Querying Runtime Information in the IDE

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

Abstract

Code queries focus mainly on the static structure of a system. To comprehend the dynamic behavior of a system however, a software engineer needs to be able to reason about the dynamics of this system, for instance by querying a database of dynamic information. Such a querying mechanism should be directly available in the IDE where the developers implements, navigates and reasons about the software system. We propose (i) concepts to gather dynamic information, (ii) the means to query this information, and (iii) tools and techniques to integrate querying of dynamic information in the IDE, including the presentation of results generated by queries.

Keywords: querying techniques, dynamic analysis, development environments, partial behavioral reflection, program comprehension

1 Introduction

To maintain a software system, developers need to gain an understanding of this system. Querying mechanisms help developers to ask questions about how systems are implemented and structured and about how different entities (*e.g.*, classes) are related to each other. However, these querying tools are often limited to the static structure of a system, (*e.g.*, methods, classes or static relationships between them), but do not take into account dynamic behavior of a system (*e.g.*, message passing, dynamic dependencies between objects) [6].

Object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand an application by purely asking questions about the static structure of a system [1, 4, 10]. It is even more difficult to gain an understanding of programs implemented in dynamically-typed languages such as Smalltalk or Ruby, as there is no explicit type information encoded in the source code. In these situations it is crucial to be able to reason about the dynamics of a program by means of querying techniques. Other researches have already recognized the value of combining static and dynamic views for program comprehension [4, 9, 10].

Nowadays developers use elaborated IDEs (*e.g.*, Eclipse [5]) in their daily work to gain an understanding of software systems by navigating them. This increases the need to seamlessly integrate querying tools in the IDE so that a developer can directly pose questions about the system under study. In this paper we emphasize that query tools also need to cover dynamic information about programs and that these tools need to be seamlessly integrated in the IDE.

We identify three key research questions to achieve this goal. In the rest of the paper we address these questions:

- *How to efficiently gather dynamic information from within the IDE? (Section 2)*
- *How do we query collected dynamic information? (Section 3)*
- *How can we integrate these query tools and techniques into the IDE? (Section 4)*

2 Dynamic Information of Interest

A developer trying to understand a software system wants to ask questions about this system. In the following, we describe two typical questions a developer needs to get answered to better understand the subject system. We chose a web-based Wiki application as an exemplary system to illustrate concrete scenarios.

Class collaborators. Suppose the class Page of a Wiki application is hard to understand for a developer, in particular how this class is dependent on other parts of the system. For instance, she wants to know whether Page is communicating at runtime with the presentation layer of the Wiki, how the Wiki syntax of a page is parsed in order to correctly layout the text for the web, or how various authors of the very same page are handled by the Page class. Knowing all other parts of the system (*e.g.*, other classes) with which Page is collaborating at runtime would clearly help the developer finding answers to all these

questions more efficiently and more focused.

Frequent communication paths. The developer also discovers the existence of a performance bottleneck and hence wonders where so much communication is occurring in the Wiki application, *i.e.*, which methods get invoked most often. These methods can benefit most when they get optimized to perform their tasks more efficiently. The developer thus wants to know for example the ten methods that get most frequently executed in a specific feature being slow to learn whether the execution performance can be improved by optimizing any of these ten methods.

Querying techniques integrated in the IDE allow the developer to find answers to these two problems at hand directly where she works with the source code of this Wiki application. A prerequisite to successfully run such queries is of course the availability of dynamic information about the Wiki application under study. We describe shortly a working mechanism to gather dynamic information from within the IDE that has already been verified by other works [3, 7].

Collecting dynamic information. Analyzing the runtime behavior of applications, *e.g.*, by using tracing tools, is time-consuming and generates large amount of data. This makes these tools inappropriate for integration in IDEs, as developers want immediate benefit from the results of analyses. Partial behavioral reflection overcomes these problems as it enables us to select in a very fine-grained manner on what dynamic parts of a system we want to reflect upon.

To answer these questions mentioned above we basically need to know what kind of messages get sent to which receivers at runtime. To find out to which other classes Page communicates, it is enough to just dynamically observe the Page class. Using partial behavioral reflection we are capable to define that we are only interested in reasoning about runtime events occurring in Page and only in those events that send messages to other objects. We hence very precisely restrict the amount of resulting dynamic information already before any analysis takes place.

Answering the second question, *i.e.*, finding out the most frequently invoked methods, requires more dynamic information. In this case, we observe the whole Wiki package, but ignore invocations of system class methods. Again we are only interested in message send events, but not for instance in variable accesses.

Partial behavioral reflection is a suitable concept to limit the amounts of data being gathered dynamically in order to be able to directly collect the data from within the IDE without tremendously impeding neither the performance of the IDE nor of the executing application. A deeper treatment of the approach to partial behavioral reflection we are applying is given in the work of Denker [2], which was used to annotate feature execution [3] or to enrich the source code in the IDE with dynamic information [7] .

