View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Bern Open Repository and Information System (BORIS)

Embedding Moose Facilities Directly in IDEs*

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

Abstract

Moose is a powerful reverse engineering platform, but its facilities and means to analyze software are separated from the tools developers typically use to develop and maintain their software systems: development environments such as Eclipse, VisualWorks, or Squeak. In practice, this requires developers to work with two distinct environments, one to actually develop the software, and another one (e.g., Moose) to analyze it. We worked on several different techniques, using both dynamic and static analyzes to provide software analysis capabilities to developers directly in the IDE. The immediate availability of analysis tools in an IDE significantly increases the likelihood that developers integrate software analysis in their daily work, as we discovered by conducting user studies with developers. Finally, we identified several important aspect of integrating software analysis in IDEs that need to be addressed in the future to increase the adoption of these techniques by developers.

Keywords: software analysis, dynamic analysis, static analysis, development environments, visualizations

1 Introduction

When performing maintenance or software enhancements, developers typically navigate a system's source code in a development environment (IDE) to gain an understanding of how the system functions. Software analysis tools such as Moose provide advanced analysis facilities such as visualizations or metrics to provide developers with concise information about the static structure or the runtime behavior of the subject software system. With these facilities, developers can often much more efficiently solve maintenance tasks or implement feature enhancements to the system. Having available such means is in particular important for dynamic object-oriented languages making heavily use of inheritance, polymorphism, and late binding, concepts that make the static source code hard to understand without additional aids by visualizations or dynamic analyzes.

Unfortunately, the powerful facilities of Moose are separated from the development environment and embedded in a completely different environment with different navigation and browsing metaphors. This separation impedes the adoption of analysis techniques in practice as their usage requires developers to spend extraneous time and effort to gain an understanding for the underlying concepts behind Moose or any other reverse engineering and software analysis environment. In addition, using two separate environments in parallel, *i.e.*, the IDE to develop and maintain the software system, and the re-engineering platform to analyze it, might often be too much of a hassle for a developer working under time pressure.

In this paper, we present an integration of software analysis techniques such as visualizations, dynamic analysis, or feature analysis in the IDE. We integrated similar techniques as provided by Moose or CodeCrawler, the most important difference is their direct and immediate availability in the IDE, even for techniques relying on dynamic information about the software system. Lanza [8] or Kuhn [7] also propose the integration of visualizations into IDEs (*e.g.*, RBCrawler is an extension for Cincom Smalltalk [12]), however the integrated visualizations typically do not take into account dynamic information about the subject system. We have chosen Squeak Smalltalk [6] for the integration of these techniques, as this IDE is easily extensible. Most techniques could also be integrated into other IDEs such as Eclipse as they are not Smalltalk-specific in any way.

Concretely, we embedded three different analysis techniques in the Squeak IDE: (i) traditional visualizations focusing on static information such as class blueprints or system complexity views, (ii) dynamic analysis techniques to explicitly represent package or class collaborations, and (iii) explicit feature representations to reason about features directly in the IDE. In the remaining of this paper we elaborate on these three analysis techniques integrated in the IDE. In Section 2 we describe how they function and how they are implemented. These techniques have been used by various developers. We asked these developers to assess the integrated techniques and report about their feedback in Section 3. Our work on integrating software analysis facili-

* *FAMOOSr (Workshop on FAMIX and Moose in Reengineering)*, 2008

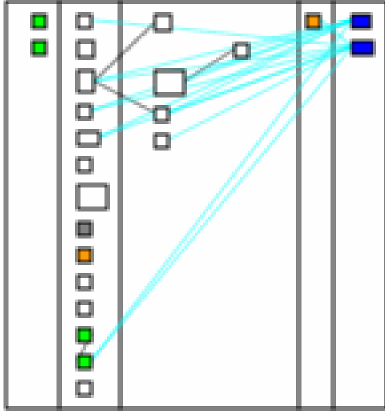

Figure 1. Class blueprint embedded in the Squeak IDE.

ties into IDE is still in its infancy, we thus also give insights in planned future work towards a more complete integration of software analyses in IDEs in Section 3.

