Selective Tracing for Dynamic Analyses*

Matthias Meyer, Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[mm|lowende]@uni-paderborn.de

Abstract

Reverse engineering based on dynamic analyses often uses method traces of the program under analysis. 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.

In this paper, we present an approach to collect runtime information by selectively recording method calls during a program's execution. Only relevant classes and methods are monitored to reduce the amount of information. We developed the JavaTracer *which we use for the recording of method calls in Java programs.*

1. Introduction

In the last years, we developed a tool-supported semiautomatic approach to design recovery [5]. Our approach facilitates the recognition of design pattern [3] instances in the source code of a system. We recently extended this approach by combining the existing static analysis with a dynamic analysis [7]. The static analysis identifies pattern instance candidates based on their structural properties. The subsequent dynamic analysis confirms or rejects the candidates by checking their behavior.

The behavior of a design pattern is specified by UML 2.0 sequence diagrams [8]. In our approach, these specifications are called behavioral patterns. Behavioral patterns describe typical sequences of method calls between objects of classes that participate in a design pattern instance. To check the conformance of a given design pattern instance to the behavioral pattern, method traces have to be gathered during the execution of the program under analysis.

Recording all method traces during a program's execution not only produces too much information, but also reduces the runtime performance of the program significantly. Consequently, the tracing should be restricted to those method calls that are really needed in the dynamic analysis. In our approach, only specific methods of pattern instance candidates have to be monitored, which means only to record a "slice" of method calls of the whole program.

For this purpose, we developed a selective tracer which takes a list of classes and methods to be monitored as input. The tracer executes the program to be analyzed and records only calls to the given methods. The gathered information is saved to a file which can be used by post-mortem analyses.

*This work is part of the Finite project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-2.

In the next section we present the application scenario for our selective tracer in more detail by means of a concrete example. We will refer to this example throughout the rest of the paper. In Section 3 we report about related work. Our approach to selective tracing is described in detail in Section 4 whereas its good performance is shown in Section 5. The paper is concluded with future work in Section 6.

2. Application Scenario

In a case study of our design recovery approach, we analyzed the Eclipse platform [2]. Among others, our static analysis identified several candidates of the *Strategy* design pattern in the source code.

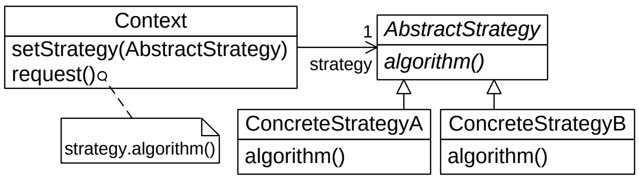


Figure 1: The *Strategy* Design Pattern

A *Strategy* design pattern (Figure 1) lets an algorithm vary independently from the client that uses it. An abstract class defines the algorithm interface, which is implemented by different concrete classes (the strategies). A context class references a strategy and delegates requests received from its clients to the strategy. Usually, the clients configure a context object with the appropriate concrete strategy.

Classes	Methods
org.eclipse.swt.widgets.Composite	setLayout WM_SIZE
org.eclipse.swt.widgets.Layout	*layout*
org.eclipse.jface.viewers.StructuredViewer	addFilter filter getSortedChildren setSorter
org.eclipse.jface.viewers.ViewerSorter	sort
org.eclipse.jface.viewers.ViewerFilter	select

Table 1: Classes and Methods Identified as Parts of Pattern Candidates.

Table 1 shows the classes and methods[1] that have been identified as parts of three *Strategy* pattern candidates. The first candidate consists of the classes Composite and Layout (cf. Table 1) which were recognized as context and abstract

[1]Abstract classes and methods are written in italic.

strategy, respectively. The method setLayout was identified as the method to configure the context with a strategy and WM_SIZE is called by clients to place a request. The method layout of class Layout was recognized as the method implementing the actual algorithm. The other classes and methods listed in the table belong to other candidates.