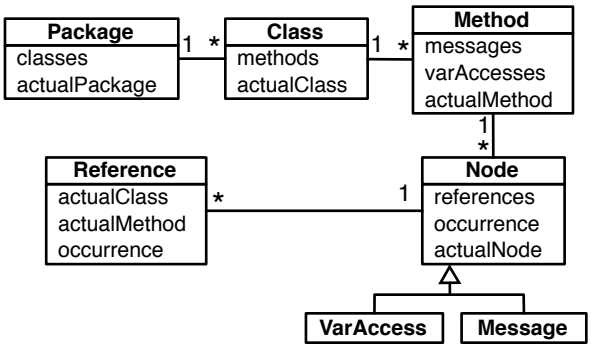

Figure 1. Schema of how dynamic data is stored in a database.

3 Querying Dynamic Information

In the preceding Section 2 we described two tasks to be solved by querying dynamic information. The dynamic information gathered with partial behavioral reflection gets stored in a database. The data in the database can, for instance, be structured as shown in Figure 1. A meta-modeled package holds a list of (meta-modeled) classes in it, each class can navigate to all methods that have been executed at runtime. Each method can navigate to all messages sent and all variables accessed in it. Message sends and variables are nodes (*i.e.*, method body operations) and each node has a reference to a class (in case of a variable access and a message send) and to an actual method (only in case of a message send). Furthermore, we store the number of occurrences of a given event, *e.g.*, of a specific variable access of a certain type.

We are now formulating queries to be sent to this database to find answers to the two questions we raised about the Wiki system. We postulate an SQL-like query language that is simple to use and understand.

Class collaborators. To identify all collaborators of the class Page we simply submit the query SHOW collaborators OF Page to the database. The answer to this query is a collection of classes dynamically collaborating with Page. All collaborators stored in the database are included in this result, but we can restrict the scope of the result to some particular features of the Wiki by enumerating the name of those features in a LIMIT clause of the query. Moreover, the query language also allows us to ask for collaborating methods instead of collaborating classes and also to ask for all collaboration occurring in one specific method of a class. The query answering for instance all methods that are invoked from within the method Page.text has this form: SHOW collaboratingMethods OF Page.text.

Frequent communication paths. Finding the most frequently invoked methods in a package is similar. The query

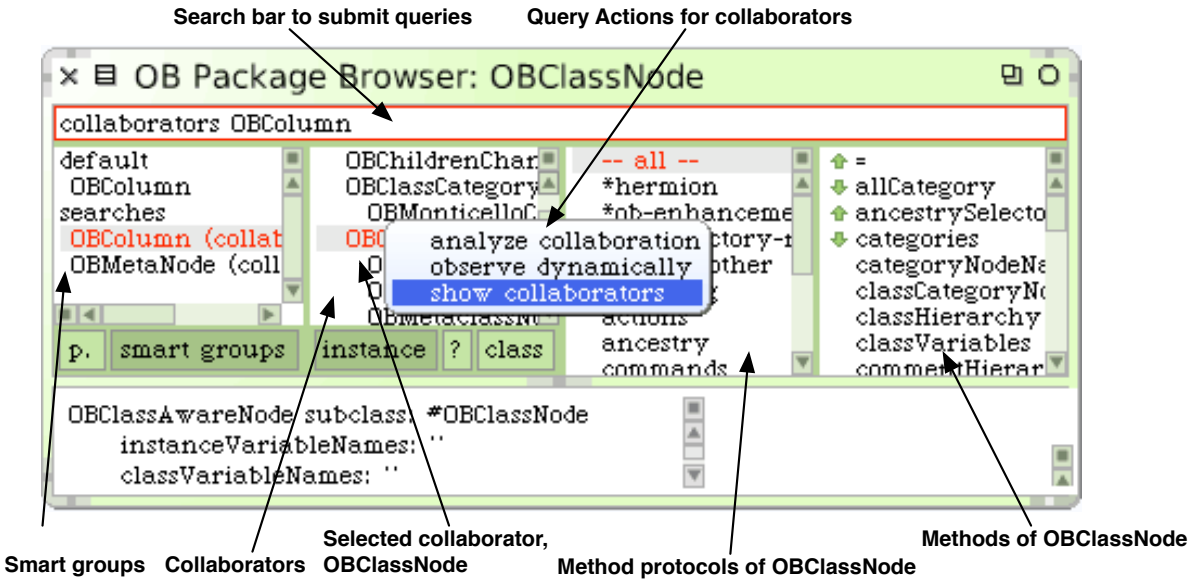

Figure 2. Smart Group showing class collaborators

answering the ten most invoked methods in the Wiki package looks like this: SHOW method invocations IN Wiki ORDER BY frequency LIMIT 10. Instead of covering a whole package, we can also ask for the methods most frequently invoked from within a specific class or even from within a single method. Furthermore, it is also possible to focus on the senders of those invoked methods to reveal who is invoking particular methods most frequently. The following query answers all senders of the Page.text method: SHOW senders OF Page.text.

Evaluating the method invocation query simply means traversing the object tree (modeled as shown in Figure 1) from a package, an instance of Package, down to all references by going over all classes and methods in this package. In Smalltalk, such an object graph traversal can be defined using just a few lines of code.

