Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Selektive Aufzeichnung von Laufzeitinformationen bei der Ausführung von Java-Programmen

Studienarbeit
zur Erlangung des Grades
Bachelor of Computer Science
im Rahmen des Studiengangs Informatik

Andreas Elsner
Grunigerstr. 12
33102 Paderborn

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Hans Kleine Büning

Paderborn, September 2004

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegeben Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Inhalt

Kapitel 1
Einleitung 5
1.1 Engineering Prozesse 5
1.2 Reverse Engineering mit Fujaba 6
1.2.1 Statische Analyse 7
1.2.2 Dynamische Analyse 9
1.3 Zielsetzung 10
1.4 Aufbau der Arbeit 11

Kapitel 2
Grundlagen 13
2.1 Instrumentierung 14
2.1.1 Java Bytecode-Instrumentierung am Beispiel jclasslib 16
2.1.2 Omniscient Debugger – ODB 16
2.2 Steuerung der Laufzeitumgebung 17
2.2.1 Profiling mit JVMPI 19
2.2.2 JPDA 20
2.2.3 JPDA am Beispiel JaVis 23

Kapitel 3
Konzepte 25
3.1 Statische Analyse 25
3.2 Dynamische Analyse 27
3.3 Selektive Aufzeichnung von Laufzeitinformationen 30
3.3.1 Trace-Types 30
3.3.2 Eingabe 31
3.3.3 Funktion 32
3.3.4 Ausgabe 36

Kapitel 4
Technische Realisierung 39
4.1 Verwendete Technologien 39
4.1.1 Java Platform Debugger Architecture 39
4.1.2 Effiziente Überwachung 41
4.2 Aufbau 43

4.2.1 Einlesen der Eingabe 45
4.2.2 Laufzeitüberwachung 46
4.2.3 Ausgabe 48
4.3 Document Type Definitions 50

Kapitel 5
Beispielsitzung 53

Kapitel 6
Zusammenfassung 59

Der Autor macht darauf aufmerksam, dass die genannten Firmen- und Markennamen sowie Produktbezeichnungen in der Regel marken-, patent- oder warenzeichenrechtlichem Schutz unterliegen Die Erwähnung von oder Bezugnahme auf diese stellt keine Verletzung des Copyrights dar.

Kapitel 1
Einleitung

1.1 Engineering Prozesse

Ein wichtiger Schwerpunkt der Softwareentwicklung liegt neben dem Entwurf neuer Software in der Pflege bereits vorhandener Systeme. Nicht selten finden Pflege und Wartung über lange Zeiträume hinweg statt oder werden durch eine Änderung der Anforderungen an das System nach längerer Einsatzzeit notwendig. In vielen Fällen können dann die ursprünglichen Entwickler nicht mehr zu Rate gezogen werden. Eine fehlende oder unzureichende Dokumentation des Systems durch Entwurfsunterlagen und Quellcodekommentierung oder eine obsolete Dokumentation, die bei vorangegangenen Arbeiten an der Software nicht auf den aktuellen Stand gebracht wurde, erfordert dann, die zur Weiterentwicklung benötigten Informationen direkt aus dem Quellcode zu gewinnen.

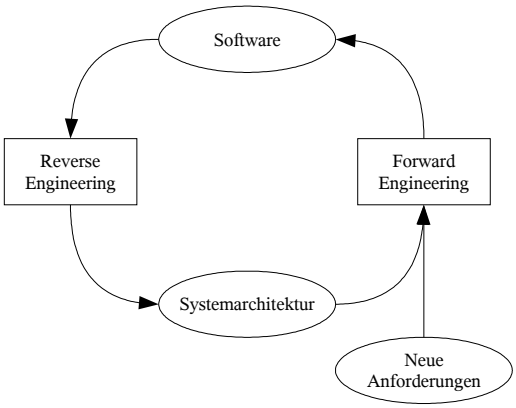

Abbildung 1.1: Reengineering Zyklus

Der Prozess der näheren Betrachtung eines Systems zur Identifizierung seiner Komponenten und deren Beziehungen untereinander wird als Reverse Engineering bezeichnet [CC90]. Ausgehend von den durch Reverse Engineering gewonnenen Informationen zur Systemarchitektur lassen sich neue Anforderungen in das Konzept integrieren, die dann durch klassisches Forward Engineering realisiert werden.

Den Zyklus aus Reverse Engineering mit anschließendem Forward Engineering (siehe Abbildung 1.1) nennt man Reengineering, das Chikovsky und Cross in [CC90] wie folgt definieren:

> » It is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. «

Insbesondere der erste Schritt des Zyklus, das Reverse Engineering, gestaltet sich jedoch mit der wachsenden Komplexität moderner Softwaresysteme als zunehmend schwieriger und wird zu einem nicht zu vernachlässigenden Zeit- und Kostenfaktor bei der Pflege bestehender Softwaresysteme.

1.2 Reverse Engineering mit Fujaba

Computer Aided Software Engineering (CASE) Tools mit Reverse Engineering Fähigkeiten, wie das seit 1997 entwickelte Fujaba [FUJA], leisten bei dieser Arbeit eine wertvolle Hilfe. Fujaba bietet sowohl Reverse- als auch Forward-Engineering-Fähigkeiten zwischen der grafischen Modellierungssprache Unified Modelling Language (UML) und der Programmiersprache Java 2 von Sun Microsystems.

Die UML wurde ursprünglich ab 1994 bei der Rational Software Company von G. Booch und J. Rumbaugh entwickelt und kombiniert Aspekte der Booch Diagramme, der Object Modeling Technique (OMT) von J. Rumbaugh sowie den von I. Jacobson beschriebenen Object-Oriented Software Engineering (OOSE) Methoden. Sie bietet umfangreiche Möglichkeiten zur Spezifikation, Konstruktion und Visualisierung und von Softwarekomponenten und hat sich als Standard zur objekt-orientierten Modellierung von Softwaresystemen etabliert.

Fujaba ermöglicht Forward-Engineering auf der Basis von UML Klassen- und Aktivitätsdiagrammen, die zu diesem Zweck mit einer eigenen Graphgrammatik, den Storydiagrammen, erweitert wurden. Dadurch lassen sich innerhalb von Aktivitätsdiagrammen Interaktionen zwischen Objekten beschreiben und ermöglichen eine weitgehend graphbasierte Entwicklung von Software. Aus dem erzeugten Modell kann Fujaba Java-Quellcode exportieren.

Fujaba unterstützt Reverse Engineering auf zwei Ebenen:

Im ersten Schritt wird der Quellcode durch einen Parser eingelesen, der aus den syntaktischen Informationen der Quelle einen abstrakten Syntaxgraphen (ASG) aufbaut. Aus dem ASG generiert Fujaba dann zur Darstellungen der statischen Eigenschaften UML Klassendiagramme sowie Aktivitätsdiagramme, die den Kontrollfluss von Methoden abbilden.

Basierend auf dem abstrakten Syntaxgraphen kann im zweiten Schritt eine Cliché und Mustererkennung durchgeführt werden. Diese untersucht den abstrakten Syntaxgraphen nach Anhaltspunkten für den Einsatz von Design Pattern.

1.2.1 Statische Analyse

Design Patterns wurden erstmals 1995 von E. Gamma et al. in [GHJV94] vorgestellt und beschreiben abstrakte Softwareentwurfsmuster für die Lösung typischer und häufig wiederkehrender Probleme im Softwaredesign. Insgesamt differenziert [GHJV94] zwischen 23 verschiedenen Entwurfsmustern für unterschiedliche Problem- und Aufgabenkategorien, wobei ihr Einsatz dort zunächst auf die Standardisierung und Vereinfachung des Forward-Engineering beschränkt ist.

Gamma et al. gewannen die Ideen zu den verschiedenen Entwurfsmustern ursprünglich durch eine Betrachtung von existierenden Softwaresystemen bzw. der Vorgehensweise ihrer Entwickler. So lassen sich selbst in Systemen, die ohne konkrete Beachtung der in [GHJV94] beschriebenen Entwurfsmuster entwickelt wurden, deren Merkmale finden. Um eine abstraktere Sicht auf die Systemarchitektur und einen besseren Überblick über Funktionalität und Design zu ermöglichen, ist deshalb eine Analyse auf Entwurfsmuster während des Reverse-Engineering Prozesses erstrebenswert.
Ein solcher Ansatz wird in [Wen01] beschrieben und liegt als Plugin für Fujaba vor. Er ermöglicht mit Hilfe einer grafischen Musterspezifikationssprache Beispielausprägungen eines Entwurfsmusters zu spezifizieren, nach denen im ASG gesucht wird.

Beispielausprägungen repräsentieren jedoch nur die strukturellen Eigenschaften eines Entwurfsmusters, das eigentliche Verhalten zur Laufzeit bleibt unberücksichtigt. Das Resultat der statischen Analyse auf dem ASG ist somit nur eine Hypothese über die Existenz eines Entwurfsmusters, basierend auf einer partiellen Suche nach notwendigen aber nicht hinreichenden Kriterien.

Um das Problem zu verdeutlichen, vergleichen wir das *State*-Entwurfsmuster mit dem *Strategy*-Entwurfsmuster.

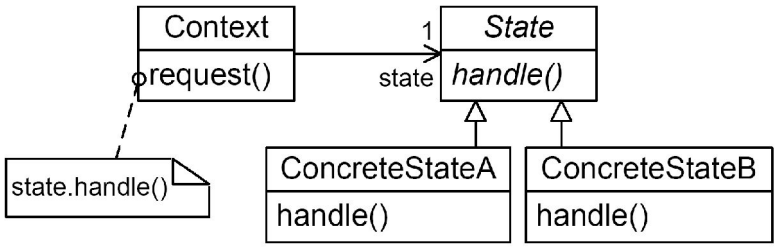

Abbildung 1.2: Klassendiagramm des State-Entwurfsmusters

Abbildung 1.2 zeigt die Struktur des State-Entwurfsmusters in UML Notation. Eine Kontextklasse `Context` verwaltet über eine Relation *state* einen für sie je nach Kontext veränderbaren Zustand. Diese Zustände werden durch Unterklassen der Klasse `State` beschrieben und implementieren die für den momentanen Zustand von `Context` notwendige Aktion über ihre `handle` Methoden. Anfragen eines Klienten über die `request` Methode von `Context` lösen diese Aktion über den momentanen Zustand, repräsentiert durch die jeweilige `State`-Implementierung, aus. Ein Zustandswechsel erfolgt als **Ereignis eines momentanen Zustands** oder durch **in der Kontextklasse definierte Bedingungen**.

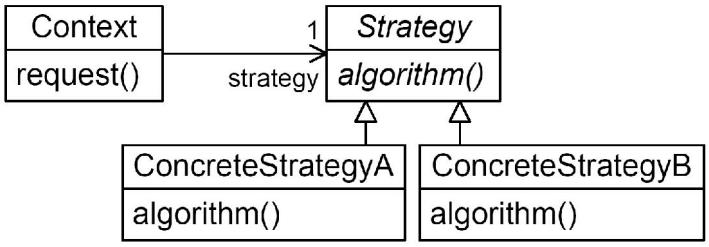

Abbildung 1.3: Klassendiagramm des Strategy-Entwurfsmusters

Das Strategy-Entwurfsmuster ist in Abbildung 1.3 dargestellt. Die Kontextklasse `Context` referenziert auf eine Unterklasse von `Strategy`, die einen Algorithmus zur Lösung einer bestimmten Aufgabe implementiert. Anfragen eines Klienten über die `request` Methode von `Context` führen zur Ausführung des momentanen Algorithmus über die `handle` Methode seiner Implementierung. In **Abhängigkeit von der Aufgabe** wird der verwendete Algorithmus über die `strategy` Referenz **vom Klient gewählt**.

Offensichtlich sind das State- und Strategy-Entwurfsmuster strukturell identisch, unterscheiden sich jedoch in ihrer Aufgabe und in ihrem Verhalten. [GHJV94] liefert neben der Entwurfsmusterstruktur deshalb eine textuelle Beschreibung der Verhaltensweise für das State-Entwurfsmuster:

> » [...] Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly. Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances. «

Und ferner für das Strategy-Entwurfsmuster:

> » A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context [...] «

Diese Verhaltensweisen werden – wie oben erwähnt – jedoch durch die statische Mustersuche nicht beachtet. Die Folge sind falsche Treffer bzw. nicht richtig zugeordnete Entwurfsmuster durch die Mustererkennung. Erst durch eine anschließende manuelle Verifikation der annotierten Softwarekomponenten können die Hypothesen der Mustererkennung dann bestätigt oder verworfen werden, indem der Software-Architekt relevante Passagen des Quellcodes auf ihr Verhalten hin untersucht.

Problematisch an der manuellen Verifikation ist der zu erwartende Zeitaufwand bei umfangreichen Softwaresystemen, sowie der konzeptuelle Widerspruch, durch eine automatisierte Analyse die händische Betrachtung des Quellcodes zunächst überflüssig zu machen, wobei aber anschließend durch den Hypothesencharakter des Resultates ein Wechsel auf diese Ebene unvermeidbar ist.

Aus diesen Gründen ist ein weiterer Analyseschritt wünschenswert, welcher die Resultate der Mustererkennung validiert und einen Wechsel auf die Ebene der Quellcodebetrachtung weitgehend überflüssig macht.

1.2.2 Dynamische Analyse

Aufbauend auf den Hypothesen über die Existenz von Entwurfsmustern der statischen Analyse lässt sich dieses Ziel mit einer dynamischen Analyse erfüllen. Dazu betrachtet die dynamische Analyse das Verhalten der Software zur Laufzeit und sucht nach Anhaltspunkten, um die vorher getroffenen Hypothesen zu bestätigen oder zu verwerfen.

Kandidaten einer Laufzeitüberwachung sind diejenigen Klassen, die bei einer erfolgreichen Mustersuche durch Elemente der Beispielausprägung repräsentiert worden sind. Werden während der Laufzeit von diesen Klassen Objekte instanziiert, untersucht man ihre Interaktionen dahingehend, ob sie den funktionalen Eigenschaften des Entwurfsmusters entsprechen.

Der eigentliche Analyseschritt erfolgt entweder als *online* Analyse während der Aufzeichnung der Laufzeitinformationen, oder in Form einer *post-mortem* Analyse nach einem Testlauf des Softwaresystems.

Eine online Analyse erweist sich dann als vorteilhaft, wenn während des Analyseprozesses durch Berücksichtigung bereits vorhandener Teilergebnisse die Kandidatenmenge angepasst werden soll. Eine post-mortem Analyse bietet eine saubere Trennung zwischen der Aufzeichnung der Laufzeitinformationen und ihrer Auswertung. Dies ermöglicht eine unabhängige Entwicklung beider Komponenten, so dass sich der verwendete Algorithmus bzw. der Aufzeichnungsmechanismus zur Analyse leicht austauschen lässt.

1.3 Zielsetzung

Im Rahmen dieser Studienarbeit soll ein Ansatz zur selektiven Aufzeichnung von Laufzeitinformationen für Java Programme vorgestellt werden, mit deren Hilfe eine dynamische post-mortem Analyse durchgeführt werden kann.