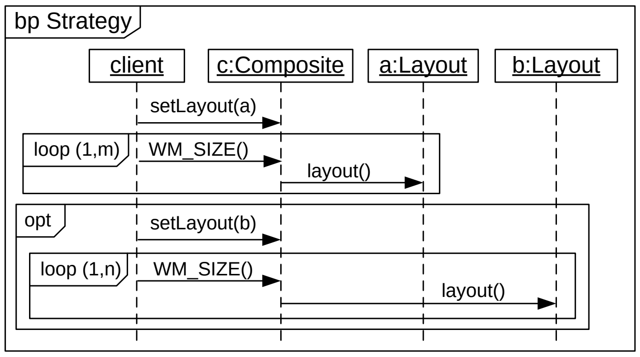


Figure 2: Behavioral Pattern for a Concrete *Strategy* Candidate.

The dynamic analysis now has to check whether the interaction of instances of the candidate's classes conforms to the behavioral pattern of a *Strategy* design pattern, i.e. the identified methods are called in the specified sequence.

Figure 2 shows the behavioral pattern of *Strategy* in which the methods and object types have been replaced by the classes and methods of the first pattern candidate. The behavioral pattern requires that a context object c of type Composite is configured with a strategy object a of type Layout by calling setLayout. Afterwards, a client has to place at least one request which has to be delegated to the strategy, i.e. WM_SIZE and layout have to be called consecutively an arbitrary number of times (indicated by the loop fragment)[2]. Furthermore, after several requests have been handled, the concrete strategy may be changed by another call to setLayout with a different b:Layout object. After that, requests on the context c:Composite have to be delegated to the new strategy object by calling layout on b:Layout. However, the change of the strategy is not required and is thus enclosed by an optional fragment.

In order to check if the pattern candidate behaves as specified by the concrete behavioral pattern shown in Figure 2, we need to record method call traces at runtime. However, a behavioral pattern does not define a complete trace. Only significant method calls are specified. Other calls of methods that are not mentioned in the behavioral pattern may interleave the given sequence. Consequently, we do not need to record a complete program trace but only calls to those methods explicitly mentioned by the pattern.

Furthermore, since some of the classes and methods identified in the source code are abstract, e.g. Layout and its layout method, they cannot be monitored directly during runtime. Instead, classes and methods that implement the abstract classes and methods must be monitored. Due to polymorphism and dynamic method binding, the same holds for methods which override methods to be monitored. The concrete classes and methods could be determined by static analysis easily. In our approach, however, this is done by our selective tracer as well.

[2]Since no methods are called on the client object, its class needs not to be determined and can be ignored during analysis.

3. Related Work

The Java Debug Interface (JDI) [6] offers debuggers a native technique to receive *MethodEntry-* and *MethodExitEvents*. The debugger has to provide a filter which specifies the classes to be monitored. This approach can not be used to monitor specific methods. Instead, all methods of classes given in the filter are monitored during the execution of the program under analysis. For each method call, *MethodEntry-* and *MethodExitEvents* are sent to the debugger. This technique is not practicable, since it slows down the analyzed program significantly (cf. Section 5).

The Omniscient Debugger [4] records method calls and variable state changes of Java programs. It instruments the source code on the byte code level, i.e. additional code is inserted into the original source code of the program to be analyzed. The code is used to inform the debugger about method calls. The instrumentation is also done in a non-selective way. The author reports about 100MB/sec of data produced during the execution.

The Instrumentation, Execution, and Coverage Tool InsECT [1] allows for collecting different kinds of dynamic information including method traces by instrumenting and executing the program under analysis. Instrumentation tasks are used to specify which entities of the program are to be instrumented and which kind of information is to be collected. Monitors can be implemented to process the collected information. In [1] it is shown that InsECT is efficient.

However, a problem of instrumentation is that it strongly depends on the programming language and the runtime environment used. This approach is difficult to transfer to other languages, especially those that do not use intermediate code such as C or C++. Instrumentation may also affect the synchronization of concurrent threads, since instrumented code directly influences the runtime of threads. This may cause for example time outs in the synchronization, thus resulting in a completely different behavior of the analyzed program.

4. Selective Tracing

We developed the JavaTracer [9] for selective tracing of Java programs. As input, it gets a list of classes and interfaces as well as methods that have to be monitored during the execution of the program under analysis. The JavaTracer acts as a debugger and executes the program, called the debuggee. JDI is used for connecting to the debuggee's virtual machine.

