Debugging into Examples

Leveraging Tests for Program Comprehension

Bastian Steinert, Michael Perscheid, Martin Beck,
Jens Lincke, and Robert Hirschfeld

Software Architecture Group
Hasso Plattner Institute, University of Potsdam, Germany
firstname.lastname@hpi.uni-potsdam.de

Abstract. Enhancing and maintaining a complex software system requires detailed understanding of the underlying source code. Gaining this understanding by reading source code is difficult. Since software systems are inherently dynamic, it is complex and time consuming to imagine, for example, the effects of a method's source code at run-time. The inspection of software systems during execution, as encouraged by debugging tools, contributes to source code comprehension. Leveraged by test cases as entry points, we want to make it easy for developers to experience selected execution paths in their code by debugging into examples. We show how links between test cases and application code can be established by means of dynamic analysis while executing regular tests.

Keywords: Program comprehension, dynamic analysis, test coverage.

1 Introduction

Developers of object-oriented software systems spend a significant amount of time reading code. Using, extending, or modifying parts of a system's source code requires an in-depth understanding, ranging from the intended use of interfaces to the interplay of multiple, interdependent system parts. Comprehending source code is an essential and important part of software development. It is, however, difficult for several reasons:

Abstraction: Source code describes the general behavior desired for a class of scenarios by abstracting from several concrete execution paths. When reading source code, developers have to imagine the effects on concrete execution paths.

Context: Every single class or method of a complex program can contribute to a large-scale collaboration of object teams. Knowing related classes and their contributions is important for efficient and exact source code comprehension. This context information, however, is not apparent in standard development environments.

Object-oriented Language Concepts: Object-oriented language concepts such as inheritance, sub-typing, and polymorphism are well-suited to describe system behavior, but late binding makes understanding the effects of

M. Núñez et al. (Eds.): TESTCOM/FATES 2009, LNCS 5826, pp. 235–240, 2009.

a method and following a message flow more difficult [4,16]. For instance, behavioral properties, such as message exchange in object-oriented programs, can only be determined precisely at run-time [1].

We argue that comprehending source code, imagining its effects, and understanding usage contexts can be supported by enabling developers to experience source code by examples of concrete execution paths from which the source code abstracts.

Related research results have already emphasized the usefulness of examples and extensions to Integrated Development Environments (IDEs) for source code comprehension [5,6,12,14]. In [5], for example, the author argues that unit tests can often be considered as usage examples for the units under test. He suggests using coding conventions to make the link between tests and the corresponding units explicit. To our knowledge, an approach that allows to experience execution examples directly while browsing source code has not been reported. Debugging tools also support familiarization with source code. They offer the opportunity to understand source code by means of stack trace inspection. Still, being able to debug requires an appropriate entry point.

In this paper, we suggest to regard test cases as natural entry points to source code, provided they are linked to the application code. We present an efficient approach to run-time analysis for establishing these links during the execution of tests.

In Section 2, we describe our approach to debugging into concrete execution paths by leveraging test cases. Section 3 explains how the required run-time information can be collected efficiently during the execution of tests. Section 4 discusses related work and Section 5 summarizes our results.

2 Using Test Cases as Entry Points

We suggest the use of debugging tools to support source code comprehension by leveraging test cases as entry points. This requires each method of interest to be covered by at least one test case. Looking at this from another perspective, every test case covers a set of methods of interest during execution. Consequently, the set of test cases covering a particular method describes prospective entry points for debugging this method. So, this set needs to be determined upfront by making the implicit relationship between test cases and covered methods explicit.

To discover relationships between source code entities, static as well as dynamic analysis [2] can be used. Static source code analysis, however, has limited applicability due to programming language concepts such as inheritance and polymorphism. In contrast to static analysis, dynamic analysis allows to collect run-time information such as method bindings [13] during the execution of a particular path. As this is essential for making coverage links explicit, our approach is based on dynamic analysis.