Folgende Ziele sind dabei zu erreichen:

- *Entwurf eines Konzepts zur selektiven Aufzeichnung von Laufzeitinformationen*
  Mit Hilfe einer geeigneten Eingabe sollen sich die Kandidaten für die Laufzeitüberwachung spezifizieren lassen. Die verwendete Technik sollte die Ausführung der zu analysierenden Software so wenig wie möglich beeinflussen und sich einfach auf andere Programmiersprachen übertragen lassen. Die Ausgabe soll vollständig im Sinne der angestrebten post-mortem Analyse sein.
- *Implementierung als Plugin für Fujaba und als eigenständige Software*
  Die Integration in Fujaba soll einen nahtlosen Übergang zwischen verschiedenen Reverse-Engineering Schritten ermöglichen. Die optionale Verwendung als eigenständige Software spart während der Aufzeichnung Sys-

temresourcen und ermöglicht zusammen mit dem post-mortem Konzept einen flexibleren Einsatz.

1.4 Aufbau der Arbeit

Kapitel 2 stellt verschiedene Techniken zur Laufzeitanalyse von Softwaresystemen vor, umreisst das jeweilige Anwendungsgebiet sowie Vorzüge und Nachteile.

In Kapitel 3 wird das in diesem Ansatz verwendete Konzept zur Aufzeichnung von Laufzeitinformationen vorgestellt.

Daraufhin beschreibt Kapitel 4 die technische Realisierung des Ansatzes durch den JavaTracer, einer Software zur selektiven Aufzeichnung von Laufzeitinformationen bei Java Programmen.

Kapitel 5 zeigt eine Beispielsitzung des JavaTracers und erläutert die Vorgehensweise zur Verwendung.

Das Kapitel 6 gibt schließlich eine Zusammenfassung der Arbeit und führt Möglichkeiten zur Erweiterung des Konzepts auf.

Kapitel 2
Grundlagen

Dieses Kapitel stellt unterschiedliche Arten und Methoden zur Laufzeitanalyse von Softwaresystemen vor. Der Begriff Debugee bezeichne dabei unabhängig von der gewählten Methode das Kompilat des Quellcodes, der analysiert werden soll.

Grundsätzlich motivieren die folgenden Problemstellungen Laufzeitanalysen, da sie durch eine rein Quellcode basierte Analyse nicht oder nur unzureichend beantwortet werden können:

Analyse von Laufzeitfehlern - Debugging

Beim Debugging interessiert das Laufzeitverhalten von Code oder Algorithmen, die reproduzierbare Fehler verursachen, sowie Inhalte der innerhalb des Codes oder Algorithmus verwendeten Variablen. Folglich kommen Techniken zum Einsatz, um die Ablaufstruktur eines bestimmten Teils einer Software zu visualisieren, zu steuern, sowie einen Einblick in die von dieser Komponente verwendeten Variablen zu geben.

Memory leak Fehler, die auf der fehlenden Freigabe von referenziertem aber nicht mehr benötigten Speicheradressraum basieren, lassen sich mit Hilfe von Debugging ihrem genauen Verursacher zuordnen.

Analyse von Laufzeiten und Häufigkeiten der Ausführung - Profiling

Das Profiling dient zur Optimierung der Laufzeiten einer Software. Es soll ineffiziente Algorithmen, deren Implementierungen und Position im Quellcode aufspüren, die bei einer rein Quellcode basierten Analyse nicht aufgefallen sind. Statt Ablaufstruktur und Variableninhalt interessieren hier Ausführungszeit und -häufigkeit und Dauer als metrische Kriterien. Aus den gewonnenen Daten lassen sich diejenigen Methoden ermitteln, die einer Optimierung bedürfen.

Nimmt man die Speicherverwendung des Debugees als metrisches Kriterium hinzu, kann Profiling das Vorhandensein von Memory leak Fehlern erkennen.

Analyse der semantischen Zusammenhänge zwischen Codekomponenten

Als Teil eines Reverse-Engineering Prozesses hilft eine Laufzeitanalyse, Abhängigkeiten zwischen Softwarekomponenten zu erkennen, die wiederum Rückschlüsse auf die Funktion zulassen. Wie auch beim Debugging liegt der Fokus der Analyse auf der Ablaufstruktur, er beschränkt sich allerdings nicht nur auf eine bestimmte Stelle im Code, sondern wird auf eine Menge von Softwarekomponenten ausgeweitet, deren semantischer Zusammenhang in Frage steht. Inhalte von Variablen sind nur von Interesse, wenn sich aus ihnen Rückschlüsse auf den Datenfluss zwischen den Softwarekomponenten ziehen lassen. Folglich betrachtet man hier Argumente und Rückgabewerte von Methodenaufrufen. Da die Analyse der Interaktion von Softwarekomponenten und nicht einzelne Fragmente derer im Vordergrund steht, verzichtet man auf eine Steuerung des Ablaufs durch den Software-Architekten. Die gesammelten Daten werden nach der Debug-Sitzung ausgewertet, gefiltert und visualisiert und ergeben dann ein Bild der Ablaufstruktur.

Als Debugger oder Profiler werden Programme bezeichnet, die einem Software-Architekt bei der Lösung der oben genannten Problemstellungen behilflich sind, indem sie Überwachungs- und Steuerungstechniken am Debugee zur Verfügung stellen. Die Überwachung am Debugee bezeichnet man auch als *Monitoring*, die Steuerung durch den Debugger *Steering*. Betreibt ein Debugger reines Monitoring und werden die gesammelten Daten erst nach der Ausführung des Debugees analysiert, spricht man von einer *post-mortem* oder *offline* Analyse, anderenfalls von einer *online* Analyse.

Im folgenden werden die dabei verwendeten Methoden kurz vorgestellt und erläutert.

2.1 Instrumentierung

Bei der *Quellcode basierten* Instrumentierung erweitert der Debugger zur Ablaufverfolgung Methoden des Debugees um automatisch generierten Quellcode. Nach der Erweiterung kompiliert der Debugger den modifizierten Quellcode und führt ihn aus, wobei der eingefügte Code als Teil des Debugees entweder die gewünschten Ablauf- und Variableninformationen selbsttätig aus dem laufenden Programm abgreift und speichert oder sie über eine Schnittstelle an einen

Debugger weitergibt. Nachteilig an diesem Verfahren ist die Prämisse, den Debugee als Quelltext zur Erweiterung mit dem vom Debugger erzeugten Code vorliegen zu haben.

Abhilfe schafft hier eine *Bytecode basierte* Instrumentierung. Dabei wird der Code des Debuggers erst zur Laufzeit des Debugees injiziert, indem der Debugger über eine geeignete Schnittstelle Ladevorgänge der Laufzeitumgebung von kompiliertem Code in den Speicher überwacht und noch vor dessen Ausführung den eigenen Code in binärer Form einstreut. Ein verfügbarer Quelltext und das Kompilieren im Zuge der Instrumentierung sind somit nicht notwendig.

Bei der *dynamischen* (Bytecode-basierten) Instrumentierung entscheidet der Debugger ferner aufgrund von bereits durch die Laufzeitanalyse gesammelten Informationen und auf der Basis von Parametern, die die Zielsetzung des Software-Architekten beschreiben, ob die Instrumentierung einer gerade neu geladenen Softwarekomponente der Analyse dienlich ist. Dadurch vermeidet dynamische Instrumentierung einen Overhead an eingestreuten Code.

Bytecode basierte Instrumentierung erfordert jedoch genaue Kenntnisse über das Speichermanagement des Debugees sowie dessen Laufzeitumgebung, da:

- Bei nativem Code die Adresstranslation in den Arbeitsspeicher bereits stattgefunden hat,
- die MMU (Memory Management Unit) oder das OS Segmentierung einsetzen und somit ursprünglicher Adressraum des Codes und Adressraum im Speicher nicht mehr korrelieren sowie
- Objekte, Methoden und Variablen im Kompilat keine natürlichsprachigen Namen mehr besitzen.

Abhilfe für den letztgenannten Punkt schaffen Debuginformationen, die sich von gängigen Kompilern im Kompilat hinterlegen lassen. Diese werden zur Laufzeit selbst nicht ausgeführt, enthalten aber alle notwendigen Informationen, mit denen sich Objekt-, Methoden- und Variablennamen sowie deren Typ zu ihren Verwendungsorten im Kompilat zuordnen lassen.

Durch die direkte Manipulation am Debugee ist es unvermeidbar, dass Instrumentierung während der Laufzeit zu unerwünschten Nebeneffekten führt. Diese Effekte nennt man Überwachungseffekte oder Heisenbugs [ADN01]. So sorgt der eingestreute Code unter anderem für einen erhöhten Speicherbedarf des Debugees und einer Erhöhung der Ausführungszeiten. Letzteres kann bei nicht-de-

terministischem Code die erfolgreiche Fehlersuche verhindern. Einen einfachen Fall dieses Problems bilden offensichtlich zwei Threads, die auf einen gemeinsamen Datenbestand zugreifen. Wird einer dieser Threads instrumentiert, verlangsamt sich sein Ausführungsintervall und die Reihenfolge des Zugriffs auf die Daten weicht von der uninstrumentierten Ausführung ab.

Eine weitere, sehr primitive Form der Instrumentierung stellt das manuelle Einstreuen von Konsolenausgaben durch den Software-Architekten in den Quelltext dar, nach dem C++ Hochsprachenbefehl auch „printf-Statement“-Verfahren genannt [Glas01].

Der Vollständigkeit halber werden an dieser Stelle zwei Ansätze beschrieben, die Instrumentierung einsetzen. Neben diesen existieren noch weitere, ein detaillierter Vergleich würde aber den Umfang der vorliegenden Arbeit sprengen.

2.1.1 Java Bytecode-Instrumentierung am Beispiel jclasslib

Mit Hilfe von jclasslib [JCLA], einem Open-Source Projekt der Firma ej-technologies, lässt sich der Bytecode von Java Klassendateien betrachten, extrahieren und zurückschreiben. Dabei ermöglicht jclasslib sowohl die Manipulation von Methoden, Attributen und lokalen Variablen als auch von Konstanten.

Neben dem API enthält jclasslib einen Bytecode-Betrachter, der zur besseren Lesbarkeit den angezeigten Bytecode mit Debuginformationen aus der Klassendatei annotiert und Aufrufe mit Hyperlinks zum Aufrufziel versieht. Der ebenfalls von dieser Firma vertriebene JProfiler setzt unter anderem jclasslib zur oben beschriebenen dynamischen Instrumentierung ein.

2.1.2 Omniscient Debugger – ODB

In [Lew03] kommt Instrumentierung zum Einsatz, um zur Laufzeit eines Java Debugees Informationen über sämtliche Methodenaufrufe und Variablenänderungen aufzuzeichnen. Nach der Ausführung des Debugees, die sonst nicht weiter von dem ODB beeinflusst wird, kann der Software-Architekt über die Oberfläche des ODB aufgrund der gesammelten Daten den Ablauf des Debugees schrittweise zurückzuverfolgen. Der ODB zeigt dabei die jeweilige Position im Quelltext und Veränderungen an Variablen an.

Problematisch bei diesem Ansatz ist nach der Aussage des Autors das entstehende Datenvolumen, welches nach seiner worst-case Abschätzung Größen von

100MB/sek. annehmen kann sowie eine Laufzeiterhöhung durch Instrumentierung um den Faktor 10. Eine Lösung besteht seiner Ansicht nach in dem Verzicht von Aufzeichnungen über sich deterministisch verhaltende Codesequenzen und ihren Variablen sowie der Beschränkung auf die vom Software-Architekt als für die Analyse relevant betrachteten Klassen. Auf eine genaue Angabe der Vorgehensweise zur Lösung dieser Probleme wird jedoch verzichtet.

2.2 Steuerung der Laufzeitumgebung

Anstatt wie bei der Instrumentierung Manipulationen am Debugee vorzunehmen, lassen sich Überwachung und Steuerung auch durch eine Einflussnahme auf die Laufzeitumgebung des Debugees durchführen.

Dabei werden folgende Fälle unterscheiden:

Hardware- oder Hybridmonitoring

- Der Debugee läuft auf einer CPU, die über einen Befehlssatz zur ereignisbasierten Steuerung der Laufzeitumgebung verfügt (z. B. 32-bit Intel x86) [HL]. Wird die Überwachung im Falle eines Ereignisses durch eine Software unterstützt, die in der gleichen Laufzeitumgebung wie der Debugee läuft, spricht man von Hybridmonitoring.

Softwaremonitoring

- Ein Debugger emuliert eine virtuelle CPU (z. B. 68000VCpu, NO$GBA) [Kor].
- Bei dem Kompilat handelt es sich um eine interpretierte Sprache, die auf einer virtuellen Maschine (VM) mit entsprechender Unterstützung zur ereignisbasierten Steuerung läuft.

In allen drei Fällen erfolgt die Steuerung der Laufzeitumgebung über einen Debugger, der entweder als Schnittstelle zwischen der steuerbaren Laufzeitumgebung fungiert oder selbst die Laufzeitumgebung emuliert.

Während bei der Instrumentierung die Ablaufverfolgung über eingestreuten Code als Teil des ausgeführten Kompilats erfolgt, wird hier die Ablaufverfolgung des Debuggers mit Hilfe von Ereignissen initiiert:

Breakpoint Ereignisse

Adressen des ausführbaren Codes des Debugees können mit Breakpoints versehen werden. Erreicht der Programmzähler der Laufzeitumgebung die Adresse eines Breakpoints, wird die Ausführung an dieser Stelle gestoppt und die Kontrolle über die Laufzeitumgebung geht an den Debugger über.

Watchpoint Ereignisse

Auf Adressen des Datenbereichs eines Debugees können Watchpoint gesetzt werden. Diese veranlassen die Laufzeitumgebung die Ausführung zu stoppen, sobald der Inhalt einer Watchpoint Adresse vom Debugee geändert wird. Die Kontrolle über die Laufzeitumgebung geht an den Debugger über.

Tritt ein Ereignis ein, wird es im Falle einer offline Analyse durch den Debugger aufgezeichnet. Danach wird die Ausführung des Debugees unmittelbar fortgesetzt.

Bei einer online Analyse unterbricht der Debugger an dieser Stelle die Ausführung des Debugees. Der Software-Architekt erhält dann über den Debugger Einblick auf den Stand des Programmzählers und den ihn umgebenden Code sowie auf den Inhalt des Laufzeitkellers und Werte von Speicheradressen. Mit Hilfe von Debuginformationen lassen sich bestimmten Adressräumen sowohl die in ihnen enthaltenen Objekte, Methoden und Attribute zuordnen als auch Referenzen zwischen diesen und ihrem Ursprung im Quellcode herstellen.

Der Debugger kann die Laufzeitumgebung ferner dazu veranlassen, dass der Debugee dem Quellcode entsprechend zeilenweise ausgeführt wird. Über eine Visualisierung im Debugger kann der Software-Architekt dann die Ablaufstruktur im Quellcode verfolgen. Außerdem lassen sich zu jedem Zeitpunkt weitere Break- oder Watchpoints setzen oder bestehende wieder entfernen, so dass der Debug-Prozess an die momentane Zielsetzung des Software-Architekten angepasst werden kann.

Der Vorteil dieser nicht invasiven Technik besteht im Vergleich zur Instrumentierung darin, dass das Kompilat nicht durch die Einstreuung von zusätzlichem Code verändert werden muss. Der sonst notwendige Zugriff auf den Quellcode entfällt und die Ablaufverfolgung kann während der Ausführung des Kompilats durch Einflussnahme auf die Laufzeitumgebung geändert werden, ohne dass der Code neu kompiliert werden muss oder komplexe Mechanismen zur direkten Veränderung des Kompilats im Arbeitsspeicher benötigt werden.

Performanceeinbussen durch eingestreuten Code werden vermieden, was insbesondere beim Einsatz der Technik zum Profiling hilfreich ist.