The principle idea of selective tracing is rather simple. The JavaTracer is informed by the virtual machine each time a class is loaded. If this class belongs to the classes in the input, it adds a breakpoint at the beginning and the end of the body[3] of each method given in the input, indicating when a method is called and when it returns.

Abstract methods declared by interfaces or abstract classes can also be monitored, even though they don't have a method body. The JavaTracer determines each time a

[3]The Java VM creates a virtual code line at the end of each method body that will be passed regardless of the actual executed return statement.

class is loaded if it is a sub class of the classes given as input. If the loaded class is a sub class, it adds breakpoints to methods which implement or override one of the given methods, thus supporting analyses that include polymorphism and dynamic method binding.

The advantage of this simple idea is that the approach is not bound to Java even though the JavaTracer is implemented for Java programs only. Breakpoints are a common feature of debuggers for nearly all languages. The JavaTracer just needs another implementation for the interface that is used to set breakpoints and receive breakpoint events to adapt to another debugger.

The JavaTracer will be informed when a breakpoint is reached during the program's execution. It then halts the debuggee. This guarantees that all threads of the program are halted, not only the thread that is currently running. Thus, concurrent threads depending on the current thread are not affected by halting just the current thread, since they are halted, too.

In the case of a breakpoint event at the beginning of a method call, the JavaTracer 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 and values of objects passed as arguments as well as the current thread. Then the debuggee's execution is continued. This information is recorded as a method entry event. Breakpoint events at the end of a method call are recorded as method exit events. Events about loaded classes are recorded as well.

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

Input for Tracing

The JavaTracer is started with a trace definition document describing the classes and methods that have to be monitored during the program's execution. Figure 3 shows an excerpt of this document using the example of Table 1.

```
<TraceDefinition>
  <ConsiderTrace>
    <Class name="org.eclipse.swt.widgets.Composite">
      <Method name="setLayout"/>
      <Method name="WM_SIZE">
        <Parameter type="int"/>
        <Parameter type="int"/>
      </Method>
    </Class>
    <Class name="org.eclipse.swt.widgets.Layout">
      <Method name="layout">
        <Parameter
          type="org.eclipse.swt.widgets.Composite"/>
        <Parameter type="boolean"/>
      </Method>
    </Class>
    ...
  </ConsiderTrace>
  <CriticalTrace>
  ...
  </CriticalTrace>
</TraceDefinition>
```

Figure 3: Example of the JavaTracer's Input

The trace definition has two sections. Within the ConsiderTrace section, classes are listed for which only selected methods are monitored. That means, only the given methods and overriding methods are considered in the tracing, calls of other methods are ignored.

The JavaTracer also provides a tracing on the class level, the so-called *critical* monitoring of classes. Using critical tracing, all methods of a class are monitored. This facilitates analyses where all method calls on objects of specific classes have to be recorded. These classes are specified within the CriticalTrace section of the input.

Output of Tracing

Figure 4 shows an excerpt of the JavaTracer's output. The output consists of a list of class loading events as well as method entry and exit events in the order of their occurrence.

```
<TraceResult>
  <ProcessStart name="main" time="1127705886787"/>

  <ClassLoaded name="org.eclipse.swt.widgets.Composite">
  </ClassLoaded>

  <ClassLoaded name="org.eclipse.swt.widgets.Shell">
    <SuperType name="org.eclipse.swt.widgets.
                     Composite"/>
  </ClassLoaded>

  <ClassLoaded name="org.eclipse.swt.widgets.Layout">
  </ClassLoaded>

  <ClassLoaded name="org.eclipse.swt.layout.GridLayout">
    <SuperType name="org.eclipse.swt.widgets.Layout"/>
  </ClassLoaded>
  ...
  <MethodEntry id="22" name="WM_SIZE" thread="main"
               time="1127705893547">
    <Caller id="1515"
            type="org.eclipse.swt.widgets.Shell"/>
    <Callee id="1515"
            type="org.eclipse.swt.widgets.Shell"/>
    <Argument value="0" type="int"/>
    <Argument value="3473906" type="int"/>
  </MethodEntry>

  <MethodEntry id="23" name="layout" thread="main"
               time="1127705893557">
    <Caller id="1515"
            type="org.eclipse.swt.widgets.Shell"/>
    <Callee id="1516"
            type="org.eclipse.swt.layout.GridLayout"/>
    <Argument id="1515"
              type="org.eclipse.swt.widgets.Composite"/>
    <Argument value="false" type="boolean"/>
  </MethodEntry>
  ...

  <MethodExit id="23" time="1127705893617"/>
  <MethodExit id="22" time="1127705893627"/>
  ...
  <ProcessEnd time="1127705926565"/>
</TraceResult>
```

Figure 4: Example of the JavaTracer's Output

The class loaded events comprise not only the class that was actually loaded, but also its super class, if the super class was given in the input. This information is needed in dynamic analysis to identify where polymorphism and dynamic method binding was used.

