Institut für Programmstrukturen und Datenorganisation
Prof. Dr. rer. nat. G. Goos
Universität Karlsruhe
Fakultät für Informatik

Entwicklung eines Debuggers mit Rückwärtsschrittfunktion

Diplomarbeit
von

Hans Kratz

Mai bis November 2004

Betreuer: Prof. Dr. rer. nat. G. Goos
Betreuender Mitarbeiter: Dipl. Inform. Rubino Geiß

Ich versichere, die vorliegende Diplomarbeit selbstständig angefertigt zu haben.
Alle verwendeten Hilfsmittel und Quellen habe ich vollständig aufgeführt.

Karlsruhe, den 5. November 2004.

____

Hans Kratz

Inhaltsverzeichnis

INHALTSVERZEICHNIS 5

1 EINLEITUNG 9

1.1 AUFGABENSTELLUNG 9
1.2 MOTIVATION 9
1.3 ZIELKRITERIEN 10
1.3.1 NAVIGATION IN UND VISUALISIERUNG VON FRÜHEREN PROGRAMMZUSTÄNDEN 11
1.3.2 INTERAKTIVE FEHLERSUCHE 11
1.3.3 PRAXISTAUGLICHKEIT 11
1.3.4 INTEGRATION IN EINE ENTWICKLUNGSUMGEBUNG 12
1.4 GLIEDERUNG 12

2 STAND DER TECHNIK 13

2.1 ÜBERBLICK 13
2.2 TECHNISCHE GRUNDLAGEN VON JAVA DEBUGGERN 13
2.2.1 DEBUGGER-SCHNITTSTELLEN 13
2.2.2 QUELLTEXTBEZUG 14
2.2.3 FEHLERSUCHE MITHILFE VON INSTRUMENTIERUNG 14
2.3 DEBUGGER MIT RÜCKWÄRTSSCHRITTFUNKTION 15
2.3.1 IMPLEMENTIERUNGSTECHNIKEN 15
2.3.2 BDBJ 15
2.3.3 VISICOMP RETROVUE 16
2.3.4 ODB 17
2.4 ZUSAMMENFASSUNG 18

3 ENTWURF 19

3.1 ÜBERBLICK 19
3.2 ANALYSE MÖGLICHER IMPLEMENTIERUNGSANSÄTZE 19
3.2.1 ECHTE ODER SIMULIERTE RÜCKWÄRTSSCHRITTFUNKTION 19
3.2.2 VM-MODIFIKATION ODER INSTRUMENTIERUNG 21
3.2.3 INTERAKTIVE FEHLERSUCHE 21
3.2.4 PROTOKOLLIERUNG 21
3.3 SYSTEMARCHITEKTUR 22
3.4 INSTRUMENTIERUNG 23
3.4.1 FILTERFUNKTION 23
3.4.2 PROTOKOLLIERUNG 26
3.4.3 BEISPIEL: INSTRUMENTIERUNG EINER STATISCHEN METHODE 27

3.5 PROTOKOLLIERUNGSBIBLIOTHEK **29**
3.5.1 MEHRFÄDIGKEIT 29
3.5.2 DATENSTRUKTUREN 30
3.5.3 PROTOKOLLIERUNGSFUNKTIONEN 31
3.5.4 PROTOKOLLZUGRIFFSFUNKTIONEN 31
3.6 FILTERKONFIGURATION **32**
3.7 ZUSAMMENFASSUNG **33**

4 UMSETZUNG **35**

4.1 ÜBERBLICK **35**
4.2 ASPEKTE DER IMPLEMENTIERUNG **35**
4.2.1 BENUTZERSCHNITTSTELLE 35
4.2.2 PROTOKOLLDATEN 36
4.2.3 VERWENDETE EIGENSCHAFTEN MODERNER JAVA-LAUFZEITUMGEBUNGEN 37
4.2.4 INSTRUMENTIERUNG 37
4.3 GELÖSTE PROBLEME **37**
4.3.1 HOTSWAP UND DATENPROTOKOLLIERUNG 38
4.3.2 INITIALISIERUNG DER PROTOKOLLIERUNGSFILTER 38
4.3.3 PROTOKOLLIERUNG VON KONSTRUKTOREN 39
4.3.4 PROTOKOLLIERUNG BOOLESCHER AUSDRÜCKE 39
4.3.5 IMPLEMENTIERUNG KONVENTIONELLER DEBUGFUNKTIONEN 40
4.4 ZUSÄTZLICHE LEISTUNGEN **40**
4.4.1 ERWEITERTE SCHRITTFUNKTIONEN 40
4.4.2 ZUSÄTZLICHE MÖGLICHKEITEN ZUR DATENINSPEKTION 41
4.4.3 STEUERFLUSSINSPEKTION 41
4.4.4 AUSNAHMEHALTEPUNKTE 43
4.4.5 PROTOKOLLIERUNGSLEVEL 44
4.5 ZUSAMMENFASSUNG **44**

5 ERGEBNISSE **45**

5.1 ÜBERBLICK **45**
5.2 VERGLEICHENDE MESSUNGEN **45**
5.2.1 TESTUMGEBUNG UND –METHODE 45
5.2.2 TESTPROGRAMM 46
5.2.3 ABLAUFGESCHWINDIGKEIT IN DER FEHLERVORBEREITUNGSPHASE 46
5.2.4 ABLAUFGESCHWINDIGKEIT WÄHREND DER PROTOKOLLIERUNG 46
5.2.5 SPEICHERVERBRAUCH IN DER FEHLERVORBEREITUNGSPHASE 47
5.3 VERBESSERUNGSMÖGLICHKEITEN **47**
5.3.1 PROTOKOLLIERUNG 47
5.3.2 PROTOKOLLIERUNGSBIBLIOTHEK 48
5.3.3 UNTERSTÜTZUNG VON HOTSWAP 48
5.3.4 INSTRUMENTIERUNG 48
5.3.5 FILTERFUNKTION 49
5.4 ZUSAMMENFASSUNG **49**

6 ZUSAMMENFASSUNG UND AUSBLICK **50**

6.1 Überblick 50
6.2 Rekapitulation der Zielkriterien 50
6.3 Bewertung anhand der Zielkriterien 51
6.4 Nutzen des Werkzeugs in der Praxis 51
6.5 Ausblick 51

LITERATURVERZEICHNIS 53

1 Einleitung

1.1 Aufgabenstellung

Diese Arbeit hat das Ziel, den Prozess der Fehlersuche in Java-Programmen zu verbessern, so dass Entwickler schneller und effizienter die Ursache von Laufzeitfehlern in Programmen aufspüren und beseitigen können. Traditionell werden zur Untersuchung von Laufzeitfehlern Debugger eingesetzt, mit denen der Programmablauf Schritt für Schritt untersucht werden kann. Wir werden einen Debugger für Java entwerfen und implementieren, mit dem der Entwickler nicht nur Schritt für Schritt den Programmablauf untersuchen, sondern auch visualisieren kann, welche Abläufe vor dem aktuellen Ausführungspunkt stattgefunden haben. Um die Produktivität des Entwicklers nicht durch Programmwechsel zu mindern, wollen wir den Debugger in die Entwicklungsumgebung CodeGuide integrieren.

1.2 Motivation

Da Computerprogramme von Menschen entwickelt werden, enthalten sie Fehler. Um die Qualität von Computerprogrammen zu verbessern, kann man an vielen Punkten ansetzen. So versucht man mithilfe von Verifikationswerkzeugen zu beweisen, dass Programme eine formale Spezifikation erfüllen. Da ein solcher Spezifikations- und Verifikationsprozess recht aufwändig ist, kann man ihn nur bei wenigen kritischen Programmen einsetzen, wo der Mehraufwand gerechtfertigt erscheint (z. B. Steuersoftware für Nuklearanlagen). Im Bereich der Softwarearchitektur wurden weitere Techniken entwickelt, um die Zahl der Laufzeitfehler zu verringern. Nach dem Prinzip, dass die Chance einen Fehler in einem Programm zu finden umso höher ist, desto mehr Entwickler sich den Quelltext angeschaut haben, wurden „Peer review" und „Pair programming" eingeführt. „Peer review" bedeutet, dass mindestens ein anderer Entwickler den Quelltext noch einmal durchgeht, um ihn auf mögliche Fehler zu untersuchen. Bei „Pair programming" werden schon während der Entwicklung zwei Programmierer eingesetzt, die zusammen vor einem Rechner das Programm erstellen und sich so gegenseitig kontrollieren können.

Moderne Programmiersprachen haben sowohl die Häufigkeit von Laufzeitfehlern in Computerprogrammen verringert als auch die Untersuchung von Laufzeitfehler vereinfacht. So stürzen in Java programmierte Anwendungen – korrektes Funktionieren der Laufzeitumgebung vorausgesetzt – nicht einfach ab, sondern zeigen definierte Fehlermeldungen. Abstürze mit undefiniertem Verhalten durch Indexüberschreitungen sind ausgeschlossen.

Trotz all dieser Ansätze ist nicht abzusehen, dass mit vertretbarem Aufwand Programme ohne Laufzeitfehler entwickelt werden können. Deshalb wird die Untersuchung von Laufzeitfehlern auch in absehbarer Zukunft ein substantieller Teil der Arbeit des Entwicklers bei jedem Softwareprojekt bleiben. Neben Ausgabe von Zwischenergebnissen auf der Konsole ist das Benutzen eines Debuggers dabei die meistverwendete Methode. Mithilfe von Debuggern kann man durch Anhalten der Programmausführung und Einzelschrittausführung im Quelltext genau überprüfen, was das Programm gerade macht. Das Problem ist, dass man das Programm schon vor Auftreten des Fehlers anhalten muss, um dann die Ausführung im Einzelschrittmodus fortzusetzen. Da aber die vorhergehende Quelltextinspektion keine Anhaltspunkte für die Ursache des Fehlverhaltens geliefert hat, weiß der Entwickler nicht, wo er den Haltepunkt („breakpoint") im Quelltext zu setzen hat, und es ist gut möglich, dass das Programm erst steht, nachdem fehlerhafter Programmcode die Datenstrukturen korrumpiert hat. Außerdem kann es leicht passieren, dass der Entwickler den Aufruf der Methode überspringt („step over"), die für den Fehler verantwortlich ist. Da oft in jeder Zeile bis jeder zweiten Zeile ein Methodenaufruf steht, muss der Entwickler ständig entscheiden, ob er die Ausführung einer Methode untersuchen will oder nicht.

Dieses Problem lösen wir, indem wir dem Entwickler die Möglichkeit geben, den Programmablauf vor dem aktuellen Ausführungspunkt nachzuvollziehen. Dabei sollen die dem Entwickler bekannten Debugger-Metaphern wie Einzelschrittmodus, Anzeige der Werte von Variablen und benutzerdefinierten Ausdrücken beibehalten und wo nötig entsprechend erweitert werden. Um dem Entwickler den Umstieg auf das neue Werkzeug so einfach wie möglich zu machen, sollte der Debugger auch alle konventionellen Funktionen (Haltepunkte, Einzelschritt-Ausführung, Inspektion des Programmzustandes) bieten.

Es soll hier versucht werden, mit einigen Zielkriterien den Minimalanspruch an einen verwendbaren Debugger mit Rückwärtsschrittfunktion zu bestimmen. Dabei darf das übergeordnete Ziel „echter" Praxistauglichkeit nicht aus den Augen verloren werden.

1.3 Zielkriterien

Der Erfolg der Arbeit muss sich an den folgenden Zielkriterien messen lassen:

K1 Navigation in und Visualisierung von früheren Programmzuständen

K2 Interaktive Fehlersuche

K3 Praxistauglichkeit

K4 Integration in eine Entwicklungsumgebung

Diese Zielkriterien werden soweit nötig in den nachfolgenden Abschnitten erläutert und verfeinert, so dass eine einfache Überprüfbarkeit gewährleistet ist.

1.3.1 Navigation in und Visualisierung von früheren Programmzuständen

Die Navigation in vorangegangenen Programmzuständen soll einem Entwickler, der bereits einen konventionellen Debugger benutzt hat, wie eine logische Erweiterung schon bekannter Funktionalität vorkommen. Deshalb müssen Rückwärtsschrittfunktionen angeboten werden, die wie konventionelle Einzelschrittfunktionen arbeiten nur entgegengesetzt zur normalen Programmausführung. Bei der Visualisierung von vorangegangen Programmzuständen beschränken wir uns auf die Werte von Variablen zu den entsprechenden Ausführungszeitpunkten und die Werte von Ausdrucksberechnungen. Frühere Feldwerte müssen nicht betrachtet werden können. Deshalb zerlegen wir dieses Zielkriterium in drei Unterkriterien:

K1-1 Rückwärtsschrittfunktionen

K1-2 Anzeige des Variableninhalts zu einem früheren Ausführungszeitpunkt

K1-3 Anzeige der Werte früherer Ausdrucksberechnungen

1.3.2 Interaktive Fehlersuche

Der Entwickler soll den Debugger auch weiterhin wie einen konventionellen Debugger benutzen, mit ihm also ein Programm anhalten und im Einzelschrittmodus ausführen können. Dadurch wird der Umstieg von konventionellen Debuggern erleichtert. Außerdem können moderne Debugfunktionen, wie der dynamische Austausch von Methoden („Hotswap“ oder „Fix and Continue“) weiter benutzt werden.

1.3.3 Praxistauglichkeit

Die Praxistauglichkeit misst sich vor allem daran, ob Entwickler den Debugger nicht nur mit kleinen Beispielprojekten sondern auch in der täglichen Arbeit mit komplexen Applikationen einsetzen können. Dazu müssen mehrfädige Applikationen unterstützt werden und der Debugger muss mit aktuellen Java-Laufzeitumgebungen (Java 1.4 und Java 1.5) kompatibel sein.

Bei interaktiven Applikationen muss der Entwickler oft viele Schritte manuell durchführen, um eine Fehlersituation zu reproduzieren. So müssen bei Applikationen mit graphischer Oberfläche bestimmte Bedienelemente angewählt und Eingaben gemacht werden. Bei Server-Applikationen müssen bestimmte Anfragen gestellt werden. Damit die Wiederherstellung des Fehlerzustandes nicht zuviel Zeit beansprucht, sollte der Debugger die zu untersuchende Applikation in dieser Fehler-Vorbereitungsphase nicht mehr als nötig verlangsamen und wenig zusätzliche Ressourcen beanspruchen. Diese weichen Kriterien sind nicht gut messbar, deshalb werden genauere Kriterien gewählt, anhand derer eine erfolgreiche Umsetzung leicht zu bewerten ist. So soll in der Fehler-Vorbereitungsphase die Ausführungsgeschwindigkeit maximal 10% geringer sein als während der Ausführung

ohne Unterstützung einer Fehlersuche mit Rückwärtsschrittfunktion. Während der Fehler-Vorbereitungsphase soll von der untersuchten Applikation nicht mehr als 100KB zusätzlicher Speicher benötigt werden.