Für den in dieser Arbeit vorgestellten Ansatz sind insbesondere Techniken zur Steuerung der Laufzeitumgebung der Sun Java 2 Virtual Machine von Relevanz.

Die folgenden zwei Kapitel beschreiben mit dem Java Virtual Machine Profiler Interface (JVMPI) und der Java Platform Debugger Architecture (JDPA) zwei von dieser Laufzeitumgebung unterstützte Vorgehensweisen zur Gewinnung von Laufzeitinformationen.

2.2.1 Profiling mit JVMPI

Suns Java Virtual Machine Profiler Interface (JVMPI) beschreibt eine Schnittstelle zur Erweiterung der Sun Java 2 Virtual Machine um Profiling Fähigkeiten. Die Umsetzung erfolgt als native Bibliothek, die zu Profilingzwecken zusammen mit der Virtual Machine geladen und dann in der gleichen Laufzeitumgebung wie die VM ausgeführt wird.

Der Aufruf erfolgt als Kommandozeilenschalter der Virtual Machine in der Form:

```
java -Xrunmyprofiler ToBeProfiledClass
```

Wobei **`myprofiler`** den Namen der nativen Bibliothek angibt.

Über das JVMPI lassen sich unter anderem folgende Ereignisse der Virtual Machine überwachen:

- Laden und Entladen von Klassen,
- Allozierung und Freigabe von Objekten ,
- Aufruf und Verlassen von Methoden,
- Start und Beendigung von Threads,
- Veränderungen des Heaps,
- Start und Abbruch des Garbage Collectors sowie das
- Setzen von und Ereignisse auf Monitoren.

Darüber hinaus gestattet das JVMPI Zugriff auf Daten im Umfeld der Ereignisse und der Laufzeitumgebung, insbesondere auf den Laufzeitkeller und Inhalte von Variablen.

Eine Referenzimplementierung eines Profilers für diese Schnittstelle, Suns HPROF Profiling Agent liegt allerdings nur in Versionen für Solaris und Windows vor. Ferner definiert sie keine Schnittstelle des JVMPI zur Kommunikation mit Softwarekomponenten außerhalb der VM, die zur Steuerung dieser z. B. durch einen Debugger notwendig wäre.

Entsprechend ist ein erheblicher Aufwand durch eine Erweiterung des JVMPI auf der back-end Seite um ein Protokoll zur Kommunikation von VM und externem Debugger, um Möglichkeiten zum Setzen von Break- und Watchpoints, um eine Anbindung des Protokolls an einen Java Front-end sowie durch entsprechende Implementierungen für verschiedene Plattformen von Nöten, möchte man sich das JVMPI als Debugging-Schnittstelle nutzbar machen. Das JVMPI hat ferner nach Angaben von Sun experimentellen Status und wird in zukünftigen Versionen des SDK durch das im Funktionsumfang universellere Java Virtual Machine Tool Interface (JVMTI) ersetzt. Einem auf dem JVMPI beruhenden Ansatz würde somit in naher Zukunft seine Existenzgrundlage entzogen werden.

Mit der Java Platform Debugger Architecture (JPDA) des Java 2 SDK existiert jedoch eine weitere Schnittstelle der Virtual Machine, die den obigen Anforderungen bezüglich eines Protokolls und eines Java Front-Ends entspricht.

2.2.2 JPDA

Die Java Platform Debugger Architecture (JPDA) definiert als Bestandteil des Sun Java 2 SDK [JPDA] ein API zur einfachen Realisierung von Debug Systemen über die Programmiersprache Java 2.

Das JPDA besteht aus drei Schichten (siehe Abbildung 2.1), die jeweils ein Interface zur Anbindung an die Virtual Machine (JVMDI), ein Kommunikationsprotokoll zwischen diesem und der Debugger-Software (JDWP) sowie ein Interface zwischen Protokoll und Java Programmiersprache (JDI) spezifizieren. Diese Unterteilung erlaubt durch das Kommunikationsprotokoll den getrennten Ablauf von Debugger und Debugee auf zwei unterschiedlichen Systemen. Ferner vereinfacht sie die Portierung auf andere Systeme, da hierfür nur die back-end Schnittstelle des JVMDI angepasst werden muss.

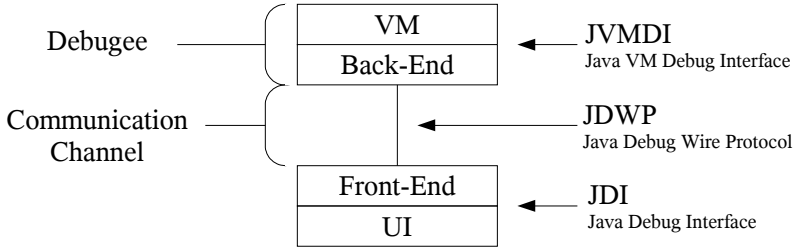

Abbildung 2.1: Aufbau JPDA

Java Virtual Machine Debug Interface (JVMDI)

Das JVMDI spezifiziert die back-end Schnittstelle zwischen der Virtual Machine, die den Debug-Kandidaten ausführt, sowie dem darüber liegenden Kommunikationsprotokoll JDWP.

Ein JVMDI besteht aus einer nativen Bibliothek, die über das Java Virtual Machine Tool Interface (JVMTI) an eine VM angebunden wird. Zur Laufzeit der VM nimmt das JVMDI Aufforderungen zur Ereignisüberwachung, z. B. in der Form von Breakpoints, über das JDWP entgegen und initiiert die entsprechende Ereignisüberwachung in der VM. Tritt ein Ereignis auf, übergibt die VM Daten über das Ereignis an das JVMDI und hält bei Bedarf die Ausführung des Debugees an.

Referenzimplementierungen für alle von Java 2 unterstützten Plattformen sind im Java 2 SDK enthalten, so dass trotz der nativen Form des JVMDI die Plattformunabhängigkeit gewährleistet ist.

Java Debug Wire Protokoll (JDWP)

Das Java Debug Wire Protokoll beschreibt das Format des Informationsflusses von Aufforderungen zur Ereignisüberwachung und den aus überwachten Ereignissen gewonnenen Daten zwischen Debugger und Virtual Machine. Implementierungen des Protokolls erfolgen auf back-end Seite durch das JVMDI und auf der front-end Seite durch das JDI. Das JDWP erlaubt ferner, Debugee und Debugger auf zwei verschiedenen Systemen laufen zu lassen, die dann über ein Netzwerk miteinander kommunizieren.

Java Debug Interface (JDI)

Als front-end Interface zwischen einem in Java 2 implementierten Debugger und dem JDWP definiert das JDI die oberste Schicht des JPDA. Entsprechende Funk-

tionalität in dieser Schicht erlaubt sowohl Steuerung als auch Monitoring auf dem Debugee.

Eine Referenzimplementierung des JDI ist im Java 2 SDK enthalten und lässt sich unter allen von Java 2 unterstützen Plattformen verwenden.

Für den in dieser Arbeit vorgestellten Ansatz sind insbesondere die Fähigkeiten des JDI interessant. Die der anderen Schichten ermöglichen in diesem Zusammenhang den von Hardware, Betriebssystem als auch verwendeter Virtual Machine (solange für diese eine Implementierung des JVMDI vorliegt) unabhängigen Einsatz.

Das JDI bietet Funktionen zum:

- Starten und Steuern von Laufzeitumgebungen der VM,
- Setzen von Break- und Watchpoints,
- Übernahme der Kontrolle der Laufzeitumgebung beim Einsprung oder Verlassen aller Methoden einer Klasse,
- Übernahme der Kontrolle der Laufzeitumgebung bei Exceptions, Classloader-Ereignissen und startenden Threads,
- Anhalten und Wiederaufnahme von Threads,
- Zugriff auf Attribute in Form von Inhalt und Typ,
- Zugriff auf den Laufzeitkeller,
- Zugriff auf Debuginformationen wie Klassen-, Methoden- und Attributnamen sowie
- Zuordnung zwischen der Position des Programmzählers eines Threads und der Repräsentation des Codesegments im Quellcode.

Der Debugger startet zunächst über eine Instanz der JDI Klasse `com.jdi.LaunchingConnector` eine neue Virtual Machine zur Ausführung des Debugees und setzt dann die zu überwachenden Ereignisse auf dieser VM. Anschließend startet der Debugger die Ausführung des Debugees und wartet auf eintreffende Ereignisse. Dabei ist es jederzeit möglich, weitere Ereignisüberwachungen hinzuzufügen oder zu löschen, die Ausführung zu unterbrechen oder fortzufahren sowie auf Daten der Laufzeitumgebung zuzugreifen.

Alternativ kann die Virtual Machine auch mit dem Kommandozeilenschalter

```
java -Xrunjdwp:transport=dt_socket,server=y,suspend=y,
address=8000 -Xdebug -Xnoagent ToBeDebuggedClass
```

gestartet werden. Diese öffnet den mit `address` spezifizierten Port und wartet dann solange mit der Ausführung des Debugees, bis sich über das JDWP ein Debugger mit ihr verbunden hat und den Befehl zur Wiederaufnahme der Ausführung erteilt. Debuggerseitig erfolgt die Verbindung über eine Instanz der Klasse `com.jdi.AttachingConnector` unter Angabe von Hostname und Port des Systems, dessen VM auf die Verbindung wartet.

Weitere hier nicht näher berücksichtigte Möglichkeiten des JPDA bestehen in der Vorgabe von Ports durch den Debugger sowie der automatischen Ausführung des Debuggers durch die VM des Debugees bei Auftreten im Falle einer nicht gefangenen Exception.

2.2.3 JPDA am Beispiel JaVis

JaVis, » A UML-Based Visualization and Debugging Environment for Concurrent Java Programs « [Meh02a], verwendet das JPDA zur Aufzeichnung sämtlicher Methodenaufrufe- und austritte einer Software zur Visualisierung ihres Ablaufs und zur Erkennung von Deadlocks. Dazu werden während der Aufzeichnung alle Ereignisse zunächst textuell gespeichert und nach der Ausführung über die IDE TogetherJ [TOGJ] als UML Sequenzdiagramm dargestellt. Der Ansatz arbeitet jedoch nicht selektiv unter Verwendung der JDI Ereignisanforderung zur Erzeugung von Ereignissen auf allen Methodeneinsprüngen bzw. -austritten, was erhebliche Performanzeinbussen bei der Ausführung der zu überwachenden Software zur Folge hat. Die Autorin von [Meh02a] thematisiert dieses Problem in [Meh02b].

Kapitel 3
Konzepte

Im ersten Kapitel wurde die von Fujaba unterstützte Reverse-Engineering Methode zur Analyse eines Softwaresystems auf Entwurfsmuster vorgestellt. Der auf einer statischen Analyse beruhende Ansatz hat – wie dort bereits gezeigt – den Nachteil, dass er neben richtigen auch viele falsche Treffer, so genannte »false Positives«, liefert.

Dieses Kapitel beschreibt ein Konzept zur selektiven Aufzeichnung von Laufzeitinformationen. Mit diesen Informationen lassen sich dann falsche Treffer durch eine dynamische Analyse aus der Ergebnismenge herausfiltern. Die Aufzeichnung stützt sich dabei auf das Resultat der vorangegangenen statischen Analyse, um die Laufzeitüberwachung auf für eine dynamische Analyse relevante Ereignisse einzuschränken.

Abschnitt 3.1 fasst dazu den Ansatz der statischen Analyse zusammen und beschreibt ihre Ergebnisse, mit denen sich eine Kandidatenbasis für die Laufzeitüberwachung bilden lässt. Abschnitt 3.2 zeigt einen Ansatz einer dynamischen Analyse auf der Basis von Verhaltensmustern. Durch die Berücksichtigung der Verhaltensmusterspezifikation lassen sich sowohl die Kandidaten der Laufzeitanalyse weiter eingrenzen als auch die notwendige Ausgabe definieren. Dazu führt der Abschnitt mit UML 2.0 Sequenzdiagrammen eine Möglichkeit zur Definition dieser Verhaltensmuster ein. Im Abschnitt 3.3 wird der auf diesen Voraussetzungen beruhende Ansatz zur Aufzeichnung von Laufzeitinformationen beschrieben.

3.1 Statische Analyse

Resümierend zusammengefasst verwendet die in Abschnitt 1.2.1 vorgestellte Mustererkennung strukturelle Eigenschaften des Quellcodes als Indikatoren zur Erkennung von Entwurfsmustern. Die strukturellen Eigenschaften von Entwurfsmustern werden dazu über eine grafische Musterspezifikationssprache in Form von Beispielausprägungen beschrieben. Auf dieser Basis durchsucht die Muster-

erkennung den abstrakten Syntaxgraphen. Bei einem Treffer wird der abstrakte Syntaxgraph um eine entsprechende Annotation erweitert.

Der hier vorgestellte Ansatz zur Aufzeichnung von Laufzeitinformationen soll selektiv auf der Basis der Ergebnisse der statischen Mustererkennung operieren. Zunächst müssen deshalb die Kandidaten einer Laufzeitüberwachung bestimmt werden. Eine mögliche Kandidatenmenge wird bereits durch die der Mustererkennung zu Grunde liegenden Beispielausprägungen definiert. Betrachten wir dazu den Aufbau einer solchen Musterspezifikation.

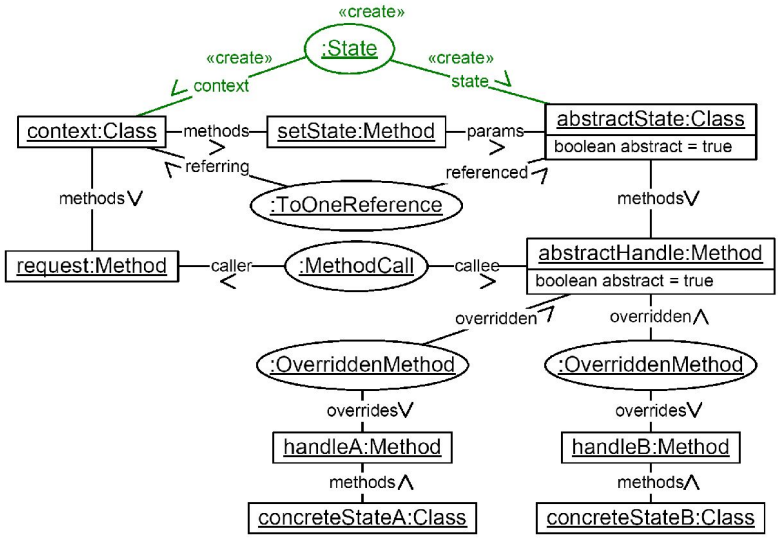

Abbildung 3.1: State-Beispielausprägung

Abbildung 3.1 zeigt eine grafische Musterspezifikation für das State-Entwurfsmuster. Die Grammatik dieser Spezifikationssprache wird in [Pal01] vorgestellt und durch [Wen01] erweitert. Eine Beispielausprägung definiert strukturelle Eigenschaften eines Entwurfsmusters und bildet somit eine Äquivalenzklasse über seine konkreten Ausprägungen.

Sie setzt sich aus ASG-Objekten, die in der grafischen Musterspezifikation von Rechtecken verkörpert werden, Musterobjekten in Form von Ovalen, sowie Verknüpfungen zwischen diesen, dargestellt durch verbindende Kanten, zusammen. ASG-Objekte vertreten konkrete Ausprägungen von Objekten im abstrakten Syntaxgraphen, an die Bedingungen geknüpft werden können. Musterobjekte repräsentieren bereits im ASG gefundene Muster, auf deren Existenz die Suche

dieser Beispielausprägung aufbaut. Verknüpfungen verbinden Knoten miteinander und beschreiben Beziehungen, die eine konkrete Ausprägung erfüllen muss.

