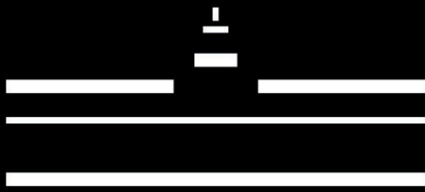


Entwicklung und Implementierung eines hybriden Debuggers für Java

Christian Hermanns

Entwicklung und Implementierung eines hybriden Debuggers für Java

Inauguraldissertation
zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften durch die
Wirtschaftswissenschaftliche Fakultät der
Westfälischen Wilhelms-Universität Münster

vorgelegt von
Dipl.-Wirt.-Inf. Christian Hermanns
aus Bielefeld

Münster, Dezember 2010

Dekan:	Prof. Dr. Thomas Apolte
Erstberichterstatter:	Prof. Dr. Herbert Kuchen
Zweitberichterstatter:	Prof. Dr. Müller-Olm
Tag der mündlichen Prüfung:	18. November 2010

Christian Hermanns

Entwicklung und Implementierung eines hybriden Debuggers für Java

MV WISSENSCHAFT

Westfälische
Wilhelms-Universität
Münster

Wissenschaftliche Schriften der WWU Münster

Reihe IV

Band 3

ulb
Universitäts- und
Landesbibliothek Münster

Christian Hermanns

Entwicklung und Implementierung eines hybriden Debuggers für Java

MV WISSENSCHAFT

Wissenschaftliche Schriften der WWU Münster
herausgegeben von der Universitäts- und Landesbibliothek Münster
http://www.ulb.uni-muenster.de

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Dieses Buch steht gleichzeitig in einer elektronischen Version über den Publikations- und Archivierungsserver der WWU Münster zur Verfügung.
http://www.ulb.uni-muenster.de/wissenschaftliche-schriften

Christian Hermanns
„Entwicklung und Implementierung eines hybriden Debuggers für Java“
Wissenschaftliche Schriften der WWU Münster, Reihe IV, Band 3

© 2010 der vorliegenden Ausgabe:
Die Reihe „Wissenschaftliche Schriften der WWU Münster“ erscheint im Verlagshaus Monsenstein und Vannerdat OHG Münster
www.mv-wissenschaft.com

ISBN 978-3-8405-0030-5 (Druckausgabe)
URN urn:nbn:de:hbz:6-55489474854 (elektronische Version)

© 2010 Christian Hermanns
Alle Rechte vorbehalten

Satz: Christian Hermanns
Umschlag: MV-Verlag
Druck und Bindung: MV-Verlag

Abstract

Das Debugging ist ein komplexer und arbeitsintensiver Prozess in der Softwareentwicklung. Für das Debugging von Java-Programmen werden bis heute vor allem sogenannte Trace-Debugger verwendet. Diese unterstützen die Fehlersuche, indem sie es ermöglichen, ein untersuchtes Programm schrittweise auszuführen. Im Bereich der Forschung sind viele neue Methoden und Werkzeuge entwickelt worden, die im Vergleich zum Trace-Debugging eine erhebliche Verbesserung und Vereinfachung des Debugging-Prozesses versprechen. Auf die in der Praxis eingesetzten Verfahren hatten diese Entwicklungen bisher nur einen äußert geringen Einfluss.

In der vorliegenden Arbeit wird die Entwicklung und Implementierung einer neuen hybriden Debugging-Methode für Java-Programme beschrieben. Die Methode kombiniert deklaratives Debugging und Omniscient-Debugging.

Deklaratives Debugging ist eine Methode, deren Ursprünge im Bereich der logischen Programmierung liegen. Im Gegensatz zum Trace-Debugging muss bei dieser Methode der Programmablauf nicht mehr schrittweise nachvollzogen werden. Dies erlaubt es, den Debugging-Prozess von den Details der Implementierung zu abstrahieren. Ein deklarativer Debugger erzeugt einen Berechnungsbaum, der die Struktur des untersuchten Programmablaufs repräsentiert. Die Knoten dieses Baumes repräsentieren Teile des Quelltextes, die während des Programmablaufs ausgeführt wurden. In einem halbautomatischen Debugging-Prozess fordert der Debugger den Benutzer auf, bestimmte Knoten des Baumes zu klassifizieren. Hierbei muss der Benutzer beurteilen, ob die durch einen Knoten repräsentierte Teilberechnung valide ist. Der Debugging-Prozess endet, wenn der Debugger einen defekten Knoten identifiziert hat. Der Teil des Quelltextes, dessen Ausführung durch den defekten Knoten repräsentiert wird, enthält den gesuchten Defekt.

Omniscient-Debugging erweitert den Ansatz des Trace-Debugging und ermöglicht es, die Ausführung eines Programms auch rückwärts, d. h. entge-

gen der Ausführungsrichtung, zu untersuchen. Die Stärken des Omniscient-Debugging resultieren aus der Möglichkeit, die beobachtete Fehlerwirkung zum verursachenden Defekt einfach und effizient zurückverfolgen zu können.

Beide Methoden werden erweitert, an die Bedürfnisse der Programmiersprache Java angepasst und zu einer hybriden Debugging-Methode kombiniert. Die Realisierbarkeit und Praktikabilität dieser Methode wird durch den Entwurf und die Implementierung des Java-Hybrid-Debuggers (JHyde) demonstriert.

Inhaltsverzeichnis

Abbildungsverzeichnis vii

Tabellenverzeichnis ix

Listingsverzeichnis xi

1 Einführung 1
1.1 Zielsetzung 3
1.2 Aufbau der Arbeit 5

2 Java 7
2.1 Die Java-Technik 8
2.2 Die Programmiersprache Java 9
2.2.1 Historie und Entwurfsziele 9
2.2.2 Grundlegende Sprachmerkmale 11
2.3 Die Java Virtual Machine 13
2.3.1 Architektur 14
2.3.2 Speicherarchitektur 16
2.3.2.1 Methodenbereich 16
2.3.2.2 Heap 17
2.3.2.3 Thread-Speicher 17
2.3.3 Ausführungseinheit 22
2.3.4 Datentypen 24
2.3.5 Bytecode-Instruktionen 25
2.3.5.1 Aufbau 25
2.3.5.2 Datenflussoperationen 26
2.3.5.3 Kontrollflussoperationen 29
2.3.5.4 Rechenoperationen 31
2.3.6 Format der Java-Klassendatei 32
2.4 Fazit 34

3 Grundlagen der Fehlersuche 37
3.1 Ursache und Wirkung von Programmfehlern 37
3.2 Infektionstypen 40
3.3 Debugging von Programmen 45
3.4 Fazit 48

4 Debugging-Techniken 49
4.1 Trace-Debugging 50
4.2 Omniscient-Debugging 54
4.3 Deklaratives Debugging 57
4.3.1 Verfahren 57
4.3.2 Adaption für die Programmiersprache Java 66
4.3.2.1 Berechnungsbaum für Java-Programme . . . 66
4.3.2.2 Seiteneffekte in Java-Programmen 66
4.3.2.3 Erforderliche Informationen zur Klassifizierung eines Methodenaufrufs 67
4.3.2.4 Der Zustandsraum eines Methodenaufrufs . 72
4.3.2.5 Aufwandsvergleich zu deklarativen Sprachen 74
4.4 Hybrides Debugging 76
4.5 Fazit 80

5 Deklarative Debugging-Strategien 83
5.1 Top-Down 85
5.2 Divide-and-Query 86
5.3 D&Q mit gewichtsunabhängigen Infektionswahrscheinlichkeiten 91
5.3.1 Verfahren 91
5.3.2 Kontrollflussgraph 95
5.3.3 Kontrollflussbasierte Infektionswahrscheinlichkeit . . . 95
5.3.4 Datenflussorientierte Infektionswahrscheinlichkeit . . . 100
5.3.5 Schätzung der Infektionswahrscheinlichkeit 103
5.4 Empirische Untersuchung 107
5.5 Fazit 111

6 Der Java-Hybrid-Debugger 113
6.1 Benutzeroberfläche 113
6.1.1 Berechnungsbaumansicht 115
6.1.2 Knotenansicht 117

6.1.3 Ereignisansicht . . . 121
6.1.4 Variablenansicht . . . 126
6.2 Debugging eines defekten Java-Programms . . . 128
6.2.1 Defekter Mergesort-Algorithmus . . . 128
6.2.2 Suche des Defekts mit JHyde . . . 131
6.3 Fazit . . . 144

7 Entwurf und Implementierung 145
7.1 Architektur . . . 146
7.2 Transmitter . . . 148
7.2.1 Die Ereignisschnittstelle . . . 149
7.2.1.1 Klassenstruktur . . . 150
7.2.1.2 Erzeugen von Objekten . . . 152
7.2.1.3 Lesen von Variablen . . . 153
7.2.1.4 Schreiben von Variablen . . . 154
7.2.1.5 Methodenaufruf . . . 154
7.2.1.6 Sonstiger Kontrollfluss . . . 155
7.2.2 Architektur . . . 156
7.2.3 Sendeprozess . . . 159
7.2.4 Empfangsprozess . . . 164
7.2.5 Konfiguration . . . 168
7.3 Instrumentierer . . . 169
7.3.1 Instrumentierung von Java-Programmen . . . 169
7.3.1.1 Java-Instrumentation-API . . . 170
7.3.1.2 Frameworks . . . 171
7.3.2 Klassen in der Prüflings-VM . . . 176
7.3.3 Instrumentierungsschema . . . 178
7.3.4 Architektur . . . 193
7.3.5 Instrumentierungsprozess . . . 194
7.3.5.1 Komposition der `Visitor`-Instanzen . . . 195
7.3.5.2 Dynamische Instrumentierung . . . 196
7.3.5.3 Statische Instrumentierung . . . 199
7.3.5.4 Instrumentierung der Registrierungsmechanismen . . . 199
7.3.6 Konfiguration . . . 200
7.4 Rekorder . . . 201
7.4.1 Modell des Programmablaufs . . . 201

7.4.2 Debugging-Strategien . . . 205
7.5 Benutzeroberfläche . . . 209
7.6 Fazit . . . 213

8 Stand der Forschung 217
8.1 Abfragebasiertes Debugging . . . 218
8.2 Record-Replay-Techniken . . . 221
8.3 Omniscient-Debugging . . . 227
8.4 Deklaratives Debugging . . . 229
8.5 Debugging-Strategien . . . 232
8.6 Fazit . . . 235

9 Schlussbetrachtungen 237
9.1 Zusammenfassung . . . 237
9.2 Ausblick . . . 240

Literaturverzeichnis 243

Abbildungsverzeichnis

1.1 Aufbau der Arbeit 5

2.1 Die Java-Plattform 9
2.2 Architektur der Java-VM 15
2.3 Speichermodell der Java-VM 17
2.4 Operanden-Stack bei Ausführung der Methode `calc` 20
2.5 Explizite Typkonvertierungen in der Java-VM 32
2.6 Format der Java-Klassendatei 35

3.1 Vom Defekt zur Fehlerwirkung 38

4.1 Der Trace-Debugging-Prozess 51
4.2 Der Omniscient-Debugging-Prozess 55
4.3 Repräsentation eines Programmablaufs durch einen Berechnungsbaum 58
4.4 Suche des defekten Knotens 60
4.5 Der deklarative Debugging-Prozess 62
4.6 Der hybride Debugging-Prozess. 79

5.1 Suche des defekten Knotens mit Top-Down-Strategie 85
5.2 Suche des defekten Knotens mit der Divide-and-Query-Strategie 88
5.3 Worst Case der Divide-and-Query-Strategie 90
5.4 Suche des defekten Knotens mit der Divide-and-Query-Strategie, basierend auf Überdeckungsinformationen 93
5.5 Methode `sumUp` mit Kontrollflussgraph 95
5.6 If-Anweisung mit Kontrollflussgraph 100
5.7 Methode `sumUp` mit möglichen Definition-Use-Ketten 102

6.1 Benutzeroberfläche von JHyde 114
6.2 Berechnungsbaumansicht der JHyde Benutzeroberfläche . . . 116

6.3 Knotenansicht der JHyde Benutzeroberfläche 119
6.4 Ereignisansicht der JHyde Benutzeroberfläche 121
6.5 Variablenansicht der JHyde Benutzeroberfläche 126
6.6 Start der Debugging-Sitzung mit JHyde für den Programmablauf der defekten Mergesort-Implementierung 132
6.7 Knoten- und Ereignisansicht für den Methodenaufruf `sort` . 134
6.8 Knoten- und Ereignisansicht für den ersten `distribute`-Aufruf 135
6.9 Knoten- und Ereignisansicht für den ersten `merge`-Aufruf . . 136
6.10 Knoten- und Ereignisansicht für den zweiten `merge`-Aufruf . 137
6.11 Knoten- und Ereignisansicht für den zweiten `distribute`-Aufruf 138
6.12 Knoten- und Ereignisansicht für den zweiten `merge`-Aufruf . 139
6.13 Variablenansicht für das infizierte Array-Element `a[2]` . . . 141
6.14 Variablenansicht für das infizierte Array-Element `a[2]` des modifizierten Mergesort-Programms 142

7.1 Architektur von JHyde 147
7.2 Klassendiagramm des Transmitters 157
7.3 Sequenzdiagramm der Initialisierung des `SocketWriter` . 160
7.4 Sequenzdiagramm des `SocketWriter`-Threads 161
7.5 Sequenzdiagramm für die Übermittlung der Ereignissen vom Prüfling an den `SocketWriter` 163
7.6 Sequenzdiagramm der `SocketReader`-Initialisierung . . . 165
7.7 Sequenzdiagramm des Ereignisempfangs durch den `SocketReader` 167
7.8 Klassen in der Java-VM des untersuchten Programms 176
7.9 Instrumentierungsschema für Java-Klassen 180
7.10 Instrumentierung einer `if`-Anweisung mit Kurzschlussauswertung 187
7.11 Klassendiagramm der wichtigsten Instrumentierungsklassen . 193
7.12 Kommunikationsdiagramm des Instrumentierers 195
7.13 Sequenzdiagramm des Instrumentierungsprozesses 197
7.14 Instrumentierung der Registrierungsmechanismen 200
7.15 Datenmodell des Programmablaufs 202
7.16 Klassendiagramm der Debugging-Strategien 206
7.17 Klassendiagramm der Benutzeroberfläche 213

Tabellenverzeichnis

2.1 Anweisungen der Programmiersprache Java 13
2.2 Datentypen der Java-Plattform 25
2.3 Klassifikation der Bytecode-Instruktionen 27

3.1 Typen von infizierten Werten 40
3.2 Programmablauf der defekten Methode `sumUp` aus Listing 3.1. 41
3.3 Programmablauf der defekten Methode `sumUp` aus Listing 3.2. 43

4.1 Durch den Aufruf der Methode `bar` verursachte Zustandsänderungen 69
4.2 Zustandsraum für den Aufruf der Methode `bar` 75
4.3 Ranking der Debugging-Methoden 77

5.1 Kontrollflussorientierte Überdeckungstests 96
5.2 Abs. Effizienzvergleich deklarativer Debugging-Strategien . . 109
5.3 Rel. Effizienzvergleich deklarativer Debugging-Strategien . . . 110

Listingsverzeichnis

2.1 Schematischer Programmablauf der Ausführungseinheit . . . 22

3.1 sumUp Methode mit einem Defekt in Zeile 3 42
3.2 sumUp Methode mit einem Defekt in Zeile 4 43

4.1 Java-Klasse Foo mit Nebeneffekten 68

6.1 Defekte Implementierung des Mergesort-Algorithmus 130

7.1 Die Konfigurationsdatei TransmissionConfig.properties 168
7.2 Die premain-Methode des Instrumentierers 171
7.3 Das Interface ClassFileTransformer 172
7.4 Instrumentierte Methode bar 183
7.5 Registrierungsmechanismus für die Klasse Foo 188
7.6 Registrierungsmechanismus für die Instanzattribute der Klasse Foo 190
7.7 Erzeugte Methoden zur Übermittlung einer unbekannten Instanz 191
7.8 Die Konfigurationsdatei InstrumentationConfig.properties 201

Kapitel 1

Einführung

Inhalt

1.1 **Zielsetzung** 3
1.2 **Aufbau der Arbeit** 5

Das englische Wort „bug“ zur umgangssprachlichen Beschreibung des Defekts oder des Fehlverhaltens einer Maschine geht laut Shapiro [131] auf das 19. Jahrhundert zurück und stammt ursprünglich aus dem Bereich des Ingenieurwesens. Einer der frühesten schriftlichen Belege für diese Bedeutung des Begriffs „bug“ findet sich in einen Brief des Erfinders Thomas Edison an seinen Kollegen Puskas vom 13. November 1878. Hughes zitiert in [69, S. 75] folgenden Ausschnitt der Korrespondenz:

> It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise – this thing gives out and [it is] then that 'Bugs' – as such little faults and difficulties are called – show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.

Ein früher Beleg für die Verwendung des Begriffs im Bereich der Informationstechnik ist der berühmte Logbucheintrag der Computerpionierin Grace Hopper vom 9. September 1947. Nachdem einer der Techniker eine in einem Relais eingeklemmte Motte als Ursache des Fehlverhaltens des Mark II Computers identifiziert hatte, fixierte sie diese mit einem Klebeband in ihrem Logbuch und notierte dazu folgenden Eintrag:

> 15:45 Relay #70 Panel F (moth) in relay. First actual case of bug being found.

Das Logbuch mitsamt der Motte ist im Smithsonian National Museum of American History ausgestellt.

Neben dieser vergleichsweise amüsanten Anekdote aus den Anfängen des Computerzeitalters gibt es viele weitere Beispiele, in denen Soft- oder Hardwarefehler weitaus schlimmere Konsequenzen hatten. Am 4. Juni 1996 endete der Jungfernflug der unbemannten Ariane 5 Rakete 40 Sekunden nach dem Start in einer Katastrophe. Durch einen Softwarefehler kam die Rakete 37 Sekunden nach dem Start vom Kurs ab und wurde 3 Sekunden später durch den automatischen Selbstzerstörungsmechanismus zerstört. Dabei entstand ein geschätzter Schaden von 370 Million US$ [42]. Die Ursache der Katastrophe war die Typkonvertierung einer 64-Bit-Fließkommazahl in eine 16-Bit-Ganzzahl, bei der es zu einem arithmetischen Überlauf kam. Dieser Fehler führte zum Absturz des Trägheitsnavigationssystems, welches daraufhin nur noch Diagnoseinformationen an den Bordcomputer übermittelte. Der Bordcomputer interpretierte die Diagnoseinformationen fälschlicherweise als Flugdaten und ermittelte eine starke Abweichung von der vorhergesehenen Flugbahn. Die folgende starke Kurskorrektur führte zum Auseinanderbrechen der Rakete und schließlich zur automatischen Selbstzerstörung.

Ein weiterer Beleg für die durch Softwarefehler verursachten Schäden ist ein Bericht des National Institute of Standards & Technology aus dem Jahre 2002. Dieser Bericht schätzt die durch Softwarefehler verursachten jährlichen Kosten für die USA auf 59,5 Milliarden US$, was zum damaligen Zeitpunkt einem Anteil von 0,6 Prozent des Bruttosozialprodukts entsprach [153].

Diese Beispiele belegen, dass Softwarefehler ein ernstzunehmendes Problem in der Softwareentwicklung darstellen und große ökonomische Anreize bestehen, Softwarefehler zu entdecken und zu entfernen. Die Beseitigung von Softwarefehlern geschieht durch den Debugging-Prozess. Dieser wird angestoßen, sobald in der Testphase das Fehlverhalten eines Programms beobachtet wurde. Ziel des Debugging-Prozesses ist es, die Ursache des Fehlverhaltens zu finden. Hierfür muss von dem fehlerhaften Programmablauf auf den Fehler im Quelltext des Programms geschlossen werden.

Für größere Programme, die äußerst komplexe Programmabläufe produzieren können, ist diese Suche alles andere als trivial. In der heutigen Softwareentwicklung ist der Debugging-Prozess daher äußerst zeit- und arbeitsintensiv [49, 62].

Eines der am häufigsten verwendeten Werkzeuge zum Auffinden von Defekten in Java-Programmen ist der Trace-Debugger. Ein Trace-Debugger dient dazu, ein Java-Programm schrittweise auszuführen, um dabei die verursachten Änderungen des Programmzustands zu beobachten.

Im Bereich der Forschung gibt es viele Ansätze und Verfahren, die gegenüber dem Trace-Debugging eine Erleichterung und Beschleunigung der Defektsuche ermöglichen. Dennoch werden in der praktischen Softwareentwicklung bis heute nahezu ausnahmslos Trace-Debugger verwendet. Diese Lücke zwischen wissenschaftlicher Theorie und alltäglicher Praxis bildet den Ansatzpunkt für diese Arbeit.

1.1 Zielsetzung

Diese Dissertation basiert in Teilen auf zwei veröffentlichten Arbeiten. Die erste Arbeit ist in Zusammenarbeit mit Caballero und Kuchen entstanden und beschreibt die Entwicklung eines prototypischen deklarativen Debuggers für Java [25].

Die zweite Arbeit ist in Zusammenarbeit mit Kuchen entstanden und erweitert den deklarativen Debugger um einen Ansatz zur Reduzierung der Anzahl der zu klassifizierenden Methodenaufrufe [64]. Zu diesem Zweck zeichnet der Debugger während der Ausführung des untersuchten Programms Überdeckungsinformationen auf. Basierend auf den Überdeckungsinformationen werden Äquivalenzklassen von Methodenaufrufen gebildet. Wenn ein Methodenaufruf c_1, der zu einer Äquivalenzklasse E gehört klassifiziert wird, dann verwendet der Debugger diese Klassifikation für nachfolgende Methodenaufrufe. Wenn im weiteren Verlauf des Debugging-Prozesses ein Methodenaufruf c_2 derselben Äquivalenzklasse E klassifiziert werden soll, dann benutzt der Debugger die Klassifikation von c_1, um den Methodenaufruf c_2 automatisch zu klassifizieren. Auf diese Weise muss der

Benutzer weniger Knoten klassifizieren, wodurch die Defektsuche beschleunigt wird.

Diese Dissertation greift die Ansätze und Ergebnisse dieser beiden Arbeiten auf und entwickelt sie weiter. Den Ansatzpunkt für diese Arbeit bilden zwei Unzulänglichkeiten der zuvor entwickelten Debugging-Methode:

- Das deklarative Debugging identifiziert ausschließlich die defekte Methode eines Java-Programms. Im Vergleich zum Trace-Debugging, mit dem nicht nur die defekte Methode, sondern auch die defekte Anweisung ermittelt werden kann, stellt dies einen Genauigkeitsverlust dar.

- Der Ansatz, äquivalente Methodenaufrufe automatisiert zu klassifizieren, kann zu fehlerhaften Klassifikationen führen, wodurch die Defektsuche zu einem falschen Ergebnis kommt. Dieses Verhalten ist unerwünscht.

Das Ziel dieser Arbeit ist die Entwicklung eines hybriden Debuggers für die Programmiersprache Java. Durch die Erweiterung des deklarativen Debuggers um eine Komponente für das Omniscient-Debugging kann die mangelnde Präzision des deklarativen Debugging-Prozesses behoben werden. Mithilfe des Omniscient-Debugging ist es möglich, neben dem defekten Methodenaufruf auch die defekte Anweisung zu identifizieren.

Darüber hinaus soll eine neue Strategie zur Beschleunigung des deklarativen Debugging entwickelt werden. Diese Strategie soll ebenfalls auf der Aufzeichnung von Überdeckungsinformationen basieren. Im Unterschied zu dem früheren Ansatz sollen Methodenaufrufe allerdings nicht mehr automatisch klassifiziert werden. Bei dieser neuen Strategie soll ausschließlich die Reihenfolge der Klassifikationen angepasst werden. Auf diese Weise wird ebenfalls eine Beschleunigung des Debugging-Prozesses erreicht, allerdings werden fehlerhafte automatische Klassifikationen ausgeschlossen.

Die hybride Debugging-Methode wurde durch den Java-Hybrid-Debugger (JHyde) implementiert. Im Rahmen dieser Arbeit soll gezeigt werden, dass JHyde und damit auch die hybride Debugging-Methode für das Debugging von Java-Programmen geeignet sind. Ferner sollen die wesentlichen Aspekte des Entwurfs und der Implementierung von JHyde vorgestellt werden.

1.2 Aufbau der Arbeit

Diese Arbeit gliedert sich wie folgt (vgl. Abbildung 1.1). Im Anschluss an diese Einführung werden zunächst einige Grundlagen für die Entwicklung eines hybriden Debuggers erläutert. Hierzu werden im Kapitel 2 die für diese Arbeit relevanten Strukturen der Programmiersprache Java und der Java Virtual Machine beschrieben. Die Funktionsweise der Java Virtual Machine ist vor allem für die Implementierung von JHyde relevant. Das Kapitel 3 erläutert die Grundlagen der Fehlersuche, wie die Ursache und Wirkung von Programmfehlern und die grundsätzliche Zielsetzung des Debugging-Prozesses.

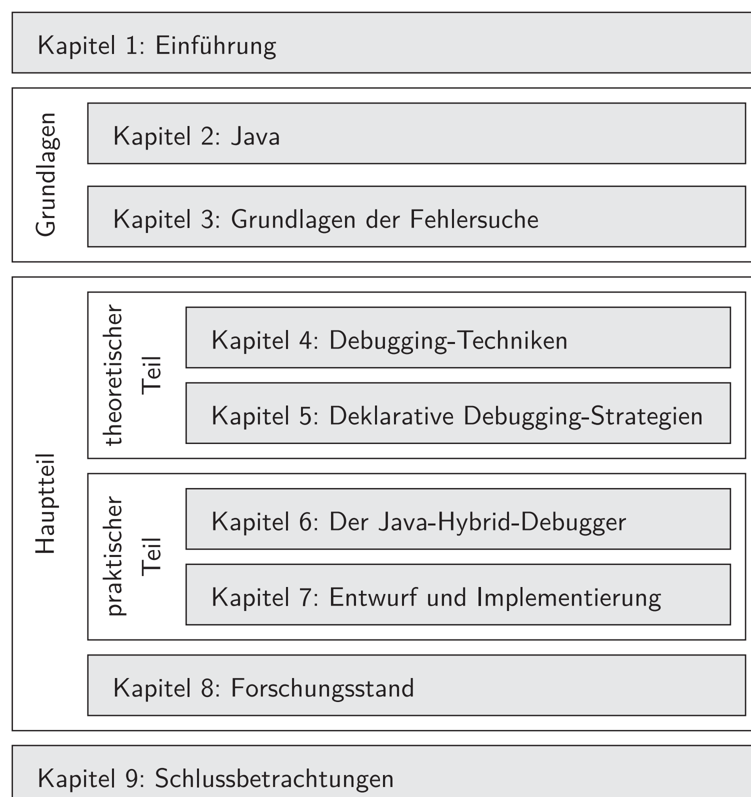


Abbildung 1.1: Aufbau der Arbeit.

Nach den Grundlagen folgt der Hauptteil dieser Arbeit. Der Hauptteil beginnt mit einem theoretischen Teil, in dem die hybride Debugging-Methode

für Java-Programme entwickelt wird. Im Kapitel 4 werden zunächst Trace-Debugging, Omniscient-Debugging und deklaratives Debugging vorgestellt und verglichen. Anschließend wird aus Omniscient-Debugging und Trace-Debugging eine hybride Debugging-Methode entwickelt. Das Kapitel 5 beschreibt die Entwicklung einer neuen Strategie für das deklarative Debugging.

Im praktischen Teil dieser Arbeit wird in Kapitel 6 JHyde vorgestellt. Dabei werden die Funktionen der Benutzeroberfläche beschrieben und der Ablauf des hybriden Debugging-Prozesses demonstriert. Das Kapitel 7 stellt die wesentlichen Aspekte des Entwurfs und der Implementierung von JHyde dar.

Im Kapitel 8 werden die Ergebnisse des theoretischen und praktischen Teils mit den in anderen Arbeiten entwickelten Methoden und Verfahren verglichen. Den Abschluss dieser Arbeit bilden in Kapitel 9 eine Zusammenfassung der Ergebnisse dieser Arbeit und ein Ausblick auf mögliche Erweiterungen und Verbesserungen von JHyde.

Kapitel 2

Java

Inhalt

2.1 Die Java-Technik 8

2.2 Die Programmiersprache Java 9

2.3 Die Java Virtual Machine 13

2.4 Fazit 34

Java zählt heute zu den populärsten Programmiersprachen in Wissenschaft und Praxis [91, 156]. Auch in dieser Arbeit spielt Java als Zielsprache für die Entwicklung des hybriden Debuggers JHyde eine zentrale Rolle.

Ziel dieses Kapitels ist es einen Überblick über die wichtigsten Aspekte und Eigenschaften von Java zu geben. Java ist bei weiten zu umfangreich, um es detailliert in diesem Kapitel beschreiben zu können. Die Java Language Specification [59] allein hat einen Umfang von über 600 Seiten. Im Rahmen dieses Kapitels werden daher nur die für diese Arbeit wesentlichen und essenziellen Aspekte von Java vorgestellt.

Im Folgenden wird zunächst in Abschnitt 2.1 der Aufbau der Java-Technik beschrieben. Die für diese Arbeit wichtigsten Elemente der Java-Technik, die Programmiersprache Java und die Java Virtual Machine, werden in den Abschnitten 2.2 bzw. 2.3 vorgestellt. Den Abschluss bildet eine Zusammenfassung der Ergebnisse im Abschnitt 2.4.

2.1 Die Java-Technik

Die Java-Technik (Java technology) ist eine Menge von Spezifikationen, die im Wesentlichen die Programmiersprache Java und die Java-Plattform (Java platform) definieren. Die wichtigsten Elemente der Java-Technik sind in der Abbildung 2.1 dargestellt. Zu diesen Elementen gehören:

Die Programmiersprache Java - Java-Programme werden in der Programmiersprache Java geschrieben. Diese Programme werden durch einen Java-Compiler in Klassendateien übersetzt. Die Klassendateien enthalten plattformunabhängigen Bytecode. Dieser Bytecode ist die Maschinensprache der Java Virtual Machine. Die Programmiersprache Java wird im Abschnitt 2.2 genauer beschrieben.

Das Java Development Kit (JDK) - Das JDK ist eine von Sun Microsystems herausgegebene Sammlung von Werkzeugen zur Entwicklung von Java-Programmen. Neben dem Java-Compiler gibt es zahlreiche weitere Werkzeuge, wie zum Beispiel `javadoc` zur automatischen Generierung von Dokumentationen aus Quelltextdateien und `jar` zur Verwaltung von Java-Archiven. Darüber hinaus enthält das JDK auch eine Laufzeitumgebung für Java-Programme, das sogenannte Java Runtime Environment.

Das Java Runtime Environment (JRE) - Das JRE bildet die Java-Plattform zur Ausführung von Java-Programmen. Eine Java-Plattform besteht aus der *Java-API* und einer Instanz der *Java Virtual Machine*. Die Java-API [145] ist eine Sammlung von Standardbibliotheken, die grundlegende Klassen, Datenstrukturen und Funktionen der Java-Plattform implementieren. Von der Java-Plattform gibt es drei sogenannte Editionen, die *Java Platform Standard Edition* (Java SE) [144] für den Einsatz auf PCs, Servern und vergleichbaren Geräten, die *Java Platform Micro Edition* (Java ME) [143] für eingebettete Systeme [107], wie Mobiltelefone oder PDAs, und die *Java Platform Enterprise Edition* (Java EE) [142]. Die Grundlage der Java-Plattform ist die Java-VM, die das zugrunde liegende Betriebssystem abstrahiert und eine plattformunabhängige Laufzeitumgebung für den Bytecode der Java-Klassen bereitstellt. Die Java-VM wird im Abschnitt 2.3 beschrieben.

<table>
<tr><td colspan="2">Programmiersprache</td><td>Java-Quelltext (Abschnitt 2.2)</td></tr>
<tr><td rowspan="3">JDK</td><td></td><td>Java-Compiler</td></tr>
<tr><td rowspan="2">JRE</td><td>Java-API</td></tr>
<tr><td>Java Virtual Machine (Abschnitt 2.3)</td></tr>
<tr><td colspan="2">Betriebssystem</td><td>Solaris, Linux, Windows usw.</td></tr>
</table>

Abbildung 2.1: Die Java-Plattform.

2.2 Die Programmiersprache Java

Im Folgenden werden zunächst die Historie und die Entwurfsziele der Programmiersprache Java beschrieben. Im Anschluss daran werden die wichtigsten Sprachmerkmale und Konstrukte der Sprache Java vorgestellt.

2.2.1 Historie und Entwurfsziele

Java ist aus der Programmiersprache OAK (Object Application Kernel), die 1992 von Patrick Naughton, Mike Sheridan, James Gosling und Bill Joy und weiteren Programmierern im Auftrag von Sun Microsystems entwickelt wurde, hervorgegangen. Die Programmiersprache Java wurde 1995 erstmals der Öffentlichkeit vorgestellt. Die Integration in den Webbrowser Netscape Navigator gilt als einer der entscheidenden Auslöser, der Java zu einem frühen Durchbruch verholfen hat.

Die wichtigsten Ziele, die mit dem Entwurf der Programmiersprache Java verfolgt wurden, sind (vgl. [148, 149]):

Einfach, objektorientiert und vertraut - Der einfache Aufbau der Sprache Java soll Einsteigern zu einer steilen Lernkurve verhelfen. Die fundamentalen Konzepte der Java-Technik sollen schnell zu erfassen sein und Programmierern soll von Beginn an ein produktiver Umgang mit Java ermöglicht werden.

Durch den objektorientierten Ansatz folgt Java einem Programmierparadigma, das heute, ca. 30 Jahre nach seiner Entwicklung [162],

zu den verbreitetsten Paradigmen in der Softwareentwicklung zählt. Objektorientierte Konzepte sind gut geeignet, um die Anforderungen komplexer und netzwerkbasierter Umgebungen zu erfüllen.

Die Syntax von Java basiert zu größten Teilen auf der Syntax von C++, verzichtet aber auf einige fehleranfällige Eigenschaften und Sprachkonstrukte. Java verfügt zum Beispiel über eine automatische Speicherverwaltung, wodurch das Speichermanagement sehr viel einfacher und weniger fehleranfällig ist. Die Tatsache, dass Java viele Eigenschaften und Sprachkonstrukte von C++ übernommen hat, erleichtert für viele Programmierer den Umstieg auf Java.

Robust und sicher - Java Programme werden sowohl durch den Compiler als auch beim Laden durch die Java-VM verifiziert. Darüber hinaus hilft vor allem die extrem einfache Speicherverwaltung, die Verlässlichkeit von Java-Programmen zu erhöhen. In Java existiert keine explizite Zeigerarithmetik und Objekte werden ausschließlich durch einen automatischen Garbage-Collector [72] aus dem Speicher entfernt.

Java verfügt über viele eingebaute Mechanismen, die die Sicherheit vor schadhaftem Verhalten erhöhen. Zum Beispiel wird das Zuführen von Klassendateien durch einen sogenannten Class-Loader gesteuert. Zudem wird jede geladene Klassendatei vor der ersten Ausführung verifiziert. Ferner wird der Zugriff auf sicherheitskritische Funktionen durch einen Security-Manager koordiniert.

Architekturneutral und portabel - Java-Programme zielen auf den Einsatz in heterogenen und verteilten Systemen [38][152] und müssen daher portabel und architekturneutral sein. Die Java-Technik stellt diese Eigenschaften im Wesentlichen durch den Java-Bytecode und die Java-VM sicher. Der Java-Bytecode ermöglicht eine kompakte und plattformunabhängige Speicherung von Java-Programmen und sorgt für eine gute Portabilität. Die Plattformunabhängigkeit des Bytecodes wird gewährleistet, indem dieser auf der Zielplattform durch eine Java-VM interpretiert wird.

Leistungsfähig - Da Java eine interpretierte Sprache ist, besitzt sie einen Geschwindigkeitsnachteil gegenüber kompilierten Sprachen wie C++,

die direkt in Maschinencode übersetzt werden. Durch die Einführung von Just-in-time-Compilern [1][88] konnte die Effizienz von Java-Programmen deutlich gesteigert werden, so dass der Geschwindigkeitsnachteil heute deutlich geringer ausfällt.

Thread-basiert und dynamisch - Die Unterstützung von Multithreading ist direkt in die Programmiersprache Java integriert. Java bietet eigene Sprachkonstrukte, mit denen das von Hoare beschriebene Konzept von Monitor und Bedingungsvariablen [66] zur Synchronisation von Threads umgesetzt werden kann.

In Java werden Klassen, Attribute und Methoden über den Konstantenpool referenziert. Diese Referenzen werden erst dann gebunden, wenn sie während der Laufzeit tatsächlich benötigt werden. Da das Linking bei Java nicht statisch, sondern dynamisch durchgeführt wird, ermöglicht es Java zum Beispiel während der Laufzeit weitere Klassen zu laden und auch vorhandene Klassen zu überschreiben.

2.2.2 Grundlegende Sprachmerkmale

Der Aufbau und die Struktur der Programmiersprache Java wird durch die Java Language Specification [150] definiert. Zu den wichtigsten Strukturen und Merkmalen von Java zählen:

Datentypen - Java ist eine streng getypte Sprache und verfügt über acht primitive Datentypen und drei Referenz-Datentypen. Zu den primitiven Datentypen zählen: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` und `double`. Zu den Referenztypen gehören Arrays, Klassen-Instanzen und Interface-Instanzen.

Variablen - Variablen entsprechen den Speicherplätze eines Java-Programms. Jede Variable ist typisiert und speichert einen Wert, der während des Programmablaufs ausgelesen und überschrieben werden kann. In Java gibt es vier Typen von Variablen: Instanzvariablen, Klassenvariablen, Array-Variablen und lokale Variablen.

Ausdrücke - In Java ist ein Ausdruck eine Kombination von Werten, Variablen, Operatoren und Methodenaufrufen, die entsprechend den Regeln der Operatorpräzedenz zu einem Wert, einer Variablen oder nichts

ausgewertet wird. Ein Ausdruck wertet zu einer Variablen aus, wenn er einem Zuweisungsoperator entspricht. Das Ergebnis der Auswertung der rechten Seite der Zuweisungsoperation wird in diesem Fall in der Variablen auf der linken Seite der Zuweisungsoperation gespeichert. Die Variable der linken Seite wird in diesem Zusammenhang auch als L-Wert bezeichnet. Ein Ausdruck wertet genau dann zu nichts aus, wenn es sich um einen Methodenaufruf mit dem Rückgabewert `void` handelt. In allen anderen Fällen wertet ein Ausdruck zu einem Wert oder einer Variablen aus.

Die Auswertung eines Ausdrucks kann Seiteneffekte produzieren, da Ausdrücke eingebettete Zuweisungen, Inkrementoperatoren, Dekrementoperatoren und Methodenaufrufe enthalten können.

Anweisungen - In Java werden Kontroll- und Datenfluss eines Programms durch Anweisungen gesteuert. Einige Anweisungen sind hierarchisch aufgebaut und enthalten weitere Anweisungen als Teil ihrer Struktur. Eine Übersicht der Anweisungen ist in der Tabelle 2.1 dargestellt. Zu der ersten Gruppe von Anweisungen gehören alle Ausdrücke, die als Anweisung verwendet werden dürfen. Hierzu zählen: Zuweisung, Inkrement, Dekrement, Methodenaufruf und der `new`-Operator zur Erzeugung einer neuen Instanz. Durch Verzweigungsanweisungen kann die Ausführung einzelner Anweisungsblöcke an Bedingungen geknüpft werden. Die Schleifenanweisungen dienen dazu, eine Folge von Anweisungen in Abhängigkeit von einer Bedingung mehrfach auszuführen. Des Weiteren gibt es Anweisungen, die dem Werfen und Fangen von Exceptions dienen. Exceptions sind in Java ein Mittel zur Steuerung des Kontrollflusses in Ausnahmefällen. Die `synchronized`-Anweisung dient zur Koordination des Kontrollflusses zwischen mehreren Threads. Durch die `return`-Anweisung wird die Ausführung der Methode beendet und die Kontrolle an die aufrufende Methode zurück gegeben.

Objektorientierung - Java ist eine klassenbasierte, objektorientierte Sprache. In Java gibt es neben Klassen-Instanzen zwei weitere spezielle Objekttypen: Array-Instanzen und Interface-Instanzen. In Java ist ein Array ein Objekt mit namenlosen Attributen, die sich über ihren Index adressieren lassen. Ein Interface entspricht einer abstrakten Klasse mit

ausschließlich abstrakten Methoden. In Java kann jede Klasse nur eine Oberklasse haben, jedoch mehrere Interfaces implementieren. Eine Klasse besitzt Attribute und Methoden und Konstruktoren. In Java sind alle Methoden mit Ausnahme von statischen Methoden, privaten Methoden und Konstruktoren polymorph. Für eine polymorphe Methode wird erst zur Laufzeit durch dynamisches Binden entschieden, welche Implementierung einer Methode ausgeführt wird.

Tabelle 2.1: Anweisungen der Programmiersprache Java.

Kategorie	Anweisung
Ausdruck	`=`, `++`, `--`, `m(...)`, `new`
Verzweigung	`if`, `switch`
Schleife	`while`, `do-while`, `for`, `break`, `continue`
Exceptions	`throw`, `assert`, `try`
Threads	`synchronized`
Rückgabe	`return`

Die Beschreibung der Programmiersprache Java ist an dieser Stelle absichtlich abstrakt gehalten, da nur ein genereller Überblick über die Eigenschaften und Merkmale von Java gegeben werden soll. Eine exakte Definition der Programmiersprache Java erfolgt in der Java Language Specification [150]. Ausführliche Erläuterungen und Beschreibungen der Sprache Java finden sich zum Beispiel bei Krüger [89] oder Eckel [47].

2.3 Die Java Virtual Machine

Die Java Virtual Machine (Java-VM) ist das Herzstück der Java-Technologie. Sie ist eine abstrakte Maschine, die die Hardware- und Plattformunabhängigkeit der Java-Programme gewährleistet. In diesem Abschnitt wird die grundlegende Funktionsweise der Java Virtual Machine erläutert. Im Folgenden werden die Architektur (2.3.1), die Speicherarchitektur (2.3.2), die

Ausführungseinheit (2.3.3), die Datentypen (2.3.4), die Struktur des Java-Bytecodes (2.3.5) und das Format der Java-Klassendatei (2.3.6) beschrieben.

2.3.1 Architektur

Die Java-VM ist eine von einer konkreten Rechnerarchitektur abstrahierte Laufzeitumgebung zur Ausführung von Java-Bytecode, die für viele Plattformen und Betriebssysteme (Windows, Solaris, Linux, OS X, usw.) verfügbar ist. Eine Java-VM ist eine Art Adapter zwischen dem Java-Bytecode und der realen Maschine bzw. dem Betriebssystem, auf dem sie ausgeführt wird. Durch die Java-VM ist es möglich, den plattformunabhängigen Java-Bytecode auf unterschiedlichen Systemen und Architekturen auszuführen.

Der Aufbau und die Struktur der Java-VM werden in der *Java Virtual Machine Specification* (JVM-Spezifikation) [151] festgelegt. Die wesentlichen Funktionseinheiten und Speicherbereiche der Java-VM sind in der Abbildung 2.2 dargestellt, wobei die Funktionseinheiten weiß und die Speicherbereiche grau hinterlegt sind.

Zu den Funktionseinheiten gehören der Klassenlader, der Verifizierer und die Ausführungseinheit. Der Klassenlader lädt Java-Klassendateien in den Speicher der Java-VM. Zu diesen Klassendateien gehören sowohl die Basisklassen der Java-API [145] als auch beliebige weitere benutzerdefinierte Klassen. Nach dem Laden wird eine Klasse an den Verifizierer weitergeleitet und auf syntaktische Korrektheit überprüft. Im Anschluss an die erfolgreiche Verifikation wird die Klasse im Methodenbereich der Java-VM gespeichert. Der Methodenbereich ist der gemeinsame Speicher aller Threads der Java-VM und enthält die implementierungsabhängigen internen Repräsentationen der geladenen Klassen. Neben dem Methodenbereich besitzt eine Java-VM zwei weitere Speicherbereichstypen, den Heap-Speicher und die Thread-Speicher. Der Heap-Speicher wird ebenfalls von allen Threads gemeinsam benutzt. Er verwaltet die Klassen- und Array-Instanzen der Java-VM. Ein Thread-Speicher ist immer exklusiv genau einem Thread zugeordnet. Er dient dazu, die für die Ausführung eines Threads benötigten Informationen zu speichern. Der Aufbau der Speicherarchitektur wird im Abschnitt 2.3.2 ausführlicher beschrieben.

Die Ausführungseinheit entspricht dem virtuellen Prozessor der Java-VM. Sie ist für die Steuerung des Programmablaufs zuständig. Die Ausführungseinheit einer Java-VM kann bei Bedarf auf die Funktionen des Betriebssystems zurückgreifen. Dies ist zum Beispiel für alle Eingabe- und Ausgabeoperationen notwendig. Die plattformabhängigen Betriebssystemoperationen sind innerhalb der virtuellen Maschine durch eine einheitliche Schnittstelle gekapselt. Da die kompilierten Java-Klassen nur mit der Laufzeitumgebung der abstrakten virtuellen Maschine interagieren, sind sie grundsätzlich in jeder konkreten Java-VM lauffähig, die der JVM-Spezifikation entspricht. Die Funktionsweise der Ausführungseinheit wird im Abschnitt 2.3.3 dargestellt.

Die Menge an Instruktionen, welche durch die Ausführungseinheit ausgeführt werden können, wird als Bytecode bezeichnet. Die Struktur einer Bytecode-Instruktion und die wichtigsten Klassen von Bytecode-Instruktionen werden im Abschnitt 2.3.5 erläutert. Abschließend wird im Abschnitt 2.3.6 das Format der Java-Klassendateien vorgestellt.

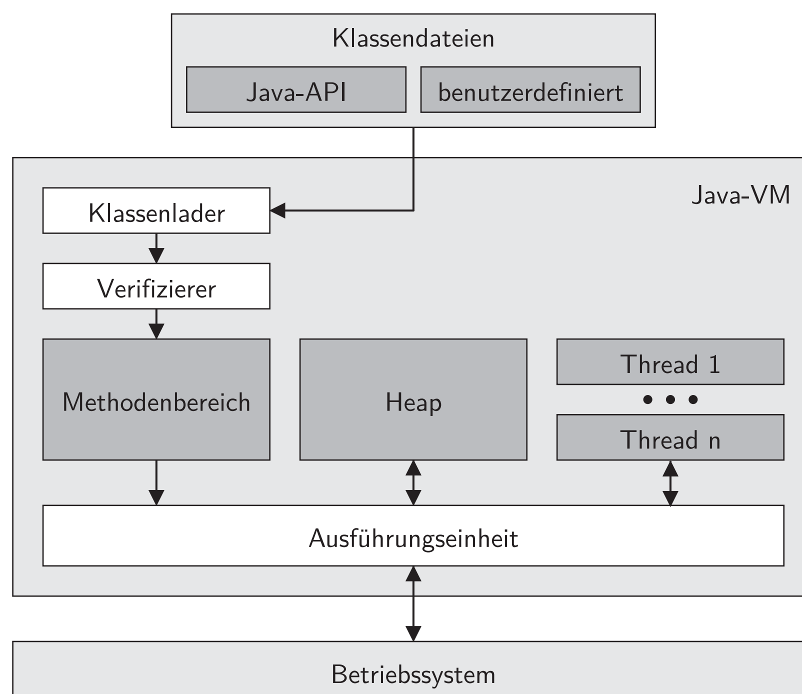


Abbildung 2.2: Architektur der Java-VM.

2.3.2 Speicherarchitektur

Das Speichermodell der Java-VM ist in der Abbildung 2.3 dargestellt. Das Modell unterscheidet zwischen Speicherbereichen, die von allen Threads gemeinsam benutzt werden und Speicherbereichen, die exklusiv einem einzigen Thread zugeordnet sind. Zu den gemeinsam genutzten Speicherbereichen gehören der Methodenbereich (Method Area) und der Heap.

2.3.2.1 Methodenbereich

Der Methodenbereich speichert für jede geladene Klasse eine interne Repräsentation. Die interne Repräsentation einer Klasse besteht aus den folgenden Elementen:

- Der *Konstantenpool* (Runtime Constant Pool) enthält die numerische Konstanten sowie Methoden- und Attributreferenzen einer Klasse. An den Stellen im Bytecode der Klasse, an der die Konstanten oder Referenzen benötigt werden, befindet sich eine Referenz auf den entsprechenden Eintrag des Konstantenpools. Der Konstantenpool ähnelt somit in seiner Funktion der Symboltabelle eines Compilers [4, S. 85–90] und ermöglicht es, den Bytecode einer Klasse kompakt zu halten.

- Zu der *Klassenstruktur* gehören der vollqualifizierte Klassenname, der vollqualifizierte Name der Oberklasse, die Modifikatoren (public, final, usw.), die implementierten Schnittstellen sowie die deklarierten Attribute und Methoden der Klasse.

- Zusätzlich besitzt jede Klasse einen Bereich, in dem die Werte der *Klassenattribute* hinterlegt sind.

- Zu jeder definierten Methode ist darüber hinaus in der Repräsentation der Klasse, der *Methoden-Bytecode* zur Ausführung der Methode gespeichert.

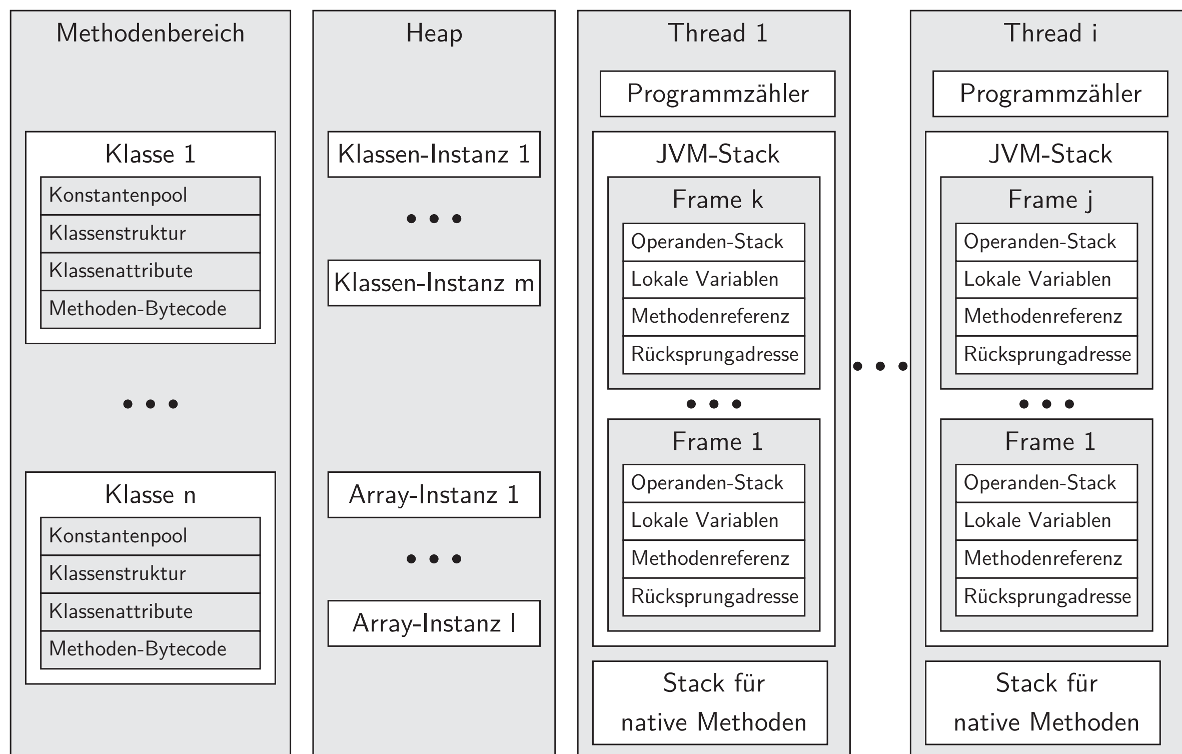


Abbildung 2.3: Speichermodell der Java-VM.

2.3.2.2 Heap

Der Heap dient zur Verwaltung aller Klassen- und Array-Instanzen der Java-VM. Der Speicherbereich einer Instanz kann in der Java-VM nie explizit freigegeben werden, da die JVM-Spezifikation für den Heap eine automatische Speicherverwaltung festlegt. Dies bedeutet, dass der Speicherbereich von nicht mehr referenzierten bzw. verwendeten Objekten automatisch durch einen sogenannten Garbage Collector [72] wieder freigegeben wird. Der Algorithmus und die technische Umsetzung der automatischen Speicherverwaltung werden durch die Spezifikation nicht vorgegeben; beide können bei der Implementierung einer Java-VM frei gewählt werden.

2.3.2.3 Thread-Speicher

In der Java-VM besitzt jeder Thread einen privaten Speicherbereich. In der Abbildung 2.3 sind dies die Bereiche Thread 1,. . . , Thread i. Um die parallele Ausführung mehrerer Threads zu unterstützen, benötigt jeder Thread einen eigenen *Programmzähler* (Program Counter). Der Programmzähler

enthält die Adresse der Anweisung, die durch den zugehörigen Thread gegenwärtig ausgeführt wird. Die Adresse bezieht sich dabei auf die Position im Methoden-Bytecode der gegenwärtig ausgeführten Methode.

Um die Methodenaufrufe der Threads verwalten zu können, besitzt jeder Thread einen eigenen JVM-Stack. Grundsätzlich entspricht der JVM-Stack in seiner Funktionsweise einem gewöhnlichen Stapelspeicher [37, S. 201 f.], der nach dem LIFO-Prinzip (Last In – First Out) arbeitet. Wenn während der Ausführung eines Threads ein Methodenaufruf auftritt, so wird für den neuen Methodenaufruf ein sogenannter Frame erzeugt und auf den JVM-Stack gelegt. Wird die Ausführung des gegenwärtigen Methodenaufrufs beendet, so wird der oberste Frame vom JVM-Stack entfernt. Infolgedessen entspricht der oberste Frame des JVM-Stacks immer dem gegenwärtig ausgeführten Methodenaufruf; er wird aus diesem Grund auch als aktiver Frame (active frame) bezeichnet.

Ein Frame wird verwendet, um Daten und Teilergebnisse zu speichern, das dynamische Auflösen (dynamic linking) von referenzierten Attributen und Methoden durchzuführen, Rückgabewerte zu speichern und Exceptions zu verarbeiten.

Operanden-Stack

Ein wesentliches Element des Frames ist der Operanden-Stack (operand stack). Der Operanden-Stack dient als Zwischenspeicher für die Operanden und die Ergebnisse aller Bytecode-Instruktionen. Wegen des verwendeten Operanden-Stacks entspricht das Rechnermodell der Java-VM einer Stack-Maschine [109, S. 17–91]. Neben dem Java-Bytecode basieren zum Beispiel die Programmiersprachen Forth [21] und PostScript [2] ebenfalls auf dem Rechnermodell der Stack-Maschine. Weil in einer Stack-Maschine die Operanden einer Instruktion implizit über den Stack übergeben werden, besitzen zum Beispiel die arithmetischen Instruktionen einer Stack-Maschine keine expliziten Operanden. Der Programmcode einer Stack-Maschine ist daher sehr kompakt und in der Regel deutlich kürzer als beispielsweise der Programmcode einer Registermaschine. Bei einer Registermaschine besitzen die Instruktionen in der Regel zwei oder drei Operanden.

Die Stack-Maschine ist zudem ein relativ abstraktes Rechnermodell. Im Falle der Java-VM erleichtert dies zum Beispiel die Implementierung der Java-VM auf Rechnerarchitekturen, die nur über wenige oder irreguläre Register verfügen. Zusätzlich erleichtert die kompakte und Stack-basierte Architektur des Befehlssatzes der Java-VM die Code-Optimierung durch Just-in-time-Compiler.

Die Funktionsweise des Operanden-Stack wird in der Abbildung 2.4 dargestellt. Die Abbildung zeigt den Java-Quelltext der Methode `calc`. Die Methode erwartet die drei Argumente `a`, `b` und `c`, für die sie das Ergebnis der Berechnung $\texttt{a} + \texttt{b} * \texttt{c}$ zurückgibt. Durch einen Java-Compiler wird der Quelltext der Methode in die ebenfalls in der Abbildung dargestellte Folge von sechs Bytecode-Instruktionen übersetzt. Für den Aufruf `calc(2, 3, 4)` sind die Auswirkung der Instruktionen auf den Operanden-Stack im unteren Teil der Abbildung dargestellt. Der Zustand des Operanden-Stack wird zu insgesamt 7 Zeitpunkten der Ausführung von `calc` gezeigt. Dabei zeigt der x-te Zustand den Operanden-Stack vor der Ausführung der x-ten Bytecode-Instruktion. Der erste Zustand zeigt somit den Operanden-Stack zu Beginn des Methodenaufrufs, d. h. vor der Ausführung der ersten Instruktion. Entsprechend zeigt der siebte Zustand den Operanden-Stack am Ende des Methodenaufrufs, d. h. nach der Ausführung der letzten Bytecode-Instruktion.