The two pairs of method entry and exit events describe two method calls. The first method call WM_SIZE (id 22) was called by an object of org.eclipse.swt.widgets.Shell on itself. The second method call with id 23 is nested in the first one which means that the method layout is called within the first method WM_SIZE on an object of type org.eclipse.swt.layout.GridLayout.

The output of the JavaTracer can be optimized for the analysis it is used for. Some information can be omitted such as the time stamps or even method exit events if information about method stack traces are not needed. Since tracing can produce huge amounts of information, it is vital to cut down the recording to a minimum.

The JavaTracer

Figure 5 depicts a screen shot of the JavaTracer Eclipse plug-in. We made this screen shot during the monitoring of Eclipse in the application scenario. On the right hand, the currently used trace definition document is displayed. In the upper left corner, the *Execution Monitor* view shows a tree of classes and methods that are monitored. For each method, the number of executions is given and an icon indicates if the method was executed at all. In the lower left corner, the *JavaTracer* view displays events occurred during the monitoring, whereas the *Console* view displays the output of the monitored program.

5. Performance

We measured the performance of our approach by comparing the startup times of Eclipse with and without tracing. Without tracing or instrumentation, it is very difficult to measure the startup time due to the lack of well-defined measuring points. Since we only want to make a qualitative statement of the performance, we decided to measure the time manually. The time was stopped when the CPU-load of the Eclipse process dropped to 0%. We run the scenarios ten times and calculated the average duration.

The performance was measured on a Pentium 4-M machine with 1.8 GHz and 1024 MB RAM. The system was running Windows XP Professional SP2 and Java 2 Standard Edition 5.0 Update 4. All other processes were stopped as far as possible. The workspace of the Eclipse platform consisted of one Java project, which was initially loaded during the startup of Eclipse.

Scenario	#c	#m	$\#act_c$	$\#act_m$	#mc
1	5	9	59	107	2945
2	8	13	204	336	12314

Table 2: Performance Measuring Scenarios

Table 2 shows two different scenarios. In the first scenario, we monitored the 5 classes (#c) and 9 methods (#m) given in the example. The actual number of monitored subjects were 59 classes ($\#act_c$) and 107 methods ($\#act_m$) due to implementations of abstract classes and methods as well as polymorphism. During the startup of Eclipse, there were 2945 method calls (#mc) of the 107 methods recorded.

The second scenario comprised 8 classes/interfaces and 13 methods to be monitored. All classes of the first scenario plus additional classes and interfaces that play a central role in the Eclipse environment are monitored. The additional classes are org.eclipse.core.runtime.Plugin, org.eclipse.core.runtime.IAdaptable and org.eclipse.core.runtime.IAdapterFactory. These classes and interfaces are extended or implemented by multiple other classes. This resulted in a scenario where 204 classes and 336 methods were actually monitored. We used this second scenario to show the scalability of our approach.

Scenario	$t_{w/o}$	t_{break}	t_{events}
1	16 sec.	41 sec.	36 min.
2	16 sec.	65 sec.	?

Table 3: Duration of Program Tracings

In Table 3, we present the average startup time for each scenario. First, the program was executed without any tracing ($t_{w/o}$). Then, the program was monitored using our breakpoint events (t_{break}) and at last (t_{events}) by using the native tracing technique offered by the Java Debug Interface (JDI) [6]. This technique is limited to monitor all methods of a class. To compare the native tracing of JDI to our approach, we recorded only entry and exit events of those methods given in the input.

The startup times without any tracing are of course equal for both scenarios. The performance results show that our approach to selectively trace method calls is feasible. Even though the number of monitored methods is three times higher than in the first scenario and the number of method calls is four times higher, the startup time rises by less than 60%.

