Declarative and Visual Debugging in Eclipse

Jeffrey K. Czyz
Computer Science and Engineering
University at Buffalo (SUNY)
jkczyz@cse.buffalo.edu

Bharat Jayaraman
Computer Science and Engineering
University at Buffalo (SUNY)
bharat@cse.buffalo.edu

ABSTRACT

We present a declarative and visual debugging environment for Eclipse called JIVE.[1] Traditional debugging is *procedural* in that a programmer must proceed step-by-step and object-by-object in order to uncover the cause of an error. In contrast, we present a *declarative* approach to debugging consisting of a flexible set of queries over a program's execution history as well as over individual runtime states. This runtime information is depicted in a visual manner during program execution in order to aid the debugging process. The current state of execution is depicted through an enhanced object diagram, and the history of execution is depicted by a sequence diagram. Our methodology makes use of these diagrams as a means of formulating queries and reporting results in a visual manner. It also supports revisiting past runtime states, either through reverse stepping of the program or through queries that report information from past states. Eclipse serves as an ideal framework for implementing JIVE since, like the JIVE architecture, it makes crucial use of the Java Platform Debugging Architecture (JPDA). This paper presents details of the JIVE architecture and its integration into Eclipse.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids, Tracing; D.2.6 [**Programming Environments**]: Graphical environments, Interactive environments

1. MOTIVATION

There has been increasing interest in recent years in object-oriented technologies and integrated development environments (IDEs). While much attention has been given to the areas of software design and implementation, tools that enhance understanding program behavior at runtime have been lagging behind. According to a recent survey, the cost of inadequate software testing and debugging on the U.S. economy is estimated at nearly $60 billion annually [16]. Hence, there is a critical need for improved techniques for debugging object-oriented software.

The state of the art in runtime environments for object-oriented programs is exemplified by IDEs such as Eclipse, NetBeans, and Visual Studio. Typical features found in such systems include setting of breakpoints, spying on variables, stepping forward in execution, and examining variables on the call stack. Using these features the programmer must proceed step-by-step and object-by-object to uncover the cause of an error. While these IDEs also provide graphical interfaces, they serve mainly as front-ends for traditional text-based debugging. Such debuggers may be categorized as procedural and textual in nature. The main contribution of our research is in providing a declarative and visual environment for program comprehension and debugging.

There have been a few recent studies on the nature of errors in object-oriented programs. In general, errors may be due to a flawed design, incorrect use of language constructs, algorithmic errors, and oversights in coding [8]. Central to the understanding of any program, whether object-oriented or not, is comprehending how variables take on certain values. Often a program error arises because a variable has taken on an unexpected value, resulting in an exception such as a null pointer exception. In order to identify such errors, it is desirable that the debugging environment provides queries that, for example, elicit all changes to a variable, determine at what execution point a variable takes on a certain value, check when an invariant is first violated, etc. This requires runtime support for examining the current state as well as past states. Such a query-based debugger may be categorized as declarative in nature because it allows the programmer to search the entire execution history without engaging in a laborious step-by-step procedural process. We believe the declarative approach complements procedural debugging just as web searching complements web browsing.

Another common cause of program errors lies in the difference between a programmer's mental model of runtime states and the actual states. Visual depiction of runtime states can help highlight these difference more easily. In designing an appropriate visualization, it should be noted that object-oriented programs differ from procedural programs in that objects are not just data structures but serve as environments within which method activations occur, and also that OO methodology engenders the use of smaller methods and results in more complex object interactions. Hence,

[1] `http://www.cse.buffalo.edu/jive/`

it is desirable to visualize both the current state and the history of execution. However, practical experience with visualizations shows that they do not scale gracefully for large executions due to the large number of objects and method activations that arise [2]. In order to achieve scalable visualizations, we not only need the ability to construct concise and abstract diagrams, but also a query capability by means of which only relevant portions of the visualization are depicted. Thus, declarative queries and visualization work in concert to provide a more effective runtime comprehension and debugging environment.