Zu Beginn der Ausführung ist der Operanden-Stack leer. Durch die Ausführung der Befehle 1–3 werden nacheinander die Werte der lokalen Variablen 0–2, die den Argumenten der Methode `a`, `b` und `c` entsprechen, auf den Stack kopiert. Nach der Ausführung des dritten Befehls enthält der Stack (4. Zustand) die Werte 2, 3 und 4, wobei der Wert 4 an oberster Stelle des Stacks liegt. Die vierte Instruktion (`imul`) benötigt insgesamt zwei Operanden, um diese miteinander zu multiplizieren. Die Operanden entnimmt sie nacheinander aus der obersten Position des Stacks. Anschließend wird die Multiplikation der entnommenen Operanden 4 und 3 ausgeführt und das Ergebnis zurück auf den Stack geschrieben. Nach der vollständigen Ausführung der Multiplikation enthält der Stack (5. Zustand) die Werte 2 und 12. Durch den nachfolgenden `iadd`-Befehl werden ebenfalls die beiden obersten Werte des Stacks als Operanden entnommen. Nach der Addition der Werte 12 und 2 wird das Ergebnis (14) zurück auf den Stack geschrieben. Der `ireturn`-Befehl führt dazu, dass die Ausführung der Methode `calc` be-

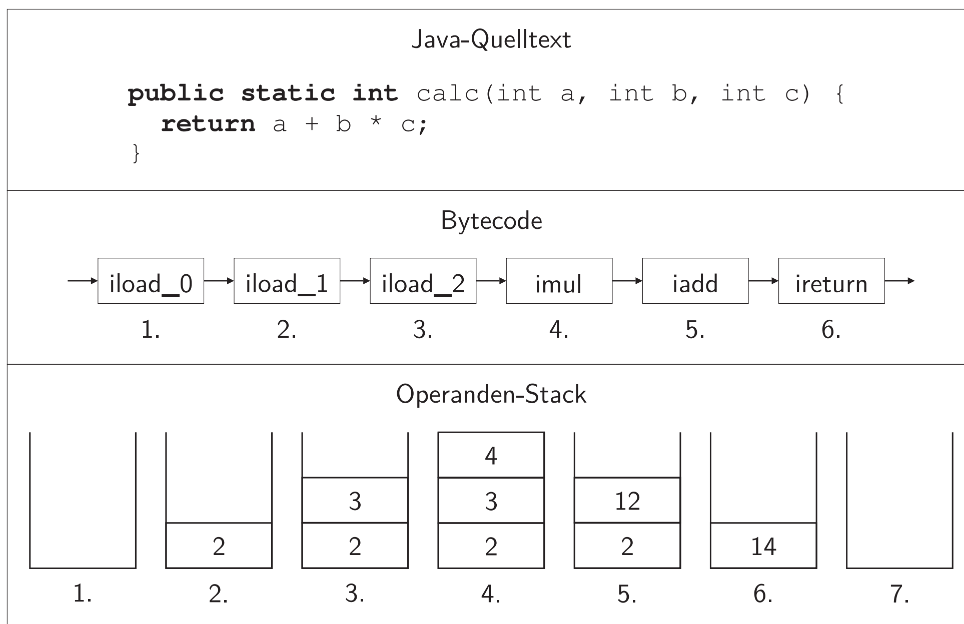


Abbildung 2.4: Operanden-Stack bei Ausführung der Methode `calc`.

endet wird und der oberste Eintrag vom Stack entnommen und als Resultat des Methodenaufrufs zurückgegeben wird.

Weitere Elemente eines Frames

Neben dem Operanden-Stack enthält ein Frame ein Array von *lokalen Variablen*. Dieses Array speichert die Werte aller Argumente des Methodenaufrufs und die Werte aller lokal definierten Variablen. Die Länge des Arrays ist fest und wird für jede Methode durch den Compiler bestimmt und in der Klassendatei gespeichert. Die lokalen Variablen werden zusammen mit dem Operanden-Stack für die Übergabe der Argumente eines Methodenaufrufs benutzt. Wird während der Programmausführung ein Methodenaufruf durchgeführt, dann werden zunächst die Argumente des Aufrufs aus dem Operanden-Stack des aktiven Frames entnommen. Anschließend wird ein neuer Frame auf den JVM-Stack gelegt und die Operanden werden in die entsprechenden Stellen des Arrays der lokalen Variablen des neuen Frames geschrieben.

Wenn ein neuer Methodenaufruf ausgeführt wird, dann muss die Position p, an der der Methodenaufruf aufgetreten ist, gespeichert werden, um nach Beendigung des Methodenaufrufs den Programmablauf an dieser Position

fortsetzen zu können. Bei der Erzeugung eines neuen Frames wird die Position p, d. h. der gegenwärtige Inhalt des Programmzählers, als *Rücksprungadresse* im neuen Frame gespeichert. Der Programmzähler wird anschließend auf den Anfang der Instruktionsliste gesetzt. Wird die Ausführung des Methodenaufrufs beendet, dann wird vor der Entfernung des aktiven Frames der Wert der Rücksprungadresse in den Programmzähler kopiert. Die Ausführung wird dann an der Position p des Methodenaufrufs fortgesetzt.

Jeder Frame besitzt zudem eine *Methodenreferenz*, die auf die Methodendeklaration der gegenwärtig ausgeführten Methode im Methodenbereich verweist. Die Methodendeklaration ist in der Klassenstruktur der entsprechenden Klasse gespeichert. Über die Methodenreferenz kann auf den Methoden-Bytecode und den Konstantenpool zugegriffen werden. Der Methoden-Bytecode wird zur Ausführung des Methodenaufrufs benötigt. Hierfür müssen zudem die im Bytecode vorhandenen Referenzen auf die Einträge des Konstantenpools zur Laufzeit aufgelöst werden. Zu diesem Zweck muss auf den Konstantenpool zugegriffen werden.

Die Übertragung von *Rückgabewerten* eines Methodenaufrufs wird mithilfe der Operanden-Stacks durchgeführt. Wenn ein Methodenaufruf einen Wert zurückgibt, dann liegt dieser, wie in der Abbildung 2.4, vor dem Aufruf der `return`-Anweisung auf der obersten Position des Operanden-Stacks. Durch die Ausführung der `return`-Anweisung wird dieser Wert vor der Entfernung des aktiven Frames dem Operanden-Stack entnommen. Anschließend wird der Rückgabewert auf die oberste Position des Operanden-Stacks des neuen aktiven Frames gelegt. In der aufrufenden Methode kann der Rückgabewert dann für die weitere Verarbeitung dem Operanden-Stack entnommen werden.

Stack für native Methoden

Neben dem JVM-Stack zur Koordination der Aufrufe von Java-Methoden besitzt der Thread-Speicher auch einen Stack zur Koordination von nativen Methodenaufrufen. Die Struktur und die Funktionsweise dieses Stacks gleichen dem JVM-Stack, ist jedoch von der Architektur des zugrundeliegenden Betriebssystems bzw. der Rechnerarchitektur, auf der die Java-VM läuft, abhängig. Der genaue Aufbau dieses Stacks ist daher durch die JVM-Spezifikation nicht weiter definiert.

2.3.3 Ausführungseinheit

Die Ausführungseinheit der Java-VM entspricht dem Prozessor der virtuellen Maschine und ist für die Steuerung des Programmablaufs zuständig. Der Ausführungseinheit ist zu jeder Zeit der Speicherbereich eines Threads zugeordnet. Dieser Thread wird als aktiver Thread bezeichnet. Die Ausführungseinheit interagiert mit dem Programmzähler, dem JVM-Stack und ggf. dem Stack für native Methodenaufrufe des aktiven Threads. Auf diese Weise setzt sie die Ausführung des aktiven Threads fort. Im Wesentlichen durchläuft die Ausführungseinheit hierfür die in Listing 2.1 dargestellte Schleife.

Listing 2.1: Schematischer Programmablauf der Ausführungseinheit.

```
1 do {
2   hole nächste Instruktion;
3   hole Operanden vom Operanden-Stack;
4   wandle Instruktion in nativen Maschinencode um;
5   führe nativen Maschinencode aus;
6   schreibe Resultat(e) auf den Operanden-Stack;
7   Programmzähler anpassen;
8 } while (Ausführung nicht beendet)
```

Zu Beginn eines Schleifendurchlaufs ermittelt die Ausführungseinheit die auszuführende Bytecode-Instruktion. Hierfür liest sie aus dem im Methodenbereich hinterlegten Methoden-Bytecode die Instruktion, die durch den Programmzähler referenziert wird. Dann entnimmt sie die benötigten Parameter der Operation aus dem Operanden-Stack. Die Instruktion wird dann in nativen Maschinencode umgewandelt und anschließend ausgeführt. Wenn die Instruktion ein oder mehrere Resultate produziert, dann werden diese auf den Operanden-Stack gelegt. Zuletzt wird der Programmzähler auf die Position der nächsten auszuführenden Instruktion gesetzt. Für den Fall, dass die zuletzt ausgeführte Instruktion keine Sprunganweisung oder eine bedingte Sprunganweisung mit nicht erfüllter Bedingung war, ist dies die nachfolgende Instruktion im Methoden-Bytecode. Anderenfalls wird der Programmzähler auf die Zieladresse der Sprunganweisung gesetzt. Solan-

ge der Programmablauf nicht beendet ist, wird die Ausführung mit dem nächsten Schleifendurchlauf fortgesetzt.

Aus konzeptioneller Sicht besteht die Ausführungseinheit aus einer großen Mehrfachverzweigung (`switch`-Anweisung), die jeder Bytecode-Instruktion eine äquivalente Folge von nativem Maschinencode zuordnet. Wenn die Ausführungseinheit ein klassischer Interpreter ist, dann geschieht die Übersetzung des Bytecodes unmittelbar vor der Ausführung. Verbesserungen bezüglich der Ausführungsgeschwindigkeit lassen sich mit einem sogenannten Just-in-time-Compiler (JIT-Compiler) [1, 88] erreichen. Dieser stellt einen Mittelweg zwischen einem Interpreter und einem statischen Ahead-of-time-Compiler (AOT-Compiler) dar. Ein JIT-Compiler übersetzt ein Programm nicht, wie ein AOT-Compiler, vor der Ausführung, sondern dynamisch, d. h. erst während der Ausführung, in nativen Maschinencode. Im Unterschied zu einem Interpreter werden ganze Anweisungsblöcke in Maschinencode übersetzt und gespeichert. Einmal übersetzte Programmteile müssen bei erneuter Ausführung nicht erneut übersetzt werden. Unter bestimmten Voraussetzungen kann ein JIT-Compiler schnelleren Code erzeugen als ein AOT-Compiler, da er Closed-World-Annahmen [18, S. 75] treffen und dynamische Optimierungen [12, 23] durchführen kann.

Wenn ein Java-Programm aus mehreren Threads besteht, dann wird der aktive Thread während des Programmablaufs fortlaufend gewechselt. Der Thread-Speicher, der der Ausführungseinheit zugeordnet ist, wird dann im Verlauf der Programmausführung immer wieder gewechselt. Die Ausführungseinheit setzt dann immer die Ausführung des aktiven Threads fort.

In neueren Implementierungen der Java-VM, wie der Hotspot-VM [146], wird jeder Java-Thread auf einen System-Thread des Betriebssystems abgebildet. Das Thread-Management wird in diesem Fall nicht mehr von der Java-VM, sondern vom Betriebssystem durchgeführt. Besonders für Rechnerarchitekturen mit mehreren Rechenkernen kann durch diese Maßnahme die Ausführungsgeschwindigkeit von Java-Programmen erheblich gesteigert werden. Wenn die Java-Threads auf System-Threads abgebildet werden, dann wird jedem Java-Thread eine separate Instanz der Ausführungseinheit zugeordnet.

2.3.4 Datentypen

Java ist eine statisch getypte Programmiersprache, da die Typkorrektheit weitestgehend vom Compiler während der Übersetzung in Java-Bytecode geprüft wird. Java besitzt einfache bzw. primitive Datentypen und Referenzdatentypen. Innerhalb der Java-VM werden die in der Tabelle 2.2 dargestellten Datentypen unterschieden.

Die Tabelle gibt für jeden Datentyp das Format, die Größe des Datentyps in Bits, ein Präfix und die Zugehörigkeit zu den Mengen P, V, O, Q und Z an. Das Format gibt an, in welcher Darstellung die Elemente des Datentyps gespeichert werden. Das Präfix eines Datentyps wird dazu verwendet, in den Mnemonics der Java-Bytecode-Instruktionen Typinformationen zu integrieren. Weitere Informationen hierzu finden sich im nachfolgenden Abschnitt über die Bytecode-Instruktionen.

Im Folgenden werden fünf Klassen bzw. Gruppen von Datentypen unterschieden:

- Die Gruppe P umfasst alle Datentypen, die auch in der Programmiersprache Java vorhanden sind. Zu dieser Gruppe gehören alle Datentypen mit Ausnahme des Datentyps `returnAddress`. Dieser Datentyp steht nur innerhalb der Java-VM im übersetzten Bytecode zur Verfügung und dient als Rücksprungadresse für die Ausführung von Subroutinen durch die Befehle `jsr`, `ret` und `jsr_w` (vgl. Abschnitt 2.3.5).
- Die Gruppe V enthält alle Datentypen, die durch die Java-VM unterstützt werden. Diese Gruppe umfasst alle Datentypen außer den Datentyp `boolean`. Dieser Datentyp wird durch die Java-VM nur in sehr geringem Umfang unterstützt. So gibt es zum Beispiel keine Bytecode-Instruktionen, die mit dem Datentyp `boolean` operieren. Bei der Übersetzung der Programmiersprache Java werden daher alle Ausdrücke, die `boolean`-Datentypen verwenden, in Ausdrücke mit `int`-Datentypen übersetzt. Der Datentyp `boolean` wird innerhalb der Java-VM nur für Methodensignaturen und Attributdatentypen benutzt.
- Die Gruppe O beinhaltet alle Datentypen, auf denen für die Java-VM Operationen definiert sind. Der Operanden-Stack kann ausschließlich

Werte enthalten, deren Datentyp in der Menge O enthalten ist. Zu der Gruppe O gehören alle Datentypen der Gruppe V mit Ausnahme der Datentypen `char`, `short` und `byte`. Diese Datentypen werden beim Laden auf den Operanden-Stack in den Datentyp `int` konvertiert. Analog können die Werte des Operanden-Stacks bei Bedarf in diese Datentypen zurück konvertiert werden, bevor sie in den Speicherbereich einer Variablen bzw. eines Attributs geschrieben werden.

- Die Gruppe Q umfasst alle nummerischen Datentypen der Menge O.
- Die Gruppe Z umfasst alle ganzzahligen Datentypen der Menge Q.

2.3.5 Bytecode-Instruktionen

2.3.5.1 Aufbau

Durch einen Java-Compiler wird der Quelltext eines Java-Programms in sogenannten Bytecode übersetzt. Der Bytecode ist eine Menge von Befehlen, die von der Java-VM ausgeführt werden können. Eine Anweisung für die Java-VM besteht aus einem Opcode, der die auszuführende Operation

Datentyp	Format	Bits	Präfix	P	V	O	Q	Z
`int`	Signed Integer	32	i	✓	✓	✓	✓	✓
`long`	Signed Integer	64	l	✓	✓	✓	✓	✓
`float`	IEEE754-Float	32	f	✓	✓	✓	✓	
`double`	IEEE754-Float	64	d	✓	✓	✓	✓	
`reference`	<Referenz>	32	a	✓	✓	✓		
`returnAddress`	<Adresse>	32	a		✓	✓		
`char`	Unicode-Zeichen	16	c	✓	✓			
`short`	Signed Integer	16	s	✓	✓			
`byte`	Signed Integer	8	b	✓	✓			
`boolean`	`true`, `false`	1	z	✓				

Tabelle 2.2: Datentypen der Java-Plattform.

spezifiziert, gefolgt von keinem oder mehr *Bytecode-Operanden*, die als Argumente der Operation dienen. Zusätzlich zu den innerhalb einer Bytecode-Instruktion spezifizierten Bytecode-Operanden erwartet eine Operation keinen oder mehr *Stack-Operanden* auf dem Operanden-Stack. Darüber hinaus wird der Operanden-Stack verwendet, um das Resultat einer Operation zurückzugeben.

Die Länge jedes Opcode beträgt genau ein Byte. Die Anzahl und die Typen der Operanden werden ausschließlich durch den Opcode einer Operation bestimmt. Die Tabelle 2.2 zeigt eine Auflistung der wichtigsten Bytecode-Instruktionen. Die Instruktionen sind nach ihrem Verwendungszweck in unterschiedliche Gruppen eingeteilt. Auf der ersten Gruppierungsebene können die im Folgenden beschriebenen drei Gruppen von Operationen unterschieden werden.

2.3.5.2 Datenflussoperationen

Die Datenflussoperationen umfassen alle Operationen, mit denen Daten zwischen den Speicherbereichen der Java-VM transferiert werden. Auf der zweiten Ebene können die Operationen anhand der Speicherbereichstypen, die an der Datenflussoperation beteiligt sind, weiter gruppiert werden. Dabei werden insgesamt 6 Speicherbereichstypen unterschieden: Attribute, Array-Elemente, lokale Variablen, Konstanten und der Operanden-Stack. Aus diesen ergeben sich die folgenden Gruppen von Datenflussoperationen:

- Die erste Gruppe von Befehlen dient dazu, Daten zwischen einem Attribut und dem Operanden-Stack auszutauschen. Die `get`-Befehle kopieren den Wert eines Attributs auf den Operanden-Stack und die `put`-Befehle speichern einen Eintrag des Operanden-Stacks in einem Attribut. Anhand der Befehlsendungen (`static` und `field`) wird unterschieden, ob ein Klassen- oder ein Instanzattribut gelesen bzw. geschrieben wird. Die Befehle besitzen zudem einen Bytecode-Operanden, der auf eine Attributreferenz im Konstantenpool der Klasse verweist. Die Attributreferenz enthält den vollqualifizierten Namen und den Datentyp des Attributs. Wird ein Instanzattribut gelesen oder geschrieben, dann wird zusätzlich die Referenz der betroffenen Instanz dem Operanden-Stack entnommen.

Tabelle 2.3: Klassifikation der Bytecode-Instruktionen.

	Verwendung	Opcode
Datenfluss	Attribut → Stack	getstatic, getfield
	Stack → Attribut	putstatic, putfield
	Arrayelement → Stack	<V>aload
	Stack → Arrayelement	<V>astore
	lokale Variable → Stack	<O>load, <O>load_<n>
	Stack → lokale Variable	<O>store, <O>store_<n>
	lokale Variable → lokale Variable	iinc
	→ Stack	new, newarray, anewarray, multianewarray
	Konstante → Stack	aconst_null, <O>const_<n>, ldc, ...
	Stack → Stack	pop, dup, swap, ...
Kontrollfluss	Sprünge	if⋆, goto⋆, jsr⋆, tableswitch, lookupswitch, ...
	Methoden	invokevirtual, invokespecial, invokestatic, invokeinterface, return, <O>return
	Exceptions	athrow, checkcast
	Threads	monitorenter, monitorexit
Rechenwerk	Grundrechenarten	<Q>add, <Q>sub, <Q>mul, <Q>div, <Q>rem, <Q>neg
	Bitoperatoren	<Z>shl, <Z>shr, <Z>ushr, <Z>and, <Z>or, <Z>xor
	Vergleichsoperatoren	dcmpg, dcmpl, fcmpg, fcmpl, lcmp
	Typkonvertierung	?2?
	Instanzeigenschaften	instanceof arraylength

- Die Befehle für den Datentransfer zwischen Array-Element und Stack besitzen ein Präfix, in dem der Datentyp des transferierten Wertes codiert ist. Für jeden Datentyp der Gruppe V gibt es einen eigenen `aload`- bzw. `astore`-Befehl. Weil die Typinformationen direkt durch den Opcode bestimmt werden, müssen diese nicht durch einen zusätzlichen Bytecode-Operanden definiert werden. Dies hat eine größere Anzahl von Bytecode-Instruktionen, aber einen kompakteren Bytecode und somit eine schnellere Ausführungsgeschwindigkeit zur Folge. Die Stack-Operanden sind die Array-Referenz und der Index des beteiligten Elements. Die `astore`-Befehle entnehmen zusätzlich den zu speichernden Wert vom Operanden-Stack.

- Die Befehle für den Datentransfer zwischen einer lokalen Variablen und dem Stack besitzen ebenfalls ein Datentyp-Präfix. Hier sind die möglichen Datentypen allerdings auf die Gruppe O beschränkt, da die Datentypen `char`, `short` und `byte` sowohl für den Operanden-Stack als auch für lokale Variablen auf den Datentyp `int` abgebildet werden. Von den `load`- und `store`-Befehlen gibt es zusätzlich Versionen mit einem sogenannten Suffix. Das Suffix ist Element der Menge $0, 1, 2, 3$ und repräsentiert den Index der lokalen Variablen, auf die zugegriffen wird. Für den Zugriff auf die besonders häufig verwendeten lokalen Variablen mit niedrigem Index wird auf diese Weise ein Bytecode-Operand eingespart. Bei den Befehlen ohne Index-Suffix muss der Index durch einen weiteren Stack-Operanden angegeben werden. Bei den `store`-Befehlen wird der zu speichernde Wert wiederum dem Operanden-Stack entnommen.

- Der Befehl `iinc` bildet eine eigene Gruppe, da er ohne Verwendung des Operanden-Stack den Wert einer lokalen `int`-Variablen direkt um einen konstanten Wert erhöht. Index und Inkrement werden durch Bytecode-Operanden spezifiziert. Der Befehl dient zum Beispiel dazu, die Manipulation von Schleifenzählern zu beschleunigen.

- Die Befehle zur Erzeugung neuer Instanzen besitzen keinen Quellspeicher, da sie keine Daten transferieren. Sie erzeugen eine neue Klassen- oder Array-Instanz und legen diese auf dem Operanden-Stack ab.

- Eine weitere Gruppe von Befehlen ermöglicht es im Konstantenpool der Klasse definierte Zahl- oder String-Konstanten auf den Operanden-Stack zu laden.
- Zur letzten Gruppe gehören alle Befehle zur Verwaltung des Operanden-Stack. Mithilfe von Befehlen wie `pop`, `dup` und `swap` können Einträge des Operanden-Stack entfernt, dupliziert und vertauscht werden.

2.3.5.3 Kontrollflussoperationen

Zur Gruppe der Kontrollflussoperationen gehören alle Operationen, mit denen der Kontrollfluss von Java-Programmen gesteuert werden kann. Diese lassen sich in die folgenden Gruppen unterteilen:

- Die Sprung-Befehle dienen dazu, den Kontrollfluss, bedingt oder unbedingt, an eine Instruktion im Bytecode der Methode zu transferieren, die nicht der nachfolgenden Instruktion entspricht.
- Der Befehl zum Aufruf einer Methode ist abhängig vom Typ der aufgerufenen Methode. Klassenmethoden werden durch den Befehl `invokestatic` aufgerufen. Für Klassenmethoden kann die aufzurufende Methode statisch gebunden, d. h. durch den Compiler bestimmt werden. Der vollqualifizierte Name und die Signatur der aufzurufenden Methode werden über eine Referenz auf einen Eintrag des Konstantenpools als Bytecode-Operand übergeben. Die Argumente des Methodenaufrufs werden dem Operanden-Stack entnommen. Instanzmethoden, mit Ausnahme von Konstruktoren und privaten Methoden, werden mit dem Befehl `invokevirtual` aufgerufen. Hierbei wird die aufzurufende Methode erst zur Laufzeit anhand der tatsächlichen Klasse der Instanz durch dynamisches Binden [18, S. 41f.][164, S. 179] bestimmt. Neben den Argumenten der Funktion liegt daher auf dem Operanden-Stack auch die Instanz, deren Methode aufgerufen wird. Der `invokeinterface`-Befehl dient zum Aufruf von Methoden eines Interface-Typen und wird ebenfalls dynamisch zur Laufzeit gebunden. Der Grund für die Unterscheidung zwischen Interface- und Instanzmethoden sind die Methodentabellen [164, S. 187], die intern von der Java-VM für das dynamische Binden

verwendet werden.[1] Eine Methodentabelle wird beim Laden einer Klasse durch die Java-VM erzeugt. Sie enthält Referenzen auf die Bytecodes aller Methoden, die auf einer Instanz der Klasse aufgerufen werden können. Eine Instanz-Methode belegt unabhängig von der tatsächlichen Klasse der Instanz immer denselben Index in der Methodentabelle. Dies ist möglich, da Java nur Einfachvererbung erlaubt. Weil eine Klasse mehrere Interfaces implementieren kann, gilt diese Eigenschaft nicht für Interface-Methoden. In zwei Klassen, die dasselbe Interface implementieren, können den Implementierungen derselben Methode unterschiedliche Indizes in der Methodentabelle zugeordnet werden. In frühen Implementierungen der Java-VM war der Aufruf von Interface-Methoden daher sehr zeitaufwendig. Mittlerweile existieren jedoch Techniken, mit denen sich der Aufruf von Interface-Methoden effizient durchführen lässt [6]. Die `invokespecial`-Methode dient ebenfalls zum Aufruf von Instanz-Methoden. Im Unterschied zu `invokevirtual` und `invokeinterface` wird die aufzurufende Methode allerdings nicht dynamisch, sondern statisch gebunden. Diese Funktionalität wird für das Schlüsselwort `super` und private Methoden benötigt. Das Schlüsselwort `super` ermöglicht es, eine Methode der Oberklasse aufzurufen. Hierfür ist der Befehl `invokevirtual` nicht ausreichend, da bei überschriebenen Methoden das dynamische Binden nicht zum gewünschten Ergebnis führen würde. Mit `invokespecial` kann durch das statische Binden eine explizite Methodenimplementierung ausgeführt werden. Für den Aufruf privater Methoden wird `invokespecial` benötigt, da eine private Methode m einer Oberklasse O in einer Unterklasse U überschrieben werden kann. Wenn nun eine Instanz von U in einer von O geerbten Methode die Methode m aufruft, dann muss die in O implementierte Version von m ausgeführt werden. Würde der Aufruf von m dynamisch gebunden, dann würde dies dazu führen, dass die in U implementierte Version von m ausgeführt wird. Die Ausführung einer Methode wird durch einen der `return`-Befehle beendet. Der einfache `return`-Befehl beendet den Methodenaufruf ohne Rückgabewert. Die übrigen `<O>return` Befehle entnehmen

[1] Die Verwendung von Methodentabellen ist durch die JVM-Spezifikation nicht vorgeschrieben, sie ist aber die gebräuchlichste Technik für die Umsetzung des dynamische Bindens.

den Rückgabewert vom Operanden-Stack des aktiven Frames und legen ihn auf den Operanden-Stack des Frames der aufrufenden Methode.

- Ein weiteres Mittel zur Steuerung des Kontrollflusses in Java sind Exceptions. Die JVM-Spezifikation definiert zwei Befehle, mit denen eine Exception explizit geworfen werden kann. Mit dem Befehl `athrow` wird das Exception-Objekt geworfen, welches sich auf der obersten Position des Operanden-Stack befindet. Das Werfen einer Exception hat zur Folge, dass der JVM-Stack ausgehend vom aktiven Frame nach dem ersten Exception-Handler durchsucht wird, der eine Exception vom Typ des geworfenen Exception-Objekts fängt. Durch Manipulation des Programmzählers wird der Kontrollfluss an der Stelle fortgesetzt, an der sich die Instruktionen zur Behandlung der Exception befinden. Der Befehl `checkcast` erzeugt und wirft eine `CheckCastException`, wenn das oberste Element des Operanden-Stack nicht `null` ist und nicht dem Typ des ersten Bytecode-Operanden entspricht.

- Die Synchronisation von Threads wird durch die Befehle `monitorenter` und `monitorexit` koordiniert. Beide Befehle erwarten als obersten Stack-Operanden ein Objekt o. Durch `monitorenter` wird die Ausführung solange angehalten, bis dem gegenwärtigen Thread exklusiver Zugang zum Monitor von o gewährt wird. Mit `monitorexit` verlässt der Thread den Monitor des Objekts o.

2.3.5.4 Rechenoperationen

Die Rechenoperationen interagieren ausschließlich mit dem Operanden-Stack und berechnen das Ergebnis einer Funktion mit einem oder zwei Parametern. Jede Rechenoperation entnimmt die benötigten Operanden aus dem Operanden-Stack und legt anschließend das Ergebnis der Operation zurück auf den Operanden-Stack. Die arithmetischen Operationen lassen sich in die folgenden Untergruppen einteilen:

- Zu den Grundrechenarten gehören Addition, Subtraktion, Multiplikation, Division, Restwert und Vorzeichenumkehr. Die Operationen sind für alle Datentypen der Menge Q definiert.

- Die Bitoperatoren werden auf die einzelnen Bits der Operanden angewendet und sind ausschließlich für die Datentypen `long` und `int` definiert.
- Die Vergleichsoperationen dienen zum Vergleich von `double`-, `float`- und `long`-Werten. Alle Operationen entnehmen zwei Werte, w_1 und w_2, aus dem Operanden-Stack und vergleichen diese. Das Ergebnis des Vergleichs ist 1 für $w_1 > w_2$, 0 für $w_1 = w_2$ und -1 für $w_1 < w_2$. Die Vergleichsoperationen werden benötigt, da die Datentypen nicht durch die bedingten Sprungbefehle (`if*`) unterstützt werden.
- Die Java-VM ermöglicht die explizite Konvertierung von numerischen Datentypen. Zum Beispiel wird durch den Befehl `i2l` ein `int`-Wert in einen `long`-Wert konvertiert. Die Menge aller Konvertierungsoperationen wird durch den Graphen in der Abbildung 2.5 dargestellt.
- Mit den Typoperatoren können die Eigenschaften bestimmter Typen ermittelt werden. Mit dem `instanceof`-Befehl kann überprüft werden, ob ein Objekt von einem bestimmten Typ ist. Der `arraylength`-Befehl ermittelt die Länge eines Feldes.

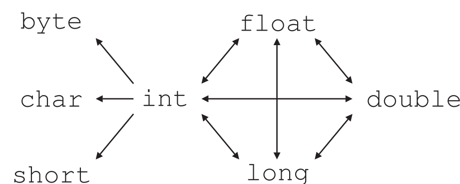


Abbildung 2.5: Explizite Typkonvertierungen in der Java-VM. Die Pfeile geben an, für welche Typen es explizite Konvertierungsoperationen gibt.

2.3.6 Format der Java-Klassendatei

Das Format einer Klassendatei (`class`-Datei) ist durch die JVM-Spezifikation exakt definiert. Jede Klassendatei enthält die vollständige Definition einer Klasse oder eines Interfaces und besitzt eine baumartige Struktur.

Der Aufbau einer Klassendatei ist in der Abbildung 2.6 dargestellt. Eine Klassendatei besteht aus den folgenden Elementen:

- Jede Klassendatei beginnt mit der sogenannten magischen Zahl $\mathrm{CAFEBABE}_{16}$, anhand derer sie als Klassendatei identifiziert werden kann. Anschließend folgt die Versionsnummer des Formats der Klassendatei. Die Versionsnummer der aktuellen Java-Plattform (Java SE 6) ist 50. Die magische Zahl und die Versionsnummer bilden zusammen den Kopf einer Klassendatei.

- Der Konstantenpool ist ein Array, dessen Elemente nummerische Konstanten, Zeichenkettenkonstanten oder Referenzen auf andere Elemente des Konstantenpools beinhalten. Die Einträge des Konstantenpools können von den Elementen der Klassendatei über ihren Index in der Klassendatei referenziert werden. Mithilfe des Konstantenpools sollen Redundanzen innerhalb der Klassendatei vermieden werden. Zum Beispiel wird innerhalb einer Methodendefinition nicht der Name der Methode selbst, sondern eine Referenz auf den Namen der Methode im Konstantenpool gespeichert. Ein Befehl zum Aufruf einer Methode enthält ebenfalls nicht den Namen der Methode, sondern eine Referenz auf die Definition des Namens im Konstantenpool. Im Konstantenpool wird der Name einer Methode nur einmal definiert, auch wenn dieser von mehreren Stellen in der Klassendatei referenziert wird. Innerhalb des Konstantenpools werden Redundanzen vermieden, indem komplexere Einträge durch Referenzen auf weniger komplexe Einträge zusammengesetzt werden.

- Nach dem Konstantenpool folgt die Beschreibung der Klasse bzw. des Interfaces welches durch die Klassendatei definiert wird. Die Beschreibung umfasst den Modifikator der Klasse (access_flag), den vollqualifizierten Namen der Klasse und den vollqualifizierten Namen der Oberklasse. Beide Namen sind im Konstantenpool definiert und werden referenziert.

- Es folgt ein Array, indem alle Schnittstellen definiert werden, die durch die Klasse bzw. das Interface implementiert werden. Jede Schnittstelle ist wiederum eine Referenz auf einen entsprechenden Eintrag im Konstantenpool.

- Auf die Schnittstellendefinitionen folgt ein Array, indem die Definitionen für alle Attribute der Klasse gespeichert sind. Von der JVM-Spezifikation werden die Attribute einer Klasse als Felder (fields) bezeichnet. Die wesentlichen Teile einer Attributdefinition sind der Modifikator, der Name und der Datentyp (descriptor) des Attributs. Der Name und der Datentyp werden durch Referenzen auf den Konstantenpool spezifiziert.
- Die Methodendefinitionen werden ebenfalls in einem Array gespeichert. Jede Methodendefinition enthält mindestens den Modifikator, den Namen, die Signatur und ein sogenanntes Code-Attribut. Das Code-Attribut speichert die maximale Höhe des Operanden-Stacks und die maximale Anzahl an lokalen Variablen. Diese Informationen werden beim Aufruf der Methode für die Erzeugung des Stack-Frames benötigt. Darüber hinaus enthält das Code-Attribut den vollständigen Bytecode der Methode.
- Zuletzt folgt ein Array mit optionalen Attributen. Hier kann zum Beispiel die Quelltextdatei angegeben werden, aus der die vorliegende Klassendatei erzeugt wurde. Ferner dienen die Attribute auch dazu, Annotationen zu speichern. Die optionalen Attribute machen das Format der Klassendatei erweiterbar. Eine Klassendatei ist auch auf einer Java-VM ausführbar, die die Attribute einer neueren Version des Klassendateiformats nicht erkennt. Die nicht erkannten Attribute werden durch die Java-VM ignoriert.

2.4 Fazit

In diesem Kapitel wurde die Struktur der Programmiersprache Java und die Funktionsweise der Java-VM erläutert. Diese Ausführungen dienen als Grundlage für die nachfolgenden Kapitel, in denen Debugging-Techniken für Java-Programme entwickelt (vgl. Abschnitt 4–5) und anschließend durch den Debugger JHyde realisiert werden (vgl. Abschnitt 6–7).

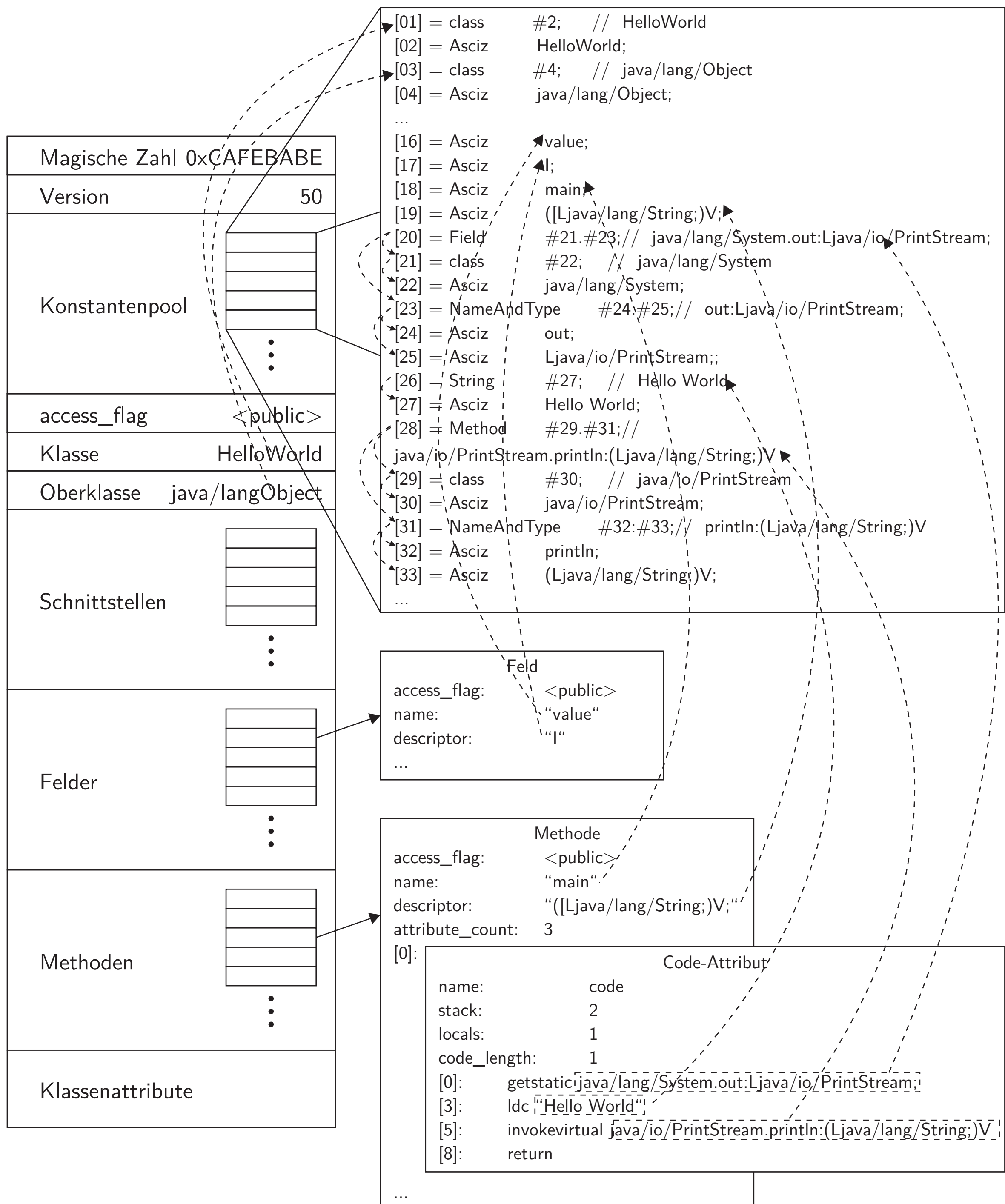


Abbildung 2.6: Format der Java-Klassendatei.

Kapitel 3

Grundlagen der Fehlersuche

Inhalt

3.1 Ursache und Wirkung von Programmfehlern 37
3.2 Infektionstypen 40
3.3 Debugging von Programmen 45
3.4 Fazit 48

Dieses Kapitel beschreibt die Grundlagen für das Debugging von Programmen. Hierzu werden im Abschnitt 3.1 zunächst grundlegende Begriffe des Debugging erläutert. Im Anschluss werden im Abschnitt 3.2 unterschiedliche Typen von Infektionen des Programmzustands analysiert. Im Abschnitt 3.3 werden der Ablauf und die Zielsetzung des Debugging-Prozesses formuliert und erläutert. Zum Abschluss werden die Ergebnisse im Abschnitt 3.4 zusammengefasst.

3.1 Ursache und Wirkung von Programmfehlern

Die Entwicklung von Software ist ein aufwendiger und komplexer Prozess. Dies führt dazu, dass die Implementierung umfangreicher Programme in der Regel fehlerbehaftet ist. Solange ein Programm nicht ausgiebig getestet und korrigiert ist, enthält es in der Regel Fehler. Diese Fehler können bei der Ausführung des Programms ein fehlerhaftes Verhalten verursachen und

schließlich zu einem fehlerhaften Ergebnis führen. Zur genauen Unterscheidung dieser unterschiedlichen Fehler werden in Anlehnung an Zeller [168, S. 3–21] sowie Müller et al. [110] drei Fehlerarten unterschieden:

- Ein *Defekt* (defect) beschreibt einen Fehler im Programmcode.
- Wird ein Defekt ausgeführt, verhält sich das Programm nicht so wie beabsichtigt; es wird von einem validen in einen *infizierten Zustand* (infection) überführt. Ein infizierter Zustand ist somit ein Fehler im Programmzustand. Während der weiteren Ausführung kann ein infizierter Zustand weitere infizierte Folgezustände verursachen.
- Ein infizierter Zustand kann zu einem nach außen hin erkennbaren Fehlverhalten des Programms führen. Diese erkennbare Abweichung des tatsächlichen vom beabsichtigten Programmverhalten wird als *Fehlerwirkung* (failure) bezeichnet. Eine Fehlerwirkung ist somit ein erkannter bzw. wahrgenommener infizierter Zustand.

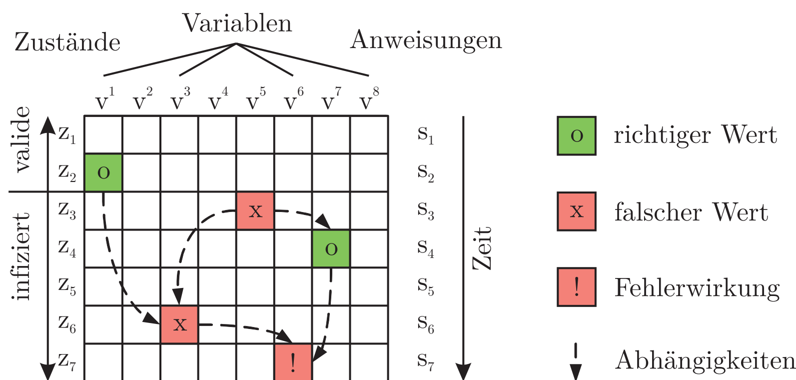


Abbildung 3.1: Vom Defekt zur Fehlerwirkung. Die defekte Anweisung s_3 überführt das Programm vom validen Zustand z_2 in den infizierten Zustand z_3, indem es der Variablen v^5 einen falschen Wert zuweist, wodurch der Wert v_3^5 infiziert wird. Der infizierte Zustand z_3 hat weitere infizierte Zustände $z_4, \ldots, z_7$ zur Folge. Diese führen schließlich zu einer erkennbaren Fehlerwirkung im Zustand z_7.

Der Zusammenhang zwischen Defekt, infiziertem Zustand und Fehlerwirkung wird in der Abbildung 3.1 verdeutlicht. Die Abbildung stellt einen Pro-

grammablauf dar. Der Ablauf ist eine Folge von Anweisungen $(s_1, \ldots, s_7)$, welche sequenziell ausgeführt werden. Die Anweisung s_i wird zum Zeitpunkt i ausgeführt und versetzt das Programm in den Zustand z_i. Der Zustand des Programms zum Zeitpunkt i umfasst die Werte aller Variablen zum Zeitpunkt i, d. h. $z_i = \{v_i^1, \ldots, v_i^8\}$. Ein Zustand z_i ist valide, wenn alle Werte $v_i^1, \ldots, v_i^8$ valide sind. Ist mindestens einer der Werte $v_i^1, \ldots, v_i^8$ infiziert, so ist auch der Zustand z_i infiziert. Zu Beginn der Programmausführung sind die Werte aller Variablen valide. Eine Anweisung kann den Wert einer oder mehrerer Variablen ändern. Ändert sich der Wert einer Variablen, so ist dies an der entsprechenden Stelle mit einem „o“ markiert, falls der neue Wert valide ist, oder mit einem „x“ markiert, falls der neue Wert infiziert ist. Nach Ausführung der Anweisungen s_1 und s_2 befindet sich das Programm weiterhin in einem validen Zustand (z_2). Durch die Ausführung der Anweisung s_3 wird der Variablen v^5 ein falscher Wert zugewiesen. Damit ist der Wert der Variablen 5 nach der Ausführung der 3. Anweisung (v_3^5) infiziert. Dies führt dazu, dass auch der Zustand z_3 infiziert ist. Während der folgenden Zustände bleibt die Variable v^3 solange infiziert, wie ihr tatsächlicher Wert nicht dem erwarteten Wert entspricht. Zwischen den Zuständen bestehen Abhängigkeiten. In der Abbildung sind diese Abhängigkeiten als Pfeile dargestellt. Zum Beispiel ist der Wert v_6^3 abhängig von den Werten v_2^1 und v_3^5. Aufgrund dieser Abhängigkeiten hat der infizierte Zustand z_3 die weiteren ebenfalls infizierten Zustände $z_4, \ldots, z_7$ zur Folge.

Häufig ist ein infizierter Zustand nicht unmittelbar erkennbar. In diesem Fall zeigt sich die Fehlerwirkung eines Defekts nicht sofort, sondern erst zu einem späteren Zeitpunkt der Programmausführung. Zum Beispiel tritt die Fehlerwirkung im vorliegenden Beispiel erst mit dem infizierten Zustand z_7 auf. Hier wird der infizierte Wert v_7^6 zum Beispiel durch Ausgabe auf der Konsole sichtbar; er wird damit zur Fehlerwirkung.

Ein Defekt bewirkt nicht zwangsläufig bei jeder Programmausführung eine Fehlerwirkung. Es ist grundsätzlich möglich, dass durch eine bestimmte Wahl der Programmparameter ein Defekt nicht ausgeführt wird und somit der Programmzustand nicht infiziert wird. Darüber hinaus kann ein Defekt einen infizierten Zustand verursachen, der nicht zu einer Fehlerwirkung führt. Dies ist der Fall, wenn infizierte Werte bzw. Zustände durch nachfolgende Anweisungen wieder in valide Werte bzw. Zustände überführt werden. Dieser Zusammenhang ist in der Abbildung 3.1 für den Wert v_4^7 dargestellt.

Dieser ist valide, obwohl er von dem infizierten Wert v_3^5 abhängig ist. Dies führt dazu, dass ein Programm, dessen Ausführung keine Fehlerwirkung zeigt, durchaus Defekte enthalten kann. Im Folgenden werden nur diejenigen Defekte betrachtet, die zu einer Fehlerwirkung führen, da nur für diese ein Debugging-Prozess angestoßen wird.

3.2 Infektionstypen

Im Rahmen dieser Arbeit wird das Klassifizierungsschema der Tabelle 3.1 verwendet, um unterschiedliche Typen von infizierten Werten zu unterscheiden. Das Schema unterscheidet in Abhängigkeit von der Ursache vier Typen von infizierten Werten.

Tabelle 3.1: Typen von infizierten Werten.

Typ	Ursache	
Ia	Defekt	fehlerhafte Anweisung(en)
Ib		fehlende Anweisung(en)
IIa	Abhängigkeit von infiziertem Wert	direkt (Datenfluss)
IIb		indirekt (Kontrollfluss)

Eine Infektion vom Typ I liegt vor, wenn der infizierte Wert *unmittelbar* durch einen Defekt im Programm verursacht wurde. Bei einer Infektion von Typ I wird zwischen zwei Untertypen differenziert:

- Ist der Defekt, welcher die Infektion unmittelbar verursacht hat, eine *fehlerhafte* Anweisung, so ist die Infektion vom Typ Ia.
- Eine Infektion vom Typ Ib liegt vor, wenn der verursachende Defekt eine *fehlende* Anweisung im Quelltext ist.

Infektionen vom Typ II zeichnen sich dadurch aus, dass sie nicht unmittelbar, sondern *mittelbar* durch einen Defekt verursacht werden. Dies bedeutet, dass die Anweisungen zur Berechnung eines infizierten Wertes w korrekt

sind, jedoch mindestens ein Wert, von dem w abhängig ist, infiziert ist. Infektionen vom Typ II werden somit durch einen infizierten Wert verursacht, der sich durch die Zustände des Programms fortpflanzt und dabei andere Werte infiziert. Bei einer Infektion vom Typ II wird ebenfalls zwischen zwei Untertypen differenziert:

- Der Typ IIa liegt vor, wenn die Infektion durch eine *direkte*, d. h. durch den Datenfluss des Programms bedingte, Abhängigkeit von einem infizierten Wert verursacht wurde. Eine Infektion vom Typ IIa kann zum Beispiel durch eine beliebige arithmetische Operation verursacht werden. Wenn die Operation semantisch korrekt ist, jedoch einer der Operanden infiziert ist, so wird in der Regel auch das Ergebnis der Operation infiziert sein. Da die Infektion in diesem Fall durch die direkte Abhängigkeit von einem bereits infizierten Wert verursacht wurde, ist sie vom Typ IIa.

- Infektionen vom Typ IIb werden durch eine *indirekte*, durch den Kontrollfluss des Programms bedingte Abhängigkeit von einem infizierten Wert verursacht. Voraussetzung für diese Art der Infektion ist eine Anweisung a, die eine Verzweigung im Kontrollflussgraphen bewirkt. Auch wenn alle Basisblöcke, die durch die Verzweigung erreicht werden können fehlerfrei sind, kann die Ausführung von a eine Infektion verursachen. Dies ist in der Regel dann der Fall, wenn der Wert, auf dessen Grundlage einer der Zweige des Kontrollflussgraphen ausgeführt wird, infiziert ist. Eine Infektion vom Typ IIb kann zum Beispiel durch eine Schleife mit semantisch korrektem Rumpf aber einer fehlerhaften Abbruchbedingung verursacht werden.

Tabelle 3.2: Programmablauf der defekten Methode sumUp aus Listing 3.1.

Schritt	Variablen					Anweisung
	from	to	result	k_1:=from<to	return	
1	1	2				sumUp(1,2)
2			0			result = 0
3				true		while (from<to)
4			1			result += from
5	2					from += 1
6				false (Ia)		while (from<to)
7			(IIb)		1 (IIa)	return result

Listing 3.1: `sumUp` Methode mit einem Defekt in Zeile 3.

```
1 int sumUp(int from, int to) {
2   int result = 0;
3   while (from < to) {  \\ korrekt: while (from <=to) {
4     result += from;
5     from += 1;
6   }
7   return result;
8 }
```

Die Entstehung der unterschiedlichen Infektionstypen lässt sich anhand der Ausführung der fehlerhaften Implementierungen der Methode `sumUp` in den Listings 3.1 und 3.2 verdeutlichen. Die Methode `sumUp` soll für die beiden Argumente `from` und `to` die Summe der ganzen Zahlen, welche im Intervall [`from`, `to`] liegen, zurückgeben.

Die Implementierung in Listing 3.1 enthält einen Defekt in der Abbruchbedingung für die while-Schleife in Zeile 3. Der Programmablauf des Aufrufs dieser defekten Methode ist in der Tabelle 3.2 dargestellt. Die Tabelle zeigt für jede ausgeführte Anweisung die Wertänderungen aller Variablen der Methode `sumUp` an. Ist ein Eintrag einer Variablen leer, so hat sich ihr Wert durch die entsprechende Anweisung nicht geändert. Für infizierte Werte ist der Typ der Infektion in Klammern angegeben. Der Defekt, d. h. die fehlerhafte Bedingung der while-Schleife, erzeugt im Schritt 6 eine Infektion vom Typ Ia, da der Wert der Kontrollflussvariablen k_1 „true" sein müsste. Durch den verfrühten Abbruch der Schleife wird im 7. Schritt die Variable `result` infiziert, da `result` nach Beendigung aller Schleifendurchläufe den Wert 3 haben muss. Der Typ der Infektion ist IIb, da die Infektion durch die indirekte Abhängigkeit von der Kontrollflussvariablen k_1 verursacht wurde. Durch die Infektion von `result` wird auch die implizite Ergebnisvariable `return` infiziert. Der Typ der Infektion ist hier IIa, da `return` direkt, d. h. über den Datenfluss, von `result` abhängig ist.

Die Implementierung in Listing 3.2 enthält einen Defekt in Zeile 4. An dieser Stelle fehlt die Berechnung des Wertes der Variablen `result`. Der Programmablauf des Aufrufs dieser Methode ist in der Tabelle 3.3 dargestellt. Im 4. Ausführungsschritt verursacht der Defekt erstmalig eine Infektion der Variablen `result`, da nach vollständigem Durchlauf des Schleifenrumpfs

das Zwischenergebnis der Intervallsumme nicht der Variablen `result` zugewiesen wurde. Weil die Infektion durch eine fehlende Anweisung verursacht wurde, ist sie vom Typ Ib. Beim nachfolgenden Schleifendurchlauf wird die Variable `result` erneut durch die fehlende Anweisung infiziert. Nachdem der Schleifendurchlauf beendet ist, verursacht die Infektion von `result` eine Infektion der impliziten Ergebnisvariablen `return`. Der Typ der Infektion ist genau wie im vorherigen Beispiel IIa, da `return` auch hier direkt, d. h. über den Datenfluss, von `result` abhängig ist.

Listing 3.2: `sumUp` Methode mit einem Defekt in Zeile 4.

```
1 int sumUp(int from, int to) {
2   int result = 0;
3   while (from < to) {
4     // fehlende Anweisung: result += from;
5     from += 1;
6   }
7   return result;
8 }
```

Am Beispiel des Programmablaufs der beiden defekten Implementierungen der Methode `sumUp` wird deutlich, wie sich eine durch einen Defekt verursachte Infektion durch die nachfolgenden Zustände des Programms

Tabelle 3.3: Programmablauf der defekten Methode `sumUp` aus Listing 3.2.

Schritt	Variablen					Anweisung
	from	to	result	k_1:=from<=to	return	
1	1	2				sumUp(1,2)
2			0			result = 0
3				true		while (from<=to)
4	2		(Ib)			from += 1
5				true		while (from<=to)
6	3		(Ib)			from += 1
7				false		while (from<=to)
8					1 (IIa)	return result

fortpflanzt. Die initiale, d. h. durch einen Defekt verursachte, Infektion ist immer vom Typ I. Darüber hinaus kann jeder infizierte Wert weitere infizierte Werte vom Typ II zur Folge haben. Ein Defekt führt normalerweise nicht unmittelbar zu einer Fehlerwirkung, sondern erzeugt in der Regel einen gerichteten, azyklischen Graphen von infizierten Werten. Der Startknoten dieses Graphen ist vom Typ I und alle weiteren Knoten vom Typ II.

Darüber hinaus lässt sich an den Beispielen erkennen, dass der Aufwand zur Ermittlung der Ursache einer Infektion von der Art der Infektion abhängig ist. In der Regel ist die Ursache der Infektion bei den Typen Ia und IIa relativ leicht zu ermitteln, da diese sich aus der Anweisung, die zur Infektion geführt hat, ergeben. Zum Beispiel ist in Tabelle 3.2 die Infektion der Kontrollflussvariablen k_1=`from<to` vom Typ Ia. Die Ursache der Infektion ergibt sich hier unmittelbar aus der ausgeführten Anweisung, die „`while (from<=to)`“ hätte lauten müssen. Ähnlich verhält es sich bei Infektionen vom Typ IIa. Hier müssen die Operanden, von denen der infizierte Wert abhängt, auf Infektionen überprüft werden. Die Operanden ergeben sich in der Regel ebenfalls unmittelbar aus der ausgeführten Anweisung. Beispielsweise verursacht in Tabelle 3.2 im 7. Schritt die Anweisung **`return`** `result` eine Infektion des Ergebnisses des Methodenaufrufs. Das Ergebnis ist hier nur von der Variablen `result` abhängig. Diese ist infiziert und verursacht somit eine Infektion vom Typ IIa.

Die Ursachen der Infektionen vom Typ Ib und IIb sind oft sehr viel schwieriger zu identifizieren, da sie sich nicht unmittelbar aus dem Quelltext des Programms ableiten lassen. Bei Infektionen vom Typ Ib muss die Frage beantwortet werden, an welcher Stelle im Programmcode eine Anweisung fehlt. Die Variable `result` in Tabelle 3.3 ist zum Beispiel zu Beginn der Ausführung der Methode `sumUp` valide und am Ende infiziert, da sich ihr Wert nicht geändert hat und daher nicht dem erwarteten Ergebnis entspricht. Um den Zeitpunkt der Infektion zu ermitteln, muss die Position in der Ausführung bestimmt werden, an der ein neuer Wert für `result` *hätte* berechnet werden müssen. Für dieses kurze Beispiel ist die Suche nach der Position einfach, bei komplexeren Programmausführungen kann sie jedoch sehr mühsam und zeitaufwendig sein.

Bei einer Infektion vom Typ IIb ist die Ursache ein falscher Kontrollfluss in der Programmausführung. In der Tabelle 3.3 ist die Variable `result` am

Ende der Ausführung der Methode `sumUp` infiziert. Diesmal ist die Ursache jedoch nicht eine fehlende Anweisung, sondern die Tatsache, dass der Schleifenrumpf nur einmal ausgeführt wurde. Um die Ursache der Infektion zu finden, muss die infizierte Kontrollflussvariable gefunden werden, die dafür gesorgt hat, dass die Anweisung nicht ausgeführt wurde. In diesem Beispiel ist dies k_1. Je komplexer die Programmausführungen, desto komplexer ist auch die Bestimmung sämtlicher Kontrollflussvariablen, von denen ein Wert abhängt. Da diese Abhängigkeiten sich nicht unmittelbar aus dem Quelltext ergeben und mitunter schwer zu entdecken sind, gibt es Ansätze, wie das dynamische Slicing [3], mit deren Hilfe solche Abhängigkeiten im Quelltext automatisiert identifiziert werden können.

3.3 Debugging von Programmen

In der Softwareentwicklung wird der Debugging-Prozess angestoßen, nachdem in der Testphase eine Fehlerwirkung entdeckt wurde. Das Debuggen von Software ist ein zweistufiger Prozess, der die genaue Bestimmung der Ursache und der Herkunft eines Programmfehlers sowie dessen Beseitigung umfasst [111, S. 157]. Im Fokus dieser Arbeit liegt die erste Stufe des Debugging-Prozesses, das Schließen von der Fehlerwirkung auf den verursachenden Defekt. Im Folgenden wird daher unter Debugging ausschließlich die erste Phase des Debugging-Prozesses verstanden. Bezogen auf den in Abbildung 3.1 dargestellten Programmablauf ist somit das Ziel des Debugging, ausgehend von der Fehlerwirkung v_7^6 über die infizierten Zustände z_6–z_4 auf den initial infizierten Zustand z_3 und somit die defekte Anweisung s_3 zu schließen.

