Selective Tracing of Java Programs*

Lothar Wendehals, Matthias Meyer, Andreas Elsner
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[lowende|mm|trinet]@upb.de

Abstract

Design recovery, which means extracting design documents from source code, is usually done by static analysis techniques. Analysing behaviour by static analysis is very imprecise. We combine static and dynamic analysis to increase the preciseness of our design recovery process. In this paper we present an approach to collect data for the dynamic analysis by recording method calls during a program's execution. To reduce the amount of information we monitor only relevant classes and methods identified by static analysis.

1. Motivation

Today software engineers spend most of their time maintaining software systems. The documentation of such systems is often not available or has become obsolete. Before a system can be changed to meet new requirements it has to be reverse engineered and its design has to be recovered which is a time consuming and expensive task.

We developed a tool-supported semiautomatic approach to design recovery [4] within the FUJABA TOOL SUITE [6]. The approach facilitates the recognition of patterns such as design patterns [1] in the source code of a system. It is a highly scaleable process which can be applied to large real world applications.

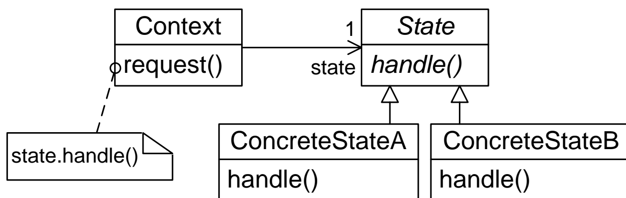

Figure 1: The *State* design pattern

So far we only perform a static analysis based on source code that focuses mainly on the structural aspects of a pattern. However, many patterns are structurally very similar and differ only in their behaviour, e.g. the design patterns *State* (cf. Figure 1) and *Strategy* [8, 1]. Those behavioural differences can only be recognized during a dynamic analysis of the system. Therefore, we will combine our static analysis with a subsequent dynamic analysis [7, 8].

*This work is part of the FINITE project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-1.

As the basis for dynamic analysis a program trace will be recorded during the execution of the program to be analysed. Since the amount of information for a complete program trace is too high, we record only relevant method traces. The relevant classes and methods to be monitored are identified by the static analysis.

In the next section we present an overview of our design recovery process. The selective tracing of Java programs is described in Section 3. Related work follows in Section 4. In Section 5 we report about the performance of our approach. A short summary of future work follows in Section 6.

2. The Design Recovery Process

Our design recovery process is based on an extended Abstract Syntax Graph (ASG) representation of the source code. The ASG includes method bodies for a rudimentary static analysis of behaviour. During design recovery the ASG will be annotated by nodes which are linked to an arbitrary number of ASG nodes to mark recognized pattern instances.

A tool-based pattern recovery requires a formal definition of patterns. Thus, for each pattern to be recognized within the source code a structural and a behavioural pattern is given. The process starts with the static analysis using the structural patterns. During this phase pattern instance candidates are recognized. These candidates will be verified by the subsequent dynamic analysis using the behavioural patterns.

2.1 Static Analysis

The structural patterns are specified as graph grammar rules with respect to the ASG [4]. Graph grammar rules consist of a left-hand side (LHS) and a right-hand side (RHS). The LHS describes a sub graph to be found within the host graph. The RHS describes the modifications of the sub graph when the rule is applied.

Figure 2 depicts a structural pattern for the *State* design pattern. The LHS and RHS of the graph grammar rule are defined by one graph. The LHS is defined by all black nodes and edges and describes the sub graph to be found within the ASG. The RHS consists of the LHS and additional nodes and edges marked with the stereotype ≪create≫. It describes how to mark the found sub graph by creating an annotation node and links to ASG nodes.