2 Integration of Software Analyzes in IDEs

In this section we briefly introduce three software analysis techniques we integrated in the Squeak IDE: (i) traditional visualizations, (ii) visualizations of dynamic information to reason about runtime collaborations, and (iii) feature visualizations to explicitly represent features in the IDE. All these techniques are tightly embedded in the IDE and fully interactive, *e.g.*, developers can use it to navigate forth and back from these enhancements to the traditional IDE tools such as source code views.

2.1 Traditional Visualizations

We extended the perspectives on static source artifacts (*e.g.*, classes) in the IDE with appropriate visualization, similar as RBCrawler [7] or inCode [5]. When the developer has selected a class, she can now also generate a class blueprint (see Figure 1), a system complexity view focusing on the class hierarchy of this particular class, or a UML class diagram directly from within the IDE instead of just looking at *e.g.*, source code. These visualizations are thus readily available along the source code view. Typically, a visualization is interactive in the sense as the developer can click on nodes to jump to *e.g.*, the class represented by that node, or context menus available for nodes or edges in a visualization allow the developer to trigger for instances searches for references to classes, for implementors of message sends, or for readers of instance variables.

We also provide visualizations for packages or for single methods, *e.g.*, system complexity views focusing on classes within a package. Many visualizations do encompass dynamic information to provide more insights to the developer into the dynamics of a system. There is for instance a dynamic class blueprint available which highlights all communication paths that have indeed been used by the system at runtime. The next section present other techniques to represent and visualize information solicited with dynamic analysis.

2.2 Dynamic Information Visualizations

Several IDEs, for instance Eclipse, already provide plugins and tools (*e.g.*, inCode [5]) that allow the developer to visualize the static structure of a system, *e.g.*, by means of UML class diagrams. Various tools separated from IDEs visualize dynamic behavior of software systems, *e.g.*, Jinsight [1]. But to the best of our knowledge, there is almost no work done to also reason about software dynamics directly in the IDE by means of visualizations. The only tool available in the IDE providing insights in a system's behavior is typically the debugger which is focusing on a single execution, thus not providing broader insights. However, developers often want to generally reason about interactions between specific static entities to for instance reveal how a class communicates to other classes. To study dynamic interaction we provide a range of *collaboration charts*. Similar collaboration charts exist for classes and methods.

Our charts show a compact representation of package, class or method runtime intercommunication. For classes, the chart is similar to a UML sequence diagram, although the order of calls is not preserved, To avoid cluttering the chart with too much information, we show communication paths between classes, *i.e.*, message sends occurring in an instance of a class with an instance of another class as a receiver, as edges in the chart, the thickness of the edge represents the relative frequency of this interaction, similar as in the work of Ducasse *et al.* [3].

These charts are always dedicated to a specific run of the subject system that has to be triggered by the developer, either by running scripts to exercise behavior or by manually interacting with the subject application. As soon as dynamic data is available, the developer can generate the charts by navigating to static artifacts (*e.g.*, classes) and triggering the appropriate action, *e.g.*, *open class collaboration chart*.

2.3 Feature Visualization

To explicitly represent features, *i.e.*, behavioral entities of a software system, we enhanced the Squeak IDE with a tool called *feature browser* to visualize and navigate features. We meta-model features with an enhancement to Moose called *Dynamix* that allows us to model dynamic be-