Es lassen sich zwei grundsätzlich unterschiedliche Herangehensweisen zum Schließen von der Fehlerwirkung auf den verursachenden Defekt unterscheiden: statische und dynamische Verfahren. Die *statischen* Verfahren verfolgen den Ansatz, vom *abstrakten* Quelltext des Programms auf die *konkrete* Ausführung und damit die Fehlerwirkung zu schließen. Im Wesentlichen versuchen diese *deduktiven* Verfahren die Menge von Anweisungen, die für die beobachtete Fehlerwirkung verantwortlich sein können, schrittweise einzugrenzen. Wichtig hierbei ist, dass es sich bei diesen Verfahren um eine statische Analyse des Quelltextes handelt, bei der das untersuchte Programm nicht ausgeführt wird. Die bekanntesten Techniken aus diesem Bereich sind

die sogenannten statischen Slicing-Verfahren [157]. Beim Programm-Slicing werden auf der Grundlage einer Kontroll- und Datenflussanalyse des Quelltextes Kontroll- und Datenabhängigkeiten zwischen den Anweisungen des Programms identifiziert. Für jede Anweisung A des Programms können basierend auf diesen Abhängigkeiten zwei Teilmengen der Programmanweisungen bestimmt werden. Als *Forward-Slice* von A bezeichnet man die Menge aller Anweisungen, deren Verhalten von A beeinflusst werden kann. Entsprechend ist ein *Backward-Slice* die Menge aller Anweisungen, deren Ausführung die Ausführung von A beeinflussen kann. Mithilfe des Slicing lassen sich somit Teilmengen des Programms bestimmen, die für eine Fehlerwirkung verantwortlich sein können. Auf diese Weise kann der Suchraum für den verursachenden Defekt eingeschränkt werden.

Ein weiteres Verfahren aus dem Bereich der statischen Analyse ist die Suche nach häufig auftretenden Fehlermustern. Zu diesen Fehlermustern zählen beispielsweise das Lesen nicht initialisierter Variablen, Variablen, deren Wert niemals ausgelesen wird, und durch den Kontrollfluss nicht erreichbare Anweisungen des Programms. Neben Hinweisen auf mögliche Fehlerursachen im Rahmen des Debugging dient die Identifizierung von Fehlermustern vor allem dazu, die Entstehung von Programmdefekten schon während der Programmentwicklung zu vermeiden. In einigen Entwicklungsumgebungen, wie zum Beispiel der Eclipse IDE [48], ist die automatische Suche nach solchen Fehlermustern bereits integriert.

Obwohl die statische Analyse einige interessante Ansätze zum Debuggen von Programmen bietet, sind ihr aus theoretischer Sicht Grenzen gesetzt [90]. Da der Programmcode nur statisch analysiert wird, ist es unvermeidbar, dass die statische Analyse sogenannte *falsche Positive* erzeugt. Im Bezug auf die Slices bedeutet dies zum Beispiel, dass in den Forward- bzw. Backward-Slices für eine Anweisung A Anweisungen enthalten sein können, die aufgrund der Struktur des Programms bei dessen Ausführung niemals von A beeinflusst werden bzw. niemals einen Einfluss auf A haben können. Genauso muss natürlich nicht jedes Fehlermuster einen Defekt enthalten. Die Ursache für diese Ungenauigkeiten resultiert aus Sprachkonstrukten, die eine statische Berechnung von exakten Kontroll- oder Datenflüssen unmöglich machen. Daher ist es möglich, dass ein statisch ermittelter Kontroll- oder Datenfluss durch keine konkrete Programmausführung überdeckt werden kann.

Den statischen Analysemethoden steht die Klasse der dynamischen Analysemethoden gegenüber [102, 137]. Bei der dynamischen Analyse werden Fakten über die Ausführung des zu untersuchenden Programms gesammelt. Mithilfe der gesammelten Fakten kann die Ausführung des untersuchten Programms nachvollzogen werden. Die gesammelten Informationen über den tatsächlichen Programmablauf sollen schließlich Aufschluss über den Fehler verursachenden Defekt geben. Im Gegensatz zu den statischen Verfahren wird hier also nicht untersucht, was passiert sein *könnte*, sondern was tatsächlich bei einem konkreten Programmlauf passiert *ist*. Eine einfache Methode zum Sammeln von Fakten über den Programmablauf ist der Einsatz von Logging-Mechanismen [46, S. 99–106]. Um bestimmte Fakten des Programmablaufs sichtbar zu machen, werden diese durch Logging-Anweisungen ausgegeben. Die Ausgabe kann dabei in einer Konsole, einer speziellen Datei oder Ähnlichem erfolgen. Die weite Verbreitung sowie die Vielzahl von dedizierten Logging-Frameworks, wie log4J [61] oder log4cxx [9], zeigen, dass Logging ein häufig verwendeter Ansatz ist, um den Ablauf von Programmen nachzuvollziehen.

Logging-Mechanismen sind in der Regel hilfreich, um die Position eines Defekts näher einzugrenzen; als alleiniges Mittel zum Debuggen von Programmen sind sie grundsätzlich jedoch nicht zu empfehlen. Dies liegt an der Tatsache, dass der Einsatz von Logging-Techniken zum Debuggen von Programmen mit einigen Nachteilen verbunden ist. Zunächst ist das Logging mit erhöhtem Aufwand für den Programmierer verbunden, da zusätzliche Logging-Anweisungen erzeugt und in den Programmcode integriert werden müssen. Ist die ungefähre Position eines Defekts mithilfe des Logging-Anweisungen eingegrenzt, so sind meist weitere Logging-Anweisungen nötig, um die Position des Defekts weiter einzugrenzen. Das Debugging mithilfe von Logging ist daher häufig ein iterativer Prozess, bei dem auf jeder Stufe versucht wird, die Position des Defekts durch Hinzufügen zusätzlicher Logging-Anweisungen weiter einzugrenzen. Hierfür muss das Programm ständig neu kompiliert und ausgeführt werden, wodurch dieses Debugging-Verfahren sehr zeit- und arbeitsaufwendig ist.

Eine weitere Möglichkeit, Fakten über den Ablauf eines Programms zu erhalten, besteht im Einsatz von Debuggern. Ein *Debugger* ist ein Softwaretool zur Untersuchung und Überwachung des Programmablaufs, das verwendet wird, um Defekte in Programmen zu finden. Im Vergleich zum Logging hat

der Einsatz eines Debuggers den Vorteil, dass keine Änderungen am Quelltext vorgenommen werden müssen, um den Ablauf zu überwachen. Mithilfe eines Debuggers kann ein zu untersuchendes Programm direkt gestartet und observiert werden. Darüber hinaus bieten Debugger in der Regel weitere Funktionalitäten, die die Suche nach Defekten zusätzlich erleichtern bzw. beschleunigen. So unterstützen zum Beispiel viele Debugger die Definition von Haltepunkten, die die Programmausführung unterbrechen, sofern eine festgelegte Bedingung erfüllt ist, heben Änderungen im Zustandsraum des Programms farblich hervor und ermöglichen es, die schrittweise Ausführung eines Methodenaufrufs zu überspringen.

3.4 Fazit

In diesem Kapitel wurden die für diese Arbeit grundlegenden Begriffe des Debugging erläutert. Ferner wurden unterschiedliche Infektionstypen identifiziert. Im folgenden Kapitel wird sich zeigen, dass diese Infektionstypen die Komplexität der Defektsuche und die Effizienz der Debugging-Techniken beeinflussen. Zusätzlich wurden der grundsätzliche Ablauf und die Zielsetzung des Debugging-Prozesses beschrieben. Die nachfolgend beschriebenen Debugging-Techniken verfolgen unterschiedliche Ansätze um die Zielsetzung des Debugging zu erfüllen.

Kapitel 4

Debugging-Techniken

Inhalt

4.1 **Trace-Debugging** 50
4.2 **Omniscient-Debugging** 54
4.3 **Deklaratives Debugging** 57
4.4 **Hybrides Debugging** 76
4.5 **Fazit** 80

In diesem Kapitel werden zunächst drei Techniken vorgestellt, mit denen ein Programmablauf nach einem Defekt durchsucht werden kann. Die am einfachsten umzusetzende und am weitesten verbreitete dieser Techniken ist das Trace-Debugging (4.1). Die beiden übrigen Methoden, das Omniscient-Debugging (4.2) und das deklarative Debugging (4.3), sind aufwendiger zu realisieren, da sie im Gegensatz zum Trace-Debugging den Ablauf des untersuchten Programms aufzeichnen. Mithilfe der zusätzlich bereitgestellten Informationen über den Programmablauf können diese beiden Methoden die Suche nach einem Defekt erheblich vereinfachen und beschleunigen.

Die Eignung von Omniscient-Debugging und deklarativem Debugging zur Defektsuche wird durch die Art des Defekts und die verursachten Infektionen beeinflusst. Dabei ist keine der beiden Techniken der anderen in allen Fällen überlegen. Aus diesem Grund wird im Abschnitt 4.4 eine hybride Debugging-Technik entwickelt, die eine Kombination von Omniscient-Debugging und deklarativem Debugging darstellt und die Vorteile der beiden Techniken miteinander verbindet. Den Abschluss dieses Kapitels bildet die Zusammenfassung der Ergebnisse in Abschnitt 4.5.

Die Debugging-Techniken können grundsätzlich auf verschiedene Programmiersprachen oder Programmierparadigmen angewendet werden. Aus diesem Grund sind die folgenden Untersuchungen und Bewertungen der Techniken allgemein gehalten und abstrahieren von den konkreten Umsetzungen für eine bestimmte Programmiersprache.

4.1 Trace-Debugging

Vor allem für imperative und objektorientierte Sprachen ist das Trace-Debugging bis heute die verbreitetste und am häufigsten eingesetzte Debugging-Technik. Debugger für populäre objektorientierte Sprachen, wie Java [89], C# [158] oder C++[99], basieren nahezu ausschließlich auf dieser Technik.

Die grundlegende Idee wird in der Abbildung 4.1 dargestellt. Beim Trace-Debugging können die Anweisungen des observierten Programms schrittweise ausgeführt werden. Nach jedem Schritt zeigt der Debugger den aktuellen Zustand des Programms an. In der Abbildung wird dies durch ein Fenster verdeutlicht, welches mit jedem Ausführungsschritt, d. h. mit jeder ausgeführten Anweisung, um eine Zeile nach unten über den Zustandsraum verschoben wird. Die Werte innerhalb des Fensters stellen den gegenwärtig betrachteten Zustandsraum dar. Die Ausführung des untersuchten Programms kann ausschließlich entlang der normalen Ausführungsrichtung verfolgt werden. Aus diesem Grund können Zustände, die zeitlich vor dem gegenwärtig untersuchten Zustand liegen, nicht wiederhergestellt werden. Anschaulich gesprochen kann das Suchfenster also nur vorwärts, d. h. in Richtung der Programmausführung, über dem Zustandsraum verschoben werden.

Im Bezug auf den Debugging-Prozess bedeutet dies, dass der Benutzer eines Trace-Debuggers ausgehend von einem gewählten Start- bzw. Haltepunkt das Programm schrittweise ausführt und dabei jeden neuen Programmzustand prüft. Der gesuchte Defekt ist genau dann gefunden, wenn der Zustand des observierten Programms erstmalig von einem validen in einen infizierten Zustand überführt wird. Die Anweisung, welche die entsprechende Zustandsänderung bewirkt hat, ist defekt und hat die beobachtete Fehlerwirkung verursacht.

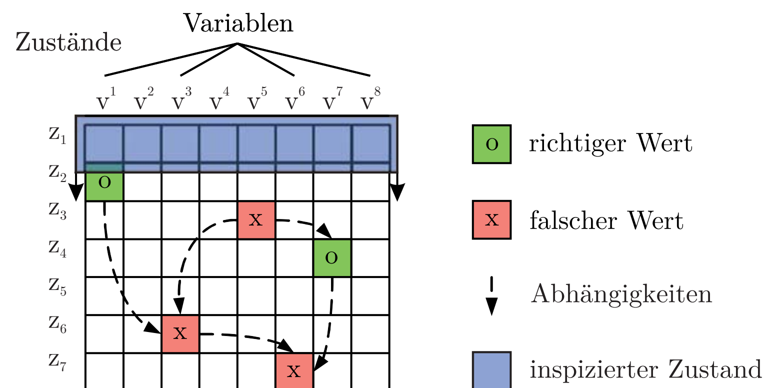


Abbildung 4.1: Der Trace-Debugging-Prozess. Die Programmausführung wird schrittweise nachvollzogen. Durch sequenzielle Inspektion der Zustände muss diejenige Anweisung gefunden werden, die das Programm erstmalig von einen validen in einen infizierten Zustand überführt.

Ein Trace-Debugger lässt sich mit vergleichsweise geringem Aufwand realisieren. Er muss die Ausführung des observierten Programms unterbrechen und schrittweise ausführen können. Darüber hinaus muss er dem Benutzer die Möglichkeit bieten, den Zustandsraum des unterbrochenen Programms zu inspizieren. Im Vergleich zur normalen Ausführung eines Programms ist der zusätzliche Ressourcenverbrauch für das Trace-Debugging relativ gering. Abgesehen vom zusätzlichen Aufwand für das Überwachen der Haltepunkte kann das observierte Programm normal ausgeführt werden. Während der Ausführung müssen keine weiteren zusätzlichen Berechnungen durchgeführt oder Daten aufgezeichnet werden. Zusätzlicher Aufwand für die Untersuchung des Programmzustands entsteht nur bei Bedarf, d. h. ausschließlich für die tatsächlich vom Benutzer inspizierten Teile des Zustandsraums. Darüber hinaus ist das Trace-Debugging für jeden Programmierer einfach zugänglich und intuitiv verständlich, da der Debugging-Prozess dem Ablauf des Programms bei der Ausführung folgt. Die leichte Realisierbarkeit, der geringe zusätzliche Ressourcenbedarf und die Einfachheit der Methode sind wichtige Gründe für die weite Verbreitung und die hohe Akzeptanz von Trace-Debuggern.

Dennoch hat das Trace-Debugging einige entscheidende Nachteile. Mithilfe eines Trace-Debuggers können Programme nur vorwärts ausgeführt werden. Tatsächlich ist der natürliche Prozess des Schließens von der Fehlerwirkung

über die infizierten Zustände auf den Defekt jedoch rückwärts gerichtet. Beim Trace-Debugging inspiziert der Benutzer hauptsächlich Zustände, die vor der Ausführung des gesuchten Defekts liegen. Diese Zustände sind valide und liefern daher in der Regel keinerlei Hinweise auf den Defekt. Es ist deshalb nur sehr schwer und ungenau abzuschätzen, wie weit ein valider Zustand von der Ausführung des Defekts entfernt ist. Anders verhält sich dies bei einem infizierten Zustand. Ein infizierter Zustand hat einen oder mehrere infizierte Werte, die einen Hinweis auf den Ursprung der Infektion geben. Ließe sich ein Programm auch rückwärts ausführen, so könnte, ausgehend von der Fehlerwirkung, über die Folge der infizierten Zustände, der Defekt sehr viel schneller zurückverfolgt werden. Darüber hinaus betrachtet das Trace-Debugging immer nur den gegenwärtigen Zustand des überwachten Programms. Dem Benutzer liegen keine Informationen über vergangene oder zukünftige Zustände des Programms vor. Diese würden die Suche nach dem Defekt deutlich erleichtern und beschleunigen.

Aufgrund dieser Defizite kann das Trace-Debugging mitunter sehr mühsam und zeitaufwendig sein. Ausgehend von einem Haltepunkt muss der Benutzer die einzelnen Ausführungsschritte des Programms nachvollziehen, bis der erste infizierte Zustand und damit der Defekt erreicht ist. Da zu Beginn des Trace-Debugging die aus der Fehlerwirkung ableitbaren Informationen über den verursachenden Defekt häufig nur sehr vage und ungenau sind, ist die Wahl eines geeigneten Startpunkts sehr schwierig. Es besteht die Gefahr, entweder einen Punkt zu wählen, der im Programmablauf sehr weit vor oder sogar hinter der Ausführung des Defekts liegt. Im ersten Fall müssen sehr viele valide Zustände untersucht werden, bis der Defekt gefunden wird. Im zweiten Fall ist der Zustand zu Beginn des Trace-Debugging bereits infiziert und der Vorgang muss mit einem früheren Startpunkt erneut gestartet werden.

Des Weiteren hat diese Debugging-Methode einen relativ geringen Abstraktionsgrad. Die Ausführung des untersuchten Programms muss schrittweise nachvollzogen und dabei die Korrektheit einzelner Anweisungen überprüft werden. Während des gesamten Debugging-Prozesses muss der Benutzer die Ausführung des untersuchten Programms somit auf der Implementierungsebene nachvollziehen und bewerten. Dieser niedrige Abstraktionsgrad ist durchaus erforderlich, um eine einzelne defekte Anweisung zu lokalisieren. Aber gerade zu Beginn der Defektsuche, wenn der Suchraum, d. h. die

möglichen Positionen des Defekts im Programmablauf, noch relativ groß ist, führt dieser niedrige Abstraktionsgrad zu einem hohen Aufwand. Das Problem ist, dass der Suchraum mit jedem Debugging-Schritt nur um eine einzige Anweisung verkleinert wird. Der Nachteil des geringen Abstraktionsgrades zeigt sich zum Beispiel am Aufruf einer Methode zum Sortieren eines Arrays. Mit einem Trace-Debugger kann entweder jeder einzelne Ausführungsschritt dieses Methodenaufrufs durchlaufen oder die schrittweise Ausführung des Aufrufs übersprungen werden. Für den Fall, dass der Methodenaufruf infiziert ist, d. h. nach dem Aufruf befindet sich das Programm in einem infizierten Zustand, kann der Defekt nur gefunden werden, wenn der Aufruf schrittweise ausgeführt wird. Wird die schrittweise Ausführung des Methodenaufrufs übersprungen, wird die Ausführung des Defekts verpasst und der Debugging-Prozess muss neu gestartet werden. Für den Fall, dass der Methodenaufruf nicht infiziert ist, kann der Aufwand durch das Überspringen des Aufrufs erheblich reduziert werden. Die schrittweise Ausführung würde in diesem Fall erheblichen zusätzlichen Aufwand bedeuten. Das Problem ist jedoch, dass vor der Ausführung eines Methodenaufrufs häufig nicht bekannt ist, ob dieser infiziert oder valide sein wird, da das Ergebnis des Aufrufs zu diesem Zeitpunkt noch unbekannt ist.

Beim Trace-Debugging wird der Aufwand hauptsächlich durch die Anzahl und die Komplexität der Programmzustände, die bis zum Auffinden des Defekts untersucht werden müssen, bestimmt. Die Anzahl der zu untersuchenden Zustände wird maßgeblich durch die Position des Startpunktes für das Trace-Debugging und die Position des Defekts in der Programmausführung bestimmt. Je geringer der Abstand zwischen diesen beiden Punkten, desto geringer der entstehende Suchaufwand. Für die Wahl eines geeigneten Startpunktes ist es entscheidend, wie genau die Position des Defekts bekannt ist. Zu Beginn des Trace-Debugging-Prozesses ist ausschließlich die Fehlerwirkung des Defekts bekannt. Daraus folgt, dass der Suchaufwand umso geringer ausfällt, je genauer von der Fehlerwirkung auf die Position geschlossen werden kann. Je genauer die Position des Defekts bekannt ist, desto besser kann der Abstand zwischen Startpunkt und Defekt reduziert werden.

Die Genauigkeit, mit der auf die Position des Defekts geschlossen werden kann, wird entscheidend durch die Kenntnis des untersuchten Programms beeinflusst. Je besser das Verhalten eines Programms während der Ausfüh-

rung bekannt ist, desto exakter wird es in der Regel möglich sein, die Ursache einer Fehlerwirkung zu bestimmen. Darüber hinaus ist aber auch die Art der Fehlerwirkung ein entscheidender Einflussfaktor. Die Position eines Defekts, der unmittelbar zu einer Fehlerwirkung führt, ist gewöhnlich leichter zu bestimmen als die Position eines Defekts, der erst nach einer langen Kette von infizierten Zuständen zu einer Fehlerwirkung führt. Zudem können auch die in Abschnitt 3.2 beschriebenen Infektionstypen die Genauigkeit der Lokalisierung des Defekts beeinflussen. Infektionen vom Typ Ia und IIa sind in der Regel leichter zurück zu verfolgen als die Typen Ib und IIb. Im Gegensatz zu dem im nachfolgenden Abschnitt beschriebenen Omniscient-Debugging ist der Einfluss der Infektionstypen auf den Aufwand des Debugging-Prozesses jedoch wesentlich geringer.

4.2 Omniscient-Debugging

Um die beschriebenen Nachteile des Trace-Debugging zu überwinden und dem Benutzer komfortablere und effizientere Funktionen zur Suche des Defekts bereitzustellen, benötigt ein Debugger zusätzliche Informationen über den Programmablauf. Wie im Abschnitt 4.1 beschrieben, wird beim Trace-Debugging das fehlerhafte Programm während der Ausführung unterbrochen, um den Zustand des Programms zum Zeitpunkt der Unterbrechung zu untersuchen. Somit steht für das Debugging immer nur der gegenwärtige Zustand des Programms zur Verfügung. Beim Omniscient-Debugging wird im Gegensatz zum Trace-Debugging das fehlerhafte Programm nicht während der Ausführung unterbrochen, sondern zunächst vollständig ausgeführt. Während dessen wird der Programmablauf mitsamt des Kontrollflusses und aller Zustandsänderungen aufgezeichnet. Der eigentliche Debugging-Prozess startet, nachdem der Programmablauf aufgezeichnet wurde. Der Ansatz des Omniscient-Debugging ist bereits durch einige Debugger umgesetzt worden [168, S. 244–246]. Die erste Umsetzung für die Programmiersprache Java ist der von Lewis entwickelte *Omniscient DeBugger* (ODB) [95].

Da beim Omniscient-Debugging der vollständige Programmablauf vorliegt, ist es möglich, den Programmzustand zu beliebigen Punkten des Programmablaufs zu rekonstruieren. Wie in der Abbildung 4.2 dargestellt, kann der

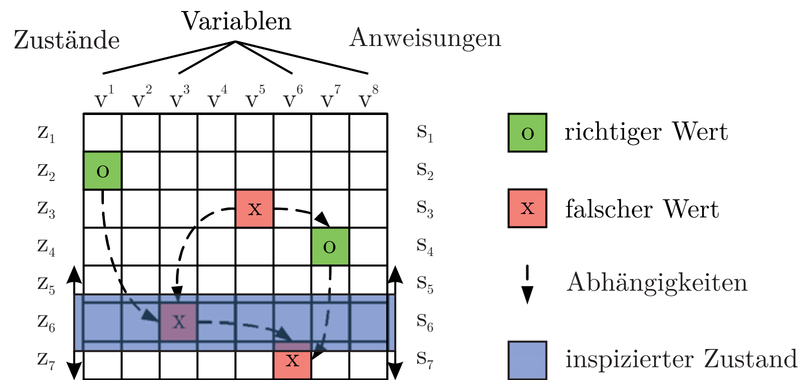


Abbildung 4.2: Der Omniscient-Debugging-Prozess.

Zustandsraum des Programms in beliebiger Richtung durchlaufen werden. Zudem ist es möglich, zu beliebigen Punkten im Programmablauf bzw. im Zustandsraum zu springen.

Für den Debugging-Prozess ergeben sich damit im Vergleich zum Trace-Debugging einige entscheidende Vorteile. Beim Omniscient-Debugging ist es möglich, die Ausführung eines Programms rückwärts, d. h. entgegengesetzt zur Ausführungsrichtung, zu verfolgen. Diese Eigenschaft erlaubt es, den Debugging-Prozess an der Stelle im Programmablauf zu starten, an der die Fehlerwirkung auftritt. Ausgehend von der Fehlerwirkung kann der Programmablauf über die infizierten Zustände bis hin zum Defekt zurückverfolgt werden.

Das rückwärts gerichtete Schließen von der Fehlerwirkung erleichtert die Suche nach dem Defekt im Vergleich zur vorwärts gerichteten Suche des Trace-Debugging erheblich. Für den in der Abbildung 4.2 dargestellten Programmablauf würde die Suche nach dem Defekt beim infizierten Wert der 6. Variablen zum Zeitpunkt 7, v_7^6, beginnen. Der infizierte Wert, welcher der Variablen v^6 durch die Anweisung s^7 zugewiesen wird, ist abhängig von den Werten v_6^3 und v_4^7. Der Wert v_7^6 könnte zum Beispiel durch die Addition der Werte v_6^3 und v_4^7 zustande gekommen sein. Da die Anweisung s^7 korrekt ist, jedoch ein fehlerhaftes Ergebnis liefert, muss die Ursache des Fehlers in einem der beiden Werte liegen, von denen v_7^6 abhängig ist. Da beim Omniscient-Debugging für jede Variable die vollständige Historie ihrer Werte vorliegt, können diese Werte direkt geprüft werden. Die Prüfung ergibt, dass v_6^3 infiziert und v_4^7 valide ist. Die Suche wird daher nach dem gleichen Schema für den Wert v_6^3 fortgesetzt. Durch die Rückverfolgung der

infizierten Werte und deren Abhängigkeiten wird schließlich der infizierte Wert v_3^5 gefunden. Die Infektion dieses Wertes resultiert nicht aus einem anderen infizierten Wert, sondern aus der defekten Anweisung s_3.

Dieses Beispiel zeigt, wie mithilfe des Omniscient-Debugging der verursachende Defekt zu einer Fehlerwirkung gefunden werden kann. Dabei können die Abhängigkeiten der infizierten Werte genutzt werden, um bei der Suche Teile des infizierten Zustandsraumes zu überspringen und den gesuchten Defekt schneller zu finden.

Im Vergleich zum Trace-Debugging bietet das Omniscient-Debugging einige entscheidende Vorteile, die das Debugging erheblich erleichtern. Diese Erleichterungen für den Benutzer werden ermöglicht durch die zusätzlichen Informationen, die über die Ausführung des untersuchten Programms gesammelt werden. Der zusätzliche Zeit- und Speicherplatzbedarf zur Aufzeichnung des gesamten Programmablaufs ist nicht unerheblich. Für ODB verringert sich die Ausführungszeit beispielsweise um den Faktor 10–300 und der Speicherplatzbedarf liegt bei ungefähr 100 MB pro Sekunde [95].

Die Effizienz des Omniscient-Debugging wird maßgeblich durch die in Abschnitt 3.2 beschriebenen Infektionstypen beeinflusst. Die entscheidende Verbesserung gegenüber dem Trace-Debugging wird durch die Möglichkeit erzielt, eine Fehlerwirkung zum verursachenden Defekt zurückverfolgen zu können. Der Infektionstyp (vgl. Abschnitt 3.2) hat allerdings entscheidenden Einfluss auf den benötigten Aufwand, mit dem die Ursache einer Infektion bestimmt werden kann. Aus diesem Grund ist das Omniscient-Debugging bei Fehlern vom Typ Ia und IIa am effizientesten. Die möglichen Ursachen lassen sich bei diesen Infektionen unmittelbar aus dem Quelltext bzw. der Anweisung, welche die Infektion verursacht hat, ermitteln. Um die Ursache einer Infektion zu bestimmen, müssen die möglichen Infektionsursachen überprüft werden, bis die tatsächliche Ursache gefunden ist. Da sich die Ursache einer Infektion vom Typ Ia oder IIa beim Omniscient-Debugging einfach ermitteln lässt, können auch Defekte, die erst nach einer langen Kette von infizierten Zuständen zu einer Fehlerwirkung führen, in der Regel schnell gefunden werden. Das Verfahren ist in diesem Fall um ein Vielfaches effizienter als das Trace-Debugging.

Bei Fehlern vom Typ Ib und IIb stößt allerdings auch das Omniscient-Debugging an seine Grenzen. Die Tatsache, dass die möglichen Ursachen

dieser Infektionstypen sehr viel schwieriger zu ermitteln sind, reduziert die Effizienz dieses Verfahrens deutlich. Der Benutzer kann bei diesen Infektionstypen die möglichen Ursachen nicht unmittelbar aus dem Quelltext des Programms ablesen. Daher ist bei diesen Infektionstypen die Suche nach der Ursache sehr viel komplexer und fehleranfälliger. Das Problem ist, dass es in diesem Fall kein einfaches Verfahren gibt, mit dem sich alle möglichen Ursachen einer Infektion schnell und effizient finden lassen. Der Benutzer ist somit darauf angewiesen, eine eigene Strategie aus seinem Wissen über die Fehlerwirkung sowie über die Struktur und den Ablauf des Programms abzuleiten. Dennoch ist das Omniscient-Debugging bei diesen Infektionstypen immer noch deutlich effizienter als das Trace-Debugging, da der Zustandsraum der Programmausführung mit den Mitteln des Omniscient-Debugging sehr viel effizienter durchsucht werden kann.

4.3 Deklaratives Debugging

Im Folgenden wird zunächst das deklarative Debugging-Verfahren erläutert. Im Anschluss wird untersucht, welche Konsequenzen sich ergeben, wenn das Verfahren auf Java-Programme angewendet wird.

4.3.1 Verfahren

Deklaratives oder algorithmisches Debugging ist eine Methode, die ursprünglich von Shapiro (1983) [130] für die logische Programmiersprache Prolog entwickelt wurde. Sie wurde später auf funktionale [118] und funktional-logische [29] Programmierparadigmen erweitert. Die grundlegende Idee des deklarativen Debugging ist die Teilautomatisierung des Debugging-Prozesses. Die Suche nach dem Fehler verursachenden Defekt wird dabei vom Debugger durch einen interaktiven Prozess gesteuert.

Die Basis des deklarativen Debugging ist ein *Berechnungsbaum*, der die Ausführung des zu untersuchenden Programms repräsentiert. Im Sinne der Graphentheorie ist ein Berechnungsbaum ein geordneter Baum, d. h. ein gerichteter, azyklischer Graph mit genau einem als Wurzel ausgezeichneten

Knoten, bei dem die Teilbäume jedes Knotens geordnet sind.[2] Ein Exemplar eines solchen Berechnungsbaums ist in der Abbildung 4.3 dargestellt. Ein Berechnungsbaum wird erzeugt, indem die Abhängigkeiten zwischen den Teilberechnungen eines Programmablaufs folgendermaßen in eine Baumstruktur überführt werden:

- Ein Knoten *k* des Berechnungsbaums repräsentiert eine Teilberechnung des ausgeführten Programms.
- Die Wurzel des Berechnungsbaums repräsentiert die Hauptberechnung.
- Ein Kindknoten eines Knotens *k* repräsentiert eine Teilberechnung, die während der Ausführung der durch *k* repräsentierten Berechnung durchgeführt wurde.
- Die Reihenfolge der Kindknoten eines Knotens *k* entspricht der Reihenfolge, in der die durch die Kindknoten repräsentierten Teilberechnungen ausgeführt wurden.

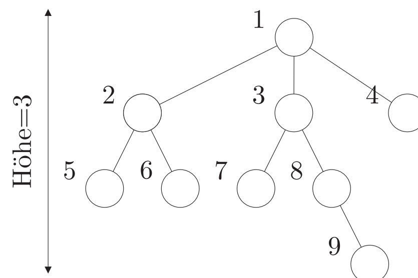


Abbildung 4.3: Repräsentation eines Programmablaufs durch einen Berechnungsbaum der Höhe 3. Die Wurzel (1) repräsentiert die Hauptberechnung. Teilberechnungen werden durch Kindknoten repräsentiert. Die Reihenfolge der Kindknoten entspricht der Reihenfolge, in der die repräsentierten Teilberechnungen ausgeführt wurden.

Während des deklarativen Debugging-Prozesses fordert der Debugger den Benutzer dazu auf, bestimmte Knoten des Berechnungsbaums zu klassifizieren. Dabei wird ein Knoten in eine der drei folgenden Klassen unterteilt:

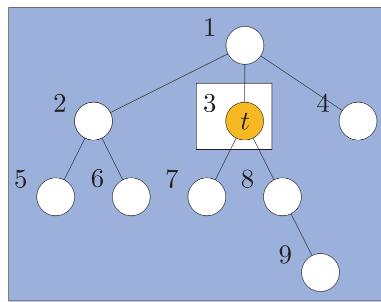
[2] Eine ausführlichere Darstellung gerichteter Bäume findet sich u. a. bei Cormen [37, S.1089–1091].

- **valide** - Ein Knoten ist valide, wenn die durch ihn repräsentierte Berechnung korrekt ist. Eine Berechnung ist genau dann korrekt, wenn das Ergebnis der Berechnung dem erwarteten Ergebnis entspricht.
- **infiziert** - Ein Knoten ist infiziert, wenn die durch ihn repräsentierte Berechnung fehlerhaft ist. Eine Berechnung ist genau dann fehlerhaft, wenn das Ergebnis der Berechnung *nicht* dem erwarteten Ergebnis entspricht.
- **vertrauenswürdig** - Wird ein Knoten als vertrauenswürdig klassifiziert, so bedeutet dies, dass der Quelltext, welcher der durch den Knoten repräsentierten Berechnung zugrunde liegt, keine Defekte enthält. Somit sind alle Knoten, die eine Berechnung repräsentieren, die auf der Ausführung des gleichen Teils des Quelltextes basieren, ebenfalls vertrauenswürdig.[3]

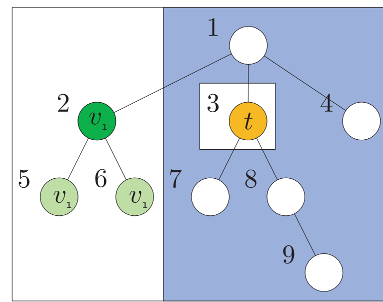
Das Ziel des deklarativen Debugging ist es, im Berechnungsbaum einen *defekten Knoten* zu finden. Ein defekter Knoten ist ein infizierter Knoten, der keine infizierten Kindknoten besitzt. Bei einem defekten Knoten ist somit keiner der Kindknoten, sondern der defekte Knoten selbst für die Infektion verantwortlich. Aus diesem Grund muss der Quelltext, welcher für die repräsentierte Berechnung ausgeführt wurde, einen Defekt enthalten.

In der Abbildung 4.4 ist dargestellt, wie die Klassifizierung der Knoten des Berechnungsbaums dazu verwendet wird, den Suchraum, in dem sich der defekte Knoten befinden kann, schrittweise einzuschränken. Die Abbildung 4.4a zeigt den Suchraum zu Beginn des Debugging-Prozesses. Zu diesem Zeitpunkt umfasst er den gesamten Berechnungsbaum mit Ausnahme der Knoten, die bereits als vertrauenswürdig klassifiziert wurden. Ein vertrauenswürdiger Knoten kann kein defekter Knoten sein, da der Programmcode, durch den dieser Knoten erzeugt wurde, keine Defekte enthält. Ein vertrauenswürdiger Knoten kann allerdings Kindknoten besitzen, die nicht vertrauenswürdig sind. Diese Kindknoten sind Teil des Suchraums.

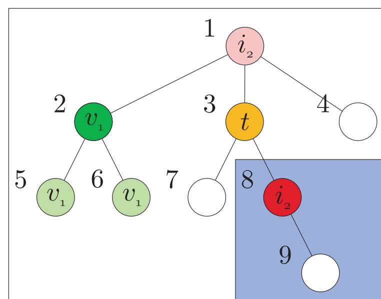
[3] Zur Durchführung des deklarativen Debugging ist die Klassifizierung der Knoten als valide oder infiziert ausreichend. Die Klasse der vertrauenswürdigen Knoten stellt eine Erweiterung dar, mit der sich der Klassifizierungsaufwand für den Benutzer reduzieren lässt, wenn bereits zu Beginn des Debugging-Prozesses bekannt ist, dass Teile des untersuchten Programms keinen Defekt enthalten.



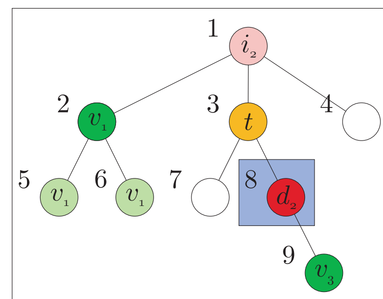
(a) Vor Beginn der Klassifizierung. Der vertrauenswürdige Knoten 3 ist nicht Teil des Suchraums.



(b) Nach Klassifizierung des Knotens 2 als *valide*.



(c) Nach Klassifizierung des Knotens 8 als *infiziert*.



(d) Nach Klassifizierung des Knotens 9 als *valide*.

Abbildung 4.4: Suche des defekten Knotens. Durch die Klassifizierung der Knoten des Berechnungsbaums wird der Suchraum für den defekten Knoten schrittweise verkleinert.

Wird, wie in der Abbildung 4.4b dargestellt, der Knoten 2 als valide klassifiziert, so bedeutet dies, dass die durch den Knoten repräsentierte Berechnung korrekt ist. Daraus folgt weiter, dass auch sämtliche Teilberechnungen, welche zum Ergebnis der korrekten Berechnung beigetragen haben korrekt sind. Aus diesem Grund sind alle Knoten des Teilbaumes, dessen Wurzel als valide klassifiziert wurde, ebenfalls valide. Da der gesuchte defekte Knoten nicht Element dieses Teilbaums sein kann, werden die Knoten des Teilbaums mit der Wurzel 2 aus dem Suchraum entfernt.

Entspricht das Ergebnis einer Berechnung nicht dem erwarteten Ergebnis, so wird der repräsentierende Knoten als infiziert klassifiziert. Für den Teilbaum, dessen Wurzel infiziert ist, gilt, dass einer der Knoten des Teilbaums der gesuchte defekte Knoten sein muss. Wie in der Abbildung 4.4c gezeigt,

führt ein Knoten, der als infiziert klassifiziert wird, dazu, dass der Suchraum auf den Teilbaum, dessen Wurzel der infizierte Knoten ist, beschränkt wird. In diesem Fall ist dies der Knoten 8.

Der deklarative Debugging-Prozess terminiert, wenn der Suchraum keine unklassifizierten Knoten mehr enthält. Wurde mindestens ein Knoten als infiziert klassifiziert, so enthält der Suchraum zum Schluss genau einen infizierten Knoten. Dieser Knoten kann aufgrund der Struktur des beschriebenen Verfahrens ausschließlich valide Kindknoten besitzen. Aus diesem Grund ist dieser Knoten der gesuchte defekte Knoten. Wurde während des gesamten Debugging-Prozesses kein einziger Knoten als infiziert klassifiziert, so endet die Suche schließlich mit einem leeren Suchraum. Dies bedeutet, dass die Wurzel des gesamten Berechnungsbaums valide ist und somit die Ausführung auch keine Fehlerwirkung produziert hat. Folgerichtig kann in diesem Fall auch kein defekter Knoten gefunden werden. In der Abbildung 4.4d endet der deklarative Debugging-Prozess, nachdem der Knoten 9 als valide klassifiziert wurde. Das Resultat der Suche ist die Identifikation des defekten Knotens 8.

Falls die untersuchte Programmausführung eine Fehlerwirkung und somit auch einen Defekt enthält, wird durch das deklarative Debugging der defekte Knoten in jedem Fall gefunden. Das Verfahren ist somit vollständig und korrekt. Dies gilt unter der Voraussetzung, dass die Klassifizierung der Knoten durch den Benutzer korrekt ist.

Ebenso wie das Trace-Debugging oder das Omniscient-Debugging, dient auch das deklarative Debugging dazu, die Position im Ablauf des untersuchten Programms zu identifizieren, an der der Programmzustand durch einen Defekt erstmalig infiziert wird, d. h. von einem validen in einen infizierten Zustand wechselt.

Im Vergleich zu den beiden zuvor beschriebenen Verfahren gibt es zwei entscheidende Unterschiede: das *höhere Abstraktionsniveau* und die *Teilautomatisierung* des Debugging-Prozesses.

Beim deklarativen Debugging werden keine einzelnen Anweisungen, sondern semantisch zusammenhängende Berechnungseinheiten auf ihre Validität überprüft. Für den Debugging-Prozess bedeutet dies, dass der Benutzer

sich von der konkreten Implementierung lösen kann. Um einen Defekt zu lokalisieren, müssen die Berechnungen nicht mehr schrittweise nachvollzogen werden, sondern die Ergebnisse von Teilberechnungen bewertet werden.

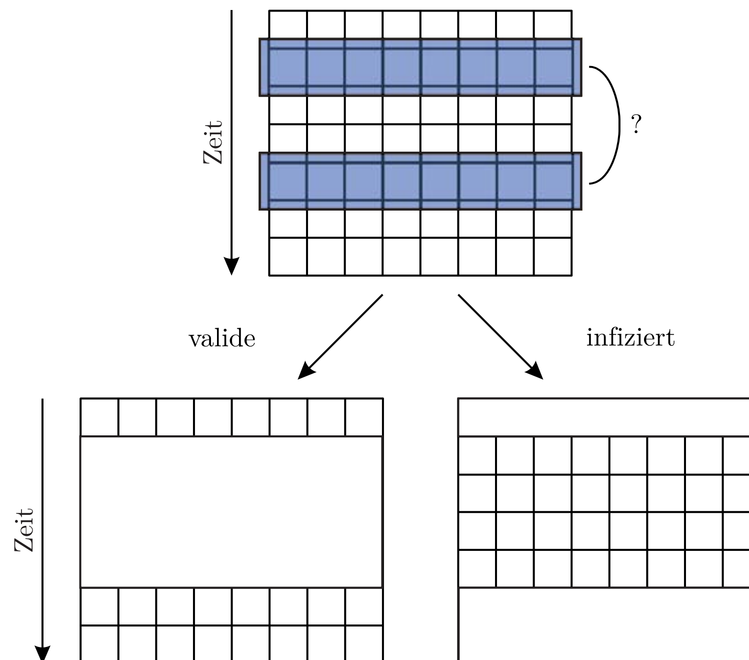


Abbildung 4.5: Der deklarative Debugging-Prozess.

Dieser grundlegende Unterschied wird in der Abbildung 4.5 verdeutlicht. Bei den zuvor beschriebenen Methoden wurden immer die durch einen einzelnen Ausführungsschritt verursachten Zustandsänderungen untersucht. Im Gegensatz dazu werden beim deklarativen Debugging mehrere zusammenhängende Zustandsänderungen bewertet. Grundlage hierfür sind die semantisch zusammenhängenden Teilberechnungen des Programmablaufs. Wie in der Abbildung 4.5 angedeutet, wird der Programmzustand zu Beginn einer Teilberechnung b mit dem Programmzustand am Ende der Ausführung von b verglichen. Dabei wird geprüft, ob der Zustand vor der Ausführung *valide* und nach der Ausführung *infiziert* ist. Ist dies der Fall, so ist die entsprechende Berechnung infiziert und der gesuchte Defekt muss sich irgendwo zwischen den beiden untersuchten Zuständen befinden. Für die anderen Fälle, in denen entweder beide Zustände valide oder beide Zustände infiziert sind, muss der entscheidende Übergang von validen zu infizierten Zuständen

außerhalb der Berechnung stattgefunden haben. In diesen Fällen können alle Zustandsänderungen, die zwischen den beiden untersuchten Zuständen liegen, aus dem Suchraum entfernt werden.

Der Vorteil dieses höheren Abstraktionsgrades lässt sich anhand eines Beispiels verdeutlichen. Angenommen die zu untersuchende Berechnung sei der Aufruf einer Methode `sort`, welcher als Argument ein zu sortierendes Array von Integer-Werten übergeben wird. Um die korrekte Ausführung des Aufrufs zu überprüfen, zeigt ein deklarativer Debugger die Programmzustände vor und nach dem Aufruf an. Der Aufruf wird dann als valide klassifiziert, wenn das übergebene Array nach dem Methodenaufruf die gleichen Werte wie vor dem Methodenaufruf enthält und die Folge der Werte im Array nach dem Aufruf sortiert ist. Andernfalls wird der Aufruf als infiziert klassifiziert. Beim deklarativen Debugging muss somit nur das tatsächliche Ergebnis des Methodenaufrufs mit dem gewünschten Ergebnis verglichen werden. Von den einzelnen Ausführungsschritten des Methodenaufrufs wird vollständig abstrahiert. Beim Trace- oder Omniscient-Debugging ist diese Abstraktion nicht implizit vorhanden. Beide Verfahren basieren grundsätzlich auf dem Ansatz, einzelne Ausführungsschritte zu untersuchen. Zwar lässt sich das Verhalten eines deklarativen Debuggers mit einem Omniscient-Debugger simulieren, indem manuell zu den Programmzuständen vor und nach dem Aufruf der Methode `sort` gesprungen wird, die Prüfung der Korrektheit des Methodenaufrufs ist dann aber aufwendiger und weniger komfortabel, da beim Omniscient-Debugging immer nur der Zustand zu einem Zeitpunkt der Programmausführung angezeigt werden kann. Bei komplexeren Programmzuständen wäre es dann notwendig, zwischen dem Vorher- und Nachher-Zustand manuell hin und her zu springen. Beim deklarativen Debugging sind beide Zustände gleichzeitig sichtbar und darüber hinaus können Zustandsänderungen, die im Verlauf des Methodenaufrufs stattgefunden haben, visuell hervorgehoben werden.

In gewisser Weise überträgt der höhere Abstraktionsgrad das Konzept der Kapselung (encapsulation) [89, S. 152][17, S. 42ff.] auf das Debugging. In der Entwicklungs- und der Implementierungsphase wird dieses Konzept seit Jahren erfolgreich eingesetzt. Das Paradigma der Objektorientierung [162] basiert zu großen Teilen auf der Kapselung von Daten in Objekten, welche wiederum durch Komponenten gekapselt werden. In objektorientierten Sprachen wie C# [158] oder Java [89] wird der Zugriff auf Daten durch

Schnittstellen gekapselt. Eine Schnittstelle definiert eine Menge von Methoden, über die mit den gekapselten Elementen interagiert werden darf. Die gekapselten Elemente werden durch die Schnittstelle vor einem direkten Zugriff von außen abgeschirmt, wodurch die Abhängigkeiten zwischen den gekapselten Elementen verringert wird. Die Kapselung reduziert somit die Komplexität von Programmen und ermöglicht es, einmal entwickelte Komponenten leichter wiederzuverwenden. Zusammenfassend lässt sich festhalten, dass die Kapselung erheblich dazu beigetragen hat, den Aufwand und die Komplexität in der Softwareentwicklung zu reduzieren.

Das deklarative Debugging nutzt die in einem Programm vorhandenen Komponentenbeziehungen, um von der konkreten Implementierung zu abstrahieren und so die Komplexität des Debugging-Prozesses zu reduzieren. Im Falle eines Java-Programms muss während des Debugging-Prozesses zum Beispiel ausschließlich die semantische Korrektheit von Methodenaufrufen überprüft werden.

Der höhere Abstraktionsgrad kann in vielen Fällen dazu beitragen, den Debugging-Prozess zu vereinfachen. Er führt allerdings auch dazu, dass die Position eines Defekts nicht mehr so exakt bestimmt werden kann. Das Ergebnis des deklarativen Debugging-Prozesses ist die Identifikation einer defekten Berechnung. Eine Berechnung besteht jedoch aus einer Folge von Berechnungsschritten. Für den Fall der oben erwähnten Methode `sort` würde dies bedeuten, dass der deklarative Debugging-Prozess zum Beispiel terminieren würde, wenn `sort` als defekter Methodenaufruf identifiziert ist. Es wäre somit nicht möglich, diejenige Anweisung zu identifizieren, die während der Ausführung von `sort` die Infektion verursacht hat. Dies ist ein klarer Nachteil gegenüber den zuvor beschriebenen Debugging-Methoden.

Eine weitere Besonderheit des deklarativen Debugging ist die Teilautomatisierung des Debugging-Prozesses. Der deklarative Debugger leitet den Debugging-Prozess, indem er den Benutzer dazu auffordert, Teilberechnungen zu klassifizieren. Aus den Klassifizierungen leitet der deklarative Debugger die intendierte Semantik des untersuchten Programms ab. Der Vergleich zwischen der intendierten Semantik und dem tatsächlichen Programmverhalten ermöglicht es, den Suchraum für die defekte Berechnung schrittweise zu verkleinern.

Ein deklarativer Debugger stellt den Programmablauf mithilfe eines Berechnungsbaums strukturiert dar. Diese Visualisierung ermöglicht es, den Programmablauf schneller zu erfassen und nachzuvollziehen. Zudem kann die Struktur des Berechnungsbaums bereits einen Hinweis auf die Position der defekten Berechnung geben. Darüber hinaus kann ein deklarativer Debugger die Klassifizierungen im Berechnungsbaum farbig hervorheben und so den noch verbleibenden Suchbereich darstellen. Der Benutzer kann so unmittelbar erkennen, welche Teile des Programmablaufs für die Position des Defekts noch infrage kommen und abzuschätzen, wie groß der verbleibende Suchaufwand ist.

Beim Trace- und Omniscient-Debugging fehlen solche Unterstützungen vollständig. Hier muss der Benutzer selbstständig durch den Zustandsraum des untersuchten Programms navigieren. Zudem gibt es auch keine Möglichkeit, Teile der Programmausführung als valide oder infiziert zu markieren. Der Benutzer hat daher einen schlechteren Überblick über den Fortschritt des Debugging-Prozesses. Da sowohl Trace- und Omniscient-Debugging aus diesem Grund sehr viel unstrukturierter sind, besteht die Gefahr, dass ein Benutzer während der manuellen Suche nach dem Defekt Zustände mehrfach untersucht.

Der Aufwand zum Auffinden eines Defekts wird beim deklarativen Debugging, ähnlich wie beim Trace-Debugging, im Wesentlichen durch die Größe des Zustandsraums des untersuchten Programmablaufs bestimmt. Je mehr Berechnungen ein Programmablauf enthält und je komplexer die zu bewertenden Zustandsänderungen, desto aufwendiger gestaltet sich die Suche. Im Vergleich zum Trace-Debugging hat das deklarative Debugging jedoch den entscheidenden Vorteil, dass die Bewertung der Zustandsänderungen nicht für einzelne Ausführungsschritte, sondern für semantisch zusammenhängende Berechnungen, die eine Folge von Ausführungsschritten umfassen, erfolgt. Durch die Bewertung ganzer Berechnungen kann der Suchraum in der Regel sehr viel schneller verkleinert werden als beim Trace-Debugging. Der Typ einer Infektion (vgl. Abschnitt 3.2) ist für den Aufwand des deklarativen Debugging weniger entscheidend. Der Aufwand, der benötigt wird, um eine Folge von Zustandsänderung zu bewerten, hängt in größerem Maße von der Komplexität der Zustände und der Position des infizierten Wertes ab als von der Art der Infektion.

4.3.2 Adaption für die Programmiersprache Java

In diesem Abschnitt wird gezeigt, welche Anpassungen vorzunehmen sind und welche Besonderheiten zu beachten sind, wenn die deklarative Debugging-Methode auf Java-Programme angewendet wird.

4.3.2.1 Berechnungsbaum für Java-Programme

Die Grundlage des deklarativen Debugging von Java-Programmen ist die Definition eines Berechnungsbaums, der die Ausführung eines Java-Programms in geeigneter Weise repräsentiert. In einem Java-Programm ist die Methode das zentrale Konstrukt, mit dem das Verhalten von Objekten bzw. ganzer Programme beschrieben wird. Der Ablauf eines Java-Programms besteht aus einer Menge von Methodenaufrufen C, die sich aufgrund ihrer gegenseitigen Beziehungen folgendermaßen in einer Baumstruktur anordnen lassen:

- Sei K die Menge aller Knoten des Berechnungsbaums, dann repräsentiert jeder Knoten $k \in K$ einen Methodenaufruf $c \in C$.
- Die Wurzel des Berechnungsbaums repräsentiert den Aufruf der `main`-Methode.
- Sei $(c_1, \ldots, c_n), n \in \mathbb{N}$ die Folge von Methodenaufrufen, die während der Ausführung des Methodenaufrufs c aufgetreten sind. Der Knoten k, welcher den Methodenaufruf c repräsentiert, besitzt dann eine Folge von Kindknoten $k_1, \ldots, k_n$ für die gilt, dass k_i für $i \in \mathbb{N}, i \leq n$ den Methodenaufruf c_i repräsentiert.

4.3.2.2 Seiteneffekte in Java-Programmen

Wie bereits erwähnt, wurde deklaratives Debugging ursprünglich für deklarative Programmiersprachen mit funktionalem und/oder logischem Programmierparadigma eingesetzt. Eine grundlegende Eigenschaft der deklarativen Programmiersprachen ist die referentielle Transparenz [133]. Dies bedeutet, dass ein Ausdruck durch einen beliebigen anderen Ausdruck mit identischem Wert ersetzt werden kann, ohne dass dadurch die Wirkung des

Programms verändert wird. Ein referentiell transparenter Ausdruck besitzt keine Nebeneffekte, d. h., seine Auswertung hat keinerlei Einfluss auf den Programmzustand. Im Rahmen des deklarativen Debugging erleichtert die Abwesenheit von Nebeneffekten die Klassifikation der Knoten des Berechnungsbaums erheblich. Um zum Beispiel den Aufruf einer Funktion ohne Nebeneffekte zu klassifizieren, müssen ausschließlich die Parameter und der Rückgabewert der Funktion untersucht werden.

In objektorientierten und imperativen Sprachen besteht ein Programm im Wesentlichen aus Anweisungen, die den Programmzustand ändern. Aus diesem Grund sind Nebeneffekte ein grundlegender Bestandteil dieser Sprachen. Beispielsweise können während eines Methodenaufrufs in Java die Argumente des Aufrufs, die Attribute des Objekts, dessen Methode aufgerufen wird, sowie statische Klassenattribute manipuliert werden. Das Ergebnis eines Methodenaufrufs besteht daher nicht nur aus dem Rückgabewert der Methode, sondern auch aus allen verursachten Nebeneffekten. Um zu beurteilen, ob ein Methodenaufruf valide oder infiziert ist, müssen diese Nebeneffekte bekannt sein.

4.3.2.3 Erforderliche Informationen zur Klassifizierung eines Methodenaufrufs

Die für die Klassifizierung des Aufrufs einer Java-Methode benötigten Informationen lassen sich am Beispiel der in Listing 4.1 dargestellten Java-Klasse `Foo` ermitteln. Die Klasse implementiert drei Methoden: den Konstruktor, die Methode `bar` und die `main`-Methode. Während der Ausführung der `main`-Methode werden nacheinander vier Instanzen der Klasse `Foo` erzeugt (Zeilen 20–23). Danach wird die Referenz `b.next` auf das durch `c` referenzierte `Foo`-Objekt gesetzt (Zeile 24) und anschließend die Methode `bar` des Objekts `a` mit dem Argument `b` aufgerufen (Zeile 25). Zuletzt wird der Rückgabewert auf der Konsole ausgegeben (Zeile 26).

Die Zustandsänderungen, die der Aufruf der Methode `bar` in Zeile 25 verursacht, sind in der Tabelle 4.1 dargestellt. Die Tabelle zeigt die Werte der lokalen Variablen `this`, `element` und `tmp`, den Rückgabewert `return`, die Werte des statischen Klassenattributs `counter` und die Werte der Attribute aller Objekte des Heaps. Für diese Variablen wird die Wertentwicklung vor (Zeile 24), während (Zeile 11–16) und nach (Zeile 25) der Ausführung

Listing 4.1: Java-Klasse `Foo` mit Nebeneffekten.

```
 1 public class Foo {
 2
 3   private static int counter = 0;
 4   private int value;
 5   private Foo next;
 6
 7   public Foo(int value) {
 8     this.value = value;
 9   }
10
11   public int bar(Foo foo) {
12     this.next = foo;
13     foo.next = new Foo(5);
14     Foo tmp = new Foo(6);
15     foo = tmp;
16     return ++counter;
17   }
18
19   public static void main(String[] args) {
20     Foo a = new Foo(1);
21     Foo b = new Foo(2);
22     Foo c = new Foo(3);
23     Foo d = new Foo(4);
24     b.next = c;
25     int i = a.bar(b);
26     System.out.println(i);
27   }
28 }
```

des Methodenaufrufs dargestellt. In den Spalten sind jeweils die Werte der Variablen nach der Ausführung der entsprechenden Zeile angegeben. In der Spalte mit der Quelltextzeilennummer 11 befinden sich die Startwerte der Variablen, dies sind die Werte der Variablen zu Beginn des Aufrufs der Methode `bar`. In der Spalte mit der Quelltextzeilennummer 16 befinden sich die Endwerte der Variablen. Diese entsprechen den Werten der Variablen am Ende des Aufrufs der Methode `bar`. Wenn sich der Wert einer Variablen durch die Ausführung einer Zeile geändert hat, dann ist der neue Wert durch Fettschrift hervorgehoben. Werte in spitzen Klammern sind eine Referenz auf ein Objekt, dabei bedeutet „$<x>$“, mit $x \in \mathbb{N}$, dass das Objekt mit der

ID x referenziert wird. Bei Variablen, die Attribut eines Objekts sind, bildet die Objekt-ID das Präfix des Variablennamens. Die Variable `<1>.value` repräsentiert zum Beispiel das Attribut `value` des Objekts mit der ID 1.

Tabelle 4.1: Durch den Aufruf der Methode `bar` (vgl. Listing 4.1 Zeile 25) verursachte Zustandsänderungen. In der Tabelle sind die Werte der Eingabemenge hellgrau (■) und die Werte der Ergebnismenge dunkelgrau (■) markiert. Die grau (■) markierten Werte gehören sowohl zur Eingabemenge als auch zur Ergebnismenge.