In comparison to our approach, the event based approach offered by JDI is not practicable. We abandoned the performance analysis of the event based approach for the second scenario, since it took too much time.

Although the XML output format may seem too verbose, it has only a very slight influence on the performance of the JavaTracer. We analyzed the JavaTracer with a profiler discovering that more than 90% of the time spent in tracing is consumed by the JDI interface.

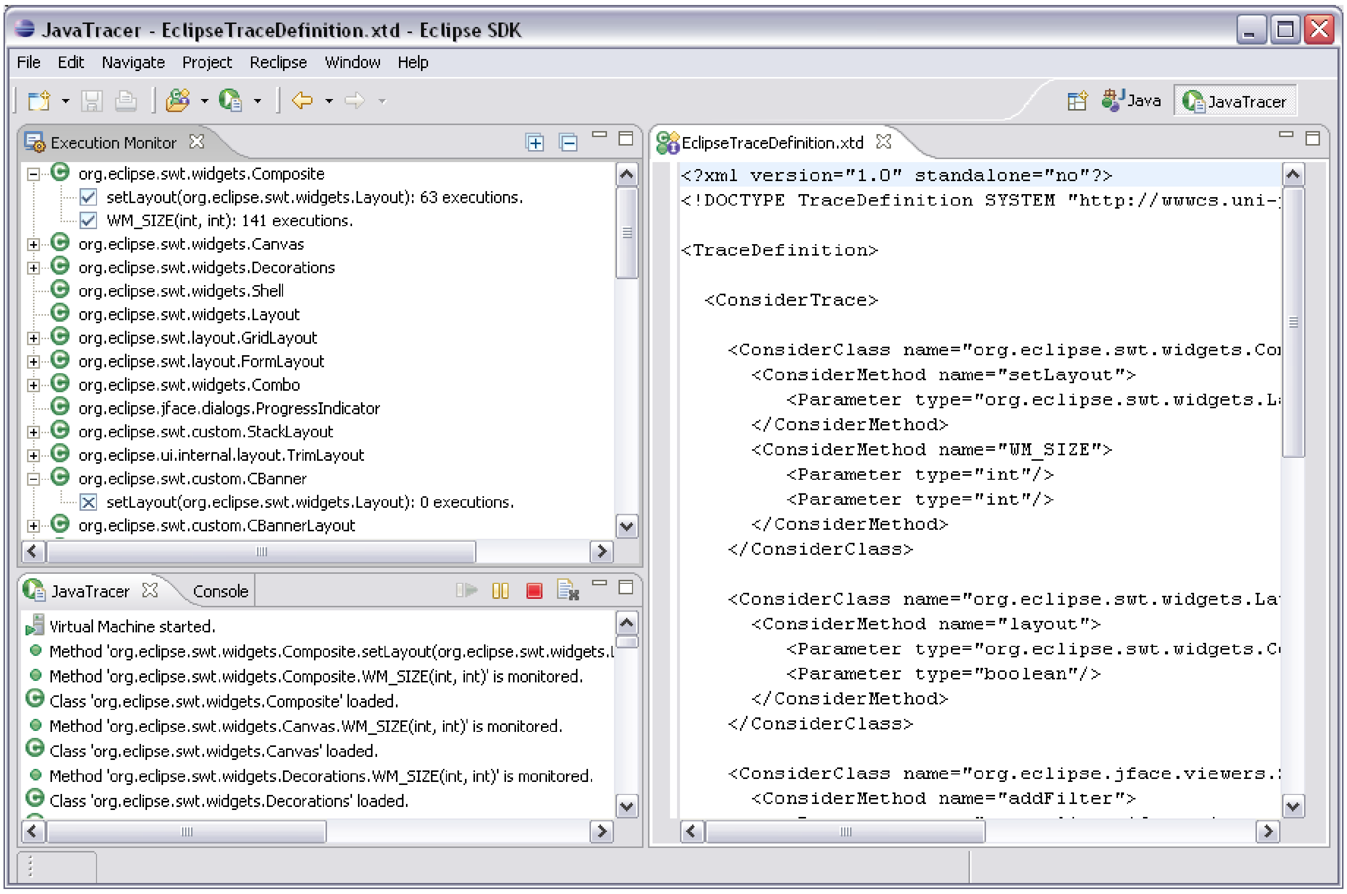
6. Future Work

We are planning to use our behavioral pattern analysis for conformance checking. When designing components, behavioral patterns can be used to describe protocols on how to use the interface of the component. In an ideal *Model Driven Development* process, the source code is completely generated from the model. In practice, a hybrid development process is often used, where parts of a system are generated and parts are implemented manually. During the implementation and testing of the components, our dynamic analysis can check if the actual behavior of the components conforms to the behavior defined by the behavioral patterns.

