Exploiting Dynamic Information in IDEs Eases Software Maintenance

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

Abstract

The integrated development environment (IDE) is the primary tool used by developers to maintain software systems. The IDE, however, narrowly focuses on the static structure of a system, neglecting dynamic behavior and dynamic relationships between static source artifacts such as classes and methods. Developers often have difficulties to understand the dynamic aspects of a system just based on the static source perspectives provided by IDEs. Existing IDE tools to analyze the running of software systems such as debuggers or profilers present volatile dynamic information from specific system executions, requiring developers to manually trigger debugging or profiling sessions. To better support the understanding and maintenance of software systems, we developed several extensions to traditional IDEs to incorporate dynamic information in the static source perspectives. In this paper we describe these extensions and report on the empirical experiments we conducted to evaluate the practical usefulness of these IDE extensions.

Keywords: dynamic analysis, development environments, software maintenance, program comprehension

1 Introduction

Object-oriented language features such as inheritance, abstract types, late-binding, or polymorphism lead to distributed and scattered code, rendering a software system hard to understand and maintain when just looking at its static source artifacts such as classes or methods [1, 3, 13]. By means of dynamic analysis developers can explore the dynamic aspects of software systems and hence better understand the implementation of these systems [2, 13]. Even though the concept of dynamic analysis has been widely studied [2, 3, 8, 12, 13], traditional integrated development environments (IDEs), the primary tools used by developers to maintain software systems, still purely operate on static source code and do not reveal dynamic relationships between distributed source artifacts, which makes it difficult for developers to understand and navigate software systems in these IDEs [10, 11]. Due to the lack of dynamic information in IDEs developers are forced to build up a mental model of a system's dynamic behavior based on a static view of the system. Such a mental model, however, is error-prone as it is very difficult to understand a large object-oriented system without having available information about its running [1, 3].

In this paper we propose to enhance the Eclipse IDE [4] for Java with dynamic information in order to help developers during program comprehension and software maintenance activities. To achieve the goal of supporting developers to more efficiently maintain object-oriented code, we augment the static source perspectives of IDEs with dynamic information. We contribute several different techniques to integrate dynamic information seamlessly into the IDE. The implemented techniques are:

- *Source code enrichments to embed dynamic information such as runtime types of variables or callers of methods directly in an IDE's source code view*
- *Presenting dynamic metrics such as number of method invocations or number of objects created in a method next to the source code perspectives*
- *Collaboration view, a navigable view showing the dynamic collaborators of source artifacts, for instance all callers and callees of a method*

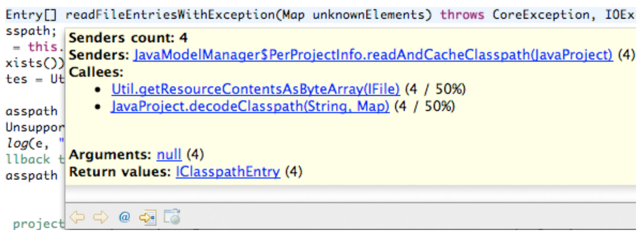
We validate these techniques by means of empirical experiments with professional developers to assess the practical usefulness of the availability of dynamic information in the traditional perspectives of IDEs.

In the following we first describe these techniques in detail and subsequently report on how we evaluated their practical usefulness. The paper is structured as follows: In Section 2 we present the enrichments to the IDE's source code view to present dynamic information, Section 3 reports on the integration of various dynamic metrics, and Section 4 explores the collaboration view we integrated into the IDE. In Section 5 we describe the conducted empirical experiment to evaluate the implemented techniques embedding dynamic information in IDEs. Section 6 concludes the paper and outlines ideas for further work.

2 Source Code Enrichments

All techniques to present dynamic information have been implemented for the Eclipse IDE. We opted for this IDE since it is the most widely adopted IDE in industry [5], at least for developing Java software. The dynamic information that is exploited by these techniques typically stems from different system executions. We aggregate dynamic information over multiple executions to obtain a more complete picture of the dynamics of a system compared to debuggers or profilers that just focus on a specific execution. The aggregation of dynamic information in the IDE supports developers in understanding the general execution patterns of a system whereas the traditional debugger can be applied to study the execution flow of a specific scenario.