	Quelltextzeile							
Variable	24	11	12	13	14	15	16	25
this	–	<1>	<1>	<1>	<1>	<1>	<1>	–
foo	–	<2>	<2>	<2>	<2>	**<6>**	<6>	–
tmp	–	–	–	–	**<6>**	<6>	<6>	–
return	–	–	–	–	–	–	**1**	1
counter	0	0	0	0	0	0	**1**	1
<1>.value	1	1	1	1	1	1	1	1
<1>.next	null	null	**<2>**	<2>	<2>	<2>	<2>	<2>
<2>.value	2	2	2	2	2	2	2	2
<2>.next	<3>	<3>	<3>	**<5>**	<5>	<5>	<5>	<5>
<3>.value	3	3	3	3	3	3	3	3
<3>.next	null	null	null	null	null	null	null	null
<4>.value	4	4	4	4	4	4	4	4
<4>.next	null	null	null	null	null	null	null	null
<5>.value	–	–	–	**5**	5	5	5	5
<5>.next	–	–	–	**null**	null	null	null	null
<6>.value	–	–	–	–	**6**	6	6	–
<6>.next	–	–	–	–	**null**	null	null	–

In der Tabelle sind zwei Mengen von Werten markiert: die Eingabemenge und die Ergebnismenge. Die hellgrau (■) markierten Werte sind Teil der Eingabemenge und die dunkelgrau (■) markierten Werte sind Teil der

Ergebnismenge. Werte, die grau (■) markiert sind, gehören sowohl zur Eingabemenge als auch zur Ergebnismenge.

Die *Eingabemenge* umfasst alle Werte, die das Verhalten des Methodenaufrufs beeinflussen können. Dies sind die Startwerte der lokalen Variablen, die als Argumente übergeben wurden, und die Startwerte der nicht lokalen Variablen, auf die während der Ausführung der Methode zugegriffen werden kann. Die Startwerte der lokalen Variablen `this` und `foo` werden der Methode `bar` als Argumente übergeben. [4] Der Startwert der Variablen `this` referenziert das `Foo`-Objekt mit der Objekt-ID 1 und der Startwert der Variablen `foo` das `Foo`-Objekt mit der Objekt-ID 2. Zu den nicht lokal definierten Variablen, auf die während der Ausführung der Methode `bar` zugegriffen werden kann, zählt zum einen das statische Klassenattribut `counter`. Darüber hinaus kann während der Ausführung auch auf die Attribute der Objekte zugegriffen werden, die von den Argumenten (`this`, `foo`) oder den statischen Klassenattributen (`counter`) direkt oder indirekt referenziert werden. Da über die Argumente des Methodenaufrufs auf die Objekte <1> und <2> zugegriffen werden kann, sind die Startwerte der Attribute dieser Objekte (`value` und `next`) ebenfalls Teil der Eingabemenge. Da der Startwert der Variablen `<2>.next` das Objekt <3> referenziert, sind auch die Startwerte der Attribute `<3>.value` und `<3>.next` Teil der Eingabemenge. Darüber hinaus enthält die Eingabemenge keine weiteren Referenzen auf Objekte, deren Attributwerte noch nicht Teil der Eingabemenge sind; die Eingabemenge ist somit vollständig definiert, da sie alle Werte enthält, durch die das Verhalten des Methodenaufrufs von außen beeinflusst werden kann.

Die *Ergebnismenge* umfasst alle Werte, die zum Resultat eines Methodenaufrufs gehören. Dazu gehören der Rückgabewert `return`[5] und der Endwert des geänderten statischen Klassenattributs `counter`. Die lokalen Variablen sind Teil des Stack-Frames des Methodenaufrufs und existieren nur für die Dauer des Methodenaufrufs. Jede lokale Variable, die kein Argument des Methodenaufrufs ist, wie zum Beispiel die Variable `tmp`, kann

[4] Die `this`-Referenz auf das Objekt, dessen `bar`-Methode aufgerufen wurde, kann in diesem Zusammenhang als weiteres Argument der Methode aufgefasst werden.

[5] Der Rückgabewert ist hier als Variable dargestellt, welche genau genommen zur Laufzeit nicht existiert. Die unechte Variable `return` wird verwendet, um eine übersichtliche und einheitliche Darstellung der Werte zu ermöglichen.

daher keinen Einfluss auf den Programmzustand nach Beendigung des Methodenaufrufs haben und ist somit auch nicht Teil der Ergebnismenge. Die Werte der lokalen Variablen, die zu den Argumenten des Methodenaufrufs gehören (`this` und `foo`), sind Teil der Ergebnismenge. Im Unterschied zu den übrigen Variablen sind hier jedoch die Startwerte und *nicht* die Endwerte Teil der Ergebnismenge. Der Grund hierfür ist die Tatsache, dass in Java alle Argumente eines Methodenaufrufs als Wertparameter und nicht als Referenzparameter übergeben werden. Dies hat zur Folge, dass die Änderungen der Argumente innerhalb eines Methodenaufrufs keinen Einfluss auf die Werte bzw. Variablen haben, die als Argumente übergeben wurden. Für den Aufruf der Methode `bar` in Zeile 25 bedeutet dies zum Beispiel, dass die lokale Variable `b` nach der Beendigung des Methodenaufrufs `bar` dasselbe Objekt referenziert wie vor dem Aufruf. Da es sich bei dem Argument `b` jedoch um eine Referenz auf ein Objekt handelt, können durch den Aufruf von `bar` die Attribute des referenzierten Objekts geändert werden. Dies gilt auch für das durch die lokale Variable `a` referenzierte Objekt, dessen Methode `bar` aufgerufen wird. Die Ergebnismenge des Methodenaufrufs umfasst somit neben dem Rückgabewert und dem Endwert des statischen Klassenattributs `counter` die Startwerte der Argumente `this` und `foo` und die Endwerte der Attribute, deren Objekte direkt oder indirekt durch die Werte der Ergebnismenge referenziert werden. Im vorliegenden Beispiel werden durch die Argumente zu Beginn des Methodenaufrufs die Objekte `<1>` und `<2>` referenziert. Nach Beendigung des Methodenaufrufs werden diese Objekte weiterhin durch die lokalen Variablen `a` und `b` der `main`-Methode referenziert. Aus diesem Grund sind die Endwerte der Attribute dieser Objekte Teil der Ergebnismenge. Weiterhin wird durch den Endwert der Variablen `<2>.next` das Objekt `<5>` referenziert. Folglich sind auch die Endwerte der Attribute des Objekts `<5>` Teil der Ergebnismenge. Weitere Objekte werden durch die Werte der Ergebnismenge nicht referenziert; sie ist damit für den Methodenaufruf `bar` vollständig definiert. Wie sich aus der Tabelle 4.1 entnehmen lässt, gibt es Attribute, deren Werte ausschließlich in der Eingabemenge enthalten sind, Attribute, deren Werte ausschließlich in der Ergebnismenge enthalten sind, und Attribute, deren Werte weder in der Eingabemenge noch in der Ergebnismenge enthalten sind. Zu denen, die nur in der Eingabemenge enthalten sind, zählen die Attribute des Objekts `<3>`, welches zum Anfang des Methodenaufrufs durch einen Wert der Eingabemenge, am Ende des Methodenaufrufs jedoch nicht durch einen Wert der

Ergebnismenge referenziert wird. Für das Objekt `<5>` verhält es sich genau andersherum. Das Objekt wird erst während der Ausführung der Methode `bar` erzeugt, weshalb die Attribute von `<5>` nicht Teil der Eingabemenge sind. Da das Objekt jedoch am Ende des Methodenaufrufs durch einen der Werte der Ergebnismenge referenziert wird, sind die Endwerte der Attribute des Objekts Teil der Ergebnismenge. Das Objekt `<6>` wird ebenfalls erst während der Ausführung der Methode `bar` erzeugt. Am Ende der Ausführung des Methodenaufrufs wird es jedoch ausschließlich durch die Endwerte der lokalen Variablen `foo` und `tmp` referenziert. Da `tmp` nur für die Dauer des Methodenaufrufs existiert und die Änderungen von `foo` keine Auswirkungen auf die lokale Variable `b` des Aufrufs der Methode `main` haben, existiert nach der Beendigung des Aufrufs von `bar` keine Referenz mehr auf das Objekt `<6>`. Aus diesem Grund sind die Werte der Attribute des Objekts `<6>` weder Teil der Eingabemenge noch Teil der Ergebnismenge. Das Objekt `<4>` wird im Aufruf der `main`-Methode durch die Variable `d` referenziert. Die Werte der Attribute dieses Objekts sind nicht Element der Eingabemenge des Aufrufs von `bar`, weil das Objekt durch keinen Wert der Eingabemenge referenziert wird. Da das Objekt nicht durch die Eingabemenge referenziert wird und auch nicht während des Aufrufs der Methode `bar` erzeugt wird, kann es folglich auch nicht Teil der Ergebnismenge sein.

4.3.2.4 Der Zustandsraum eines Methodenaufrufs

Die Eingabemenge und die Ergebnismenge definieren die Menge an Informationen, die mindestens erforderlich ist, um einen Methodenaufruf eines Java-Programms zu klassifizieren. Tatsächlich ist die Menge an Informationen, die JHyde während der Ausführung eines Methodenaufrufs speichert eine Obermenge der Eingabemenge und der Ergebnismenge. Die zusätzlichen Informationen, die JHyde speichert, werden unter anderem für das Omniscient-Debugging benötigt. Im Folgenden wird der Zustandsraum eines Methodenaufrufs beschrieben. Dieser entspricht der Menge an Informationen, die JHyde für die Klassifikation eines Methodenaufrufs bereitstellt.

Sei W die Menge aller Werte des Programmablaufs, dann lässt sich die Menge der zur Klassifizierung eines Methodenaufrufs c bereitgestellten Informationen durch die Funktion $z_c : \mathbb{N} \times \mathbb{N} \to \mathcal{P}(W)$ abbilden. Die Funktion z_c bildet für den Methodenaufruf c zwei Zeitpunkte $t_1, t_2 \in \mathbb{N}$ auf ein Element

der Potenzmenge von W, d. h. auf eine Teilmenge der Werte des Programmablaufs, ab. Das Resultat von z_c wird im Folgenden als Zustandsmenge des Methodenaufrufs c bezeichnet.

Für die Zeitpunkte t_1 und t_2 enthält die Zustandsmenge $z_c(t_1, t_2)$ des Methodenaufrufs c die folgenden Elemente:

- Die Werte aller lokalen Variablen von c zu den Zeitpunkten t_1 und t_2. Zu den lokalen Variablen zählen alle Argumente des Methodenaufrufs, alle lokal definierten Variablen und im Falle des Aufrufs einer nicht statischen Methode die Referenz auf das Objekt dessen Methode aufgerufen wird.
- Die Werte der lokalen Pseudovariablen `return` zu den Zeitpunkten t_1 und t_2. Der Endwert der Variablen `return` entspricht dem Rückgabewert des Methodenaufrufs c.
- Die Werte aller statischen Klassenattribute zu den Zeitpunkten t_1 und t_2.
- Die Werte der Attribute aller erreichbaren Objekte zu den Zeitpunkten t_1 und t_2. Für jedes Objekt o, welches durch einen Wert der Zustandsmenge referenziert wird, sind die Werte aller Attribute von o zu den Zeitpunkten t_1 und t_2 ebenfalls Teil der Zustandsmenge. Die Werte der Attribute eines referenzierten Objekts, die Teil der Zustandsmenge sind, können wiederum weitere Objekte referenzieren. Die Werte der Attribute dieser Objekte zu den Zeitpunkten t_1 und t_2 sind ebenfalls Teil der Zustandsmenge. Ein Array wird in diesem Zusammenhang als spezieller Objekttyp behandelt, bei dem alle Attribute bzw. Elemente vom selben Typ sind.

Wenn $t_{start} : C \rightarrow \mathbb{N}$ für einen Methodenaufruf c den Zeitpunkt des Beginns der Ausführung von c und $t_{ende} : C \rightarrow \mathbb{N}$ für einen Methodenaufruf c den Zeitpunkt des Ausführungsendes von c zurückgibt, dann werden für die Klassifizierung des Methodenaufrufs c die folgenden Informationen bereitgestellt:

- Der vollqualifizierte Name der durch c aufgerufenen Methode und
- die Zustandsmenge des Methodenaufrufs c für die Zeitpunkte $t_{start}(c)$ und $t_{ende}(c)$: $z_c\big(t_{start}(c), t_{ende}(c)\big)$.

Wie bereits erwähnt, ist die Zustandsmenge eine Obermenge der Eingabemenge und der Ergebnismenge. Wenn c dem Methodenaufruf der Methode `bar` in der Zeile 25 in Listing 4.1 entspricht, dann ist die Zustandsmenge $z_c\big(t_{start}(c), t_{ende}(c)\big)$ in der Tabelle 4.2 dargestellt. Die Werte, die zur Zustandsmenge gehören, sind grau markiert. Die hellgrauen (■) Werte sind sowohl Element der Zustandsmenge als auch Element der Eingabemenge oder der Ergebnismenge.

Die dunkelgrau (■) markierten Werte sind ausschließlich Teil der Zustandsmenge. Da diese Werte weder Teil der Eingabemenge noch Teil der Ergebnismenge sind, werden sie für die Klassifikation des Methodenaufrufs c nicht zwangsläufig benötigt. Während der Ausführung des Methodenaufrufs sind diese Werte jedoch Teil des Programmzustands und werden gegebenenfalls für das Omniscient-Debugging benötigt. Aus diesem Grund sind sie ebenfalls Teil der dargestellten Informationen über den Methodenaufruf c.

In der Tabelle zeigt sich, dass von einer Variablen entweder kein Wert oder der Wert zum Zeitpunkt $t_{start}(c)$ und der Wert zum Zeitpunkt $t_{ende}(c)$ Teil der Zustandsmenge sind. Da die Werte der lokalen Variablen `foo` und `tmp` zum Zeitpunkt $t_{ende}(c)$ das Objekt `<6>` referenzieren, sind die Werte der Attribute dieses Objekts zu den Zeitpunkten t_{start} und t_{ende} ebenfalls Teil der Zustandsmenge.

4.3.2.5 Aufwandsvergleich zu deklarativen Sprachen

Um die Klassifikation der Knoten des Berechnungsbaums zu ermöglichen, wird für jeden Methodenaufruf c die Zustandsmenge $z_c\big(t_{start}(c), t_{ende}(c)\big)$ gespeichert. Für eine Sprache mit Nebeneffekten ist die Menge der Informationen, die zur Klassifizierung benötigt wird, in der Regel sehr viel größer als für eine Sprache ohne Nebeneffekte. Der Aufwand für die Klassifikation der Knoten des Berechnungsbaums ist aus diesem Grund für Programmiersprachen mit Nebeneffekten ebenfalls größer. Bei der Adaption des deklarativen Debugging für die Programmiersprache Java ist es daher wichtig Eingabe- und Ergebnismenge übersichtlich und kompakt darzustellen, um den Klassifikationsaufwand so weit wie möglich zu minimieren. Die Debugging-Informationen mithilfe der Benutzeroberfläche effizient analysieren zu können, um eine einfache und schnelle Defektsuche zu ermöglichen.

Tabelle 4.2: Zustandsraum $z_c\big(t_{start}(c), t_{ende}(c)\big)$ für c :=<Aufruf der Methode `bar` in Listing 4.1, Zeile 25>. Die hellgrau (■) markierten Werte sind Element der Zustandsmenge und Element der Eingabemenge oder der Ergebnismenge. Die dunkelgrau (■) markierten Werte sind ausschließlich Element der Zustandsmenge.

	Quelltextzeile							
Variable	24	11	12	13	14	15	16	25
`this`	–	<1>	<1>	<1>	<1>	<1>	<1>	–
`foo`	–	<2>	<2>	<2>	<2>	**<6>**	<6>	–
`tmp`	–	–	–	–	**<6>**	<6>	<6>	–
`return`	–	–	–	–	–	–	**1**	1
`counter`	0	0	0	0	0	0	**1**	1
`<1>.value`	1	1	1	1	1	1	1	1
`<1>.next`	null	null	**<2>**	<2>	<2>	<2>	<2>	<2>
`<2>.value`	2	2	2	2	2	2	2	2
`<2>.next`	<3>	<3>	<3>	**<5>**	<5>	<5>	<5>	<5>
`<3>.value`	3	3	3	3	3	3	3	3
`<3>.next`	null	null	null	null	null	null	null	null
`<4>.value`	4	4	4	4	4	4	4	4
`<4>.next`	null	null	null	null	null	null	null	null
`<5>.value`	–	–	–	**5**	5	5	5	5
`<5>.next`	–	–	–	**null**	null	null	null	null
`<6>.value`	–	–	–	–	**6**	6	6	–
`<6>.next`	–	–	–	–	**null**	null	null	–

Die Gleichung von KOWALSKI [87], „algorithm = logic + control", besagt, dass ein Algorithmus aus einer logischen Komponente, die das für die Problemlösung verwendete Wissen enthält, und einer Kontrollkomponente, die die Problemlösungsstrategie definiert, besteht. In einer imperativen Programmiersprache müssen für die Implementierung eines Algorithmus beide Komponenten gemeinsam durch eine Folge von Anweisungen umgesetzt werden. In einer deklarativen Programmiersprache werden Teile der Kontrolllogik durch die Ausführungsumgebung des Programms umgesetzt [100]. In diesen Sprachen muss daher nur definiert werden, *was* berechnet werden soll und nicht *wie* diese Berechnung im Einzelnen durchzuführen ist.

Dies führt dazu, dass deklarative Programme in der Regel sehr viel kompakter sind als ihre imperativen Gegenstücke. In imperativen Programmen existiert diese Trennung von Programmlogik und Kontrollfluss nicht, wodurch diese grundsätzlich komplexer und damit auch fehleranfälliger sind. Dieser Zusammenhang führt konsequenterweise auch dazu, dass das Debugging von imperativen Programmen aufwendiger ist als das Debugging von deklarativen Programmen. Dabei ist es wichtig zu beachten, dass der höhere Debugging-Aufwand nicht durch die Wahl der Debugging-Methode, sondern durch die Art der Programmiersprache bzw. des Programmierparadigmas verursacht wird. Bei der Adaption des deklarativen Debugging für die Programmiersprache Java muss dieser erhöhte Aufwand berücksichtigt werden. Dabei ist zu beachten, dass der höhere Debugging-Aufwand durch die Wahl der Programmiersprache Java und nicht dadurch, dass die deklarative Debugging-Methode grundsätzlich für imperative bzw. objektorientierte Programme ungeeignet ist, verursacht wird. In der Tat müssen zum Beispiel beim Trace-Debugging zur Überprüfung der Korrektheit eines Methodenaufrufs implizit dieselben Fakten überprüft werden. Beim Trace-Debugging gibt es jedoch zum einen nicht die Möglichkeit, die Programmzustände zu Beginn und nach Beendigung eines Methodenaufrufs miteinander zu vergleichen, und zum anderen existiert keine automatisierte und systematische Einschränkung des Suchraums.

4.4 Hybrides Debugging

Die drei vorgestellten Debugging-Methoden besitzen unterschiedliche Vor- und Nachteile; ihre Eignung ist daher von der Art des Einsatzszenarios abhängig. In der Tabelle 4.3 werden die drei Methoden anhand von sechs Eigenschaften miteinander verglichen. Die ersten vier Eigenschaften beziehen sich auf die Eignung der jeweiligen Methode für die in Abschnitt 3.2 beschriebenen Infektionstypen. Für jeden Infektionstyp wird dabei angegeben, wie gut sich mit der entsprechenden Debugging-Methode die Ursache für eine Infektion ermitteln lässt. Die Eigenschaft „Präzision“ vergleicht die Methoden hinsichtlich der Genauigkeit, mit der sich die Position des Defekts bestimmen lässt. Der Bedarf an Rechenleistung und Speicherplatz zur Durchführung des Debugging-Prozesses wird mit der Eigenschaft „Ressourcenbedarf“ verglichen. Der Vergleich der Debugging-Methoden geschieht in

Form eines Rankings, indem für jede Eigenschaft eine Rangfolge der drei Methoden gebildet wird. Dabei wird die beste der drei Methoden mit „1“ bewertet. Sind zwei Methoden gleichwertig, so erhalten sie die gleiche Platzierung. Die Rangfolgen sind aus der Analyse der Methoden in den Abschnitten 4.1–4.3 abgeleitet und basieren nicht auf empirischen Untersuchungen.[6] Sie sind somit eine Bewertung auf Grundlage der herausgearbeiteten Eigenschaften und Funktionalitäten der untersuchten Methoden.

Tabelle 4.3: Ranking der Debugging-Methoden.

Methode	Infektionstyp				Präzision	Ressourcen-bedarf
	Ia	Ib	IIa	IIb		
Trace	3	3	3	3	1	1
Omniscient	1	2	1	2	1	2
Deklarativ	2	1	2	1	2	2

Im Bezug auf den Suchaufwand belegt das Trace-Debugging unabhängig von der Art der Infektion jeweils den letzten Platz. Da diese Methode die wenigsten Informationen über den Programmablauf zur Verfügung stellt, ist davon auszugehen, dass hier die Suche nach dem Defekt in der Regel aufwendiger ist als bei den übrigen Methoden. Beim Trace-Debugging kann die Programmausführung nur angehalten werden, um den gegenwärtigen Zustand des Programms zu untersuchen. Die anderen Methoden bieten wesentlich leistungsfähigere Möglichkeiten, den Zustandsraum eines Programms zu analysieren. Bei Infektionstypen vom Typ Ia und IIa ist das Omniscient-Debugging am besten geeignet. Bei diesen Infektionstypen führt die rückwärts gerichtete Navigation von der Fehlerwirkung zum Defekt in der Regel am schnellsten zum Ziel. Bei den Infektionen vom Typ Ib und IIb hingegen

[6] Ein objektiver empirischer Vergleich der Methoden hinsichtlich des Debugging-Aufwands ist nicht Gegenstand dieser Arbeit. Ein solcher Vergleich problematisch, da die Zeit zum Auffinden eines Defekts in großem Maße von der Erfahrung des Benutzers mit der jeweiligen Methode abhängt. Darüber hinaus kann derselbe Defekt aufgrund des Lerneffekts des Benutzers nicht mit unterschiedlichen Methoden mehrmals gesucht werden.

ist die Suche nach der Infektionsursache wesentlich komplexer und unstrukturierter. Die Ursache, welche eine Infektion dieses Typs verursacht haben können, lassen sich im Allgemeinen nur schwer ermitteln. Aus diesem Grund ist es schwierig, solche Infektionen mithilfe des Omniscient-Debugging zurückzuverfolgen. Daher wurde hier das deklarative Debugging als das am besten geeignete Verfahren eingestuft. Es bietet als einziges Verfahren bei diesen schwer zurückzuverfolgenden Fehlerwirkungen eine strukturierte Methode zur Lokalisierung des Defekts, die den Suchraum systematisch verkleinert.

Die Genauigkeit, mit der die Position eines Defekts im Quelltext bestimmt werden kann, ist beim deklarativen Debugging geringer als bei den übrigen Methoden. Da das deklarative Debugging auf einer höheren Abstraktionsebene arbeitet, kann nur die defekte Berechnung bzw. die defekte Methode bestimmt werden. Trace- und Omniscient-Debugging sind wesentlich exakter, da mit ihnen auch die genaue Position des Defekts innerhalb der defekten Methode bestimmt werden kann.

Der geringste Ressourcenbedarf liegt beim Trace-Debugging vor, da ein Trace-Debugger prinzipiell keine Informationen über die Ausführung des untersuchten Programms aufzeichnen muss. Der Zeit- und Speicherplatzbedarf für Omniscient-Debugging und deklaratives Debugging ist in etwa identisch. Bei beiden Methoden muss die gesamte Ausführung des untersuchten Programms aufgezeichnet werden. Daher ist der Ressourcenbedarf bei diesen Methoden um ein Vielfaches größer als beim Trace-Debugging.

Der Vergleich zeigt, dass keine der Debugging-Methoden den anderen in allen Punkten überlegen ist. Die beiden Methoden, die bessere und ausgereiftere Funktionalitäten als das Trace-Debugging zur Suche des Defekts bieten, benötigen hierfür zusätzliche Informationen über den Ablauf des untersuchten Programms. Um diese Informationen zu sammeln, werden zusätzlicher Speicherplatz und zusätzliche Rechenzeit benötigt. Grundsätzlich ist jedoch davon auszugehen, dass der zusätzliche Ressourcenbedarf zur Aufzeichnung des Programmablaufs lohnenswert ist, da hierdurch die Suche nach dem Defekt erheblich beschleunigt werden kann. Ferner ist anzunehmen, dass die bei der Suche eingesparte Zeit um ein Vielfaches länger ist als die zusätzlich benötigte Zeit zum Aufzeichnen des Programmablaufs. Beim Vergleich zwischen Omniscient-Debugging und deklarativem Debugging hat der Infek-

tionstyp entscheidenden Einfluss darauf, welches Verfahren besser geeignet ist, den Defekt zu finden.

Da die Informationen über den Ablauf des untersuchten Programms, welche für das Omniscient-Debugging und das deklarative Debugging aufgezeichnet werden müssen, nahezu identisch sind, ist es sinnvoll diese beiden Methoden miteinander zu kombinieren. Für die Aufzeichnung des Programmablaufs würde durch die Kombination der Methoden nahezu kein zusätzlicher Aufwand entstehen und während der Suche nach dem Defekt könnte der Benutzer diejenige Methode wählen, die ihm am effizientesten erscheint. Das hybride Debugging ist eine Kombination aus Omniscient-Debugging und deklarativem Debugging, die die Vorteile der beiden Methoden verbindet.

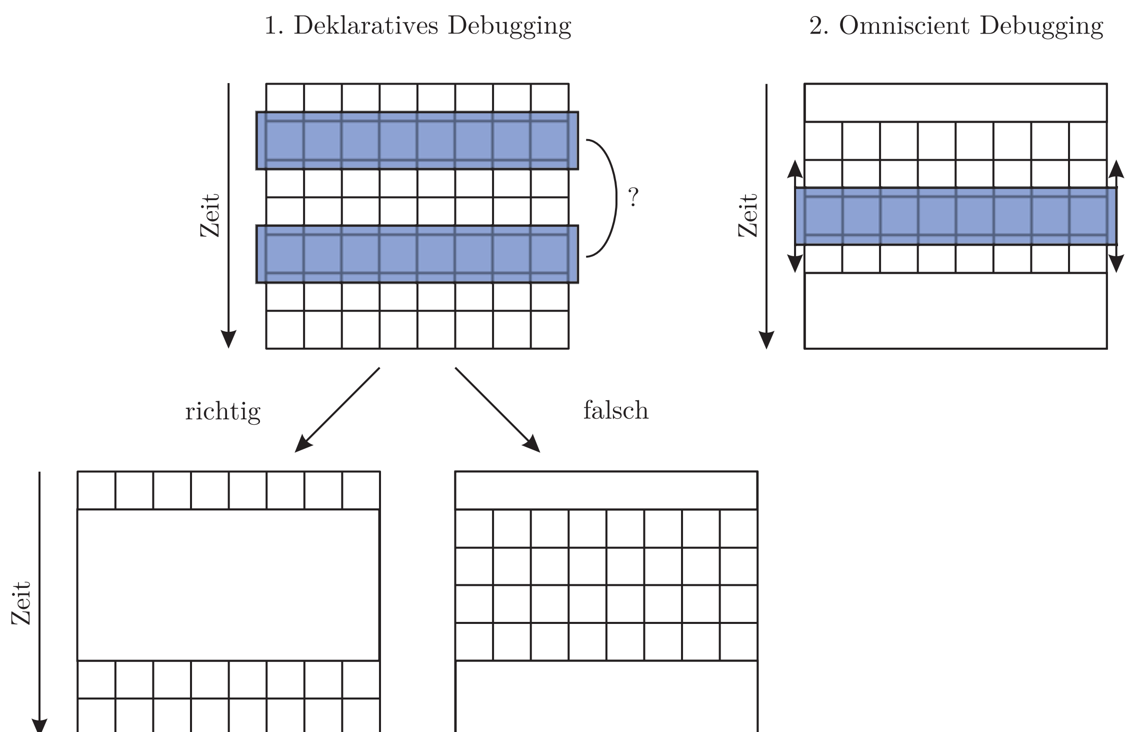


Abbildung 4.6: Der hybride Debugging-Prozess. Deklaratives Debugging und Omniscient-Debugging lassen sich kombinieren, indem zunächst mithilfe des deklarativen Debugging die defekte Methode bestimmt wird und anschließend innerhalb der defekten Methode mithilfe des Omniscient-Debugging die fehlerhafte Anweisung gesucht wird.

Wie in der Abbildung 4.6 dargestellt, kann bei Infektionen, deren Ursachen sich mit dem Omniscient-Debugging nicht so leicht ermitteln lassen, das deklarative Debugging zunächst dazu verwendet werden, die fehlerhafte Berechnung zu identifizieren. Durch den gut strukturierten und teilautomatisierten Debugging-Prozess ist das deklarative Debugging vor allem bei den schwierig zurückzuverfolgenden Infektionen gut geeignet, um die defekte Berechnung zu identifizieren. Anschließend wird das Omniscient-Debugging eingesetzt, um innerhalb der defekten Berechnung die defekte Anweisung zu finden.

Das hybride Debugging ist so aufgebaut, dass jederzeit während des laufenden Prozesses zwischen den beiden unterstützten Debugging-Techniken gewechselt werden kann. Bei der in Kapitel 6 beschriebenen Umsetzung dieser Methode durch JHyde werden die aufgezeichneten Daten eines untersuchten Programms auf unterschiedliche Art und Weise in mehreren Ansichten repräsentiert. Einige dieser Ansichten dienen dazu, den aufgezeichneten Programmablauf mithilfe der deklarativen Debugging-Technik nach dem Defekt zu durchsuchen, die anderen Ansichten ermöglichen das Omniscient-Debugging. Während der Suche können die benutzten Ansichten jederzeit gewechselt werden. Dem Benutzer ist es somit möglich zu jedem Zeitpunkt diejenige Ansicht zu wählen, die seiner Meinung nach am besten für die weitere Suche nach dem Defekt geeignet ist. Auf diese Weise können die Vorteile der beiden Methoden miteinander kombiniert werden.

4.5 Fazit

In diesem Kapitel wurden Trace-Debugging, Omniscient-Debugging und deklaratives Debugging vorgestellt. Im Gegensatz zu Trace- und Omniscient-Debugging ist das deklarative Debugging ursprünglich für deklarative Programmiersprachen entwickelt worden. Es wurde gezeigt, welche Auswirkungen die in Java vorhandenen Seiteneffekte auf das deklarative Debugging haben. Im Anschluss sind die drei Debugging-Methoden miteinander verglichen worden. Der Vergleich wurde anhand der im vorangegangenen Kapitel beschriebenen Infektionstypen durchgeführt. Das Trace-Debugging ist den anderen beiden Methoden in allen untersuchten Punkten unterlegen. I Vergleich zwischen Omniscient-Debugging und deklarativem Debugging gibt es

keinen eindeutigen Sieger. Welche der beiden Methoden besser zur Defektsuche geeignet ist, hängt wesentlich vom Typ des infizierten Programmzustandes ab.

Ausgehend von diesen Erkenntnissen wurde eine hybride Debugging-Methode entwickelt. Diese kombiniert Omniscient-Debugging und deklaratives Debugging und vereinigt so die Vorteile beider Verfahren.

Kapitel 5

Deklarative Debugging-Strategien

Inhalt

5.1 Top-Down 85
5.2 Divide-and-Query 86
5.3 D&Q mit gewichtsunabhängigen Infektionswahrscheinlichkeiten 91
5.4 Empirische Untersuchung 107
5.5 Fazit 111

Der deklarative Debugging-Prozess basiert im Wesentlichen darauf, die Knoten des Berechnungsbaums als valide oder infiziert zu klassifizieren. Nach der Klassifizierung eines Knotens wird das Ergebnis dazu verwendet, den Suchraum zu verkleinern. Das Verfahren terminiert, wenn der Suchraum entweder leer ist oder nur noch einen einzigen infizierten Knoten enthält. Im ersten Fall ist der gesamte Programmablauf valide, im zweiten Fall ist der verbleibende Knoten der gesuchte defekte Knoten.

Die Effizienz des deklarativen Debugging wird maßgeblich durch die Anzahl der Klassifizierungen beeinflusst, die notwendig sind, um eine defekte Berechnung zu finden. Die Auswahl der zu klassifizierenden Knoten sowie deren Reihenfolge hat dabei erheblichen Einfluss auf die Anzahl der zu klassifizierenden Knoten. Grundsätzlich gilt: Je mehr Knoten nach einer Klassifizierung aus dem Suchraum entfernt werden, desto geringer ist der Aufwand zur Klassifizierung der verbleibenden Knoten.

Sei K die Menge aller Knoten des Berechnungsbaums und S_i die Menge aller Knoten des Suchraums nach i Klassifizierungsschritten, d. h., nachdem der Benutzer i Knoten klassifiziert hat. Zu Beginn der Suche entspricht der Suchraum dem gesamten Berechnungsbaum, d. h. $S_0 = K$. Ferner sei T_k die Menge aller Knoten des Teilbaums mit der Wurzel k und c_k der Teil des Programmcodes, d. h. die Funktion, Methode o. ä., die zur Berechnung des Ergebnisses von k ausgeführt wurde. Wird nun im i-ten Klassifizierungsschritt ein Knoten $k \in S_{i-1}$ klassifiziert, so ergibt sich in Abhängigkeit von der Klassifizierung der neue Suchraum S_i folgendermaßen:

- Wird k als **valide** klassifiziert, dann wird der Teilbaum mit der Wurzel k aus dem Suchraum entfernt, d. h. $S_i = S_{i-1} \setminus T_k$.

- Wird k als **infiziert** klassifiziert, so wird die Suche im infizierten Teilbaum von k fortgesetzt. In diesem Fall gilt $S_i = S_{i-1} \cap T_k$.

- Wird k als **vertrauenswürdig** klassifiziert, dann sind alle Knoten, deren Berechnung auf demselben Teil des Quelltextes basiert, ebenfalls vertrauenswürdig. Daraus folgt, dass $S_i = \{a \in S_{i-1} : c_a \neq c_k\}$.

Im Bezug auf den verbleibenden Suchaufwand ist es also günstig, wenn der Teilbaum, dessen Wurzel als valide klassifiziert wird, möglichst groß ist oder der Teilbaum, dessen Wurzel als infiziert klassifiziert wird, möglichst klein ist. Wird ein Knoten k als vertrauenswürdig klassifiziert, dann ist der verbleibende Suchaufwand umso geringer, je größer die Menge der vertrauten Methodenaufrufe $\{a \in S_{i-1} : c_a = c_k\}$ ist. In den folgenden Untersuchungen spielen vertrauenswürdige Knoten eine untergeordnete Rolle, da sie für das deklarative Debugging nicht wesentlich sind. Während der Defektsuche werden alle Knoten ausschließlich als valide oder infiziert klassifiziert.

Die Such- bzw. Navigationsstrategie eines deklarativen Debuggers legt fest, in welcher Reihenfolge die verbleibenden Knoten des Suchraums zu klassifizieren sind. In diesem Kapitel werden drei Suchstrategien vorgestellt: Top-Down (5.1), Divide-and-Query (5.2) und Divide-and-Query mit Überdeckungsinformationen (5.3). Im Abschnitt 5.4 wird die Effizienz dieser Strategien anhand einer empirischen Untersuchung verglichen. Der Abschnitt 5.5 bildet den Abschluss und fasst die wichtigsten Ergebnisse dieses Kapitels zusammen.

5.1 Top-Down

Bei der Top-Down-Strategie [14] entspricht die Reihenfolge der Klassifizierung der Knoten einer Preorder-Traversierung des verbleibenden Suchbereichs. Aus der Menge der unklassifizierten Knoten des Suchraums wird derjenige Knoten k, dessen Teilberechnung im Vergleich zu den übrigen Knoten den frühesten Startzeitpunkt $t_{start}(c_k)$ im Programmablauf aufweist, als nächstes klassifiziert. Das Ergebnis der Suche nach einem defekten Knoten ist in der Abbildung 5.1 dargestellt. Um den defekten Knoten 8 zu finden, sind insgesamt 5 Klassifikationen notwendig. Es wurden die Knoten 1, 2, 7, 8 und 9 klassifiziert, die Klassifizierung der Knoten 5 und 6 ergibt sich implizit aus der Klassifizierung von 2. Die Knoten sind mit v_i bzw. i_i beschriftet, wenn sie im i-ten Klassifizierungsschritt als valide bzw. infiziert klassifiziert wurden. Die Beschriftung t bedeutet, dass der Knoten vertrauenswürdig ist. Die Suche beginnt mit der Klassifikation der Wurzel des Baumes (1), welche als infiziert klassifiziert wird. Im nächsten Schritt besitzt der Knoten 2 von allen Knoten des verbleibenden Suchraums den frühesten Startzeitpunkt. Dieser wird als valide klassifiziert. Anschließend folgen die Knoten 7, 8 und 9, so dass am Ende der Suchraum nur noch den defekten Knoten 8 enthält.

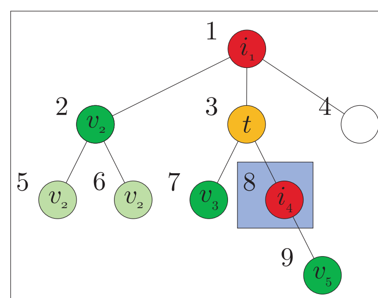


Abbildung 5.1: Ergebnis der Suche des defekten Knotens mit Top-Down-Strategie. Der defekte Knoten ist nach 5 Klassifikationen gefunden.

Wie dieses Beispiel verdeutlicht, folgt die Top-Down-Strategie der Ausführungsreihenfolge des untersuchten Programms. Bei der Klassifikation können Knoten zwar übersprungen werden, die Reihenfolge, in der die Knoten zu klassifizieren sind, ist jedoch immer chronologisch. Hierdurch wird in

der Regel die Klassifizierung der Knoten für den Benutzer vereinfacht, da aufeinanderfolgende Klassifikationen häufig einen engen zeitlichen und inhaltlichen Bezug zueinander aufweisen. Es fällt daher leichter, den Ablauf des Programms nachzuvollziehen und die Knoten zu klassifizieren.

Auf der anderen Seite hat die Top-Down-Strategie aber auch einen entscheidenden Nachteil: Sie berücksichtigt die Struktur des Berechnungsbaums nicht. Aus diesem Grund ist es möglich, dass ein vergleichsweise großer Anteil der Knoten eines Berechnungsbaums klassifiziert werden muss, bis der defekte Knoten gefunden ist. Diese Problematik tritt vor allem bei unausgeglichenen Bäumen auf. So müssen zum Beispiel im Berechnungsbaum der Abbildung 5.1 mehr als die Hälfte bzw. 5 von 9 der Knoten klassifiziert werden. Im Worst-Case würde aus dem Suchraum mit jeder Klassifizierung nur ein einziger Knoten entfernt. Der Suchaufwand wäre in diesem Fall in $O(|K|)$.

5.2 Divide-and-Query

Eine Debugging-Strategie, welche den Suchaufwand im Vergleich zur Top-Down-Strategie erheblich reduziert, ist die von Shapiro [130] entwickelte Divide-and-Query-Strategie (D&Q). D&Q verfolgt den Ansatz, die Größe des Suchraums mit jedem Klassifizierungsschritt zu halbieren.

Zu diesem Zweck werden zunächst das Gesamtgewicht des verbleibenden Suchraums S_i und die Gewichte aller Knoten von S_i bestimmt. Das Gesamtgewicht des Suchraums S_i ist definiert als die Anzahl der aller Knoten von S_i: $W_i := |S_i|$. Das Gewicht eines Knotens k nach dem i-ten Klassifizierungsschritt ist definiert als die Anzahl der Knoten, die sowohl Element des Suchbereichs S_i als auch Element des Teilbaums mit der Wurzel k, T_k, sind. Das heißt $w_i^k = |S_i \cap T_k|$.

Die D&Q-Strategie wählt für die Klassifikation immer den Knoten aus, dessen Gewicht am nächsten an der Hälfte des Gesamtgewichts des Suchraums liegt. Für die Klassifikation $(i+1)$ wird somit ein Knoten $m_i \in S_i$ gewählt, für den gilt:

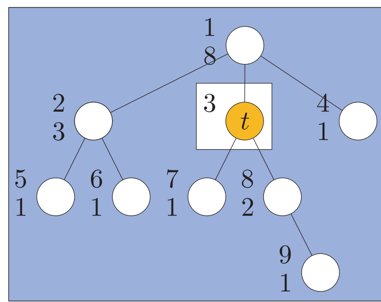
$$\left| w_i^{m_i} - \frac{W_i}{2} \right| = \min_{k \in S_i} \left| w_i^k - \frac{W_i}{2} \right|.$$

Dieser Knoten m_i zeichnet sich dadurch aus, dass die Differenz zwischen $w_i^{m_i}$, das ist die Anzahl der Knoten des Suchraums, die innerhalb des Teilbaums T_{m_i} liegen, und $W_i - w_i^{m_i}$, das ist die Anzahl der Knoten des Suchraums, die außerhalb des Teilbaumes T_{m_i} liegen, im Vergleich zu allen übrigen Knoten des Suchraums am kleinsten ist. Im Folgenden wird m_i daher auch als Mittelpunkt des Suchraums S_i bezeichnet.

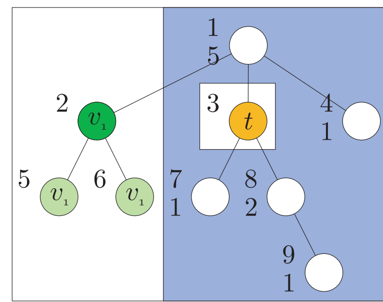
Für den Fall, dass der Mittelpunkt des Suchraums m_i als valide klassifiziert wird, gilt für das Gewicht des neuen Suchraums: $W_{i+1} = W_i - w_i(m_i)$. Sollte m_i als infiziert klassifiziert werden, dann gilt entsprechend: $W_{i+1} = w_i(m_i)$. Folglich gilt: Je kleiner die Differenz zwischen $w_i(m_i)$ und $\frac{W_i}{2}$, desto kleiner ist in beiden Fällen die Differenz zwischen W_{i+1} und $\frac{W_i}{2}$. Durch diese Strategie wird daher das Gewicht des Suchraums mit jedem Schritt unabhängig von der Art der Klassifikation annähernd halbiert.[7] Da bei der einfachen D&Q-Strategie das Gewicht der Anzahl der Knoten entspricht, wird folglich auch die Anzahl der Knoten des Suchraums mit jedem Klassifizierungsschritt annähernd halbiert.

Die Abbildung 5.2 veranschaulicht den Ablauf der D&Q-Strategie. Die Knoten sind nummeriert (obere, äußere Ziffer) und mit Gewichten versehen (untere, äußere Ziffer). Die Klassifizierung steht innerhalb des jeweiligen Knotens. Zu Beginn der Suche (Abbildung 5.2a) umfasst der Suchraum S_0 acht Knoten, dies sind alle Knoten des Suchraums bis auf den vertrauenswürdigen Knoten 3. Der Mittelpunkt von S_0 ist der Knoten 2 mit einem Gewicht von 3. Nachdem der Knoten 2 als valide klassifiziert wurde, enthält der in Abbildung 5.2b dargestellte Suchraum S_1 noch 5 Knoten. Der Mittelpunkt von S_1 ist der Knoten 8 mit einem Gewicht von 2. Nachdem 8 als infiziert klassifiziert wurde, enthält der Suchraum S_2 in Abbildung 5.2c noch 2 Knoten, von denen der Knoten 9 mit einem Gewicht von 1 den Mittelpunkt bildet. Wie in der Abbildung 5.2d dargestellt endet die Suche, nachdem Knoten 9 als valide klassifiziert wurde. S_3 enthält jetzt nur noch einen einzigen, infizierten Knoten. Dieser Knoten (8) ist der gesuchte defekte Knoten.

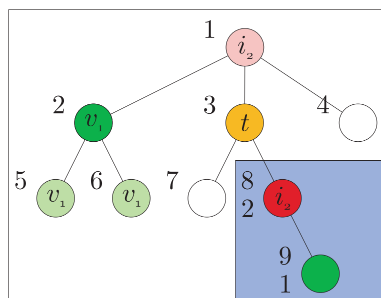
[7] Eine annähernde Halbierung des Suchraumes erreicht die D&Q-Methode ausschließlich bei den Klassifizierungen „valide“ und „infiziert“. Wird ein Knoten als vertrauenswürdig klassifiziert, so wird in der Regel keine Halbierung des Suchraums erreicht.



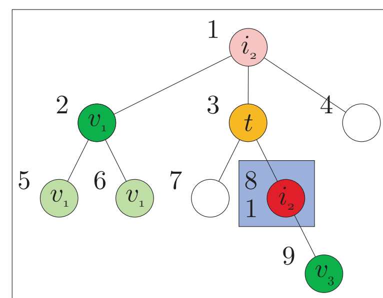
(a) Zu Beginn der Klassifizierung.



(b) Nach Klassifizierung des Knotens 2 als valide.



(c) Nach Klassifizierung des Knotens 8 als infiziert.



(d) Nach Klassifizierung des Knotens 9 als valide.

Abbildung 5.2: Suche des defekten Knotens mit der Divide-and-Query-Strategie. Zusätzlich zu der Nummerierung (oben) ist jeder Knoten mit seinem Gewicht (unten) und seiner Klassifizierung (Mitte) beschriftet. Nach 3 Klassifizierungen (2, 8 und 9) ist der defekte Knoten gefunden.

Unter der Annahme, dass die Infektionswahrscheinlichkeit eines Knotens proportional zu dessen Gewicht ist, lässt sich zeigen, dass die D&Q-Strategie optimal ist. Das heißt, es wird immer derjenige Knoten klassifiziert, für den das erwartete verbleibende Gewicht des Suchraumes minimal ist. Angenommen die Klassifizierung eines Knotens k sei ein Zufallsexperiment mit den beiden möglichen Ergebnissen „infiziert“ und „valide“. Das Gewicht des verbleibenden Suchraums nach der Klassifizierung des Knotens k kann als Zufallsvariable W_{i+1}^k folgendermaßen modelliert werden:[8]

$$W_{i+1}^k = \begin{cases} w_i^k & \text{, falls k infiziert} \\ W_i - w_i^k & \text{, falls k valide} \end{cases}$$

[8]Eine Klassifikation als „vertrauenswürdig“ wird an dieser Stelle nicht berücksichtigt.

Wenn p_i^k die Wahrscheinlichkeit ist, mit der k infiziert ist, dann ist folglich die Wahrscheinlichkeit mit der k valide ist $(1-p_i^k)$. Für den Erwartungswert von W_{i+1}^k gilt dann:

$$E(W_{i+1}^k) = w_i^k \cdot p_i^k + (W_i - w_i^k) \cdot (1 - p_i^k).$$

Aus der Annahme, dass die Infektionswahrscheinlichkeit eines Knotens, p_i^k, proportional zu dessen Gewicht w_i^k ist, folgt $p_i^k = \frac{w_i^k}{W_i}$. Somit gilt für den Erwartungswert:

$$E(W_{i+1}^k) = \frac{1}{W_i} \left((w_i^k)^2 + (W_i - w_i^k)^2 \right)$$

Durch Ableiten und Nullsetzen ergibt sich der Minimalwert für $E(W_{i+1}^k)$ an der Stelle $w_i^k = \frac{W_i}{2}$. Der minimale Erwartungswert für die Größe des verbleibenden Suchraums liegt somit bei der Klassifizierung des Knotens k vor, dessen Gewicht genau der Hälfte des Gewichts des gesamten Suchraums entspricht. In diesem Fall wäre die erwartete Größe des verbleibenden Suchbereichs $E(W_{i+1}^k) = \frac{W_{i+1}^k}{2}$. Da die D&Q-Strategie immer den Knoten wählt, dessen Gewicht die geringste Differenz zu $\frac{W_i}{2}$ aufweist, ist sie optimal in dem Sinne, dass sie die erwartete Größe des verbleibenden Suchraums minimiert. Sie gleicht somit einem Greedy-Algorithmus [37, S. 370–404], da sie immer denjenigen Konten wählt, dessen Klassifikation die größte Verkleinerung des Suchraums verspricht.

Die Abbildung 5.3 zeigt den Worst Case der D&Q-Methode für einen vollständigen Berechnungsbaum der Höhe h. Jeder innere Knoten des Baumes besitzt genau b Kindknoten, d. h., der Verzweigungsfaktor des Berechnungsbaums ist b. Im Worst Case müssen auf jeder Ebene des Baumes mit Ausnahme der Ebene 0 insgesamt b Knoten klassifiziert werden. Der Worst Case liegt dann vor, wenn der gesuchte defekte Knoten ein Blattknoten ist und für die Kindknoten der Wurzel und die Kindknoten aller infizierten Knoten gilt, dass der infizierte Kindknoten immer zuletzt, d. h. erst nach allen validen Kindknoten, klassifiziert wird. Die Gesamtzahl der Klassifikationen ist dann $h \cdot b$. Da für die Höhe des Berechnungsbaums gilt, dass $h = \log_b(|K|+1) - 1$ ist, ergibt sich $O(b \cdot \log_b |K|)$ als obere Schranke für den Klassifizierungsaufwand. Im Worst Case wird der Aufwand somit in hohem Maße von dem Verzweigungsfaktor des Berechnungsbaums beeinflusst. Für einen Berechnungsbaum mit der Höhe $h = 1$ und dem größtmöglichen Verzweigungsfaktor von $b = |K| - 1$ müssen im Worst Case alle Blattknoten klassifiziert

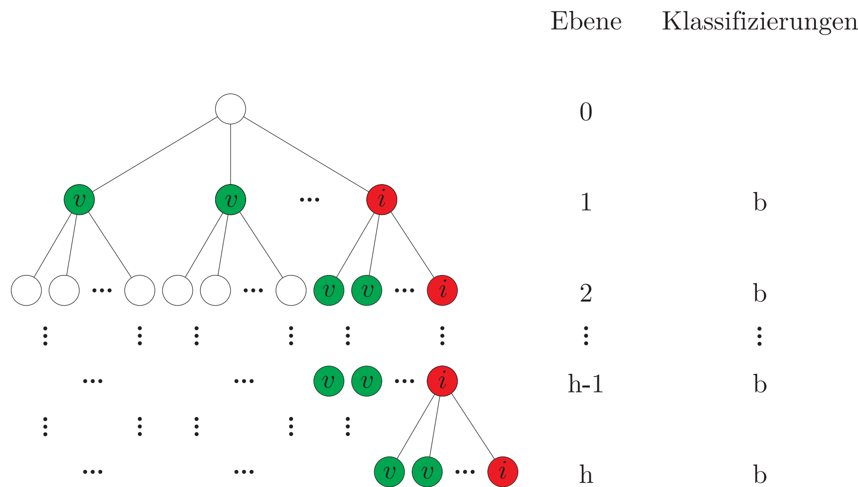


Abbildung 5.3: Worst Case der Divide-and-Query-Strategie.

werden, d. h., der Klassifizierungsaufwand liegt in $O(|K|)$. Für den kleinstmöglichen Verzweigungsfaktor 2 liegt der Aufwand in $O(\log_2 |K|)$. Für den hier beschriebenen Worst Case ist der Aufwand der D&Q-Strategie identisch mit dem Aufwand der Top-Down-Strategie, da die Reihenfolge der Klassifizierungen in einem vollständigen Berechnungsbaum, in dem jeder innere Knoten dieselbe Anzahl von Kindern hat, identisch ist. Auf die in der Praxis auftretenden Berechnungsbäume treffen diese Eigenschaften in der Regel jedoch nicht zu. Die Berechnungsbäume realer Programme sind meist unausgeglichen und der Verzweigungsfaktor der Teilbäume ist in der Regel nicht konstant. Gerade für diese Berechnungsbäume ist die D&Q-Strategie der Top-Down-Strategie im Hinblick auf die Anzahl der zu klassifizierenden Knoten überlegen. Der Worst Case der Top-Down-Strategie ist ein zu einer linearen Liste degenerierter maximal unausgeglichener Berechnungsbaum, dessen defekter Knoten das einzige Blatt des Baumes ist. In diesem Fall liegt der Klassifizierungsaufwand der Top-Down-Strategie in $O(|K|)$. Die D&Q-Strategie ist hier wesentlich effizienter, da die obere Schranke für den Klassifizierungsaufwand in $O(\log |K|)$ liegt.

Die D&Q-Strategie ist daher besonders für große und unausgeglichene Berechnungsbäume geeignet. In diesen Fällen ist der Suchaufwand erheblich geringer als bei der Top-Down-Strategie. Ein Nachteil dieser Methode ist die Tatsache, dass aufeinanderfolgende Klassifikationen in der Regel einen geringeren semantischen Bezug zueinander haben. Bei der Top-Down-Strategie ist dieser Bezug durch die Klassifizierung der Knoten in chronologischer

Reihenfolge größer. Aufgrund des geringeren semantischen Bezugs kann die Klassifizierung eines einzelnen Knotens bei der D&Q-Strategie mehr Zeit in Anspruch nehmen. Trotz dieses Nachteils überwiegt bei großen Berechnungsbäumen der Effekt der eingesparten Klassifikationen. Generell gilt, dass die D&Q-Strategie umso lohnenswerter ist, je größer und unausgeglichener der Berechnungsbaum ist.

5.3 Divide-and-Query mit gewichtsunabhängigen Infektionswahrscheinlichkeiten

5.3.1 Verfahren

Die D&Q-Strategie ist im Bezug auf den Suchaufwand effizienter als die Top-Down-Strategie, da sie die Struktur des Berechnungsbaums berücksichtigt. Bei der Wahl des nächsten Knotens geht die D&Q-Strategie allerdings davon aus, dass die Infektionswahrscheinlichkeit eines Knotens proportional zu dessen Gewicht ist. In der Regel sind die durch die Knoten repräsentierten Berechnungen jedoch von unterschiedlicher Komplexität. Aus diesem Grund ist davon auszugehen, dass Infektionswahrscheinlichkeit und Gewicht der Knoten nicht immer im gleichen Verhältnis zueinander stehen. Zum Beispiel ist die Wahrscheinlichkeit, einen Defekt zu enthalten, für einfache Zugriffsmethoden, mit denen die Attribute eines Objekts abgefragt oder geändert werden, in der Regel deutlich geringer als für Methoden, die komplexe Anwendungslogik enthalten.

Die auf gewichtsunabhängigen Infektionswahrscheinlichkeiten basierende erweiterte D&Q-Strategie (D&Q-Inf-Strategie) verfolgt ebenso wie die einfache D&Q-Strategie die Minimierung des Erwartungswertes für die Größe des verbleibenden Suchraums. Es wird somit immer derjenige Knoten k mit dem geringsten Erwartungswert $E(W_{i+1}^k)$ klassifiziert. Im Unterschied zur einfachen D&Q-Strategie wird jedoch nicht gefordert, dass $p_i^k = \frac{w_i^k}{W_i}$ gilt. Die Infektionswahrscheinlichkeit eines Knotens k ist somit eine unabhängige Variable und muss nicht dem relativen Gewicht des Knotens entsprechen. Der

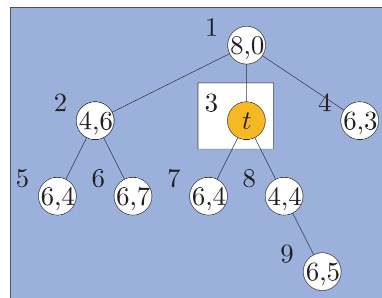
Erwartungswert für die Größe des verbleibenden Suchraums berechnet sich in diesem Fall wie bei der D&Q-Strategie durch:

$$E(W_{i+1}^k) = w_i^k \cdot p_i^k + (W_i - w_i^k) \cdot (1 - p_i^k).$$

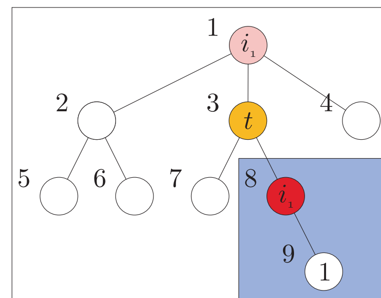
Allerdings kann die Formel hier nicht weiter vereinfacht werden, da w_i^k und p_i^k zwei unabhängige Variablen sind. Bei der D&Q-Inf-Strategie wird daher für jeden Knoten $k \in S_i$ der Erwartungswert $E(W_{i+1}^k)$ berechnet und anschließend der Knoten mit dem geringsten Erwartungswert klassifiziert.

Die beiden Strategien D&Q und D&Q-Inf sind somit im Wesentlichen identisch. Beide Methoden klassifizieren immer den Knoten mit dem geringsten Erwartungswert $E(W_{i+1}^k)$. Die D&Q-Inf-Strategie verwendet neben der Knotengewichte allerdings auch noch Infektionswahrscheinlichkeiten, um den Erwartungswert zu berechnen. Diese Erweiterung soll dazu beitragen, dass diese Strategie bezüglich des verbleibenden Suchaufwands günstigere Entscheidungen trifft und somit die Anzahl zu klassifizierender Knoten gegenüber der D&Q-Strategie verringert werden kann.