An initial step towards the realization of such a runtime environment was taken in our previous research on the Java Interactive Visualization Environment (JIVE) [6, 7]. JIVE depicts the current state through an enhanced object diagram (showing objects and method activations) and the history of execution through a UML-like sequence diagram. It is interactive in that the user may step forward or backward in the execution history to revisit previous runtime states. This feature is invaluable as programmers often step past errant statements while debugging and must re-execute the program to examine its state [1]. While the main focus of the JIVE research has been on interactive visualization, the goal of our current work is aimed at providing a more comprehensive declarative and visual execution environment.

2. JIVE DEBUGGING METHODOLOGY

We have incorporated our declarative and visual execution environment (JIVE) into Eclipse, which is an extensible development platform based upon a plug-in architecture [21]. An overview of the JIVE architecture can be found in Figure 1. An important component of Eclipse is the Java Development Tools (JDT), and hence using Eclipse allows JIVE to inherit the functionality of JDT. Since both JDT and JIVE are based upon the Java Platform Debugging Architecture (JPDA), Eclipse serves as an ideal experimental framework for our research.

JPDA is the key component of JIVE, allowing it to observe a running Java program without modifying the compiler or virtual machine. JIVE requests notification of certain runtime events using JPDA's Java Debug Interface (JDI). Such events include class preparation, step taken, variable modification, method entry and exit, thread start and death, etc. The user may specify whether to filter out events related to certain classes, such as those from the Java Platform. Upon event notification, JIVE may suspend the target virtual machine in order to glean any pertinent information from the call stack that may not be available from the JDI event. This together with the JDI event forms a JIVE event, and as the program executes a sequence of JIVE events is formed.

Figure 2 shows a snapshot from our current prototype which supports object and sequence diagrams with granularity control and zooming capability, query processing, and reverse stepping. We will explain each of these capabilities in the remainder of the section.

JIVE constructs two main models from the above event sequence: an object model and a sequence model. The object model represents the program's execution state, while the sequence model details its history of execution. These models serve as the basis from which JIVE's visualizations are derived. An object diagram depicts the program's execution state by showing objects and their structural links as well as outstanding method activations. In this sense, JIVE's object diagram defers from the UML object diagram. By depicting method activations in their object context, we not only clarify the semantics of method activations but also facilitate program comprehension and debugging. The object diagram is also helpful in clarifying Java's semantics for overriding and shadowing.

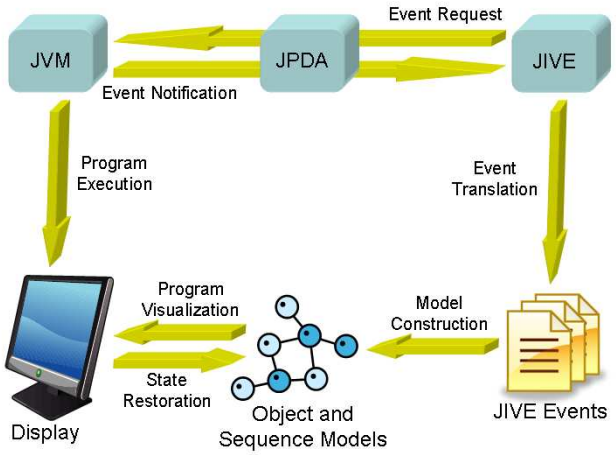

Figure 1: Overview of the JIVE architecture.

The JIVE sequence diagram is constructed interactively at execution time. Our approach differs fundamentally from the UML sequence diagram in that the latter documents design-time considerations, whereas the JIVE sequence diagram reflects the actual sequence of object interactions at execution time. In this respect, our approach also differs from other approaches that construct sequence diagrams by reverse engineering the source code [12]. Our sequence diagram also differs from other work [20] in that JIVE constructs the diagram interactively rather than as a form of postmortem analysis after the program has completed execution. In JIVE, every point on the sequence diagram is correlated with the object diagram that would have been in effect at that execution point. JIVE also supports interactive forward as well as reverse stepping of the program. Through the sequence diagram, a user may direct the JIVE engine to any previous point in the execution history in order to inspect the object diagram at that execution point.

