Towards Seamless and Ubiquitous Availability of Dynamic Information in IDEs[*]

David Röthlisberger and Orla Greevy
Software Composition Group, University of Bern, Switzerland
{roethlis, greevy}@iam.unibe.ch

Abstract

Software developers faced with unfamiliar object-oriented code need to build a mental model of the system to understand its dynamic flow. Development environments typically provide static views of the source code (e.g., classes and methods), but do not explicitly represent dynamic collaborations. The task of revealing how static source artifacts interact at runtime is thus challenging.To address this we have developed several techniques to represent dynamic behavior at various levels of granularity directly in the IDE. In this paper we outline these various techniques towards a seamless integration of dynamic information in the IDE. We elaborate on user feedback we have gathered and on our empirical experiments to validate our work. We derive several ideas and visions of further potential representations of dynamic behavior from this analysis of our approach. The missing representations we identify serve to enrich our proposed IDE, so as to provide the developer from within the IDE with a readily available and complete picture of a software's dynamics.

Keywords: dynamic analysis, dynamic collaborations, development environments, program comprehension

1 Introduction

Maintaining or enhancing object-oriented software systems requires developers not only to understand static source artifacts, but also their dynamic interaction. The primary tool available to developers, the integrated development environment (IDE), typically focuses on a static view of a system. It does not explicitly represent dynamic collaboration between static artifacts (*e.g.*, classes or methods). In the absence of IDE support developers are forced to build up a mental model of a system's dynamic behavior. Integrating explicit representations of dynamic behavior directly in the IDE would prove helpful in gaining a more accurate understanding for a system under investigation.

[*]In: *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, 2008

To achieve the goal of representing dynamic behavior seamlessly in the IDE, we are faced with several challenges, such as:

- *How can we efficiently gather dynamic information and immediately make it available from within the IDE?*
- *How do we represent dynamic behavior of a system in an IDE?*
- *How do we validate that our proposed representations are useful for developers?*

Our key focus is to present our experience to date and to identify our visions for further IDE enhancements to exploit seamless integration of dynamic information in various forms, providing developers with relevant information to understand a software's dynamics.

In this paper we report on the techniques we devised to address the above challenges. In Section 2 we present an overview of our techniques to represent dynamic behavior explicitly in the IDE, such as (i) visualizations, (ii) enrichments to the source code view, or (iii) techniques to query dynamic information from within the IDE. We present a summary of developer feedback and results of our evaluations in Section 3. Based on this, we have identified representations of dynamic behavior to support developers In Section 4, we outline ideas for further enhancements to an IDE encompassing dynamic information.

2 Existing Approaches Integrating Dynamic Analysis in IDEs

In our work to date, we have developed four different approaches to reason about dynamic information directly in the IDE. Each approach works on different levels of granularity, from the fine-grained source code level, the dynamic interaction of static artifacts to a coarse-grained representation of user-identifiable features of a system. Our techniques provide the developer with several entry points for gaining an understanding of software system, *e.g.*, to correct a specific defect. If a defect occurs in a specific feature, the developer may first gain an overview of the feature's dynamics, then locate candidate entities (*e.g.*, methods) that may contain the defect. In a next step, the developer reasons about the specific communication patterns between the candidate entities and finally drills down to the source code level to study the dynamics on a fine-grained level to pinpoint and correct the defect.

We provide a brief overview of our four proposals to represent dynamic information in the IDE. We implemented our IDE enhancements in the Squeak Smalltalk IDE [7] as it provides an extensible framework to adapt and extend its tools, We take advantage of our previously implemented a technique, partial behavioral reflection, to efficiently and selectively gather runtime information [3].. Applying our enhancements to other IDEs, *e.g.*, Eclipse could be achieved using similar techniques.

2.1 Feature Representation

To explicitly represent features, *i.e.*, behavioral entities of a software system, we introduce our Feature Browser [10], an enhancement to a traditional IDE.