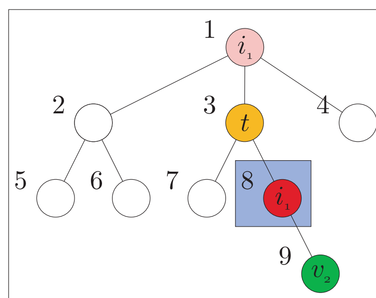
Die Abbildung 5.4 verdeutlicht den Ablauf der D&Q-Strategie mit Infektionswahrscheinlichkeiten. Die Strategie berechnet zunächst für jeden unklassifizierten Knoten des Suchraums den Erwartungswert $E(W_{i+1}^k)$. Der Berechnungsbaum vor dem 1. Klassifizierungsschritt ist in der Abbildung 5.4a dargestellt. Bis auf den Knoten 3, der vertrauenswürdig ist, sind alle Knoten Element des Suchraums. Jeder unklassifizierte Knoten k des Suchraums ist mit dem Erwartungswert $E(W_1^k)$ beschriftet. Die kleinste Größe des verbleibenden Suchraums wird bei der Klassifikation des Knotens 8 erwartet. Für die Infektionswahrscheinlichkeit von $p_1^8 = 0,4$ ergibt sich: $E(W_1^8) = 2 \cdot 0,4 + 6 \cdot 0,6 = 4,4$. Bei der einfachen D&Q-Strategie wäre der Knoten 2 gewählt worden, da dessen Gewicht (3) die geringste Differenz zu der Hälfte des Gesamtgewichts aufweist. Da der Knoten 2 in diesem Beispiel jedoch nur die Infektionswahrscheinlichkeit $p_1^2 = 0,2$ besitzt, beträgt der Erwartungswert $E(W_1^8) = 3 \cdot 0,2 + 5 \cdot 0,8 = 4,6$. Weil hier die Infektionswahrscheinlichkeiten von dem durch die einfache D&Q-Strategie vorgegebenen Verhältnis $p_i^k = w_i^k / W_i$ wie beschrieben abweichen, wird im 1. Schritt der Knoten 8 und nicht der Knoten 2 klassifiziert. Anschließend enthält der Suchbereich wie in der Abbildung 5.4b dargestellt nur noch 2 Knoten. Für den Fall, dass der Knoten 9 klassifiziert wird, ist der Erwartungswert $E(W_2^9)$ gleich 1, da die Größe des verbleibenden Suchraums



(a) Zu Beginn der Klassifizierung.



(b) Nach Klassifizierung des Knotens 8 als infiziert.



(c) Nach Klassifizierung des Knotens 9 als valide.

Abbildung 5.4: Suche des defekten Knotens mit der Divide-and-Query-Strategie mit gewichtsunabhängigen Infektionswahrscheinlichkeiten. Jeder unklassifizierte Knoten des Suchraums enthält die erwartete Größe des verbleibenden Suchraums für den Fall, dass der entsprechende Knoten klassifiziert wird. Nach 2 Klassifizierungen (8 und 9) ist der defekte Knoten gefunden.

unabhängig von der Art der Klassifikation immer gleich 1 ist. In diesem Fall ist der Knoten 9, wie in der Abbildung 5.4c dargestellt, valide und die Suche endet, da der einzige verbleibende Knoten des Suchbereichs infiziert ist. Dieser Knoten ist der gesuchte defekte Knoten. Durch die Erweiterung der D&Q-Strategie um gewichtsunabhängige Infektionswahrscheinlichkeiten konnte in diesem Beispiel die Anzahl der benötigten Klassifizierungen von 3 auf 2 reduziert werden.

In der Praxis kann die Infektionswahrscheinlichkeit einer Berechnung von einer Vielzahl unterschiedlicher Faktoren abhängen. Da diese Faktoren sowie deren Einflüsse auf die Infektionswahrscheinlichkeit in der Regel unbekannt sind, können sie nicht exakt berechnet werden. Aus diesem Grund bedient

sich die D&Q-Inf-Strategie einer einfachen Heuristik, um die Infektionswahrscheinlichkeit zu schätzen. Diese Heuristik basiert auf der Überlegung, dass die Infektionswahrscheinlichkeit einer Berechnung mit zunehmender Komplexität der Berechnung steigt.

Es stellt sich die Frage, wie sich die Komplexität einer Berechnung in geeigneter Weise schätzen lässt. Ein einfaches Komplexitätsmaß wäre zum Beispiel die Anzahl der Berechnungsschritte. Eine Berechnung wäre in diesem Fall umso komplexer, je mehr Berechnungsschritte sie umfasst. Das Problem bei dieser Heuristik ist, dass sie nicht robust gegenüber Schleifen ist. Bei der Ausführung einer Schleife kann bereits eine kleine Menge von Anweisungen eine große Anzahl an Berechnungsschritten produzieren. Da beim Durchlauf einer Schleife häufig dieselben Anweisungen ausgeführt werden, ist die Anzahl der Berechnungsschritte in diesem Fall kein geeignetes Maß für die Infektionswahrscheinlichkeit. Zum Beispiel wäre die Infektionswahrscheinlichkeit einer Schleife, die im ersten Fall 10-mal und im zweiten Fall 1000-mal durchlaufen würde, im zweiten Fall 100-mal höher als im ersten Fall. Dies wäre in den meisten Fällen eine eher unrealistische Annahme.

Im Bereich des Softwaretestens finden sich robustere Metriken mit denen sich auch die Ausführungskomplexität einer Berechnung schätzen lässt. Beim White-Box-Testen wird zum Beispiel die Vollständigkeit einer Menge von Testfällen anhand der Kontroll- oder Datenflüsse eines Programmablaufs gemessen. Je mehr Kontroll- oder Datenflüsse durch eine Menge von erfolgreichen Testfällen überdeckt werden, desto geringer ist die Wahrscheinlichkeit, dass das getestete Programm einen Defekt enthält. Das Ziel des White-Box-Testens ist es daher, eine möglichst hohe Überdeckung zu erreichen. Grundsätzlich gilt dabei, dass mit zunehmender Komplexität des Programms auch die Anzahl der möglichen Kontroll- oder Datenflüsse steigt. In der Regel steigt mit der Komplexität des zu testenden Programms auch die Anzahl an Testfällen, die benötigt wird, um eine gewünschte Überdeckung zu erreichen.

Aus der Korrelation zwischen der Programmkomplexität und der Anzahl der überdeckten Kontroll- oder Datenflüsse lässt sich eine Heuristik zur Schätzung der Ausführungskomplexität eines Programms ableiten. Demnach kann die Anzahl der durch eine Berechnung überdeckten Kontroll- bzw. Datenflüsse als Maßzahl für deren Komplexität dienen.

5.3.2 Kontrollflussgraph

Zur Bestimmung der durch die Ausführung eines Programms überdeckten Kontroll- bzw. Datenflüsse ist dessen Struktur von entscheidender Bedeutung. Die Abbildung 5.5 stellt die Struktur eines Programms durch einen Kontrollflussgraphen dar. Ein Kontrollflussgraph [98, S. 257–261] ist ein gerichteter Graph, der die Struktur eines Programms abbildet. Jeder Knoten des Kontrollflussgraphen repräsentiert einen Basisblock des Programmcodes. Ein Basisblock ist eine Folge von Anweisungen, die entweder vollständig oder gar nicht ausgeführt wird. Daraus folgt, dass jede Anweisung, die Ziel einer Sprunganweisung ist, den Beginn eines neuen Basisblocks darstellt, und jede Sprunganweisung das Ende eines Basisblocks markiert. Eine gerichtete Kante repräsentiert den Kontrollfluss zwischen zwei Basisblöcken. Darüber hinaus gibt es in jedem Kontrollflussgraphen einen Startknoten, an dem sämtliche Kontrollflüsse starten, und einen Endknoten, an dem sämtliche Kontrollflüsse enden.

5.3.3 Kontrollflussbasierte Infektionswahrscheinlichkeit

Die kontrollflussorientierten Tests basieren auf unterschiedlichen Strukturelementen des Kontrollflussgraphen. Zu diesen Strukturelementen zählen

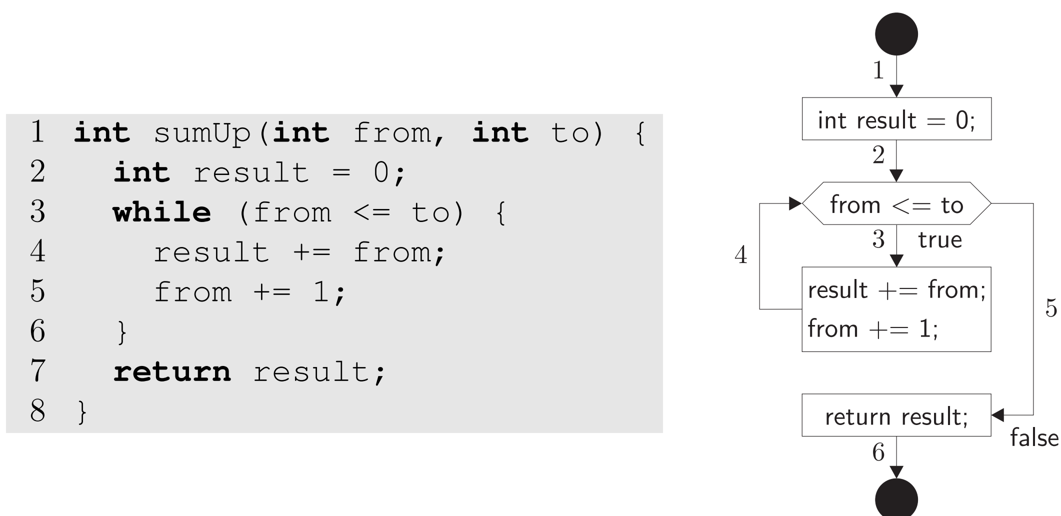


Abbildung 5.5: Methode `sumUp` mit Kontrollflussgraph.

die Knoten, Kanten, Bedingungen und Pfade des Kontrollflussgraphen. Entsprechend des Strukturelements, dessen Überdeckung getestet wird, lassen sich die kontrollflussorientierten Überdeckungstests in die folgenden in der Tabelle 5.1 dargestellten Klassen unterteilen:

- Der **Anweisungsüberdeckungstest** ist die schwächste kontrollflussorientierte Testmethode. Er fordert, dass jeder Knoten des Kontrollflussgraphen mindestens einmal ausgeführt wird. Dies ist gleichbedeutend mit einer vollständigen Überdeckung aller Anweisungen, weshalb dieser Test auch als Anweisungsüberdeckungstest bezeichnet wird. Grundsätzlich gilt das Kriterium dieses Tests als zu schwach, da er von allen Tests die geringste Identifizierungsquote für Defekte aufweist [58].

- Der **Zweigüberdeckungstest** definiert ein strengeres Testziel. Er fordert die Ausführung bzw. Überdeckung sämtlicher Zweige des Kontrollflussgraphen. Dies ist gleichbedeutend mit einer Überdeckung aller Kanten. Da aus der Überdeckung aller Kanten immer auch die Überdeckung aller Knoten des Kontrollflussgraphen folgt, ist der Anweisungsüberdeckungstest vollständig im Zweigüberdeckungstest enthalten. Der Zweigüberdeckungstest subsumiert somit den Anweisungsüberdeckungstest. Dieser Zusammenhang zeigt sich zum Beispiel am Kontrollflussgraphen der If-Anweisung in Abbildung 5.6b. Um vollständige Knoten- bzw. Anweisungsüberdeckung zu erreichen, genügt bereits ein einziger Testfall, bei dem die Bedingung der If-Anweisung

Tabelle 5.1: Kontrollflussorientierte Überdeckungstests nach [16, S.400–415].

Name	Ziel
Anweisungsüberdeckungstest	jede Anweisung wird mindestens einmal ausgeführt
Zweigüberdeckungstest	jede Kante des Kontrollflussgraphen mindestens einmal durchlaufen
Bedingungsüberdeckungstest	Überdeckung der Bedingungen der Kontrollstrukturen
Pfadüberdeckungstest	Überdeckung einer Teilmenge der möglichen Ausführungspfade

erfüllt ist. Für die vollständige Überdeckung aller Kanten bzw. Zweige der If-Anweisung werden zwei Testfälle benötigt: Ein Testfall, für den die Bedingung der If-Anweisung erfüllt ist, und ein Testfall, für den die Bedingung nicht erfüllt ist. In der Praxis ist der Zweigüberdeckungstest weit verbreitet und wird durch eine Vielzahl von Werkzeugen unterstützt [98, 88 f.].

- Der **Bedingungsüberdeckungstest** testet die logische Struktur von Bedingungen, die den Kontrollfluss steuern. Häufig besteht die Bedingung, welche den Kontrollfluss einer Verzweigung bestimmt, aus mehreren zusammengesetzten, u. U. hierarchisch gegliederten, Teilbedingungen. In diesem Fall werden durch den Zweigüberdeckungstest in der Regel nicht alle Teilbedingungen geprüft. Für den Zweigüberdeckungstest ist es unerheblich, welche der Teilbedingungen das Ergebnis einer Bedingung bestimmt haben. Für die If-Anweisung in Abbildung 5.6a würde ein Zweigüberdeckungstest unabhängig von der Komplexität der Bedingung ausschließlich berücksichtigen, ob die Bedingung zutrifft oder nicht. Wie sich anhand des erweiterten Kontrollflussgraphen für die Teilbedingungen zeigt, kann es dabei vorkommen, dass einzelne Teilbedingungen gar nicht getestet werden. Um die Bedingung zu erfüllen, würde es zum Beispiel ausreichen, wenn die Teilbedingungen `u==1` und `y<3` erfüllt sind. Die Bedingung wäre nicht erfüllt, falls `u==1` und `x>2` nicht erfüllt sind. Mit den beiden Testeingaben `(u=1, x=?, y=2, z=?)` und `(u=2, x=3, y=?, z=?)`, ließe sich somit vollständige Zweigüberdeckung erreichen. Die Fragezeichen deuten dabei an, dass der Wert der entsprechenden Variablen beliebig gewählt werden kann, da dieser das Ergebnis der Bedingung nicht beeinflusst. In beiden Testfällen wird das Ergebnis der Bedingung nicht durch die Variable `z` beeinflusst. Dies bedeutet, dass die Teilbedingung `z==4` das Ergebnis für keinen der Testfälle beeinflusst hat und diese Bedingung somit nicht getestet wurde. Diese Schwäche des Zweigüberdeckungstests wird durch den Bedingungsüberdeckungstest behoben. Es gibt verschiedene Formen des Bedingungsüberdeckungstests, die unterschiedliche Kriterien für die Überdeckung der Teilbedingungen festlegen. Der einfache Bedingungsüberdeckungstest fordert zum Beispiel, dass jede Teilbedingung mindestens einmal als zutreffend und mindestens einmal als nicht zutreffend ausgewertet werden muss. Werden die Bedingungen nach

dem Short-Circuit-Verfahren ausgewertet, dann subsumiert der einfache Bedingungsüberdeckungstest den Zweigüberdeckungstest. Bei der Short-Circuit-Auswertung werden Teilbedingungen nur solange ausgewertet, wie das Gesamtergebnis der Bedingung noch nicht feststeht. Diese unvollständige Auswertung der If-Anweisung aus Abbildung 5.6a ist zum Beispiel durch den Kontrollflussgraphen der Abbildung 5.6c dargestellt. Wie dem Kontrollflussgraphen zu entnehmen ist, wird die Auswertung bestimmter Teilbedingungen übersprungen, wenn sie das Gesamtergebnis nicht mehr beeinflussen können. Da die einfache Bedingungsüberdeckung fordert, dass jede Teilbedingung mindestens einmal als zutreffend und mindestens einmal als nicht zutreffend ausgewertet wird, müssen bei einer Short-Circuit-Auswertung sämtliche Kanten des Kontrollflussgraphen aus Abbildung 5.6c überdeckt werden. Daraus folgt automatisch auch die Überdeckung sämtlicher Kanten des Kontrollflussgraphen aus Abbildung 5.6b. Wird eine Bedingung jedoch immer vollständig ausgewertet, so lässt sich für die untersuchte If-Anweisung bereits mit den beiden Testfällen `(u=0, x=3, y=3, z=4)` und `(u=1, x=2, y=2, z=3)` eine vollständige Überdeckung aller Teilbedingungen erzielen. Jede Teilbedingung würde in diesem Fall einmal als zutreffend und einmal als nicht zutreffend ausgewertet werden. Die gesamte Bedingung würde jedoch bei beiden Testfällen als zutreffend ausgewertet. Somit wären zwar die Kriterien der einfachen Bedingungsüberdeckung, jedoch nicht die Kriterien der Zweigüberdeckung erfüllt. Weitere Verfahren, wie der minimale Mehrfach-Bedingungsüberdeckungstest und der Mehrfach-Bedingungsüberdeckungstest definieren strengere Überdeckungskriterien. Auf eine genauere Beschreibung dieser Verfahren wird an dieser Stelle verzichtet und auf die Darstellungen von Balzert [16, 408–412] und Liggesmeyer [98, 89–112] verwiesen.

- **Pfadüberdeckungstests** definieren eine bestimmte Menge von Ausführungspfaden des Kontrollflussgraphen, die zu überdecken sind. Ein Ausführungspfad ist dabei ein vollständiger Pfad vom Startknoten bis zum Endknoten des Graphen. Ein wünschenswertes Kriterium wäre beispielsweise die Überdeckung aller möglichen Pfade des Kontrollflussgraphen. Die Anzahl der möglichen Pfade eines Kontrollflussgraphen kann allerdings vor allem durch die Präsenz von Schleifen extrem hoch sein. Oftmals gibt es sogar eine unendliche Anzahl an

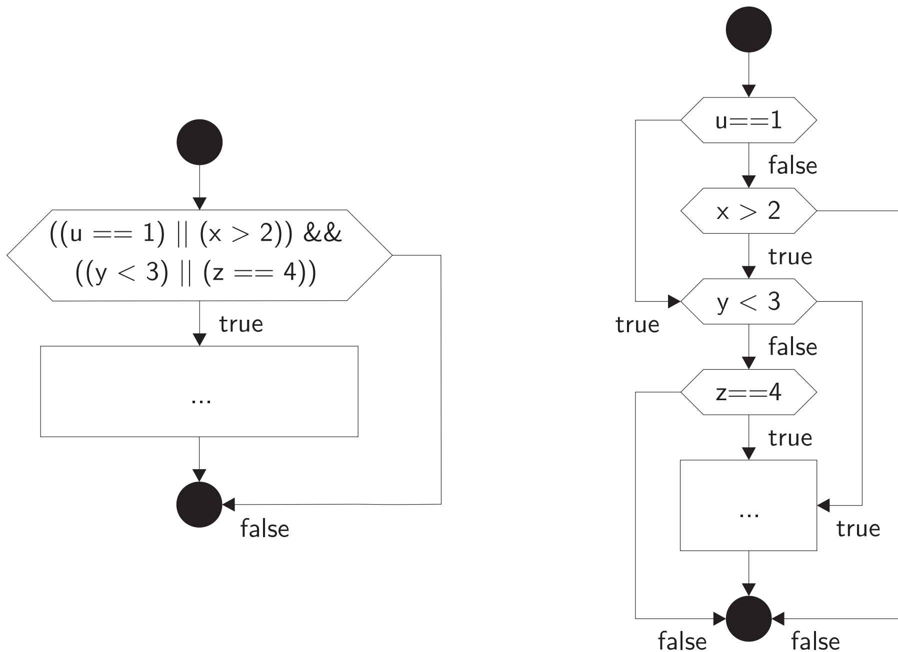
möglichen Pfaden. Die Überdeckung aller Pfade lässt sich daher in der Regel nicht realisieren. Aus diesem Grund gibt es Verfahren, wie den strukturierten Pfadtest und den Boundary-Interior-Pfadtest [98, S. 113–127], die eine Obergrenze für die maximale Anzahl an Schleifenwiederholungen festlegen, um so die Zahl der möglichen Pfade zu beschränken. Die Zweig- und Bedingungsüberdeckungstests besitzen eine entscheidende Schwachstelle: Sie testen jede Entscheidung bzw. Bedingung isoliert, d. h. losgelöst vom Kontext des Kontrollflusses des restlichen Programms. Sie prüfen somit ausschließlich, *ob* ein Strukturelement überdeckt wird. Die Reihenfolge der überdeckten Elemente wird dabei nicht berücksichtigt. Mitunter kann die Reihenfolge aber entscheidend für das Verursachen einer Infektion sein. Die Pfadüberdeckungstests berücksichtigen die Reihenfolge der überdeckten Kanten des Kontrollflussgraphen und besitzen somit grundsätzlich das Potenzial eine größere Menge von Defekten zu erkennen. In der Praxis sind Pfadüberdeckungstests jedoch weniger weit verbreitet, da deren Durchführung in der Regel mit hohem Aufwand verbunden ist [98, S. 132–135]. Der zusätzliche Testaufwand wird daher in der Regel nur bei extrem sicherheitskritischen Komponenten betrieben.

Die kontrollflussorientierte Heuristik zur Bestimmung der Ausführungskomplexität einer Berechnung sollte einerseits den Kontrollfluss möglichst exakt erfassen, andererseits aber auch robust gegenüber Schleifen sein. Eine Metrik, die die Komplexität einer Berechnung anhand der Länge des Ausführungspfades misst, führt zu einer sehr starken Gewichtung von Schleifenwiederholungen, auch wenn bei den Wiederholungen keine neuen Strukturelemente des Kontrollflussgraphen überdeckt werden. Metriken, die die Reihenfolge der überdeckten Strukturelemente nicht berücksichtigen, sind gegenüber häufigen Schleifenwiederholungen sehr viel robuster. Von diesen Metriken weist die Bedingungsüberdeckung die höchste Genauigkeit auf. Im Falle einer Short-Circuit-Auswertung subsumiert sie die übrigen von der Reihenfolge der Überdeckung unabhängigen Tests, wie den Anweisungs- und den Zweigüberdeckungstest.

Aus diesen Gründen wurde die Bedingungsüberdeckung gewählt, um die kontrollflussorientierte Komplexität einer Berechnung zu messen. Die Komplexität und damit auch die geschätzte Infektionswahrscheinlichkeit ist dann

```
1 if (((u == 1) || (x > 2)) &&
2     ((y < 3) || (z == 4))) {
3   ...
4 }
```

(a) If-Anweisung mit mehreren logisch verknüpften Teilbedingungen.



(b) Kontrollflussgraph der If-Anweisung.

(c) Kontrollflussgraph der If-Anweisung mit Teilbedingungen.

Abbildung 5.6: If-Anweisung mit Kontrollflussgraph.

umso größer, je mehr Bedingungen während ihrer Ausführung überdeckt wurden.

5.3.4 Datenflussorientierte Infektionswahrscheinlichkeit

Neben den kontrollflussorientierten Tests bilden die datenflussorientierten Tests die zweite Klasse strukturorientierter Test. Analog zu den kontrollflussorientierten Tests wird bei den datenflussorientierten Tests der Überdeckungsgrad der möglichen Datenflüsse der zu testenden Komponente gemessen. Auch für die datenflussorientierten Tests gilt: Je höher der durch eine Menge von Testfällen erzielte Überdeckungsgrad, desto geringer ist die

Infektionswahrscheinlichkeit der getesteten Komponente. Im Mittelpunkt der datenflussorientierten Tests stehen Zugriffe auf Variablen. Während der Ausführung eines Testfalls werden die Datenflüsse in Form von Variablenzugriffen aufgezeichnet. Die Vollständigkeit eines Tests wird durch den von den Testfällen erreichten Überdeckungsgrad der Variablenzugriffe gemessen. Datenflussorientierte Tests eignen sich besonders zum Testen von datenzentrierten Programmen. Aus diesem Grund sind sie zum Beispiel gut für Modul-Tests in der objektorientierten Softwareentwicklung geeignet [98, S. 138].

Die Grundlage der Datenflussanalyse bilden Definition-Use-Ketten (DU-Ketten). Eine DU-Kette [116, S. 50 ff.] [5, S. 632] beschreibt für eine Variable X eine Anweisung, die der Variablen X einen Wert zuweist, und eine Anweisung, die den Wert der Variablen X ausliest. Wenn S eine Anweisung ist, $def(S)$ die Menge der Variablen, denen durch die Anweisung S ein Wert zugewiesen wird, und $use(S)$ die Menge der Variablen, deren Werte von S ausgelesen werden, dann ist eine DU-Kette ein Tripel (X, S, S') mit

- einer Variablen X,
- einer definierenden Anweisung $S : X \in def(S)$ und
- einer lesenden Anweisung $S' : X \in use(S')$
- mit einem definitionsfreien Ausführungspfad von S zu S', auf dem für jedes S'', welches zwischen S und S' ausgeführt wird, gilt, dass $X \notin def(S'')$.

Dabei ist zu beachten, dass die definierende Anweisung S der lesenden Anweisung S' im Quelltext nicht unbedingt textlich vorausgehen muss. Zum Beispiel kann beim Durchlauf einer Schleife die Position S' im Quelltext eines Programms auch kleiner sein als die Position von S. Die möglichen DU-Ketten eines Programms lassen sich zum Beispiel mithilfe eines um Datenflussattribute erweiterten Kontrollflussgraphen [98, S. 140] ermitteln.

Die Abbildung 5.7 zeigt alle DU-Ketten der Methode `sumUp`. Die Menge der überdeckten DU-Ketten ist vom Ausführungspfad der Methode und damit von der Wahl der Argumente `from` und `to` abhängig. Für `from > to` wird der Rumpf der Schleife gar nicht, für `from == to` einmal und für `from < to` mindestens zweimal ausgeführt. Entsprechend ergeben sich für diese drei Fälle die in der Abbildung 5.7b dargestellten Mengen D_1, D_2

```
1 int sumUp(int from, int to) {
2    int result = 0;
3    while (from <= to) {
4      result = result + from;
5      from = from + 1;
6    }
7    return result;
8 }
```

(a) Quelltext der Methode `sumUp`.

- `from > to`:
  $D_1 =\{$ `[from,1,3]`, `[to,1,3]`, `[result,2,7]` $\}$
- `from == to`:
  $D_2 =(D_1 \setminus \{$ `[result,2,7]` $\}) \cup \{$ `[from,1,4]`, `[from,1,5]`, `[from,5,3]` $\}$
- `from < to`:
  $D_3 = D_2 \cup \{$ `[from,5,4]`, `[from,5,5]`, `[result,4,4]` $\}$

(b) DU-Ketten der Methode `sumUp`.

Abbildung 5.7: Methode `sumUp` mit möglichen Definition-Use-Ketten.

und D_3 von überdeckten DU-Ketten. Da die Menge D_2 eine Teilmenge von D_3 ist, kann mit 2 Testfällen, für die `from > to` bzw. `from < to` gilt, die Menge aller DU-Ketten überdeckt werden.

Für ein Programm lässt sich eine theoretische Obergrenze aller möglichen DU-Ketten ermitteln. Die Frage, ob eine DU-Kette durch eine konkrete Ausführung tatsächlich erreichbar ist, kann jedoch unentscheidbar sein [5, S. 624–633]. Daher gibt es unterschiedlich strenge Überdeckungskriterien für datenflussorientierte Strukturtestverfahren. Eine detaillierte Beschreibung dieser Kriterien findet sich zum Beispiel bei Balzert [16, S. 421–425] und Liggesmeyer [98, S. 141–157]. In einer Studie [58] konnten mit der Hilfe von datenflussorientierten Tests bis zu 70 Prozent der Programmfehler gefunden werden.

DU-Ketten können zur Messung der datenflussorientierten Komplexität einer Programmausführung verwendet werden. Dabei gilt: Je größer die An-

zahl der überdeckten DU-Ketten, desto größer die Datenflusskomplexität einer Programmausführung. Die Überdeckung von DU-Ketten zeichnet sich ebenso wie die Bedingungsüberdeckung dadurch aus, dass sie robust gegenüber Schleifenwiederholungen ist. In der Regel ist davon auszugehen, dass die Anzahl an zusätzlich überdeckten DU-Ketten mit jeder weiteren Schleifenwiederholung geringer wird. Eine Schleifenwiederholung führt daher nur dann zu einer höheren Komplexität, wenn zusätzliche DU-Ketten überdeckt werden.

Aufgrund der Datenzentrierung eignet sich dieses Verfahren jedoch hauptsächlich für Programmiersprachen, die ebenfalls eine hohe Datenzentrierung aufweisen. Zu diesen zählen vor allem die imperativen und objektorientierten Sprachen. Für die Programmiersprache Java ist die Anzahl der überdeckten DU-Ketten daher eine geeignete Metrik zur Messung der Ausführungskomplexität.

5.3.5 Schätzung der Infektionswahrscheinlichkeit

Für die Erweiterung der D&Q-Strategie werden zusätzliche Informationen über die Ausführung der einzelnen Knoten des Berechnungsbaums benötigt. Die Bedingungsüberdeckung und die Überdeckung der DU-Ketten sind Kennzahlen, durch die auf die Ausführungskomplexität eines Knotens bzw. einer Berechnung geschlossen werden kann. Beide Kennzahlen sind daher geeignete Heuristiken, mit denen die Infektionswahrscheinlichkeit eines Knotens geschätzt werden kann. Für das im Folgenden beschriebene Verfahren kann daher die Ausführungskomplexität entweder durch die überdeckten Bedingungen oder die überdeckten DU-Ketten bestimmt werden. Um von der Art der verwendeten Überdeckung zu abstrahieren, werden die Bedingungen bzw. DU-Ketten im Folgenden allgemein als Entitäten bezeichnet. Die Menge der Entitäten, die durch einen Knoten überdeckt werden, bezeichnet somit entweder die Menge der überdeckten Bedingungen oder die Menge der überdeckten DU-Ketten.

Neben der Ausführungskomplexität stehen aber noch weitere Informationen zur Verfügung, die zur Berechnung der Infektionswahrscheinlichkeiten verwendet werden können. Diese Informationen können aus der Klassifizierungshistorie des Benutzers abgeleitet werden. Wird zum Beispiel ein

Knoten des Berechnungsbaums $a \in K$, der den Aufruf einer Methode m repräsentiert als valide klassifiziert, so können die aus der Klassifizierung gewonnenen Informationen dazu verwendet werden, die Infektionswahrscheinlichkeiten der Menge der Knoten, die einen Aufruf derselben Methoden m repräsentieren, anzupassen. Wenn a als valide klassifiziert wird, dann folgt daraus, dass die durch a überdeckte Menge von Entitäten E_a keine Infektion produziert hat. Diese Information kann dazu benutzt werden, die Infektionswahrscheinlichkeit eines Knotens b, für den gilt $E_b \cap E_a \neq \emptyset$, zu verringern. Dieses Vorgehen lässt sich damit Begründen, dass eine Entität e, welche als valide klassifiziert wurde, mit geringerer Wahrscheinlichkeit eine Infektion produziert als eine Entität, welche bisher noch nicht als valide klassifiziert wurde.

Der Einfluss der Entitäten auf die Infektionswahrscheinlichkeit, der sich im Verlauf des Debugging-Prozesses ändert, kann durch die Funktion $h_i : E \to \mathbb{R}^+$ ausgedrückt werden. Dabei bezeichnet E die Menge aller Entitäten, die durch den zu untersuchenden Programmablauf überdeckt wurden. Die Funktion h_i ordnet somit jeder überdeckten Entität nach dem i-ten Klassifizierungsschritt einen Infektionswert zu. Je größer der Infektionswert einer Entität, desto größer ist ihr Einfluss auf die Infektionswahrscheinlichkeit.

Ausgehend von dem Infektionswert einer einzelnen Entität kann der Infektionswert eines Knotens berechnet werden. Der Infektionswert eines Knotens k bezeichnet den Einfluss von k auf die Infektionswahrscheinlichkeit und bezieht sich ausschließlich auf den Knoten k und nicht auf dessen Kindknoten. Der Infektionswert eines Knotens k nach der i-ten Klassifikation wird durch die Funktion $g_i : K \to \mathbb{R}$ abgebildet und entspricht der Summe der Infektionswerte aller von k überdeckten Entitäten:

$$g_i(k) = \sum_{e \in E_k} h_i(e).$$

Aus den Infektionswerten der Knoten des Suchraums lassen sich die Infektionswahrscheinlichkeiten der Knoten berechnen. Die Infektionswahrscheinlichkeit eines Knotens k ist definiert als:

$$p_i^k = \frac{1}{\sum_{a \in S_i} g_i(a)} \cdot \sum_{a \in (S_i \cap T_k)} g_i(a).$$

Sie berechnet sich somit aus der Summe der Infektionswerte der Knoten, die sowohl Element des Suchraums S_i als auch Element des Teilbaums T_k

sind. Durch die Normierung mit der Summe der Infektionswerte aller Knoten des Suchraums S_i ergibt sich die Infektionswahrscheinlichkeit p_i^k. In die Berechnung der Infektionswahrscheinlichkeit eines Knotens k fließen neben dem Infektionswert von k auch die Infektionswerte aller Nachfahren von k ein, da sich die Infektion eines Knotens immer auf dessen Vorfahren überträgt.[9] Ein Knoten k ist schließlich auch dann infiziert, wenn nicht er selbst, sondern einer seiner Nachkommen eine Infektion verursacht hat.

Um die Infektionswahrscheinlichkeiten p_i^k berechnen zu können, muss eine geeignete Funktion h_i zur Bestimmung der Infektionswerte der Entitäten festgelegt werden. Der Infektionswert einer Entität e ist dabei von der Anzahl der validen Klassifizierungen von e abhängig. Bei steigender Anzahl an validen Klassifizierungen von e sollte der Infektionswert der Entität e sinken. Sei $V_i \subset K$ die Menge Knoten des Berechnungsbaums, die nach i Klassifizierungsschritten als valide klassifiziert wurden. Ferner sei $v : E \rightarrow \mathbb{N}_0$ die Funktion, welche für eine Entität e die Häufigkeit zurückgibt, mit der e als valide klassifiziert wurde. Die Funktion v_i ist definiert als:

$$v_i(e) = \left|\{a \in V_i : e \in E_a\}\right|.$$

Um den Einfluss der validen Klassifikationen auf den Infektionswert in geeigneter Weise durch die Funktion h_i abzubilden, sollte h_i im Bezug auf die Anzahl der validen Klassifikationen $v_i(e)$ einer Entität e monoton fallend sein. Ferner sollten zwei Entitäten $e_1, e_2 \in E$, für die $v_i(e_1) = v_i(e_2)$ gilt, denselben Infektionswert besitzen, d. h., es sollte $h_i(e_1) = h_i(e_2)$ gelten. Die geforderten Eigenschaften von h_i sind sehr allgemein gehalten und wenig restriktiv, so dass h_i durch unterschiedlichste Klassen von Funktionen realisiert werden könnte. Die Eignung einer konkreten Funktion h_i lässt sich nur mithilfe von empirischen Untersuchungen bestimmen. Für die Umsetzung der erweiterten D&Q-Strategie mit gewichtsunabhängigen Infektionswahrscheinlichkeiten wurde die folgende Funktion gewählt:

$$h_i(e) = q^{v_i(e)}, \text{ mit } q \in [0, 1].$$

Diese Exponentialfunktion bewertet für $q = 1$ jede Entität unabhängig von $v_i(e)$ mit 1. In diesem Fall hätte $v_i(e)$ keinen Einfluss auf den Infektionswert. Für $q = 0$ wird jede Entität e, für die $v_i(e) = 0$ gilt, mit 1 und jede

[9] Infektionen, die sich nicht auf die Vorfahren eines Knotens übertragen und im Laufe der Programmausführung wieder verschwinden werden hier nicht berücksichtigt, da sie nicht zu einer Fehlerwirkung führen. Für diese Knoten wird kein Debugging-Prozess angestoßen, da sie nicht erkannt werden.

Entität mit $v_i(e) > 0$ mit 0 bewertet. In diesem Fall ist somit der Infektionswert der Entitäten, die mindestens einmal als valide klassifiziert wurden, gleich 0 und $v_i(e)$ hat hier den größtmöglichen Einfluss auf den Infektionswert. Grundsätzlich gilt für diese Funktion, dass der Einfluss von v_i auf den Infektionswert umso größer ist, je kleiner q ist. Für $q = 1/2$ halbiert sich zum Beispiel der Infektionswert einer Entität e mit jeder weiteren validen Klassifizierung eines Knotens k für den $e \in E_k$ gilt.

Im Gegensatz zur einfachen D&Q-Strategie berücksichtigt das hier vorgestellte, erweiterte Verfahren nicht nur die Struktur des Berechnungsbaums, sondern unterscheidet auch zwischen Knoten mit unterschiedlicher Komplexität und berücksichtigt ebenfalls bereits vorgenommene Klassifizierungen. Diese beiden zusätzlich berücksichtigten Informationen werden dazu verwendet, für die Knoten des Berechnungsbaums Infektionswahrscheinlichkeiten zu berechnen. Im Unterschied zur einfachen D&Q-Strategie stehen die Infektionswahrscheinlichkeiten p_i^k eines Knotens k bei dieser Strategie in keinem festen Verhältnis zum Gewicht w_i^k des Knotens k. Für einen Knoten k des Suchraums sollte diese Strategie somit $E(W_{i+1}^k)$, d. h. die erwartete Größe des verbleibenden Suchraums bei Klassifizierung von k, in der Regel exakter bestimmen können. Unter der Voraussetzung, dass diese Hypothese zutrifft, würde die D&Q-Inf-Strategie im Hinblick auf den Klassifizierungsaufwand in der Regel eine bessere Entscheidung bei der Auswahl der zu klassifizierenden Knoten treffen und es wäre davon auszugehen, dass der Aufwand der erweiterten D&Q-Strategie im Mittel deutlich geringer ausfällt.

Bei der erweiterte D&Q-Strategie haben ausschließlich die validen Klassifizierungen V_i einen Einfluss auf die Infektionswerte der Entitäten bzw. auf den Funktionswert von h_i. Grundsätzlich wäre es durchaus denkbar, dass der Funktionswert von h_i auch von der als infiziert klassifizierten Menge der Knoten beeinflusst wird. Für einen infizierten Knoten k ist es jedoch schwierig, exakte Auswirkungen auf die Infektionswerte der überdeckten Entitäten E_k festzulegen. Das Problem sind die Informationen über die überdeckten Entitäten, die aus einer Klassifizierung abgeleitet werden können. Wird ein Knoten k als valide klassifiziert, dann folgt, dass alle überdeckten Entitäten valide sind, da ihre Überdeckung keine Infektion produziert hat. Wird jedoch ein Knoten k als infiziert klassifiziert, dann lässt sich daraus nicht folgern, dass eine bestimmte Entität $e \in E_k$ valide oder infiziert ist. Hierfür müsste zunächst bekannt sein, ob der infizierte Knoten auch der defekte Knoten

ist. Nur wenn es sich bei dem infizierten Knoten k um einen defekten Knoten handelt, sind die Ausführung von k und damit auch die Überdeckung der Entitäten E_k Verursacher der Infektion. Der deklarative Debugging-Prozess endet allerdings mit der Identifizierung des defekten Knotens. Zu diesem Zeitpunkt werden die Informationen, die sich aus der Identifizierung des Defekten Knoten ableiten lassen, allerdings nicht mehr benötigt. Aus diesem Grund wurde für h_i eine Funktion gewählt, auf die die Menge der infizierten Knoten keinen Einfluss hat.

Darüber hinaus ist die Anzahl infizierter Klassifizierungen im Vergleich zur Anzahl valider Klassifizierungen in der Regel sehr gering. In einem Berechnungsbaum, der einen einzigen defekten Knoten enthält, sind ausschließlich die Knoten auf dem Pfad von der Wurzel zum defekten Knoten infiziert. Aus diesem Grund entspricht die maximale Anzahl infizierter Knoten der Höhe des Berechnungsbaums plus 1. Da alle nicht infizierten Knoten valide sind, ist die Anzahl der validen Knoten im Allgemeinen um ein Vielfaches größer als die Anzahl der infizierten Knoten. Daher ist üblicherweise nur ein kleiner Anteil der klassifizierten Knoten infiziert. Die Tatsache, dass infizierte Knoten nicht zur Anpassung der Infektionswerte verwendet werden, fällt somit kaum ins Gewicht.

5.4 Empirische Untersuchung

Um Aussagen über die Effizienz der vorgestellten deklarativen Debugging-Strategien treffen zu können, wurden diese durch eine empirische Untersuchung miteinander verglichen. Zu diesem Zweck wurde eine Menge von 6 Programmen gewählt, welche unterschiedliche Algorithmen oder Datenstrukturen implementieren. Für jedes dieser Programme, welches ursprünglich keine Defekte enthielt, wurde eine Menge von Testfällen generiert. Ein einzelner Testfall wurde erzeugt, indem an einer zufällig ausgewählten Stelle durch Manipulation des defektfreien Quelltextes ein Defekt verursacht wurde. Insgesamt wurden auf diese Weise 32 Testfälle mit unterschiedlichen Arten von Defekten erzeugt. Wie im Abschnitt 3.2 erläutert werden im Rahmen dieser Arbeit zwei Defektarten unterschieden. Ein Defekt kann entweder aus einer Menge von fehlerhaften Anweisungen oder einer Men-

ge von fehlenden Anweisungen bestehen. Beide Defektarten wurden bei der Erzeugung der Testfälle berücksichtigt.

Im Rahmen der Untersuchung sollten die Top-Down-Strategie (TD), die einfache D&Q-Strategie (D&Q), die erweiterte D&Q-Strategie, die auf der Überdeckung von DU-Ketten basiert (D&Q-D), und die erweiterten D&Q-Strategie, die auf der Bedingungsüberdeckung basiert (D&Q-B), miteinander verglichen werden. Zu Beginn der Untersuchung wurde für die Strategien D&Q-D und D&Q-B jeweils der optimale Parameter q für die im Abschnitt 5.3.5 bestimmt. Für D&Q-D ist die Anzahl der benötigten Klassifikationen zum Durchlauf aller Testfälle für $q = 0,55$ minimal. Die D&Q-D-Strategie ist somit optimal, wenn der Einfluss einer überdeckten DU-Kette auf die Infektionswahrscheinlichkeit mit jeder validen Klassifizierung ungefähr halbiert wird. Für D&Q-B ist der Aufwand bei $q = 0,35$ minimal. In diesem Fall wird der Einfluss einer überdeckten Bedingung auf die Infektionswahrscheinlichkeit mit jeder validen Klassifikation ungefähr gedrittelt.

Um die Effizienz der Debugging-Strategien miteinander zu vergleichen, wurde für alle Testfälle mithilfe der deklarativen Debugging-Technik die defekte Methode bestimmt. Dabei wurde für jeden Testfall die Suche nach der defekten Methode insgesamt 4-mal durchgeführt, wobei jedes Mal eine andere deklarative Debugging-Strategie verwendet wurde. Im Anschluss wurde jeden Durchlauf des deklarativen Debugging-Prozesses die Anzahl der Klassifizierungen notiert, die zur Identifikation der defekten Methode benötigt wurden. Die Auswertung der Ergebnisse dieser Untersuchung ist in den Tabellen 5.2 und 5.3 dargestellt.

Die Tabelle 5.2 zeigt den Vergleich der vier Debugging-Strategien bezüglich der absoluten Anzahl an Klassifikationen, die benötigt wurden, um die defekte Methode zu identifizieren. Die fünf unterschiedlichen Programme, die verwendet wurden, um die Effizienz der Strategien zu vergleichen, sind in der ersten Spalte der Tabelle aufgeführt. Die zweite Spalte gibt an, wie viele Testfälle jeweils nach dem oben beschriebenen Verfahren generiert wurden. Die dritte Spalte gibt den Mittelwert der Größe des Suchraums über alle Testfälle des entsprechenden Programms zu Beginn des Debugging-Prozesses an. Der Suchraums umfasst dabei ausschließlich die Knoten des Berechnungsbaums, die zu Beginn des Debugging-Prozesses unklassifiziert sind. Die durch die Testfälle erzeugten Berechnungsbäume sind in der Regel größer, da sie auch vertrauenswürdige Knoten enthalten. So sind zum

Beispiel alle Knoten, die einen Aufruf einer Methode der Java-API [145] repräsentieren, automatisch von Beginn an vertrauenswürdig. Die Spalten 4 bis 7 enthalten für jede der vier Debugging-Strategien die entsprechende Anzahl an Klassifikationen, die zur Identifikation des defekten Knotens im Mittel über alle Testfälle eines Programms benötigt wurden.

Die Top-Down-Strategie schneidet mit durchschnittlich 12,84 Klassifikationen im Vergleich zu den übrigen Strategien am schlechtesten ab. Mit durchschnittlich 10 Klassifikationen über alle Testfälle folgt die einfache D&Q-Strategie auf dem zweiten Platz. Sie ist den beiden erweiterten D&Q-Strategien mit Ausnahme der Testfälle des Heapsort-Programms in allen Fällen unterlegen. Von den beiden erweiterten D&Q-Strategien ist die auf Bedingungsüberdeckung basierende Strategie im Mittel etwas effizienter als die auf der Überdeckung von DU-Ketten basierende Strategie. Die D&Q-D-Strategie ist der D&Q-B-Strategie nur beim Debugging der Binärbaum-Testfälle knapp überlegen. In allen anderen Fällen ist die auf Bedingungsüberdeckung basierende erweiterte D&Q-Strategie effizienter.

Tabelle 5.2: Absoluter Effizienzvergleich deklarativer Debugging-Strategien.

Programm	Anzahl der Testfälle	Durchschn. Größe des Suchraums	Durchschn. Anzahl der Klassifikationen TD	D&Q	D&Q-D	D&Q-B
Avl-Baum	10	67,5	13,90	9,10	7,70	6,60
Binärbaum	5	44,0	11,80	10,20	7,40	7,80
B-Baum	7	77,0	15,57	13,43	8,57	9,43
Heapsort	5	42,2	9,60	7,20	9,00	9,00
Hashtabelle	5	51,0	11,20	9,60	9,00	5,80
Durchschnitt	6,4	56,3	12,84	10,00	8,25	7,66

Die Tabelle 5.3 zeigt einen relativen Effizienzvergleich der vier Debugging-Strategien. Die Werte sind das Ergebnis einer Verdichtung der in der Tabelle 5.2 dargestellten absoluten Messwerte. Die Spalten 2–5 enthalten den jeweiligen Anteil der Methoden des Suchraums, die zur Identifikation des defekten Knotens klassifiziert werden mussten. Dieser berechnet sich aus der durchschnittlichen Anzahl an Klassifikationen (Tabelle 5.2, Spalten 4–7) dividiert

Tabelle 5.3: Relativer Effizienzvergleich deklarativer Debugging-Strategien.

Programm	Anteil klassifizierter Methodenaufrufe [%]				Einsparungen gegenüber TD [%]		
	TD	D&Q	D&Q-D	D&Q-B	D&Q	D&Q-D	D&Q-B
Avl-Baum	21	13	11	10	35	45	53
Binärbaum	27	23	17	18	14	37	34
B-Baum	20	17	11	12	14	45	39
Heapsort	23	17	21	21	25	6	6
Hashtabelle	22	19	18	11	14	20	48
Durchschnitt	23	18	15	14	22	36	40

durch die mittlere Größe des Suchraums. Die Kennzahl gibt somit an, wie groß der Anteil des Suchraums ist, der inspiziert werden musste, um den defekten Knoten zu finden. Bei der schlechtesten Strategie müssen im Mittel über alle Testfälle 23 Prozent des Suchraums untersucht werden. Für die einfache D&Q-Strategie müssen nur 18 Prozent und für die beiden erweiterten D&Q-Strategien 15 bzw. 14 Prozent der Knoten des Suchraums klassifiziert werden. In den letzten drei Spalten der Tabelle 5.3 ist dargestellt, wie viel Prozent der Klassifizierungen mit der entsprechenden Strategie im Vergleich zur Top-Down-Strategie eingespart werden können. Die Einsparungen berechnen sich aus den Werten der Spalten 4–7 der Tabelle 5.2. Dabei wird jeweils die Differenz zwischen der Anzahl der Klassifikationen für TD und der entsprechenden Strategie berechnet und das Ergebnis durch die Anzahl der Klassifikationen von TD geteilt. Zum Beispiel ergibt sich die Einsparung der D&Q-Strategie für die Testfälle des AVL-Baums aus $\frac{13{,}9-9{,}1}{13{,}9} \approx 0,35$. Im Durchschnitt über alle Testfälle können mit der einfachen D&Q-Strategie 22 Prozent der Klassifikationen eingespart werden. Durch den Einsatz der erweiterten, überdeckungsbasierten D&Q-Strategien können sogar 36 bzw. 40 Prozent der Klassifizierungen eingespart werden. Für diese Testfälle kann durch die Erweiterung der D&Q-Strategie der eingesparte Klassifikationsaufwand nahezu verdoppelt werden. Diese Einsparungen tragen dazu bei, dass der defekte Knoten schneller gefunden werden kann. Sie reduzieren somit den Aufwand des deklarativen Debugging erheblich.

Zur Optimierung der einfachen D&Q-Strategie werden zusätzliche Informationen über den Berechnungsbaum zur Auswahl des zu klassifizierenden Knotens herangezogen. Diese zusätzlichen Informationen werden aus der

gemessenen Ausführungskomplexität des untersuchten Programms und den bisherigen Klassifikationen gewonnen. Die während der Ausführung des Programms aufgezeichneten Überdeckungsinformationen werden dazu verwendet, variierende Komplexitäten bzw. Infektionswahrscheinlichkeiten für die Knoten des Berechnungsbaums zu bestimmen. Anschließend werden die aus der Klassifikation der Knoten gewonnenen Informationen dazu verwendet, die Infektionswahrscheinlichkeiten während des Debugging-Prozesses anzupassen. Wie durch diese Untersuchung gezeigt wird, lässt sich durch diese zusätzlichen Informationen in der Regel eine bessere Entscheidung bei der Auswahl des zu klassifizierenden Knotens treffen.

5.5 Fazit

In diesem Kapitel wurden zunächst zwei bekannte deklarative Debugging-Strategien vorgestellt, die Top-Down-Strategie und die D&Q-Strategie. Die D&Q-Strategie wurde anschließend in den folgenden Punkten erweitert und verbessert:

Gewichtsunabhängige Infektionswahrscheinlichkeiten - Um Infektionswahrscheinlichkeiten zu bestimmen werden die durch einen Methodenaufruf überdeckten Bedingungen oder DU-Ketten bestimmt. Je mehr Entitäten ein Methodenaufruf überdeckt, desto größer ist seine Infektionswahrscheinlichkeit. Im Unterschied zur einfachen D&Q-Strategie ist die Infektionswahrscheinlichkeit bei diesem Ansatz nicht mehr proportional zum Gewicht eines Knotens. Sie gibt damit die tatsächliche Komplexität eines Methodenaufrufs exakter wieder und ermöglicht es bei der Auswahl des zu klassifizierenden Methodenaufrufs in der Regel eine bessere Entscheidung zu treffen.

Berücksichtigung vorangegangener Klassifikationen - Des Weiteren werden auch die vorangegangenen Klassifikationen bei der Bestimmung der Infektionswahrscheinlichkeiten berücksichtigt. Je häufiger die Überdeckung einer Entität e nicht zu einer Infektion geführt hat, d. h. je größer $v_i(e)$, desto geringer ist die Wahrscheinlichkeit, dass weitere Überdeckungen von e einen Fehler produzieren. Dies wird berücksichtigt, indem der Einfluss von e sinkt, je größer $v_i(e)$ ist.

Berechnung von Erwartungswerten - Aus dem Gewicht und der Infektionswahrscheinlichkeit wird der Erwartungswert eines Knotens k berechnet. Der Erwartungswert entspricht der erwarteten Größe des Suchraums für den Fall, dass k klassifiziert wird. Die erweiterte D&Q-Strategie folgt dem Prinzip eines Greedy-Algorithmus und klassifiziert in jedem Schritt den Knoten mit dem geringsten Erwartungswert.

Zuletzt wurden die drei Strategien Top-Down, D&Q und erweitertes D&Q in einer empirischen Untersuchung miteinander verglichen. Jede der Strategien wurde in insgesamt 32 Testfällen zur Defektsuche eingesetzt. Im Vergleich zur Top-Down-Strategie konnten mit der einfachen D&Q-Strategie 22 Prozent und mit der erweiterten D&Q-Strategie bis zu 40 Prozent der Klassifikationen eingespart werden. Durch die Erweiterungen konnten die Einsparungen der D&Q-Strategie somit fast verdoppelt werden.

Kapitel 6

Der Java-Hybrid-Debugger

Inhalt

6.1 Benutzeroberfläche 113
6.2 Debugging eines defekten Java-Programms . 128
6.3 Fazit 144

In diesem Kapitel wird JHyde vorgestellt, ein Debugger, der die in den vorangegangenen Kapiteln erläuterte hybride Debugging-Methode für Java Programme umsetzt. Zunächst werden im Abschnitt 6.1 der Aufbau und die Funktionsweise der Benutzeroberfläche von JHyde beschrieben. Im Abschnitt 6.2 wird dann anhand eines Beispiels der praktische Einsatz von JHyde demonstriert und gezeigt, wie mithilfe von JHyde die Defektsuche durchgeführt wird. Den Abschluss bildet das Fazit in Abschnitt 6.3.

6.1 Benutzeroberfläche

Die Abbildung 6.1 zeigt die Benutzeroberfläche von JHyde. Die Oberfläche besteht im Wesentlichen aus vier Ansichten, die in der Eclipse-Umgebung als Views bezeichnet werden. Die Ansichten erweitern die Debug-Perspektive der Eclipse IDE, welche in der Standardkonfiguration dazu verwendet wird, Java-Programme mittels eines Trace-Debuggers zu analysieren.

Abbildung 6.1: Benutzeroberfläche von JHyde. Die Benutzeroberfläche ist ein Eclipse-Plugin und besteht aus vier Ansichten, mit denen die Ausführung eines untersuchten Programms inspiziert werden können.

Grundsätzlich können mithilfe von Eclipse Java-Klassen, die die Methode `static void main(String[] args)` implementieren, in zwei unterschiedlichen Modi in einer externen Java-VM ausgeführt werden. Die externe Java-VM kann entweder im normalen Modus oder im Debugging-Modus gestartet werden. Im normalen Modus wird das Programm bis zum Ende ohne Unterbrechungen ausgeführt. Wird das untersuchte Programm im Debugging-Modus ausgeführt, dann ist es möglich über die *Java Platform Debugger Architecture* (JPDA) [141] den Programmablauf zu unterbrechen und den Zustandsraum des untersuchten Programms zu analysieren. Im Debugging-Modus nutzt die Eclipse IDE die JPDA, um die Funktionalitäten eines Trace-Debuggers umzusetzen. Die Eclipse IDE implementiert folglich nur die Benutzeroberfläche des Trace-Debuggers.

Um den hybriden Debugging-Prozess starten zu können, erweitert JHyde die Eclipse IDE um eine weitere Ausführungsmöglichkeit für Java-Klassen. Neben dem einfachen Modus und dem Debugging-Modus können Java-Programme nach der Installation von JHyde auch im JHyde-Modus ausgeführt werden. In diesem Modus wird das untersuchte Programm zunächst vollständig in einer externen Java-VM ausgeführt. Während der Ausführungen werden alle von JHyde benötigten Informationen über den Programmablauf aufgezeichnet. Nach Beendigung der Ausführung wird das Programm in der Debug-Perspektive dargestellt. Mithilfe der im Folgenden beschriebenen Ansichten kann dann der hybride Debugging-Prozess durchgeführt werden.

6.1.1 Berechnungsbaumansicht

In der Berechnungsbaumansicht wird der Berechnungsbaum des Programmablaufs dargestellt. Die Abbildung 6.2 zeigt den Berechnungsbaum für die defekte Mergesort-Implementierung aus Listing 6.1 (Abschnitt 6.2). Die Wurzel des Berechnungsbaums repräsentiert den Aufruf der Methode `MergeSort.main`. Teilbäume lassen sich an ihren Wurzelknoten mithilfe der Symbole „+“ bzw. „-“ aus- bzw. einklappen. Vor jedem Knoten gibt es ein Symbol, welches den Zugriffsmodifizierer der aufgerufenen Methode repräsentiert. Die Symbole Kreis, Dreieck, Raute und Quadrat stehen für die Modifizierer `public`, `protected`, `package private` und `private`. Methoden mit dem Modifizierer `static` oder `final` werden zusätzlich mit einem „S“ bzw. „F“ im Modifizierersymbol gekennzeichnet.

Die Berechnungsbaumansicht besitzt in der Werkzeugleiste mehrere Schaltflächen, mit denen die folgenden Aktionen durchgeführt werden können:

- Mithilfe der vier Schaltflächen der ersten Gruppe kann der selektierte Knoten bzw. der selektierte Methodenaufruf klassifiziert werden. Dabei klassifizieren die Symbole „Kreuz“, „Stern“, „Fragezeichen“ und „Haken“ einen Methodenaufruf als infiziert, vertrauenswürdig, unbekannt bzw. valide. Die Klassifizierungen der Methodenaufrufe werden in der Darstellung des Berechnungsbaums farblich und textlich dargestellt. Entsprechend der Klassifizierung ist die Hintergrundfarbe ent-

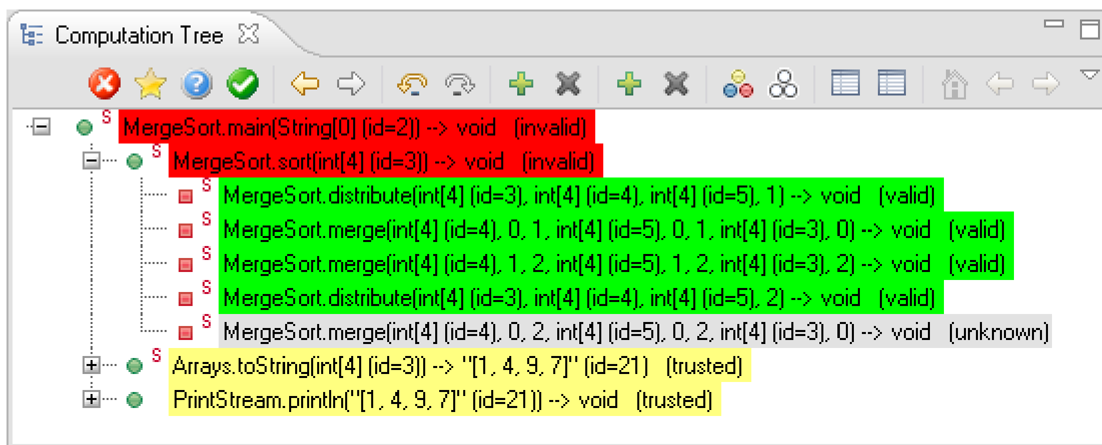


Abbildung 6.2: Berechnungsbaumansicht der JHyde Benutzeroberfläche.

weder Rot, Gelb, Weiß oder Grün. Zusätzlich steht die Klassifizierung noch ausgeschrieben in Klammern hinter jedem Methodenaufruf.