Um Praxistauglichkeit zu gewährleisten, zerlegen wir das Zielkriterium Praxistauglichkeit also in folgende Unterkriterien:

K3-1 Unterstützung mehrfädiger Applikationen

K3-2 Kompatibilität mit den Java Laufzeitumgebungen Version 1.4 und 1.5

K3-3 Ablaufgeschwindigkeit in der Fehler-Vorbereitungsphase höchstens um 10% geringer

K3-4 Zusätzliche Speichernutzung der Applikation in der Fehler-Vorbereitungsphase kleiner als 100KB

1.3.4 Integration in eine Entwicklungsumgebung

Moderne Entwicklungsumgebungen bieten dem Programmierer einen Produktivitätsvorteil, da man nicht zwischen verschiedenen Programmen zum Editieren, Kompilieren oder Fehler suchen hin- und herwechseln muss. Deshalb soll der Debugger in eine moderne Entwicklungsumgebung integriert werden.

1.4 Gliederung

Kapitel 2 beschreibt den Stand der Technik. Technische Grundlagen von modernen Debuggern für Java werden erläutert und einige bereits verfügbare Debugger mit Rückwärtsschrittfunktion werden daraufhin untersucht, ob sie die Zielkriterien erfüllen.

In Kapitel 3 werden zunächst Techniken zur Implementierung von Debuggern mit Rückwärtsschrittfunktion vorgestellt und auf ihre Tauglichkeit bezüglich der Zielkriterien getestet. Danach wird die Debuggerkomponente entworfen und der Entwurf in einigen wichtigen Bereichen verfeinert.

Kapitel 4 beschreibt die Umsetzung des Entwurfs. Neben Aspekten der Implementierung werden auch gelöste Probleme und zusätzliche Leistungen diskutiert.

In Kapitel 5 werden Messungen vorgenommen, um zu bestätigen, dass alle Zielkriterien erfüllt sind. Außerdem werden Verbesserungsmöglichkeiten der Implementierung aufgezeigt.

Die Bewertung der Arbeit anhand der Zielkriterien und ein Ausblick auf mögliche künftige Entwicklungen erfolgt in Kapitel 6.

2 Stand der Technik

2.1 Überblick

In diesem Kapitel wird zuerst beschrieben, auf welche technischen Grundlagen Debugger für Java zurückgreifen können. Im Anschluss werden schon existierende Debugger mit Rückwärtsschrittfunktion vorgestellt und untersucht, ob diese die Zielkriterien erfüllen.

2.2 Technische Grundlagen von Java Debuggern

Im Gegensatz zu klassischen Programmiersprachen wie C, C++ oder Fortran werden Java-Programme üblicherweise nicht direkt von Quellcode in Maschinencode übersetzt. Stattdessen werden binäre Klassendateien erzeugt, die Strukturinformationen und einen Zwischencode, den sog. „Bytecode" enthalten. Dieser Zwischencode wird dann während der Programmausführung von einer VM („virtual machine") interpretiert oder direkt vor der Ausführung („just-in-time") in Maschinencode übersetzt. Eine solche Übersetzungskomponente nennt man deshalb JIT-Übersetzer. Moderne Ablaufumgebungen wie z. B. die HotSpot VM von Sun Microsystems benutzen profilgesteuerte Optimierung um zu entscheiden, welche Programmteile interpretiert und welche Programmteile in Maschinencode übersetzt und optimiert werden. So wird die unnötige Arbeit des Übersetzens von wenig genutzten Programmteilen in Maschinencode vermieden.

Bis zu Version 1.4 des Java Development Kits (JDK) wurde die Kompilation von Quellcode in Maschinencode während der Fehlersuche abgeschaltet. Seit JDK 1.4 läuft der JIT-Übersetzer auch während der Fehlersuche mit und sorgt so für eine hohe Ablaufgeschwindigkeit auch in der Fehler-Vorbereitungsphase.

2.2.1 Debugger-Schnittstellen

Dadurch, dass die Ausführung von Java-Programmen in einer Laufzeitumgebung stattfindet, kann man nicht auf Debugging-Funktionen des Betriebssystems oder des Prozessors zurückgreifen. Stattdessen stellt die Virtual Machine die Infrastruktur für die Unterstützung von Debuggern bereit.

In Sun's JDK („Java Development Kit") Version 1.1 gab es nur eine undokumentierte Debugschnittstelle, die von Sun's Kommandozeilen-basierten Debugger JDB benutzt wurde. Der Quelltext von JDB war frei verfügbar, so dass man die Funktion der einzelnen

Methoden und Datenstrukturen leicht ableiten konnte. Diese Schnittstelle wurde allerdings nur von Sun's JDK und von einigen anderen Ablaufumgebungen, die von Sun's JDK abgeleitet waren, unterstützt. Insbesondere wurde diese Schnittstelle nicht von Microsoft's Implementierung der Java VM implementiert.

Ab JDK Version 1.3 wurde diese Schnittstelle nicht mehr unterstützt. Stattdessen veröffentlichte Sun die Spezifikation des Schichtenarchitektur-Rahmenwerks JPDA („Java Platform Debugger Architecture“, siehe [JPDA]), das seitdem mit einigen Erweiterungen von allen Sun JDKs unterstützt wird. Auch Drittanbieter von Java VMs wie IBM, Bea und Apple unterstützen diese Schnittstelle.

Die meisten aktuellen Java-Debugger bauen auf einer JPDA-Schicht auf.

2.2.2 Quelltextbezug

Moderne Debugger arbeiten quelltextbezogen. Damit die Verbindung zwischen Zwischencode oder Maschinencode und Quelltext hergestellt werden kann, muss der Übersetzer zusätzliche Debug-Informationen generieren. Üblicherweise muss man dazu den Übersetzer anweisen, Debug-Informationen in den Objektcode einzufügen. Die von Java-Übersetzern erzeugten Klassendateien enthalten immer noch Informationen über Namen, Signaturen und Struktur der Klassen. Das ist nötig, damit man andere Applikationen und Bibliotheken gegen diese Klassendateien binden kann, ohne dass der Quelltext offen gelegt werden muss (vergleiche [LY99]).

Trotzdem müssen auch bei Java-Übersetzern noch zusätzliche Debug-Informationen mit in den Klassendateien gespeichert werden. So wird bei Verwendung der Option für die Generierung von Debug-Informationen in jeder Klassendatei der Name der zugehörigen Java-Quelldatei, die Zuordnung von Zwischencodeindex zu Quellzeile und die Namen und Gültigkeitsbereiche der einzelnen Variablen gespeichert.

Seit Java Version 1.4 wird ein Mechanismus unterstützt, um die Zeilen-basierte Einzelschrittausführung von Programmen zu ermöglichen, die nicht selbst in Java geschrieben sind, sondern in einer Metasprache, die dann von einem Präprozessor in Java übersetzt wird. Beispiele für solche Sprachen sind JavaServer Pages (siehe [JSP]) und JSQL von Oracle (siehe [JSQL]).

Es gibt Bestrebungen, die in Klassendateien gespeicherten Debug-Informationen auszubauen, um es den Entwicklern von Debuggern zu erleichtern, zusätzliche Funktionen wie z. B. Einzelschrittausführung von Ausdrücken zu implementieren (siehe [JAVARFE1]).

2.2.3 Fehlersuche mithilfe von Instrumentierung

Obwohl die meisten Debugger Sun's JPDA Debugger-Schnittstelle nutzen, gibt es doch einige bemerkenswerte Ausnahmen. Eine davon ist das heute nicht mehr vertriebene

Metamata Debug (siehe [METAMATA]). Die Entwickler von Metamata Debug verwenden Instrumentierung um einen von der VM unabhängigen Debugger zu implementieren. Dazu ändern sie den vom Java-Übersetzer erzeugten Bytecode, so dass die Programmsemantik erhalten wird, aber zwischen jeder Quelltext-Anweisung und jeder Modifikation einer lokalen Variable die Debugger-Programmbibliothek aufgerufen wird.

Metamata Debug war damit das einzige Werkzeug, mit dem man Laufzeitfehler in Java Applets sowohl in Microsoft Internet Explorer als auch in Netscape untersuchen konnte. Zwischencode-Instrumentierung auch eine Technik, mit deren Hilfe man Debugger mit Rückwärtsschrittfunktionen implementieren kann. Der offensichtliche Nachteil dieser Technik ist, dass man alle Klassen, die man untersuchen will, instrumentieren muss, bevor sie von der VM geladen werden. Außerdem sind zum Beispiel die Klassen der System-Bibliothek der Java VM im Originalzustand nicht instrumentiert. Aufrufe von Methoden dieser Klassen können also nicht verfolgt werden, sondern werden einfach übersprungen.

2.3 Debugger mit Rückwärtsschrittfunktion

2.3.1 Implementierungstechniken

Um dem Entwickler die Untersuchung früherer Programmzustände zu ermöglichen, müssen während der Ausführung Informationen sowohl über den Programmzustand oder über Zustandsänderungen als auch über den Steuerfluss des Programms protokolliert werden. Diese Protokollierung kann durch die VM erfolgen oder indem der auszuführende Zwischencode so instrumentiert wird, dass er Protokollierungsfunktionen aufruft, die in einer Bibliothek zu Verfügung gestellt werden. Mit den protokollierten Daten können dem Entwickler frühere Programmzustände gezeigt werden. Alternativ kann der Debugger die VM direkt in einen früheren Zustand zurückversetzen.

Die anfallenden Daten können im Hauptspeicher gehalten oder auf Festplatte ausgelagert werden. In welchem Umfang eine Datenprotokollierung stattfindet und ob ältere oder „uninteressante“ Daten verworfen werden ist eine wesentliche Designentscheidung. Kapitel 3 beschreibt die verschiedenen Implementierungstechniken im Detail.

2.3.2 Bdbj

Jonathan Cook hat im Rahmen seiner Diplomarbeit am Hughes Hall College (vergleiche [Cook00]) Bdbj, einen Java-Debugger mit Rückwärtsschrittfunktion, entwickelt. Eine aktualisierte Version inklusive Quelltext ist im Internet verfügbar (siehe [BDBJ]). Neben Rückwärtsschrittfunktion wird auch die Anzeige des Variableninhalts im Rückwärtsschrittmodus unterstützt. Um die Rückwärtsschrittfunktion in Bdbj zu implementieren, wird die frei verfügbare Java VM kaffe (siehe [KAFFE]) modifiziert. Die modifizierte VM speichert dazu alle Daten, die notwendig sind, um Bytecodebefehle rückgängig zu machen, in Puffern. Bdbj zeigt dem Entwickler damit also nicht nur die früheren Zustände,

sondern stellt sie tatsächlich in der VM wieder her. Bdbj erfüllt damit die Zielkriterien **K1-1** (Rückwärtsschrittfunktion) und **K1-2** (Anzeige des Variableninhalts). Die Anzeige von Ausdruckwerten (Zielkriterium **K1-3**) wird nicht unterstützt. Bdbj setzt nicht auf JPDA auf, sondern integriert sich direkt mit kaffe, einer frei verfügbaren Java VM, erlaubt interaktive Fehlersuche und erfüllt damit Zielkriterium **K2**.

In der Praxis lässt sich Bdbj allerdings kaum zur Fehlersuche in aktuellen Java-Applikationen benutzen. Die frei verfügbare und quelloffene kaffe VM ist nicht mit aktuellen Java-Versionen kompatibel. Noch nicht einmal Java Version 1.1 wird vollständig unterstützt. Auch unterstützt Bdbj nicht die Fehlersuche in Anwendungen, die mehrere Ausführungsfäden benutzen. Bei Benutzung von Bdbj wird der JIT-Übersetzer der kaffe VM ausgeschaltet, was zu einer massiven Verlangsamung von rechenintensiven Programmen führt. Bdbj erfüllt also die Zielkriterien **K3-1**, **K3-2** und **K3-3** nicht und ist deshalb nicht praxistauglich.

Die Einbindung von Bdbj in eine Java-Entwicklungsumgebung (Zielkriterium **K4**) wäre zwar möglich, würde sich aber als schwierig gestalten, da die meisten aktuellen Java-Entwicklungsumgebungen die kaffe VM nicht unterstützen. Es müsste also zusätzlich zur Einbindung von Bdbj auch die Unterstützung für die kaffe VM implementiert werden.

Da Bdbj seit Jahren nicht weiterentwickelt wird und die kaffe VM als technologische Grundlage weit vom aktuellen Stand anderer Java-Laufzeitumgebungen entfernt ist, ist die Beseitigung dieser Defizite auch in Zukunft unwahrscheinlich.

2.3.3 VisiComp RetroVue

VisiComp RetroVue war das erste kommerzielle Java-Entwicklungswerkzeug, das eine Rückwärtsschrittfunktion angeboten hat. Leider ist keine Testversion verfügbar, weshalb sich die Analyse auf die wenigen frei verfügbaren Informationen beschränken muss (vergleiche [RETROVUE] und [Berg04]). RetroVue besteht aus zwei Komponenten. Die eine Komponente dient zum Starten und Instrumentieren der zu untersuchenden Applikation. Die Applikation wird damit von RetroVue so instrumentiert, dass während des Ablaufs alle Methodenaufrufe, Variablenzuweisungen, Ausnahmen und auch Informationen über den Steuerfluss protokolliert werden. Die Protokolldaten werden dabei auf der Festplatte gespeichert. Die zweite Komponente ist ein Anzeigeprogramm, mit dem sich der Programmablauf der untersuchten Anwendung nachvollziehen lässt. Dabei ist auch die Möglichkeit gegeben, sich vorwärts und rückwärts im Einzelschrittmodus durch das Programm zu bewegen und sich den Variableninhalt anzeigen zu lassen. RetroVue erfüllt damit die Zielkriterien **K1-1** und **K1-2**. Die Anzeige von Ausdruckswerten (Zielkriterium **K1-3**) wird in der vorliegenden Dokumentation nicht erwähnt. Interaktive Fehlersuche ist mit RetroVue nicht möglich (Zielkriterium **K2**).

Die Untersuchung von mehrfädigen Applikationen wird in der verfügbaren Produktdokumentation explizit als Vorteil für die Verwendung von RetroVue genannt. RetroVue stellt spezielle Ansichten zu Verfügung, mit deren Hilfe der Programmablauf und

insbesondere Blockierungen oder Konkurrenzsituationen, in denen sich mehrere Ausführungsfäden für eine Sperre bewerben, untersucht werden können.

Es gibt keinen sichtbaren Grund, warum die Verwendung von RetroVue an eine bestimmte VM gebunden wäre. Durch die Verwendung von Instrumentierung erscheint es wahrscheinlich, dass RetroVue mit Java 1.4 und Java 1.5 kompatibel ist oder Kompatibilität zumindest leicht herzustellen ist (Zielkriterium **K3-2**).

Die Geschwindigkeit während der Ausführung mit Protokollierung ist vermindert. Filtermöglichkeiten, um die Protokollierung von „uninteressanten“ Codeabschnitten zu unterdrücken, gibt es nicht. Deshalb kann VisiComp RetroVue, das Ziel, den Ablauf der untersuchten Applikation während der Fehler-Vorbereitungsphase um weniger als 10% zu verlangsamen (Zielkriterium **K3-3**), unmöglich erfüllen. Aussagen über den Speicherverbrauch lassen sich ohne eine Testversion nicht machen. Die Integration in eine Java-Entwicklungsumgebung ist nicht vorgesehen (Zielkriterium **K4**).