Application code is analyzed during test execution. When a test case is executed, all covered methods are recorded. Having obtained these run-time data,

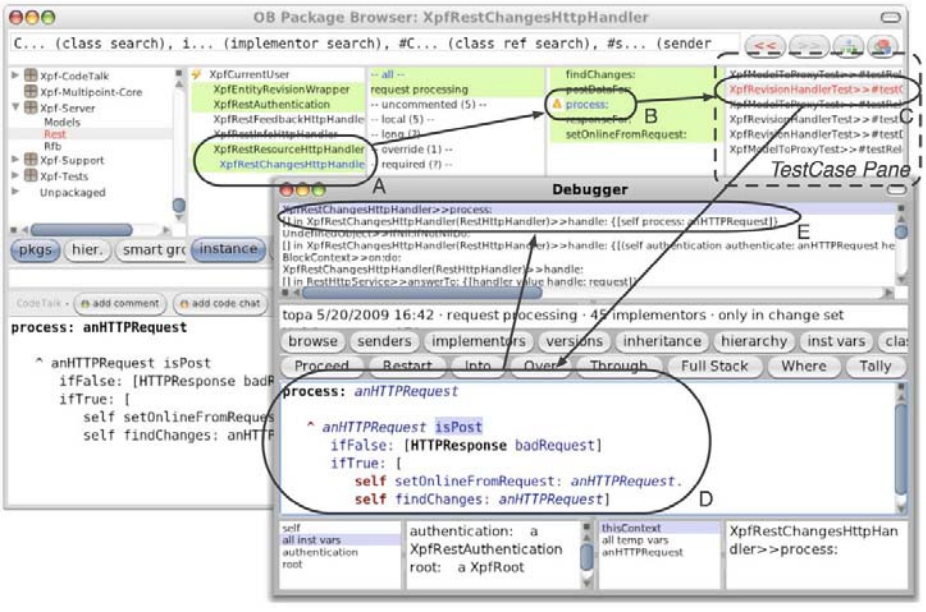

Fig. 1. An extended code browser and a debugger window in Squeak. The code browser has an additional pane on the right (C) that shows a list of test cases covering the selected method named *process:*. Classes (A) and methods (B) covered by tests are highlighted with a background color.

we can mark the set of test cases that cover a certain method during execution, so that we can provide developers with the required entry points.

The list of entry points, the test cases, can be leveraged by IDE extensions enabling developers to experience a run-time view of a method directly while browsing it. We extended the source code browsing tools of Squeak Smalltalk [9]. Figure 1 depicts an extended Squeak code browser. In its base version, it consists of four panes containing lists of available packages, classes, method categories, and methods respectively (from left to right). We added a fifth pane showing the list of all test cases covering the selected method.

Selecting a test case opens a debugger (Figure 1 D), allowing the user to analyze the covered method of interest during execution (Figure 1 E). The selected test case is executed, the execution halts at the selected method, and the debugger is opened. Developers may now explore collaborating objects and examine their state. They can further step down in the stack and inspect the execution of the calling method, or they can step into a called method.

3 Efficient Tracing during Test Execution

In this section, we discuss selected implementation details of our tracing approach. Traditional tracing approaches are typically inefficient and produce large

amounts of data [3]. As we analyze the execution of test cases, the overhead caused by tracing has to be minimal. If running tests is time-consuming, developers will either not run them very often or reject our approach.

Most tracing approaches are designed for general analysis purposes and thus have to record lots of data such as the state of all objects for each step in the execution [12]. However, creating and persisting deep copies of many objects is time consuming. We could significantly reduce the overhead by collecting only required run-time information, i.e., references to methods being covered during test execution. Test execution performance is thus only decreased by a factor of two on average, which is usually not perceivable when single tests are executed.

To record the relevant run-time data, we apply aspect-oriented programming techniques [8,10]. We developed a tracing aspect that intercepts all calls of selected application methods [7]—developers are usually not interested in tracing all libraries of the system. The tracing aspect is deployed dynamically during test execution. For each test case, the aspect code collects for each test case the set of application methods covered during test execution.

This coverage information is stored in the Squeak system so that tools such as our browser extension can access it easily and efficiently. In a Squeak system, source code entities such as classes and methods are managed as objects. We enriched the interfaces to method objects to manage and persist coverage information. This information is managed as a bidirectional relationship between test case method objects and application method objects. With that, our browser extension can easily retrieve all test cases that cover a selected method of interest.

4 Related Work

To the best of our knowledge, there are only a few approaches that integrate results of dynamic analysis into development environments.

The feature driven browser [14] is based on the ideas of feature location [6] and combines an interactive visual representation of features with its related source code entities within the Squeak IDE. The feature driven browser summarizes dynamic behavior from the features' points of view. A case study has shown that developers using this browser are faster in fixing defects when they know corresponding solution artifacts. In contrast to this approach, we offer concrete sample traces that can be further inspected with a debugging tool.

The Squeak IDE extension Hermion [15] enriches source code views with run-time data. It provides additional type information and offers dynamic reference information for locating classes actually used. Hermion further supports a new navigation technique based on executed methods. In contrast to Hermion, our approach allows developers to explore concrete execution paths at run-time and to examine how objects behave and change step by step.