We describe our Feature Browser taking as an example a Wiki application. The developer first specifies within the IDE that dynamic data should be recorded for the application (*i.e.*, at the package level) and then associates names with the features (*i.e.*, external user-understandable units of behavior of the application) under investigation, *e.g.*, "wikiEditPage". Then she exercises a feature in the application. The IDE takes care of gathering and storing dynamic data of the feature. The IDE now provides the developer with an explicit feature representation of behavioral data for "wikiEditPage". To study the features of interest, the developer selects them either in our Feature browser or invokes an action we added next to the class and method browser to open all features that use a particular class or method.

Figure 1 depicts our feature browser's core components. The *Compact Feature Overview* (1) enables visually comparison of several features . The small nodes in a feature view can represent either methods or classes and are colored according to the feature affinity metric proposed by Greevy [5]. Entities used in only one feature (colored blue) can be distinguished from entities used in several or all features (colored orange or red). The coloring scheme makes it easier to quickly grasp similarities between features, anomalies or to 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. To make this tree navigable for reasonable sized execution traces we applied several compression techniques such as subexpression removal [8] or sequence and repetition removal as proposed by Hamou-Lhadj [6]. A developer can open this feature tree by clicking on a node (*i.e.*, a method) in the compact feature overview or by selecting a feature in which a method opened in the IDE participates.

The *Feature Artifact Browser* (3) shows all entities used in a particular feature in a dedicated source browsing environment. Only entities (*e.g.*, packages, classes, or methods) which are actually used in the selected feature are shown so the developer can focus on parts of the code responsible for the feature's behavior.

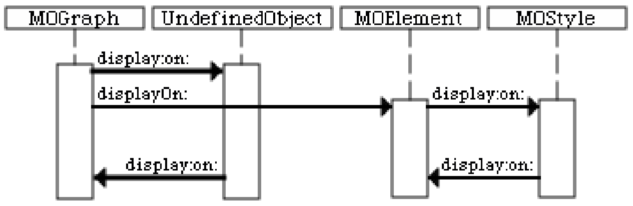


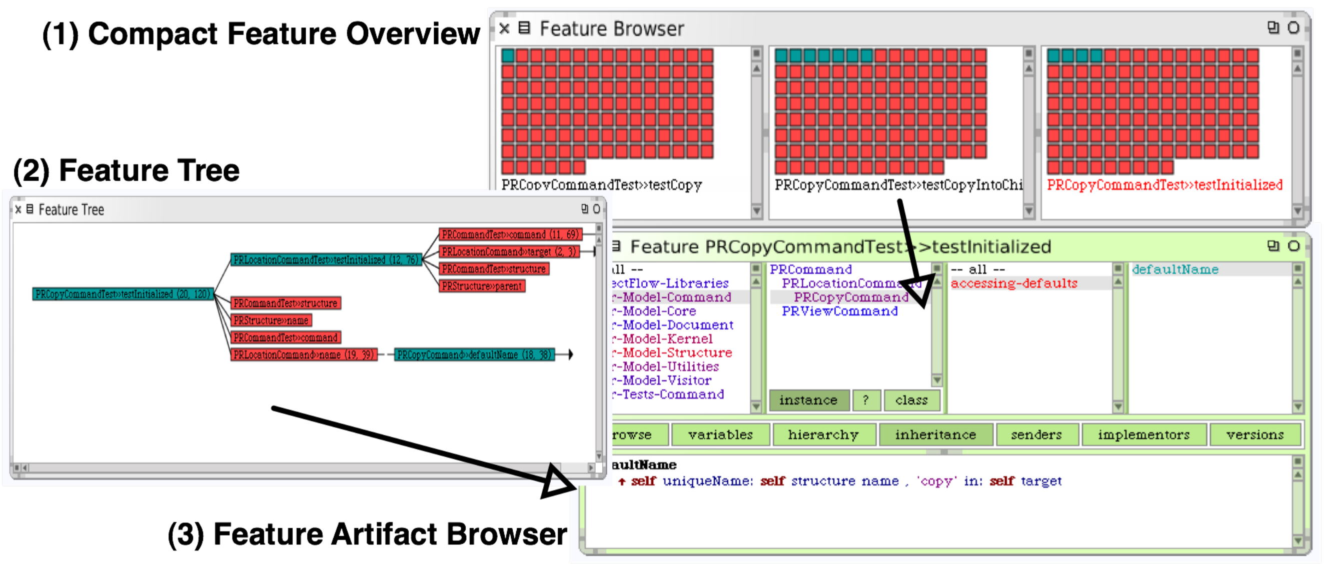
Figure 2. Class collaboration chart for class Graph.