The *State* pattern (cf. Figure 1) enables an object to change its behaviour at runtime by changing its internal state [1]. Each state is represented by a separate class which

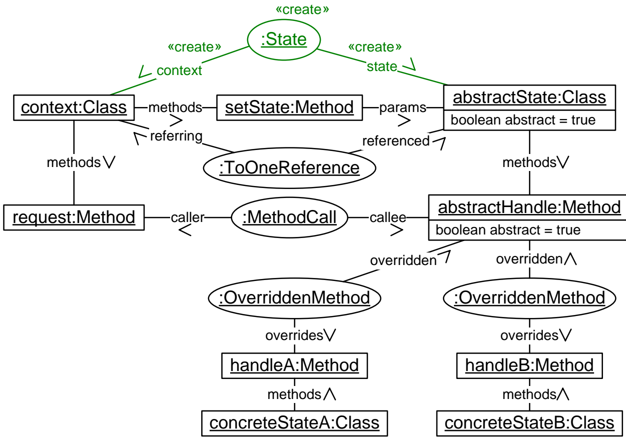

Figure 2: Structural pattern for *State*

encapsulates the state-specific behaviour. The state classes adhere to a common interface defined by an abstract super class. The object references exactly one state object and delegates requests to this state object.

This structure is described by the LHS in Figure 2. It specifies that the ASG must contain a class context:Class which references an abstract class abstractState:Class. This is expressed by the oval annotation node of type ToOneReference. Note, that the LHS may also contain annotation nodes created by the application of other rules. This enables the composition of structural patterns.

In addition, the class context is required to have a method setState:Method which has a parameter of type abstractState:Class and another method request:Method which calls (MethodCall [1]) an abstract method abstractHandle:Method of class abstractState:Class. Furthermore, the abstract method abstractHandle:Method has to be overridden by at least two concrete methods (handleA:Method and handleB:Method) in two subclasses of class abstractState:Class, namely concreteStateA:Class and concreteStateB:Class.

If the rule can be applied, i.e. the sub graph can be found, it creates a State annotation node and links it to the context:Class and abstractState:Class classes. The mapping between nodes of the LHS and the found sub graph nodes is stored for dynamic analysis.

The application of the graph grammar rules for the structural patterns recovers pattern instance candidates. For details on the rule application see [4].

2.2 Dynamic Analysis

The purpose of the dynamic analysis is to verify the pattern instance candidates recognized by the preceding static analysis. It has to be checked if the collaboration of the candidate's classes during runtime matches the pattern's behavioural description.

For the specification of behavioural patterns we use a notation based on UML 2.0 sequence diagrams [8]. As an example, Figure 3 shows the behavioural pattern for the *State* design pattern. The pattern requires the existence of four objects, namely client, context, concreteStateA, and concreteStateB. The pattern describes two alternative sequences.

[1]Polymorphism and dynamic method binding prevent a precise static analysis of method calls.

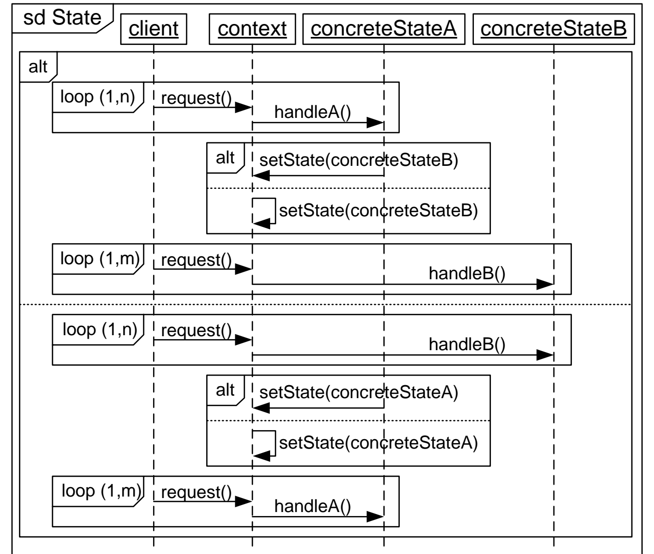

Figure 3: Behavioural pattern for *State*

In the first sequence the client object calls the method request on the context object which in turn calls handleA on object concreteStateA. This interaction fragment must occur at least once but may occur an arbitrary number of times which is specified by loop(1,n). Then either the concreteStateA or the context itself has to change the state by calling the setState method with concreteStateB as argument. After the state change the client has to call request on context at least once again. This time the behaviour of context must be handled by the state concreteStateB. This specification conforms to the behavioural description of the *State* pattern [1]. In principle the second alternative specifies the same behaviour as the first one except that the context is in state concreteStateB first and then changes to concreteStateA.