WhyLine [11] is a debugger rather than an IDE but offers sophisticated means for inspecting the system at run-time. During the execution, developers can ask a set of "why did" and "why didn't" questions derived from the program's code and behavior. Based on static and dynamic slicing, call graph analysis, and

several other techniques, WhyLine can answer questions such as why a line of code was not reached. However, the WhyLine approach can currently not meet our performance needs. Recording the huge amount of required run-time data and their analysis is too time-consuming.

5 Summary and Outlook

In this paper, we describe the need for additional views on software systems that allow developers to explore concrete examples from which source code usually abstracts. Contemporary debugging tools enable developers to inspect concrete execution paths, but do not suggest appropriate entry points for application exploration. In our approach, we propose test cases as candidates for such entry points.

We present an efficient tracing technique for collecting coverage data during the execution of test cases; and we show that these data can be used in IDEs to give developers the opportunity to debug into a method of interest and experience concrete sample executions of this method. With that, our paper reveals another benefit of developing and maintaining tests cases; they may be leveraged to provide a run-time view on source code and thus ease its comprehension.

The benefit of having these links is the ability to re-execute corresponding tests after source code modifications automatically. The debugging facilities should be integrated into standard code browsing tools, enabling a seamless transition between static and dynamic views. Furthermore, research results described in [5] might be useful to filter and order the list of possible entry points according to relevance.

Acknowledgments. We gratefully acknowledge the financial support of the Hasso Plattner Design Thinking Research Program for our project “Agile Software Development in Virtual Collaboration Environments”. We thank Michael Haupt for valuable feedback on earlier versions of this paper.

References

1. Arisholm, E.: Dynamic Coupling Measures for Object-Oriented Software. In: IEEE International Symposium on Software Metrics (2002)
2. Ball, T.: The Concept of Dynamic Analysis. In: ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, London, UK, pp. 216–234. Springer, Heidelberg (1999)
3. Denker, M., Greevy, O., Lanza, M.: Higher Abstractions for Dynamic Analysis. In: 2nd International Workshop on Program Comprehension through DynamicAnalysis (PCODA 2006), pp. 32–38 (2006)
4. Dunsmore, A., Roper, M., Wood, M.: Object-oriented Inspection in the Face of Delocalisation. In: ICSE 2000: Proceedings of the 22nd International Conference on Software Engineering, pp. 467–476. ACM, New York (2000)

5. Gaelli, M.: Modeling Examples to Test and Understand Software. PhD thesis, University of Berne (2006)
6. Greevy, O.: Enriching Reverse Engineering with Feature Analysis. PhD thesis, University of Berne (May 2007)
7. Gschwind, T., Oberleitner, J.: Improving Dynamic Data Analysis with Aspect-Oriented Programming. In: CSMR 2003: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, Washington, DC, USA, pp. 259–268. IEEE Computer Society, Los Alamitos (2003)
8. Hirschfeld, R.: AspectS – Aspect-Oriented Programming with Squeak. In: Aksit, M., Mezini, M., Unland, R. (eds.) NODe 2002. LNCS, vol. 2591, pp. 216–232. Springer, Heidelberg (2003)
9. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In: OOPSLA 1997: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 318–326. ACM, New York (1997)
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
11. Ko, A.J., Myers, B.A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 301–310. ACM Press, New York (2008)
12. Pauw, W.D., Lorenz, D., Vlissides, J., Wegman, M.: Execution Patterns in Object-Oriented Visualization. In: COOTS 1998: Proceedings of the 4th Conference on Object-Oriented Technologies and Systems, Berkeley, CA, USA, pp. 16–16. USENIX Association (1998)
13. Richner, T., Ducasse, S.: Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In: ICSM 1999: Proceedings of the IEEE International Conference on Software Maintenance, Washington, DC, USA, pp. 13–22. IEEE Computer Society, Los Alamitos (1999)
14. Röthlisberger, D., Greevy, O., Nierstrasz, O.: Feature Driven Browsing. In: ICDL 2007: Proceedings of the 2007 International Conference on Dynamic Languages, Lugano, Switzerland, pp. 79–100. ACM Press, New York (2007)
15. Röthlisberger, D., Greevy, O., Nierstrasz, O.: Exploiting Runtime Information in the IDE. In: ICPC 2008: Proceedings of the 16th IEEE International Conference on Program Comprehension, Washington, DC, USA, pp. 63–72. IEEE Computer Society, Los Alamitos (2008)
16. Wilde, N., Huitt, R.: Maintenance Support for Object-oriented Programs. IEEE Transactions on Software Engineering 18(12), 1038–1044 (1992)