2.3.4 ODB

ODB (**O**mniscient **Deb**ugger) ist ein experimentelles Werkzeug von Bil Lewis, der auch den Begriff „Omniscient debugging“ geprägt hat (siehe [ODB]). ODB selbst und auch die Quellen sind frei verfügbar unter der GPL (**G**NU **P**ublic **L**icense). Konzeptionell funktioniert ODB ähnlich wie VisiComp RetroVue. Auch bei ODB wird der Zwischencode der auszuführenden Applikation instrumentiert, um Daten zu sammeln, die dann später in einer grafischen Benutzeroberfläche angezeigt werden. Es ist möglich, Variableninhalte zu untersuchen und sich mit Einzelschrittfunktionen vorwärts und rückwärts im Steuerfluss zu bewegen. Die Zielkriterien **K1-1** und **K1-2** werden damit erfüllt. Die Anzeige von tatsächlichen Ausdruckswerten (Zielkriterium **K1-3)** ist nicht möglich, es ist aber möglich einfache Java-Ausdrücke einzugeben, die dann unter Berücksichtigung der aufgenommenen Daten ausgewertet werden.

Anders als bei RetroVue gibt es kein getrenntes Programm zur Datensammlung. Sobald man ODB mit den entsprechenden Kommandozeilenoptionen startet, erscheint ein Kontrollfenster und die zu untersuchende Applikation wird gestartet. Wenn die Applikation beendet oder mithilfe des Kontrollfensters abgebrochen wird, startet ODB die grafische Benutzeroberfläche in dem gleichen Prozess. Die Möglichkeit, die Protokolldaten auf Festplatte zu speichern, besteht nicht. Mit ODB ist keine interaktive Fehlersuche möglich (Zielkriterium **K2**).

ODB unterstützt die Untersuchung von mehrfädigen Applikationen (Zielkriterium **K3-1**) und auch die aktuellen Java-Versionen 1.4 und 1.5 (Zielkriterium **K3-2**).

In der Grundeinstellung instrumentiert ODB nur die Klassen, die sich im selben Paket befinden wie die Startklasse. ODB bieten Filtermöglichkeiten, mit denen der Entwickler manuell bestimmen kann, welche Methoden in welchen Klassen instrumentiert werden und für welche Methoden Daten gesammelt werden. Man kann ODB so konfigurieren, dass nicht gleich von Anfang an Daten gesammelt werden. Der Entwickler kann die Daten-

sammlung dann manuell über das Kontrollfenster einschalten oder bestimmen, dass die Datensammlung bei der ersten Ausgabe der Applikation automatisch gestartet wird. Wie im Kapitel 5 gezeigt wird, erfüllt ODB das Zielkriterium **K3-3** (nahezu unverminderte Ablaufgeschwindigkeit während der Fehlervorbereitungsphase) nicht.

Eine Integration in eine Entwicklungsumgebung ist bisher nicht verfügbar. Dadurch, dass ODB im Quelltext verfügbar ist, ist eine Integration bei entsprechendem Interesse von Nutzern leicht zu bewerkstelligen (Zielkriterium **K4**).

Da ODB frei verfügbar ist, wird er in dieser Arbeit zu Vergleichszwecken im Kapitel 5 herangezogen.

2.4 Zusammenfassung

Wir haben die technischen Grundlagen von Java-Debuggern vorgestellt. Bereits verfügbare Debugger mit Rückwärtsschrittfunktion wurden auf die Erfüllung der Zielkriterien untersucht. Keiner der bisher entwickelten Debugger erfüllt die Zielkriterien auch nur annähernd, wie sich der nachfolgenden Tabelle entnehmen lässt. Insbesondere ist es mit keinem der untersuchten Werkzeuge möglich, komplexe Applikationen zu untersuchen und gleichzeitig auch interaktive Fehlersuche zu benutzen.

		BDBJ	**RetroVue**	**ODB**
K1-1	Rückwärtsschrittfunktionen	+	+	+
K1-2	Anzeige von früheren Variablenwerten	+	+	+
K1-3	Anzeige von früheren Ausdruckswerten	-	?	-
K2	Interaktive Fehlersuche	+	-	-
K3-1	Unterstützung von Mehrfädigkeit	-	+	+
K3-2	Kompatibilität mit Java 1.4 und 1.5	-	(+)	+
K3-3	Nahezu unverminderte Geschwindigkeit	-	-	-
K3-4	Geringer Speicherbedarf	?	?	?
K4	Integration in eine Entwicklungsumgebung	-	-	-

Ist ein Ergebnis in der Tabelle in Klammern gesetzt, so ist es nicht unmittelbar aus der verfügbaren Information ableitbar aber sehr wahrscheinlich.

3 Entwurf

3.1 Überblick

In diesem Kapitel werden zunächst mögliche Implementierungsansätze für einen Debugger mit Rückwärtsschrittfunktion vorgestellt und ihre Tauglichkeit zur Erfüllung der Zielkriterien untersucht. Dann wird die Architektur des Gesamtsystems vorgestellt. Im Anschluss werden einige wichtige Teilbereiche genauer betrachtet und der Entwurf entsprechend verfeinert. Schon während der Entwurfsphase werden bestimmte Entwurfsentscheidungen mithilfe von Prototypen überprüft.

Wir wollen das Ziel, ein in der Praxis einsetzbares Werkzeug zu entwickeln, nicht aus den Augen verlieren. Entwurfsentscheidungen, die sich nicht direkt mit der Erfüllung der Zielkriterien begründen lassen, wurden unter diesem Gesichtspunkt getroffen.

3.2 Analyse möglicher Implementierungsansätze

Grundsätzlich kann die Navigation in und die Anzeige von früheren Programmzuständen auf verschiedene Arten implementiert werden. Die nachfolgend vorgestellten Implementierungsansätze werden danach beurteilt, ob sie für die Erfüllung der Zielkriterien geeignet sind.

Zur Erfüllung von Zielkriterium **K4** (Integration in eine Entwicklungsumgebung) wurde die Java-Entwicklungsumgebung Omnicore CodeGuide gewählt (siehe [CG]). CodeGuide bietet bereits eine existierende auf JPDA basierende Debuggerschnittstelle und eine entsprechende Benutzeroberfläche, die einen guten Rahmen für diese Arbeit bietet. Außerdem ist der Autor als Mitentwickler mit der Architektur und Implementierung von CodeGuide vertraut.

3.2.1 Echte oder simulierte Rückwärtsschrittfunktion

Um dem Entwickler frühere Zustände der Java-Laufzeitumgebung zu zeigen und eine Rückwärtsschrittfunktion zu ermöglichen, kann ein Debugger entweder die Ausführungsrichtung umkehren und bisher erfolgte Veränderungen des Zustands rückgängig machen oder dem Entwickler einfach nur einen früheren Zustand anzeigen. Von allen bisher betrachteten Debuggern mit Rückwärtsschrittfunktion verfolgt nur BDB den ersten Ansatz. Die Umkehr der Ausführungsrichtung und die tatsächliche Wiederherstellung eines

früheren Zustandes haben den Nachteil, dass nur Zustandsänderungen innerhalb der VM rückgängig gemacht werden können. Befehle, die den Zustand außerhalb der VM verändern (z.B. Ein-/Ausgabe, Aufruf von nicht in Java implementierten Bibliotheksfunktionen, Datenbanktransaktionen, etc.) können nicht oder zumindest nicht mit vertretbarem Aufwand rückgängig gemacht werden. Wenn dann die Ausführungsrichtung wieder umgekehrt wird, werden diese Funktionen erneut ausgeführt, was bestenfalls keinen Effekt hat und schlimmstenfalls zu Datenverlust führt. Es kann deshalb auch nicht sichergestellt werden, dass eine erneute Vorwärts-Ausführung von Programmcode zu dem gleichen Ergebnis führt oder auch nur denselben Steuerfluss nimmt.

Damit eine Rückwärtsausführung zumindest für alle VM-internen Zustandsänderungen funktioniert, muss grundsätzlich die Ausführung aller Funktionen protokolliert werden, ohne dass die Möglichkeit besteht, bestimmte Codeabschnitte (z. B. Bibliotheksfunktionen) auszunehmen. Um eine Rückwärtsschrittfunktion auf Zeilenebene mit Variablenanzeige zu ermöglichen, muss entweder die VM selbst modifiziert werden oder es müssen komplexere Instrumentierungen vorgenommen werden als bei einer reinen Protokollierung. Denn dann müssen in jeder Methode nicht nur Änderungen protokolliert sondern auch Möglichkeiten bereitgestellt werden, um einen früheren Zustand wiederherzustellen. Es ist unwahrscheinlich, dass mit dieser Technik eine ausreichende Ausführungsgeschwindigkeit in der Fehler-Vorbereitungsphase (Zielkriterium **K3-3**) erreicht werden kann.

Ein weiteres Problem ist, dass eine Parallelausführung mehrerer Ausführungsfäden praktisch nicht mehr stattfinden kann. Alle Zustandsänderungen müssen serialisiert werden, damit sie Semantik-erhaltend rückgängig gemacht werden können. Grundsätzlich können mehrere Zustandsänderungen in einer Zeile stattfinden, die der Entwickler rückwärts überspringen können soll. Man kann aber nicht zulassen, dass Ausführungsfadenwechsel bei der serialisierten Ausführung nur zwischen Zeilen stattfinden, da bestimmte Java-Anweisungen den Ausführungsfaden blockieren bis ein anderer Ausführungsfaden beispielsweise eine Sperre löst. Deshalb kann eine Rückwärtsschrittfunktion, die in nur einem Ausführungsfaden einen Rückwärtsschritt ausführt, nicht implementiert werden.

Der Implementierungsansatz der Ausführungsumkehrung hat allerdings auch Vorteile. Er entspricht nämlich der intuitiven Vorstellung der meisten Entwickler. Entwickler hätten auch gerne die Möglichkeit, den Wert von Variablen zu ändern, nachdem sie einige Rückwärtsschritte ausgeführt haben, um zu sehen, wie die Ausführung dann verlaufen wäre. Das wäre mit diesem Ansatz möglich. Es ist auch denkbar, dass man so Programmbefehle rückgängig machen kann, bis man sich vor dem letzten Methodenaufruf befindet, und dann den Zwischencode für die Methode durch die ab Java Version 1.4 mögliche dynamische Codeersetzung austauscht. So kann man Fehler praktisch „ungeschehen" machen und den fehlerhaften Code ersetzen, bevor man ihn nochmals ausführen lässt.

Insgesamt ist dieser Ansatz zur Erfüllung der Zielkriterien aber nicht geeignet, insbesondere auch weil er sich ohne VM-Modifikation nur sehr umständlich verwirklichen lässt.

3.2.2 VM-Modifikation oder Instrumentierung

VM-Modifikation scheidet als Implementierungstechnik praktisch aus, da die Quellen für aktuelle Java-Laufzeitumgebungen nur unter einer sehr restriktiven Lizenz verfügbar sind (vergleiche [SCSL]). Außerdem verfügen alle modernen Java-Laufzeitumgebungen über einen optimierenden JIT-Übersetzer, der aus Java-Zwischencode schnellen Maschinencode erzeugt. Eine VM-Modifikation, die Protokollierung auf VM-Ebene ermöglicht, ohne den JIT-Übersetzer auszuschalten, scheint kaum zu verwirklichen. Ohne JIT-Übersetzer wird die Ablaufgeschwindigkeit aber stark vermindert, so dass Zielkriterium **K3-3** nicht erfüllt werden kann.

Es stehen zwei Instrumentierungstechniken zur Auswahl: Quelltextinstrumentierung und Zwischencodeinstrumentierung. Quelltextinstrumentierung wird bei Java selten verwendet, da der Java-Zwischencode noch genügend Informationen für eine Instrumentierung enthält und so keine Quelltextanalyse implementiert werden muss. Für die Realisierung einer Ausdruckswertanzeige (Zielkriterium **K1-3**) sind die in Klassendateien enthaltenen Informationen allerdings unzureichend. Durch die Verwendung von CodeGuide als Basis besteht die Möglichkeit, den integrierten inkrementellen Java-Übersetzer (vergleiche [Strein02]) anzupassen, so dass direkt instrumentierter Zwischencode erzeugt werden kann. Dadurch ist es möglich, zusätzlich zur Anzeige von Ausdruckswerten auch andere Debuggerfunktionenen, wie z. B. einen Einzelschrittmodus auf Ausdrucksbasis zu implementieren.

3.2.3 Interaktive Fehlersuche

Interaktive Fehlersuche lässt sich wie bei Metamata Debug allein durch Instrumentierung verwirklichen. Das ist dann aber immer mit Leistungseinbußen im Vergleich zur uninstrumentierten Ausführung verbunden, und neue Funktionen wie der dynamische Austausch des Zwischencodes einer Methode („HotSwap“) können nicht unterstützt werden. Im Gegensatz zu früheren Versionen der Java-Laufzeitumgebung unterstützen alle Sun Java-Laufzeitumgebungen ab Version 1.4 das Ausführen einer Applikation im JPDA-Debugmodus ohne Geschwindigkeitseinbußen. Da in CodeGuide schon eine gut funktionierende JPDA-Bibliothek existiert hat, war es nahe liegend JPDA und Instrumentierung miteinander zu kombinieren, um interaktive Fehlersuche zu ermöglichen. So können die Zielkriterien **K2** und **K3-3** verwirklicht werden.

3.2.4 Protokollierung

Durch viele Java Zwischencodebefehle geht Information über den Zustand vor Ausführung des Befehls verloren (Beispiel: der Zwischencodebefehl PUTFIELD ändert den Wert einer Instanzvariablen, ohne den vorherigen Wert zu sichern), so dass der vorherige Zustand nicht angezeigt oder wieder hergestellt werden kann, wenn diese Information nicht gespeichert wird. Der naive Ansatz, dieses Problem zu lösen, ist, den kompletten Zustand der VM vor jeder Codesequenz, die atomar rückgängig gemacht werden soll, zu sichern.

Dieser Ansatz lässt sich in der Praxis aufgrund des hohen Ressourcenbedarfs nicht verwirklichen. Stattdessen kann man natürlich auch nur die Werte sichern, die in dem darauf folgenden Codeabschnitt geändert werden.

Die Protokolldaten können entweder im Speicher gehalten oder auf Festplatte ausgelagert werden. Die Vorteile der Auslagerung auf Festplatte sind, dass die in der Protokolldatei vorhandenen Daten immer wieder angezeigt werden können (dann allerdings ohne Interaktivität) und dass ein nahezu beliebig großes Protokoll gespeichert werden kann. Leider würde dadurch die Ausführungsgeschwindigkeit stark herabgesetzt. Die Lagerung der Protokolldaten im Hauptspeicher ist deshalb vorzuziehen. Grundsätzlich können die Daten dabei entweder in dem Speicherbereich der zu untersuchenden Anwendung gehalten werden oder über Prozesskommunikation zur Entwicklungsumgebung übermittelt werden. Die erste Methode bietet die größere Ausführungsgeschwindigkeit, aber es muss eine Möglichkeit gefunden werden, effizient auf die Daten zuzugreifen, sobald diese zur Darstellung eines früheren Systemzustandes benötigt werden.