Scalable Visualizations. In general, there could be a large number of objects and method activations arising during program execution, and hence we are interested in constructing compact object and sequence diagrams. JIVE supports both abstraction and geometric reduction of the diagrams. Abstraction involves suppressing the internal details of objects and their interactions. For object diagrams, this includes suppressing superclass details, hiding field tables, showing only objects involved in the call path, hiding aggregated objects, etc. For sequence diagrams, we are exploring how to obtain reduced sequence diagrams in which sub-

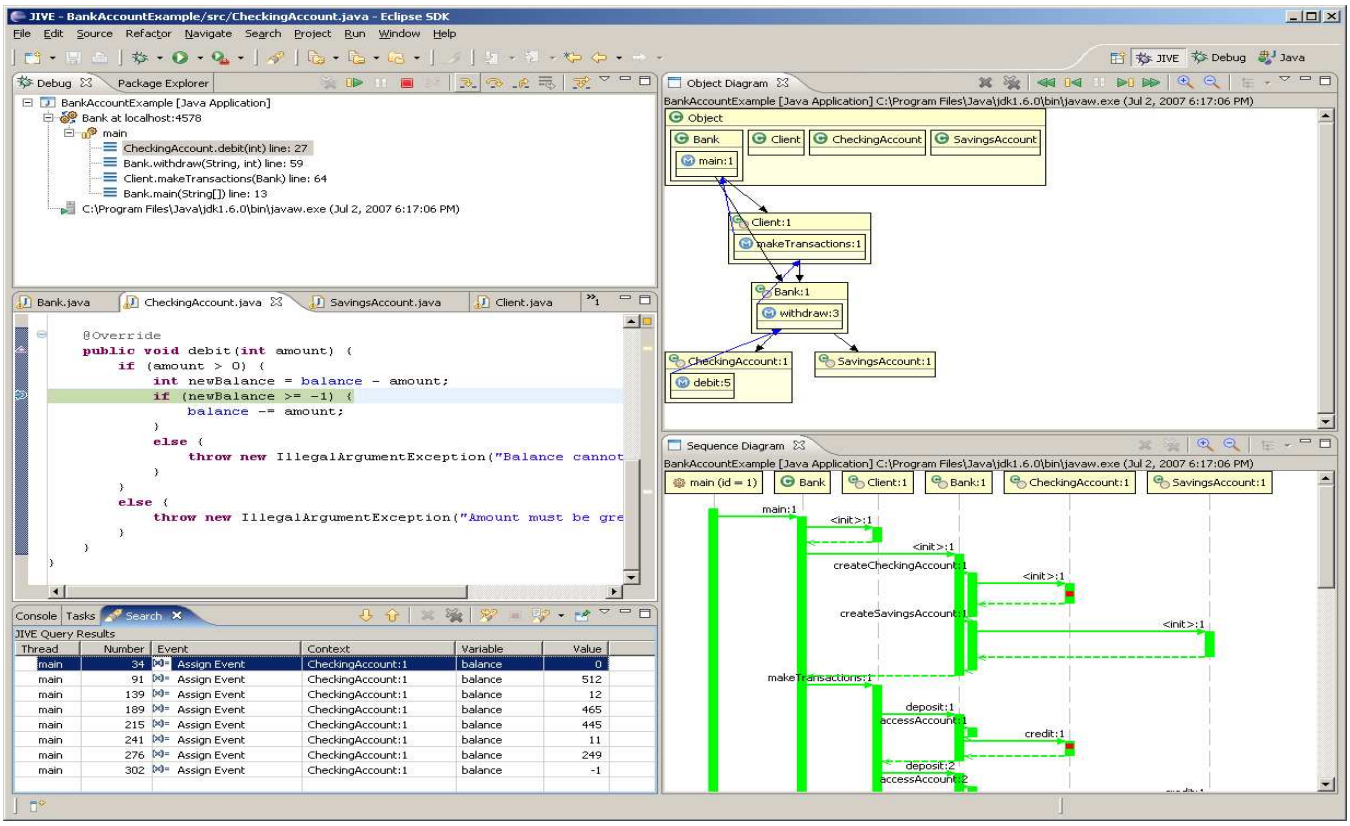

Figure 2: JIVE prototype showing object and sequence diagrams, call stack, source code, and query results.

computations corresponding to large call trees are replaced by a single, abstract node. This in turn helps compact the timeline and obtain a concise drawing. Geometric reduction is similar to a zooming out operation.