2.2 Representing Dynamic Collaboration

To refine their mental model of a feature's behavior, developers typically want to reason about more fine-grained interactions to reveal how classes communicates with each other. Studying this kind of dynamic interaction may uncover unwanted behavior, such as incorrect or missing communication between instances. To study this level of interaction we provide a range of *collaboration charts*. A class collaboration chart of the class *Graph* of a visualization tool is shown in Figure 2. Similar charts exist for packages or methods.

Our charts show compact representations of package, class or method runtime communications. Our class collaboration 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 an edge reflects the relative frequency of the interaction, as in the work of Ducasse *et al.* [4].

Our charts are directly accessible either from within the feature browser, or from the static view on source code of the IDE, *e.g.*, by selecting a particular class and opening a class collaboration chart for this class. In the latter case, the application has to be executed before the class collaboration chart can be shown. The charts are always dedicated


(1) Compact Feature Overview
Feature Browser
PRCopyCommandTest>>testCopy
PRCopyCommandTest>>testCopyIntoChi
PRCopyCommandTest>>testInitialized
(2) Feature Tree
Feature Tree
PRCopyCommandTest>>testInitialized (20, 120)
PRLocationCommandTest>>testInitialized (12, 76)
PRCommandTest>>command (11, 69)
PRLocationCommand>>target (2, 3)
PRCommandTest>>structure
PRStructure>>parent
PRCommandTest>>structure
PRStructure>>name
PRCommandTest>>command
PRLocationCommand>>name (19, 39)
PRCopyCommand>>defaultName (18, 38)
Feature PRCopyCommandTest>>testInitialized
PRCommand
PRLocationCommand
PRCopyCommand
PRViewCommand
-- all --
accessing-defaults
defaultName
instance
?
class
variables
hierarchy
inheritance
senders
implementors
versions
(3) Feature Artifact Browser

```
aultName
  self uniqueName: self structure name , 'copy' in: self target
```

Figure 1. Schema of the Feature Browser.

to a specific run of the subject system triggered by the developer, either by running scripts to exercise behavior or by manually interacting with the subject application.

2.3 Dynamic Data Querying

Dynamic analysis approaches need to deal with vast amounts of data [1]. In the two previous approaches, we addressed this by focusing on one particular execution of a system and by compressing the resulting execution trace. However, developers often want to understand the dynamic behavior of the application "in general", *i.e.*, for as many different executions as possible, although full coverage is of course not achievable for any reasonable big system [1]. For this reason, we keep the data generated by observed entities in a central database accessible from within the IDE [9].

To effectively reason about the permanently stored dynamic data, we extended the IDE's search capabilities consider both static and dynamic information. Our extended search enables the developer for instance to search for senders of messages to a specific receiver type, *e.g.*, only for methods invoking the *size* method of class *Graph*. The query to solve this problem is shown in the code section below (query 1).

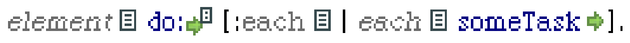
```
SHOW senders OF Graph.size
SHOW collaborators OF Graph
SHOW method invocation IN Wiki ORDER BY
    frequency
```

The query language syntax is similar to SQL. Query 2 returns all classes collaborating with *Graph* at runtime, while query 3 provides a list of all methods being invoked in the package *Graph*, ordered by invocation frequency. Other search facilities are dynamic implementors, package or method collaborators, or method execution times. The results of such queries are directly embedded in the IDE and can be browsed using IDE functionalities.