Zur Mustererkennung wird der abstrakte Syntaxgraph nach den Elementen der Beispielausprägung abgesucht. Bei einer positiven Suche werden die konkreten Ausprägungen im ASG um die in der Beispielausprägung mit dem Stereotyp «`create`» gekennzeichneten Elemente erweitert, die diesen Kontext als mögliche Ausprägung des gefundenen Entwurfsmusters markieren.

Zur Erfüllung der Kriterien der State-Entwurfsmuster Beispielausprägung muss eine konkrete Ausprägung im ASG eine Klasse (`context:Class`) besitzen, die eine Referenz auf eine andere Klasse hält (`:ToOneReference`). Für diese Referenz muss ferner eine Zugriffsmethode existieren (`setState:Method`). Eine Methode dieser Klasse (`request:Method`) muss ein Methodenaufruf (`:MethodCall`) auf einer Methode (`handle:Method`) der referenzierten Klasse durchführen.

Sowohl die Referenz als auch der Methodenaufruf beziehen sich dabei auf eine abstrakte Klasse und eine abstrakte Methode, beschrieben durch die Bedingung (`boolean abstract = true`). Beide müssen jedoch über mindestens zwei nicht abstrakte Ausprägungen (`:OverriddenMethod`) verfügen.

Trifft die Mustererkennung auf eine solche Musterausprägung im abstrakten Syntaxgraphen, werden die grün dargestellten «`create`» Elemente produziert und der ASG somit um eine `:State` Annotation erweitert.

Die explizit als Vertreter des Entwurfsmusters annotierten Klassen – im Beispiel `context` und `abstractState` – sind als Kandidaten zur Laufzeitüberwachung offensichtlich unzureichend, da sie nur Teilaspekte des Entwurfsmusters repräsentieren. Deshalb wird die Kandidatenmenge zur Validierung einer Entwurfsmusterhypothese h zunächst um alle durch Elemente der Beispielausprägung vertreten Klassen K_h und deren Methoden M_h erweitert. Die Kandidatenmenge enthält damit alle Elemente, die schon ein notwendiges Kriterium bei der strukturellen Suche nach einem Entwurfsmuster sind.

3.2 Dynamische Analyse

Das Ziel der dynamischen Analyse liegt nun in der Validierung einer Entwurfsmusterhypothese auf dieser Kandidatenmenge durch die Berücksichtigung des

Laufzeitverhaltens. Dazu wird für jedes Entwurfsmuster ein Verhaltensmuster definiert, das eine zeitliche Abfolge von Interaktionen zwischen einer Menge von Objekten beschreibt. Elemente des Verhaltensmusters referenzieren auf ASG-Elemente der Beispielausprägung und somit auf konkrete Klassen und Methoden der Kandidatenmenge. Folgt das Verhalten der Kandidaten zur Laufzeit dem Verhaltensmuster, bestätigt die dynamische Analyse die Hypothese der Mustererkennung, ansonsten wird sie als falscher Treffer verworfen [Wen03].

Werden nicht alle Elemente der Beispielausprägung referenziert, reicht es, die ursprüngliche Kandidatenmenge der Laufzeitüberwachung auf die vom Verhaltensmuster berücksichtigten Klassen K_v und deren Methoden M_v zu beschränken (dabei gilt offensichtlich $K_v \subseteq K_h$, $M_v \subseteq M_h$).

Zur Definition der Verhaltensmuster eignen sich UML 2.0 Sequenzdiagramme [Wen04], mit denen sich eine zeitliche Abfolge von Interaktionen zwischen einer Menge von Objekten innerhalb eines zeitlich begrenzten Kontextes beschreiben lässt. Die genaue Spezifikation der Syntax von Sequenzdiagrammen findet sich in [UML03].

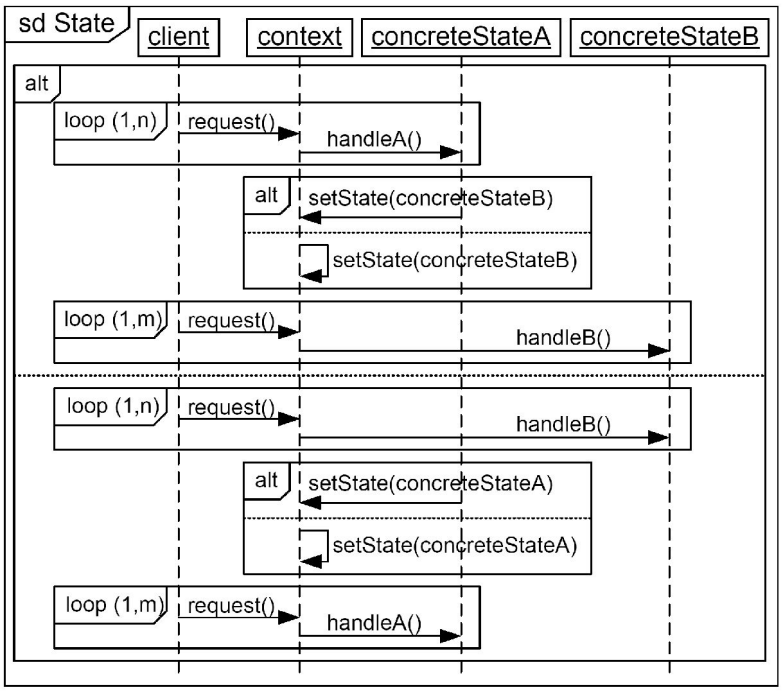

Abbildung 3.2: State Verhaltensmuster

Abbildung 3.2 zeigt das Verhaltensmuster für State-Entwurfsmuster. Im Beispiel sind konkrete Ausprägungen der ASG-Klassen-Elemente `context`, `concreteStateA` und `concreteStateB` die Kandidaten der dynamischen Analyse. Das Element `client` referenziert auf kein ASG-Element in der Beispielausprägung, so dass dessen Rolle durch ein beliebiges Objekt übernommen werden kann.

In Übereinstimmung mit der textuellen Spezifikation des State-Entwurfsmusters in [GHJV94] (siehe auch Abschnitt 1.2.1) verlangt das Verhaltensmuster, dass zur Laufzeit Objektinstanzen dieser vier Klassen existieren, die auf zwei alternative (`alt`) Arten miteinander interagieren:

In der ersten möglichen Interaktionssequenz ruft das Objekt `client` die Methode `request` von `context` auf, die ihrerseits die Methode `handleA` auf `concreteStateA` ausführt. Diese Interaktion muss mindestens einmal erfolgen, kann sich jedoch beliebig wiederholen (`loop(1,n)`). Anschließend erfolgt ein Wechsel des Zustands über die Methode `setState` von `context`, der entweder von `context` selbst initiiert wird oder über den aktuellen Zustand `concreteStateA` erfolgt. Daraufhin muss von `client` mindestens ein weiterer Aufruf von `request` auf `context` erfolgen, der jetzt jedoch eine Ausführung von `handleB` auf `concreteStateB` verursacht.

Die zweite mögliche Interaktionssequenz beschreibt im Prinzip denselben Ablauf wie oben, allerdings sind hier in Bezug auf die Reihenfolge `concreteStateA` und `concreteStateB` vertauscht.

Ein derart spezifiziertes Verhaltensmuster folgt implizit der *Consider*-Semantik von UML 2.0, wonach zu jeder Zeit beliebige weitere Methodenaufrufe auf nicht vom Verhaltensmuster berücksichtigte Methoden gestattet sind. Lediglich die Reihenfolge der vom Verhaltensmuster beschriebenen Methodenaufrufe ist demnach Kriterium zu seiner Erfüllung. Schreiben die Kriterien eines Entwurfsmusters allerdings vor, dass auf bestimmten Objekten keine anderen als die in der Spezifikation angegebenen Interaktionen stattzufinden haben, ist die consider Semantik unzureichend.

In diesem Fall deklariert man entsprechende Teile des Verhaltensmusters in Konformität mit der UML 2.0 Spezifikation als einen *Critical*-Abschnitt. Das Verhalten der beteiligten Objekte wird dann zur Laufzeit auf sämtliche Interaktionen hin untersucht, nicht im Verhaltensmuster spezifizierte falsifizieren automatisch die Entwurfsmusterhypothese.

3.3 Selektive Aufzeichnung von Laufzeitinformationen

Um das Laufzeitverhalten der Software mit dem Verhaltensmuster zu vergleichen, müssen zur Laufzeit auf der Kandidatenmenge folgende Informationen gesammelt werden:

- I_1 Reihenfolge der Ausführung der Methoden aus M_v
- I_2 übergebene Parameter
- I_3 ausführendes Objekt (Instanz einer Klasse aus K_v)
- I_4 aufrufendes Objekt
- I_5 ausführender Thread

Die Notwendigkeit zur Aufzeichnung von $I_1 - I_4$ ergibt sich unmittelbar aus der Semantik der Verhaltensmusterspezifikation. Zusammen mit I_5 gestatten sie eine vollständige Rekonstruktion des Kontrollflusses auf den überwachten Methoden.

Kriterien für die Aufzeichnung der Laufzeitinformationen sind:

- K_1 Die Eingabe erlaubt eine hinreichende Beschreibung der zu berücksichtigenden Methoden aus M_v und der consider/critical Semantik der Verhaltensmusterspezifikation.
- K_2 Die Aufzeichnung verursacht möglichst wenig Überwachungseffekte.
- K_3 Während der Aufzeichnung wird der Software-Architekt darüber informiert, ob das aufzuzeichnende Ereignis bereits eingetreten ist.
- K_4 Die Ausgabe ist hinreichend im Sinne der für eine dynamische Analyse notwendigen Informationen.

3.3.1 Trace-Types

Zur Erfüllung des Kriteriums K_1 muss die Laufzeitüberwachung Consider- und Critical-Abschnitte in einem Verhaltensmuster berücksichtigen. Consider-Abschnitte betrachten die Ausführungsreihenfolge bestimmter Methoden zwischen Objekten, ignorieren allerdings alle Ereignisse außerhalb dieser Betrachtung. Critical-Abschnitte schreiben vor, dass keine Ereignisse außer den im Abschnitt definierten zwischen den betrachteten Objekten auftreten.

Dazu korrespondierend werden folgende Überwachungstypen unterstützt:

Consider-Überwachungen

Dieser Überwachungstyp dient zur Berücksichtigung der in M_v enthaltenen Methoden. Eine Consider-Überwachung besteht aus einem voll qualifizierten Klassennamen N_K sowie einer Menge aus zu überwachenden Methoden dieser Klasse beschrieben durch ihre Signatur S. Consider-Überwachungen eignen sich zur Verifikation, ob die durch ein Methodentupel aus M_v beschriebene Reihenfolge eingehalten wird, ignorieren aber alle weiteren Ereignisse auf der Kandidatenmenge K_v, also Ereignisse von Methoden, die nicht in M_v enthalten sind. Zusätzlich lässt sich ein Filter auf K_v definieren, der bei der Aufzeichnung selektiv nur die Methodenaufrufe berücksichtigt, die durch Elemente des Verhaltensmusters ausgelöst wurden.

Critical-Überwachungen

Critical-Überwachungen stellen sicher, das keine Methoden außerhalb der durch das Verhaltensmuster definierten Menge M_v auf den Klassen K_v ausgeführt werden. Da die Nicht-Ausführung von Methoden nicht überwacht werden kann, werden stattdessen alle Methoden eines durch einen voll qualifizierten Klassennamen N_K angegebenen Kandidaten überwacht. Enthält die Aufzeichnung Ereignisse einer Methode von K_v, die nicht in M_v enthalten ist, wurde gegen das Critical-Verhaltensmuster verstoßen.

3.3.2 Eingabe

Die Eingabe besteht dem Kapitel 3.3.1 folgend aus einer Menge von Consider- und Critical-Überwachungen. Eine Consider-Überwachung wird definiert durch ein Tupel

$$E_{Consider}(K) = (N_K, M, C)$$

für eine Klasse K mit voll qualifiziertem Namen N_K, der Menge

$$M = \{S_{m1}, S_{m2}, ..., S_{mi}\}$$

an zu überwachenden Methoden $m_i \in$ Methode(K) für $1 \leq i \leq n$ mit Signatur

$$S = (N_m, P_1, P_2, ...)$$

ohne Rückgabetyp aus Methodenname N_m und Parametertyp P_v für $v \geq 0$.
K bestehe aus n Methodendeklarationen. $C = (N_1, N_2, ...)$ definiert einen Filter über aufrufende Klassen der Methoden von K mit voll qualifiziertem Namen N_j.

Für eine Critical-Überwachung unter Berücksichtigung aller Methoden von K genügt offensichtlich

$$E_{Critical}(K) = (N_K) .$$

Zusätzlich enthält die Eingabe weitere zur Laufzeitüberwachung notwendige sowie optionale Parameter. Darunter fallen die Hauptklasse der zu überwachenden Software, Parameter zur Bestimmung der verwendeten Aufzeichnungstechnik und Optionen zur Ausgabe.

3.3.3 Funktion

Dieser Abschnitt beschreibt die Funktionsweise der Laufzeitüberwachung – im folgenden *JavaTracer* genannt –, ohne jedoch auf die genaue Implementierung einzugehen. Auf diese wird in Kapitel 4 näher eingegangen.

In Kapitel 2 wurden mit der Instrumentierung von Software in Abschnitt 2.1 und der Steuerung der Laufzeitumgebung in Abschnitt 2.2 zwei Konzepte zur Gewinnung von Laufzeitinformationen vorgestellt. Der hier beschriebene Ansatz verwendet die Steuerung der Laufzeitumgebung, welche neben den in Kapitel 2 genannten Vorteilen davon profitiert, dass sie ohne Berücksichtigung des genauen Sprachaufbaus auf beliebige andere objektorientierte Programmiersprachen abgebildet werden kann. Die eigentliche Überwachung von Methodeneinsprüngen und -austritten erfolgt dabei mit Hilfe von Breakpoints, die als Debugging-Konzept in gängigen Hochsprachen Unterstützung finden. Abschnitt 4.1.2 des Kapitels Technische Realisierung beschreibt, warum Breakpoints dafür geeignet sind, das Kriterium K_2 zu erfüllen. In Anlehnung zum Überwachungskonzept wird die zu überwachende Software im folgenden als *Debugee* bezeichnet.

Der Ablauf einer JavaTracer-Sitzung lässt sich in die folgenden vier Phasen unterteilen:

- Einlesen der Eingabe
- Erzeugen einer oder Verbinden mit der Virtual Machine des Debugees

- Überwachung des Classloaders der VM des Debugees
- Ausgabe von Laufzeitinformationen

Einlesen der Eingabe

Der JavaTracer erwartet eine Eingabe entsprechend der Definition aus Absatz 3.3.2. Diese enthält alle während der Laufzeitanalyse zu berücksichtigenden Consider- und Critical-Überwachungen sowie den Namen einer der Java Sprachspezifikation [GJS+00] Kapitel 12.1 entsprechenden Hauptklasse des Debugees. Weitere optionale Parameter definieren die Formatierung und den Ort der Ausgabe der Laufzeitinformationen, die eingesetzte Überwachungsstrategie und die Laufzeitumgebung des Debugees.