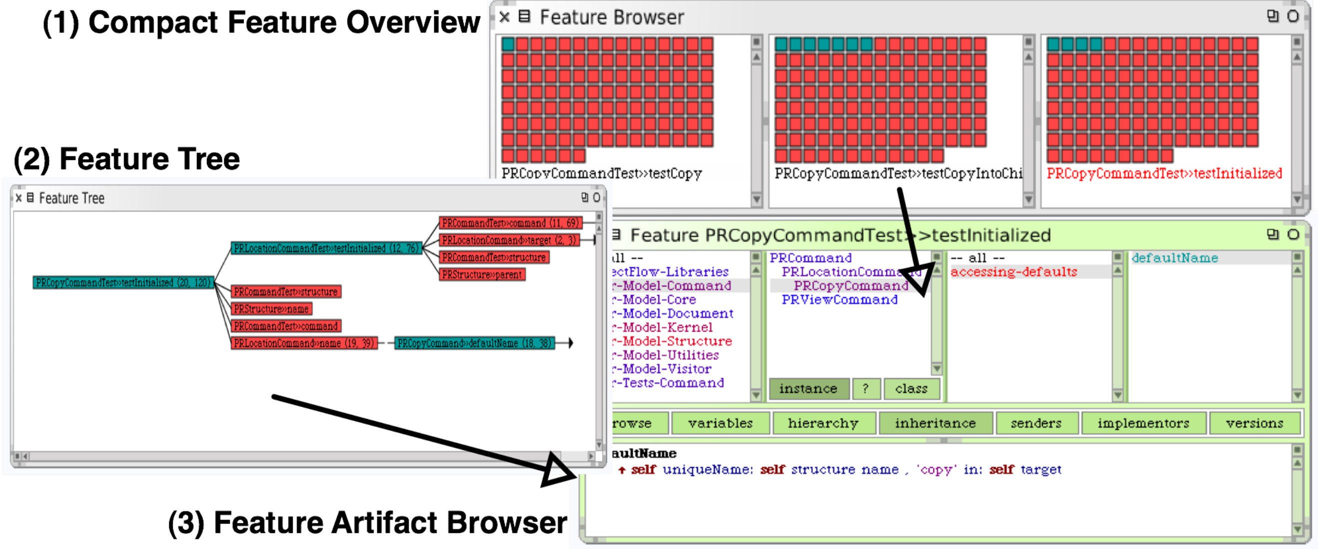

Figure 2. Schema of the Feature Browser.

havior as described by Greevy [4]. We refer to a feature as a specific execution of a software system triggered by its end user, *e.g.*, editing a page in a Wiki application. Our enhanced IDE gathers automatically the information about this feature's execution and generates a feature view. Developers trigger the generation of such a feature view by tools integrated in the IDE, such as buttons to start and stop data gathering or to launch the feature view. Such a view consists of three parts that are depicted in Figure 2.

The *Compact Feature Overview* (1) enables the developer to visually compare several features at once. The small nodes in this feature view represent methods, colored according to the feature affinity metric proposed by Greevy [4]. Entities used in only one feature (colored in blue) can be quickly distinguished from entities used in several or even in all features (colored in orange or red). Such a coloring scheme allows the developer to immediately grasp similarities between different features or anomalies in a specific feature and locate erroneous behavior.

The *Feature Tree* (2) provides the developer a more detailed view on a feature by representing the method call tree triggered while it was exercised. The root of the tree is the first, *e.g.*, the "main" method of the feature, child nodes are methods being invoked by this main method. All nodes in this tree are colored according to the feature affinity metric.

The *Feature Artifact Browser* (3) shows all entities used in a particular feature in a dedicated source browsing environment. This browser only shows entities (*e.g.*, packages, classes, or methods) which are actually used in the selected feature. Thus the developer can focus on the classes and methods that are responsible for the feature's behavior.