Declarative Queries. Another important way to achieve scalable visualizations is to depict only those portions of the object and sequence diagrams that are relevant to a query of interest to the programmer. We presently support queries not only on a given execution state but also over the entire history of execution or any subinterval. This is illustrated in Figure 2 for a query that elicits all changes to variable `balance` in object `CheckingAccount:1`. The query itself was formulated through the Search dialog, but in general such queries may also be composed using the object diagram, sequence diagram, or the source code. The results are shown in tabular form in the search view and also as highlighted execution points on the sequence diagram. Thus, the sequence diagram serves as an excellent framework for visually reporting the answers to queries, as it helps identify where answers lie spatially and temporally. In general, the queries may elicit information about objects, method activations, exceptions, fields, local variables, as well as whether certain properties are maintained or violated at various points in the execution history.

Interactive Execution. Another important component of our debugging environment is the support for both forward and reverse interactive execution. JIVE currently performs incremental state-saving during forward execution and performs incremental state-restoration during reverse execution. While this approach incurs minimal storage overhead, it is less efficient for performing jumps back to distant runtime states because each state must be restored one-by-one until the desired state is reached. (Query results may also refer to points in the execution history, necessitating jumps to disparate execution states.) We are exploring the use of three types of checkpoints to facilitate back jumping. A full checkpoint contains all the information necessary to recreate a program state. A differential checkpoint records changes in program state since the last differential or full checkpoint. An incremental checkpoint records changes in program state after each step in execution. The current implementation supports interactive execution using incremental checkpoints.

3. INTEGRATION IN ECLIPSE

An important component of Eclipse is the Java Development Tools (JDT), which includes debugging support in the JDT Debug module. JDT Debug utilizes JPDA in order to provide traditional debugging capabilities such as setting of breakpoints, stepping through execution, examining variables on the call stack, etc. Our goal has been to seamlessly integrate JIVE into Eclipse in such a way that it inherits the functionality of JDT Debug. This provides users with all the traditional debugging tools they are accustomed to using in addition to the declarative and visual debugging features which JIVE has to offer. It also saves us the effort of re-inventing the wheel by not needing to write a full-blown Java debugger.

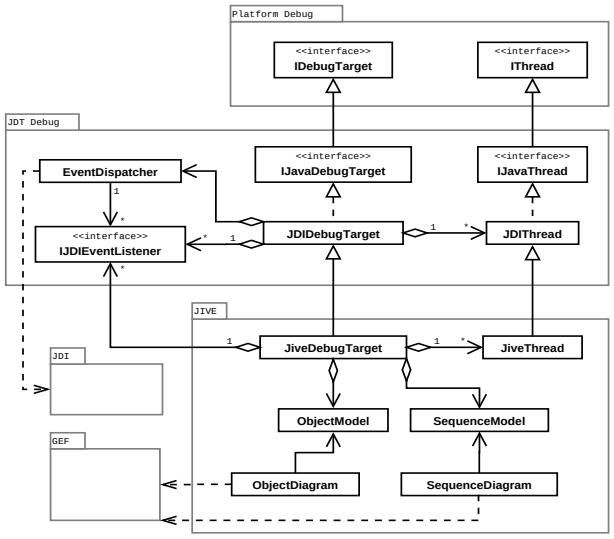

Figure 3: Integrating the JIVE debugger in Eclipse.

JDT Debug is a Java-specific implementation of Eclipse's language-independent debug model, Platform Debug. The debug model consists of various debug elements such as debug targets, threads, stack frames, and variables. JDT Debug implements these elements using JDI. Event requests are made through a `JDIDebugTarget` and events are handled by `IJDIEventListener`s. These act as a façade to the JDI event request model. Unfortunately, this framework is internal to JDT Debug, and therefore its use is discouraged. However, in order for JIVE to work correctly alongside JDT Debug, this framework had to be used. Attempting to use JDI directly would interfere with its use by JDT Debug. Our implementation involves deriving classes from `JDIDebugTarget` and `JDIThread` which then employ custom `IJDIEventListener`s. The JIVE-specific JDI event handling is performed by these listeners before delegating to JDT Debug's event handlers. Particular care had to be given in handling the JDI `StepEvent`. JDT Debug uses step events to implement stepping from breakpoints. However, JIVE also needs to use step events to determine when local variables have been modified as JDI does not provide this functionality. Hence, custom step handlers were developed to collect step events for JIVE and at the same time simulate stepping from breakpoints when needed. The relationships between these components are shown in Figure 3.