References

[1] A. Chawla and A. Orso. A Generic Instrumentation Framework for Collecting Dynamic Information. *SIGSOFT Software Engineering Notes, Section: Workshop on Empirical Research in Software Testing. ACM Press, New York, NY, USA*, 29(5):1–4, September 2004.

[2] Eclipse Foundation. *The Eclipse Platform. Online at http://www.eclipse.org.* Last visited: September 2005.


JavaTracer - EclipseTraceDefinition.xtd - Eclipse SDK
File Edit Navigate Project Reclipse Window Help
Java
JavaTracer
Execution Monitor
EclipseTraceDefinition.xtd
JavaTracer
Console

```
org.eclipse.swt.widgets.Composite
    setLayout(org.eclipse.swt.widgets.Layout): 63 executions.
    WM_SIZE(int, int): 141 executions.
org.eclipse.swt.widgets.Canvas
org.eclipse.swt.widgets.Decorations
org.eclipse.swt.widgets.Shell
org.eclipse.swt.widgets.Layout
org.eclipse.swt.layout.GridLayout
org.eclipse.swt.layout.FormLayout
org.eclipse.swt.widgets.Combo
org.eclipse.jface.dialogs.ProgressIndicator
org.eclipse.swt.custom.StackLayout
org.eclipse.ui.internal.layout.TrimLayout
org.eclipse.swt.custom.CBanner
    setLayout(org.eclipse.swt.widgets.Layout): 0 executions.
org.eclipse.swt.custom.CBannerLayout
```

```
Virtual Machine started.
Method 'org.eclipse.swt.widgets.Composite.setLayout(org.eclipse.swt.widgets.L
Method 'org.eclipse.swt.widgets.Composite.WM_SIZE(int, int)' is monitored.
Class 'org.eclipse.swt.widgets.Composite' loaded.
Method 'org.eclipse.swt.widgets.Canvas.WM_SIZE(int, int)' is monitored.
Class 'org.eclipse.swt.widgets.Canvas' loaded.
Method 'org.eclipse.swt.widgets.Decorations.WM_SIZE(int, int)' is monitored.
Class 'org.eclipse.swt.widgets.Decorations' loaded.
```

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE TraceDefinition SYSTEM "http://wwwcs.uni-

<TraceDefinition>

  <ConsiderTrace>

    <ConsiderClass name="org.eclipse.swt.widgets.Co
      <ConsiderMethod name="setLayout">
         <Parameter type="org.eclipse.swt.widgets.L
      </ConsiderMethod>
      <ConsiderMethod name="WM_SIZE">
         <Parameter type="int"/>
         <Parameter type="int"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="org.eclipse.swt.widgets.La
      <ConsiderMethod name="layout">
         <Parameter type="org.eclipse.swt.widgets.C
         <Parameter type="boolean"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="org.eclipse.jface.viewers.
      <ConsiderMethod name="addFilter">
```

Figure 5: The JavaTracer implemented as an Eclipse Plug-In

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[4] B. Lewis. Recording Events to Analyze Programs. In *Object-Oriented Technology. ECOOP 2003 Workshop Reader*. Lecture notes on computer science (LNCS 3013), Springer, July 2003.

[5] 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.

[6] Sun Microsystems. *Java Platform Debugger Architecture(JPDA). Online at http://java.sun.com/products/jpda/index.jsp*. Last visited: September 2005.

[7] 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.

[8] 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.

[9] L. Wendehals. Tool Demonstration: Selective Tracer for Java Programs. In *Proc. of the 12th Working Conference on Reverse Engineering, Pittsburgh, Pennsylvania, USA*, November 2005. to appear.