3.3 Systemarchitektur

Das Gesamtsystem lässt sich in mehrere Komponenten aufteilen, die genau abgegrenzte Funktionen erfüllen und über klar definierte Schnittstellen miteinander kommunizieren. Von der Benutzerschnittstelle (GUI) der Entwicklungsumgebung (IDE) aus werden der Übersetzer und der Debugger gesteuert. Der Übersetzer generiert die Protokollierungsbibliothek und instrumentierte Klassendateien aus dem Quelltext der Applikation. Außerdem stellt er dem Debugger Funktionen zu Verfügung, mit deren Hilfe der Debugger die protokollierten Daten mit Codeabschnitten in Beziehung setzen kann. Sobald der Entwickler die zu untersuchende Anwendung von der Entwicklungsumgebung aus startet, kontrolliert die Debugger-Komponente die Laufzeitumgebung der gestarteten Applikation über JPDA. Instrumentierte Klassen der Applikation protokollieren den Ablauf der Codeausführung unter Verwendung der Protokollierungsschnittstelle der Protokollierungsbibliothek.

Wenn der Entwickler auf frühere Programmzustände zugreifen möchte, dann ruft die Debugger-Komponente über JPDA Methoden in der Zugriffsschnittstelle der Protokollierungsbibliothek auf und setzt die zurückgegebenen Daten mit Hilfe der Übersetzer-Komponente in nützliche Informationen um. Diese Informationen gibt sie dann an die Benutzerschnittstelle der Entwicklungsumgebung zurück.

Bibliotheksklassen (z. B. Klassen der Java-Laufzeitbibliothek) werden nicht instrumentiert. In diesen Klassen können also nur konventionelle Debuggerfunktionen verwendet werden.

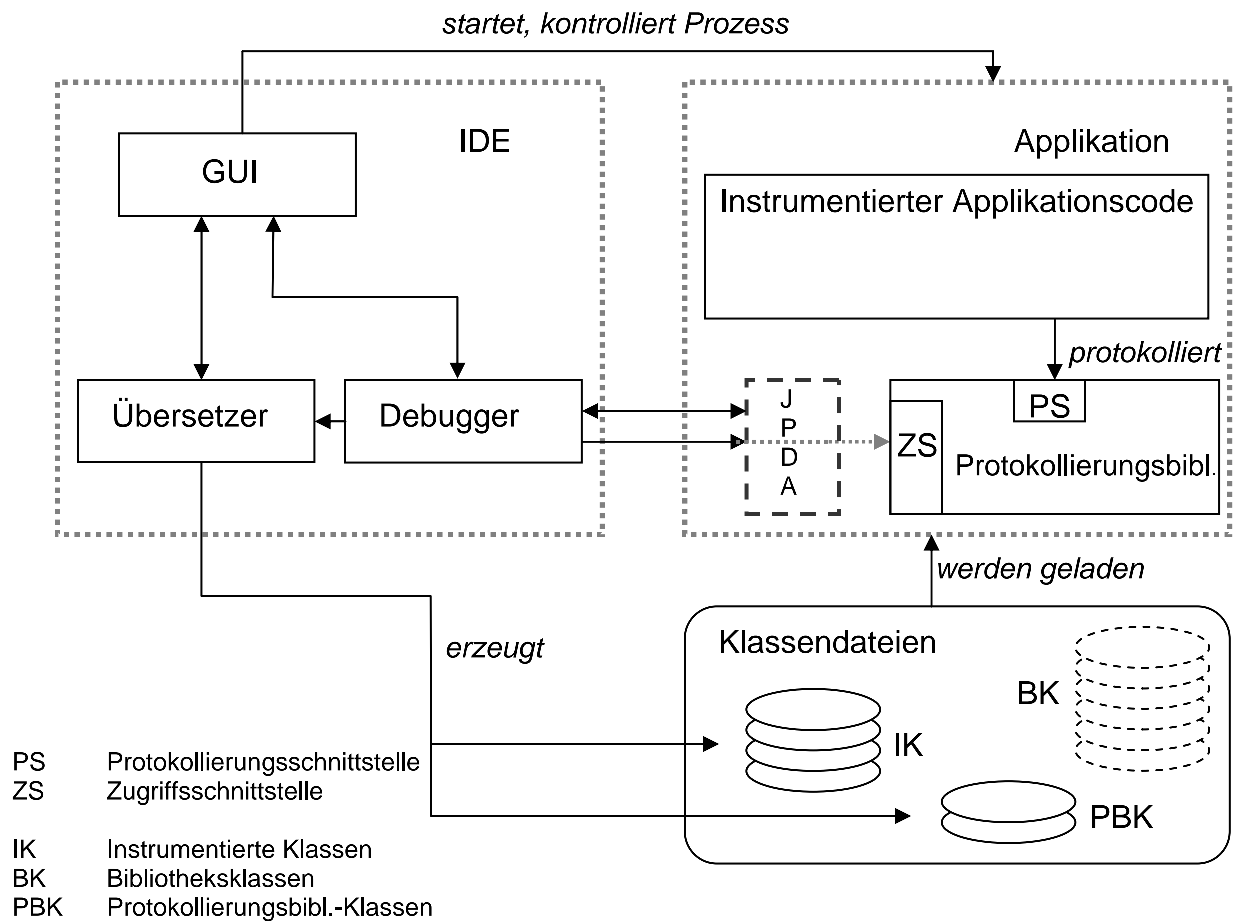

Abbildung 3-1 Architektur des Gesamtsystems

Abbildung 3-1 stellt die Architektur des Gesamtsystems schematisch dar.

3.4 Instrumentierung

3.4.1 Filterfunktion

Naive Implementierungen scheitern leicht daran, dass die Instrumentierung die auszuführende Applikation auch in der Fehler-Vorbereitungsphase zu sehr verlangsamt. Es muss also ein Weg gefunden werden, wie in der Fehlervorbereitungsphase nicht nur die Protokollierung vermieden sondern sogar die Ausführung von zusätzlichem Zwischencode minimiert werden kann. Nur so kann Zielkriterium **K3-3** erfüllt werden.

Um die Ausführungsgeschwindigkeit nicht mehr als nötig zu beeinträchtigen, kann man dem Entwickler die Möglichkeit geben festzulegen, welche Codebereiche „interessant“, also zu protokollieren sind. Dabei ist es sinnvoll, beim Entwurf eine möglichst feine

Kontrolle der Instrumentierung vorzusehen. Deshalb sollte die Protokollierung für einzelne Methoden getrennt und nicht nur auf Klassenebene an- und ausgeschaltet werden können.

Um eine dynamische Änderung der Menge der instrumentierten Methoden zu ermöglichen, kann die „HotSwap"-Funktion verwendet werden, die JPDA seit Java Version 1.4 bereitstellt. Mit HotSwap lässt sich der Zwischencode einer Klasse zur Laufzeit ersetzen. Sollte ein Ausführungsfaden noch mit der Ausführung einer Methode beschäftigt sein, die mit HotSwap ersetzt worden ist, so wird er weiterhin die alte Version der Methode ausführen und erst bei dem nächsten Aufruf dieser Methode den neuen Zwischencode abarbeiten. Leider ist die HotSwap-Funktion recht langsam. Wenn eine größere Menge von Klassen geändert wird, dann kann der Methodenaustausch mehrere Minuten in Anspruch nehmen. Das ist dem Entwickler in der Praxis nicht zuzumuten.

Es muss also eine Möglichkeit gefunden werden, die Instrumentierung von Methoden dynamisch an- und auszuschalten, ohne dass der Zwischencode modifiziert werden muss. Minimale Leistungseinbußen können dafür in Kauf genommen werden, solange Zielkriterium **K3-3** erfüllt wird. Die Grundidee der hier vorgestellten Technik ist, dass das Auswerten von einfachen Bedingungen durch die Optimierungen der HotSpot VM kaum Zeit beansprucht, vor allem wenn die Auswertung nur einmal pro Methodenaufruf erfolgt. Es muss die Möglichkeit bestehen, über JPDA den Wahrheitswert der Bedingung zu ändern, um eine dynamische Änderung der Filtermenge zu ermöglich. Es liegt nahe, für jede Methode ein statisches Feld des Typs `boolean` zu erzeugen, dass die Ausführung von protokollierendem oder uninstrumentiertem Code steuert. Dieses Feld kann dann von der Debuggerkomponente über JPDA geändert werden.

Codeausschnitt 3-1 Beispielklasse ohne Instrumentierung

```
public class Foo
{
     public void bar()
     {
          // Methodencode
     }
}
```

Codeausschnitt 3-1 zeigt eine Beispielklasse, die von dem instrumentierenden Übersetzer in Zwischencode übersetzt wird, der zu dem Quelltext in Codeausschnitt 3-2 äquivalent ist.

Codeausschnitt 3-2 Beispielklasse mit Instrumentierung (Übersetzungsschma 1)

```
public class Foo
{
     private static boolean debugEnabled$1;

     public void bar()
```

```
	{
		if (debugEnabled$1)
		{
			// Protokollierender Methodencode
		}
		else
		{
			// Original-Methodencode
		}
	}
}
```

Erste Prototypen, die dieses Verfahren verwendet haben, haben leider nicht die gewünschten Eigenschaften gezeigt. Insbesondere war die Ablaufgeschwindigkeit stark vermindert. Auch die Gesamtleistung der Laufzeitumgebung war beeinträchtigt. Es gibt mehrere mögliche Gründe für dieses Verhalten. Zum einen werden die erzeugten Klassendateien um ein Vielfaches größer. Entsprechend nimmt die notwendige Arbeit bei Klassenverifikation und JIT-Übersetzung zu. Zum anderen arbeitet der JIT-Übersetzer der Java 1.4/1.5 Laufzeitumgebung offensichtlich methodenorientiert und kann große Methoden nicht so gut oder nur mit viel Zeitaufwand optimieren.

Um dieses Problem zu umgehen, kann man sich eine Eigenschaft moderner Java-Laufzeitumgebungen zunutze machen. In allen getesteten VMs werden Klassen erst geladen, sobald Instanzen der Klasse erzeugt oder Methoden der Klasse aufgerufen werden. Wenn man die instrumentierten Codeabschnitte in eine andere Klasse auslagert, dann werden die ohne aktivierte Protokollierung in der Fehlervorbereitungsphase geladenen Klassen nicht wesentlich größer und belasten damit die Laufzeitumgebung auch nicht mehr als gänzlich uninstrumentierte Klassen. Erst wenn die Protokollierung für Methoden dieser Klasse eingeschaltet wird, lädt die VM die zweite Klasse, die die instrumentierten Methoden enthält.

Codeausschnitt 3-3 Beispielklasse mit Instrumentierung (Übersetzungsschma 2)

```
public class Foo
{
	private static boolean debugEnabled$1;

	public void bar()
	{
		if (debugEnabled$1)
		{
			Foo$0$debug.bar$(this);
		}
		else
		{
			// Original-Methodencode
```

```
        }
    }
}

public class Foo$0$debug
{
    public static void bar$(Foo foo)
    {
        // Protokollierender Methodencode
    }
}
```

Codeausschnitt 3-3 zeigt, wie die Beispielklasse mit dem modifizierten Instrumentierungsschema übersetzt wird.

Methoden in den synthetisch erzeugten Klassen können augrund von Sichtbarkeitsbeschränkungen im Java-Zwischencode nicht auf private Felder und Methoden der Originalklasse zugreifen. Um dieses Problem zu beheben, werden die gleichen Techniken eingesetzt wie bei der Übersetzung von inneren und anonymen Klassen: Der Übersetzer erzeugt zusätzliche Methoden, die einen Zugriff auf private Felder und Methoden ermöglichen (vergleiche [Strein02]).

3.4.2 Protokollierung

In der instrumentierten Version des Methodencodes müssen während der Übersetzung natürlich auch noch der Steuerfluss (für Zielkriterium **K1-1** – Rückwärtsschrittfunktion), Änderungen von Variablenwerten (für Zielkriterium **K1-2**) und die Werte von Ausdrücken (für Zielkriterium **K1-3**) protokolliert werden. Dazu wird die Codeerzeugungskomponente des in CodeGuide integrierten Übersetzers entsprechend angepasst. Um genau und mit geringem Speicherverbrauch jedem protokollierten Wert später eine Quelltextstelle zuordnen zu können, verteilt die Übersetzungskomponente fortlaufende Identifikationsnummern für jede Protokollierungsaktion innerhalb einer Klasse. Eine solche Identifikationsnummer entspricht einem Knoten im Syntaxbaum, z.B. einem Ausdruck. Diese Identifikationsnummern müssen nicht bis zum Gebrauch durch den Debugger vorgehalten werden, sondern können neu generiert werden, sobald der Debugger eine Rückübersetzung in Quelltextposition und Protokollierungstyp benötigt. Damit das funktioniert, muss sichergestellt werden, dass die Übersetzung streng deterministisch erfolgt, dass also immer die gleichen Identifikationsnummern für die gleichen Quelltextstellen vergeben werden und dass sich der Quelltext bis zur Anfrage durch den Debugger nicht ändert. Es kann trotzdem sinnvoll sein, die Zuordnung von Identifikationsnummer zu Quelltextposition und Protokollierungstyp während der Fehlersuche für schnelleren Zugriff zwischenzuspeichern. Da Identifikationsnummern nur innerhalb einer Quelltextdatei eindeutig sind, muss zusätzlich auch noch der Pfad der Quelltextdatei protokolliert werden. Der Pfad der Quelltextdatei ist für alle Protokollierungsaktionen in einer Methode gleich und wird deshalb nur einmal bei Eintritt in die Methode protokolliert. Bei nachfolgenden Aufrufen

von Protokollierungsfunktionen kann die Protokollierungsbibliothek einfach den vorher registrierten aktuellen Pfad ablegen. Allerdings ändert sich der Pfad der Quelltextdatei auch bei dem Aufruf anderer protokollierender Methoden aus dieser Methode, weshalb die Protokollierungsbibliothek einen Stapel für diese Pfade verwalten muss.

Damit der Entwickler sich zwischen früheren Programmzuständen im Einzelschrittmodus bewegen kann, muss der Steuerfluss protokolliert werden. Dazu muss der Eintritt und das Verlassen jeder Methode protokolliert werden. Ansonsten ist es nicht möglich zuzuordnen, ob ein Protokolldatum zu der aktuellen Methodenausführung oder einer früheren Methodenausführung gehört. Üblicherweise werden dem Entwickler in Java-Debuggern Einzelschrittfunktionen auf Zeilenbasis zu Verfügung gestellt. Wenn man aber Instrumentierung mit Zugriff auf Quelltextinformationen einsetzt, dann hat man in der Wiedergabefunktion die Möglichkeit, eine Einzelschrittausführung auf Anweisungsbasis zu implementieren. Dazu wird vor jeder Java-Anweisung eine Protokollierungsfunktion aufgerufen.

Für die Protokollierung von Ausdruckswerten wird die Stapel-orientierte Architektur der Java VM genutzt. Nachdem der Zwischencode für die Ausdrucksberechnung emittiert worden ist, wird der oberste Stapelwert verdoppelt und eine Protokollierungsfunktion aufgerufen. Für jeden Java-Primitivtyp wird in der Protokollierungsbibliothek eine Protokollierungsfunktion bereitgestellt.