Die Eingabe wird gelesen und in einer Baumstruktur, dem *Überwachungsbaum B* gespeichert. Dabei enthält der Wurzelknoten den Namen der Hauptklasse sowie die optionalen Parameter. Knoten des Niveaus 1 bilden die Klassen und Knoten des Niveaus 2 die Methoden der Consider-Überwachungen ab. Da Critical-Überwachungen stets alle Methoden der überwachten Klasse berücksichtigen, ist es in diesem Fall ausreichend, eine Liste mit allen Klassen des Critical-Überwachungstyps im Wurzelknoten zu speichern. Der Verzicht auf einen Klassen-Knoten mit einer vollständigen Menge an Methoden-Blättern reduziert die Größe der Datenstruktur.

Klassen-Knoten enthalten als eindeutigen Bezeichner den voll qualifizierten Klassennamen. Methoden-Blätter enthalten als eindeutigen Bezeichner die Methodensignatur ohne Rückgabetyp. Der Verzicht auf den Rückgabetyp ist zulässig, da sich in Java Methoden nicht über ihren Rückgabetyp überladen lassen, wohl aber über ihre Parametersignatur. Aus dem beschriebenen Aufbau folgt, dass der Baum die maximale Tiefe 2 besitzt.

Erzeugen einer oder Verbinden mit der Virtual Machine des Debugees

Sobald der Benutzer die Laufzeitanalyse startet, versucht der JavaTracer je nach Konfiguration der zu verwendenden Laufzeitumgebung eine neue Virtual Machine für den Debugee zu erzeugen oder eine Port-Verbindung mit einer existierenden VM aufzubauen. Scheitert der Versuch, etwa durch eine fehlerhafte Angabe der Hauptklasse, Hostname, Portnummer oder durch das Nicht-Vorhandensein der Virtual Machine, mit der die Verbindung hergestellt werden soll, bricht der JavaTracer an dieser Stelle mit einer entsprechenden Fehlermeldung ab. Ansons-

ten hält der JavaTracer jetzt eine Assoziation auf der Virtual Machine des Debugees im Ruhezustand.

Überwachung des Classloaders der VM des Debugees

Vor der Initialisierung des ersten Threads des Debugees auf seiner Hauptklasse muss die Laufzeitumgebung der Eingabe entsprechend zur Überwachung konfiguriert werden. Die zur Überwachung notwendigen Breakpoints lassen sich allerdings nur auf Klassen setzen, die aufgrund eines Zugriffs nach Kapitel 12.4.1 der Java Sprachspezifikation [GJS+00] durch den Classloader der Virtual Machine geladen und initialisiert worden sind. Deshalb erzeugt der JavaTracer eine Klassenmenge *K,* die aus einer Vereinigung aller durch Critical- und Consider-Überwachungen abgedeckten Klassen gebildet wird. *K* ist Basis einer Classloader Überwachung an der Virtual Machine des Debugees, die einen Thread halten lässt, sobald durch seinen Ablauf eine Klasse aus *K* initialisiert wird.

Kommt es zu einem solchen Ereignis, überprüft der JavaTracer den Wurzelknoten des Überwachungsbaums *B,* ob die initialisierte Klasse *I* in der Liste der Critical-Überwachungen enthalten ist. Trifft diese Bedingung zu, erzeugt er Breakpoints auf allen Methoden von *I.* Trifft diese Bedingung nicht zu, wird der Wurzelknoten auf einen Kindknoten untersucht, der den voll qualifizierten Namen der initialisierten Klasse *I* enthält. Die Blätter dieses Knotens enthalten die Signaturen der Methoden, die für eine Consider-Überwachung mit Breakpoints versehen werden. Da *K* aus einer Vereinigung aller durch Critical- und Consider-Überwachungen abgedeckten Klassen besteht, muss folglich mindestens eines der beiden Ereignisse eintreffen.

Ist der JavaTracer bei Critical-Überwachungen für eine Berücksichtigung geerbter Methoden konfiguriert, iteriert er in diesem Fall durch die Klassenhierarchie von *I* und versieht die Methoden etwaiger Oberklassen ebenso mit Breakpoints. Dabei werden solche Oberklassen ignoriert, deren voll qualifizierter Name auf einen durch Regular Expressions beschriebenen Filter erfasst werden. Diese Filter lassen sich den Anforderungen der Überwachung entsprechend in der Eingabe definieren. Durch den Ausschluss von Klassen, wie etwa der Java Swing Bibliothek zur Darstellung von Benutzeroberflächen, der generischen Oberklasse `java.lang.Object` und anderen für die Analyse nicht hilfreichen Klassen und Bibliotheken, lassen sich Überwachungseffekte weiter reduzieren.

Zusätzlich fügt der JavaTracer den voll qualifizierten Klassennamen der Klasse I als Knoten sowie alle mit Breakpoints versehenen Methoden als Blätter zu einer grafischen Baumdarstellung der GUI hinzu.

Nach dem Setzen der Breakpoints lässt der JavaTracer über die Virtual Machine den angehaltenen Thread in der Ausführung fortfahren.

Ausgabe von Laufzeitinformationen

Erreicht der Programmzähler eines Threads während der Ausführung des Debugees einen Breakpoint, wird der Thread an dieser Stelle durch die Virtual Machine angehalten und ein Breakpoint-Ereignis E auf eine FIFO (First in, First Out) Queue Q_1 abgelegt. Das Halten des Ereignis auslösenden Threads ist erforderlich, damit sich der Inhalt von Laufzeitkeller und Variablen, die durch die Aufzeichnung der Laufzeitinformationen erfasst werden sollen, bis zum Moment der Aufzeichnung nicht ändert. Der Inhalt der Queue wird durch einen Thread des JavaTracers überwacht. Liegt ein Breakpoint-Ereignis auf der Queue, bestimmt der JavaTracer mit Hilfe der Breakpoint-Annotation zunächst seinen Typ.

Ist der Breakpoint vom Methodeneinsprung-Typ, generiert der JavaTracer eine eindeutige Ereignis-Identifikationsnummer ID_e und hängt sie an das Ende einer FILO (First in, Last out) Queue Q_2, die zusammen mit der eindeutigen Identifikationsnummer des auslösenden Threads ID_t als Tupel (ID_t, Q_2) in einer Hash-Map H liegt. Zusammen mit einem Zeitstempel des Ereignisses, Name und Informationen über die übergebenen Argumente an die Methode, eindeutige Identifikationsnummer des Objekts und des aufrufenden Threads, Name und Informationen über die übergebenen Argumente der aufrufenden Methode sowie eindeutige Identifikationsnummer dessen Objekts bildet sie die Ausgabe. Siehe dazu auch Kapitel 3.3.4.

Handelt es sich um einen Breakpoint vom Methodenaustritts-Typ, ermittelt der JavaTracer über ID_t aus H die Queue Q_2, an deren Ende sich die Ereignis-Identifikationsnummer ID_e des korrespondierenden Methodeneinsprungs befindet. Zusammen mit einem Zeitstempel und der Methodensignatur, wobei letztere als optionale und redundante Information nur zum Zwecke der Lesbarkeit durch den menschlichen Benutzer dient, bildet sie die Ausgabe. Siehe hierzu wiederum Kapitel 3.3.4. ID_e wird anschließend aus Q_2 gelöscht. Ferner wird der Repräsentant der Methode in der grafischen Baumdarstellung der GUI des JavaTracers als besucht markiert, falls dieser Breakpoint das erste mal vom Programmzähler erreicht wurde.

Nach erfolgter Ausgabe wird das Breakpoint-Ereignis E aus der Queue Q_1 gelöscht und die Ausführung des angehaltenen Threads fortgesetzt.

Über jede Methode einer Consider-Überwachung lässt sich ferner ein Filter definieren, der die Ausgabe auf bestimmte, aufrufende Klassen beschränkt.

Im Gegensatz zu *regelbasierten* Programmen ist bei *ereignisbasierten* Programmen eine Interaktion – meist mit dem Benutzer – erforderlich, um gewisse Prozesse innerhalb der Software zur Ausführung zu bringen. Enthält die Kandidatenmenge K Klassen, die erst durch das Auslösen eines solchen Interaktionsereignisses instanziiert werden oder zur Ausführung kommen, muss der Software-Architekt diese zur Gewinnung der Laufzeitinformationen entsprechend verursachen. Über die grafische Baumdarstellung von Überwachungsereignissen des JavaTracers kann der Software-Architekt dazu während der Laufzeit des Debugees mitverfolgen, welche Interaktionsschritte Überwachungsereignisse auslösen und welche Überwachungen bereits erfolgreich waren. Diese Darstellung erfüllt das Kriterium K_3.

3.3.4 Ausgabe

Dieser Abschnitt beschreibt den Aufbau der Ausgabe.

Die Ausgabe A_e für Methodeneinsprung-Ereignisse ist ein Tupel

$$A_e = (T, ID_e, O, S, C)$$

wobei Objekt O, Methoden-Signatur S und aufrufende Klasse C durch

$$O = (N_K, ID_o, ID_t),\ S = (N_m, P_1, P_2, ..., P_i) \text{ für } i \geq 0 \text{ mit Parametern}$$
$$P_i = (N_p, ID_p) \vee (N_p, V),\ C = (O, N_c) \vee \emptyset$$

gegeben sind.

Die Ausgabe A_a für Methodenaustritt-Ereignisse ist ein Tupel

$$A_a=(T, ID_e, S)$$

Dabei beschreibt

- T einen Zeitstempel,
- ID_e eine eindeutige Ereignis-Identifikationsnummer,

- N_K den voll qualifizierten Namen der Klasse, von der das Objekt O instanziiert wurde,
- ID_o eine eindeutige Identifikationsnummer für das Objekt O im Speicher
- ID_t eine eindeutige Identifikationsnummer für den Thread, der das Ereignis auf dem Objekt auslöste,
- N_m den Namen der Methode, in der das Ereignis ausgelöst wurde,
- N_p den voll qualifizierten Namen der Klasse, von der das durch den Parameter P_i übergebene Objekt instanziiert wurde,
- ID_p eine eindeutige Identifikationsnummer für das durch P_i repräsentierte Objekt,
- V einen Zahlenwert bei einem primitiven Parametertyp,
- N_c den Namen der Methode, die zum Aufruf von N_m geführt hat.

T, ID_e, N_m und N_K bilden die Information I_1 zur Bestimmung der Reihenfolge der Ausführung der Methoden aus M_v. Zusammen mit der Information I_5, in der Ausgabe vertreten durch ID_t, lassen sich Sequenzen über die zeitliche Abfolge ausgeführter Methoden eines Threads bilden.

Die von A_e unterscheidbare Ausgabe A_a konkretisiert die Information I_1, indem 2-Tupel von A_e und A_a mit identischer ID_e und ID_t Rahmen für die Schachtelung von Methodeneinsprung-Ereignissen aus M_v erzeugen. Alle Methodeneinsprung-Ereignisse innerhalb dieses Rahmens, die sich mit ihm eine gemeinsame ID_t teilen, erfolgten während der Ausführung der Rahmenmethode. A_e und A_a bilden somit eine selektive Aufzeichnung von Push- und Pop- Ereignissen auf dem Laufzeitkeller und beschreiben den Kontrollfluss der Methoden aus M_v.

Die Information I_2 wird unmittelbar repräsentiert durch P, I_3 durch ID_o, sowie I_4 durch C.

Die Ausgabe erfüllt damit das Kriterium K_4.

Kapitel 4
Technische Realisierung

In Kapitel 3 wurden die Kriterien für eine selektive Aufzeichnung von Laufzeitinformationen bei der Ausführung von Java-Programmen sowie die Konzepte zur Erfüllung dieser Kriterien erläutert.

Dieses Kapitel beschreibt die konkrete technische Umsetzung dieser Konzepte durch den JavaTracer. Abschnitt 4.1 stellt zunächst den für die Umsetzung wichtigen Teil der JPDA vor. Der Abschnitt 4.2 vermittelt dann einen Überblick über die Gesamtarchitektur des JavaTracers und beschreibt anhand einzelner Implementierungsdetails die Realisierung des Konzepts mit Hilfe der oben genannten Technologie. Abschließend werden in Abschnitt 4.3 die verwendeten Document Type Definitions (DTD) für Ein- und Ausgabe vorgestellt.

4.1 Verwendete Technologien

In Kapitel 2.2.2 wurde bereits das Java Platform Debugger Architecture (JPDA) zur Steuerung der Java Laufzeitumgebung kurz beschrieben. Der JavaTracer verwendet das Front-end Java Debug Interface (JDI) der JPDA zur Steuerung der Laufzeitumgebung des Debugees, dessen grundlegenden Konzepte in Abschnitt 4.1.1 vorgestellt werden.

4.1.1 Java Platform Debugger Architecture

Das JDI besteht im wesentlichen aus speziellen Interfaces, die Aspekte der Virtual Machine des Debugees in einer hierarchischen, objektbasierten Anordnung repräsentieren. Im einzelnen sind das die Virtual Machine selbst, die geladenen Klassen sowie von ihnen instanzierte Objekte, Attribute als auch laufende Threads. Sie spiegeln debuggerseitig die Eigenschaften von VM und Debugee und nennen sich folglich auch Spiegelklassen (engl. Mirrors).

Konkrete Instanzen der Spiegelklassen erhält man über die Referenzimplementation des JDI. Dazu wird über die Klasse `Bootstrap` eine Instanz von `VirtualMachineManager` angefordert, über die dann mit Hilfe eines

`Connector` Objekts eine neue VM erzeugt (`LaunchingConnector`) oder die Verbindung zu einer bestehenden (`AttachingConnector`) etabliert wird. In beiden Fällen liefert Connector die `VirtualMachine` des Debugees zurück.

Sein Ablauf kann über die Methoden `suspend`, `resume` und `exit` von `VirtualMachine` gesteuert werden. Außerdem liefert `VirtualMachine` eine Instanz von `EventRequestManager`, mit dem sich `EventRequests` setzen lassen. `ClassPrepareRequest` überwacht Klasseninitialisierungen auf dem Classloader, `MethodEntryRequest` und `MethodExitRequest` Methodeneinsprünge und -austritte und `BreakpointRequest` ermöglicht das Setzen von Breakpoints.

Tritt ein Ereignis auf, wird eine Instanz von `Event` auf dem `EventQueue` der `VirtualMachine` abgelegt, simultane Ereignisse werden als `EventSet` gebündelt. Je nach Ereignistyp lassen sich über `ClassPrepareEvent`, `MethodEntryEvent` und `MethodExitEvent` sowie `BreakpointEvent` unmittelbar Informationen über den Ort (`Location`) des Ereignisses im Quellcode, den auslösenden Thread (`ThreadReference`) oder im Fall von `ClassPrepareEvents` der Klasse (`ReferenceType`) gewinnen.

Für die Abfrage von dynamischen Informationen wie den Inhalt des Laufzeitkellers (`StackFrame`) oder die an eine Methode übergebene Argumente (`LocalVariable`) muss der ausführende Thread zuvor gestoppt werden. Das Stoppen lässt sich über eine Option von `EventRequest` automatisieren. Nach der Abfrage wird der Thread über die Methode `suspend` von `ThreadReference` wieder gestartet.

Eine genaue Übersicht über die weiteren Funktionen und Klassen des JDI findet sich in [JPDA]. Im wesentlichen besteht das JDI aus vier Bibliotheken, die vorwiegend Spiegelklassen zur Abbildung von Virtual Machine-, VM-Ereignissen- und Debugee-Aspekten zusammensetzen:

- `com.sun.jdi` enthält Interfaces für die Virtual Machine des Debugees, Threads, Typen und Werte.
- `com.sun.jdi.connect` beinhaltet Interfaces zum Verbindungsaufbau zwischen Debugger und Virtual Machine des Debugees.
- `com.sun.jdi.request` besteht aus Interfaces zur Anforderung von Ereignisüberwachungen an die Virtual Machine des Debugees.