2.4 Dynamic Information Integrated in Source Code

On the lowest level of granularity, we embed dynamic information directly into the source code of methods [11]. When reading source code of dynamically-types languages such as Smalltalk, it may be difficult to completely understand the code as there is no type information. Polymorphism further complicates the task. It is unclear which methods are invoked at runtime and what kind of objects are stored in variables. We enrich the source code view to feed in information obtained by dynamic analysis. The code statement in Figure 3 highlights our enhancements to the source view. We add icons to message sends and variables accesses in source code. Clicking on an icon either reveals what methods were executed for a message send or show all type of objects a variable stored at runtime. Of course the developer can directly navigate to a method or a variable type shown in the respective list by clicking on the item. An interesting side-effect of these enhancements is that they also reveal which parts of a method have never been executed, as these parts will be missing these icons for dynamic information.

To obtain the dynamic data for these extensions we query



```
element do: [:each | each someTask ].
```

Figure 3. Dynamic information embedded in source code.

the database mentioned in Section 2.3. We apply caching strategies: as soon as a method's source code has been displayed once, including dynamic information icons. We cache results of various queries submitted for this method until either the method's source code has changed or more dynamic information has been gathered.

3 Validation of Existing Techniques

We validate our various approaches to integrate dynamic information in the IDE from a user perspective. In previous works we also validate it from an efficiency and performance perspective [10, 11].

3.1 User Validation

We validated out *Feature Browser* (Section 2.1) by means of an empirical study involving twelve developers familiar with both the Smalltalk language and IDE. We asked the subjects to correct two defects of similar complexity in a Wiki system. For one defect the developers used the traditional Squeak IDE, for the other we provided them with our IDE-embedded feature browser. The order in which they used each environment was randomly assigned. We then compared both, the efficiency (*i.e.*, time spent) to correctly locate the cause of the defect in source code and to actually correct the defect entirely. The performance of the subjects was in average 30% better with the feature browser concerning defect location and 10% better concerning defect correction. Both figures are statistically significant. For more details of this study we refer the reader to our previous work [10].

We validated the other techniques, collaboration charts, dynamic information querying, and enriched source code view, by means of providing a questionnaire to several developers and asked them to apply our techniques in a controlled experiment we defined. We also involved the subjects of the feature browser. For all techniques, we used set of general questions as well as specific questions for each technique. Every questionnaire was answered by at least three subjects. We used the same Wiki application (*i.e.*, Pier) for each experiment as none of the subjects had prior knowledge of this system. We assigned the subjects specific tasks to solve, providing them with just one of our four techniques. The tasks were for instance to describe the role of an key model class, to enhance the system with a feature

Statement	Av. rating
Impact of feature browsing in program comprehension	4.2
Impact of collaboration charts on program understanding	3.2
Effect of collaboration charts on execution overview	4.3
Impact of querying dynamic inf. on prog. understanding	3.4
Impact of querying dynamic inf. on navigation of static artifacts	3.8
Effect of source code enrichments on execution overview	4.0
Effect of source code enrichments on navigation of static artifacts	3.9
Impact of source code enrichments on program comprehension	3.3

Figure 4. Answers obtained from our questionnaires

similar to an already existing feature, or to adapt a feature without impacting any other system behavior. After solving three tasks we gave the subjects the questionnaire. Table 4 provides a selection of answers from the questionnaires.

We obtained many suggestions, ideas, or wishes for future enhancements to represent dynamic information in the IDE, this feedback incorporates in Section 4.

4 Competing the Representation of Software Dynamics in IDEs

We elaborate on several opportunities to extend our existing work on integrating dynamic information in the IDE. We identify shortcomings, problems, or issues in the current work and present ideas and suggestions obtained from developers that participated in our experiments.