- Die beiden Pfeil-Schaltflächen der zweiten Gruppe von Werkzeugleistensymbolen dienen dazu, die Historie der selektierten Knoten zu navigieren. Durch Betätigen der entsprechenden Schaltfläche kann der zuvor bzw. nachfolgend selektierte Knoten ausgewählt werden.

- Mit den beiden Pfeil-Schaltern der dritten Gruppe können Klassifizierungen rückgängig gemacht oder wiederhergestellt werden.

- Über die Schaltflächen der vierten Gruppe kann eine Debugging-Strategie ausgewählt werden. Es ist möglich zwischen Top-Down, Divide-and-Query, Divide-and-Query mit Überdeckung von DU-Ketten und Divide-and-Query mit Bedingungsüberdeckung zu wählen. Nach der Klassifizierung eines Methodenaufrufs wird die ausgewählte Debugging-Strategie verwendet, um einen neuen, bisher noch nicht klassifizierten Knoten zu selektieren. Eine detailliertere Beschreibung der einzelnen Strategien findet sich in Abschnitt 5.

- Durch Betätigung der Schaltfläche der fünften Gruppe öffnet sich ein Dialog, in dem statistische Informationen über den Zustand des gegenwärtig untersuchten Berechnungsbaums angezeigt werden. Im Wesentlichen wird dabei für jede Klasse (infiziert, valide, usw.) die Anzahl der Methodenaufrufe angegeben, die der entsprechenden Klasse zugeordnet sind.

- Die Symbole der letzten Gruppe dienen dazu, die Berechnungsbaum-Ansicht auf Teilbäume des gesamten Berechnungsbaums zu beschränken. Bei größeren Berechnungsbäumen können damit irrelevante Teile des Berechnungsbaums ausgeblendet werden.

Die Berechnungsbaumansicht stellt den Programmablauf auf Ebene der Methodenaufrufe dar und ermöglicht es den Berechnungsbaum zu navigieren, indem Teilbäume aus- bzw. eingeklappt werden. Wird ein Knoten des Berechnungsbaums selektiert, so wird die Implementierung der Methode im Quelltexteditor von Eclipse angezeigt. Auf diese Weise vermittelt die Berechnungsbaumansicht einen abstrakten Überblick über den Ablauf des untersuchten Programms. Darüber hinaus unterstützt sie einen teilautomatisierten, deklarativen Debugging-Prozess, der dazu führt, dass nach der Klassifikation eines Methodenaufrufs in Abhängigkeit von der gewählten Debugging-Strategie ein neuer, nicht klassifizierter Knoten selektiert wird. Während des Debugging-Prozesses müssen die durch JHyde selektierten Knoten solange klassifiziert werden, bis JHyde den defekten Methodenaufruf identifiziert hat. Der Benutzer kann während des Prozesses jedoch von der durch JHyde vorgeschlagenen Klassifizierungsreihenfolge abweichen und einen Methodenaufruf seiner Wahl klassifizieren. Auf diese Weise bietet JHyde die Möglichkeit, den deklarativen Debugging-Prozess zu beschleunigen. Wenn der Benutzer aufgrund seiner Kenntnis des Programmablaufs die Menge der potenziell defekten Methodenaufrufe enger einschränken kann als JHyde, so kann der diese Kenntnis nutzen und einen selbst gewählten Methodenaufruf klassifizieren, um die Defektsuche zu beschleunigen.

6.1.2 Knotenansicht

Die Knotenansicht dient zur Analyse des Zustandsraums des untersuchten Programms. Mit ihr ist es möglich, die Werte einer Teilmenge von Variablen des Zustandsraums zu zwei unterschiedlichen Zeitpunkten, einem Startzeitpunkt und einem Endzeitpunkt, darzustellen und miteinander zu vergleichen. Die Knotenansicht zeigt den Wert jeder dargestellten Variablen zum Startzeitpunkt und zum Endzeitpunkt.

Wie in der Abbildung 6.3 dargestellt, entspricht die Knotenansicht einer Tabelle, in der jede Zeile eine einzelne Variable repräsentiert. In den drei

Spalten der Tabelle werden der Name, der Startwert, dies ist der Wert der Variablen zum definierten Startzeitpunkt, und der Endwert, dies ist der Wert der Variablen zum definierten Endzeitpunkt, angezeigt. Handelt es sich bei den Werten einer Variablen um Referenzen auf ein Objekt oder ein Array, dann werden die Attribute bzw. Elemente des referenzierten Objekts als Kindknoten der referenzierenden Variablen dargestellt. Die Kindknoten einer Variablen lassen sich durch die Schaltflächen „+“ bzw. „-“ ausklappen bzw. einklappen. In der Abbildung 6.3 sind zum Beispiel die Variablen „return“ und „this“ Referenzen auf Objekte der Klasse „Element“. Beide Variablen verfügen über weitere Kindknoten, von denen jedoch nur die Kindknoten der Variablen „this“ ausgeklappt sind. Die Kindknoten hängen vom Start- und Endwert der referenzierenden Variablen ab. Sind beide Werte „null“, so hat die referenzierende Variable keine Kindknoten. Ein Beispiel hierfür ist die Variable „succ“ in Zeile 8 der Knotenansicht. Sind Start- und Endwert *identisch* und *ungleich* `null`, dann entsprechen die Kindknoten den Attributen bzw. Elementen des referenzierten Objekts. Zum Beispiel repräsentieren die Kinder der Variablen „this“ die Attribute „value“ und „succ“ des `Element`-Objekts mit der Objekt-ID 3. Sind Start- und Endwert einer referenzierenden Variablen *ungleich* und *nicht gleich* `null`, so hat die Variable genau 2 Kindknoten. Diese heißen „<start value>“ und „<end value>“ und repräsentieren den Startwert bzw. den Endwert der Variablen. Die Kindknoten von „<start value>“ entsprechen dann den Attributen des Objekts, welches durch den Startwert referenziert wird und die Kindknoten von „<end value>“ entsprechend den Attributen des Objekts, welches durch den Endwert referenziert wird. In der Abbildung 6.3 hat sich beispielsweise der Wert der Variablen „succ“ in Zeile 4 geändert. Da Start- und Endwert unterschiedliche Objekte referenzieren, besitzt die Variable einen Kindknoten für den Startwert und einen Kindknoten für den Endwert.

Die Knotenansicht bildet referenzierte Objekte in einer Art Baumstruktur ab, indem die Attribute eines referenzierten Objekts als Kindknoten der referenzierenden Variablen dargestellt werden. Somit befinden sich alle direkt oder indirekt durch eine Variable v referenzierten Objekte im Teilbaum mit der Wurzel v. Eine Baumstruktur hat den Vorteil, dass sie sich leichter darstellen lässt als ein Graph. Aus diesem Grund wählen die meisten Debugger diese Methode, um Objektgraphen in einer Benutzeroberfläche darzustellen. Da ein Baum allerdings keine Zyklen besitzt, lassen sich nicht alle Eigen-

Node

Name	Start value	End value
return	null	Element (id=3)
this	Element (id=3)	Element (id=3)
value	1	1
succ	Element (id=4)	Element (id=5)
<start value>	Element (id=4)	Element (id=4)
<end value>	Element (id=5)	Element (id=5)
value	0	3
succ	null	null
a	3	3

Abbildung 6.3: Knotenansicht der JHyde Benutzeroberfläche.

schaften des Objektgraphen durch einen Baum abbilden. Wird ein Zyklus nach dem oben beschriebenen Ansatz in eine Baumstruktur überführt, so ergibt sich ein Baum mit unendlicher Höhe. Grundsätzlich sind Teilbäume unendlicher Größe unproblematisch, da einerseits die Teilbäume durch die Knotenansicht erst erzeugt werden müssen, wenn sie benötigt werden, und andererseits die Teilbaumkonstruktion vollständig ist in dem Sinne, dass für jede Variable alle direkt oder indirekt referenzierten Objekte erfasst werden. Für den Benutzer sind die durch eine Baumstruktur abgebildeten Zyklen jedoch schwerer zu erkennen, da sie nur anhand der IDs der referenzierten Objekte identifiziert werden können.

Durch die Knotenansicht werden alle Änderungen des Zustandsraums, die zwischen dem Startzeitpunkt und dem Endzeitpunkt stattgefunden haben, farblich hervorgehoben. Die Hintergrundfarbe einer Variablen wird nach den folgenden Regeln geändert:

- Eine Variable wird rot markiert, wenn sich ihr Wert zwischen Start- und Endzeitpunkt geändert hat. Entscheidend ist dabei, ob der Variablen im definierten Zeitraum ein Wert zugewiesen wurde. Die Art des zugewiesenen Wertes ist unerheblich. Aus diesem Grund kann eine Variable auch rot markiert sein, wenn ihr Startwert und ihr Endwert identisch sind. Dies ist der Fall, wenn der letzte vor dem Endzeitpunkt zugewiesene Wert dem Startwert der Variablen entspricht.

- Eine Variable v wird gelb markiert, wenn nicht ihr eigener Wert, sondern der Wert mindestens einer der Variablen des Teilbaums mit der Wurzel v zwischen Start- und Endzeitpunkt geändert wurde. Für eine

gelb markierte Variable hat sich somit nicht ihr eigener Wert, sondern der Wert einer Variablen eines direkt oder indirekt referenzierten Objekts geändert.

Die Knotenansicht ist an die Berechnungsbaumansicht gekoppelt. Daher führt die Selektion eines Methodenaufrufs c in der Berechnungsbaumansicht dazu, dass in der Knotenansicht alle lokalen Variablen des Methodenaufrufs und alle statischen Variablen angezeigt werden. Darüber hinaus werden Start- und Endzeitpunkt der Knotenansicht auf den Start- bzw. Endzeitpunkt der Ausführung des Methodenaufrufs, d. h. $t_{start}(c)$ und $t_{end}(c)$, gesetzt. Dies hat zur Folge, dass in der Knotenansicht genau alle Werte der Zustandsmenge $z_c\big(t_{start}(c), t_{end}(c)\big)$ (vgl. Abschnitt 4.3.2.4) angezeigt werden. Damit stellt die Knotenansicht alle Informationen zur Verfügung, die benötigt werden, um den in der Berechnungsbaumansicht selektierten Methodenaufruf zu klassifizieren. Durch die farbliche Hervorhebung aller Variablen, die zwischen den Zeitpunkten t_{start} und t_{end} eine Wertänderung erfahren haben, sind die Auswirkungen des untersuchten Methodenaufrufs auf den Programmzustand schneller zu erkennen und die Klassifizierung der Methodenaufrufe wird beschleunigt.

Neben der Kopplung an die Berechnungsbaumansicht ist die Knotenansicht auch an die Ereignisansicht gebunden (vgl. Abschnitt 6.1.3). Wird in der Ereignisansicht ein Ereignis e mit dem Zeitstempel t_e selektiert, so führt dies zu einer Anpassung des Endzeitpunkts in der Knotenansicht. Der Wert des Endzeitpunkts entspricht dann dem Zeitstempel t_e. Dies hat zur Folge, dass die Knotenansicht die Zustandsmenge $z_c\big(t_{start}(c), t_e\big)$ für den Methodenaufruf c anzeigt, in dem das selektierte Ereignis e aufgetreten ist. Die Knotenansicht stellt dann die Zustandsänderungen aller Ereignisse dar, die zwischen dem Beginn des Methodenaufrufs c und der vollständigen Ausführung des selektierten Ereignisses e liegen. Aufgrund ihrer gleichzeitigen Bindung an die Berechnungsbaum- und die Ereignisansicht wird die Knotenansicht sowohl für das deklarative Debugging als auch für das Omniscient-Debugging eingesetzt. In beiden Debugging-Prozessen dient sie dazu, durch den Programmablauf verursachte Zustandsänderungen darzustellen.

6.1.3 Ereignisansicht

Die in der Abbildung 6.4 dargestellte Ereignisansicht dient zur Durchführung des Omniscient-Debugging. Die Ereignisansicht ist an die Berechnungsbaumansicht gekoppelt. Wenn ein Methodenaufruf c in der Berechnungsbaumansicht selektiert wird, dann zeigt die Ereignisansicht alle Ereignisse an, die während der Ausführung von c aufgezeichnet wurden. Die Ereignisansicht ermöglicht es, die exakte Position eines Defekts im Quelltext zu identifiziert. Hierfür muss in der Ereignisliste eines defekten Methodenaufrufs das Ereignis gefunden werden, welches das Programm erstmalig von einem validen in einen infizierten Zustand überführt.

Event List

Event	Timestamp	Line
new Element (id=3)	6	17
⊞ Element.<init>(1) --> void	7	17
e=Element (id=3)	15	17
new Element[3] (id=4)	16	18
elements=Element[3] (id=4)	17	18
i=0	18	19
branch	19	19
⊟ loop (3)	20	20
⊟ iteration 1	21	20
branch	22	20
⊟ Element.insert(0) --> Element (id=3)	23	21
new Element (id=5)	29	12
⊞ Element.<init>(0) --> void	30	12
this.succ=Element (id=5)	38	12
return=Element (id=3)	39	13
return Element (id=3)	40	13
i=1	41	19
branch	42	19
⊞ iteration 2	43	20
⊞ iteration 3	65	20
return void	87	29

Abbildung 6.4: Ereignisansicht der JHyde Benutzeroberfläche.

Mit der Ereignisansicht ist es möglich, die einzelnen Ausführungsschritte innerhalb eines Methodenaufrufs zu untersuchen. Die Ansicht besteht aus drei Spalten, in denen für jedes Ereignis das Ereignis selbst („Event“), die Zeile im Quelltext („Line“), die das Ereignis verursacht hat, und der Zeitstempel

des Ereignisses („Timestamp") dargestellt sind. Der Zeitstempel dient dazu, die Folge von Ereignissen, die während der Ausführung des untersuchten Programms aufgezeichnet wird, zu indizieren. Während der Aufzeichnung des Programmablaufs bekommt jedes Ereignis einen eindeutigen Zeitstempel zugeordnet. Über den Zeitstempel werden die Ereignisse in zeitlicher Reihenfolge geordnet. Dabei gilt für zwei Ereignisse e_1 und e_2, dass der Zeitstempel t_{e_1} von e_1 genau dann kleiner ist als t_{e_2} von e_2, wenn das Ereignis e_1 zeitlich vor dem Ereignis e_2 aufgetreten ist.

Ereignistypen

Grundsätzlich gibt es zwei Gruppen von Ereignissen, die in der Ereignisansicht dargestellt werden: Kontrollflussereignisse und Datenflussereignisse. Zu den Kontrollflussereignissen zählen:

Methodenaufrufe - Methodenaufrufe werden durch den vollqualifizierten Namen der aufgerufenen Methode, die übergebenen Argumente und den Rückgabewert dargestellt. In der Abbildung 6.4 repräsentieren die Ereignisse mit den Zeitstempeln (timestamp) 7 und 30 den Aufruf des Konstruktors eines `Element`-Objekts. Das Ereignis mit dem Zeitstempel 23 repräsentiert den Aufruf der `insert`-Methode eines Element-Objekts. Jeder Methodenaufruf ist die Wurzel eines Teilbaums, der sich aus und einklappen lässt. Der Teilbaum enthält alle Ereignisse, die während der Ausführung des entsprechenden Methodenaufrufs aufgetreten sind.

Verzweigungen - Zu den Verzweigungen zählen mit Ausnahme der Schleifen-Ereignisse alle Ereignisse, die eine Verzweigung im Kontrollfluss repräsentieren. Diese Ereignisse werden verursacht durch `if`- und `switch`-Anweisungen sowie durch den Fragezeichenoperator. In der Ereignisansicht werden solche Verzweigungen durch das Schlüsselwort „branch" gekennzeichnet. In der Abbildung 6.4 repräsentieren die Ereignisse mit den Zeitstempeln 19, 22 und 42 eine Verzweigung.

Schleifen - Schleifen mit einer hohen Anzahl an Durchläufen können eine große Anzahl an Ereignissen produzieren, die in der Ereignisansicht dargestellt werden müssen. Würden diese als einfache Sequenz von Ereignissen in der Ereignisansicht dargestellt, könnte dies zu einer sehr

langen und unübersichtlichen Liste von Ereignissen führen. Aus diesem Grund werden die in einem Schleifenrumpf auftretenden Ereignisse in einer baumartigen Struktur, wie in der Abbildung 6.4 (Zeitstempel 20–65), dargestellt. Die Wurzel des Baumes bildet das Schlüsselwort „loop(X)“, wobei X für die Gesamtzahl der Schleifendurchläufe steht. Die Knoten der Ebene 1 des Baumes repräsentieren die einzelnen Schleifendurchläufe und sind mit „iteration x“ bezeichnet, wobei x für die fortlaufende Nummer des Schleifendurchlaufs steht. Im Vergleich zu einer unstrukturierten Darstellung ist die Baumdarstellung besonders für Schleifen mit vielen Wiederholungen wesentlich kompakter und übersichtlicher.

Zu den Datenflussereignissen zählen:

Objekterzeugung - Die Erzeugung neuer Objekte wird durch das Schlüsselwort „new“ dargestellt. In der Ereignisansicht werden sowohl die Erzeugung neuer Objekte (vgl. Ereignis mit Zeitstempel 2) als auch die Erzeugung neuer Arrays (vgl. Ereignis mit Zeitstempel 16) dargestellt.

Zuweisungen - Zuweisungen ändern den Programmzustand, indem sie einer Variablen einen Wert zuweisen. Zuweisungsereignisse werden in der Form „<Variablenname>=<Wert>“ dargestellt. In der Abbildung 6.4 handelt es sich bei den Ereignissen mit den Zeitstempeln 15, 17, 18, 39 und 41 um Zuweisungen.

Große Ereignislisten

Die Anzahl der Ereignisse, die durch einen Methodenaufruf erzeugt werden, kann, vor allem durch den Einsatz von Schleifen, sehr groß werden. Die Darstellung einer großen Anzahl von Ereignissen in einer flachen Liste ist problematisch, da diese Liste schnell unübersichtlich wird. Um die Navigation zu erleichtern und die Übersichtlichkeit für den Benutzer zu verbessern, ist es von Vorteil lange Ereignislisten in einer Baumstruktur darzustellen. Für Schleifen ist dieses Verfahren bereits beschrieben worden. Allerdings stößt auch diese Darstellungsform an ihre Grenzen, wenn eine Schleife mehrere tausend Schleifendurchläufe erzeugt.

Aus diesem Grund existiert ein weiterer Mechanismus zur Strukturierung langer Ereignislisten. Der Mechanismus betrachtet die gesamte Ereignisliste eines Methodenaufrufs c als Baum. Die Wurzel dieses Baumes ist der Methodenaufruf c. Sei k ein Knoten des Baumes und $k_1, \ldots, k_n$, $n \in \mathbb{N}$, die Folge der Kindknoten von k. Bevor die Kinder eines Knotens k dargestellt werden, überprüft die Ereignisansicht die Anzahl der Kinder. Sollte der Knoten k mehr als 100 Kinder besitzen ($n > 100$), dann wird für k eine neue Folge von Kindknoten $l_1, \ldots, l_m$, $m = \lceil n/100 \rceil$, erzeugt. Die alten Kindknoten $k_1, \ldots, k_n$ werden anschließend zu Kindknoten von $l_1, \ldots, l_m$. Das heißt dem Knoten l_i, $1 \leq i < m$, werden die Kindknoten $k_{(i-1)*100+1}, \ldots, k_{i*100}$ zugeordnet und l_m werden die Kindknoten $k_{(m-1)*100+1}, \ldots, k_n$ zugeordnet. Auf diese Weise lassen sich auch lange Folgen von Ereignissen in strukturierter Form darstellen.

Selektion von Ereignissen

Wird ein Ereignis e, das zum Methodenaufruf c_e gehört, in der Ansicht selektiert, so hat dies Auswirkungen auf den Quelltexteditor von Eclipse und die Knotenansicht von JHyde. Im Quelltexteditor wird die Quelltextzeile, die das Ereignis e verursacht hat, markiert und angezeigt. In der Knotenansicht wird der Endzeitpunkt auf den Zeitstempel des selektierten Ereignisses t_e gesetzt. Folglich werden ausschließlich die Zustandsänderungen der Ereignisse angezeigt, für deren Zeitstempel t gilt, dass $t_{start}(c_e) \leq t \leq t_e$ ist.

Wird in der Ereignisansicht aus Abbildung 6.4 zum Beispiel das Ereignis mit dem Zeitstempel 18 selektiert, dann wird im Quelltexteditor die Zeile 19 der Java-Datei angezeigt, in der die gegenwärtig untersuchte Methode implementiert ist. Durch die Verknüpfung der Ereignisansicht mit dem Quelltexteditor ist es leichter nachzuvollziehen, an welcher Position im Programmablauf sich das gegenwärtig selektierte Ereignis befindet. Zudem ist es einfacher zu beurteilen, ob das gegenwärtig selektierte Ereignis korrekt ist. Darüber hinaus wird der Endzeitpunkt der Knotenansicht auf 18 gesetzt und die Knotenansicht zeigt somit alle Zustandsänderungen an, die durch die Ereignisse verursacht wurden, deren Zeitstempel im Intervall $[6, 18]$ liegt.

Werden die Ereignisse der Ereignisansicht nacheinander schrittweise selektiert, so lässt sich parallel in der Knotenansicht die Änderung des Zustands-

raums beobachten. Bei der Selektion der Ereignisse in der Ereignisfolge kann beliebig vor- und zurückgesprungen werden. Auf diese Weise unterstützt die Ereignisansicht das Omniscient-Debugging.

Markierung von Ereignissen

In der Ereignisansicht können darüber hinaus alle Ereignisse farblich markiert werden, die den Wert einer bestimmten Variablen verändert haben. Hierfür muss die entsprechende Variable v in der Knoten Ansicht selektiert werden. Anschließend wird die Hintergrundfarbe eines Ereignisses in der Ereignisansicht nach folgenden Regeln geändert:

- Wenn ein Ereignis den Wert der Variablen v verändert hat, dann wird dieses Ereignis mit einer roten Hintergrundfarbe markiert.
- Wenn ein Ereignis e den Wert der Variablen v nicht verändert hat, jedoch mindestens ein Ereignis, welches Element des Teilbaumes mit der Wurzel e ist, den Wert von v verändert hat, dann wird das Ereignis mit einer gelben Hintergrundfarbe markiert.

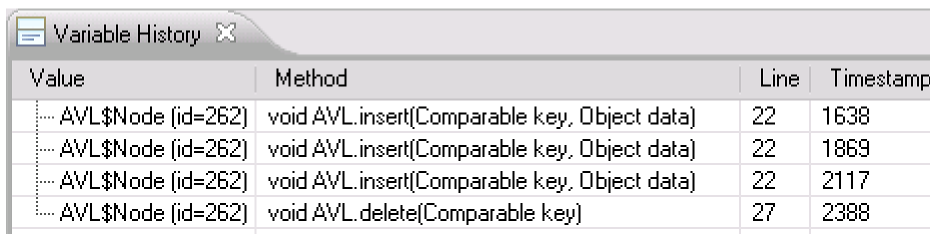
In der Abbildung 6.4 sind beispielsweise alle Ereignisse markiert, die für die Änderung des Attributs `succ` des `Element`-Objekts mit der Objekt-ID 3 verantwortlich sind. Die Ereignisse mit den Zeitstempeln 20, 21, 23, 43 und 65 sind gelb markiert, da diese einen Nachkommen besitzen, welcher den Wert von `succ` verändert. Das Ereignis mit dem Zeitstempel 38 wird rot gefärbt, da es der Variablen `succ` einen neuen Wert zuweist.

Mithilfe der farblichen Markierung von Ereignissen kann gezielt nach der Ursache einer Zustandsänderung gesucht werden. Wenn der Wert einer Variablen am Ende eines Methodenaufrufs infiziert ist, dann können mit diesem Hilfsmittel Ereignisse markiert werden, die eine potenzielle Fehlerursache sein können. Dieses Verfahren funktioniert jedoch nur für Infektionen vom Typ Ia und IIa, d. h. für Infektionen, die unmittelbar aus einem Defekt im Quelltext oder einem anderen infizierten Wert resultieren (vgl. Abschnitt 3.2). Für diese Infektionstypen ist das Omniscient-Debugging in der Regel die effizienteste Methode, um den gesuchten Defekt im Quelltext zu identifizieren.

6.1.4 Variablenansicht

Durch die Ereignisansicht wird die Suche nach Ereignissen, die den Wert einer Variablen verändert haben, auf die gegenwärtig dargestellte Ereignisliste beschränkt. Mithilfe der Variablenansicht (Variable History) ist es möglich für den gesamten Ablauf des untersuchten Programms alle Ereignisse zu identifizieren, die den Wert einer Variablen verändert haben. Hierfür kann der Variablenansicht über die Knotenansicht eine Variable zugewiesen werden. Die Zuweisung erfolgt über ein Kontextmenü, welches sich nach einem Rechts-Klick auf die gewünschte Variable öffnet. Die Auswahl der Menüpunktes „Show in Varible History View“ führt dazu, dass die entsprechende Variable der Variablenansicht zugewiesen wird.

Wie in der Abbildung 6.5 dargestellt, werden für die zugewiesene Variable chronologisch alle Ereignisse aufgelistet, durch die der Wert der Variablen während des Programmablaufs geändert wurde. Die Variablenansicht besitzt insgesamt vier Spalten, in denen für jede Wertänderung der untersuchten Variablen die folgenden Informationen angezeigt werden: der zugewiesene Wert („Value“), die vollqualifizierte Signatur der Methode („Method“), die Zeile im Quelltext („Line“), deren Ausführung die Wertänderung verursacht hat, und der Zeitstempel des Ereignisses („Timestamp“), welches die Wertänderung verursacht hat.



Value	Method	Line	Timestamp
AVL$Node (id=262)	void AVL.insert(Comparable key, Object data)	22	1638
AVL$Node (id=262)	void AVL.insert(Comparable key, Object data)	22	1869
AVL$Node (id=262)	void AVL.insert(Comparable key, Object data)	22	2117
AVL$Node (id=262)	void AVL.delete(Comparable key)	27	2388

Abbildung 6.5: Variablenansicht der JHyde Benutzeroberfläche.

Wird die durch ein Ereignis e verursachte Wertänderung in der Variablenansicht selektiert, dann hat dies die folgenden Auswirkungen:

- In der Berechnungsbaumansicht wird automatisch der Methodenaufruf c_e selektiert, dessen Ausführung die Wertänderung verursacht hat.

- In der Ereignisansicht wird das Ereignis e selektiert.
- In der Knotenansicht wird die Zustandsmenge $z_{c_e}\big(t_{start}(c_e), t_e\big)$ dargestellt. Damit zeigt die Knotenansicht für den Methodenaufruf c_e die Zustandsänderungen aller Ereignisse an, für deren Zeitstempel t gilt, dass $t_{start}(c_e) \leq t \leq t_e$.

Rückverfolgung von Infektionen

Da die Variablenansicht für eine ausgewählte Variable v alle Ereignisse anzeigt, die während des Programmablaufs den Wert von v geändert haben, unterstützt sie die Suche nach der Ursache eines infizierten Wertes. Wenn bei der Suche nach dem Defekt im Programmablauf eine Variable mit infiziertem Wert identifiziert wurde, dann kann mithilfe der Variablenansicht direkt zu der Stelle im Programmablauf gesprungen werden, an der die Infektion stattgefunden hat.

Gesetzt den Fall, dass mit der Variablenansicht die Infektion einer Variablen `a` zu der Anweisung „`a = b + c`“ zurückverfolgt wird, dann ist die Ursache der Infektion von c am einfachsten zu ermitteln, wenn die Infektion vom Typ Ia oder IIa (vgl. 3.2) ist:

- Eine Infektion vom Typ Ia liegt vor, wenn die Anweisung zur Berechnung des neuen Wertes von `a` fehlerhaft ist. Dies ist zum Beispiel der Fall, wenn die Anweisung „`a = b - c`“ hätte lauten müssen. Die Ursache der Infektion kann somit direkt gefunden werden.
- Eine Infektion vom Typ IIa liegt vor, wenn die Anweisung „`a = b + c`“ korrekt ist, jedoch einer der Werte `b` oder `c` infiziert ist. In diesem Fall muss die Ursache der Infektion weiter zurückverfolgt werden. Zunächst ist zu prüfen, ob `b` oder `c` infiziert ist und anschließend muss mithilfe der Variablenansicht die Wertentwicklung der entsprechenden Variablen weiter zurückverfolgt werden.

Bei den Infektionen vom Typ Ib und IIb ist die Zurückverfolgung etwas schwieriger:

- Im ersten Fall ist der Wert, welcher durch die Anweisung „`a = b + c`“ zugewiesen wurde korrekt, jedoch fehlt im folgenden Programmablauf eine Anweisung, durch die der Variablen `a` ein weiterer Wert

zugewiesen wird. Der Programmablauf muss in diesem Fall manuell nach der Position durchsucht werden, an der die Zuweisung hätte erfolgen müssen.

- Im zweiten Fall ist die Anweisung ebenfalls korrekt; sie hätte jedoch nicht ausgeführt werden dürfen. Hier ist die Ursache für die Infektion eine fehlerhafte Verzweigung im Kontrollfluss. Der Kontrollfluss des Programms muss manuell bis zur Stelle der fehlerhaften Verzweigung zurückverfolgt werden.

Wie gezeigt, unterstützt die Variablenansicht die Zurückverfolgung von infizierten Werten. Sie ermöglicht einen rückwärts gerichteten Debugging-Prozess, durch den eine Fehlerwirkung zum verursachenden Defekt zurückverfolgt werden kann. In Verbindung mit der Ereignisansicht ist sie somit ein entscheidendes Hilfsmittel für das Omniscient-Debugging.

6.2 Debugging eines defekten Java-Programms

In den folgenden Abschnitten wird gezeigt, wie durch den Einsatz von JHyde ein Defekt in einem fehlerhaften Java-Programm gefunden werden kann. Zu diesem Zweck wird zunächst ein defekter Mergesort-Algorithmus vorgestellt (Abschnitt 6.2.1). Im Anschluss wird gezeigt, wie mithilfe von JHyde der Defekt im Quelltext der Klasse `Mergesort` gefunden werden kann.

6.2.1 Defekter Mergesort-Algorithmus

Die in Listing 6.1 dargestellte Klasse `Mergesort` implementiert den Mergesort-Algorithmus [37] für `int`-Arrays und besteht insgesamt aus vier Methoden mit der folgenden Semantik:

- Die Methode `merge` verschmilzt die Werte des Arrays `a`, für deren Index `i` gilt, dass $\texttt{la} \leq \texttt{i} < \texttt{ra}$, und die Werte des Arrays `b`, für deren Index `j` gilt, dass $\texttt{lb} \leq \texttt{j} < \texttt{rb}$, zur Folge d. Unter der Voraussetzung, dass die Folgen $(\texttt{a[la]}, \ldots, \texttt{a[ra - 1]})$ und

(`b[lb],...,b[rb - 1]`) monoton steigend sind, ist auch die resultierende Folge d monoton steigend. Für $k \in \mathbb{N}, k < \texttt{ra}-\texttt{la}+\texttt{rb}-\texttt{lb}$ wird das k-te Element der Folge d an der Position $\texttt{lc} + k$ im Array `c` gespeichert.

- Die Methode `distribute` verteilt die Elemente des Arrays `a` auf die Arrays `b` und `c`. Hierfür wird das Array `a` in Blöcke der Länge `l` unterteilt. Sei d die Folge der Blöcke in `a`, dann werden die Blöcke abwechselnd in das Array `b` oder das Array `c` kopiert, so dass am Ende `b` die Folge der Blöcke $(d_0, d_2, \ldots, d_{\texttt{a.length}/l-2})$ und `c` die Folge der Blöcke $(d_1, d_3, \ldots, d_{a.length/l-1})$ enthält.

- Die Methode `sort` verwendet die Methoden `distribute` und `merge`, um das Array `a` aufsteigend zu sortieren. Dafür werden in mehreren Durchläufen die Elemente des Arrays `a` zunächst durch einen `distribute`-Aufruf auf die Arrays `b` und `c` verteilt und anschließend durch eine Folge von `merge`-Aufrufen wieder in `a` verschmolzen. Um ein Array `a` der Länge n vollständig zu sortieren, müssen insgesamt $\log_2 n$ dieser Durchläufe ausgeführt werden. Die Folge der durch die `distribute`-Aufrufe erzeugten Blockgrößen ist dabei $(2^0, 2^1, \ldots, 2^{\log_2(n)-1})$. Entsprechend lautet die Folge für die Anzahl der `merge`-Aufrufe, die während der Durchläufe der äußeren Schleife auszuführen sind, um die Arrays `b` und `c` vollständig in `a` zu verschmelzen, $\left(n \cdot \left(\frac{1}{2}\right)^1, n \cdot \left(\frac{1}{2}\right)^2, \ldots, n \cdot \left(\frac{1}{2}\right)^{\log_2 n} = 1\right)$. Zu beachten ist, dass diese einfache Implementierung des Mergesort-Algorithmus ausschließlich für Arrays der Länge $2^i, i \in \mathbb{N}$, korrekt funktioniert.

- Die Methode `main` erzeugt das Array `a` mit der Elementfolge $[4, 9, 1, 7]$, sortiert dieses Array und gibt das Ergebnis auf der Konsole aus.

Die Implementierung der Methode `merge` enthält einen Defekt in der Zeile 12. Statt „`c[`**`lb`**`++] = a[la++];`“ müsste die Zeile korrekterweise „`c[`**`lc`**`++] = a[la++];`“ lauten. Der Aufruf der `main`-Methode der defekten `Mergesort`-Klasse zeigt eine Fehlerwirkung, da das in der Konsole ausgegebene Array $[1, 4, 9, 7]$ falsch sortiert ist.

Listing 6.1: Defekte Implementierung des Mergesort-Algorithmus. Das Programm enthält einen Defekt in Zeile 12. Die Zeilennummern beginnen bei 5, da sie den Zeilennummern des Quelltexteditors der Abbildung 6.1 gleichen.

```
 5 public class Mergesort {
 6
 7   private static void merge(int[] a, int la, int ra,
 8       int[] b, int lb, int rb, int[] c, int lc) {
 9     while (la < ra && lb < rb)
10       c[lc++] = (a[la] < b[lb]) ? a[la++] : b[lb++];
11     while (la < ra)
12       c[lb++] = a[la++]; // korrekte Anweisung: c[lc++]
           = a[la++];
13     while (lb < rb)
14       c[lc++] = b[lb++];
15   }
16
17   private static void distribute(int[] a, int[] b,
      int[] c, int l) {
18     int i = 0;
19     for (int j = 0; j < a.length; j += 2 * l) {
20       for (int k = j / 2; k < j / 2 + l; k++)
21         b[k] = a[i++];
22       for (int k = j / 2; k < j / 2 + l; k++)
23         c[k] = a[i++];
24     }
25   }
26
27   public static void sort(int[] a) {
28     int[] b = new int[a.length];
29     int[] c = new int[a.length];
30     for (int size=1; size < a.length; size *= 2) {
31       distribute(a, b, c, size);
32       for (int i = 0; i < a.length / 2; i += size)
33         merge(b, i, i + size, c, i, i + size, a, 2 * i);
34     }
35   }
36
37   public static void main(String[] args) {
38     int[] a = new int[] {4,9,1,7};
39     MergeSort.sort(a);
40     System.out.println(Arrays.toString(a));
41   }
42 }
```

6.2.2 Suche des Defekts mit JHyde

Im Folgenden wird gezeigt, wie mithilfe von JHyde von der Fehlerwirkung des `Mergesort`-Programms aus Listing 6.1 auf den verursachenden Defekt geschlossen werden kann. Die Suche nach dem Defekt ist in zwei Phasen unterteilt:

- In der ersten Phase wird mithilfe des deklarativen Debugging nach dem Defekt gesucht. In dieser Phase werden hauptsächlich die Berechnungsbaumansicht und die Knotenansicht verwendet, um Methodenaufrufe des Programmablaufs zu klassifizieren. Nach jeder Klassifizierung wählt JHyde automatisch den nächsten zu klassifizierenden Methodenaufruf anhand der Top-Down-Strategie. Die Methodenaufrufe des Berechnungsbaums werden solange klassifiziert, bis JHyde meldet, dass der defekte Methodenaufruf gefunden ist.

- Anschließend wird in der zweiten Phase das Omniscient-Debugging dazu verwendet, innerhalb der defekten Methode die defekte Anweisung zu identifizieren. Während des Omniscient-Debugging werden im Wesentlichen die Knotenansicht, die Ereignisansicht und die Variablenansicht verwendet. Mithilfe dieser Ansichten muss in der Ereignisliste die Position bestimmt werden, die den Programmzustand erstmalig von einem validen in einen infizierten Zustand überführt.

Um den Debugging-Prozess für das `Mergesort`-Programm durchführen zu können, muss zunächst der vollständige Programmablauf mit JHyde aufgezeichnet werden. Hierfür wird im Package-Explorer der Eclipse-Umgebung durch einen Rechts-Klick auf die Klasse `Mergesort` das Kontextmenü aktiviert. Die Auswahl der Option „Debug As → JHyde Application“ führt dazu, dass die `main`-Methode der Klasse `Mergesort` vollständig ausgeführt und dabei der Programmablauf durch JHyde aufgezeichnet wird. In der Debug-Perspektive von Eclipse wird anschließend der aufgezeichnete Programmablauf durch die Ansichten des JHyde-Plugins dargestellt.

Die Abbildung 6.6 zeigt den Zustand der Benutzeroberfläche von JHyde zu Beginn der Debugging-Sitzung. In der Berechnungsbaumansicht ist die Wurzel des Berechnungsbaums, d. h. der Aufruf der `main`-Methode, selektiert. Die Wurzel besitzt insgesamt drei Kindknoten für die Aufrufe der Methoden

`sort`, `toString` und `println`. Alle aufgerufenen Methoden der Teilbäume, deren Wurzel die Methodenaufrufe `toString` und `println` sind, gehören zur Java-API. Aus diesem Grund sind alle Knoten dieser Teilbäume in der Berechnungsbaumansicht schon zu Beginn des Debugging-Prozesses als vertrauenswürdig („trusted") markiert. Die Klassifizierung der übrigen Knoten der Ansicht ist noch nicht bekannt („unknown"). Darüber hinaus ist in der Berechnungsbaumansicht die Top-Down-Strategie als deklarative Debugging-Strategie selektiert.

Computation Tree

```
Mergesort.main(String[0] (id=2)) --> void (unknown)
  Mergesort.sort(int[4] (id=3)) --> void (unknown)
    Mergesort.distribute(int[4] (id=3), int[4] (id=4), int[4] (id=5), 1) --> void (unknown)
    Mergesort.merge(int[4] (id=4), 0, 1, int[4] (id=5), 0, 1, int[4] (id=3), 0) --> void (unknown)
    Mergesort.merge(int[4] (id=4), 1, 2, int[4] (id=5), 1, 2, int[4] (id=3), 2) --> void (unknown)
    Mergesort.distribute(int[4] (id=3), int[4] (id=4), int[4] (id=5), 2) --> void (unknown)
    Mergesort.merge(int[4] (id=4), 0, 2, int[4] (id=5), 0, 2, int[4] (id=3), 0) --> void (unknown)
  Arrays.toString(int[4] (id=3)) --> "[1, 4, 9, 7]" (id=21) (trusted)
  PrintStream.println("[1, 4, 9, 7]" (id=21)) --> void (trusted)
```

Node

Name	Start value	End value
return	void	void
args	String[0] (id...	String[0] (i...
a	null	int[4] (id=3)
<end value>	int[4] (id=3)	int[4] (id=3)
[0]	0	1
[1]	0	4
[2]	0	9
[3]	0	7

Event List

Event	Timestamp	Line
new int[4] (id=3)	6	38
int[0]=4	7	38
int[1]=9	8	38
int[2]=1	9	38
int[3]=7	10	38
a=int[4] (id=3)	11	38
Mergesort.sort(int[4] (id=3)) --> void	12	39
Arrays.toString(int[4] (id=3)) --> String (id=20)	244	40
PrintStream.println(String (id=20)) --> void	2569	40
return void	22470	41

Mergesort.java

```
37    public static void main(String[] args) {
38        int[] a = new int[] {4,9,1,7};
39        Mergesort.sort(a);
40        System.out.println(Arrays.toString(a));
41    }
```

Abbildung 6.6: Start der Debugging-Sitzung mit JHyde für den Programmablauf der defekten Mergesort-Implementierung.

Die Knotenansicht zeigt die Start- und Endwerte aller lokalen Variablen des Aufrufs der `main`-Methode. Der Wert der lokalen Variablen `a` hat sich im

Laufe der Ausführung der main-Methode geändert. Zu Beginn des Aufrufs war der Wert „null“, am Ende des Aufrufs verweist die Variable auf ein Array der Länge 4 mit den Werten „[1, 4, 9, 7]“.

In der Ereignisansicht sind alle Ereignisse, die während der Ausführung der main-Methode aufgezeichnet wurden, aufgelistet. Die Ereignisse mit den Zeitstempeln 6–11 erzeugen und initialisieren das Array, welches der lokalen Variablen a zugewiesen wird. Anschließend folgen drei Ereignisse, die die Aufrufe der Methoden sort, toString und println repräsentieren. Das Ereignis „return void“ deutet das Ende der Ausführung der main-Methode an. Im Quelltexteditor ist die erste Zeile der Implementierung der main-Methode markiert. Durch die Implementierung sowie mögliche Kommentierungen der Methode können häufig Informationen über die intendierte Semantik des gegenwärtig untersuchten Methodenaufrufs gewonnen werden. In diesem Fall ist für die Klassifizierung die Untersuchung der lokalen Variablen a ausreichend. Das durch die lokale Variable a referenzierte Array wird während der Ausführung der main-Methode dem Methodenaufruf von sort als Argument übergeben. Die Endwerte der Elemente des Arrays sollten daher sortiert sein. Da dies nicht der Fall ist, muss die Wurzel des Berechnungsbaums als infiziert klassifiziert werden.

Nach der Wurzel des Berechnungsbaums wählt die Top-Down-Strategie den Aufruf der Methode sort als nächsten zu klassifizierenden Knoten. Für diesen Knoten sind die Knotenansicht und die Ereignisansicht in der Abbildung 6.7 dargestellt. Wie in der Ereignisansicht gezeigt, werden während des Aufrufs die temporären Arrays b und c erzeugt (Zeitstempel 17–20), mit deren Hilfe in zwei Distribute-Merge-Durchläufen (Zeitstempel 24–146 und 147–242) das übergebene Array a sortiert wird. Die Startwerte der Elemente von a zeigen die ursprüngliche Anordnung der Elemente und die Endwerte die Anordnung der Elemente von a am Ende des Aufrufs der sort-Methode. Da die Elemente nicht aufsteigend sortiert wurden, ist der Aufruf ebenfalls als infiziert zu klassifizieren.

Im nächsten Schritt ist der erste Aufruf der Methode distribute zu klassifizieren (vgl. Abbildung 6.8. Die Methode besitzt die Argumente a, b, c, und l. Die tatsächliche Eingabe besteht jedoch nur aus dem Array a und dem Parameter l, der die Länge der in a gespeicherten und zu verteilenden Blöcke definiert. Das Ergebnis der Verteilung der Blöcke des Arrays a wird über die Argumente b und c zurückgegeben. Die durch den Parameter l

Node

Name	Start value	End value
return	void	void
a	int[4] (id=3)	int[4] (id=3)
[0]	4	1
[1]	9	4
[2]	1	9
[3]	7	7
b	null	int[4] (id=4)
<end value>	int[4] (id=4)	int[4] (id=4)
[0]	0	4
[1]	0	9
[2]	0	0
[3]	0	0
c	null	int[4] (id=5)
<end value>	int[4] (id=5)	int[4] (id=5)
[0]	0	1
[1]	0	7
[2]	0	0
[3]	0	0
size	0	4
i	0	2

Event List

Event	Timestamp	Line
new int[4] (id=4)	17	28
b=int[4] (id=4)	18	28
new int[4] (id=5)	19	29
c=int[4] (id=5)	20	29
size=1	21	30
branch	22	30
loop (2)	23	31
iteration 1	24	31
MergeSort.distribute(int[4] (id=3), int[4]	25	31
i=0	76	32
branch	77	32
loop (2)	78	33
iteration 1	79	33
MergeSort.merge(int[4] (id=4),	80	33
i=1	110	32
branch	111	32
iteration 2	112	33
MergeSort.merge(int[4] (id=4),	113	33
i=2	143	32
branch	144	32
size=2	145	30
branch	146	30
iteration 2	147	31
return void	243	35

Abbildung 6.7: Knoten- und Ereignisansicht für den Methodenaufruf `sort`.

festgelegte Blocklänge ist 1. Wie im Abschnitt 6.2.1 beschrieben, müssen die Blöcke des Arrays `a` auf die Arrays `b` und `c` verteilt werden. Die Blöcke werden dazu abwechselnd entweder in das Array `b` oder das Array `c` kopiert, wobei mit dem Array `b` begonnen wird. Am Ende der Aufteilung muss daher `b` die Blöcke 0 und 2 bzw. die Werte 4 und 1 enthalten und `c` muss die Blöcke 1 und 3 bzw. die Werte 9 und 7 enthalten. Die in der Abbildung 6.8 dargestellten Elemente der Arrays `b` und `c` werden durch die Ausführung des Methodenaufrufs so manipuliert, dass sie am Ende die erwarteten Werte enthalten. Der Aufruf der Methode `sort` ist folglich als valide zu klassifizieren.

Es folgt die Untersuchung des ersten `merge`-Aufrufs, für den die Knotenansicht und die Ereignisansicht in der Abbildung 6.9 dargestellt sind. Durch die Startwerte der Argumente `la` (0), `ra` (1), `lb` (0) und `rb` (1) wird festgelegt, dass der `merge`-Aufruf die Werte an der Indexposition 0 der Arrays `a` und `b` zu einem sortierten Block verschmelzen soll. Das Argument `lc` legt fest, dass der erzeugte Block an der Position 0 im Array `c` gespeichert werden soll. Die Endwerte der Elemente 0 und 1 des Arrays `c` sind 4 und

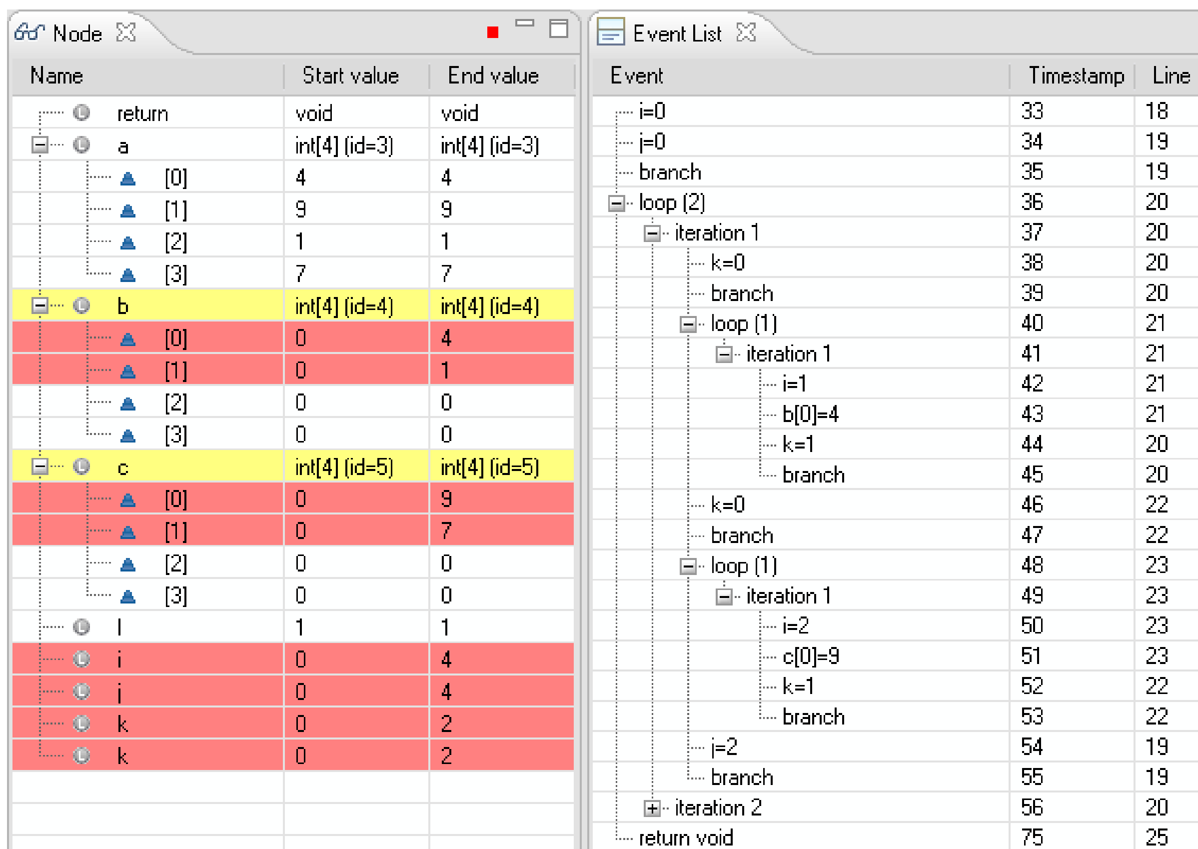


Abbildung 6.8: Knoten- und Ereignisansicht für den ersten `distribute`-Aufruf.

9. Diese sind aufsteigend sortiert und entsprechen den Einträgen der Arrays `a` und `b` an der Position 0. Der erste Aufruf der Methode `merge` ist daher valide.

Beim zweiten Aufruf der `merge`-Methode (vgl. Abbildung 6.10) legen die Startwerte der Argumente `la` (1), `ra` (2), `lb` (1) und `rb` (2) fest, dass die Einträge an der Indexposition 1 der Arrays `a` und `b` zu einem sortierten Block zu verschmelzen sind. Wegen des Startwertes des Arguments `lc` (2) ist dieser Block an der Indexposition 2 im Array `c` zu speichern. Die Endwerte der Elemente 2 und 3 des Arrays `c` sind 4 und 9. Diese sind aufsteigend sortiert und entsprechen den Einträgen der Arrays `a` und `b` an der Position 1. Die übrigen Werte von `c` wurden nicht geändert. Der zweite Aufruf der Methode `merge` ist somit valide.

Als nächstes ist der zweite Aufruf der Methode `distribute` zu klassifizieren (vgl. Abbildung 6.11. Die Argumente `a` und `l` zählen wiederum zur Eingabe der Methode, während die Argumente `b` und `c` dazu dienen, das

Node

Name	Start value	End value
return	void	void
a	int[4] (id=4)	int[4] (id=4)
[0]	4	4
[1]	1	1
[2]	0	0
[3]	0	0
la	0	1
ra	1	1
b	int[4] (id=5)	int[4] (id=5)
[0]	9	9
[1]	7	7
[2]	0	0
[3]	0	0
lb	0	1
rb	1	1
c	int[4] (id=3)	int[4] (id=3)
[0]	4	4
[1]	9	9
[2]	1	1
[3]	7	7
lc	0	2

Event List

Event	Timestamp	Line
branch	92	9
branch	93	9
loop (1)	94	10
iteration 1	95	10
lc=1	96	10
branch	97	10
la=1	98	10
c[0]=4	99	10
branch	100	9
branch	101	11
branch	102	13
loop (1)	103	14
iteration 1	104	14
lc=2	105	14
lb=1	106	14
c[1]=9	107	14
branch	108	13
return void	109	15

Abbildung 6.9: Knoten- und Ereignisansicht für den ersten `merge`-Aufruf.

Resultat des Aufrufs zurückzugeben. Durch den Startwert des Arguments `l` wird das Array in Blöcke der Länge 2 unterteilt. Die Blöcke sind durch den `distribute`-Aufruf auf die Arrays `b` und `c` zu verteilen. Wie in der Knotenansicht zu erkennen ist, sind durch den Aufruf die Werte der Elemente 0 und 1 der Arrays `b` und `c` geändert worden. Am Ende des Aufrufs enthält das Array `b` an den Positionen 0 und 1 den ersten Block des Arrays `a` und das Array `c` enthält an den Positionen 0 und 1 den zweiten Block des Arrays `a`. Das erwartete Ergebnis entspricht somit auch beim zweiten Aufruf der `distribute`-Methode dem tatsächlichen Ergebnis und der Aufruf wird als valide klassifiziert.

Die Abbildung 6.12 zeigt die Benutzeroberfläche von JHyde. Die Berechnungsbaumansicht präsentiert die zuvor untersuchten Methodenaufrufe sowie deren Klassifizierung. In der Ansicht ist der letzte `merge`-Aufruf selektiert. Die Start- und Endwerte der lokalen Variablen des Aufrufs sind in der Knotenansicht dargestellt. Die Startwerte `la` (0), `ra` (2), `lb` (0), `rb` (2) definieren für die Arrays `a` und `b` jeweils einen Block, der die Elemente mit den Indizes 0 und 1 umfasst. Durch den Aufruf sollen diese Blöcke zu einem einzigen Block der Länge 4 verschmolzen werden. Der erzeugte

Node

Name	Start value	End value
return	void	void
a	int[4] (id=4)	int[4] (id=4)
[0]	4	4
[1]	1	1
[2]	0	0
[3]	0	0
la	1	2
ra	2	2
b	int[4] (id=5)	int[4] (id=5)
[0]	9	9
[1]	7	7
[2]	0	0
[3]	0	0
lb	1	2
rb	2	2
c	int[4] (id=3)	int[4] (id=3)
[0]	4	4
[1]	9	9
[2]	1	1
[3]	7	7
lc	2	4

Event List

Event	Timestamp	Line
branch	125	9
branch	126	9
loop (1)	127	10
iteration 1	128	10
lc=3	129	10
branch	130	10
la=2	131	10
c[2]=1	132	10
branch	133	9
branch	134	11
branch	135	13
loop (1)	136	14
iteration 1	137	14
lc=4	138	14
lb=2	139	14
c[3]=7	140	14
branch	141	13
return void	142	15

Abbildung 6.10: Knoten- und Ereignisansicht für den zweiten `merge`-Aufruf.

Block muss alle Elemente der Ursprungsblöcke enthalten und diese Elemente müssen aufsteigend sortiert sein. In der Knotenansicht zeigt sich, dass das Resultat der Verschmelzungsoperation, welches im Array `c` gespeichert werden soll, nicht dem erwarteten Ergebnis entspricht. Statt der sortierten Folge $(1, 4, 7, 9)$ enthält das Array `c` die Folge $(1, 4, 9, 7)$. Daher ist der Methodenaufruf `a` infiziert. Weil der Aufruf keine weiteren Kinder besitzt bzw. der Suchbereich keine unklassifizierten Knoten mehr enthält, meldet JHyde, dass der zweite Aufruf der `merge`-Methode der gesuchte defekte Knoten ist. Der Aufruf hat das Programm von einem validen in einen infizierten Zustand überführt. Dies äußert sich darin, dass alle übergebenen Argumente valide waren, jedoch das Resultat, d. h. die Reihenfolge der Werte des Arrays `c`, nicht den Erwartungen entspricht.

Mit der Identifikation des defekten Methodenaufrufs endet die deklarative Phase des Debugging-Prozesses. Zu diesem Zeitpunkt ist bekannt, dass die Implementierung der Methode `merge` einen Defekt enthalten muss. Um diesen Defekt genau zu lokalisieren, wird in der zweiten Phase das Omniscient-Debugging verwendet.

Node

Name	Start value	End value
return	void	void
a	int[4] (id=3)	int[4] (id=3)
[0]	4	4
[1]	9	9
[2]	1	1
[3]	7	7
b	int[4] (id=4)	int[4] (id=4)
[0]	4	4
[1]	1	9
[2]	0	0
[3]	0	0
c	int[4] (id=5)	int[4] (id=5)
[0]	9	1
[1]	7	7
[2]	0	0
[3]	0	0
l	2	2
i	0	4
j	0	4
k	0	2
k	0	2

Event List

Event	Timestamp	Line
i=0	156	18
j=0	157	19
branch	158	19
loop (1)	159	20
iteration 1	160	20
k=0	161	20
branch	162	20
loop (2)	163	21
iteration 1	164	21
i=1	165	21
b[0]=4	166	21
k=1	167	20
branch	168	20
iteration 2	169	21
i=2	170	21
b[1]=9	171	21
k=2	172	20
branch	173	20
k=0	174	22
branch	175	22
loop (2)	176	23
j=4	187	19
branch	188	19
return void	189	25

Abbildung 6.11: Knoten- und Ereignisansicht für den zweiten `distribute`-Aufruf.