All these queries can certainly only give results about those parts of the application that have been executed. Parts not being covered by any execution or by any data gathering mechanism are simply invisible for the queries. Querying thus only leads to results if the application under study has been executed. The developer can tag specific executions with a name, *e.g.*, the name of the feature being executed. Queries can reason about these tags to *e.g.*, limit the scope of the results to specific runs of a system.

In Section 4 we elaborate how a developer can actually trigger the execution of such a query from within the IDE. It is crucial in order to effectively integrate querying of dynamic information in the development process, *e.g.*, to comprehend a software system, that the developer can rely on exhaustive tool support to submit those queries and to navigate their results. In the next section, we study a concept to integrate querying techniques in the IDE.

4 Integrating Query Tools in the IDE

As an IDE to integrate query tools we have chosen the Hermion browser in Squeak Smalltalk [8]. Hermion already incorporates tools to dynamically observe applications being developed in this IDE [7].

We consider it crucial that query tools we intend to integrate should complement or enhance the existing IDE tools. This means querying of dynamic information should not just be a tool somewhere available in the IDE, rather it should be tightly and seamlessly integrated in existing concepts and mind sets applied in the IDE. The Squeak IDE is basically a class browser with four columns used to navigate the source code (see Figure 2). These columns hold entities in the following order: packages, classes, method categories, methods. Below these four columns there is a text area to create or modify classes and methods. Every element in any column provides several actions to be triggered, *e.g.*, to create, browse or search some entities.

We propose three extensions to integrate querying tools into this IDE that can easily be ported to other IDEs as well:

- A search bar in the browser to submit textual queries
- New categories for source entities, holding the results of submitted queries, called *smart groups*
- New actions for elements (*e.g.*, classes, methods, dependencies between them) to submit specific queries

We now look in more detail at these three extensions:

Search bar. The search bar, basically a text field, expects a textual query. Such a query is for instance SHOW collaborators OF Page to see all classes with which *Page* communicates dynamically. Another query is SHOW method invocations IN Wiki, giving us a list of all methods being invoked in the *Wiki* package. The results of the queries are then presented in smart groups to categorize them (*e.g.*, all collaborator queries are stored in the same group). These groups are permanently accessible, this means results of queries get stored, but dynamically updated if they change over time, *e.g.*, because more information has been gathered about the running system.

Smart groups. Smart groups are a categorizing mechanisms orthogonal to the standard categorization applied in Squeak Smalltalk which is based on (static) packages. Smart groups are displayed in the same column as packages (*i.e.*, the first column), a switch allows us to go from one category mechanism to the other. A smart group holds results of queries and makes these results permanently accessible. The result of a query searching for class collaborators of the class OBColumn is displayed in the other three columns, *i.e.*, in the second column holds the classes collaborating with OBColumn, the next column the method protocols and the last column the methods of the selected collaborating class (see Figure 2). Like this the results of queries can be browsed in the same manner as the static source code of the application, so that the same tools and actions used there can also be used to browse query results.

Query actions. As it is tedious to enter textual queries, we provide actions that can be executed on a selected element, *e.g.*, a class. There is an action called 'dynamic collaborators' that just executes the collaborators query and jumps to the installed smart group. Or when looking at such a resulting collaborator there is the action 'analyze collaboration' which gives the developer a list of all methods in which a collaboration to the selected class is occurring. Similarly we can study in detail a method invocation to locate all methods in which such an invocation actually occurs at runtime. The action for this is called 'analyze method invocation'.

5 Discussion and Conclusion

In this paper we argue for querying dynamic information and integrating these querying techniques in the IDE.

A key issue of any dynamic analysis technique is efficiency. We experimented with medium-sized applications, observed parts of them dynamically using partial behavioral reflection and ran some benchmarks. The preliminary results reveal that there is a measurable slowdown between factor 2 and 4 for every application we analyzed when the core packages are fully covered and when both message sending and variable accessing is being observed. However, the applications were still usable and reasonably reactive.

A second issue we evaluated is the usability of the querying tools, *e.g.*, whether they indeed leverage program comprehension for the developer. We conducted a small experiment with some Squeak developers and asked them to answer a questionnaire. The preliminary feedback says that querying dynamic information is a very handy and useful feature that developers missed up to now. The developers in particular consider the effect on program comprehension as important to them in their daily work. Otherwise hidden dynamic information proves useful to more efficiently and also more thoroughly understand a software.

Other aspects we need to study further are the query language to be used or the optimization of the data gathering and of the database structure with respect to queries most frequently submitted. This we leave as future work.

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] S. Demeyer, S. Ducasse, K. Mens, A. Trifu, and R. Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering, 2003.

[2] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, Oct. 2007.

[3] M. Denker, O. Greevy, and O. Nierstrasz. Supporting feature analysis with runtime annotations. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis*, pages 29–33, 2007.

[4] 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.

[5] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[6] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *AOSD'03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 178–187, 2003. ACM.

[7] 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.

[8] Squeak home page. http://www.squeak.org/.

[9] 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.

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