Identifying Missing Features. A shortcoming of the current solution is the requirement to select specific static artifacts of the subject system (*e.g.*, packages or classes) to collect dynamic data, and to then run one or many system's features. The IDE should *automatically* take care of gathering dynamic information from all system entities. Dynamic data should be as readily available as static information (*e.g.*, list of methods or instance variables of a class). Moreover, developers want to be able to associate a particular execution with the dynamic data it generated, but for other scenarios they also want to access all gathered information about an artifact in order to achieve a *high level of coverage*. If the IDE were to automatically collect dynamic information, developers would be freed from this responsibility and would be more likely to incorporate views on system's dynamics in their daily work, in particular when these views show reliable, complete and accurate information.

To gather dynamic data, a system first needs to be executed. Instead of relying on the developer to run the application manually or with scripts, the IDE could *continuously run the system* in the background, in particular after changes to the system's code base. The developer could record some scripts on a high level (*e.g.*, by recording user actions in the application) that could be fed into the IDE so it could run the system. The IDE could easily determine code (*e.g.*, methods or source code statements) that have never been executed and either try to find execution paths for this code or alert the developer to refine the provided scripts. This procedure would improve code coverage. The general, idea is to empower the IDE and to relieve the developer of the responsibility to ensure that as many parts of the system as possible are covered by dynamic analysis. The IDE is the appropriate tool to assume this responsibility as it is very familiar to developers (they spend most of their working time in this environment), and as it already provides sophisticated means to work with static code. We understand dynamic views as being orthogonal to static views and hence nicely completing IDE's mostly static perspective.

Developer Suggestions. One suggestion of developers was to use dynamic information not only to enhance and complete the static perspective of a system, but to build means and concepts to browse, develop and maintain software in a environment that *primarily display entities by their dynamic relationships*, *e.g.*, a browser that shows classes on a two-dimensional map showing communication as paths while the distance between any two classes represents how heavily they communicate with each other. Entities are placed closer to each other the more they collaborate. Another developer mentioned the importance of having *full coverage*, *i.e.*, he often wants to know whether two entities will ever communicate to each other in any possible system execution. Of similar importance is a *big picture view*: While focusing on a particular feature or execution is interesting in many scenarios, there is often also a need to get an overview of all possible dynamic communication occurring in an application, *e.g.*, to present to a new developer how the system generally functions at runtime.

5 Conclusions

In this paper we described four different techniques to seamlessly integrate dynamic information in IDEs to reason about software's dynamics. These four techniques are (1) a feature browser to reason about features, (2) collaboration charts to visualize dynamic communication between static artifacts in the IDE, (3) facilities to query dynamic information, and (4) enrichments to source code to embed information of its dynamic behavior. We performed several user experiments to evaluate these techniques and to solicit feedback from developers about ideas for future enhancements. We presented both the results from the various studies (*e.g.*, results of questionnaires) and a comprehensive list of issues in the current approach. Finally we identified further opportunities for extend and complete the representation of software's dynamics 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] T. Ball. The concept of dynamic analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS, pages 216–234, Heidelberg, 1999. Springer Verlag.

[2] M. Denker and S. Ducasse. Software evolution from the field: an experience report from the Squeak maintainers. In *Proceedings of the ERCIM Working Group on Software Evolution (2006)*, volume 166 of *Electronic Notes in Theoretical Computer Science*, pages 81–91. Elsevier, Jan. 2007.

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

[4] 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.

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

[6] A. Hamou-Lhadj and T. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proceedings of 1st International Workshop on Dynamic Analysis (WODA)*, May 2003.

[7] 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.

[8] J.-M. S. Philippe Flajolet, Paolo Sipala. Analytic variations on the common subexpression problem. In *Automata, Languages, and Programming*, volume 443 of *LNCS*, pages 220–234. Springer Verlag, 1990.

[9] D. Röthlisberger. Querying runtime information in the ide. In *Proceedings of the 2008 workshop on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008. To appear.

[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] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting runtime information in the ide. In *Proceedings of the 2008 International Conference on Program Comprehension (ICPC 2008)*, 2008. To appear.