Zu Beginn des Omniscient-Debugging ist bekannt, dass das Resultat des Methodenaufrufs, d. h. die Werte des Arrays `c` fehlerhaft sind. Genauer gesagt sind die Endwerte der Elemente mit dem Index 2 und 3 infiziert. Der Endwert des Elements 2 ist 9 anstelle von 7 und der Endwert des Elements 3 ist 7 anstelle von 9. Da der Wert des Elements 3 während des `merge`-Aufrufs nicht geändert wurde, ist die Infektion des Elements dadurch zustande gekommen, dass während des Aufrufs der richtige Wert nicht zugewiesen wurde. Die Infektion lässt sich aufgrund der *fehlenden Wertzuweisung* auch mithilfe des Omniscient-Debugging nicht direkt zurückverfolgen. Die Infektion des Elements mit dem Index 2 ist zustande gekommen, da dem Element im Laufe des Aufrufs ein *falscher Wert* zugewiesen wurde. Diese Art der Infektion lässt sich mithilfe des Omniscient-Debugging direkt zurückverfolgen. Hierfür muss die Variable bzw. das Element in der Knotenansicht selektiert werden. Anschließend werden, wie in der Abbildung 6.12 gezeigt, in der Ereignisansicht alle Ereignisse farblich hervorgehoben, die den Wert der Variablen verändert haben. In diesem Fall ist der Wert des Elements

Computation Tree

- Mergesort.main(String[0] (id=2)) --> void (invalid)
  - Mergesort.sort(int[4] (id=3)) --> void (invalid)
    - Mergesort.distribute(int[4] (id=3), int[4] (id=4), int[4] (id=5), 1) --> void (valid)
    - Mergesort.merge(int[4] (id=4), 0, 1, int[4] (id=5), 0, 1, int[4] (id=3), 0) --> void (valid)
    - Mergesort.merge(int[4] (id=4), 1, 2, int[4] (id=5), 1, 2, int[4] (id=3), 2) --> void (valid)
    - Mergesort.distribute(int[4] (id=3), int[4] (id=4), int[4] (id=5), 2) --> void (valid)
    - Mergesort.merge(int[4] (id=4), 0, 2, int[4] (id=5), 0, 2, int[4] (id=3), 0) --> void (buggy)
  - Arrays.toString(int[4] (id=3)) --> "[1, 4, 9, 7]" (id=21) (trusted)
  - PrintStream.println("[1, 4, 9, 7]" (id=21)) --> void (trusted)

Node

Name	Start value	End value
return	void	void
a	int[4] (id=4)	int[4] (id=4)
[0]	4	4
[1]	9	9
[2]	0	0
[3]	0	0
la	0	2
ra	2	2
b	int[4] (id=5)	int[4] (id=5)
[0]	1	1
[1]	7	7
[2]	0	0
[3]	0	0
lb	0	3
rb	2	2
c	int[4] (id=3)	int[4] (id=3)
[0]	4	1
[1]	9	4
[2]	1	9
[3]	7	7
lc	0	3

Event List

Event	Timestamp	Line
branch	206	9
branch	207	9
loop (3)	208	10
iteration 1	209	10
iteration 2	216	10
iteration 3	223	10
lc=3	224	10
branch	225	10
lb=2	226	10
c[2]=7	227	10
branch	228	9
branch	229	9
branch	230	11
loop (1)	231	12
iteration 1	232	12
lb=3	233	12
la=2	234	12
c[2]=9	235	12
branch	236	11
branch	237	13
return void	238	15

Mergesort.java

```
 7  private static void merge(int[] a, int la, int ra,
 8          int[] b, int lb, int rb, int[] c, int lc) {
 9      while (la < ra && lb < rb)
10          c[lc++] = (a[la] < b[lb]) ? a[la++] : b[lb++];
11      while (la < ra)
12          c[lb++] = a[la++]; // must be: c[lc++] = a[la++];
13      while (lb < rb)
14          c[lc++] = b[lb++];
15  }
```

Abbildung 6.12: Knoten- und Ereignisansicht für den dritten `merge`-Aufruf.

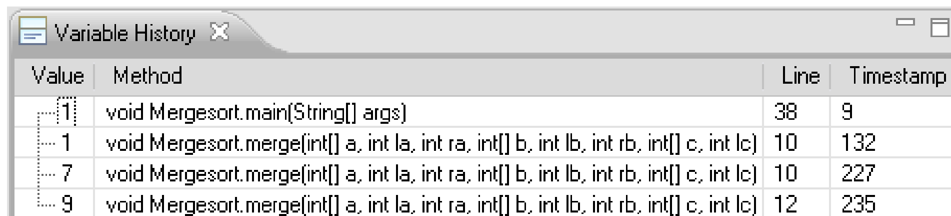
2 durch die Ereignisse mit den Zeitstempeln 227 und 235 geändert worden. Die erste Änderung hat in der ersten `while`-Schleife in der Zeile 10 stattgefunden, in der dem Element der Wert 7 zugewiesen wurde. In dieser Schleife wird in jedem Durchlauf das kleinste verbleibende Element der zu verschmelzenden Blöcke der Arrays `a` und `b` in das Array `c` kopiert. Die Ausführung der Schleife wird abgebrochen, sobald aus einem der Blöcke alle Elemente kopiert wurden. Nachdem der Wert 7 durch das Ereignis 227

und die Position 2 im Array `c` kopiert wurde, sind alle Werte des Arrays `b` kopiert worden. Zu diesem Zeitpunkt wurde die `while`-Schleife insgesamt dreimal durchlaufen und alle Werte außer `a[1]` sind nach `c` kopiert worden. Der verbleibende Wert des Arrays `a` soll durch die `while`-Schleife in der Zeile 11 in das Array `c` kopiert werden. Durch das Ereignis 235 wird korrekterweise der letzte verbleibende Wert des Arrays `a` kopiert, allerdings wird dieser Wert an die falsche Position im Array `c` geschrieben. Anstatt den Wert an der Position 2 zu überschreiben, hätte der Wert in das Element mit dem Index 3 geschrieben werden müssen. Das Ereignis bzw. die zugehörige Zeile im Quelltext ist somit defekt. Offensichtlich ist die Berechnung des Speicherplatzes in Zeile 12. An dieser Stelle ist der Debugging-Prozess beendet, da die defekte Anweisung gefunden wurde.

Deklaratives Debugging oder Omniscient-Debugging

Die durch den Defekt des `Mergesort`-Programms in Zeile 12 verursachte Infektion ist besonders einfach mithilfe des Omniscient-Debugging zurückzuverfolgen. In diesem Fall lässt sich die Position des Defekts im Programm sogar direkt mit dem Omniscient-Debugging ermitteln, ohne dass zuvor das deklarative Debugging verwendet wird. In der Abbildung 6.6, die den Zustand der Benutzeroberfläche von JHyde zu Beginn des Debugging-Prozesses zeigt, sind in der Knotenansicht die infizierten Endwerte des fehlerhaft sortierten Arrays `a` dargestellt. Mithilfe der Variablenansicht lässt sich die Ursache der Infektion auch ausgehend von der Ausführung der `main`-Methode zurückverfolgen. Hierfür muss die Variable, die die Werte des Elements mit dem Index 2 im Array `a` repräsentiert, in der Variablenansicht angezeigt werden. Die Abbildung 6.13 zeigt die Variablenansicht mit allen Ereignissen, die den Zustand des infizierten Array-Elements verändert haben. Insgesamt ist der Wert des Array-Elements durch 4 Ereignisse verändert worden. Das vorletzte Ereignis mit dem Zeitstempel 227 hat dem Element den Wert 7 zugewiesen. Dieser Wert entspricht dem erwarteten Wert am Ende des Programmablaufs. Durch das vierte Ereignis mit dem Zeitstempel 235 ist der valide Wert des Elements durch den Wert 9 überschrieben worden. Somit hat die vierte Wertzuweisung eine Infektion verursacht. Durch die Auswahl des vierten Ereignisses in der Variablenansicht wird in den übrigen Ansichten die Stelle des Programmablaufs angezeigt, an der das Ergebnis aufgetreten ist. Wie in der Abbildung 6.12 dargestellt, ist dies exakt die Stelle, an

der der Defekt aufgetreten ist. Mithilfe des Omniscient-Debugging lässt sich der gesuchte Defekt somit auch direkt und ohne den Einsatz der deklarativen Debugging-Methode finden. Durch den Verzicht auf die deklarative Debugging-Strategie ist die Lokalisierung sogar um einiges schneller.


Variable History

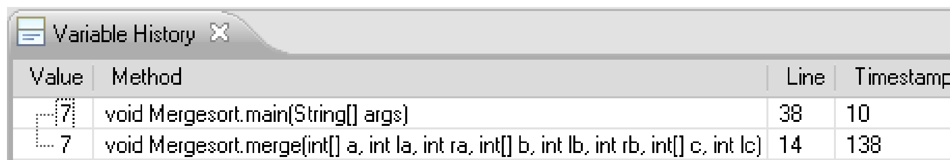
Value	Method	Line	Timestamp
1	void Mergesort.main(String[] args)	38	9
1	void Mergesort.merge(int[] a, int la, int ra, int[] b, int lb, int rb, int[] c, int lc)	10	132
7	void Mergesort.merge(int[] a, int la, int ra, int[] b, int lb, int rb, int[] c, int lc)	10	227
9	void Mergesort.merge(int[] a, int la, int ra, int[] b, int lb, int rb, int[] c, int lc)	12	235

Abbildung 6.13: Variablenansicht für das infizierte Array-Element `a[2]`.

Die Effizienz des Omniscient-Debugging ist in hohem Maße von der Art der untersuchten Infektion abhängig. In dem oben betrachteten Fall ist die Suche zum Beispiel wesentlich effizienter, wenn auf das deklarative Debugging verzichtet wird und ausschließlich mithilfe des Omniscient-Debugging nach der Ursache der Infektion gesucht wird. Für andere Arten von Infektionen ist es jedoch sinnvoll, zunächst durch Einsatz der deklarativen Debugging-Methode den defekten Methodenaufruf zu identifizieren und anschließend innerhalb des Aufrufs mithilfe des Omniscient-Debugging nach der Ausführung der defekten Anweisung zu suchen.

Eine Infektion, bei der das Omniscient-Debugging für die Defektsuche weniger gut geeignet ist, tritt zum Beispiel auf, wenn in der `Mergesort`-Implementierung die Zeilen 11 und 12 fehlen. Während der Ausführung des defekten Programms würde es dann nicht mehr zu einer Infektion durch eine *fehlerhafte* Zuweisung, sondern durch eine *fehlende* Zuweisung kommen. Statt `[1,4,9,7]` lautet das Ergebnis des `Mergesort`-Aufrufs dann `[1,4,7,7]`. Wegen der fehlenden `while`-Schleife wird im letzten `merge`-Aufruf der Wert 9 des Arrays `a` nicht in das Array `c` kopiert. Die Infektion des Elements mit dem Index 3, welches am Ende der Ausführung den Wert 7 anstelle des Wertes 9 hat, lässt sich durch die Variablenansicht nicht zurückverfolgen. Die Abbildung 6.14 zeigt, dass dem infizierten Element des Ergebnis-Arrays nur an zwei Stellen im Programmablauf ein Wert zugewiesen wurde. Zum einen ist dies der Wert 7 bei der Initialisie-

rung des Arrays während des Aufrufs der `main`-Methode und zum anderen ebenfalls der Wert 7 im dritten Aufruf der `merge` Methode. Beide Zuweisungen sind korrekt, es fehlt allerdings die Zuweisung des erwarteten Wertes 9. Hier zeigt sich, dass die Suche nach der Position im Programmablauf, an der eine *fehlende* Wertzuweisung hätte erfolgen müssen, deutlich aufwendiger ist als die Bestimmung der Position, an der ein *falscher* Wert zugewiesen wurde. Für die Suche nach fehlenden Zuweisungen bietet das Omniscient-Debugging kein einfaches und effizientes Verfahren, so dass der Programmablauf manuell nach der entsprechenden Position durchsucht werden muss. Das deklarative Debugging bietet in diesen Fällen eine gute Möglichkeit, den Suchraum durch die strukturierte und teilautomatisierte Methode schrittweise zu verkleinern. Nachdem die Position des Defekts im Programmablauf durch das deklarative Debugging auf einen einzigen Methodenaufruf eingeschränkt wurde, kann das Omniscient-Debugging innerhalb des eingeschränkten Suchbereichs durchgeführt werden.



Variable History

Value	Method	Line	Timestamp
7	void Mergesort.main(String[] args)	38	10
7	void Mergesort.merge(int[] a, int la, int ra, int[] b, int lb, int rb, int[] c, int lc)	14	138

Abbildung 6.14: [Variablenansicht für das infizierte Array-Element `a[2]` des modifizierten Mergesort-Programms.

Vergleich der deklarativen Debugging-Strategien

Für den hier dargestellten deklarativen Debugging-Prozess ist die Reihenfolge der klassifizierten Knoten durch die Top-Down-Strategie (vgl. 5.1) bestimmt worden. Diese Strategie folgt der Ausführungsreihenfolge der Methodenaufrufe, indem sie immer den Methodenaufruf des Suchbereichs mit dem frühesten Aufrufzeitpunkt klassifiziert. Werden die Methodenaufrufe der Berechnungsbaumansicht mit 1 beginnend von oben nach unten nummeriert, dann ist die Folge der klassifizierten Knoten $(1, 2, 3, 4, 5, 6, 7)$.

Für die einfache D&Q-Strategie lautet die Folge der klassifizierten Knoten $(2, 3, 4, 5, 6, 7)$. Im Vergleich zur Top-Down-Strategie wird durch die D&Q-Strategie nur die Klassifizierung des Knotens 1 eingespart. Der relativ geringe Effizienzvorteil resultiert aus der Struktur des Berechnungsbaums. Dieser weist nur eine geringe Knotenanzahl auf und seine Form kommt dem im Abschnitt 5.2 beschriebenen Worst Case der D&Q-Strategie relativ nahe. Der Berechnungsbaum hat eine relativ geringe Höhe und die meisten Knoten sind Blätter des Baumes.

Deutlich effizienter sind in diesem Fall die erweiterten D&Q-Strategien mit gewichtsunabhängigen Infektionswahrscheinlichkeiten. Für die auf der DU-Ketten-Überdeckung basierende Strategie sind ausschließlich die Knoten 3 und 7 zu klassifizieren. Für die auf der Bedingungsüberdeckung basierende Strategie muss sogar nur der Knoten 7 klassifiziert werden.

Die Knoten 3 und 7 besitzen bei beiden erweiterten D&Q-Strategien einen relativ niedrigen Erwartungswert, da sie im Vergleich zu den übrigen Blattknoten eine höhere Infektionswahrscheinlichkeit besitzen. Die größeren Infektionswahrscheinlichkeiten resultieren aus einer größeren Anzahl überdeckter Entitäten der entsprechenden Methodenaufrufe. Durch die Berücksichtigung der überdeckungsbasierten Infektionswahrscheinlichkeiten konnte das deklarative Debugging in diesem Fall erheblich beschleunigt werden.

Neben der Ausführungskomplexität sind die Infektionswahrscheinlichkeiten auch von der Klassifikation der übrigen Knoten abhängig. Für die auf DU-Ketten-Überdeckung basierende erweiterte D&Q-Strategie ist zu Beginn des deklarativen Debugging die festgelegte, vollständige Reihenfolge der zu klassifizierenden Knoten $(3, 6, 7, 4, 5, 2, 1)$. Nachdem der Knoten 3, der einen Aufruf der Methode `distribute` repräsentiert, im ersten Schritt als valide klassifiziert wurde, ändert sich die Reihenfolge der zu klassifizierenden Knoten. Der Einfluss auf die Infektionswahrscheinlichkeit aller DU-Ketten, welche durch den als valide klassifizierten Aufruf der `distribute`-Methode überdeckt wurden, wird verringert. Die geringere Infektionswahrscheinlichkeit des Knotens 6 führt für diesen Knoten zu einem größeren Erwartungswert für die Größe des verbleibenden Suchraums. Aus diesem Grund ändert sich die Reihenfolge der zu klassifizierenden Knoten. Nach der ersten Klassifizierung ist die neue Reihenfolge der Knoten $(3, 7, 4, 5, 6, 2, 1)$. Durch die Anpassung der Infektionswahrscheinlichkeiten nach der ersten

Klassifizierung kann der defekte Knoten bereits nach der zweiten und nicht erst nach der dritten Klassifikation gefunden werden.

6.3 Fazit

In diesem Kapitel wurde die Benutzeroberfläche von JHyde vorgestellt. Die Oberfläche ist ein Plugin für die Eclipse-Plattform, das aus insgesamt vier unterschiedlichen Ansichten besteht. Zunächst wurde die Funktionsweise dieser Ansichten erläutert. Anschließend wurde anhand eines defekten Java-Programms demonstriert, wie diese Ansichten dazu verwendet werden können, den hybriden Debugging-Prozess durchzuführen. Es wurde zudem gezeigt, wie die Leistungsfähigkeit der beiden Debugging-Methoden durch den Infektionstyp beeinflusst wird und welchen Einfluss die verschiedenen deklarativen Debugging-Strategien auf die Effizienz der Defektsuche haben.

Kapitel 7

Entwurf und Implementierung

Inhalt

7.1 Architektur 146

7.2 Transmitter 148

7.3 Instrumentierer 169

7.4 Rekorder 201

7.5 Benutzeroberfläche 209

7.6 Fazit 213

In diesem Kapitel werden die wichtigsten Aspekte des Entwurfs und der Implementierung von JHyde beschrieben. Im Abschnitt 7.1 wird zunächst die Architektur von JHyde dargestellt. In den nachfolgenden Abschnitten werden die Funktionseinheiten von JHyde genauer erläutert. Im Abschnitt 7.2 wird der Aufbau des Transmitters beschrieben, da dieser eine einheitliche Schnittstelle für die aufgezeichneten Ereignisse des Programmablaufs definiert. Im Abschnitt 7.3 wird anschließend der Prozess zur Instrumentierung des Bytecodes in der Prüflings-VM dargestellt. Im Abschnitt 7.4 folgt die Beschreibung des Rekorders in der Debugger-VM und eine Erläuterung des Modells zur Abbildung des Programmablaufs. Im Abschnitt 7.5 werden grundlegende Aspekte des Entwurfs und der Implementierung der Benutzeroberfläche von JHyde erläutert. Den Abschluss bildet das Fazit im Abschnitt 7.6.

7.1 Architektur

Die Architektur von JHyde ist in der Abbildung 7.1 dargestellt. Das System besteht aus mehreren Funktionseinheiten, die über zwei separate Java-VMs verteilt sind, die Debugger-VM und die Prüflings-VM. Während in der Prüflings-VM der Prüfling, d. h. das zu untersuchende Programm, ausgeführt wird, dient die Debugger-VM dazu, die aufgezeichneten Daten über die Ausführung des Prüflings nach Defekten zu durchsuchen. Die beiden Phasen des hybriden Debugging-Prozesses, die aus der Ausführung des Prüflings bzw. der Suche nach dem Defekt bestehen, werden somit auf zwei unterschiedliche virtuelle Maschinen verteilt. Die Trennung von Debugger und Prüfling durch separate Laufzeitumgebungen ist in zahlreichen Debugging-Systemen zu finden. Die Java-Plattform (Java SE 6) [145], stellt zum Beispiel über die *Java Platform Debugger Architecture* (JPDA) eine Menge von Schnittstellen bereit, mit denen der Programmablauf innerhalb einer Java-VM von außen gesteuert, überwacht und untersucht werden kann. Viele Entwicklungsumgebungen, wie NetBeans [147], Eclipse [48] und IntelliJ [71], verwenden die JPDA zum Debugging von Java-Programmen. Die Trennung von Debugger-VM und Prüflings-VM bietet zwei entscheidende Vorteile:

- Aus Sicht der Debugger-VM ist die Verteilung der virtuellen Maschinen transparent. Die Architektur ermöglicht daher auch das Remote-Debugging einer Prüflings-VM, die auf einem entfernten Rechner ausgeführt wird.
- Da der Debugger und der Prüfling in zwei separaten virtuellen Maschinen laufen, werden die Interferenzen zwischen beiden Programmen auf ein Minimum reduziert. Der Debugger ist dadurch vor den Defekten des Prüflings geschützt und ein Absturz des Prüflings bzw. der Prüflings-VM führt nicht dazu, dass die Ausführung des Debuggers beeinflusst wird.

Die JPDA eignet sich besonders für die Implementierung eines Trace-Debuggers, da sie sämtliche Funktionen eines Trace-Debuggers zur Verfügung stellt. Sie ermöglicht es, die Ausführung des Prüflings an bestimmten Positionen im Programmablauf zu unterbrechen und erlaubt es, den Programmzustand des unterbrochenen Programms vollständig zu untersuchen. Da alle Funktionalitäten der JPDA durch die Java-VM implementiert sind,

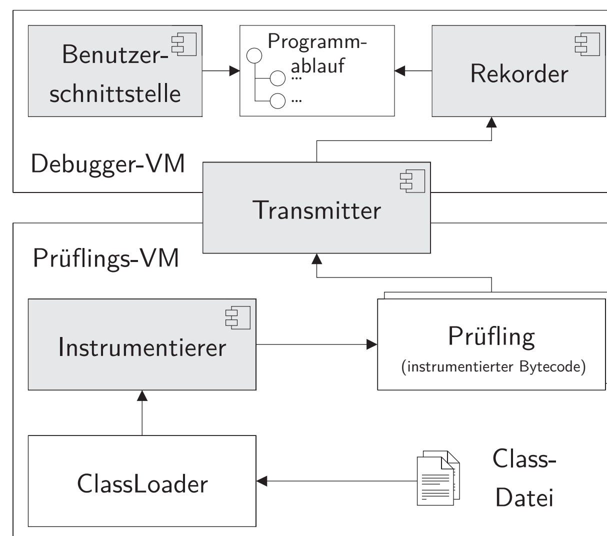


Abbildung 7.1: Architektur von JHyde.

ist es für den Einsatz der JPDA nicht notwendig, den Quelltext bzw. den Bytecode des untersuchten Programms zu verändern. Um den Ablauf eines Programms mit JPDA zu analysieren, muss die Java-VM, die den Prüfling ausführt, mit bestimmten Parametern gestartet werden. Der Ablauf eines Programms kann dann mithilfe von JPDA während der Ausführung beobachtet werden. Hierfür sendet JPDA Ereignisse, die den Programmablauf beschreiben. Für die Implementierung von JHyde wäre es unter diesen Gesichtspunkten ebenfalls sinnvoll, auf die Funktionalitäten von JPDA zurückzugreifen. Allerdings ist JPDA nicht in der Lage alle benötigten Arten von Ereignissen über den Programmablauf zu erzeugen. So ist es mit JPDA zum Beispiel nicht möglich, Informationen über die überdeckten DU-Ketten oder Bedingungen des Kontrollflussgraphen zu sammeln. Aus diesem Grund ist der Einsatz von JPDA für die Implementierung von JHyde nicht geeignet.

JHyde basiert auf der Technik der Instrumentierung. Im Rahmen der Softwareentwicklung bezeichnet Instrumentierung die Anreicherung des Quelltextes oder des Programmcodes um zusätzliche Informationen bzw. Anweisungen, mit denen das Verhalten des Programms untersucht werden kann. Instrumentierung wird zum Beispiel im Rahmen des Softwaretestens verwendet, um die Testabdeckung von Testfällen zu bestimmen. Durch die Instrumentierung des untersuchten Programms lässt sich genau spezifizieren, welche Informationen über den Programmablauf aufgezeichnet werden.

Wie die Abbildung 7.1 zeigt, besteht JHyde im Wesentlichen aus den folgenden vier Modulen bzw. Funktionseinheiten:

Instrumentierer - Der Instrumentierer wird in der Prüflings-VM ausgeführt und ist für die Instrumentierung des Prüflings verantwortlich. Grundsätzlich ist die Instrumentierung der Java-Klassen des untersuchten Programms ein dynamischer Prozess, der während des Ladens einer Klasse durch den `ClassLoader` angestoßen wird. Der `ClassLoader` lädt den ursprünglichen Bytecode einer Java-Klasse und übergibt ihn anschließend an den Instrumentierer. Dieser ergänzt den Bytecode um zusätzliche Anweisungen mit denen die Ausführung der Methoden der Klasse verfolgt werden kann. Im Anschluss wird der instrumentierte Bytecode in den Speicher der Java-VM geladen. Neben dem ursprünglichen Programmablauf bewirkt die Ausführung des instrumentierten Bytecodes, dass Ereignisse über den Programmablauf des Prüflings an den Transmitter gesendet werden.

Transmitter - Der Transmitter ist über beide virtuelle Maschinen verteilt und übernimmt den Transport der Ereignisse von der Prüflings-VM zur Debugger-VM. In der Debugger-VM reicht der Transmitter die empfangenen Ereignisse an den Rekorder weiter.

Rekorder - Der Rekorder strukturiert und speichert die empfangenen Informationen und erzeugt ein Modell, welches den Programmablauf des Prüflings repräsentiert. Das Modell wird anschließend in der Benutzerschnittstelle angezeigt.

Benutzerschnittstelle - Die Benutzerschnittstelle von JHyde erfüllt die Aufgabe, das aufgezeichnete Modell anzuzeigen und zu durchsuchen. Darüber hinaus ermöglicht sie dem Benutzer die Durchführung des hybriden Debugging-Prozesses.

7.2 Transmitter

Im Folgenden werden zunächst die Ereignisschnittstelle und die Architektur des Transmitters beschrieben. Im Anschluss wird die Funktionsweise des Sendeprozesses und des Empfangsprozesses erläutert. Zuletzt werden die wichtigsten Konfigurationsmöglichkeiten des Transmitters erläutert.

7.2.1 Die Ereignisschnittstelle

Der Transmitter legt eine standardisierte Schnittstelle für die Ereignisse fest, die während des Ablaufs eines untersuchten Programms auftreten können. Die Schnittstelle wird durch das Java-Interface `IJHydeEvents` definiert und umfasst alle Ereignisse über den Programmablauf, die für die Durchführung des hybriden Debugging-Prozesses (vgl. Abschnitt 4.4) benötigt werden. Das Interface `IJHydeEvents` wird sowohl in der Prüflings-VM als auch in der Debugger-VM verwendet. In der Prüflings-VM werden die Methoden der `IJHydeEvents`-Schnittstelle durch die instrumentierten Klassen des Prüflings aufgerufen, um die relevanten Ereignisse des Programmablaufs an den Transmitter zu übermitteln. Der Transmitter wandelt die Ereignisse in einen Datenstrom um, der an die Transmitter-Komponente der Debugger-VM geschickt wird. Dort wird der empfangene Datenstrom ausgelesen und in die durch die Schnittstelle `IJHydeEvents` definierten Ereignisse zurücktransformiert. Für die Umwandlung des Datenstroms liest die Transmitter-Komponente der Debugger-VM den empfangenen Datenstrom aus und ruft die Methoden der `IJHydeEvents`-Schnittstelle auf, die durch die Rekorder-Komponente implementiert wird. Die Rekorder-Komponente kann die so empfangenen Ereignisse dann für den weiteren Debugging-Prozess verarbeiten.

In den folgenden Abschnitten werden die durch die Schnittstelle `IJHydeEvents` definierten Ereignisse vorgestellt. In Abhängigkeit von der Art des Ereignisses sind die Methodenaufrufe der Schnittstelle in 6 Gruppen unterteilt. Die erste Gruppe (vgl. Abschnitt 7.2.1.1) enthält die Ereignisse, welche die statische Klassenstruktur des untersuchten Programms beschreiben. In den Gruppen 2–4 sind Ereignisse zusammengefasst, die das Erzeugen von Objekten (Abschnitt 7.2.1.2), das Lesen von Variablen (Abschnitt 7.2.1.3) und das Schreiben von Variablen (Abschnitt 7.2.1.4) betreffen. Die Ereignisse der drei Gruppen beschreiben die Datenflüsse eines Programmablaufs. Zu den Gruppen 5 und 6 gehören Ereignisse, die den Kontrollfluss eines Programmablaufs beschreiben. Dabei fasst die vierte Gruppe (vgl. Abschnitt 7.2.1.5) alle Ereignisse zusammen, die sich auf Methodenaufrufe beziehen. Zur fünften Gruppe (vgl. Abschnitt 7.2.1.6) gehören alle übrigen Ereignisse, die den Kontrollfluss betreffen.

7.2.1.1 Klassenstruktur

Für den hybriden Debugging-Prozess werden neben den dynamischen Informationen über den Kontroll- und den Datenfluss eines Programmablaufs auch statische Informationen über die Struktur des untersuchten Programms benötigt. Die Schnittstelle `IJHydeEvents` definiert insgesamt 6 Ereignisse, mit denen die Klassenstruktur des untersuchten Programms beschrieben wird. Diese Ereignisse werden dynamisch während der Ausführung des Prüflings erzeugt und versendet. Dies geschieht unmittelbar, bevor sie benötigt werden, d. h., wenn zum ersten Mal auf ein Element einer Klassenstruktur zugegriffen wird.

Die folgenden Methoden der `IJHydeEvents`-Schnittstelle dienen zur Beschreibung der Klassenstruktur des untersuchten Programms:

void `registerClass(`**int** `modifiers, String sign, String superSign,` **int** `classId)` - Das durch diese Methode beschriebene Ereignis informiert über eine Java-Klasse, die im folgenden Programmablauf verwendet wird. Über die Argumente der Methode werden die Modifizierer [10] (`modifiers`), die Signaturen, d. h. die vollqualifizierten Klassennamen, der Klasse (`sign`) sowie der Oberklasse (`superSign`) und die ID der Klasse (`classId`) festgelegt. Das Argument `classId` ordnet der beschriebenen Klasse eine eindeutige ID zu. Falls die Klasse durch nachfolgende Ereignisse referenziert wird, so geschieht dies durch die Verwendung der `classId`.

void `registerArraySignature(String signature,` **int** `arrayTypeId)` - Registriert für den folgenden Programmablauf einen Array-Typen. Die Signatur (`signature`) legt den Typen des

[10] Die Zugriffsmodifizierer lauten `private`, `protected`, „`package private`“ und `public`. Diese werden für Methoden und Attribute verwendet. Die Modifizierer „`package private`“ und `public` können darüber hinaus für Klassen verwendet werden. Ein Element in Java ist „`package private`“, wenn es keinen expliziten Modifizierer besitzt. In diesem Fall ist das Element nur innerhalb desselben Pakets sichtbar. Zu weiteren Modifizierern zählen `abstract`, `final`, `static`, `strictfp` und `native`. Von diesen ist `static` ausschließlich für Methoden und Attribute, `strictfp` ausschließlich für Klassen und Methoden und `native` ausschließlich für Methoden zulässig. Durch die `IJHydeEvents`-Schnittstelle werden alle Modifizierer vor dem Versenden als `int`-Werte kodiert.

Arrays eindeutig fest. Die Notation ist konform zu der Grammatik, die von der JVM-Spezifikation [151] für Array-Typen vorgegeben wird.

void `registerAttribute(`**int** `modifiers,` **int** `classId, String type, String name,` **int** `attributId)` - Die Methode informiert den Debugger über ein Klassenattribut der Klasse mit der ID `classId`. Die Beschreibung des Attributs umfasst die Modifizierer (`modifiers`), den Datentypen des Attributs (`type`) und den Namen (`name`) des Attributs. Die Notation zur Beschreibung des Datentyps folgt der in der JVM-Spezifikation festgelegten Grammatik für Attributtypen. Über das Argument `attributId` wird dem Attribut eine eindeutige ID zugeordnet.

void `registerMethod(`**int** `modifiers,` **int** `classId, String name, String desc,` **int** `maxLocals,` **int** `size,` **int** `mId)` - Registriert für die Klasse mit der ID `classId` eine neue Methode mit den Modifizierern (`modifiers`), Methodennamen (`name`) und Methodendeskriptor (`desc`). Der Methodendeskriptor legt die Argumenttypen der Methode sowie deren Reihenfolge fest. Das Format des Methodendeskriptors ist in der JVM-Spezifikation definiert. Methodenname und Methodendeskriptor ergeben zusammen die Methodensignatur einer Methode. Innerhalb einer Klasse kann jede Methode über ihre Signatur eindeutig identifiziert werden. Darüber hinaus werden die Anzahl der lokalen Variablen (`maxLocals`) und die Anzahl der Einträge in der Zeilennummerntabelle[11] (`maxNumber`) definiert. Durch diese Parameter wird der Rekorder-Komponente mitgeteilt, wie viel Speicherplatz für lokale Variablen und Einträge in der Zeilennummerntabelle reserviert werden muss. Die lokalen Variablen und die Einträge der Zeilennummerntabelle werden mit den beiden nachfolgenden Methoden registriert. Über das Argument `mId` wird jeder registrierten Methode eine eindeutige ID zugeordnet.

void `registerLocalVariable(`**int** `mId, String type, String name,` **int** `idx)` - Registriert für die Methode mit der

[11]Die Zeilennummerntabelle (line number table) wird durch den Compiler für jede Methode erzeugt und in der `class`-Datei gespeichert. Über diese Tabelle kann jede Bytecode-Instruktion der Zeile des Quelltextes zugeordnet werden, aus der sie durch den Compiler generiert wurde.

ID `mId` eine lokale Variable mit Namen (`name`) und dem Datentyp (`type`). Jeder lokalen Variablen wird ein Index zugeordnet (`idx`), über den sie innerhalb der zugehörigen Methode eindeutig identifiziert werden kann.

`void` `registerLineNumber(`**`int`** `mId,` **`int`** `idx,` **`int`** `offset,` **`int`** `lineNumber)` - Registriert für die Methode mit der ID `mId` einen Eintrag in der Zeilennummerntabelle. Jeder Eintrag in der Tabelle besteht aus einem Zeilenindex (`tableIdx`), der die Position des Eintrags in der Tabelle definiert, einem Bytecode-Offset (`offset`) und einer Zeilennummer (`lineNumber`). Der Bytecode-Offset legt die Position in der Instruktionsliste der Methode fest, ab der dieser Eintrag Gültigkeit hat. Ein Eintrag in der Zeilennummerntabelle ordnet somit jedem Ereignis, dessen Bytecode-Offset größer oder gleich dem Argument `offset` und gleichzeitig kleiner als der `offset`-Wert des nachfolgenden Eintrags ist, die Zeile `lineNumber` des Quelltextes zu.

7.2.1.2 Erzeugen von Objekten

Die Erzeugung neuer Objekte während der Laufzeit des untersuchten Programms wird durch spezielle Ereignisse mitgeteilt. Bei den erzeugten Objekten wird dabei zwischen Instanzen einer Klasse und Array-Instanzen unterschieden. Die im Folgenden beschriebenen Methoden besitzen die Argumente `tId` und `offset`. Über die Thread-ID `tId` wird das jeweilige Ereignis eindeutig einem bestimmten Thread in der Programmausführung zugeordnet.[12] Über das Argument `offset` wird ein Ereignis der exakten Position in der Instruktionsliste der Methode zugeordnet, an der es aufgetreten ist. In Verbindung mit der Zeilennummerntabelle kann in der Benutzeroberfläche von JHyde anhand des `offset` die Position im Quelltext markiert werden, an der ein Ereignis aufgetreten ist.

[12] Da jedes Kontroll- und Datenfluss-Ereignis eindeutig dem erzeugenden Thread zugeordnet ist, können auch nebenläufige Programme mit mehreren Threads grundsätzlich aufgezeichnet werden. Das Debugging von Programmen mit mehreren Threads bzw. Ausführungssträngen ist jedoch äußerst komplex und nicht Gegenstand dieser Arbeit. Die Debugging-Methoden von JHyde sind daher auf das Debugging von sequenziell ausgeführten Programmen beschränkt.

void createArray(**int** length, **long** oId, **long** tId, **int** arrayTypeId, **int** offset)- Signalisiert die Erzeugung eines neuen Arrays der Länge length mit der Objekt-ID oId. Über die Objekt-ID kann das Array in nachfolgenden Ereignissen eindeutig referenziert werden. Der Typ des Arrays wird durch die ID arrayTypeId festgelegt.

void createObject(**long** oId, **long** tId, **int** classId, **int** offset) - Signalisiert die Erzeugung einer neuen Instanz der Klasse mit der ID classId. Die Objekt-ID der Instanz ist oId.

7.2.1.3 Lesen von Variablen

Wird während des Ablaufs des untersuchten Programms der Wert einer Variablen ausgelesen, so wird dies durch ein entsprechendes Ereignis signalisiert. Die Schnittstelle IJHydeEvents unterscheidet zwischen drei Arten von Variablen: Attributen, Array-Elementen und lokalen Variablen.

void attributeUse([**long** oId,]**long** tId, **int** offset, **int** attributId) - Teilt das Auslesen des Attributs mit der ID attributId mit. Das Argument oId gibt die Objekt-ID des Objekts an, dessen Attribut ausgelesen wurde. Falls es sich bei dem ausgelesenen Attribut um ein Klassenattribut handelt, entfällt dieses Argument. Die attributeUse-Methode ist somit durch eine Deklaration für Objektattribute und eine Deklaration für Klassenattribute überladen.

void arrayElementUse(**int** elementIdx, **long** oId, **long** tId, **int** offset) - Signalisiert das Auslesen des Wertes des Array-Elements mit dem Index elementIdx aus dem Array mit der Objekt-ID oId.

void localVariableUse(**long** tId, **int** offset, **int** idx) - Informiert über das Auslesen der lokalen Variablen mit dem Index idx in der gegenwärtig durch den Thread mit der ID tId ausgeführten Methode.

7.2.1.4 Schreiben von Variablen

Neben lesendem Zugriff wird auch über schreibenden Zugriff auf Variablen informiert. Die Methoden zur Signalisierung von Wertänderungen übermitteln zusätzlich zu den Informationen zur Identifikation der geänderten Variablen den zugewiesenen Wert. Der Datentyp des zugewiesenen Wertes ist entweder ein primitiver Datentyp (`boolean`, `byte` usw.) oder ein Referenztyp. Jeder der folgenden drei Methoden ist durch die Schnittstelle `IJHydeEvents` daher insgesamt neunmal deklariert, da es acht primitive Datentypen und einen Referenztypen gibt. Referenzen werden in Form der Objekt-ID, deren Datentyp `long` ist, übermittelt. In den Deklarationen dient die Zeichenfolge „`XXX`" als Platzhalter für die verschiedenen Datentypen.

void `arrayElementChangeXXX(XXX value,` **int** `elementIdx,` **long** `oId,` **long** `tId,` **int** `offset)` - Signalisiert, dass dem Array-Element mit dem Index `elementIdx` des Arrays mit der Objekt-ID `oId` der Wert `value` zugewiesen wurde.

void `attributeChangeXXX(xxx value, [`**long** `oId, ]`**long** `tId,` **int** `offset,` **int** `attributeId)` - Dem Attribut mit der ID `attributeId` wurde der Wert `value` zugewiesen. Falls das Attribut ein Instanzattribut ist, wird die ID des Objekts (`oId`) ebenfalls übergeben.

void `localVariableChangeXXX(xxx value,` **long** `tId,` **int** `offset,` **int** `idx)` - Teilt mit, dass der lokalen Variablen, die in der gegenwärtig durch den Thread mit der ID `tId` ausgeführten Methode den Index `idx` besitzt, der Wert `value` zugewiesen wurde.

7.2.1.5 Methodenaufruf

Die Schnittstelle `IJHydeEvents` verfügt über drei Methoden, mit denen die Abfolge der Methodenaufrufe des untersuchten Programms übermittelt werden kann. Die Schnittstelle sieht vor, den Beginn eines Methodenaufrufs, das Ende der Initialisierung eines Methodenaufrufs und das Ende eines Methodenaufrufs zu signalisieren. Diese Informationen sind ausreichend, um den vollständigen Baum der Methodenaufrufe zu rekonstruieren.

void `beginMethodCall(`**long** `tId,` **int** `mId)` - Signalisiert den Beginn eines neuen Aufrufs der Methode mit der ID `mId`.

void `endMethodCallInit(`**long** `tId)` - Informiert über das Ende der Initialisierung des Methodenaufrufs, der zuletzt für den Thread mit der ID `tId` durch den Aufruf der Methode `beginMethodCall` gestartet wurde. In der Initialisierungsphase werden die Werte der übergebenen Argumente durch `localVariableChangeXXX`-Ereignisse übermittelt. Der Aufruf der `endMethodCallInit`-Methode trennt somit die Initialisierung der Argumente von der übrigen Ausführung des Methodenaufrufs.

void `endMethodCall(`**long** `tId,` **int** `bytecodeId)` - Signalisiert das Ende des Methodenaufrufs, der zuletzt für den Thread mit der ID `tId` durch den Aufruf der Methode `beginMethodCall` gestartet wurde.

7.2.1.6 Sonstiger Kontrollfluss

Neben den Ereignissen, die sich auf die aufgerufenen Methoden des Prüflings beziehen, gibt es weitere Ereignisse, die ebenfalls den Kontrollfluss repräsentieren.

void `conditionCovered(`**long** `tId,` **int** `offset)` - Signalisiert die Überdeckung einer Bedingung im Kontrollflussgraphen der gegenwärtig durch den Thread mit der ID `tId` ausgeführten Methode. Innerhalb einer Methode besitzt jedes Überdeckungsereignis, das aus der überdeckten Bedingung und der Art der Überdeckung (`true` oder `false`) besteht, ein eindeutiges `offset`.

void `loopBody(`**long** `tId,` **int** `offset)` - Dieses Ereignis bedeutet, dass die Ausführung den Anfang eines Schleifenrumpfes erreicht hat.

void `loopExit(`**long** `tId,` **int** `offset)` - Signalisiert das Ende einer Schleifenausführung.

void `beginHide(`**long** `tId)` - Markiert für den Thread mit der ID `tId` den Beginn einer Folge von Ereignissen, die nicht in der Ereignisansicht (vgl. 6.1.3) dargestellt werden sollen. Dies ist zum Beispiel der Fall, wenn der Ablauf des untersuchten Programms auf ein Objekt

trifft, dessen Existenz dem Debugger bisher nicht durch Ereignisse mitgeteilt wurde. In diesem Fall wird ein `createObject`-Ereignis erzeugt. Da die tatsächliche Erzeugung des Objekts jedoch nicht zum gegenwärtigen, sondern zu einem früheren Zeitpunkt in der Ausführung aufgetreten ist, soll die Erzeugung des Objekts in der Ereignisliste des gegenwärtigen Methodenaufrufs nicht angezeigt werden. Dies wird durch die Definition einer nicht sichtbaren Sektion erreicht. Die nachträgliche Übermittlung von unbekannten Objekten wird im Abschnitt 7.3.3, der das Instrumentierungsschema erläutert, genauer beschrieben.

void `endHide(`**long** `tId)` - Markiert das Ende der nicht sichtbaren Folge von Ereignissen, deren Beginn durch den letzten Aufruf von `beginHide` für den Thread mit der ID `tId` markiert wurde.

void `stopDebugging()` - Signalisiert das Ende der Ausführung des untersuchten Programms.

7.2.2 Architektur

Der Transmitter verwendet zum Verschicken von Nachrichten zwischen der Debugger-VM und der Prüflings-VM eine Socket-Verbindung [30, S. 7 f.]. Ein Socket ist der Endpunkt einer bidirektionalen Interprozesskommunikation auf der Grundlage eines Internetprotokoll-basierten [68] Computernetzwerks. Ein Socket wird durch das verwendete Protokoll, die Internetadresse und einen Port eindeutig identifiziert. Die Verwendung einer Socket-Verbindung ermöglicht eine ortstransparente Übertragung der Ereignisse von der Prüflings-VM zur Debugger-VM. Für den Transmitter ist es somit unbedeutend, ob die beiden virtuellen Maschinen auf demselben oder auf entfernten Rechnern laufen. Grundsätzlich sind auch andere Arten des Datenaustauschs zwischen Prüfling und Debugger, zum Beispiel durch die Verwendung von Java RMI [60] oder des Dateisystems, denkbar. Im Vergleich zu einer RMI-basierten Verbindung ist die Socket-Verbindung jedoch im Bezug auf die Datentransferrate wesentlich effizienter und die Verwendung des Dateisystems bietet keine Ortstransparenz.

Der Aufbau des Transmitters ist im Klassendiagramm der Abbildung 7.2 dargestellt. Die grau hinterlegten Klassen gehören zur Java-API [145], die

übrigen Klassen sind Teil des Transmitter-Pakets. Die zentrale Schnittstelle des Transmitters, welche die Struktur sämtlicher Ereignisse definiert, die während eines untersuchten Programmablaufs auftreten können, ist das Interface `IJHydeEvents` (vgl. Abschnitt 7.2.1). Darüber hinaus definiert der Transmitter die beiden Schnittstellen `IJHydeEventSender` und `IJHydeEventReceiver`.

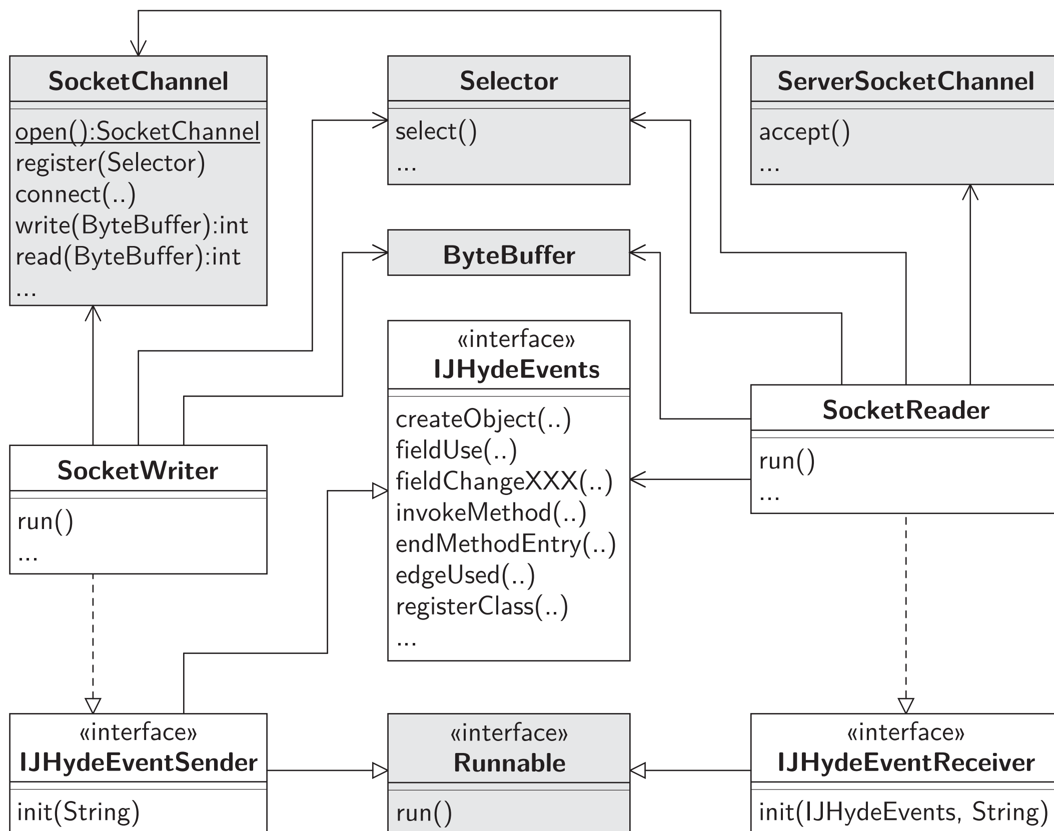


Abbildung 7.2: Klassendiagramm des Transmitters.

Die Schnittstelle `IJHydeEventSender` erweitert die Interfaces `IJHydeEvents` und `Runnable` und definiert die Operationen der Sendeeinheit. Die Sendeeinheit ist in der Prüflings-VM dafür zuständig, alle Ereignisse, die ihr über die Schnittstelle `IJHydeEvents` übermittelt werden, an die Empfangseinheit zu übermitteln. Da die Sendeeinheit das Interface `Runnable` implementiert, kann sie als eigenständiger Thread in der Prüflings-VM ausgeführt werden. Durch die nebenläufige Ausführung

von Prüfling und Sendeeinheit können die Übermittlung der Ereignisse vom Prüfling an die Sendeeinheit und das Versenden der Ereignisse durch die Sendeeinheit parallel zueinander ablaufen. Die Ausführung des Prüflings muss daher nicht blockiert werden, bis das Ereignis an den Empfänger versendet wurde. Darüber hinaus kann die Sendeeinheit in einem Sendevorgang mehrere Nachrichten gleichzeitig versenden. Aufgrund der nebenläufigen Ausführung wird somit die Ausführungsgeschwindigkeit des Prüflings durch den Sendevorgang weniger stark reduziert.

Die Empfangseinheit in der Debugger-VM muss das Interface `IJHydeEventReceiver` implementieren. `IJHydeEventReceiver` erweitert ebenfalls die `Runnable`-Schnittstelle und definiert die Methode `init` zur Initialisierung. In der Debugger-VM wird die Empfangseinheit ebenfalls in einem separaten Thread ausgeführt, damit die Oberfläche des Debuggers während der Aufzeichnung des Programmablaufs des Prüflings nicht blockiert. Über die `init`-Methode werden der Empfangseinheit ein Rekorder, an den die empfangenen Ereignisse weiterzuleiten sind, und ein `String` zur Konfiguration der Verbindung zum Empfänger übergeben. Die Empfangseinheit leitet die empfangenen Ereignisse an den Rekorder weiter, indem sie die Methoden der `IJHydeEvents`-Schnittstelle aufruft.

Die Schnittstellen `IJHydeEventSender` und `IJHydeEventReceiver` werden durch die Klassen `SocketWriter` und `SocketReader` implementiert. Diese verbinden sich über eine Socket-Verbindung, um die Ereignisse von der Prüflings-VM zur Debugger-VM zu übertragen. Für die Übertragung der Ereignisse verwenden `SocketWriter` und `SocketReader` die Klassen `SocketChannel`, `Selector`, `ByteBuffer` und `ServerSocketChannel`, die Teil des `java.nio`-Paketes [65] sind.

Neben der implementierten Socket-Verbindung können andere Verbindungsarten durch weitere Klassen, die `IJHydeEventSender` bzw. `IJHydeEventReceiver` implementieren, realisiert werden. Die für die Ereignisübermittlung verwendeten Klassen lassen sich in den Konfigurationseinstellungen des Transmitters (7.2.5) festlegen.

7.2.3 Sendeprozess

Das Versenden der Nachrichten durch die Klasse `SocketWriter` wird in der Prüflings-VM durch einen separaten Thread ausgeführt. Bevor dieser Thread gestartet werden kann, muss der `SocketWriter` zunächst initialisiert werden. Wie im Sequenzdiagramm der Abbildung 7.3 dargestellt, geschieht dies durch den Aufruf `init(destination)`. Über das Argument `destination` wird dem `SocketWriter` die Adresse bzw. die URL des Empfängers mitgeteilt. Anschließend erzeugt der `SocketWriter` eine `Selector`-, eine `SocketChannel`- und eine `ByteBuffer`-Instanz. Der `Selector` wird beim Versenden der Ereignisse benötigt, um auf die Schreibberechtigung für den `SocketChannel` zu warten. Der `SocketChannel` repräsentiert eine Datenstrom-basierte Verbindung zu einem anderen Socket. Über den `SocketChannel` können Daten an eine entfernte `SocketChannel`-Instanz, zu der eine aktive Verbindung besteht, geschickt werden. Der `ByteBuffer` dient als Pufferspeicher für die Daten, die noch über den `SocketChannel` zu verschicken sind.

Nach der Erzeugung der benötigten Instanzen des `java.nio`-Paketes wird der `Selector` durch den Aufruf der `register`-Methode beim `SocketChannel` registriert und kann anschließend dazu verwendet werden, um auf die Verfügbarkeit der `SocketChannel`-Instanz für Schreiboperationen zu warten. Im Anschluss wird durch den Aufruf von `connect(dest)` eine Verbindung zum `SocketReader` in der Debugger-VM aufgebaut. Nachdem die Verbindung zustande gekommen ist, wird der `SocketWriter`-Thread, der für die Übermittlung der Ereignisse an den `SocketReader` zuständig ist, initialisiert. Hierfür wird eine neue Instanz der Klasse `Thread` erzeugt. Über den Konstruktor wird dem `Thread` die `SocketWriter`-Instanz, die das `Runnable`-Interface implementiert, übergeben. Der Aufruf der Methode `start` führt dazu, dass die Ausführung des `SocketWriter`-Threads gestartet wird. Da der Aufruf von `start` asynchron ist, kann die Ausführung der Methode `init` unmittelbar im Anschluss beendet werden. In der Java-VM führt der Aufruf der `start`-Methode dazu, dass ein neuer Ausführungsstrang gestartet wird. Dieser neue Ausführungsstrang ruft dann die Methode `run` von `SocketWriter` auf. Die Details der Ausführung des `SocketWriter`-Threads sind im nachfolgenden Sequenzdiagramm separat dargestellt.

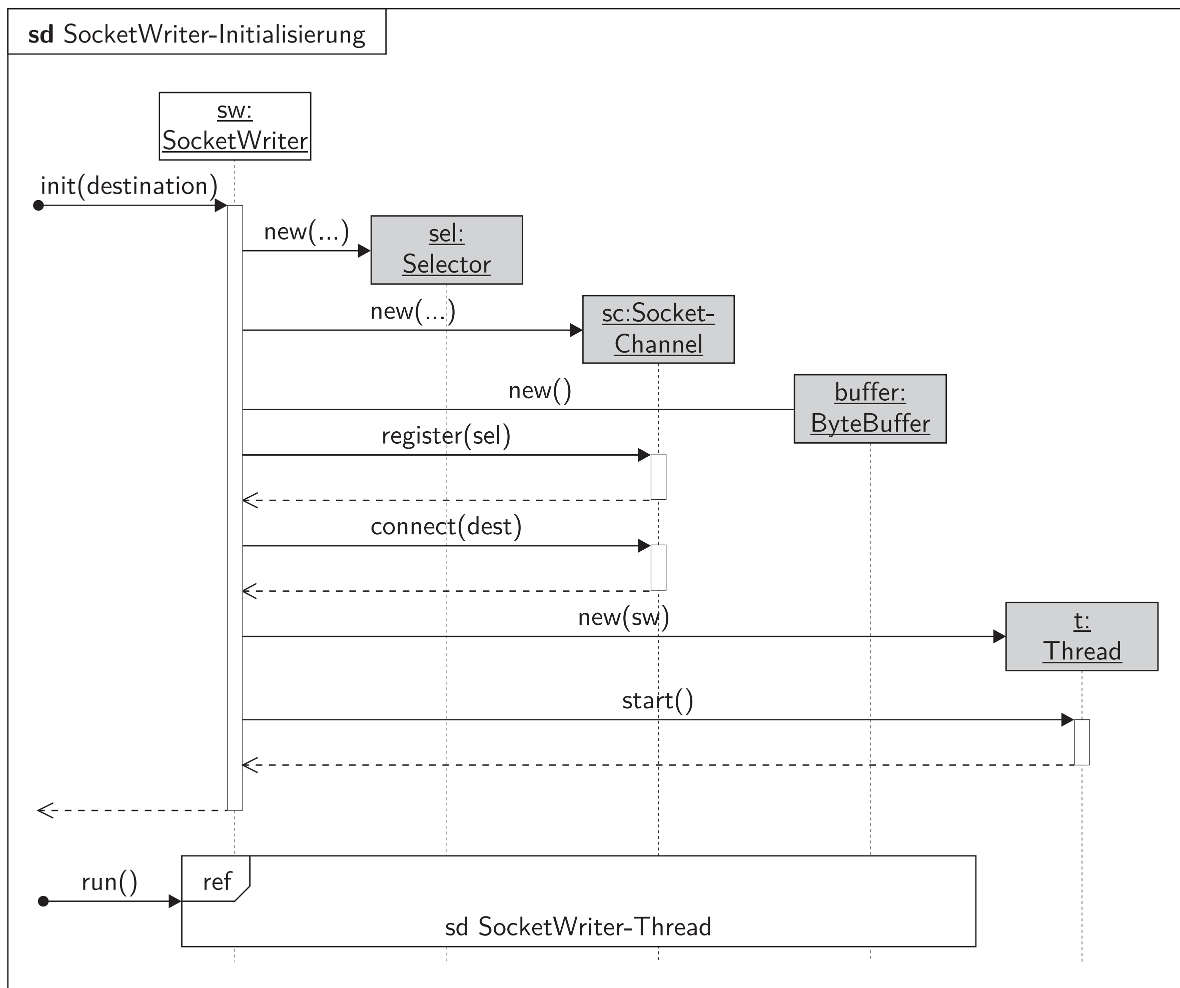


Abbildung 7.3: Sequenzdiagramm der Initialisierung des `SocketWriter`.

Der `SocketWriter`-Thread ist im Wesentlichen dafür zuständig, die Inhalte des `ByteBuffer` in den `SocketChannel` zu übertragen. Die Ausführung des Threads wird beendet, wenn die Ausführung des Prüflings beendet wurde und alle Daten vom `ByteBuffer` zum `SocketChannel` übertragen wurden.

Der Ablauf des `SocketWriter`-Threads wird in der Abbildung 7.4 veranschaulicht. Im Wesentlichen besteht die Ausführung der `run`-Methode aus einer Schleife, die solange durchlaufen wird, wie die Ausführung des Prüflings noch nicht beendet ist oder der `ByteBuffer` noch zu sendende Daten enthält. Innerhalb des Schleifenrumpfes wird zunächst durch den Aufruf von `select` gewartet, bis der `SocketChannel` für Schreiboperationen zur Verfügung steht. Auf die Freigabe für Schreiboperationen muss zum Beispiel gewartet werden, wenn der Empfangspuffer des `SocketChannel` in

der Debugger-VM vollgelaufen ist und deshalb keine weiteren Daten mehr empfangen kann.