Um dem Entwickler die Anzeige von früheren Variablenwerten zu ermöglichen, müssen alle Zuweisungen an Variablen gesondert protokolliert werden. Methodenparameter sind Variablen, die automatisch bei Eintritt in die Funktion einen Wert bekommen. Ihr Wert muss also auch ohne gesonderte Zuweisung direkt nach Methodeneintritt protokolliert werden. Wie bei Ausdruckswerten wird für jeden Java-Primitivtyp in der Protokollierungsbibliothek eine Protokollierungsfunktion bereitgestellt. `this` wird analog zu der Zwischencodearchitektur wie ein Methodenparameter behandelt und bei Methodeneintritt protokolliert.

3.4.3 Beispiel: Instrumentierung einer statischen Methode

Die folgenden Zwischencodeausschnitte erläutern die Instrumentierungstechnik an dem Beispiel einer einfachen statischen Methode.

Codeausschnitt 3-4 Beispielmethode

```
private static int test(int x, int y, int z)
{
    return (x+y)*z;
}
```

Zwischencodeausschnitt 3-5 zeigt den von einem nicht instrumentierenden Java-Übersetzer erzeugten Zwischencode für diese Methode.

Zwischencodeausschnitt 3-5 Kommentierter Zwischencode (uninstrumentiert)

```
private static int test(int x, int y, int z)
     0:  iload_0        // load x
     1:  iload_1        // load y
     2:  iadd           // (x, y) -> (x+y)
     3:  iload_2        // load z
     4:  imul           // (x+y, z) -> (x+y*z)
     5:  ireturn        // return x+y*z
```

Zwischencodeausschnitt 3-6 zeigt den Zwischencode, den ein Übersetzer mit der eben vorgestellten Implementierungstechnik für diese Methode generiert. Der Übersicht halber wurden Zwischencodebefehle weggelassen, mit denen zusätzliche Debugger-Funktionen implementiert werden.

Zwischencodeausschnitt 3-6 Zwischencode (instrumentiert)

```
public static final int test$(int, int, int);
  0:  ldc "C:\\test\\Test.java"
  2:  bipush 101
  4:  invokestatic DebuggerRT.method_start(String;I)V
  14: iload_0
  15: bipush 103
  17: invokestatic DebuggerRT.variable_value_set(II)V
  26: iload_1
  27: bipush 105
  29: invokestatic DebuggerRT.variable_value_set(II)V
  38: iload_2
  39: bipush 107
  41: invokestatic DebuggerRT.variable_value_set(II)V
  44: bipush 108
  46: invokestatic DebuggerRT.statement_start(I)V
  50: iload_0
  51: dup
  52: bipush 109
  54: invokestatic DebuggerRT.expression_end(II)V
  58: iload_1
  59: dup
  60: bipush 110
  62: invokestatic DebuggerRT.expression_end(II)V
  66: iadd
  67: dup
  68: bipush 111
  70: invokestatic DebuggerRT.expression_end(II)V
  74: iload_2
```

```
75: dup
76: bipush 112
78: invokestatic DebuggerRT.expression_end(II)V
82: imul
83: dup
84: bipush 113
86: invokestatic DebuggerRT.expression_end(II)V
90: bipush 113
92: invokestatic DebuggerRT.method_end(I)V
95: ireturn
```

Kommentar zu Zwischencodeausschnitt 3-6:

Codeindex	**Kommentar**
0-4	Protokollierung des Methodenstarts
14-17	Protokollierung des initialen Wertes von `x`
26-29	Protokollierung des initialen Wertes von `y`
38-42	Protokollierung des initialen Wertes von `z`
44-46	Protokollierung des Anfangs der `return ...` Anweisung
50-54	Laden der Variable `x` und Protokollierung des Ausdrucks `x`
58-62	Laden der Variable `y` und Protokollierung des Ausdrucks `y`
66-70	Addition der obersten zwei Stapelwerte und Protokollierung des Ausdrucks `x+y`
74-78	Laden der Variable `z` und Protokollierung des Ausdrucks `z`
82-86	Multiplikation der obersten zwei Stapelwerte und Protokollierung des Ausdrucks `(x+y)*z`
90-95	Protokollierung des Methodenendes

3.5 Protokollierungsbibliothek

3.5.1 Mehrfädigkeit

Der Entwurf der Protokollierungsbibliothek muss von Grund auf für die Untersuchung von mehrfädigen Applikationen (Zielkriterium **K3-1**) ausgelegt sein, wobei mehrere Ausführungsfäden natürlich auch zur selben Zeit denselben Code ausführen können. Wenn man einfache Synchronisationsmechanismen einsetzt, um die Konsistenz der Datenstrukturen sicherzustellen (*„locking“*), dann kommt es zu Einbrüchen bei der Ausführungsgeschwindigkeit, sobald mehr als ein Ausführungsfaden eine protokollierende Methode ausführt. Abhilfe schafft hier die seit Java-Version 1.2 verfügbare `ThreadLocal` Klasse, welche die Möglichkeit bietet, schnell auf Ausführungsfaden-spezifische Daten zuzugreifen. Mithilfe der `ThreadLocal` Klasse kann für jeden Ausführungsfaden ein getrennter Protokollpuffer angelegt werden, auf den ohne Einbußen bei der

Ausführungsgeschwindigkeit während der Protokollierung durch mehrere Ausführungsfäden zugegriffen werden kann. Ein einfacher Prototyp hat bestätigt, dass dieser Ansatz bei Verwendung mehrerer protokollierender Ausführungsfäden keine zusätzlichen Geschwindigkeitseinbußen mit sich bringt, während die Ausführungsgeschwindigkeit bei einem Prototyp mit Synchronisationsmechanismen schon bei drei Ausführungsfäden stark vermindert ist, da alle Ausführungsfäden an den kritischen Sektionen in Protokollierungsfunktionen warten müssen.

Wenn bei Untersuchung der Anwendung nur ein Ausführungsfaden protokolliert wird, könnte man das natürlich schneller komplett ohne Synchronisation und mit Zugriff auf nur einen Protokollpuffer erledigen. Da aber vorher nicht bekannt ist, ob mehrere Ausführungsfäden protokolliert werden müssen, ist dieser Ansatz unbrauchbar.

3.5.2 Datenstrukturen

Um den automatischen Speicherbereiniger nicht unnötig zu belasten, werden während der Protokollierung außer dem Puffer bei der ersten Protokollierungsaktion eines Ausführungsfadens keine weiteren Objekte erzeugt. Die Größe der Protokollpuffer und damit auch die Region, die mithilfe der Rückwärtsschrittfunktion untersucht werden kann, wird durch den Entwickler vor Starten des Debuggers konfiguriert. Pro protokollierenden Ausführungsfaden gibt es ein `ThreadData` Objekt, das alle Protokollierungsdaten für den Ausführungsfaden enthält. Ein Datenpuffer pro Ausführungsfaden hat zusätzlich den Vorteil, dass auch die Ausführungsfäden untersucht werden können, die schon länger keine Daten mehr zu Protokoll gegeben haben. Ein Nachteil ist, dass die Größe der Pufferdaten direkt von der Anzahl der protokollierenden Ausführungsfäden abhängig ist.

Da eine Protokollierung auf Festplatte wie schon beschrieben zu langsam ist und die Größe des Hauptspeichers limitiert, kann nur eine begrenzte Menge an Protokolldaten pro Ausführungsfaden gespeichert werden. Es müssen also alte Daten automatisch entfernt werden. Eine Datenstruktur, die das ohne weiteren zusätzlichen Aufwand ermöglicht, ist der Ringpuffer. Die Protokolldaten werden in das als Ringpuffer fungierende Feld geschrieben, und bei Erreichen des Feldendes wird einfach wieder bei Index null angefangen. Der Index für das nächste Protokolldatum wird bei jeder Protokollierungsaktion um eins modulo der Feldlänge hochgezählt. Um das Ziel, keine zusätzlichen Objekte während der Protokollierung zu erzeugen, zu erfüllen und generell die Belastung des Speicherbereinigers so gering wie möglich zu halten, werden mehrere Ringpuffer für die Protokolldaten verwendet. Es gibt einen Ringpuffer, in dem der Typ der jeweiligen Protokollierungsaktion und zwei weitere Ringpuffer, in denen Werte zu der jeweiligen Protokollierungsaktion gespeichert werden. Man braucht zwei Ringpuffer um Werte zu speichern, da das Java-Typsystem zwischen Primitivwerten und Objektwerten unterscheidet. Ein Ringpuffer wird für Objekte verwendet, der andere vom Elementtyp `long` für alle Primitivwerte. Bei einer Protokollierungsaktion wird jeweils nur einer dieser beiden Ringpuffer genutzt, der andere Ringpuffer bleibt an diesem Index unbelegt. In einem weiteren Ringpuffer werden Identifikationsnummern gespeichert und in dem letzten Referenzen auf den Pfad der Quelltextdatei. Der Pfad, der zu der aktuell protokollierenden Methode gehörenden Quelltextdatei und ein Stapel, der die Pfade von den Quelltextdateien der Aufrufkette der

protokollierenden Methoden enthält, werden ebenfalls in dem `ThreadData` Objekt gespeichert.

3.5.3 Protokollierungsfunktionen

Wie schon im Unterkapitel 3.4 angesprochen, wird für jeden Primitivtyp eine eigene Funktion zur Protokollierung des Ausdruckswertes bereitgestellt. Das gleiche gilt für die Protokollierung von Variablenzuweisungen. Alle Protokollierungsfunktionen müssen grundsätzlich gleich implementiert werden: Zuerst wird das `ThreadData` Objekt für den aktuellen Ausführungsfaden geholt, dann werden die Ringpuffer mit den aktuellen Protokollierungsdaten gefüllt und das Feld, das das Ringpufferende markiert um eins erhöht.

Codeausschnitt 3-7 Protokollierungsfunktionen

```
public static void method_start(String filepath, int infoID)
public static void method_end(int infoID)

public static void statement_start(int infoID)
public static void statement_end(int infoID)

public static void expression_end(int value, int infoID)
public static void expression_end(char value, int infoID)
public static void expression_end(byte value, int infoID)
...

public static void variable_value_set(int value, int infoID)
public static void variable_value_set(short value, int
infoID)
public static void variable_value_set(char value, int infoID)
...
```

3.5.4 Protokollzugriffsfunktionen

Der Zugriff auf Protokolldaten erfolgt durch direkten Aufruf von Protokollbibliotheksmethoden aus der Entwicklungsumgebung über die JPDA-Schnittstelle. Alle Protokollfunktionen arbeiten mit Positionen, die von dem aktuellen Ende der Ringpuffer abgezogen werden. Um zum Beispiel in einem bestimmten Ausführungsfaden von der aktuellen Position einen Rückwärtsschritt auszuführen, wird der Ausführungsfaden und die aktuelle „Rückwärtsposition“ übergeben und die Bibliothek liefert die vorhergehende Position zurück. Um den Debugger in den Wiedergabemodus zu versetzen, wird von der aktuellen Position („0“) aus eine Rückwärtsschrittfunktion ausgeführt.

Codeausschnitt 3-8 Schrittmethoden in der Protokollierungsbibliothek

```
public static int stepBackOver(Thread thread, int tracepos)
public static int stepBackOverExpression(Thread th, int tpos)
public static int stepBackOut(Thread thread, int tracepos)
public static int stepOver(Thread thread, int tracepos)
public static int stepOverExpression(Thread th, int tracepos)
public static int stepIn(Thread thread, int tracepos)
public static int stepOut(Thread thread, int tracepos)
```

Es müssen zusätzlich Funktionen bereitgestellt werden, mit denen auf den Pfad der Quelltextdatei, die Identifikationsnummer, und den Typ und Wert von Variablen und Ausdrücken an Rückwärtspositionen zugegriffen werden kann.

Codeausschnitt 3-9 Zugriffsmethoden in der Protokollierungsbibliothek

```
public static int getInfoID(Thread thread, int tracepos)
public static String getFilePath(Thread thread, int tracepos)
public static String getValueType(Thread thread, int tpos)
public static Object getReferenceValue(Thread th, int tpos)
public static char getCharValue(Thread thread, int tracepos)
public static byte getByteValue(Thread thread, int tracepos)
```

3.6 Filterkonfiguration

Leider kann aus Performance-Gründen nur ein Teil aller Methoden auf protokollierend geschaltet werden. Die Auswahl, welche Methode instrumentiert ausgeführt und damit protokolliert werden soll, kann nur der Entwickler treffen. In den meisten Fällen weiß allerdings auch der Entwickler nicht vorher, welche Methoden er später genauer untersuchen will. Es müssen also heuristische Verfahren eingesetzt werden, um die Menge der „interessanten" Methoden zu bestimmen. Damit der Entwickler aber auf jeden Fall die Möglichkeit hat, die Rückwärtsschrittfunktionen zu nutzen, sobald die Applikation an einem Haltepunkt zum Stehen gekommen ist, müssen zumindest alle Methoden mit Haltepunkten protokolliert ausgeführt werden. Zusätzlich will der Entwickler sich auch von der aktuellen Position in die aufrufende Methode zurückbewegen können oder eine vorher von dieser Methode aus aufgerufene Funktion untersuchen. Genereller kann man sagen, dass sich interessanter Code, den der Entwickler genauer untersuchen möchte, meistens in der näheren Umgebung der Aufrufkette von Methoden mit Haltepunkten und der aktuellen Methode des momentan betrachteten Ausführungsfadens befindet.

Um heuristisch die Menge der zu instrumentierenden Methoden zu bestimmen, muss zunächst ein gerichteter Graph definiert werden, dessen Eckenmenge alle Methoden, die von dem in die Entwicklungsumgebung integrierten Übersetzer übersetzt werden, enthält.

Wenn eine Methode eine andere Methode aufrufen kann, dann existiert eine Kante in diesem Graph von der potentiell aufrufenden Methode zur potentiell aufgerufenen Methode. Zu beachten ist, dass bei der Grapherzeugung auch Schnittstellenimplementierung („interface implementation") und Überladung („overloading") berücksichtigt werden muss.

Wir definieren zu der Menge *M* und der natürlichen Zahl *K* die Menge *Aufruf*(*M, K*) als die Menge aller Methoden, von denen aus ein gerichteter Pfad der maximalen Länge *K* zu einem Element aus *M* in diesem Graph besteht, vereinigt mit der Menge *M*.

Wir definieren zu der Menge *M* und der natürlichen Zahl *K* die Menge *Aufgerufen*(*M, K*) als die Menge aller Methoden, zu denen ein gerichteter Pfad der maximalen Länge *K* von einem Element aus *M* in diesem Graph besteht, vereinigt mit der Menge *M*.

Um dann die Menge *TM* der zu protokollierenden Methoden zu bestimmen wird wie folgt vorgegangen:

1. Bestimme die Menge *H* alle Methoden mit Haltepunkten.
2. Zu der Menge *H* kommt die Methode, in der sich der aktuell untersuchte Ausführungsfaden befindet.
3. Bestimme die Menge *TM* als *Aufruf*(*H*, 2).
4. Füge zu der Menge *TM* die Menge *Aufgerufen*(*TM*, 2) hinzu.
5. Füge zu der Menge *TM* die Menge *Aufgerufen*(*H*, 3) hinzu.
6. Füge zu der Menge *TM* die Menge *Aufruf*(*H*, 15) hinzu.