As a technique to complement source code with dynamic information without impeding its readability we opted to use *hovers*, small windows that pop up when the mouse hovers over a source element (a method name, a variable, etc.). Hovers are interactive, which means the developer can for instance open the class of a receiver type by clicking on it. We now describe the integration of dynamic information into the source code:



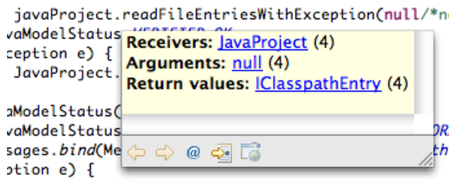
```
Entry[] readFileEntriesWithException(Map unknownElements) throws CoreException, IOEx
```

Senders count: 4
Senders: JavaModelManager$PerProjectInfo.readAndCacheClasspath(JavaProject) (4)
Callees:

- Util.getResourceContentsAsByteArray(IFile) (4 / 50%)
- JavaProject.decodeClasspath(String, Map) (4 / 50%)

Arguments: null (4)
Return values: IClasspathEntry (4)

Figure 1. Hover appearing for a method name in its declaration.



```
javaProject.readFileEntriesWithException(null/*n
```

Receivers: JavaProject (4)
Arguments: null (4)
Return values: IClasspathEntry (4)

Figure 2. Hover for a message send occurring in a method.

Method header. The hover that appears on mouse over the method name in a method header shows (i) all senders invoking that particular method, (ii) all callees, that is, all methods invoked by this method, and optionally (iii) all argument and return value types. For each piece of information we also show how often a particular invocation occurred. For instance for a sender, we display the qualified name of the method containing the send (that is, the calling method) and the number of invocations from this sender. 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 1 shows a concrete method name hover for method `readFileEntriesWithException`.

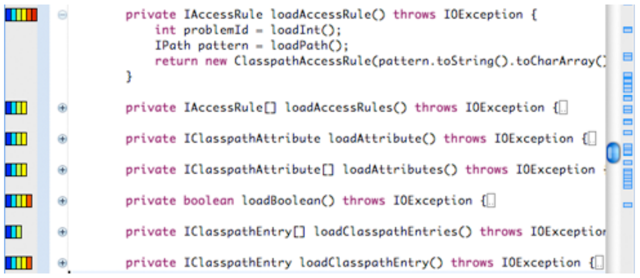
In a method header, we can optionally show information about argument and return types, if developers have chosen to gather such data. Hovers presenting this information appear when the mouse is over the declared arguments of a method or the defined return type. These hovers also include numbers about how often specific argument and return value types occurred at runtime.

Method body. We also augment source elements in the method body with hovers. For each message send defined in the method, we provide the dynamic callee information similarly as for the method name, optionally along with argument or return types that occurred in this method for that particular message send at runtime, as shown in Figure 2. Of course all these types listed are always accompanied with the number of occurrences and the relative frequency of the specific types at runtime.

3 Dynamic Metrics

There are two kind of rulers next to the source editor: (i) the standard ruler on the left showing local information and (ii) the overview ruler on the right giving an overview over 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. 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. A completely filled bar for a method denotes methods that have been invoked the most in this application. The dynamic metrics in these two rulers allow developers to quickly identify hot spots in their code, that is, methods being invoked frequently. 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 3. Rulers left and right of the editor view showing dynamic metrics.

To associate the continuous distribution of metric values to a discrete scale with for instance three representations (*e.g.*, red, yellow, and blue), we use the *k-means* clustering algorithm [7].

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

Furthermore, developers can choose between different dynamic metrics to be visualized in the rulers. Besides the number of invocations of methods, we also provide metrics such as the number of objects a method creates, the number of bytecodes it executes, and the amount of memory it allocates, either on average or in total over all executions. Such metrics allow developers to quickly assess the runtime complexity of specific methods and thus to locate candidate methods for optimization. Changing the dynamic metrics to be displayed is done in the Eclipse preferences; the chosen metric is immediately displayed in the rulers.

The following dynamic metrics are collected at runtime and can be visualized in the ruler columns:

Number of invocations. This dynamic metric helps the developer quickly identify hot spots in code, that is, 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 a particular method.

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 in 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.