2.4 Technical Realization

We briefly describe how we gather dynamic information to generate collaboration charts and feature views. As we want to permanently gather dynamic information to provide the developer with accurate and immediate information directly in the IDE, we rely on a technique called *partial behavioral reflection* (PBR) [11]. PBR, available for Java [11] and Smalltalk [2], provides very fine-grained means to select parts of the application about which to gather dynamic information. To for instance generate a dynamic class blueprint we only need to reason about message sends occurring within a particular class, *i.e.*, which methods invokes which other methods of the same class. PBR allows us to just gather this particular information, hence limiting the overhead typically encompassing dynamic analysis. We developed extensions to the IDE enabling the developer to choose the parts of a software's behavior to reason about to very effectively gather data just about these entities (*e.g.*, classes) or operations (*e.g.*, message sends) using PBR.

3 Validation and Future Work

To validate our work we mostly discussed with developers familiar to the Squeak IDE and asked them to assess our integration of reverse engineering tools into the IDE by answering questionnaires. For the validation of the feature browser we even conducted a full-fledged empirical study, we refer to [10] for details. In the questionnaires, we particular asked developers to evaluate the impact of visualizations such as blueprints, system complexity view, or col-

Statement	Av. rating
Impact of static visualizations (system complexity view UMLs, blueprint) on program comprehension	3.5
Impact of dynamic visualizations on program understanding	3.2
General effect of software analysis in IDE on program understanding	3.6
Usefulness of software analysis available directly in IDE	4.3

Figure 3. Answers obtained from our questionnaires

laboration charts on program comprehension, *i.e.*, whether they could more efficiently or more correctly understand a subject system. We also asked them how often they used these visualizations and in what scenarios, and whether they consider integration of software analysis techniques in the IDE as worthwhile. An extract of the most important results from the user feedback is given in Table 3. For the rating schema we used a Likert scale from '1' to '5' where '1' means 'strongly disagree' and '5' means 'strongly agree'.

Although we consider this developer feedback as preliminary (we want to assess the quality of our work by comprehensive empirical studies), we still obtained an interesting and motivating assessment by consulting developers. Several developers also gave us qualitative feedback, such as feature requests. For instance, they want an overview of the overall dynamic communication occurring in a system, *e.g.*, a map of a software's dynamics. A drawback of the current solution is the requirement to specify what is covered by dynamic analysis (*e.g.*, entities or operations) and the need to manually run system features to exercise behavior. They wish the IDE to provide dynamic information in an ubiquitous manner, as it provides views on the static part. Another area for future work is to integrate more features of Moose into the IDE, not just visualizations, but also metrics assessing both, the statics and dynamics of software systems. In particular, we are interested in metrics highlighting crucial dynamic properties such as memory consumption. We also plan to not only add our enhancements to the Squeak IDE, but also to mainstream IDEs such as Eclipse. In particular visualizations reasoning about system's dynamics is also interesting in the context of Eclipse.

4 Conclusions

In this paper we argued for a tight and seamless integration of software analysis in IDEs by reporting on the status of this integration into the Squeak IDE. We ported several analysis techniques of Moose such as visualizations (class blueprints, system complexity view) and integrated them in Squeak or developed new means, in particular to reason about software dynamics, such as collaboration charts or feature views (based on the Dynamix meta-model available in Moose). We validated our work by asking developers and provided an outlook on future work to extend and enhance the integration of software analysis in IDEs.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

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

[2] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. In *Proceedings of TOOLS Europe 2007*, volume 6, pages 231–251. ETH, Oct. 2007.

[3] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[4] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, May 2007.

[5] inCode — eclipse plugin for code analysis. http://www.intooitus.com/inCode.html.

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

[7] A. Kuhn. Rbcrawler — a visual navigation system for Smalltalk's Refactoring Browser. European Smalltalk User Group Innovation Technology Award, Aug. 2007.

[8] M. Lanza. Program visualization support for highly iterative development environments. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 62–67. IEEE, 2003.

[9] B. A. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. Answering why and why not questions in user interfaces. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 397–406, New York, NY, USA, 2006. ACM Press.

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

[11] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[12] Cincom Smalltalk, Sept. 2003. http://www.cincom.com/scripts/smalltalk.dll/.