Die Schritte 3 und 4 bestimmen interessante Methoden im Nahbereich der Methoden mit Haltepunkten. Die Annahme für Schritt 5 ist, dass Entwickler oft von einer Methode mit Haltepunkt ausgehend vorher von dieser Methode aus aufgerufene Methoden untersuchen möchten. Schritt 6 ist nötig, damit der Entwickler sich möglichst weit in der Aufrufkette einer Funktion mit Haltepunkt zurückbewegen kann.

Die Schritte 1 und 2 werden in der Debuggerkomponente vorgenommen, die Schritte 3 bis 6 werden von der Übersetzerkomponente übernommen, die Zugriff auf den Graph potentieller Aufrufe hat und diesen inkrementell aktualisiert.

Diese Methode der Bestimmung der Filtermenge ist zwar für die Erfüllung der Zielkriterien nicht unbedingt erforderlich, aber für die Praxistauglichkeit des Debuggers von großer Wichtigkeit. Der Entwickler muss sich effektiv keine Gedanken um die Protokollierung machen, und die Anwendung läuft trotz der Protokollierung und instrumentierten Ausführung einiger Methoden in den meisten Fällen noch immer mit mehr als akzeptabler Geschwindigkeit.

3.7 Zusammenfassung

Wir haben Implementierungsverfahren auf ihre Tauglichkeit zur Erfüllung der Zielkriterien untersucht und einen Gesamtentwurf vorgestellt, der es ermöglicht, ein Werkzeug zu

entwickeln, das nicht nur die Zielkriterien erfüllt sondern auch in der Praxis eingesetzt werden kann.

Zur Erfüllung von Zielkriterium **K1** haben wir eine geeignete Instrumentierungsmethode und eine Protokollierungsbibliothek entworfen, dabei wurde besonderer Wert auf die Unterstützung der Protokollierung mehrfädiger Applikationen (Zielkriterium **K3-2**) gelegt. Um die Integration mit einer Entwicklungsumgebung (Zielkriterium **K4**) zu ermöglichen, wurde auf die schon vorhandene Entwicklungsumgebung CodeGuide zurückgegriffen. Durch Anbindung der schon in CodeGuide existierenden Debuggerkomponente ist interaktive Fehlersuche (Zielkriterium **K2**) leicht zu implementieren. Durch Entwicklung und Test mehrerer Instrumentierungsmethoden mithilfe von Prototypen wurde sichergestellt, dass einer Implementierung, die die erforderlichen Leistungsmerkmale aufweist (Zielkriterien **K3-3** und **K3-4**), nichts mehr im Weg steht.

4 Umsetzung

4.1 Überblick

In Kapitel 4 werden zunächst interessante Aspekte der Implementierung diskutiert. Dann stellen wir Probleme vor, die in dem Entwurf noch nicht behandelt wurden, aber trotzdem gelöst werden konnten, ohne dass der Entwurf grundlegend geändert werden musste. Am Schluss werden zusätzliche Leistungen aufgeführt, die nicht zur Erfüllung der Zielkriterien implementiert wurden, sondern um dem Entwickler einen Zusatznutzen in der täglichen Praxis zu bieten.

4.2 Aspekte der Implementierung

4.2.1 Benutzerschnittstelle

Die bereits vorhandene Debuggerintegration in CodeGuide muss nur im Detail abgeändert werden, um die Wiedergabe von vorherigen Programmzuständen und Ausdruckswerten zu unterstützen. Die grafische Darstellung des aktuellen Ausführungspunktes und der aktuellen Werte der lokalen Variablen soll auch bei Verwendung der Rückwärtsschritt-funktion gleich erfolgen. Zusätzlich wird dem Entwickler noch angezeigt, dass er sich gerade im Wiedergabemodus (engl. „replay mode“) befindet.

Für die Darstellung von Ausdruckswerten werden Popup-Fenster verwendet, die den Ausdruckswert anzeigen, während der Ausdruck im Quelltext gleichzeitig farbig markiert wird. Es hat sich als sehr nützlich erwiesen, die Werte von vorher in der Methode berechneten Ausdrücken anzuzeigen, sobald die Maus über diesen Ausdrücken steht.

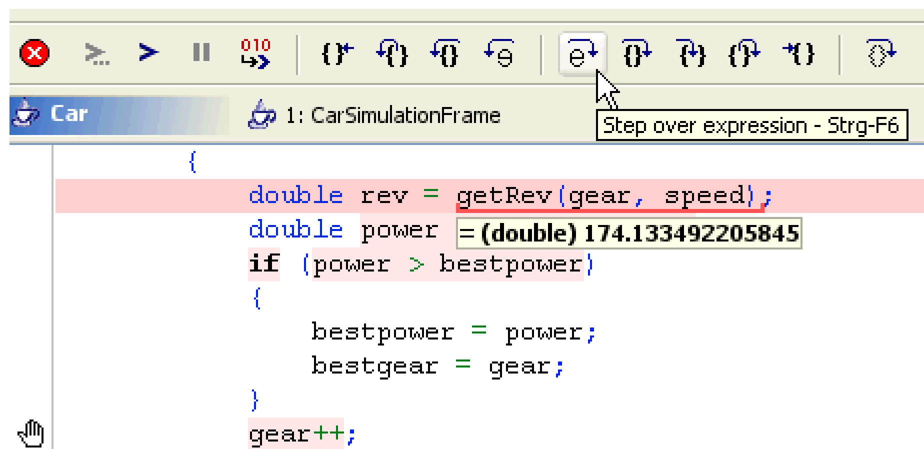

Abbildung 4-1 Ausdruckswertanzeige

4.2.2 Protokolldaten

Die in der gleichen VM ausgeführte Protokollbibliothek sammelt die Protokolldaten auf dem Java-Heap. Eigentlich müssten die Objekte selbst nicht von den Protokolldaten aus referenziert werden. Stattdessen könnten der Klassenname und die Identität des Objektes gespeichert und dem Entwickler im Wiedergabemodus angezeigt werden. Dazu muss die Bibliothek allerdings während der Protokollierung zusätzliche Objekte erzeugen, die diese Informationen beinhalten. Außerdem ist es für den Entwickler nützlich, auch die Feldwerte von Objekten inspizieren zu können, selbst wenn nicht der frühere Wert sondern nur der aktuelle Wert angezeigt wird. Da diese Werte vielleicht nicht mit den Werten an der aktuellen Rückwärtsposition übereinstimmen, werden sie in der graphischen Benutzeroberfläche mit einem Fragezeichen markiert.

Solange Objekte von der Protokollbibliothek aus referenziert werden, kann der Speicherbereiniger der VM diese Objekte nicht freigeben. Dadurch ist die Speichernutzung der Protokollbibliothek theoretisch unbegrenzt, weil so beliebig große Objektgraphen durch eine Referenz in der Protokollbibliothek nicht freigegeben werden können. Der zusätzliche Speicherverbrauch hat sich in der Praxis nicht als Problem erwiesen. Ferner ist sogar wünschenswert, dass die Objekte nicht freigegeben werden, damit der Entwickler sie im Wiedergabemodus inspizieren kann.

Die Protokolldaten der verschiedenen Ausführungsfäden haben keinen zeitlichen Bezug zueinander, so dass Probleme bei der Interaktion von Ausführungsfäden nicht im Wiedergabemodus untersucht werden können. Nur so kann aber eine hohe Ausführungsgeschwindigkeit auch bei Protokollierung von mehreren Ausführungsfäden gleichzeitig erreicht werden.

4.2.3 Verwendete Eigenschaften moderner Java-Laufzeitumgebungen

Die Wahl von Java als Plattform hat sich als sehr vorteilhaft erwiesen. Insbesondere haben die Stapel-basierte Struktur der VM und der einfache Aufbau von Java-Zwischencode eine einfache Umsetzung der für die Protokollierung nötigen Instrumentierungstechniken ermöglicht. Auch wäre die Verwaltung von Objektreferenzen in den Protokollierungspuffern ohne automatische Speicherbereinigung nicht zu verwirklichen gewesen. Wenn Klassen in Java nicht dynamisch geladen würden, hätte das zu erheblichen Problemen bei Ausführungsgeschwindigkeit und Speicherverbrauch geführt.

Da die Zielkriterien nur Unterstützung von den aktuellen Java-Versionen 1.4 und 1.5 verlangen, konnte auch auf neue Funktionen, die in früheren Java-Versionen nicht verfügbar waren, zurückgegriffen werden. Dem neuen JIT-Übersetzer in der Java VM ist es zu verdanken, dass die Instrumentierung die Ablaufgeschwindigkeit der untersuchten Anwendung während der Fehlervorbereitungsphase nicht wesentlich verlangsamt. Die schnelle synchronisationsfreie Protokollierung von mehrfädigen Applikationen ist nur durch Ausführungsfaden-spezifische Speicherung möglich gewesen.

4.2.4 Instrumentierung

Der instrumentierte Zwischencode läuft auf modernen Java-Laufzeitumgebungen mit gleicher Semantik und bei nicht laufender Protokollierung mit nahezu unverminderter Geschwindigkeit auch bei komplexen Applikationen mit mehr als 100.000 Zeilen Quellcode.

Die Größe des erzeugten Zwischencodes muss überprüft werden, da auch bei modernen VMs die Größe des Zwischencodes einer Methode 64KB nicht überschreiten darf. Sollte der instrumentierte Zwischencode zu groß werden, wird auf eine Instrumentierung verzichtet. In der Praxis sind davon aber nur sehr wenige Methoden betroffen.

Die Instrumentierung von Klassen verändert die automatisch berechnete Serialisierungsversion (vergleiche Abschnitt 4.6 von [SERIALIZE]), obwohl das Dateiformat sich nicht ändert. Dieses Problem tritt aber auch auf, wenn ein Java-Quelltext (Version 1.1 oder später) von verschiedenen Übersetzern in Zwischencode übersetzt wird. Der von Sun definierte Algorithmus zur Berechung der Serialisierungsversion konnte, um Kompatibilität zu Java Version 1.0 zu wahren, nicht geändert werden.

Instrumentierter Zwischencode kann bei manchen schlecht programmierten Applikationen, die selbst Zwischencode transformieren, Probleme verursachen. Da das aber nur wenige, selten verwendete Applikationen betrifft, erschienen eine genauere Untersuchung und die Implementierung von Hilfskonstruktionen nicht lohnenswert.

4.3 Gelöste Probleme

4.3.1 Hotswap und Datenprotokollierung

Die Hotswap-Funktion zum Austausch des Zwischencodes einer Methode zur Laufzeit ist ein wichtiges Werkzeug. Entwickler erwarten von einem modernen Debugger, dass diese Funktion unterstützt wird. Bei einer Änderung des Quelltextes stimmen die von dem Übersetzer berechneten Identifikationsnummern nicht mehr mit den bereits protokollierten Identifikationsnummern überein. Das führt dazu, dass bei Verwendung der Rückwärtsschrittfunktion falsche Quelltextstellen angezeigt werden und auch Ausdrucks- und Variablenwerte nicht mehr richtig zugeordnet werden können. In der Umsetzung wurde zur Lösung dieses Problems ein Versionszähler eingeführt, der bei jeder HotSwap-Aktion erhöht wird. Sobald die Protokollierung einer Methode gestartet wird, wird der Wert dieses Zählers zwischengespeichert und bei jeder Protokollierungsaktion in einen zusätzlichen Ringpuffer mitgesichert. Bei Ausführung der Rückwärtsschrittfunktion wird getestet, ob die protokollierten Daten aktuell sind, ansonsten werden keine Quelltextstellen, Ausdrucks- und Variablenwerte angezeigt.

4.3.2 Initialisierung der Protokollierungsfilter

Die statischen Felder, mit denen der instrumentierte Zwischencode bestimmt, ob die Methodenausführung protokolliert werden soll oder nicht, müssen beim Klassenladen auf den richtigen Wert initialisiert werden. Dazu wird vor Start der Applikation und bei jeder Neuberechnung der Filtermenge aufgrund von Änderungen bei den Haltepunkten, die Menge aller statischen Felder, die den Wert `true` erhalten sollen, an die Protokollierungsbibliothek übergeben. Jede instrumentierte Klasse erhält einen statischen Initialisierungsblock, der beim Laden der Klasse diese Felder initialisiert.

Codeausschnitt 4-1 Beispiel: Initialisierung der Instrumentierung

```
public class Foo
{

static
{
      debugEnabled$1 =
DebuggerRT.isLogged("Foo.debugEnabled$1");
      …
}

      …
}
```

Codeausschnitt 4-1 illustriert diese Initialisierung an einem Beispiel.

4.3.3 Protokollierung von Konstruktoren

Nach der Java-Sprachspezifikation (vergleiche [GSJ96] Abschnitt 8.8.5) können Konstruktoren in Klassen als erste Anweisung den Aufruf eines anderen Konstruktors derselben Klasse oder eines Konstruktors der Vaterklasse enthalten. Gibt es keinen solchen Aufruf, so wird ein impliziter Aufruf des parameterlosen Konstruktors der Vaterklasse eingefügt. Diese Konstruktoraufrufe werden durch den `invokespecial` Zwischencodebefehl (siehe [LY99]) übersetzt. Moderne Java Laufzeitumgebungen haben die Einschränkung, dass der Aufruf von Konstruktoren der Vaterklasse mit `invokespecial` nur aus einem Konstruktor einer Tochterklasse erfolgen kann. Deshalb kann das übliche Instrumentierungsschema bei Konstruktoren keine Anwendung finden. Stattdessen wird für den Konstruktor, wie im ursprünglichen Übersetzungsschema vorgesehen, Zwischencode für beide Ausführungsvarianten (protokolliert und nicht protokolliert) in derselben Methode erzeugt.

4.3.4 Protokollierung boolescher Ausdrücke

Boolesche Ausdrücke kommen in den meisten Fällen als Bedingung in einer Schleife oder if-Anweisung vor. Im Java-Zwischencode werden sie normalerweise nicht als Werte auf den Stapel gelegt, stattdessen gibt es bedingte Sprungbefehle, die einen booleschen Ausdruck auswerten und bei erfüllter Bedingung zu einer Zieladresse im Zwischencode springen. Diese Übersetzungsmethode wird in Codeausschnitt 4-2 und Zwichencodeausschnitt 4-3 illustriert.

Codeausschnitt 4-2 Beispielmethode

```
private static int testBoolean(int x)
{
     if (x < 5)
          return 0;
     else
          return 1;
}
```

Zwischencodeausschnitt 4-3 Kommentierter Zwischencode (uninstrumentiert)

```
private static int testBoolean(int x);
     0:  iload_0         // Lade x auf den Stapel
     1:  iconst_5        // Lade 5 auf den Stapel
     2:  if_icmpge 10    // Spring zu Adresse 10 falls x < 5
     5:  iconst_0        // Lade 0 auf den Stapel
     6:  ireturn         // Gebe den obersten Stapelwert
zurück
```

```
     10: iconst_1         // Lade 1 auf den Stapel
     11: ireturn          // Gebe den obersten Stapelwert
zurück
```

Die Instrumentierungsmethode muss also modifiziert werden, so dass in jedem der beiden Grundblöcke, die für die beiden Ausführungszweige stehen, am Anfang der Wert des booleschen Ausdrucks protokolliert wird.