- `com.sun.jdi.event` enthält Interfaces für die von der Virtual Machine erzeugten Ereignisse.

4.1.2 Effiziente Überwachung

Wie oben beschrieben bietet das JDI Funktionen zur Überwachung von Methodeneinsprüngen und -austritten auf instanzierten Objekten einer Klasse. Dazu werden zunächst Instanzen von `MethodEntryRequest` und `MethodExitRequest` angefordert, mit denen sich dann über einen Filter die zu überwachenden Klassen definieren lassen. Dieser Ansatz ist jedoch stark grobgranular, da entweder alle Methoden oder gar keine Methode einer Klasse überwacht wird.

Bei dem im Kapitel 3.3.1 vorgestellten Consider-Überwachungstyp würde diese Vorgehensweise dazu führen, dass selbst bei nur einer beobachteten Methode für alle Methodenaufrufe und -austritte zunächst von der Virtual Machine Ereignisse generiert werden. Der JavaTracer müsste dann in einem weiteren Schritt über einen Filtermechanismus uninteressante Ereignisse herausfiltern, bevor die Laufzeitdaten an die Ausgabeschnittstelle weitergegeben werden. Es ist offensichtlich, dass diese Überwachung von irrelevanten Ereignissen vermeidbare Überwachungseffekte provoziert und gegen das Kriterium K_2 verstößt.

Eine Alternative besteht darin, stattdessen Breakpoints auf die zu überwachende Methode zu setzen. Um ein Ereignis beim Einsprung in die Methode zu erzeugen, wird ein Breakpoint in der erste Zeile des durch den Bytecode der Klasse abgebildeten Quellcodes gesetzt. Dadurch stoppt die Ausführung des Debugee, noch bevor diese Zeile abgearbeitet wird. Für Ereignisse beim Methodenaustritt macht man sich zu Nutze, dass die JPDA am Ende einer Methode hinter der letzten Zeile des abgebildeten Quellcodes eine virtuelle Zeile einfügt. Ein Breakpoint in dieser virtuellen Zeile stoppt erst, nachdem die letzte Zeile ausgeführt worden ist.

Tritt ein Breakpoint Ereignis ein, ist es für die Laufzeitanalyse relevant, unterscheiden zu können, ob dieser Breakpoint einen Methodeneinsprung oder -austritt markiert. Der JavaTracer benutzt hierzu eine Funktion des JDI, mit der sich einzelne Breakpoints um Annotationen ergänzen lassen. Diese Annotationen sind dann bei einem Breakpoint-Ereignis abrufbar und erlauben die Unterscheidung zwischen Methodeneinsprung und -austritt sowie dem Überwachungstyp. Der JavaTracer verwendet diesen Ansatz zur Behandlung von Consider-Überwachungen.

Performanztests[1] zeigen, dass Breakpoints selbst dann noch performanter als `MethodEntryRequests` und `MethodExitRequests` sind, wenn sie auf alle Methoden einer Klasse gesetzt werden (siehe Tabelle 4.1). Im Vergleich wurden die Zeiten zum Starten von Fujaba sowie zum Laden zweier unterschiedlich umfangreicher Projekte mit anschließender Anzeige des Klassendiagramms gemessen. Die Laufzeitüberwachung umfasste dabei 95 Methoden in 5 Klassen, die rudimentäre Funktionen zum Aufbau der GUI, zur Darstellung der Diagramme und als Oberklasse von ASG Elementen erfüllen und demnach vergleichsweise häufig Instanziiert bzw. Ziel eines Methodenaufrufs werden.

Es ist hierbei insbesondere bemerkenswert, dass `MethodEntryRequests` und `MethodExitRequests` auch die Ausführung nicht überwachter Klassen des Debugee deutlich verlangsamen. Eine Laufzeitüberwachung mit Breakpoints hingegen beeinflusst unbewachten Code nur unwesentlich. Entsprechende Vergleichswerte einer Dummyaufzeichnung, bei der die überwachte Klasse nur einmal instanziiert und ausgeführt wurde, sind der Tabelle zu entnehmen (Laden Projekt I D. A.).

Aktion	$t_{w/o}$	t_{break}	t_{events}
Fujaba Start	5,39 Sek.	8,40 Sek.	103,37 Sek.
Laden Projekt I	2,85 Sek.	20,65 Sek.	241,78 Sek.
Laden Projekt II	6,28 Sek	49,58 Sek.	923,03 Sek.
Laden Projekt I D. A.	2,85 Sek.	4,45 Sek.	129,47 Sek.

Tabelle 4.1: Performanzvergleich

Deshalb benutzt der JavaTracer Breakpoints auch für Critical-Überwachungen. In diesem Fall werden Breakpoints mit Hilfe von Debuginformationen des JDI für alle in einer Klasse deklarierten Methoden gesetzt.

Resümierend zusammengefasst erlauben Breakpoints bei der Gewinnung von Laufzeitinformationen ein selektiveres Vorgehen und verhalten sich performanter als die zu diesem Zwecke vom JPDA angebotenen `MethodEntryRequest` und `MethodExitRequest` Ereignisüberwachungen.

1 Testsystem: AMD Athlon 1GHz, FSB 100; 640MB RAM; Win. 98SE; VM Version 1.4.2_04

4.2 Aufbau

Dieser Abschnitt beschreibt die Architektur und einzelne Details der Implementierung des JavaTracers.

Einen Gesamtüberblick über den Aufbau liefet das Package-Diagramm in Abbildung 4.1. Das Paket `tracer` enthält als unterste Ebene der Architektur XML Dateien und die Plugin-Klasse `TracerPlugin` zur Anbindung des JavaTracers an Fujaba. Im Paket `tracer.app` liegt die Hauptklasse `TracerApp` zur Ausführung des JavaTracers als eigenständige Software sowie Klassen zur Initialisierung und Aktualisierung der über die GUI angezeigten Daten. Die eigentlichen GUI Komponenten befinden sich im Paket `tracer.gui`, die aus Gründen der Übersichtlichkeit im Diagramm nicht mit aufgeführt werden. Aktionen des Benutzers auf der GUI zum Einlesen der Konfiguration und der Steuerung der Virtual Machine des Debugees werden über Listener in `tracer.app.action` funktional umgesetzt. Das Packet `tracer.xml` enthält Klassen zur Umsetzung der XML basierten Konfiguration in den Überwachungsbaum, der durch Klassen aus `tracer.xml.observer` realisiert wird, sowie zur Ausgabe der während der Laufzeitüberwachung gesammelten Informationen als XML Dokument. Der Überwachungsvorgang wird über die Klassen aus `tracer.trace` kontrolliert, Überwachungsereignisse werden an `TraceThread` angemeldete Erben des `TraceListener` zur Verarbeitung übergeben.

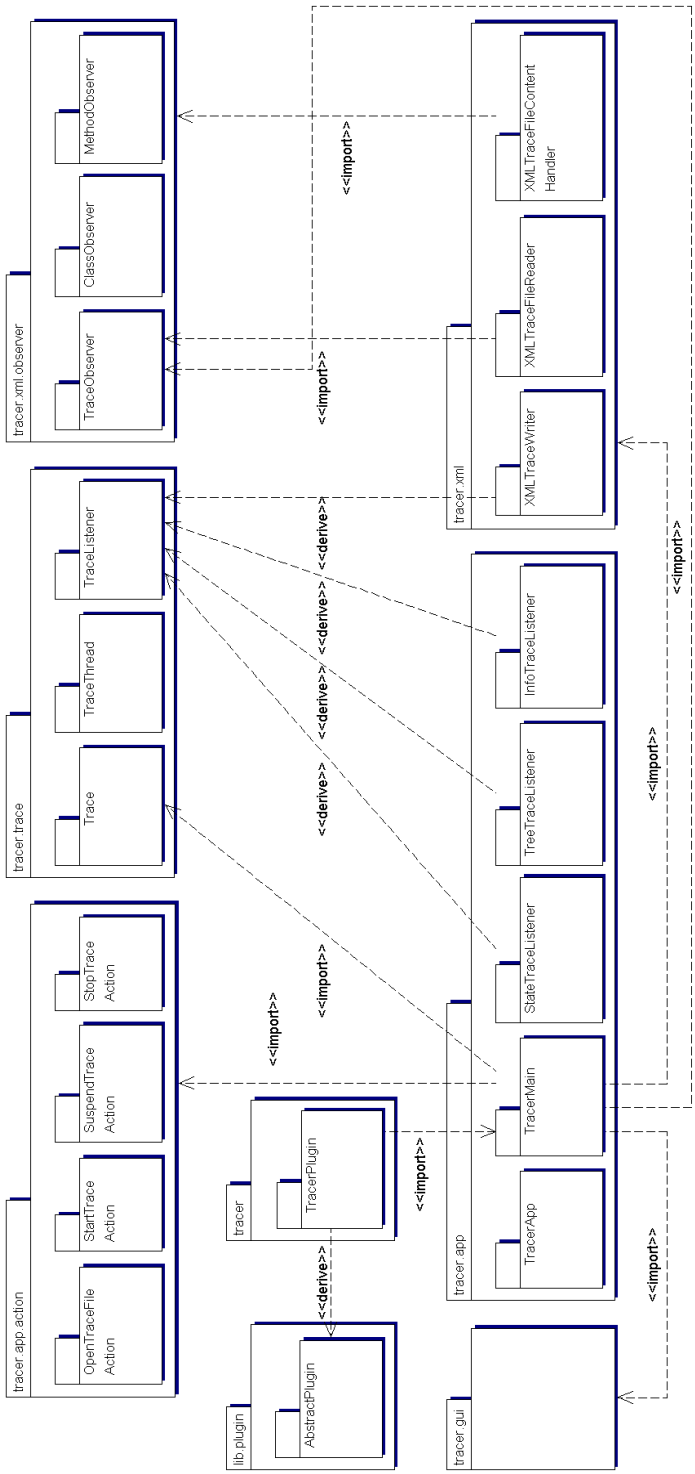

Abbildung 4.1: Package Diagramm des JavaTracers

4.2.1 Einlesen der Eingabe

Die im Abschnitt 3.3.2 beschriebene Eingabe des JavaTracers wird über das Paket `trace.xml` als XML Dokument eingelesen und in einer eigenen Datenstruktur, dem Überwachungsbaum, abgespeichert. Die zur Darstellung der Eingabe verwendete Dokument Type Definition zeigt Abbildung 4.4.

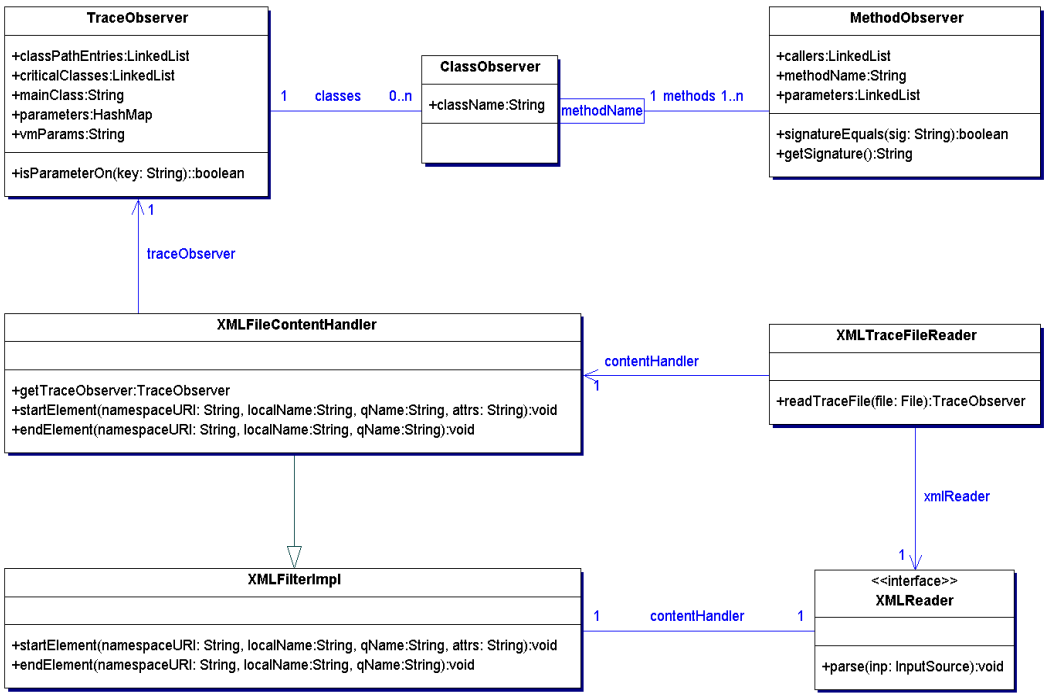

Abbildung 4.2: Klassendiagramm der Pakete trace.xml, trace.xml.observer

Das Einlesen des XML Dokuments erfolgt über die Klasse `XMLFileContentHandler`. Im Wurzelknoten `TraceObserver` des Überwachungsbaums wird neben verschiedenen Parametern zur Laufzeitüberwachung eine Liste über alle Critical-Überwachungen abgelegt. Consider-Überwachungen werden durch `ClassObserver` Instanzen gespeichert, die neben einem voll qualifizierten Klassennamen eine qualifizierte Assoziation auf die zu überwachenden Methoden dieser Klasse halten. Die qualifizierte Assoziation dient zur Berücksichtigung identischer Methodennamen mit unterschiedlichen Signaturen. Entsprechend lässt sich in jeder `MethodObserver` Instanz eine Signatur ablegen. Über die Liste `callers` kann ein Filter über Klassennnamen definiert werden, der selektiv nur die Ereignisse bei der Aufzeichnung berücksichtigt, die durch Elemente des Verhaltensmusters ausgelöst wurden.

4.2.2 Laufzeitüberwachung

Der so erzeugte Überwachungsbaum wird nach dem Starten des Überwachungsvorgangs durch den Benutzer über die Klasse `TracerMain` an die Klasse `Trace` des Pakets `tracer.trace` weitergegeben (siehe Abbildung 4.3).

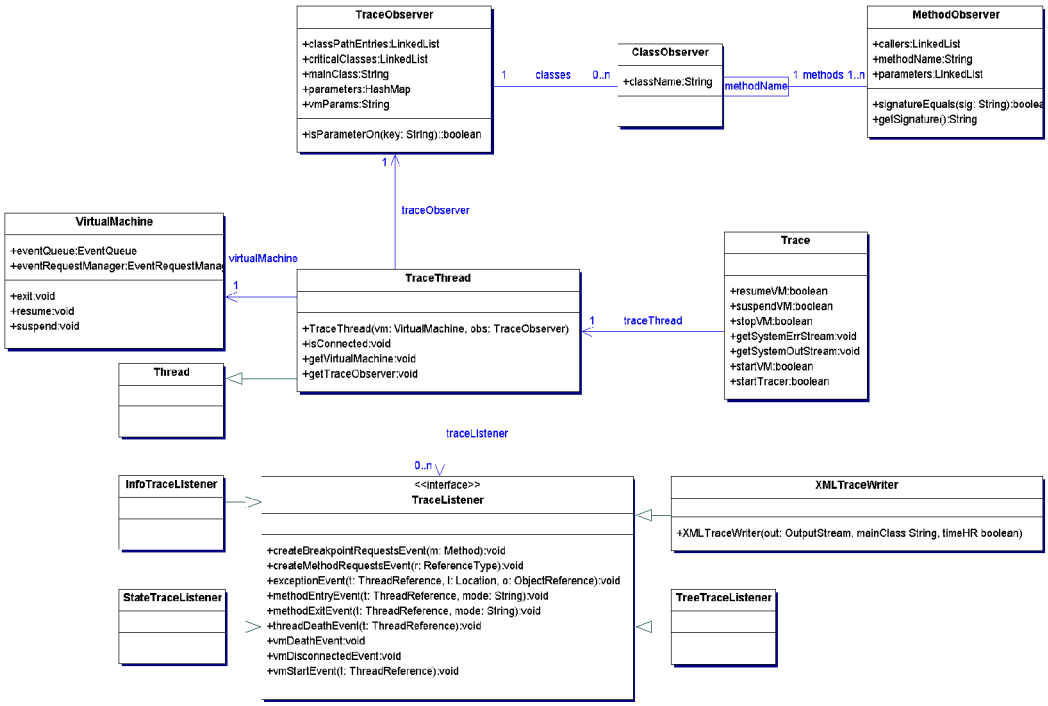

Abbildung 4.3: Klassendiagramm des Pakets tracer.trace

`Trace` kümmert sich zunächst je nach Konfiguration mit Hilfe eines `LaunchingConnectors` oder eines `AttachingConnectors` (siehe Abschnitt 4.1.1) um den Aufbau einer Verbindung zur `VirtualMachine` des Debugees. Die `VirtualMachine` wird im Ruhezustand belassen und zusammen mit dem Überwachungsbaum an `TraceThread` übergeben.