Note that between the specified method calls an arbitrary number of other methods may be called as long as they are not already mentioned by the pattern. However, the method calls specified by the pattern have to occur in exactly the specified sequence. This conforms to the semantics of the UML 2.0 *consider* interaction operator which implicitly holds for all behavioural patterns. To facilitate a more restrictive specification we also support the *critical* operator which may be assigned to interaction fragments to prohibit method calls which are not specified explicitly.

To verify the conformance of a pattern candidate to its corresponding behavioural pattern we consider method traces recorded during the execution of the program.

3. Selective Tracing

As mentioned before, recording all method traces during a program's execution produces too much information. Furthermore, the monitoring of a complete program extremely reduces the runtime performance. For most analyses a "slice" of all method traces is sufficient. In this approach the static analysis provides a set of pattern instance candidates that has to be further analysed by dynamic analysis. All other classes of the program can be ignored.

Figure 4 shows an example for a *State* candidate. It has been recovered and annotated by the static analysis. For the

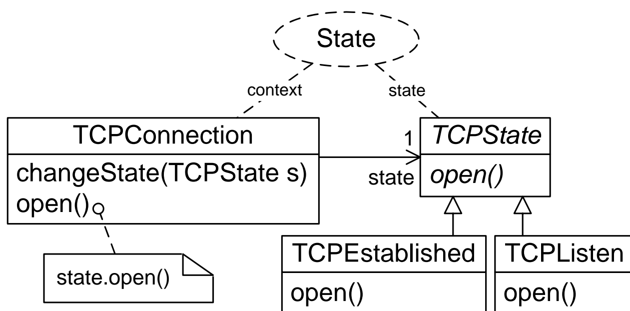

Figure 4: Example of a *State* instance

dynamic analysis the method traces for the candidate have to be recorded. This is done by the JavaTracer. It can be used as a stand-alone tool or as a plug-in for the Fujaba Tool Suite.

Input for JavaTracer

The input for the JavaTracer is given as an XML document. Within this document the candidate's classes and some of their methods are listed that have to be monitored during program execution. The information is retrieved from the candidate and the structural and behavioural patterns.

The classes to be monitored can be gathered from the behavioural pattern in Figure 3. There are three objects within the sequence diagram on which methods are called, namely context, concreteStateA and concreteStateB. The names of these three objects refer to the nodes context:Class, concreteStateA:Class and concreteStateB:Class within the structural pattern in Figure 2. During static analysis the nodes from the structural pattern have been mapped to the nodes of the candidate in Figure 4. By using this mapping we can extract the classes from the candidate that have to be monitored, namely TCPConnection, TCPEstablished and TCPListen.

The methods can be extracted on the same way. In Figure 3 the four different methods request, setState, handleA and handleB are called. They refer to request:Method, setState:Method, handleA:Method and handleB:Method from the structural pattern. They have been mapped to the methods TCPConnection.open(), TCPConnection.changeState(TCPState s), TCPEstablished.open() and TCPListen. open().

The JavaTracer can also restrict the recording of method calls to a given caller. The handleA and handleB methods in Figure 3 are called by the context object. So the caller for the TCPEstablished.open() and TCPListen.open() methods is the TCPConnection class. The method setState in the behavioural pattern is called by three different objects. So for the method TCPConnection.changeState() the three caller classes TCPConnection, TCPEstablished and TCPListen have to be monitored.

Figure 5 shows an excerpt of the input for the JavaTracer. The candidate's classes given in the input will be monitored using the *consider* semantics, i.e. only the given methods will be monitored, method calls of other methods will be ignored. These classes are listed within the ConsiderTrace section of the input.

The JavaTracer also provides *critical* monitoring of classes where all methods of a class are monitored. This facilitates the checking of critical method call sequences. The classes are specified within an CriticalTrace section of the input.

```
<Trace>
  ...
  <ConsiderTrace>
    <Class name="TCPConnection">
      <Method name="open"/>
      <Method name="changeState">
        <Parameter type="TCPState"/>
        <Caller name="TCPConnection"/>
        <Caller name="TCPEstablished"/>
        <Caller name="TCPListen"/>
      </Method>
    </Class>
    <Class name="TCPEstablished">
      <Method name="open">
        <Caller name="TCPConnection"/>
      </Method>
    </Class>
    <Class name="TCPListen">
      <Method name="open">
        <Caller name="TCPConnection"/>
      </Method>
    </Class>
  </ConsiderTrace>
  ...
</Trace>
```

Figure 5: Example of JavaTracer input

Tracing

The JavaTracer acts as a debugger and executes the program to be analysed, called the debuggee. It uses the Java Debugging Interface (JDI) [5] for connecting to the debuggee's virtual machine. For each method given in the input two breakpoints are set at the beginning and the end of the method body. The JavaTracer is informed, when a breakpoint is reached during program execution.