Having formed our debug model as an extension of JDT Debug, we next had to allow Eclipse to use it in place of the standard debug model. Our goal was to allow the user to choose between the models. To this end, we made use of the mixed mode launching extension points introduced in Eclipse 3.3. Users can specify whether or not they would like to debug a program with JIVE through a launch configuration. Enabling JIVE will cause Eclipse to use a customized launch delegate for JIVE instead of the standard launch delegate provided by JDT Debug. Our delegate works identical to the standard delegate except that it uses our customized debug model elements mentioned earlier. Thus, this allows the monitoring of JDI events and the formation of the JIVE event model which supports our visualizations.

The visualization aspect of JIVE takes the form of an Eclipse view for each of our diagrams. We chose to implement the drawings using the Graphical Editing Framework (GEF), now standard with Eclipse. GEF was chosen because its model-view-controller architecture is a natural fit with our own architecture. While the business-logic of our models is not editable by users, the presentation-logic (such as positioning of objects and granularity control) is modifiable. This makes GEF an excellent choice for our visualization needs. GEF also comes standard with useful diagram functionality such as zooming controls and model element selection. The ability to select model elements through their diagrammatic representations allows us to support context-sensitive commands such as diagram compaction and declarative querying.

A new feature to JIVE is its support of querying a program's execution history. Queries over a program's execution history can be formulated through either the source code or the diagrams. Query results typically correspond to points on the sequence diagram. We choose to utilize the extension points available through the Eclipse search plug-in to contribute our `ISearchPage`s and `ISearchResultPage`s to the Eclipse UI. Our search pages may be filled in automatically based on the view context (e.g., selected diagram element) or manually by the user. Results are displayed in the Search view, which allows for easy result navigation. This also allows us to provide diagram and gutter annotations for search results in a fashion similar to the JDT search capabilities.

4. CONCLUSIONS AND FURTHER WORK

The integration of JIVE in the Eclipse plug-in architecture is a continuation of our earlier work on a stand-alone visual debugging environment for Java. This integration overcomes several shortcomings of the stand-alone version, and also provides substantial new capabilities especially in the area of declarative queries and scalable visualizations. JIVE has been tested out in a number of settings including in courses on programming languages. The integration of JIVE into Eclipse allows this tool to be used in any course where Eclipse is the chosen development environment. We plan to use JIVE more extensively in courses at Buffalo to better determine its effectiveness.

Our initial experiments with JIVE have been conducted on a laptop with a 1.6 GHz processor and 1 GB of memory. These experiments indicate that its interactive performance on small- to medium-sized programs is satisfactory. (While we have not tested JIVE on large programs, the recent focus on Test-Driven Development places more emphasis on executing smaller units of code.) Although we have not performed a formal overhead analysis of JIVE, it may be noted there are two main areas of interest: (i) event gathering and (ii) visualization. In the former case, the inherent overhead to observing a running program is unavoidable. Event gathering may further slowdown the execution as a result of suspending the virtual machine in order to elicit additional information from the call stack (e.g., to determine which local variables have changed). Ideally, the visualizations should be updated after each JIVE event. However, in order to enhance interactive performance, JIVE updates the visualizations periodically at a user-defined interval.

Our current and future work can be broadly divided into the following areas. In declarative queries, we are developing a more comprehensive and composable set of queries as well as their visual formulation. In scalable visualization, we are exploring techniques for abstracted and reduced object and sequence diagrams, as well as optimal/aesthetic drawings of these diagrams. In reverse execution, we plan to determine optimal placement of differential and full checkpoints to enhance the performance of reverse jumping. In the area of query processing, we plan to explore efficient data structures that will facilitate improved performance. We also plan to explore 'why' queries, aggregate queries, etc.