`TraceThread` startet einen `Thread` zur Überwachung der FIFO Ereignisqueue `EventQueue` Q_I der Virtual Machine und initialisiert über die Klassenmenge K aller Consider- und Critical-Überwachungen `ClassPrepareRequests` zur Überwachung des Classloaders. Anschließend registriert `TracerMain` die Implementierungen des `TraceListener` Interface `StateTraceListener`, `InfoTraceListener`, `TreeTraceListener` und `XMLTraceWriter` auf `TracerThread`.

4.2 Aufbau

`TraceListener` und `TracerThread` verwenden das Observer-Entwurfsmuster, um `TraceListener`-Erben im Falle eines Überwachungsereignisses zu informieren. Die Klasse `StateTraceListener` sorgt für eine Konsistenzerhaltung zwischen dem aktuellen Zustand (Gestartet, Gestoppt, Pausiert) der Virtual Machine und der Anzeige des Zustands über die GUI des JavaTracers. Über die Klasse `TreeTraceListener` werden Initialisierungs- und Methodeneinsprungsereignisse in einer grafischen Baumdarstellung der GUI visualisiert, `InfoTraceListener` protokolliert auf der Konsole der GUI gesetzte Breakpoints und das Starten und Stoppen der Virtual Machine. `XMLTraceWriter` erzeugt aus den Überwachungsereignissen die Ausgabe des JavaTracers (siehe Abschnitt 4.2.3).

Die Methoden von TraceListener werden bei folgenden Ereignissen ausgeführt:

- `vmStartEvent` beim Starten der Virtual Machine,
- `vmDeathEvent` beim Terminieren der Virtual Machine nach Ablauf des Programms,
- `vmDisconnectedEvent` durch einen Abbruch der Verbindung zur Virtual Machine, z. B. durch eine Störung in der Portverbindung oder ein vorzeitiges Beenden der VM durch das Betriebssystem,
- `threadDeathEvent` sobald ein Thread terminiert,
- `createBreakpointRequestsEvent` wenn eine im Überwachungsbaum enthaltene Klasse durch den Classloader initialisiert wird und die Konfiguration Breakpoints zur Überwachung vorschreibt,
- `createMethodRequestsEvent` wie oben, aber die Überwachung soll durch Verwendung von `MethodEntryRequest` und `MethodExitRequest` erfolgen,
- `methodEntryEvent` wenn ein Thread in eine überwachte Methode einspringt,
- `methodExitEvent` ein Thread verlässt eine überwachte Methode,
- `exceptionEvent` falls durch die Ausführung des Debugees oder
- `traceExceptionEvent` durch TraceThread eine Exception auftritt.

Anschließend wird die Virtual Machine zur Ausführung des Debugees gestartet. Bei einem durch den Classloader ausgelöstem `ClassPrepareEvent` auf dem

`EventQueue` Q_I überprüft `TraceThread`, ob die gerade initialisierte Klasse I in der Liste der Critical-Überwachungen `criticalClasses` enthalten ist.

Trifft diese Bedingung zu, wird vom `ClassPrepareEvent` eine Instanz von `ReferenceType` abgerufen und über dessen Liste der Methoden (`Method`) von I iteriert. `Method` hält wiederum eine Liste von `Locations`, die Bytecodesequenzen der Methode auf Quellcodezeilen abbilden. Auf der ersten und der letzten, virtuellen Zeile der Methode wird ein Breakpoint gesetzt (siehe Abschnitt 4.1.2), der über eine Annotation als Breakpoint einer Critical-Überwachung sowie als Methodeneinsprungstyp oder -austrittstyp markiert wird. Legt die Konfiguration eine Überwachung aller Oberklassen fest, iteriert `TraceThread` durch die Klassenhierarchie von I und versieht die Methoden der Oberklassen, falls noch nicht geschehen, ebenso mit Breakpoints.

Ist die Klasse I nicht Teil der Critical-Überwachung ruft `TraceThread` über `TraceObserver` den mit I korrespondierenden `ClassObserver` ab und iteriert über dessen Assoziation `methods`. Für jede durch ein `MethodObserver` vertretene Methode werden gemäß Abschnitt 4.1.2 Breakpoints gesetzt und als Breakpoint einer Consider-Überwachung annotiert.

Anschließend ruft `TraceThread` auf allen registrierten `TraceListener` Implementoren die Methode `createBreakpointRequestsEvent` auf, `InfoTraceListener` und `TreeTraceListener` visualisieren das Ereignis daraufhin auf der GUI. Danach wird mit der Ausführung des Angehaltenen Threads fortgefahren.

4.2.3 Ausgabe

Sobald der Programmzähler eines Threads des Debugees einen Breakpoint erreicht, erzeugt das JVMDI ein `BreakpointEvent` auf dem `EventQueue` Q_I und stoppt den betreffenden Thread. `TraceThread` nimmt das Ereignis vom `EventQueue` entgegen und überprüft über die Annotation des Breakpoints zunächst seinen Typ (Methodeneinsprung oder -austritt und Consider- oder Critical-Überwachung). Wurde das Breakpoint Ereignis durch eine Consider-Überwachung erzeugt, ermittelt `TraceThread` den `MethodObserver`, für den dieser Breakpoint gesetzt wurde. Über den Laufzeitkeller (`StackFrame`) wird der Aufrufer ermittelt und mit der im `MethodObserver` definierten Filtermenge `callers` verglichen. Ereignisse, die nicht durch die Filtermenge abgedeckt werden, ignoriert TraceThread.

Anschließend ruft er auf allen registrierten `TraceListener` Implementoren die Methode `methodEntryEvent` oder `methodExitEvent` auf. Das Argument `ThreadReference` der beiden Methoden erlaubt einen Zugriff auf den Laufzeitkeller (`StackFrame`) und damit auf alle weiteren benötigten Daten zur Aufzeichnung des Ereignisses.

Die Ausgabe der Laufzeitinformationen als XML Dokument erfolgt über die Klasse `XMLTraceWriter`, deren Methoden die Hierarchie der in Abbildung 4.5 dargestellten DTD abbilden. Jede Methode verkörpert eine Ebene dieser Hierarchie, erzeugt für diese XML Tags, ergänzt sie um die erforderlichen Daten der Aufzeichnung und führt Methoden zur Generierung der Unterebenen aus. Über das von `TraceThread` übergebene `ThreadReference` [ID_t] Argument greift `XMLTraceWriter` auf den Laufzeitkeller (`StackFrame`) zu. Auf der obersten Ebene des Laufzeitkellers liegen Informationen zur Position des Programmzahlers in Bezug auf die aktuelle Methode [N_m] (`Method`), der die Methode deklarierenden Klasse [N_K] (`ClassType`) sowie des Objekts [ID_o] (`ObjectReference`), auf das der Programmzähler referenziert. `Method` verfügt über eine Liste aller an sie übergebenen Argumente (`LocalVariable`) bestehend aus Typ [N_p] und Wert [V oder ID_p] (`Value`). Die gleichen Informationen lassen sich auf der nächst tieferen Ebene des Laufzeitkellers abfragen und bilden die Informationen über den Aufrufer [C].

Diese Informationen werden wie in Abschnitt 3.3.3 beschrieben um einen Zeitstempel [T] und eine eindeutige Ereignis-Identifikationsnummer [ID_e] erweitert und in einer Datei gespeichert. Dabei kommt die `java.nio` API zum Einstz, um eine vom Betriebssystem optimierte, schnelle Ausgabe über nicht blockierende Dateiausgabefunktionen zu ermöglichen.

4.3 Document Type Definitions

Dieses Kapitel stellt die Document Type Definitions(DTD) zur Beschreibung der Eingabe und Ausgabe des JavaTracers vor.

```
<!ELEMENT Trace (VMConfig, CriticalTrace?,
   ConsiderTrace?)>

<!ELEMENT VMConfig EMPTY>
<!ATTLIST VMConfig mainClass CDATA #REQUIRED
       classPath CDATA #IMPLIED
       classPathDelimiter CDATA #IMPLIED
       vmParams CDATA #IMPLIED
       parameter CDATA #IMPLIED
       traceFile CDATA #IMPLIED
       methodRequestMode CDATA #IMPLIED
       allVisibleMethods CDATA #IMPLIED>

<!ELEMENT CriticalTrace (Callee+)>

<!ELEMENT ConsiderTrace (Class+)>

<!ELEMENT Class (Method+)>
<!ATTLIST Class name CDATA #REQUIRED>

<!ELEMENT Method (Parameter*, Caller*)>
<!ATTLIST Method name CDATA #REQUIRED>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter type CDATA #REQUIRED>

<!ELEMENT Callee EMPTY>
<!ATTLIST Callee name CDATA #REQUIRED>

<!ELEMENT Caller EMPTY>
<!ATTLIST Caller name CDATA #REQUIRED>
```

Abbildung 4.4: DTD der Eingabe

Das Ettikett `VMConfig` bündelt Parameter der Laufzeitüberwachung durch die Virtual Machine. `CriticalTrace` und `ConsiderTrace` trennen die Eingabe in die entsprechenden Überwachungstypen. `Callee` spezifiziert eine Klasse für die Critical Überwachung. `Class` und bildet den Überwachungsbaum im Ni-

veau 1 (`ClassObserver`), `Method`, `Parameter` und `Caller` bilden ihn im Niveau 2 (`MethodObserver`) zur Consider-Überwachung ab.

```
<!ELEMENT TraceResult (ProcessInformation, TraceEvent*, Process
   StopTime)>
<!ELEMENT ProcessInformation EMPTY>
<!ATTLIST ProcessInformation processName CDATA #REQUIRED
       processStartTime CDATA #REQUIRED
       processStartDate CDATA #REQUIRED>

<!ELEMENT TraceEvent (Callee | MethodExit, Caller?)>
<!ATTLIST TraceEvent time CDATA #REQUIRED>

<!ELEMENT Callee (Object, Method)>
<!ATTLIST Callee id CDATA #REQUIRED>

<!ELEMENT Object EMPTY>
<!ATTLIST Object className CDATA #REQUIRED
       hashCode CDATA #REQUIRED
       owningThread CDATA #REQUIRED>

<!ELEMENT Method (Parameter*)>

<!ATTLIST Method methodName CDATA #REQUIRED>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter parameterType CDATA #REQUIRED
       parameterValue CDATA #IMPLIED
       hashCode CDATA #IMPLIED>

<!ELEMENT MethodExit (Method?)EMPTY>
<!ATTLIST MethodExit id CDATA #REQUIRED>

<!ELEMENT Caller (Object, Method)>

<!ELEMENT ProcessStopTime EMPTY>
<!ATTLIST ProcessStopTime processStopTime CDATA #REQUIRED
       processStopDate CDATA #REQUIRED>
```

Abbildung 4.5: DTD der Ausgabe

`TraceEvent` beschreibt ein einzelnes Ereignis der Laufzeitüberwachung. Bei einem Methodeneinsprung bezeichnet `Callee` das Objekt und seine aufgerufene Methode. `Object` enthält den voll qualifizierten Klassennamen

(`className`) und eindeutige Identifikationsnummern für Objekt (`hashCode`) und ausführenden Thread (`owningThread`). `Method` enthält den Namen der aufgerufenen Methode sowie ihre Argumente (`Parameter`) mit Typ (`parameterType`) und Wert (`parameterValue` bei primitiven Typen, sonst `hashCode`). `Caller` definiert über `Object` und `Method` den Aufrufer. Bei einem Methodenaussprung gibt `MethodExit` die gerade verlassene Methode an sowie die eindeutige Identifikationsnummer des korrespondierenden Methodeneinsprung-Abschnitts (`Callee`). Optional enthält `MethodExit` den Namen der Methode und deren Argumente zur besseren Lesbarkeit durch den Menschen.

Kapitel 5
Beispielsitzung

Dieses Kapitel beschreibt eine Beispielsitzung zur Aufzeichnung von Laufzeitinformationen mit dem JavaTracer. Überwacht werden soll ein vermutliches Strategy-Entwurfsmuster im Fujaba Plugin InferenceEngine.

Abbildung 5.1 zeigt die Eingabe für diese Laufzeitüberwachung. Über den Tag `VMConfig` werden Parameter der Laufzeitumgebung und des Überwachungsvorgangs gesetzt. Das Attribut `mainClass` bestimmt die Hauptklasse der zu überwachenden Software, in diesem Fall die Hauptklasse von Fujaba. Anschließend werden alle zur Ausführung benötigten Klassenpfade über das Attribut `classPath` angegeben. Mit `traceFile` wird die Ausgabedatei der Laufzeitüberwachung spezifiziert.