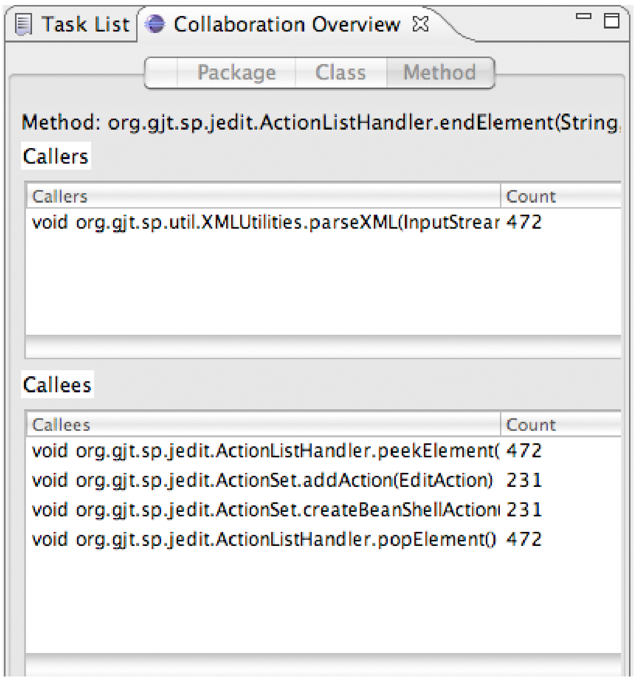
Allocated memory. Different objects vary in memory size. Having many but very tiny objects might not be an issue, whereas creating a few but very huge objects could be a sign of an efficiency problem. Hence, we also provide a dynamic metric recording memory usage of various source artifacts such as classes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

Number of executed bytecode instructions. The static source code does not disclose how many bytecode instructions have to be executed during the runtime of the code. Calling a particular method in a piece of code might trigger the execution of very complex code, that is, many bytecode instructions. To assess the complexity of the execution of a piece of code we also gather the number of bytecode instructions executed when invoking a particular method. This metric is also an estimator for the execution time of a particular method invocation.

4 Collaboration View

In a separate view next to the source code editor of Eclipse (Figure 4), we present all dynamic collaborators for the currently selected artifact. For instance, if a method has been selected, the collaboration view shows the collaborators at the package, class, or method level; that is, it lists all packages or classes invoking methods of the package or class in which the selected method is declared (callers). The collaboration view also shows all packages or classes with which the package or class declaring the method is actively communicating (callees). For the method itself, the collaboration view lists all direct callers and callees.

This collaboration view allows developers to navigate the callers and calleers. If for instance a caller of a method is selected, the view is refreshed to show all callers of the selected caller, and so on. Like this, developers can easily navigate through all dynamic callers of source artifacts of interest.



Task List | Collaboration Overview

Package | Class | Method

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

Callers

Callers	Count
void org.gjt.sp.util.XMLUtilities.parseXML(InputStrear	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.createBeanShellActioni	231
void org.gjt.sp.jedit.ActionListHandler.popElement()	472

Figure 4. A view of all collaborators of the selected artifact (package, class, or method).

5 Validation

We conducted a controlled experiment with 30 professional Java developers to evaluate the benefits for software maintenance that arise from the three presented means to integrate dynamic information into IDEs.

Experimental Procedure. We asked the experiment subjects to solve five typical software maintenance tasks and analyzed the time spent to solve these tasks and the correctness of the solutions. Each subject was either assigned to the control group or to the experimental group. The experimental group had available dynamic information integrated with the aforementioned techniques while the control group used a standard Eclipse IDE. With an expertise questionnaire we collected information about a developer's expertise concerning software development, software analysis, Java, or Eclipse. Based on the results of this questionnaire we assigned developers to either the experimental or the control group to obtain two groups of nearly equal expertise. As all subjects were unfamiliar with the tools integrating dynamic information into Eclipse we gave the subjects of the experimental group a 30 minutes introduction to the extensions to Eclipse.

As a subject system we had 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. The tasks we gave the subjects are concerned with analyzing and gaining an understanding for various features of jEdit. We selected tasks representative for real maintenance scenarios and not being biased towards dynamic analysis by following the framework of Pacione *et al.* [9]. For solving the tasks, subjects had to provide an answer in free text, they could not select from multiple choices.

The dynamic information shown in Eclipse to the subjects of the experimental group was obtained by executing all actions from the menu bar of jEdit to make sure that this pre-recorded information is not biased towards the experiment tasks. 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.

The two dependent variables studied in this experiment, *time* the subjects spent to answer the questions, and *correctness* of the answers, were manually determined by the experimenters. The time spent on a task is the time span between the starting time of one task and the next. Correctness is measured using a score from 0 to 4 according to the overlap with the model answers, which forms a set of expected answer elements that have been identified by the experimenters beforehand.