While there a number of approaches for program visualization [3, 6, 17, 18, 19, 22] querying [10, 13, 15], and interactive forward and reverse execution [6, 14], JIVE is unique in combining all these capabilities into a single environment for debugging. We survey closely related approaches below. BlueJ [11] is an early example of a well-known pedagogic tool supporting static and dynamic visualization. JavaVis [17] supports visualization with object and sequence diagrams. However, it does not support querying or revisiting past states. Eclipse's Test & Performance Tools Platform (TPTP) Project provides UML sequence diagrams of running programs with abstraction techniques, but it does not support object diagrams, querying, or reverse execution. De Pauw et al[2] and Sharp et al [19] offer many useful techniques for reducing the size of program execution visualizations that could be adapted to our work. Finally, approaches such as [3, 18] differ fundamentally from our approach in that they are focused on visualizing runtime performance metrics.

A few recent projects use the concept of declarative queries for program analysis (finding errors and security flaws) [9, 13, 15]. An important difference in our approach is our emphasis on interactive debugging without having to develop a compiler/preprocessor to perform source code instrumentation or modifying the JVM. Our queries related to the cause of various program events are closely related to the Whyline interrogative debugger [10]. Our analysis of the execution history is related to the trace-based analyzer of Opium [4].

5. ACKNOWLEDGEMENTS

Funding for this research was provided by a 2006 IBM Eclipse Innovation Award. Thanks to Paul Gestwicki for developing the first version of JIVE [5] which served as an inspiration and a basis for our Eclipse implementation. Thanks to Dennis Patrone for implementing an initial set of debugging queries for JIVE.

6. REFERENCES

[1] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Soft.—Prac. & Exper.*, 23(6):589–616, June 1993.

[2] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234, April 1998.

[3] W. De Pauw and J. M. Vlissides. Visualizing object-oriented programs with Jinsight. In *ECOOP '98: Workshop ion on Object-Oriented Technology*, pages 541–542, London, UK, 1998. Springer-Verlag.

[4] M. Ducassé. Opium: An extendable trace analyser for Prolog. *The Journal of Logic Programming*, 1999.

[5] P. V. Gestwicki. *Interactive Visualization of Object-Oriented Programs*. PhD thesis, University at Buffalo, June 2005.

[6] P. V. Gestwicki and B. Jayaraman. Interactive visualization of Java programs. In *Proceedings of the IEEE 2002 Symposium on Human-Centric Computing, Languages, and Enviornments (HCC '02)*, pages 226–235, September 2002.

[7] P. V. Gestwicki and B. Jayaraman. Methodology and architecture of JIVE. In *SoftVis*, pages 95–104, 2005.

[8] H. Z. Girgis, B. Jayaraman, and P. V. Gestwicki. Visualizing errors in object-oriented programs. In *OOPSLA '05 Companion*, pages 156–157, 2005.

[9] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402, October 2005.

[10] A. J. Ko and B. A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *CHI*, pages 151–158, April 2004.

[11] M. Kölling and J. Rosenberg. Guidelines for teaching object-orientation with Java. *ACM SIGCSE Bulletin*, 33(3):33–36, 2001.

[12] R. Kollmann and M. Gogolla. Capturing dynamic program behavior with uml collaboration diagrams. In *Proc. CSMR '01*, page 58, 2001.

[13] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA*, pages 304–317, October 1997.

[14] B. Lewis. Debugging backwards in time. In *ArXiv Computer Science*, pages 225–235, 2003.

[15] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, pages 365–383, October 2005.

[16] National Institute of Standards and Technology (NIST). The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, May 2002.

[17] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). 2002.

[18] S. P. Reiss. Visualizing Java in action. In *ACM SoftVis*, pages 57–65, 2003.

[19] R. Sharp and A. Rountev. Interactive exploration of UML sequence diagrams. In *VisSoft*, pages 8–13, 2005.

[20] T. Systä, K. Koskimies, and H. Müller. Shimba—an environment for reverse engineering java software systems. *Soft.—Prac. & Exper.*, 31(4):371–394, 2001.

[21] The Eclipse Foundation. Eclipse Platform. `http://www.eclipse.org/`.

[22] A. Zeller and D. Lütkehaus. Ddd—a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.