4.3.5 Implementierung konventioneller Debugfunktionen

Der Entwickler soll die gleichen Schrittfunktionen benutzen können unabhängig davon, ob er sich protokollierte Daten anschaut oder ob er den aktuellen Steuerfluss inspiziert. JPDA unterstützt aber nur die Einzelschrittausführung auf Zeilenbasis oder auf Zwischencode-indexbasis. Leider ist die Einzelschrittausführung auf Zwischencodeindexbasis und Überprüfung in der Debuggerkomponente, ob der jeweilige Zwischencodeindex dem Anfang einer Anweisung oder einer Ausdrucksauswertung bestimmt, zu langsam. Deshalb bleibt nur die Möglichkeit in protokollierendem Code synthetische Zeilennummern zu generieren, die den Anfängen von Anweisungen und Ausdrucksauswertungen entsprechen.

Debugger bieten üblicherweise auch die Möglichkeit, die Ausführung jederzeit zu unterbrechen und den Zustand aller laufenden Ausführungsfäden zu inspizieren. Bei einer solchen Unterbrechung kann es passieren, dass sich ein Ausführungsfaden gerade in der Protokollierungbibliothek befindet. Die Protokollierung soll für den Entwickler aber transparent erfolgen, deshalb wird in diesem Fall über JPDA das Verlassen der Protokollierungsmethode erzwungen, bevor dem Entwickler die aktuelle Position angezeigt wird.

Wie schon im vorigen Kapitel erläutert, werden Funktionen zum Protokollzugriff direkt über JPDA aufgerufen. Leider ist ein solcher Aufruf nur möglich, wenn der Ausführungsfaden an einem Haltepunkt oder nach einer Einzelschrittausführung angehalten hat (Vergleiche [JPDASPEC]). Das ist aber bei einer Unterbrechung der Programmausführung durch den Entwickler nicht der Fall. Um auch in diesem Fall die Anzeige von Protokolldaten zu ermöglichen, wird von der Protokollierungsbibliothek ein zusätzlicher Ausführungsfaden verwendet, der alle 50ms aufwacht. In diesem Ausführungsfaden wird eine Einzelschrittausführung, die maximal 50ms dauert, durchgeführt. Danach kann er zum Aufruf von Protokollfunktionen verwendet werden.

4.4 Zusätzliche Leistungen

4.4.1 Erweiterte Schrittfunktionen

Zusätzlich zur Einzelschrittausführung auf Anweisungsbasis wird in protokollierendem Code auch die Einzelschrittausführung auf Ausdrucksbasis unterstützt. Dazu wird immer

nach der Auswertung eines Ausdrucks, der Ausdruckswert angezeigt und bei Ausführung der Einzelschrittfunktion auf Ausdrucksbasis der nächste Ausdruck ausgewertet. Das ist besonders nützlich, wenn der Entwickler entscheiden will, ob ihn der in einem Methodenaufruf ausgeführte Code interessiert. Er kann sich alle Methodenparameter ansehen, bevor er entscheidet, ob er dann den Methodenaufruf überspringt oder in die Methodenausführung hineinspringt.

Bei konventionellen Debuggern gibt es eine Schrittfunktion, die aus der aktuellen Methode herausspringt. Meistens interessiert den Entwickler aber auch, ob die Ausführung der Methode normal endet, welcher Wert zurückgegeben wird und ob eine Ausnahme aufgetreten ist. Damit der Entwickler das sehen kann, wird beim Aufruf dieser Schrittfunktion in CodeGuide die aktuelle Methode zu Ende ausgeführt und der Wert angezeigt, der zurückgegeben wird, oder die aufgetretene Ausnahme.

Um sich effektiv im Wiedergabemodus zu bewegen, wurden Rückwärts- und Vorwärtsschrittfunktionen eingeführt, mit denen man sich direkt zur nächsten bzw. vorigen Ausführung einer bestimmten Stelle im Quelltext bewegen kann.

4.4.2 Zusätzliche Möglichkeiten zur Dateninspektion

Wird ein Ausdruck in einer Schleife mehrfach ausgewertet, so werden zusätzlich zum letzten Wert des Ausdrucks auch die vorigen Werte in der aktuellen Methodenausführung angezeigt.

Der Entwickler hat auch die Möglichkeit, Ausdrücke einzugeben und auszuwerten. Diese Ausdrücke werden dann an der aktuellen Rückwärtsposition ausgewertet. Sollte hierbei auf Felder zugegriffen werden oder sollten Methoden aufgerufen werden, dann wird dem Entwickler mitgeteilt, dass auch das Ergebnis der Auswertung eventuell nicht mit dem Ergebnis einer imaginären Auswertung an der Rückwärtsposition übereinstimmt.

Der Entwickler kann, indem er mit der Maus auf die Parameterdeklaration der aktuell untersuchten Methode zeigt, sich die Parameterwerte anzeigen zu lassen. Um das zu ermöglichen, werden die Parameterwerte am Anfang der Protokollierungsmethode wie Ausdrücke protokolliert.

4.4.3 Steuerflussinspektion

Die protokollierten Steuerflussinformationen werden auch dazu verwendet, dem Entwickler anzuzeigen, welche Teile der aktuellen Methode tatsächlich ausgeführt wurden. Dazu werden die entsprechenden Quelltextstücke im Editorfenster farbig markiert. So kann der Entwickler mit einem Blick sehen, ob beispielsweise ein Zweig einer bedingten Anweisung ausgeführt wurde oder nicht.

```
int gear = 1;
int bestgear = 1;
double bestpower = 0;
while (gear <= gearRatios.length)
{
    double rev = getRev(gear, speed);
    double power = (double) 282.418879995608
    if (power > bestpower)
    {
        bestpower = power;
        bestgear = gear;
    }
    gear++;
```

Abbildung 4-2 Steuerflussanzeige

Zusätzlich zu dieser Funktion werden die Steuerflussinformationen dazu benutzt, dem Entwickler einen Aufrufbaum für den aktuell ausgewählten Ausführungsfaden anzuzeigen. In diesem Aufrufbaum sieht der Entwickler, in welcher Schachtel des aktuellen Ausführungsfadens welche Methode aufgerufen wurde, und welchen Wert sie zurückgegeben hat. Er kann dann die Schachteln früherer Methodenaufrufe weiter untersuchen.

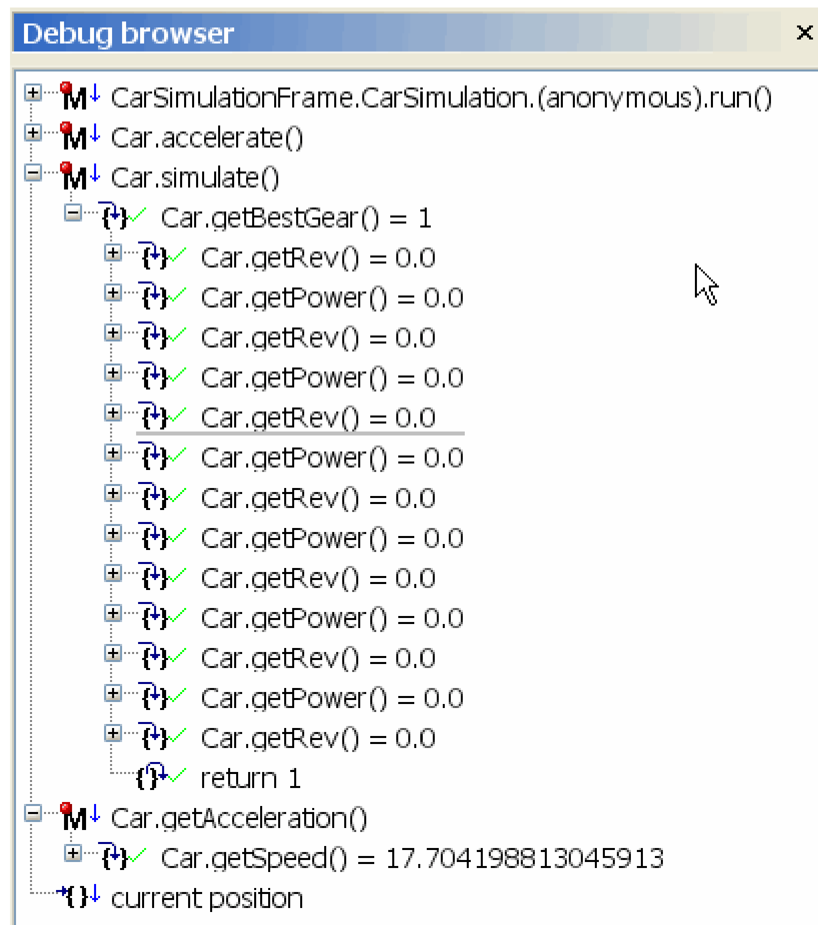

Abbildung 4-3 Anzeige des Aufrufbaums

4.4.4 Ausnahmehaltepunkte

Die Protokollierungsfunktion ist gerade bei der Untersuchung von Ausnahmefehlern besonders nützlich. So kann der Entwickler nämlich genau nachvollziehen, wie es zu der Ausnahme gekommen ist. Leider können Ausnahmen in Java-Programmen in praktisch jeder Methode auftreten. Deshalb kann der Protokollierungsfilter nicht wie bei normalen Haltepunkten automatisch konfiguriert werden. Sobald also ein Haltepunkt für eine bestimmte Ausnahmengruppe gesetzt wird, schaltet CodeGuide die Protokollierung für alle Methoden ein.

4.4.5 Protokollierungslevel

Damit der Entwickler auch geschwindigkeitskritische Methoden untersuchen kann, kann er den Protokollierungslevel eines Haltepunkts einstellen. Drei Möglichkeiten stehen zur Auswahl:

1. Vollprotokollierung. Bei dieser Option findet der in Kapitel 3.6 vorgestellte heuristische Auswahlalgorithmus seine Anwendung. Nicht nur die Methode selbst sondern auch aufrufende und aufgerufenen Methoden werden protokolliert.
2. Methodenprotokollierung. Bei dieser Option wird nur die Methode selbst protokolliert, nicht aufgerufene und aufrufenden Methoden.
3. Keine Protokollierung. Bei Verwendung dieser Option verhält sich der Haltepunkt wie ein Haltepunkt in konventionellen Java-Debuggern. Erweiterte Funktionen, wie die Einzelschrittausführung auf Anweisungsbasis oder Ausdruckswertanzeige werden nicht unterstützt.

Der Entwickler hat die Möglichkeit, die Anwendung mit vollständiger Protokollierung laufen zu lassen. Das verlangsamt die Anwendung zwar sehr stark, dafür können bei einer Unterbrechung des Programms durch den Entwickler alle Ausführungsfäden im Wiedergabemodus untersucht werden.

4.5 Zusammenfassung

In diesem Kapitel wurden zentrale Aspekte der Implementierung vorgestellt und Probleme diskutiert, die während der Implementierung auftraten. Alle diese Probleme konnten durch Detaillösungen beseitigt werden. Um die Praxistauglichkeit des Debuggers zu gewährleisten, wurden zusätzliche Funktionen implementiert, die dem Entwickler besseren Zugriff auf die gesammelten Informationen oder genauere Steuermöglichkeiten bieten.

5 Ergebnisse

5.1 Überblick

In diesem Kapitel werden vergleichende Messungen vorgestellt, die zeigen, dass die Implementierung die Zielkriterien **K3-3** (Ablaufgeschwindigkeit in der Fehler-Vorbereitungsphase höchstens um 10% geringer) und **K3-4** (Zusätzliche Speichernutzung der Applikation in der Fehler-Vorbereitungsphase kleiner als 100KB) erfüllt. Außerdem wird die Implementierung auf künftige Verbesserungsmöglichkeiten untersucht.

5.2 Vergleichende Messungen

Um die Erfüllung der Zielkriterien **K3-3** und **K3-4** bestätigen zu können, müssen an der vorhandenen Implementierung einige Messungen vorgenommen werden. Die Zielkriterien beinhalten zwar nicht den Vergleich mit anderen Implementierungen, es scheint aber sinnvoll, ODB in die Messungen mit einzubeziehen, da ODB frei verfügbar ist und bei ihm als einzigem der vorgestellten Kandidaten eine Chance besteht, dass das Zielkriterium **K3-3** ebenfalls erfüllt wird.

5.2.1 Testumgebung und –methode

Alle Messungen wurden auf einem Dell Inspiron 8500 Notebook mit 1GB Speicher unter Windows XP SP2 vorgenommen. Als Java-Laufzeitumgebung wurde das Sun JDK 1.5.0 eingesetzt. Alle Messungen wurden mithilfe der Java-eigenen `java.lang.System.currentTimeMillis()` Funktion durchgeführt. Der Mittelwert von drei Messungen wurde als Ergebnis verwendet.

Der Speicherverbrauch wurde nach mehrmaligem manuellem Aufruf des Speicherbereinigers vor der Eingabe und unter Verwendung von Methoden der Klasse `java.lang.Runtime` ermittelt.

CodeGuide wurde in Version 7.0 build 822 getestet. Bei ODB wurde die Version „9.Sept.04“ eingesetzt.

5.2.2 Testprogramm

Das Testprogramm berechnet wiederholt Fibonacci-Zahlen in der Aufwärmphase, um Zwischenspeichereffekte zu minimieren, startet danach die gemessene Fehlervorbereitungsphase, erwartet dann eine Eingabe und berechnet darauf noch einmal Fibonacci-Zahlen. Die letzte Berechnung repräsentiert die Fehleruntersuchungsphase und wird daher nur protokolliert und nicht gemessen. Die Berechnungen in der Fehlervorbereitungsphase werden in zwei Sektionen, eine einfädige und eine mehrfädige, unterteilt. In der mehrfädigen Sektion werden fünf Ausführungsfäden erzeugt, danach wird bis zur Beendigung aller Fäden gewartet. Durch Anpassung der Konstante `LOOP_COUNT` lässt sich die Arbeitsmenge in der Fehlervorbereitungsphase feingranular variieren.

In der Fehlervorbereitungsphase werden die gleichen Methoden verwendet, die später protokolliert werden sollen. Deshalb müssen diese Methoden für die Protokollierung vorgesehen sein und können nicht herausgefiltert werden.

5.2.3 Ablaufgeschwindigkeit in der Fehlervorbereitungsphase

Um Zielkriterium **K3-3** zu erfüllen, darf die Ablaufgeschwindigkeit in der Fehlervorbereitungsphase um nicht mehr als 10% gemindert sein.

Test	**Zeit (einfädig)**		**Zeit (mehrfädig)**	
Uninstrumentiert	3165 ms	(1.00)	15549 m	(1.00)
Bei Ausführung in CodeGuide	3245 ms	(1.03)	16146 ms	(1.04)
Bei Ausführung mit ODB	~98 s	(~31)	~9368 s*	(~602)

Die Werte für ODB sind hochgerechnet, die Arbeitsmenge in der Fehlervorbereitungsphase wurde um den Faktor 100 verringert, um überhaupt eine Messung zu ermöglichen. Die Faktoren in Klammern bezeichnen den Zeitbedarf relativ zum uninstrumentierten Ablauf.

