Selective Tracer for Java Programs

Lothar Wendehals
Software Engineering Group
Department of Computer Science, University of Paderborn
Warburger Straße 100, 33098 Paderborn, Germany
lowende@upb.de

1. Introduction

Reverse engineering based on dynamic analyses often uses method traces of the program to be analyzed. Recording all method traces during a program's execution produces too much data, though for most analyses, a "slice" of all method traces is sufficient. Furthermore, the monitoring of a complete program extremely reduces the runtime performance.

Our tool JavaTracer records method traces during the execution of a Java program. To reduce the amount of data, the JavaTracer is able to record calls of selected methods only. In comparison to the Java Debugging Interface (JDI) [2] that also provides a native tracing of a Java program, the JavaTracer is much more efficient.

The JavaTracer is used in our tool-supported semiautomatic approach to design recovery [1] within our UML Case Tool Fujaba Tool Suite [3]. This approach facilitates the recognition of design pattern instances in the source code of a system. We combine static and dynamic analysis [4, 5]. The static analysis identifies pattern instance candidates, whereas the subsequent dynamic analysis verifies the candidates. Therefore, only methods of pattern instance candidates have to be monitored by the JavaTracer.

2. The JavaTracer

The JavaTracer gets a list of classes and selected methods that have to be monitored during the program's execution. It executes the program, called the debuggee, by connecting to the debuggee's virtual machine. For each selected method, two breakpoints are set at the beginning and at the end of the method body.

Abstract methods of interfaces or abstract classes can also be selected. The JavaTracer is able to record all calls of methods implementing those abstract methods. Even overridden methods can be monitored, such providing analyses that include polymorphism and dynamic method binding.

When the debuggee reaches a breakpoint, the JavaTracer will be informed. It halts the debuggee and asks the debuggee's virtual machine for additional information about the method call. This includes information about the method name, the time stamp for the method call, the names and unique identifiers of the caller and callee objects, the identifiers of objects passed as arguments as well as the current thread. Then the debuggee's execution is continued.

The debuggee is controlled either manually by the reengineer or by automated tests. The output consists of a list of method entry and exit events in the order of their occurrence which can be further analyzed, e.g. by the dynamic pattern analysis.

3. Performance

Table 1 shows the performance of different executions of the Fujaba Tool Suite. In the first scenario, the duration of starting Fujaba was measured[1]. In the second and the third scenario, a project was opened in Fujaba. The first project consists of a class diagram with 12 classes, the second one of a class diagram with 27 classes and 178 activity diagrams. Four major classes were monitored.

Scenario	$t_{w/o}$	t_{break}	t_{events}
Starting Fujaba	5,39 sec.	8,4 sec.	103,37 sec.
Open Project I	2,85 sec.	20,65 sec.	241,78 sec.
Open Project II	6,28 sec.	49,58 sec.	923,03 sec.

Table 1: Duration of program tracings

The program was executed without any tracing ($t_{w/o}$), using our breakpoint approach (t_{break}) and using the native *MethodEntry* and *MethodExit* event mechanism of JDI (t_{events}). JDI was configured by filters to monitor the same methods as the JavaTracer. Even though the JavaTracer uses JDI for setting breakpoints, it improves the performance of method tracing significantly compared to the native tracing provided by JDI.

References

[1] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[2] Sun Microsystems. *Java Platform Debugger Architecture. http://java.sun.com/products/jpda.*

[3] University of Paderborn, Germany. *Fujaba Tool Suite. http://www.fujaba.de.*

[4] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In J. Cook and M. Ernst, editors, *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, pages 29–32, May 2003.

[5] L. Wendehals. Specifying patterns for dynamic pattern instance recognition with UML 2.0 sequence diagrams. In E.-E. Doberkat and U. Kelter, editors, *Proc. of the 6th Workshop Software Reengineering (WSR), Bad Honnef, Germany, Softwaretechnik-Trends*, volume 24/2, pages 63–64, May 2004.

[1]The analysis was done on 1GHz Athlon, 640MB RAM, Windows 98 2nd edition, JDK 1.4.2