Alastair Dunsmore, Marc Roper, and Murray 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.

[24] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings of ICSM '01 (International Conference on Software Maintenance)*. IEEE Computer Society Press, 2001.

[25] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-Driven Program Understanding using Concept Analysis of Execution Traces. In *Proceedings of IWPC '01 (9th International Workshop on Program Comprehension)*, pages 300–309. IEEE Computer Society Press, 2001.

[26] David Erni. Codemap—improving the mental model of software developers through cartographic visualization. Master's thesis, University of Bern, January 2010.

[27] Tudor Gîrba and Michele Lanza. Visualizing and characterizing the evolution of class hierarchies. In *WOOR 2004 (5th ECOOP Workshop on Object-Oriented Reengineering)*, 2004.

[28] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.

[29] Dean Jerding, John Stasko, and Thomas Ball. Visualizing message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, May 1996.

[30] Mik Kersten and Gail 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, New York, NY, USA, 2005. ACM Press.

[31] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[32] Adrian Kuhn, David Erni, and Oscar Nierstrasz. Towards improving the mental model of software developers through cartographic visualization, 2010. Under submission to NIER track of ICSE 2010.

[33] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–218, Los Alamitos CA, October 2008. IEEE Computer Society Press.

[34] Michele Lanza and Stéphane Ducasse. The class blueprint: A visualization of the internal structure of classes. In *Workshop Proceedings of OOPSLA 2001*, 2001.

[35] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin 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.

[36] Patrik Larsson. Analyzing and adapting graph algorithms for large persistent graphs, 2008.

[37] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24(2):197–218, 1994.

[38] Adrian Lienhard. *Dynamic Object Flow Analysis*. Phd thesis, University of Bern, December 2008.

[39] Sam P. LLoyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

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

[41] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[42] Philippe Moret, Walter Binder, Danilo Ansaloni, and Alex 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.

[43] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[44] Michael Pacione, Marc Roper, and Murray 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, November 2004.

[45] Massimiliano Di Penta, R. E. K. Stirewalt, and Eileen 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.

[46] John Pleviak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of OOPSLA '94*, pages 324–340, 1994.

[47] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[48] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[49] Jochen 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.

[50] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. Dynamic type inference to support object-oriented reengineering in Smalltalk, 1998. Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS).

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

[52] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pages 317–326, 2008.

[53] Martin P. Robillard and Gail C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234, October 2003.

[54] Martin P. Robillard and Gail C. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, pages 822–823, May 2003.

[55] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):3, 2007.

[56] David Röthlisberger. Hermion — exploiting the dynamics of software. European Smalltalk User Group Innovation Technology Award, August 2008.

[57] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Feature driven browsing. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 79–100. ACM Digital Library, 2007.

[58] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[59] David Röthlisberger, Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet, and Romain Robbes. Supporting task-oriented navigation in IDEs with configurable HeatMaps. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC 2009)*, pages 253–257, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[60] A. Schwind. Visualisierung von Strukturen in Softwaresystemen. Diploma's thesis, Fakultaet für Informatik, Universität Karlsruhe, 2000.

[61] Janice Singer, Robert Elves, and Margaret-Anne 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.

[62] S. Alexander Spoon and Olin Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[63] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://www.webcitation.org/5qDBdghWu`, 2000.

[64] Alex Villazón, Walter Binder, Philippe Moret, and Danilo Ansaloni. Major: rapid tool development with aspect-oriented programming. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 125–128, New York, NY, USA, 2009. ACM.

[65] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings ECOOP '01*, volume 2072 of *LNCS*, pages 99–118, Budapest, Hungary, June 2001. Springer-Verlag.

[66] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[67] Roel Wuyts. RoelTyper, a fast type reconstructor for Smalltalk, 2005. `http://www.webcitation.org/5qDBnw6kE`.