Anschließend werden im Abschnitt `ConsiderTrace` Consider-Überwachungen auf den Methoden `calculateFTT` der Klassen `FTTStrategyJahnke` und `FTTStrategyHeitbreder` sowie die Methode `setFTTStrategy` der Klasse `FPN` gesetzt. Ein Abschnitt mit Critical-Überwachungen (`CriticalTrace`) ist in der Eingabe nicht vorhanden, da sie zur Erkennung eines Strategy-Entwurfsmusters nicht benötigt werden.

```
<Trace>
<VMConfig mainClass="de.uni_paderborn.fujaba.app.FujabaApp"
   classPath="\Fujaba Tool Suite RE 4.1.0\fujaba.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\RuntimeTools.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\jdom.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\junit.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\libCoObRA.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\libXMLReflect.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\log4j-1.2.8.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\upb.jar;
      \Fujaba Tool Suite RE 4.1.0\libs\xerces.jar;
      \Fujaba Tool Suite RE 4.1.0\plugins\InferenceEngine\
         InferenceEngine.jar"
   vmParams="-mx256m"
   traceFile="myTrace.xml"
/>

<ConsiderTrace>
   <Class name="de.upb.inferenceengine
         .fpn.FTTStrategyJahnke">
      <Method name="calculateFTT">
         <Parameter type="de.upb.inferenceengine
            .fpn.Transition"/>
      </Method>
   </Class>
   <Class name="de.upb.inferenceengine
      .fpn.FTTStrategyHeitbreder">
      <Method name="calculateFTT">
         <Parameter type="de.upb.inferenceengine
            .fpn.Transition"/>
      </Method>
   </Class>
   <Class name="de.upb.inferenceengine.fpn.FPN">
      <Method name="setFttStrategy">
         <Parameter type="de.upb.inferenceengine
            .fpn.FTTStrategy"/>
      </Method>
   </Class>
</ConsiderTrace>
</Trace>
```

Abbildung 5.1: Eingabe XML Dokument

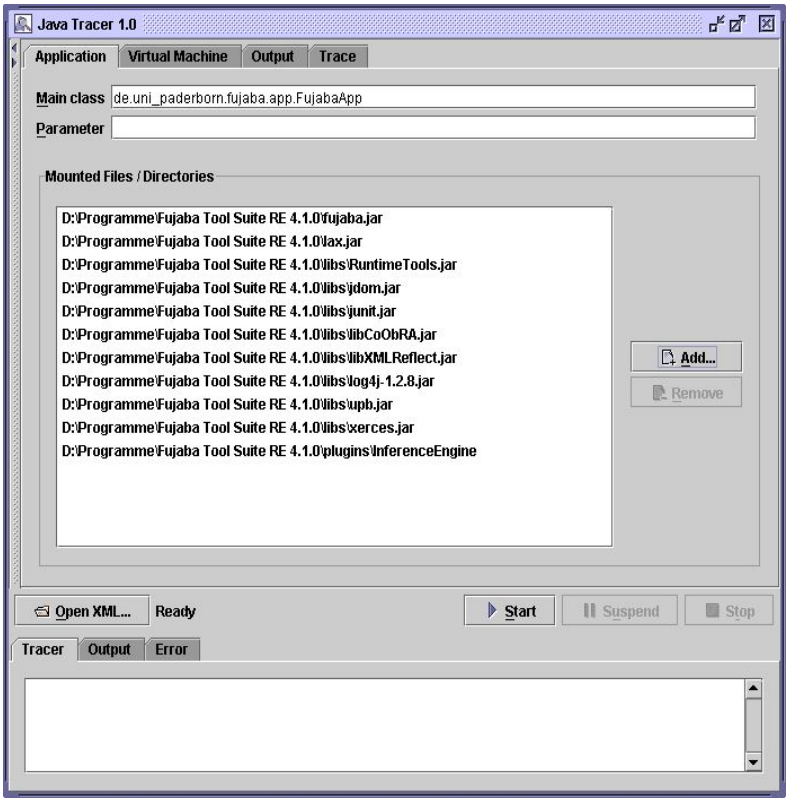

Abbildung 5.2: Konfiguration des JavaTracers

Die Eingabe wird nach dem Starten des JavaTracers über die Schaltfläche *Open XML* eingelesen. Die im Abschnitt `VMConfig` enthaltene Konfiguration der Virtual Machine, Einstellungen zur Ausgabe und zur Überwachungsstrategie sowie die zur Ausführung des Debugees verwendete Hauptklasse und der Klassenpfad wird auf der GUI angezeigt (siehe Abbildung 5.2) und lässt sich gegebenenfalls ändern.

Optional können für die Eingabe der Consider- und Critical-Überwachungen und der weiteren Konfigurationsparameter auch getrennte XML Dokumente verwendet werden, um systemabhängige Einstellungen von der Überwachung zu entkoppeln.

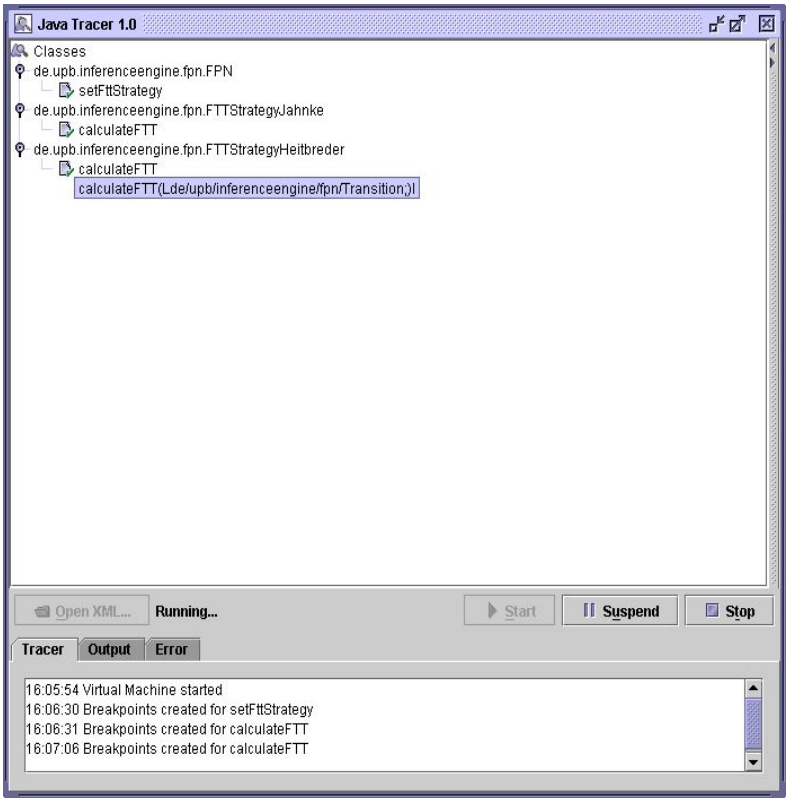

Abbildung 5.3: Laufzeitüberwachung

Sobald die Konfiguration abgeschlossen ist, wird der Überwachungsvorgang über die Schaltfläche *Start* gestartet. Der obere Bereich der GUI (Abbildung 5.3) zeigt dann in einer Baumdarstellung initialisierte Klassen sowie deren Methoden an, falls diese Teil einer Consider- oder Critical-Überwachung sind. Wird eine überwachte Methode erstmalig ausgeführt, wird das Icon der entsprechenden Methode in der Darstellung um ein Häkchen ergänzt. Über diese Information kann der Benutzer mitverfolgen, ob die zur dynamischen Analyse notwendigen Ereignisse bereits aufgezeichnet wurden. Wie in der Abbildung zu sehen ist, erscheint für eine Methode in der grafischen Baumdarstellung ein Tooltip zur Anzeige ihrer Signatur, sobald der Benutzer sie mit der Maus berührt.

Im unteren Teil der GUI befinden sich drei Reiter *Trace*, *Output* und *Error*, die Textausgaben zur Steuerung der Laufzeitumgebung sowie Konsolenausgaben des Debugees beinhalten.

Abbildung 5.3 zeigt die GUI während der Überwachung des Strategy-Entwurfsmuster Kandidaten. Sowohl die Methode `setFttStrategy` der vermut-

lichen Kontextklasse `FPN` zum Setzen der Strategie als auch die durch `calculateFTT` Methoden von `FTTStrategyJahnke` und `FTTStrategyHeitbreder` repräsentierten konkreten Strategien wurden bereits ausgeführt.

```
<TraceEvent time="1091282791430">
   <Callee id="2">
      <Object objectName="de.upb.inferenceengine.fpn.FPN"
         uniqueID="2608" owningThread="Thread-4"/>
      <Method methodName="setFttStrategy"/>
         <Parameter parameterType="de.upb.inferenceengine
            .fpn.FTTStrategy" uniqueID="2609"/>
      </Method>
   </Callee>
   <Caller>
      <Object objectName="de.upb.inferenceengine
         InferenceEngine" uniqueID="2601"
         owningThread="Thread-4"/>
      <Method methodName="run"/>
      </Method>
   </Caller>
</TraceEvent>
<TraceEvent time="1091282791440">
   <MethodExit id="2">
   </MethodExit>
</TraceEvent>
<TraceEvent time="1091282791485">
   <Callee id="3">
      <Object objectName="de.upb.inferenceengine
         .fpn.FTTStrategyJahnke" uniqueID="2609"
         owningThread="Thread-4"/>
      <Method methodName="calculateFTT"/>
         <Parameter parameterType="de.upb.inferenceengine
            .fpn.Transition" uniqueID="2604"/>
      </Method>
   </Callee>
   [...]
</TraceEvent>
```

Abbildung 5.4: Ausschnitt des Ausgabe XML Dokuments

Abbildung 5.4 zeigt einen Ausschnitt der Ausgabe dieser Überwachung. Zunächst wurde die Methode `setFttStrategy` auf `FPN` aufgerufen. Als Argument wurde ein Objekt der Oberklasse `FTTStrategy` mit der ID_o 2609 übergeben. Nachdem der Thread diese Methode beendet hat (`MethodExit`), wird auf

dem Objekt mit der ID_o 2609, das eine Instanz von `FTTStrategyJahnke` beschreibt, die Methode `calculateFTT` ausgeführt.

Der Wechsel der Strategie durch einen Klienten (`InferenceEngine`) und die anschließende Verwendung der gewählten Strategie bestätigen das vermutete Strategy-Entwurfsmuster.

Kapitel 6
Zusammenfassung

Das CASE Tool Fujaba unterstützt zur Vereinfachung von Reverse-Engineering Prozessen eine Entwurfsmustererkennung auf der Basis einer statischen Analyse. Mit Hilfe von Beispielausprägungen untersucht sie den Quellcode auf strukturelle Eigenschaften, die Voraussetzung für ein bestimmtes Entwurfsmuster sind. Wird die Mustersuche fündig, annotiert sie die beteiligten Elemente in einem vorher über die Software erzeugten Klassendiagramm und ermöglicht somit eine abstraktere Sicht auf die Systemarchitektur.

Strukturelle Eigenschaften berücksichtigen jedoch nur einen Teilaspekt von Entwurfsmustern. So sind einige strukturell identisch und lassen sich nur aufgrund ihrer Verhaltensweisen differenzieren. In anderen Fällen sind strukturelle Eigenschaften eines Entwurfsmusters im Quellcode zu finden, obwohl seine Verhaltensweise diesem nicht entspricht. Treffer der statischen Analyse stellen somit nur eine Hypothese über die Existenz eines Entwurfsmusters dar, basierend auf einer partiellen Suche nach notwendigen aber nicht hinreichenden Kriterien.

Deshalb ist eine dynamische Analyse wünschenswert, welche die Ergebnisse der statischen Analyse unter Berücksichtigung des Verhaltens der Software während ihrer Laufzeit validiert. Dazu muss zunächst das Verhalten durch eine Überwachung der Software und einer Aufzeichnung von Laufzeitinformationen festgehalten werden.

Im Rahmen dieser Studienarbeit wurde ein Konzept zur selektiven Aufzeichnung von Laufzeitinformationen bei der Ausführung von Java Programmen entwickelt. Die Eingabe zur Definition der Überwachungskandidaten berücksichtigt ein Verfahren, das aufgrund der Ergebnisse der statischen Analyse als auch unter Berücksichtigung der Anforderungen der dynamischen Analyse die Kandidatenmenge an zu überwachenden Klassen und Methoden möglichst weit einschränkt. Dieser selektive Ansatz reduziert Überwachungseffekte und beschränkt die Ausgabe auf die für eine dynamische Analyse notwendigen Ereignisse.

Die Überwachung der zur Rekonstruktion des Verhaltens erforderlichen Methodeneinsprung- und austrittsereignisse erfolgt mit Hilfe von Breakpoints, einem Konzept, das in allen gängigen Hochsprachen Unterstützung findet. Der hier vorgestellte Ansatz lässt sich somit auch auf andere Programmiersprachen als

Java übertragen. Ein weiterer Vorteil von Breakpoints ist, dass sie als nicht invasive Technik keine Änderungen an der zu analysierenden Software verlangen. Ferner operieren sie für den vorgestellten Anwendungsfall unter Verwendung des JVMDI performanter als eine auf der Granularität von Klassen basierende Aufzeichnung über Methodeneinsprung- und austrittsereignissen, wie sie z. B. in [Meh02a] verwendet wird.

Das Konzept wurde sowohl als Plugin für Fujaba als auch in einer stand-alone Version unter Verwendung der Java Platform Debugger Architecture realisiert und anhand einer konkreten Performanzmessung wurden die oben genannten Aussagen bestätigt. Während der Laufzeit informiert eine grafische Baumdarstellung der bereits initialisierten Klassen und zur Ausführung gekommenen Methoden den Benutzer, ob das Überwachungsziel bereits erreicht wurde. Somit lassen sich in Zukunft die durch die statische Analyse getroffenen Hypothesen über Entwurfsmuster schnell und teilautomatisiert validieren.

Für die dazu notwendige dynamische Analyse existieren schon Ansätze, genaue Konzepte und die Implementierung stehen aber noch aus. Neben einer dynamischen offline Analyse, die wegen der Entkopplung von Aufzeichnung und Analyseschritt in dem hier vorgestellten Ansatz angestrebt wurde, währe auch eine online Analyse denkbar. Bei der Überwachung von mehreren Entwurfsmuster könnten bereits abgeschlossene Analyseschritte die Kandidatenmenge sukzessiv vermindern. Neben der grafischen Baumdarstellung über initialisierte Klassen und Methoden könnte der Benutzer dann direkt anhand der bereits erkannten Entwurfsmuster entscheiden, ob das Analyseergebnis für das angestrebte Reverse-Engineering bereits ausreichend ist und er den Vorgang abbrechen kann.

Literatur

[ADN01] Advanced Database Networking.
http://www.jargon.net/jargonfile/h/heisenbug.html

[CC90] Elliot J. Chikofsky, James H. Cross II: Reverse Engineering and Design Recovery: A Taxonomy. In: IEEE Software 7 (1990), Januar, Nr. 1

[FUJA] FUJABA Tool Suite Developer Team: Fujaba Tool Suite.
http://www.fujaba.de

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software. Reading, MA: Addison Wesley, 1994.

[GJS+00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification. 2nd edition, Addison Wesley, 2000.

[Glas01] Christian Glassner: Dynamic Instrumentation, Monitoring and Steering of Parallel Programs. Johannes Keppler Universität, Linz, Diplomarbeit, August 2001.

[HL] Jens Hohmuth, Prof. Dr. Lenk: Die Debug-Register.
http://www.fh-zwickau.de/doc/prmo/pmtutor/text/deb

[JCLA] ej-technologies: jclasslib.
http://www.ej-technologies.com/products/jclasslib

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

[Kor] Martin Korth: Gameboy Advance Debugger.
http://www.work.de/nocash/gba.htm

[Lew03] B. Lewis: Recording Events to Analyze Programs. In: Proc. Advancing the State-of-the-Art in Run-Time Inspection, 2003.

[Meh02a] Katharina Mehner: JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programms. In: Software Visualizations, LNCS 2269, S. 163-175, 2002: Springer-Verlag Berlin Heidelberg 2002.

[Meh02b] Katharina Mehner: Zur Performanz der Überwachung von Methodenaufrufen mit der Java Platform Debugger Architecture. In S. Diehl (Ed.): Software Visualization International Seminar, Schloss Dagstuhl, Deutschland, Mai 20-25, 2001.

[Pal01] M. Palasdies: Design-Pattern Spezifikation und Erkennung auf Basis von Story-Diagrammen. Universität Paderborn, Paderborn, Diplomarbeit, Mai 2001.

[TOGJ] TogetherJ.
http://www.togethersoft.com

[UML03] Object Management Group: UML 2.0 Superstructure Specification. 2003

[Wen01] Lothar Wendehals: Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets. Universität Paderborn, Paderborn, Diplomarbeit, Oktober 2001.

[Wen03] Lothar Wendehals: Improving Design Pattern Instance Recognition by Dynamic Analysis. In Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, Mai 2003.

[Wen04] Lothar 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, Deutschland, Mai 2004.