To determine whether dynamic information has a statistically significant effect on the variables *time spent* and *correctness*, we applied the parametric, one-tailed Student's t-test at a confidence level of 95% (α=0.05).

Results. The results of the experiment were promising. On average, the experimental group spent significantly less time solving the maintenance tasks, we could measure a 17.5% decrease in time spent on the tasks for the experimental group compared to the control group. With the Student's t-test we verified whether the availability of dynamic information in the IDE had an impact on the time to solve the maintenance tasks. The p-value resulting from the t-test is with 0.0016 considerably lower than α=0.05, which means that the time spent is statistically significantly reduced by the availability of dynamic information.

For the correctness variable we could discover a 33.5% increase for the subjects having available dynamic information compared to those using a traditional Eclipse IDE. Applying the t-test to the correctness variables shows that this increase is statistically significant: the t-test gives a p-value of 0.0001 which is clearly below α=0.05. This means that having available dynamic information during software maintenance activities helps developers to more correctly solve maintenance tasks.

Qualitative Feedback. In a debriefing questionnaire the subjects provided qualitative feedback about the usefulness of dynamic information integrated into the IDE. On a Likert scale from 0 (useless) to 4 (very useful), the subjects

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

rated the various techniques to embed dynamic information in the IDE. The source code enrichments obtained an average rating of 3.6, the dynamic metrics embedded in the ruler columns were rated with 3.2, and the collaboration view got an average rating of 3.7. These ratings clearly show that subjects considered all techniques to be useful for solving the software maintenance tasks of this experiment. In particular the collaboration view but also the source code enrichments have been an important aid to quicker and more accurately complete the imposed tasks.

Subjects also gave feedback about how often they used what kind of dynamic information during the experiment. It turned out that information about dynamic collaborators has been used the most, nearly by all subjects in all tasks, while dynamic information shown in the source code views, *e.g.*, information about runtime types has been used less often. Most subjects have just occasionally used dynamic metrics, for instance for tasks where they had to assess performance aspects of the application.

These results from the controlled experiment make us confident that the implemented techniques to embed dynamic information seamlessly and tightly in the IDE are indeed helpful for developers to more efficiently and more correctly solve typical software maintenance tasks. A more detailed report on this experiment is available in the Masters thesis of Marcel Haerry [6].

6 Conclusions

In this paper we described three different techniques we implemented to seamlessly integrate dynamic information into the Eclipse IDE to help developers to easier and more accurately understand the dynamic behavior of Java systems. These three techniques are (i) source code enrichments embedding dynamic information such as runtime types, callers or callees of a method; (ii) dynamic metrics such as number of created objects, memory size of these objects, or number of executed bytecode instructions; and (iii) a collaboration view presenting the dynamic collaborators, that is, the callers and callees of particular source artifacts such as packages, classes, or methods. We evaluated the practical usefulness of these three contributed techniques by means of a controlled experiment with 30 professional software developers solving typical software maintenance tasks in a large unfamiliar application (*i.e.*, jEdit). The results of this experiment show that developers can 17.5% faster and 33.5% more correctly solve the maintenance tasks when Eclipse is enhanced with the aforementioned dynamic information compared to a standard Eclipse installation.

In the future we aim at extending the integration of dynamic information into Eclipse, for instance by embedding visualizations of the collaboration between source artifacts to ease the navigation and understanding of collaboration patterns. Furthermore, we plan to apply the proposed techniques to large industrial systems such as rich client platform systems in the finance sector to gather more empirical feedback from practice.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010)

References

[1] S. Demeyer, S. Ducasse, K. Mens, A. Trifu, and R. Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering. In *Object-Oriented Technology (ECOOP'03 Workshop Reader)*, LNCS, pages 72–85. Springer-Verlag, 2003.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, New York NY, 2000. ACM Press.

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

[4] Eclipse platform: Technical overview, 2003.

[5] G. Goth. Beware the march of this IDE: Eclipse is overshadowing other tool technologies. *IEEE Software*, 22(4):108–111, 2005.

[6] M. Haerry. Augmenting eclipse with dynamic information. Master's thesis, University of Bern, May 2010.

[7] S. P. LLoyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

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

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

[10] D. Röthlisberger, O. Greevy, and O. 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.

[11] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 253–262, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[12] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Working Conference on Reverse Engineering (WCRE99)*, pages 304–313, Oct. 1999.

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