Offensichtlich erfüllt die Instrumentierung mit CodeGuide das Zielkriterium **K3-3**. Auch im mehrfädigen Fall ist kein Einbruch zu verzeichnen. Obwohl auch ODB in der Fehlervobereitungsphase keinerlei Protokollierungsaktionen durchführt, ist die Ausführungsgeschwindigkeit im einfädigen Fall um den Faktor 30 geringer. Im mehrfädigen Fall bricht die Leistung dann völlig zusammen.

5.2.4 Ablaufgeschwindigkeit während der Protokollierung

Durch die Protokollierung wird der Code in der Fehleruntersuchungsphase verlangsamt. Eine möglichst geringe Verminderung der Ausführungsgeschwindigkeit ist zur Erfüllung der Zielkriterien zwar nicht notwendig, in der Praxis aber wünschenswert. Eine Verlangsamung um mehr als den Faktor 100 ist zu erwarten, die Ablaufgeschwindigkeit sollte aber bei der Protokollierung mehrerer Ausführungsfäden nicht völlig degenerieren.

Die Implementierungen von ODB und CodeGuide sind hier nicht direkt zu vergleichen, da die gesammelten Informationen unterschiedlich sind. ODB sammelt keine Informationen über Ausdruckswerte, dafür z.B. aber über Feldzuweisungen.

Die Arbeitsmenge in der Fehlervorbereitungsphase wurde gegenüber dem ursprünglichen Test um den Faktor 1000 verringert.

Test	**Zeit (einfädig)**	**Zeit (mehrfädig)**
Bei Ausführung in CodeGuide	2006 ms	10919 ms
Bei Ausführung mit ODB	3498 ms	70832 ms

Die Ausführungsgeschwindigkeit des von CodeGuide instrumentierten Codes skaliert auch bei Protokollierung mit der Anzahl der verwendeten Ausführungsfäden. Auch hier bricht die Leistung von ODB bei der Protokollierung mehrer Ausführungsfäden zusammen.

5.2.5 Speicherverbrauch in der Fehlervorbereitungsphase

Der zusätzliche Speicherverbrauch auf dem Java-Heap in der Fehlervorbereitungsphase sollte nach dem Entwurf minimal sein, wenn keine Daten protokolliert werden. Eine Messung zur Überprüfung bestätigt das.

Test	**Speicherverbrauch**
Uninstrumentiert	614536 bytes
Bei Ausführung in CodeGuide	615296 bytes (+760 bytes)

Zielkriterium **K3-4** wird also erfüllt.

5.3 Verbesserungsmöglichkeiten

5.3.1 Protokollierung

Es wäre sinnvoll, zusätzlich zu der Protokollierung von Variablenzuweisungen und Ausdruckswerten auch die Änderungen von Feldern zu protokollieren. So könnte der Entwickler im Wiedergabemodus sich auch frühere Feldwerte anzeigen lassen. Wenn mehr Informationen protokolliert werden, um dem Entwickler genauere Informationen über den Ablauf des untersuchten Programms anzuzeigen, dann ist aber auch der Ausschnitt des Programmablaufs, der angezeigt werden kann, bei konstantem Speicherverbrauch kleiner.

5.3.2 Protokollierungsbibliothek

Das Prinzip, genau eine Datenstruktur pro Ausführungsfaden zu verwenden, erfüllt den Zweck, Synchronisationskosten bei der gleichzeitigen Protokollierung mehrerer Ausführungsfäden zu vermeiden, sehr gut. Das hat sich in den Messungen bestätigt. Aber es entstehen auch einige Nachteile. So ist der Speicherverbrauch nicht absolut nach oben begrenzt, sondern von der Anzahl der protokollierten Ausführungsfäden abhängig. Erschwerend kommt dazu, dass die Protokoll-Datenstruktur unter bestimmten Umständen nicht freigegeben wird, wenn die Ausführung eines Ausführungsfadens endet. Möglicherweise wäre es sinnvoll, nur eine globale Datenstruktur zu verwenden, wenn die Leistungsvorgaben dabei trotzdem erfüllt werden können.

Ab Java Version 1.5 bietet die Laufzeitbibliothek atomar arbeitende Zähler in dem `java.util.concurrent.atomic` Paket an. Diese Zähler bieten zumindest auf Einprozessorsystemen auch bei mehrfädigen Applikationen eine wesentlich bessere Leistung als eine Implementierung mit kritischen Sektionen. Man könnte damit einen zentralen Ringpuffer implementieren, in dem die Protokolldaten aller Ausführungsfäden gespeichert werden. Um herauszufinden, an welchem Index des Ringpuffers der Protokolleintrag stattfinden soll, ruft die Protokollierungsfunktion den Wert des Zählers ab, der gleichzeitig erhöht wird. So können alle Ausführungsfäden gleichzeitig ohne Synchronisation auf den Ringpuffer zugreifen. Diese Implementierung würde nicht die gleiche Art der Analyse der Interaktion mehrer Ausführungsfäden einer Applikationen bieten wie synchroner Datenzugriff, da keine Ordnung für die Protokollierungsaktionen aller Ausführungsfäden erzwungen wird. Mit dieser Datenstruktur würden aber nur Daten über die letzten Aktionen aller Ausführungsfäden gespeichert werden und die vorher angesprochenen Probleme gelöst.

5.3.3 Unterstützung von HotSwap

Die in dem Debugger angewandte Methode, alle vor einem Methodenaustausch protokollierten Daten als potentiell nicht mehr zuzuordnen zu verwerfen ist etwas grob. Eine genauere Analyse, welche Daten nach einem Methodenaustausch zu verwerfen sind, könnte implementiert werden. Auch könnte der Quelltext vor dem Methodenaustausch gesichert, und für die Zuordnung herangezogen werden.

5.3.4 Instrumentierung

Momentan wird die Einzelschrittausführung auf Anweisungsbasis und die Anzeige von Ausdruckswerten nur in Programmcode unterstützt, der auch direkt vor der Fehlersuche innerhalb der Entwicklungsumgebung übersetzt wurde. Die Übersetzerkomponente wird während der Fehlersuche benötigt, um die Zuordnung von Identifikationsnummer zu Quelltextstelle und Protokollierungstyp durchzuführen. Diese Metainformationen könnten aber auch in einem dokumentierten Format in den Klassendateien selbst abgelegt werden.

So würde die Übersetzerkomponente während der Fehlersuche nicht benötigt. In diesem Fall wäre es möglich, Teile einer größeren Anwendung als Bibliothek zu übersetzen und auch dort die Einzelschrittausführung auf Anweisungsbasis und die Anzeige von Ausdruckswerten zuzulassen.

Mit einem anderen Ansatz wären die erweiterten Debuggerfunktionen auch bei der Fehlersuche in der Java-Laufzeitbibliothek und anderen Fremdbibliotheken möglich. Mithilfe der seit Java Version 1.5 angebotenen `java.lang.instrument` Schnittstelle (siehe [JLI]) könnten auch Systemklassen „just-in-time" instrumentiert werden, direkt bevor sie geladen werden.

Wie schon beschrieben, werden Methoden nicht instrumentiert, wenn die Größe des Zwischencodes der resultierenden protokollierenden Methode 64KB überschreitet. Theoretisch gibt es die Möglichkeit, eine Methode programmatisch in mehrere Methoden aufzuspalten, um auch bei diesen Methoden eine Protokollierung und damit die Anwendung der Rückwärtsschrittfunktion zu ermöglichen.

5.3.5 Filterfunktion

Die in Kapitel 3.6 vorgestellte Filterfunktion hat sich in der Praxis bewährt, es könnte aber trotzdem in Einzelfällen hilfreich sein, wenn der Entwickler die Möglichkeit hätte, die Filtermenge manuell zu beeinflussen, um zum Beispiel bestimmte Methoden von der Protokollierung auszuschließen. Auch könnte der Übersetzer selbst auf die Instrumentierung von ganz einfachen Methoden wie z. B. Methoden, die nur Feldzugriffe kapseln, verzichten.

Die Möglichkeit, auf Ausführungsfadenebene zu filtern, verdient ebenfalls genauere Betrachtung.

5.4 Zusammenfassung

Es wurde gezeigt, dass die Implementierung die Zielkriterien **K3-3** und **K3-4** erfüllt und insbesondere auch bei der Untersuchung von mehrfädigen Applikationen keine Leistungseinbußen zu befürchten sind. Die vorgestellten Verbesserungsmöglichkeiten könnten getestet und bei entsprechenden Resultaten auch in einer zukünftigen Version implementiert werden.

6 Zusammenfassung und Ausblick

6.1 Überblick

In diesem Kapitel werden die Zielkriterien rekapituliert und die Arbeit anhand der Zielkriterien beurteilt. Danach stellen wir Erfahrungen mit dem Werkzeug in der Praxis vor. Die Arbeit schließt mit einem Ausblick auf mögliche künftige Ansätze in diesem Bereich.

6.2 Rekapitulation der Zielkriterien

Wir rekapitulieren die Zielkriterien:

K1 Navigation in und Visualisierung von früheren Programmzuständen

- **K1-1** Rückwärtsschrittfunktionen
- **K1-2** Anzeige des Variableninhalts zu einem früheren Ausführungszeitpunkt
- **K1-3** Anzeige der Werte früherer Ausdrucksberechnungen

K2 Interaktive Fehlersuche

K3 Praxistauglichkeit

- **K3-1** Unterstützung mehrfädiger Applikationen
- **K3-2** Kompatibilität mit den Java Laufzeitumgebungen Version 1.4 und 1.5
- **K3-3** Ablaufgeschwindigkeit in der Fehler-Vorbereitungsphase höchstens um 10% geringer
- **K3-4** Zusätzliche Speichernutzung der Applikation in der Fehler-Vorbereitungsphase kleiner als 100KB

K4 Integration in eine Entwicklungsumgebung

6.3 Bewertung anhand der Zielkriterien

Durch Integration in CodeGuide konnten die Zielkriterien **K2** und **K4** erfüllt werden. Mithilfe des Entwurfs ließ sich ein Debugger mit Rückwärtsschrittfunktionen implementieren, der die Inspektion von früheren Programmzuständen anhand von Ausdrucks- und Variablenwerten zulässt (Zielkriterium **K1**).

Bei der Wahl geeigneter Datenstrukturen und Implementierungstechniken wurde ein Werkzeug entwickelt, dass sich auch in der Praxis zur Untersuchung mehrfädiger Applikationen mit modernen Java-Laufzeitumgebungen benutzen lässt (Zielkriterien **K3-1**, **K3-2**). Dass dabei die strengen Leistungsanforderungen (Zielkriterien **K3-3**, **K3-4**) erfüllt werden, wurde in Kapitel 5 überprüft.

Gemessen an den Zielkriterien ist die Arbeit ein voller Erfolg.

6.4 Nutzen des Werkzeugs in der Praxis

Auch in der Praxis hat sich der inzwischen mit der neuesten Version von CodeGuide ausgelieferte Debugger bewährt. Der Debugger hat vielfältige Verbesserungen erfahren, die weit über die in den Zielkriterien definierten Anforderungen hinausgehen und zum Teil in Kapitel 4 angesprochen wurden. Er findet in vielen Arbeitsumgebungen mit Applikationen jeder Größe und jeden Einsatzgebietes seine Anwendung. Insgesamt können wir annehmen, dass der in CodeGuide integrierte Debugger im Moment das am weitesten entwickelte Java-Fehlersuchwerkzeug ist.

6.5 Ausblick

Neben den schon in Kapitel 5 genannten Verbesserungsmöglichkeiten der Implementierung des Debuggers sind natürlich auch noch weiter gehende Ansätze denkbar. So könnte nachdem sich die Rückwärtsschrittfunktion durchgesetzt hat, auch eine Abkehr von ursprünglichen Debugger-Metaphern erfolgen. Man könnten andere Darstellungsmethoden entwickeln, die die Fülle der protokollierten Funktionen besser aufbereiten und dem Entwickler ganz andere Sichten auf die Applikation ermöglichen.

Die Entwickler von modernen Debuggern mit Rückwärtsschrittfunktion könnten zusammen einen Standard entwickeln, in dem Informationen abgelegt werden oder sich auf eine Schnittstelle einigen, die Protokollierungen innerhalb der VM durchzuführen und mit dem Debugger nur darauf zuzugreifen.

Eine weitere Entwicklungsrichtung ist die Unterstützung anderer Programmiersprachen. Auf Java basierende Programmiersprachen wie JavaServerPages und Groovy (siehe [GROOVY]) könnten mit relativ geringem Aufwand unterstützt werden, allerdings vielleicht nur mit Rückwärtsschrittfunktion auf Zeilenbasis, wenn kein kompletter Übersetzer für diese Sprachen entwickelt werden soll. C# und die .NET Plattform (siehe

[DOTNET]) bieten sich als größte Konkurrenz zu Java ebenfalls an. Die Entwicklung eines Debuggers mit Rückwärtsschrittfunktion für .NET würde aber sicher einen großen Entwicklungsaufwand benötigen, insbesondere da viele der hier verwendeten Verfahren sich nicht direkt auf .NET Laufzeitumgebungen übertragen lassen.

Literaturverzeichnis

[BDBJ] Jonathan Cook's BDBJ 1.2.1, http://homepages.inf.ed.ac.uk/s0090668/bdbj/bdbj-1.2.1/dl.html

[Berg04] Klaus Berg. RetroVue 1.1. Java Developers Journal, 2004. http://www.sys-con.com/story/?storyid=44379&DE=1

[Cook00] Jonathan Cook. *Reverse Debugging of Java Programs*. Diploma Dissertation, Hughes Hall College, 2000.

[CG] Omnicore CodeGuide, http://www.omnicore.com

[DOTNET] Microsoft .NET, http://www.microsoft.com/net/

[GROOVY] The Groovy programming lanugage, http://groovy.codehaus.org/

[GSJ96] James Gosling, Bill Joy und Guy Steele. *The Java TM Language Specification*. Addison-Wesley, 2. Auflage, 1996.

[LY99] Tim Lindholm und Frank Yellin. *The Java TM Virtual Machine Specification*. Addison-Wesley, 2. Auflage, 1999.

[JAVARFE1] Java enhancement request "Add support for expression level stepping – character range attribute and APIs", http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5024112

[JLI] Sun JDK 1.5 `java.lang.instrument` API, http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html

[JPDA] Sun's Java Platform Debugger Architecture (JPDA), http://java.sun.com/products/jpda/

[JPDASPEC] JPDA Specification, http://java.sun.com/j2se/1.5.0/docs/guide/jpda/

[JSP] JavaServer Pages, http://java.sun.com/products/jsp/

[JSQL] Oracle JSQL, http://www.orafaq.com/faqjdbc.htm

[KAFFE] Kaffe Virtual Machine, http://www.kaffe.org/

[METAMATA] Metamata Debug, http://web.archive.org/web/20001202223600/charlie.metamata.com/metamata/matrix.html

[ODB] Bil Lewis' ODB, http://www.lambdacs.com/debugger/debugger.html

[RETROVUE] VisiComp RetroVue, http://www.visicomp.com/

[SCSL] Sun Community Source License, http://wwws.sun.com/software/communitysource/

[SERIALIZE] Sun JDK 1.5.0 Serialization Specification, http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html

[Strein02] Dennis Strein. *Entwicklung eines inkrementellen nebenläufigen Übersetzers für generisches Java*. Diplomarbeit, Universität Karlsruhe, 2002.