This 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. So in principle a selective tracer for different languages can be implemented in the same way.

When the debuggee reaches a breakpoint the JavaTracer will be informed. The JavaTracer 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 execution of the program is controlled either manually by the reengineer or by automated tests. The JavaTracer informs the reengineer which classes have been loaded and which methods have been executed.

Output of JavaTracer

Figure 6 shows an excerpt from the JavaTracer's output. The output consists of a list of method entry and exit events in the order of their occurrence.

The three trace events describe a call of method open on an object of class TCPConnection. This method calls another method open on an object of class TCPEstablished. The last

```
<TraceResult>
  ...
  <TraceEvent time="1089792972829">
    <Callee id="3">
      <Object objectName="TCPConnection"
       uniqueID="42" owningThread="main"/>
      <Method methodName="open"/>
    </Callee>
  </TraceEvent>
  <TraceEvent time="1089792972830">
    <Callee id="15">
      <Object objectName="TCPEstablished"
       uniqueID="48" owningThread="main"/>
      <Method methodName="open"/>
    </Callee>
    <Caller>
      <Object objectName="TCPConnection"
       uniqueID="42" owningThread="main"/>
      <Method methodName="open"/>
    </Caller>
  </TraceEvent>
  <TraceEvent time="1089792972845">
    <MethodExit id="15">
      <Method methodName="open"/>
    </MethodExit>
  </TraceEvent>
  ...
</TraceResult>
```

Figure 6: Example of JavaTracer output

method call immediately returns. These three events cover the first loop within the behavioural pattern of Figure 3.

4. Related Work

The JaVis environment [3] visualizes and debugs concurrent Java programs to detect deadlocks. The information about a running program is collected by tracing, which is implemented using the JDI [5]. However, this approach uses another technique of the JDI. The debugger has to provide a filter, which specifies the classes and methods to be monitored. During the debuggee's execution all classes and all methods are monitored. For methods passing the filter *MethodEntry-* and *MethodExitEvents* are sent to the debugger. Since all methods are monitored this technique can slow down the debuggee up to 10.000 times.

The Omniscient Debugger [2] 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 debuggee. 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 information produced during the execution as the main problem of this approach.

5. Performance

Table 1 shows the performance of different executions of the FUJABA TOOL SUITE. In the first case the duration of starting FUJABA was measured[2]. In the second and the third case a project was opened in FUJABA. The first project consists of one class diagram with 12 classes, the second one of one class diagram with 27 classes and 178 activity diagrams. Four major classes were monitored.

Action	$t_{w/o}$	t_{break}	t_{events}
Starting Fujaba	8 sec.	10 sec.	81 sec.
Open Project I	4 sec.	23 sec.	416 sec.
Open Project II	9 sec.	100 sec.	1267 sec.

Table 1: Duration of program tracings

First, the program was executed without any tracing ($t_{w/o}$). Then, the program was monitored using the breakpoint approach (t_{break}) and at last by filtering the *MethodEntry-* and *MethodExitEvents* (t_{events}). The table shows that selective tracing with breakpoints improves the performance significantly compared to the event based approach.

6. Future Work

The behavioural pattern recognition has not been implemented yet, but basically the same techniques as in static analysis can be used. The behavioural patterns will be translated into graph grammar rules. The output from the JAVATRACER will be transformed into a method call graph for each candidate. If the graph grammar rule for the behavioural pattern can be applied to the call graph, the candidate can be verified as a correct design pattern instance.

References

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

[2] 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.

[3] K. Mehner. *JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs*, pages 163–175. LNCS 2269. Springer Verlag, May 2001.

[4] J. Niere, W. Schäfer, J. 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.

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

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

[7] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003.

[8] L. Wendehals. Specifying patterns for dynamic pattern instance recognition with UML 2.0 sequence diagrams. In *Proc. of the* 6th *Workshop Software Reengineering (WSR), Bad Honnef, Germany*, May 2004. to appear.

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