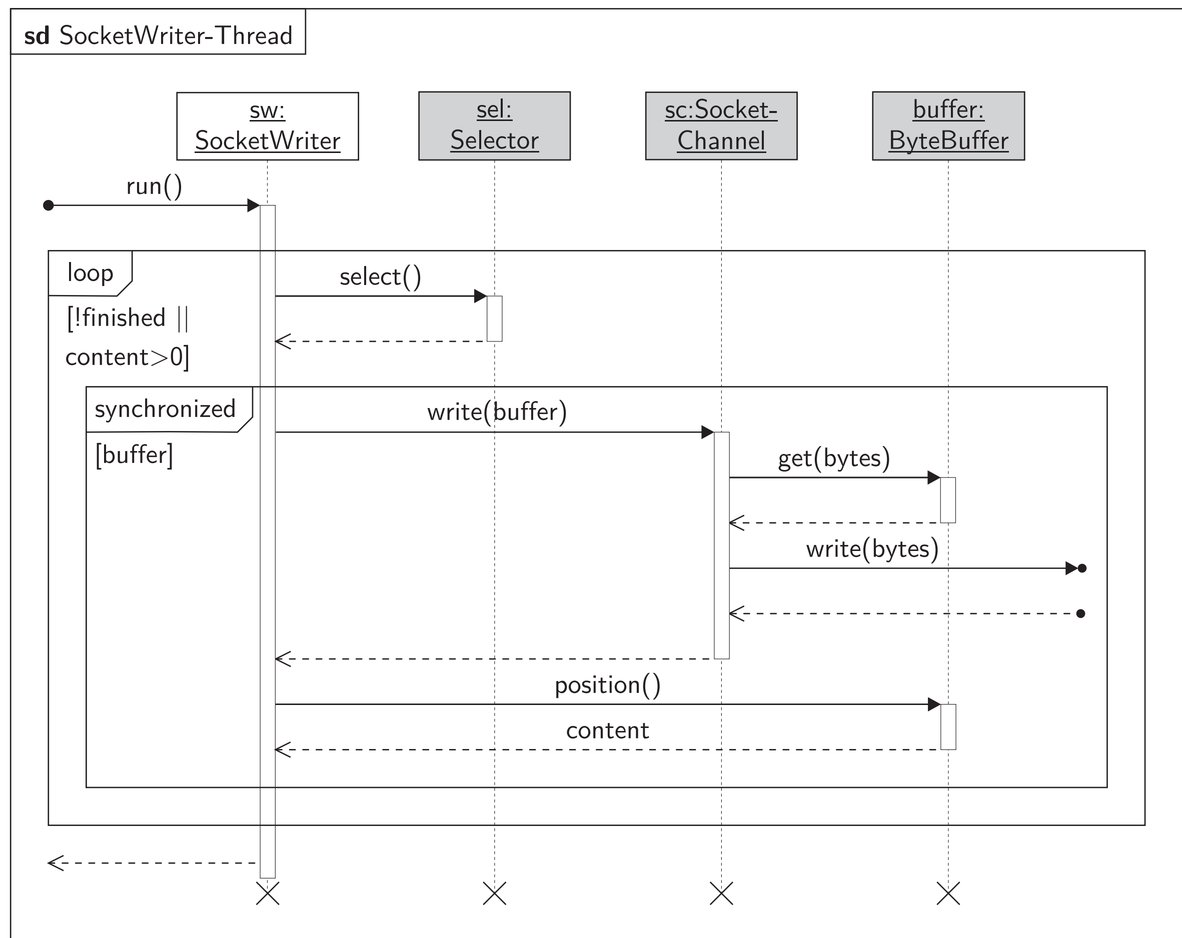


Abbildung 7.4: Sequenzdiagramm des `SocketWriter`-Threads.

Nach dem Erhalt der Schreibberechtigung werden Daten aus dem `Byte-Buffer` in den `SocketChannel` übertragen. Um gleichzeitig auftretende Lese- und Schreiboperationen auf dem `ByteBuffer` zu unterbinden, werden die folgenden Operationen synchronisiert ausgeführt. Auf diese Weise wird sichergestellt, dass zu jedem Zeitpunkt immer nur einem einzigen Thread Zugriff auf den `ByteBuffer` gewährt wird. Nachdem durch den Eintritt in den synchronisierten Abschnitt exklusiver Zugriff auf den `ByteBuffer` besteht, werden durch den Aufruf von `write(buffer)` Inhalte des `ByteBuffer` in den `SocketChannel`

geschrieben. Der `SocketChannel` veranlasst durch den Aufruf der Methode `getBytes(bytes)` den `ByteBuffer` dazu, eine Folge von Bytes in das übergebene Array `bytes` zu verschieben. Diese Byte-Folge wird im Anschluss durch den Aufruf von `write(bytes)` an den `SocketChannel` in der Debugger-VM gesendet. Nach der Beendigung des Schreibvorgangs wird mithilfe des Aufrufs der Methode `position` der Füllstand des `ByteBuffer` ermittelt. Anschließend wird mit der Schleifenbedingung überprüft, ob der Sendevorgang fortgesetzt werden muss. Die Ausführung des `SocketWriter`-Threads wird beendet, wenn die Ausführung des Prüflings beendet ist und der `ByteBuffer` keine zu sendenden Daten mehr enthält.

Im folgenden Sequenzdiagramm (vgl. Abbildung 7.5) ist dargestellt, wie die in der Ausführung des Prüflings auftretenden Ereignisse zum `SocketWriter` übermittelt werden. Der Prüfling sendet die Ereignisse zunächst an die Klasse `DebuggingEvent`. Die Klasse `DebuggingEvent` dient als Adapter und reicht die Ereignisse vom Prüfling an die Sendeeinheit weiter. Die durch den Prüfling aufgerufenen Methoden der Klasse `DebuggingEvent` sind statisch, damit sie sich durch den Instrumentierungsmechanismus einfacher in den Bytecode des Prüflings einfügen lassen. Auf den Ablauf der Instrumentierung wird im Abschnitt 7.3 näher eingegangen.

Die Methodenaufrufe des folgenden Sequenzdiagramms werden durch den Thread ausgeführt, der auch für die Ausführung des Prüflings verantwortlich ist. Wie bereits erwähnt, handelt es sich dabei nicht um den zuvor beschriebenen `SocketWriter`-Thread, welcher für die Übertragung der Ereignisse zum `SocketReader` in der Debugger-VM verantwortlich ist.

Zu Beginn der Ausführung des Prüflings wird die Klasse `DebuggingEvent` durch den Aufruf von `initSender` initialisiert. Während der Initialisierung wird mithilfe der Reflexionsmechanismen von Java [52] eine neue `IJHydeEventSender`-Instanz erzeugt. In der gegenwärtigen Konfiguration des Transmitters (vgl. 7.2.5) ist dies eine `SocketWriter`-Instanz. Anschließend wird, wie zuvor beschrieben, durch die Initialisierung des `SocketWriter` der `SocketWriter`-Thread gestartet. Damit ist die Initialisierung der Klasse `DebuggingEvent` abgeschlossen.

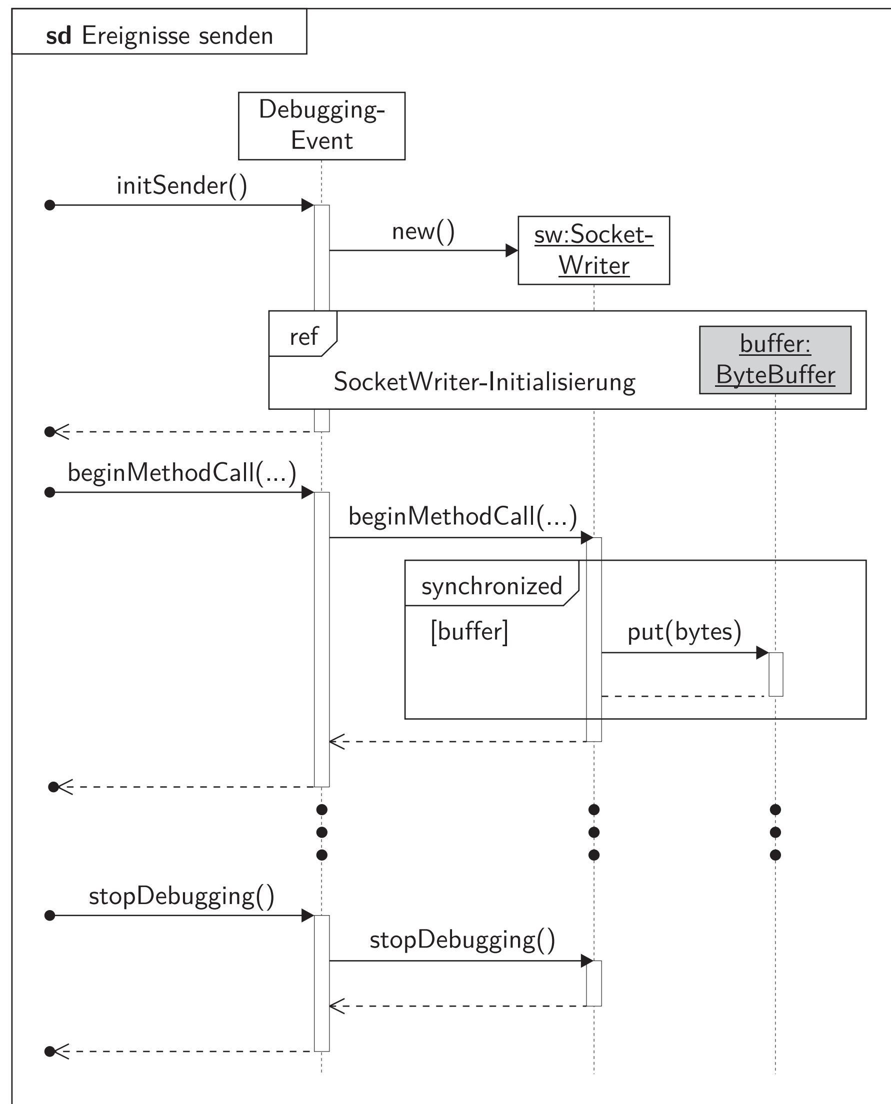


Abbildung 7.5: Sequenzdiagramm für die Übermittlung der Ereignisse vom Prüfling an den `SocketWriter`.

Während der folgenden Ausführung des Prüflings werden die auftretenden Ereignisse durch Aufrufe der Methoden der Klasse `DebuggingEvent` an den `SocketWriter` weitergeleitet. Im Sequenzdiagramm ist dies exemplarisch für das Ereignis, welches den Beginn eines neuen Methodenaufrufs signalisiert, veranschaulicht. Der Prüfling ruft zum Erzeugen des Ereignisses die Methode `beginMethodCall` der Klasse `DebuggingEvent` mit den entsprechenden Argumenten auf. Die Klasse `DebuggingEvent` leitet den Aufruf an die zuvor erzeugte `SocketWriter`-Instanz weiter. Daraufhin wartet der `SocketWriter`, bis ihm durch die Synchronisationsmechanismen exklusiver Zugriff auf den `ByteBuffer` gewährt wird

und darüber hinaus die verbleibende Kapazität des ByteBuffer für die folgende Übermittlung des Ereignisses ausreichend ist. Innerhalb des synchronisierten Abschnitts wandelt der SocketWriter die durch den Aufruf beginMethodCall übermittelten Argumente in eine Folge von Bytes um. Diese Folge wird durch den Aufruf von put an den ByteBuffer übermittelt. Anschließend ist die Übertragung des Ereignisses abgeschlossen. Im Verlauf der Ausführung des Prüflings werden weitere Ereignisse erzeugt, die nach demselben Schema an den ByteBuffer übertragen werden. In dem Sequenzdiagramm ist dies durch die Folge von Punkten angedeutet.

Am Ende der Ausführung des Prüflings wird die stopDebugging-Methode der Klasse DebuggingEvent aufgerufen. Dieser Aufruf wird ebenfalls an den SocketWriter weitergeleitet. Auf diese Weise wird der SocketWriter darüber informiert, dass die Ausführung des Prüflings beendet ist und somit keine weiteren Ereignisse zu übermitteln sind. Der SocketWriter-Thread, der die Ereignisse an den SocketReader in der Debugger-VM sendet, muss nun die eventuell noch verbleibenden Daten des ByteBuffer versenden, bevor er seine Ausführung ebenfalls beenden kann.

7.2.4 Empfangsprozess

In der Debugger-VM ist der SocketReader für den Empfang der durch den SocketWriter versendeten Ereignisse zuständig. Der SocketReader wird ebenfalls in einem separaten Thread ausgeführt, da während des Empfangs der Ereignisse die Benutzeroberfläche des Debuggers nicht blockieren soll. Um Ereignisse empfangen zu können, muss der SocketReader zunächst initialisiert werden. Dieser Vorgang ist im Sequenzdiagramm der Abbildung 7.6 dargestellt.

Zu Beginn der Initialisierung der Empfangseinheit wird in der Debugger-VM mithilfe der Java-Reflection-API [52] eine IJHydeEventReader-Instanz erzeugt. In der Standardkonfiguration des Transmitters ist dies eine SocketReader-Instanz. Durch den Aufruf von init(lst, port) werden dem SocketReader eine IJHydeEvents-Instanz und ein Port übergeben. Die IJHydeEvents-Instanz ist der Empfänger, an den der

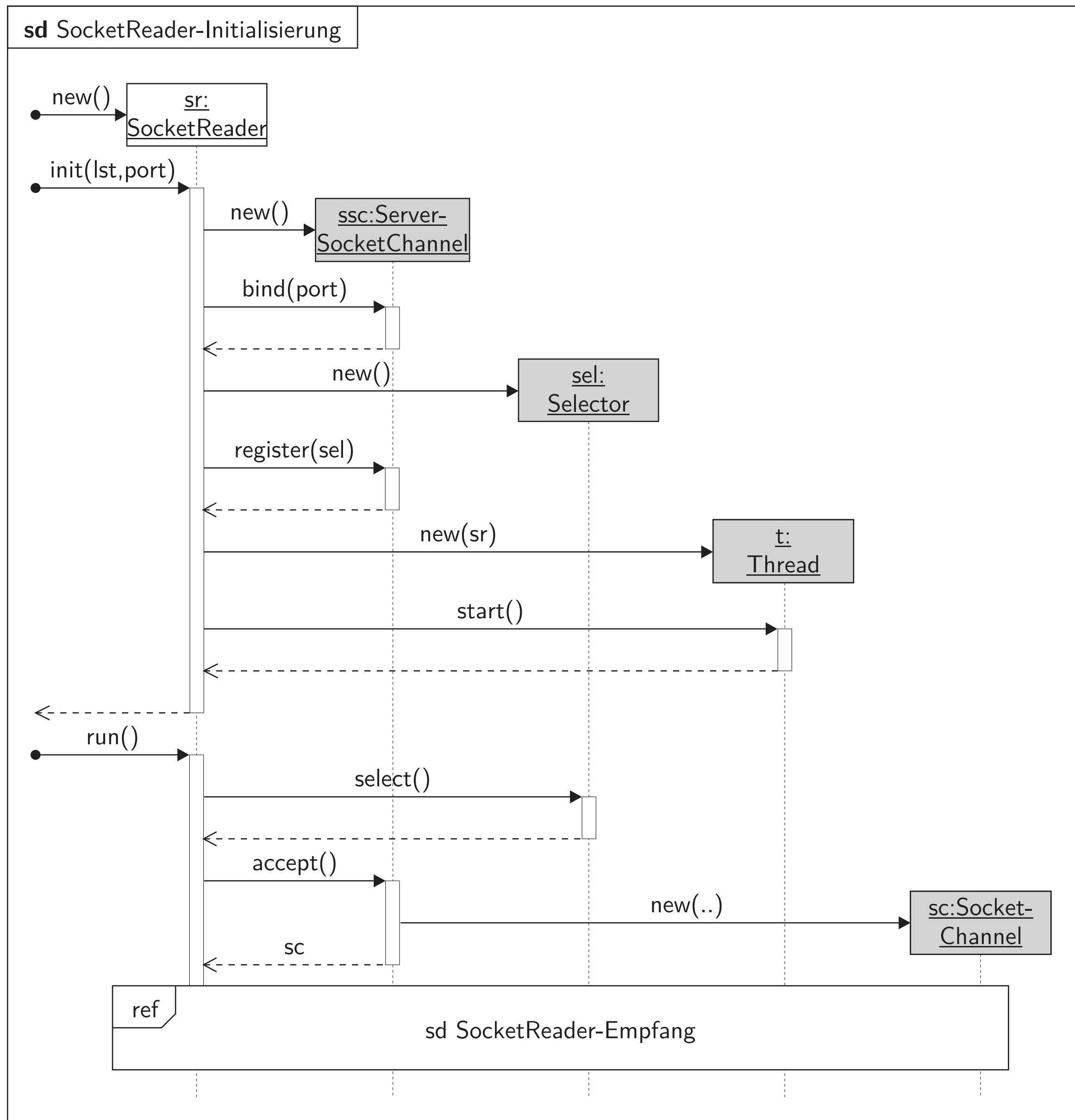


Abbildung 7.6: Sequenzdiagramm der `SocketReader`-Initialisierung.

`SocketReader` alle Ereignisse weiterleitet. Der Port dient für den Verbindungsaufbau zum `SocketWriter`. Zunächst wird durch die Ausführung der `init`-Methode eine neue `ServerSocketChannel`-Instanz erzeugt, die durch den Aufruf von `bind(port)` an die übergebene Portnummer gebunden wird. Der Server ist nun in der Lage Verbindungsanfragen durch den `SocketWriter` für die entsprechende Portnummer entgegenzunehmen. Um später auf die Verbindungsanfrage durch den `SocketWriter` zu

warten, wird eine neue `Selector`-Instanz erzeugt und durch den Aufruf von `register(sel)` beim `ServerSocketChannel` registriert.

Im letzten Schritt der Initialisierung wird die Ausführung des `SocketReader` durch einen dedizierten Thread vorbereitet. Zu diesem Zweck wird zunächst eine neue `Thread`-Instanz durch den Aufruf von `new(sr)` erzeugt. Im Konstruktor wird dabei die `SocketReader`-Instanz übergeben. Anschließend wird die Ausführung des neuen Threads durch den Aufruf der `start`-Methode gestartet. Der Aufruf der `init`-Methode durch den Thread der Benutzeroberfläche des Debuggers ist damit beendet.

In der Java-VM bewirkt der Aufruf der `start`-Methode der `Thread`-Instanz, dass ein neuer Ausführungsstrang erzeugt wird. Dieser Ausführungsstrang ruft die `run`-Methode der `SocketReader`-Instanz auf. Zu Beginn des Aufrufs wird die Ausführung durch den Aufruf von `select` angehalten. An dieser Stelle wartet der `Selector` auf den Verbindungsaufbau durch den `SocketWriter` der Prüflings-VM. Wenn eine Verbindungsanfrage vorliegt, wird die Ausführung durch den Aufruf der `accept`-Methode fortgesetzt. Der `ServerSocketChannel` erzeugt daraufhin eine neue `SocketChannel`-Instanz, welche die zustande gekommene Verbindung repräsentiert. Die `SocketChannel`-Instanz wird als Rückgabewert an den `SocketReader` übergeben und kann im Folgenden zum Empfang der Ereignisse verwendet werden.

Die Verarbeitung der über den `SocketChannel` empfangenen Daten ist im Sequenzdiagramm der Abbildung 7.7 dargestellt. Der `SocketReader` erzeugt zunächst einen neuen `Selektor` und registriert diesen durch den Aufruf `register(sel)` beim `SocketChannel`. Der `Selektor` kann im Folgenden dazu verwendet werden, auf das Eintreffen neuer Daten im SocketChannel zu warten. Um die eingetroffenen Daten aus dem `SocketChannel` auslesen zu können, wird darüber hinaus eine neue `ByteBuffer`-Instanz erzeugt.

Der Datenempfang des `SocketReader` wird durch die äußere Schleife gesteuert, die solange ausgeführt wird, bis die `SocketChannel`-Verbindung durch den `SocketWriter` in der Prüflings-VM geschlossen wird. Der `SocketWriter` schließt die Verbindung, sobald die Ausführung des Prüflings beendet ist und alle Ereignisse an den `SocketReader` übertragen

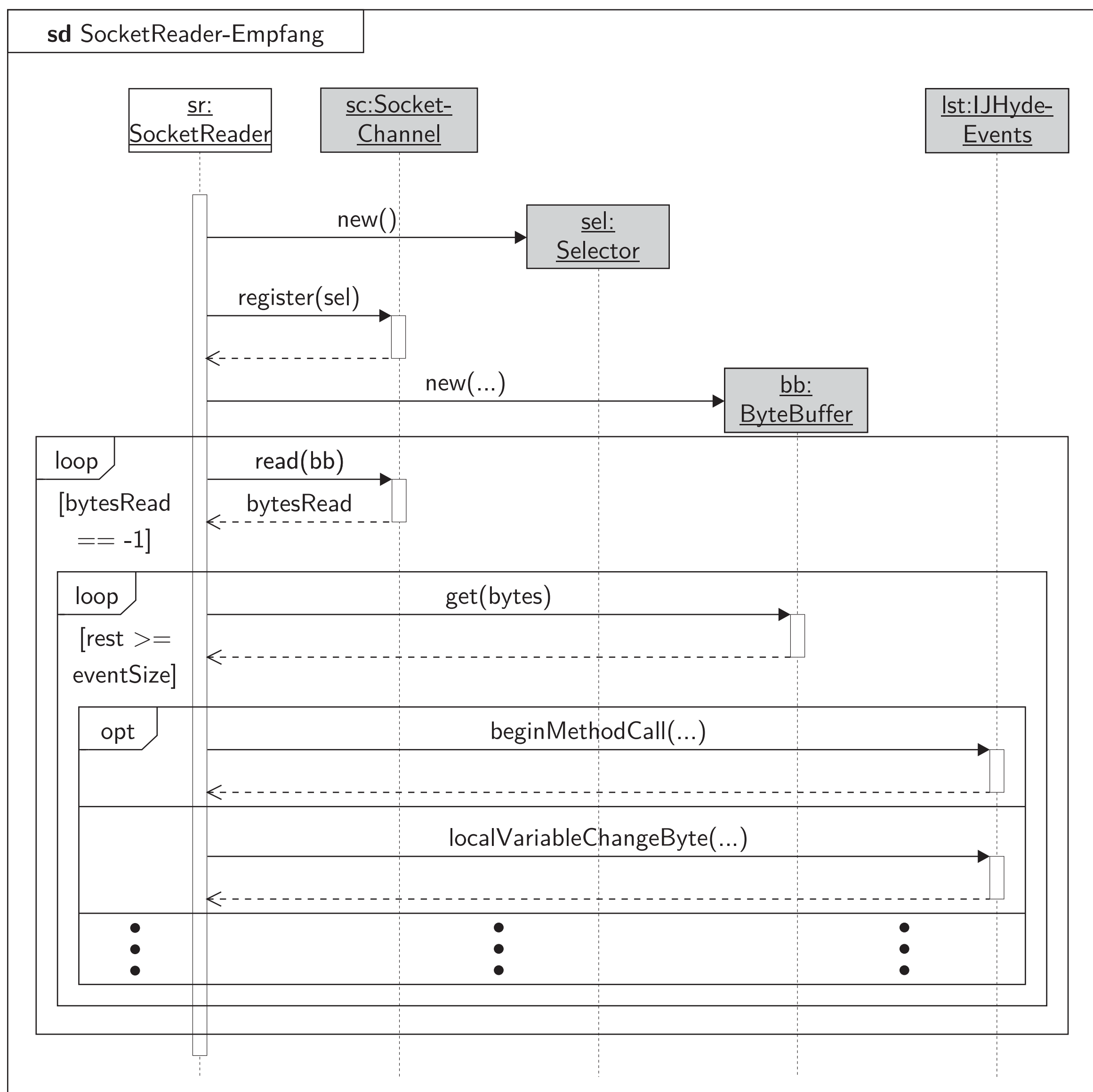


Abbildung 7.7: Sequenzdiagramm des Ereignisempfangs durch den `SocketReader`.

wurden. Zu Beginn des Durchlaufs der äußeren Schleife wird durch den Aufruf von `select` gewartet, bis Daten aus dem `SocketChannel` ausgelesen werden können. Anschließend wird durch den Aufruf von `read(bb)` eine Menge von Daten vom `SocketChannel` in den `ByteBuffer` übertragen. Der Rückgabewert entspricht der Anzahl der gelesenen Bytes und wird in der Variablen `bytesRead` gespeichert. Der zurückgegebene Wert entspricht -1, wenn die `SocketChannel`-Verbindung getrennt wurde und alle empfangenen Daten durch den `read` Aufruf ausgelesen wurden.

Listing 7.1: Die Konfigurationsdatei `TransmissionConfig.properties`.

```
1 de.wwu.jhyde.transmission.log4j.properties =
     log4j.properties
2 de.wwu.jhyde.transmission.sender =
     de.wwu.jhyde.transmission.send.SocketWriter
3 de.wwu.jhyde.transmission.receiver =
     de.wwu.jhyde.transmission.receive.SocketReader
```

Nach der Übertragung der Daten in den `ByteBuffer` wird die innere Schleife solange ausgeführt, wie der `ByteBuffer` noch genügend Daten enthält, um mindestens ein Ereignis auslesen zu können. Zu Beginn der Ausführung des Schleifenrumpfes wird die Bytefolge eines Ereignisses durch den Aufruf `get(bytes)` in das Array `bytes` übertragen. Anhand der Bytefolge erkennt der `SocketReader`, um welche Art von Ereignis es sich handelt. Dies geschieht im Prinzip durch eine große `switch`-Anweisung, deren Ausführung im Sequenzdiagramm angedeutet ist. Anschließend rekonstruiert er die Argumente des Ereignisses aus der Bytefolge und leitet das Ergebnis an die während der Initialisierung registrierte `IJHydeEvents`-Instanz weiter. Hierfür wird die entsprechende Methode der `IJHydeEvents`-Schnittstelle aufgerufen.

Die Ausführung des `SocketReader`-Threads wird beendet, wenn der `SocketChannel` geschlossen ist, alle Daten aus dem `SocketChannel` in den `ByteBuffer` übertragen wurden, der `ByteBuffer` vollständig entleert wurde und alle Ereignisse in Form von Methodenaufrufen an die `IJHydeEvents`-Instanz weitergeleitet sind.

7.2.5 Konfiguration

Der Transmitter wird über die Konfigurationsdatei `TransmissionConfig.properties` (vgl. Listing 7.1) konfiguriert. Diese Datei befindet sich im Paket (Package) `de.wwu.jhyde.transmission.config`. In der Datei können die Werte von drei so genannten Properties gesetzt werden.

Der Wert des ersten Property ist eine relative Pfadangabe für eine weitere Properties-Datei, in der der Logging-Mechanismus für den Transmitter

konfiguriert werden kann. Der Transmitter benutzt das Logging-Framework log4J [10, 61]. Mithilfe der Properties-Datei können für Pakete und Klassen des Transmitters individuelle Logger-Klassen und Log-Level definiert werden.

Der Wert des zweiten Property enthält den vollqualifizierten Klassennamen der Klasse, die in der Prüflings-VM zum Versenden der Ereignisse an die Debugger-VM verwendet wird. Wie zuvor beschrieben, muss die Klasse das Interface `IJHydeEventSender` implementieren.

Über den Wert des dritten Property wird die Klasse festgelegt, die in der Debugger-VM zum Empfang der Ereignisse verwendet wird, die durch die Sendeeinheit der Prüflings-VM verschickt werden. Der Wert muss ebenfalls dem vollqualifizierten Namen einer Klasse entsprechen und die Klasse muss darüber hinaus das Interface `IJHydeEventReceiver` implementieren.

7.3 Instrumentierer

Im Folgenden wird der Instrumentierer von JHyde vorgestellt. Zunächst werden unterschiedliche Möglichkeiten untersucht, Java-Programme zu instrumentieren. Nach der Auswahl eines geeigneten Instrumentierungs-Frameworks wird analysiert, welche Klassen in der Prüflings-VM instrumentiert werden müssen. Im Anschluss wird das grundsätzliche Schema für die Instrumentierung von Java-Klassen entwickelt. Aufbauend auf diesen Erkenntnissen wird die Architektur des Instrumentierers beschrieben und anschließend der Ablauf des Instrumentierungsprozesses dargestellt. Zuletzt werden einige Konfigurationsmöglichkeiten für den Instrumentierer beschrieben.

7.3.1 Instrumentierung von Java-Programmen

Die Aufgabe des Instrumentierers ist es, die Klassen in der Prüflings-VM so zu manipulieren, dass diese während der Ausführung des Prüflings Ereignisse über den Programmablauf an den Transmitter senden. Vor der Auswahl der Methoden und Werkzeuge zur Instrumentierung der Klassen sind zunächst

zwei grundsätzliche Entwurfsentscheidungen über die Art der Instrumentierung zu treffen:

Statische oder dynamische Instrumentierung - Bei der statischen Instrumentierung werden die Klassen vor dem Start der Java-VM instrumentiert und anschließend zur Ausführung in instrumentierter Form in die Prüflings-VM geladen. Bei der dynamischen Instrumentierung wird eine Klasse erst dann instrumentiert, wenn sie benötigt wird, d. h., wenn sie in die Java-VM geladen wird. Grundsätzlich ist die dynamische Instrumentierung zu bevorzugen, da sie deutlich flexibler ist und ausschließlich die benötigten Klassen instrumentiert. Darüber hinaus kann die Menge der Klassen, die für die Ausführung des Prüflings benötigt werden, im Vorfeld der Ausführung gar nicht exakt bestimmt werden. Es ist daher problematisch, ausschließlich statische Instrumentierung zu verwenden. Wie sich im Abschnitt 7.3.2 zeigen wird, kann auf die statische Instrumentierung nicht vollständig verzichtet werden. Somit ist das Ziel, bei der Instrumentierung einen möglichst großen Teil der Klassen dynamisch zu instrumentieren.

Quelltext- oder Bytecode-Instrumentierung - Zur Manipulation einer Klasse kann entweder ihr Quelltext oder ihr Bytecode instrumentiert werden. In beiden Fällen muss die Klasse erst durch einen Parser eingelesen werden. Das Parsen eines Quelltextes ist jedoch sehr viel aufwendiger als das Parsen des kompakten und streng strukturierten Bytecodes. Nach der Instrumentierung eines Quelltextes muss dieser anschließend noch in Bytecode kompiliert werden. Bei der Bytecode-Instrumentierung wird der Quelltext bereits vor der Instrumentierung übersetzt. Nach der Instrumentierung ist der erzeugte Bytecode direkt ausführbar und muss nicht mehr übersetzt werden. Für die dynamische Instrumentierung ist die Bytecode-Instrumentierung zu empfehlen, da die Klassen bereits vor und nicht während der Laufzeit der Prüflings-VM kompiliert werden. Diese Methode verursacht daher zur Laufzeit einen wesentlich geringeren Aufwand.

7.3.1.1 Java-Instrumentation-API

Die Instrumentation-API [140] gehört zum Package `java.lang.instrument` und ist Teil der Java-API. Die Instrumentation-API ermöglicht

es, sogenannten Agenten zur Laufzeit der Java-VM den Bytecode von neu geladenen Klassen zu instrumentieren. Ein Agent besteht aus einer JAR-Datei, in deren Manifest-Datei das Attribut `PremainClass` definiert ist. Der Wert dieses Attributs entspricht dem vollqualifizierten Namen einer Klasse, die die in der Abbildung 7.2 dargestellte `premain`-Methode implementiert. Beim Start der Java-VM wird der vollständige Pfad des Agenten über den Kommandozeilenparameter `-javaagent:<jarpath>` gesetzt. Nach der Initialisierung der Java-VM wird der Agent geladen und die in der Manifest-Datei spezifizierte `premain`-Methode aufgerufen. Die `premain`-Methode erzeugt einen neuen `BytecodeTransformer` und registriert diesen in der Instrumentation-API.

Listing 7.2: Die `premain`-Methode zur Installation des Instrumentierers.

```
1 public static void premain(String agentArgs,
     Instrumentation inst) {
2   inst.addTransformer(new BytecodeTransformer());
3 }
```

Der `BytecodeTransformer` implementiert das Interface `ClassFileTransformer` (vgl. Listing 7.3). Nach der Registrierung wird für jede Klasse, die durch die Java-VM geladen wird, die Methode `transform` der `BytecodeTransformer`-Instanz aufgerufen. Die wichtigsten Argumente des Methodenaufrufs sind der Klassenname (`className`) und der Bytecode der geladenen Klasse (`classfileBuffer`). Der Bytecode besitzt das Format einer Java-Klassendatei, welches in der JVM-Spezifikation definiert ist und im Abschnitt 2.3.6 beschriebenen wurde. Der `BytecodeTransformer` manipuliert den übergebenen Bytecode und gibt das Resultat der Instrumentierung zurück.

7.3.1.2 Frameworks

Die Instrumentierung von Java-Bytecode ist eine komplexe Aufgabe, die mit einigen Problemen behaftet ist. Das grundlegende Problem ist, dass eine

Listing 7.3: Das Interface `ClassFileTransformer`.

```
1 public interface ClassFileTransformer {
2   byte[]  transform(ClassLoader loader, String
       className, Class<?> classBeingRedefined,
       ProtectionDomain protectionDomain, byte[]
       classfileBuffer)
3 }
```

Klassendatei ein Array von Bytes ist. Dies macht es nahezu unmöglich, eine Klassendatei direkt zu manipulieren. Aus diesem Grund ist es notwendig, vor der Manipulation die Baumstruktur einer Klassendatei zu analysieren.

Ein weiteres Problem sind die Referenzen auf den Konstantenpool. Die Einträge des Konstantenpools werden von verschiedenen Elementen der Klassendatei über ihren Index im Konstantenpool referenziert. Während der Manipulation einer Klassendatei können Einträge in den Konstantenpool eingefügt oder aus dem Konstantenpool entfernt werden. Die Folge ist, dass sich die Indizes der Einträge durch die Manipulation einer Klassendatei ändern können. Dies führt dazu, dass die Referenzen auf die betroffenen Einträge aktualisiert werden müssen.

Darüber hinaus enthält eine Klassendatei auch Referenzen auf Instruktionen. Zum Beispiel definiert ein Sprungbefehl wie `goto` oder `ifeq` über einen Index ein bestimmtes Sprungziel in der Instruktionsliste einer Methode. Ebenso enthält eine `try-catch`-Anweisung Referenzen auf Indizes der Instruktionsliste, mit denen der Gültigkeitsbereich des `try`-Blocks und der Start des `catch`-Blocks festgelegt werden. Durch Hinzufügen und Entfernen von Instruktionen kann sich der Index einer Instruktion ändern. Wenn diese Anweisung referenziert wird, dann müssen alle Referenzen entsprechend aktualisiert werden.

Ein weiteres Problem ist die Berechnung der maximalen Größe des Operanden-Stacks einer Methode. Diese kann sich ändern und muss nach der Instrumentierung durch eine Datenflussanalyse neu berechnet werden.

Es gibt eine Vielzahl von Frameworks, mit denen sich Java-Bytecode manipulieren lässt. Diese Frameworks verfolgen unterschiedliche Methoden und

Ansätze, um die oben beschriebenen Probleme zu lösen. Zu den wichtigsten Frameworks gehören:

BCEL

BCEL [8, 39] ist eines der am häufigsten verwendeten Frameworks zur Instrumentierung von Bytecode. Das Framework führt die Manipulation einer Klassendatei in drei Phasen aus. In der ersten Phase wird der Bytecode der Klasse deserialisiert und in einen Objektgraphen transformiert. In der zweiten Phase wird der Objektgraph manipuliert, um die gewünschten Änderungen in der Klasse herbeizuführen. Abschließend folgt in der dritten Phase die Serialisierung des Objektgraphen.

Dieser Ansatz löst die oben beschriebenen Probleme, indem die Deserialisierung und die Serialisierung vollständig durch das Framework übernommen werden. Diese komplexen Vorgänge sind daher für den Benutzer vollkommen transparent. Darüber hinaus wird das Problem der Aktualisierung der Referenzen auf Elemente des Konstantenpools oder Elemente von Instruktionslisten durch den Objektgraphen elegant gelöst. Die Index-Referenzen, die zwischen den Elementen der Klassendatei existieren, werden durch die Deserialisierung in Objektreferenzen überführt. Im Objektgraph referenziert dann ein Sprungbefehl nicht mehr einen Index einer Instruktion, sondern die Instruktion selbst. Wenn sich die Position einer referenzierten Instruktion in der Instruktionsliste ändert, dann müssen die Referenzen nicht gesondert aktualisiert werden. Bei der Serialisierung werden die Objektreferenzen wieder in Index-Referenzen übersetzt.

Der Ansatz, die Instrumentierung auf Basis eines Objektgraphen durchzuführen, ist elegant, jedoch auch mit einigem Aufwand verbunden. Das BCEL-Framework erzeugt für jedes Element einer Java-Klasse, d. h. sogar für jede einzelne Bytecode-Instruktion, ein Objekt. Die Erzeugung dieser Instanzen kostet Zeit und Ressourcen.

SERP

SERP [163] verfolgt im Wesentlichen dieselbe Strategie wie BCEL. Das SERP-Framework besitzt jedoch eine geringere Anzahl an Klassen, da im

Unterschied zu BCEL nicht jede der ungefähr 200 Bytecode-Instruktionen durch eine separate Klasse repräsentiert wird. In SERP werden gleichartige Instruktionen durch eine gemeinsame Klasse repräsentiert.

ASM

ASM [22, 120] verfolgt einen grundlegend anderen Ansatz als BCEL und SERP. Das ASM-Framework verzichtet auf die Repräsentation der Klassendatei durch einen Objektgraphen. Anstelle des Graphen verwendet ASM das Visitor-Muster [55, S. 331–344]. Mit diesem Muster ist es möglich, die einzelnen Elemente der serialisierten Klassendatei zu besuchen, ohne diese vorher zu deserialisieren. Genauer gesagt werden ausschließlich primitive Datentypen und Zeichenketten aus der serialisierten Klasse extrahiert. Der genaue Ablauf der Instrumentierung mithilfe des ASM-Frameworks wird in den Abschnitten 7.3.4 und 7.3.5 beschrieben.

Durch den Verzicht auf die Transformation von Java-Klassen in einen Objektgraphen ist das ASM-Framework sehr viel kompakter als BCEL und SERP. Die JAR-Datei des ASM-Frameworks ist nur 21 kB groß und besteht aus insgesamt 13 Klassen. SERP hat einen Umfang von 150 kB und 80 Klassen und BCEL von 350 kB bzw. 270 Klassen [120]. Dieser große Unterschied spiegelt sich auch in der Ausführungsgeschwindigkeit wieder. ASM kann eine Klasse durchschnittlich 11-mal schneller als BCEL und 20-mal schneller als SERP instrumentieren [120]. Der enorme Geschwindigkeitsvorteil von ASM resultiert im Wesentlichen aus den geringen Kosten für die Deserialisierung und die Serialisierung. Im Vergleich zum Laden einer Klasse ohne Instrumentierung produziert ASM einen Mehraufwand von 60 % für Deserialisierung und Serialisierung. BCEL und SERP produzieren hingegen einen Mehraufwand von 700 % bzw. 1100% [120].

Weitere Frameworks

Es gibt eine Vielzahl weiterer Frameworks zur Manipulation von Java-Bytecode. Einige dieser Frameworks wie AspectJ [76] und Javassist [32, 33] finden Verwendung in der aspektorientierten Programmierung (AOP) [19, 50]. Im Vergleich zu den zuvor vorgestellten Frameworks sind die Möglichkeiten der Bytecode-Manipulation bei diesen Frameworks eingeschränkt. Im

Wesentlichen werden bei der AOP an bestimmten Stellen im Bytecode einer Methode, den sogenannten Join-Points, zuvor definierte Aspekte ausgeführt. Diese Funktionalität ist jedoch nicht ausreichend, um die für JHyde benötigten Informationen zu sammeln. Dies liegt hauptsächlich daran, dass die Positionen, an denen Join-Points eingefügt werden können, beschränkt sind. Zum Beispiel gibt es keine Join-Points für den Zugriff auf lokale Variablen. Diese wären allerdings notwendig, um den Datenfluss für das Omniscient-Debugging vollständig zu protokollieren.

Das JMangler-Framework [77, 78] dient ebenfalls der AOP. Dieses Framework fokussiert aber unter anderem die Instrumentierung bereits geladener Klassen. Da zwischen den geladenen Klassen in einer Java-VM Abhängigkeiten bestehen, ist beim Austausch einer bereits geladenen Klasse die Reihenfolge, in der die abhängigen Klassen angepasst werden von Bedeutung. JMangler stellt sicher, dass in diesem Fall alle Klassen in der richtigen Reihenfolge instrumentiert werden. Eine bestimmte Menge von Klassen kann mit JMangler nicht instrumentiert werden. Zu dieser Menge gehören alle Klassen, die während der Initialisierungsphase der Java-VM geladen werden und später nicht mehr manipuliert werden können. Die meisten Klassen des Pakets `java.lang`, wie zum Beispiel die Klasse `Object` oder die Klasse `String`, gehören dieser Menge an. Aufgrund dieser Einschränkungen ist JMangler nicht für die Umsetzung der Instrumentierer-Komponente von JHyde geeignet.

Auswahl eines Frameworks

Grundsätzlich sind die Frameworks BCEL, SERP und ASM für die Umsetzung des Instrumentierers von JHyde geeignet. Diese Frameworks unterstützen sowohl statische als auch dynamische Instrumentierung und unterliegen keinerlei Einschränkungen bei der Manipulation von Klassendateien. Im Hinblick auf die Geschwindigkeit, mit der Klassendateien instrumentiert werden können, ist das ASM-Framework den anderen Frameworks deutlich überlegen. Aus diesen Gründen wurde ASM für die Umsetzung des Instrumentierers verwendet.

7.3.2 Klassen in der Prüflings-VM

Während der Ausführung des Prüflings werden drei unterschiedliche Gruppen von Klassen in die Prüflings-VM geladen: die Bootstrapping-Klassen (B), die Klassen, die vom Instrumentierer oder dem Transmitter benutzt werden (D) und die Klassen die vom Prüfling benutzt werden (P). Diese Klassen sind in der Abbildung 7.8 dargestellt. Zu den Bootstrapping-Klassen gehören alle Klassen, die während der Initialisierungsphase der Java-VM geladen werden. Bei den Klassen, die für die Ausführung von Instrumentierer, Transmitter und Prüfling benutzt werden, handelt es sich sowohl um benutzerdefinierte Klassen (U) als auch um Klassen der Java-API (J).

Grundsätzlich soll eine Klasse, wenn sie während der Ausführung des Prüflings benutzt wird, Debugging-Ereignisse erzeugen und an den Transmitter übermitteln. Wird eine Klasse während der Ausführung des Instrumentierers oder des Transmitters benutzt, dann ist dieses Verhalten unerwünscht. Die Klasse soll in diesem Fall keine Debugging-Ereignisse erzeugen, da die Ausführung des Debuggers transparent sein muss und nicht Gegenstand der Ausführung des Prüflings ist, dessen Verhalten untersucht werden soll.

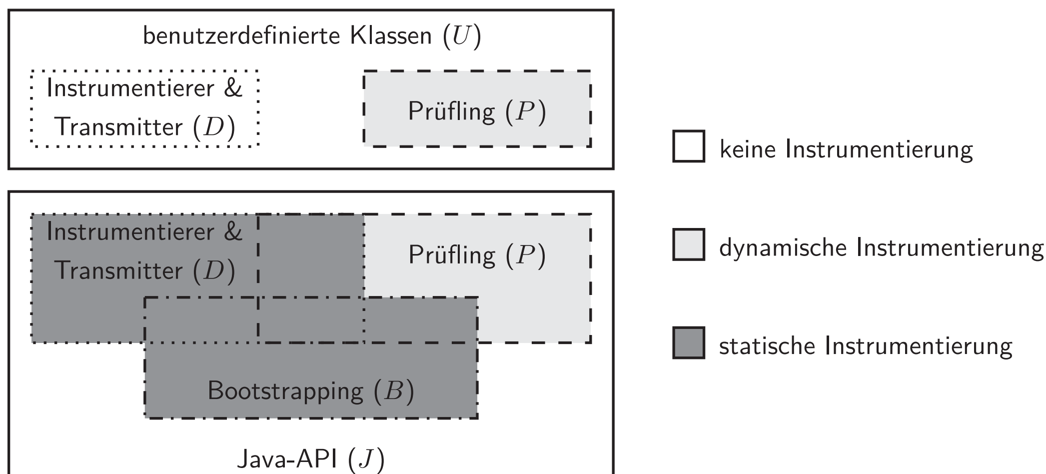


Abbildung 7.8: Klassen in der Java-VM des untersuchten Programms.

Innerhalb der Menge der benutzerdefinierten Klassen U sind die Klassen $D \cap U$ und $P \cap U$ in jedem Fall überschneidungsfrei ($D \cap P \cap U = \emptyset$),

da sie in unterschiedlichen Paketen definiert sind. Die Klassen der Menge $D \cap U$ dürfen somit nicht instrumentiert werden, wohingegen die Klassen der Menge $P \cap U$ instrumentiert werden müssen.

Innerhalb der Menge der Klassen der Java-API J sind die Klassen $D \cap J$ und $P \cap J$ nicht überschneidungsfrei ($D \cap P \cap J \neq \emptyset$). Da in Java alle Klassen von der Klasse `java.lang.Object` erben, enthält die Schnittmenge $D \cap P \cap J$ mindestens die Klasse `Object`. Welche Elemente tatsächlich Teil der Schnittmenge sind, hängt davon ab, welche Klassen von $D \cap J$ während der Ausführung des Prüflings benutzt werden. Da die Menge $P \cap J$ durch das Laufzeitverhalten des Prüflings festgelegt wird, kann die Schnittmenge nicht statisch bestimmt werden. Daraus folgt, dass alle Klassen der Menge $D \cap J$ sowohl durch den Instrumentierer oder den Transmitter als auch durch den Prüfling verwendet werden können. Die Methoden dieser Klassen müssen daher in zwei unterschiedlichen Modi ausführbar sein:

- Wenn eine Methode durch den Prüfling aufgerufen wird, dann muss sie im Debugging-Modus ausgeführt werden. Im Debugging-Modus verursacht die Ausführung der Methode Debugging-Ereignisse, die an den Transmitter übermittelt werden.

- Wenn eine Methode durch den Instrumentierer oder den Transmitter ausgeführt wird, dann muss sie im normalen Modus ausgeführt werden. In diesem Modus verursacht die Ausführung der Methode keine Debugging-Ereignisse.

Grundsätzlich sollen die Klassen in der Prüflings-VM dynamisch, d. h. während des Ladevorgangs, instrumentiert werden. Für die Menge der Klassen $(D \cap J) \cup B$ ist dies jedoch nicht möglich. Die Bootstrapping-Klassen und die Klassen der Java-API, die durch den Instrumentierer oder den Transmitter verwendet werden, sind bereits in die Prüflings-VM geladen, bevor die dynamische Instrumentierung aktiv ist. Daher müssen diese Klassen vor dem Start der Prüflings-VM statisch instrumentiert werden. Es müssen alle Bootstrapping-Klassen instrumentiert werden, da sich vor der Ausführung des Prüflings nicht bestimmen lässt, welche Bootstrapping-Klassen in der Menge $B \cap P$ liegen.

Zusammenfassend lassen sich die folgenden vier Instrumentierungsarten identifizieren:

Keine Instrumentierung - Die benutzerdefinierten Klassen des Instrumentierers und des Transmitters ($D \cap U$) werden nicht instrumentiert. Ebenso müssen natürlich auch diejenigen Klassen nicht instrumentiert werden, die während der Ausführung nicht in die Prüflings-VM geladen werden.

Dynamische Instrumentierung (Debugging-Modus) - Die Menge $(P \setminus B) \setminus D$ wird dynamisch instrumentiert. Dies sind alle Klassen des Prüflings, die nicht zu den Bootstrapping-Klassen gehören und nicht vom Instrumentierer oder Transmitter benutzt werden. Die dynamisch instrumentierten Klassen werden ausschließlich im Debugging-Modus ausgeführt.

Statische Instrumentierung (Debugging-Modus) - Die Menge $B \setminus D$ muss statisch instrumentiert werden. Diese Menge wird ausschließlich im Debugging-Modus ausgeführt.

Statische Instrumentierung (Debugging- / normaler Modus) - Die übrigen Klassen, die zur Menge $D \cap J$ gehören, müssen statisch instrumentiert werden. Darüber hinaus müssen die Methoden der Klassen dieser Menge sowohl im Debugging-Modus als auch im normalen Modus ausgeführt werden können.

7.3.3 Instrumentierungsschema

Im Folgenden wird das Instrumentierungsschema für die Klassen in der Prüflings-VM vorgestellt. Um den Instrumentierungsprozess zu vereinfachen, werden alle Klassen nach demselben Schema instrumentiert. Dabei ist es unerheblich, ob die Klassen statisch oder dynamisch instrumentiert werden und ob sie später ausschließlich im Debugging-Modus oder sowohl im Debugging-Modus als auch im normalen Modus ausgeführt werden. Das einheitliche Instrumentierungsschema vereinfacht den Instrumentierungsprozess, da alle Klassen auf dieselbe Art und Weise instrumentiert werden.

Debugging-Modus

Der Mechanismus, der es ermöglicht, die Methoden wahlweise im normalen Modus oder im Debugging-Modus auszuführen, ist in der Abbildung 7.9 dargestellt. Der Debugging-Modus kann mithilfe des Attributs `debuggingEnabled` der Klasse Thread aktiviert werden (vgl. Abbildung 7.9a). Dieses Attribut wird der Klasse Thread durch statische Instrumentierung hinzugefügt. Wenn das Attribut `true` ist, dann werden durch den entsprechenden Thread alle Methoden im Debugging-Modus ausgeführt. Wenn das Attribut `false` ist, dann werden alle durch den entsprechenden Thread aufgerufenen Methoden im normalen Modus ausgeführt. Durch diesen Mechanismus kann sichergestellt werden, dass eine Methode im Debugging-Modus ausgeführt wird, wenn sie vom Prüfling aufgerufen wird, und dass sie im normalen Modus ausgeführt wird, wenn Instrumentierer oder Transmitter sie aufrufen. Solange der Prüfling ausgeführt wird, ist das Attribut `true`, so dass alle Methoden im Debugging-Modus ausgeführt werden. Wenn der Prüfling ein Debugging-Ereignis sendet, dann ruft er eine der Methoden der Ereignisschnittstelle des Transmitters (siehe Abschnitt 7.2.1) auf. Zu Beginn der Ausführung der Ereignismethode des Transmitters wird das `debuggingEnabled`-Attribut des ausführenden Threads auf `false` gesetzt. Dadurch werden alle durch den Transmitter zum Senden des Ereignisses aufgerufenen Methoden im normalen Modus ausgeführt. Bevor die Ereignismethode beendet wird, um die Kontrolle des Programmablaufs zurück an den Prüfling zu übergeben, wird das `debuggingEnabled`-Attribut wieder auf `true` gesetzt. In gleicher Weise wird während des dynamischen Instrumentierungsprozesses das `debuggingEnabled`-Attribut verwendet, um das Senden von Debugging-Ereignissen für die Dauer der Instrumentierung zu unterbinden.

Die Steuerung des Ausführungsmodus über ein zusätzliches Attribut in der Klasse `Thread` ist problematisch, da sie die Reihenfolge, in der die Klassen in die Java-VM geladen werden, verändert. Der Ladevorgang einer Klasse durch die Java-VM wird beim ersten Zugriff auf die Klasse ausgelöst. Durch das beschriebene Instrumentierungsschema würde jede Methode zu Beginn ihrer Ausführung auf die Klasse Thread zugreifen, um zu bestimmen, ob sie im Debugging-Modus oder im normalen Modus ausgeführt werden soll. Dies würde dazu führen, dass die Klasse `Thread` zu einem sehr frühen Zeitpunkt der Initialisierung der Java-VM geladen wird. Da die Reihenfolge, in der die

```
1 public class Thread {
2   public boolean debuggingEnabled;
3   ...
4 }
```

(a) Die Klasse Thread.

```
1 public class BootstrapLock {
2   public static boolean enabled = true;
3 }
```

(b) Die Klasse BootstrapLock.

```
1 public class Foo {
2   private int value;
3   private Foo next;
4   ...
5
6   public int bar(Foo foo) {
7     if (BootstrapLock.enabled ||
         Thread.currentThread().debuggingEnabled ==
         false) {
8       \\ ursprünglicher Bytecode
9       ...
10    } else {
11      \\ instrumentierter Bytecode
12      ...
13    }
14  }
15 }
```

(c) Die Klasse Foo.

Abbildung 7.9: Instrumentierungsschema für Java-Klassen.

Klassen während der Initialisierung geladen werden, kritisch ist, würde die Verschiebung des Ladezeitpunktes der Klasse `Thread` dazu führen, dass die Java-VM während der Initialisierung abstürzt.

Um den Ladezeitpunkt der Klasse `Thread` nicht zu verändern, muss verhindert werden, dass während der Initialisierungsphase auf das Attribut `debuggingEnabled` zugegriffen wird. Dies wird zusammen mit der Klasse `BootstrapLock` (vgl. Abbildung 7.9b) sichergestellt. Die Klasse

BootstrapLock verfügt über das statische Attribut enabled, welches während der Initialisierungsphase true ist. Nach der Initialisierung der Java-VM wird das Attribut auf false gesetzt. In der Abbildung 7.9c ist für die Methode bar der Klasse Foo dargestellt, wie die Kombination der Attribute der Klassen BootstrapLock und Thread dazu verwendet wird, den Ausführungsmodus für eine Methode zu bestimmen. Jede Methode besteht wie die Methode bar aus einer if-Anweisung, die entweder den ursprünglichen Bytecode der Methode ausführt, wenn die Bedingung zu true auswertet, oder den instrumentierten Bytecode ausführt, wenn die Bedingung zu false auswertet. Die Bedingung der if-Anweisung wertet zu true aus, wenn das BootstrapLock aktiviert ist oder der Debugging-Modus für den ausführenden Thread deaktiviert ist. Die beiden Teilbedingungen sind über einen logischen Oder-Operator (||) miteinander verknüpft. Durch die Kurzschlussauswertung (Short Circuit Evaluation) dieses Operators wird der zweite Teil der Bedingung nicht ausgewertet, solange das BootstrapLock aktiviert ist. Nach der Initialisierung wird das BootstrapLock deaktiviert. Der Ausführungsmodus wird dann allein durch das debuggingEnabled-Attribut des ausführenden Threads bestimmt.

Instrumentierung einer Methode

Das Ergebnis der vollständigen Instrumentierung der Methode bar ist in Listing 7.4 dargestellt. Die Instrumentierungen sind in diesem Fall im Quelltext der Methode vorgenommen worden. Tatsächlich wird der Bytecode und nicht der Quelltext einer Methode instrumentiert. Bytecode ist jedoch aufgrund des niedrigen Abstraktionsgrades nicht für die strukturierte Darstellung eines Java-Programms geeignet. Aus diesem Grund wird das Ergebnis der Instrumentierung auf Quelltextebene, d. h. mithilfe der Programmiersprache Java, dargestellt.

Die Zeilen 6–10 zeigen den ursprünglichen Quelltext der Methode, der ausgeführt wird, wenn der Debugging-Modus nicht aktiviert ist. Die Zeilen 13–64 des Quelltextes werden ausgeführt, wenn die Methode im Debugging-Modus ausgeführt wird. In Zeile 13 wird die lokale Variable o[13] initialisiert. Diese

[13] Die Verwendung von $-Zeichen in Variablen-, Attribut- oder Methodennamen ist im Quelltext eines Java-Programms nicht erlaubt. Im Bytecode ist die Ver-

Variable speichert zu jedem Zeitpunkt die Position der zuletzt ausgeführten Instruktion. Anschließend wird in Zeile 14 mithilfe der `ObjectIdMap` die ID des ausführenden Threads ermittelt. Die `ObjectIdMap` ist Teil des Instrumentierers und speichert für jedes Objekt der Prüflings-VM eine eindeutige ID in einer Hashtabelle. Eine separate Hashtabelle ist notwendig, da der Speicherplatz einer Instanz der Klasse `Object` durch die Java-VM festgelegt ist und nicht verändert werden darf. Die Klasse `Object` kann daher nicht durch Instrumentierung um ein weiteres Attribut ergänzt werden, welches die Objekt-ID einer Instanz speichert.

Die Ereignisse, die gesendet werden, falls die Methode im Debugging-Modus ausgeführt wird, sind in Form von Aufrufen der statischen Methoden der Klasse `DebuggingEvent` in den Programmcode eingefügt. Die Verwendung von statischen Methodenaufrufen ist in diesem Fall zweckmäßig, da sie den Instrumentierungsprozess erleichtert. Die Klasse `DebuggingEvent` leitet die empfangenen Ereignisse an die Sendeeinheit des Transmitters weiter (vgl. Abschnitt 7.2.3).

Der restliche instrumentierte Quelltext ist von einem `try`-Block (Zeilen 15–55) umgeben, um auf Exceptions in Form eines entsprechenden Debugging-Ereignisses reagieren zu können. Alle Exceptions, die während der Ausführung des `try`-Blocks auftreten, werden durch den `catch`-Block in den Zeilen 55–61 behandelt. Innerhalb des `catch`-Blocks wird zunächst ein Ereignis für den durch die Exception verursachten Kontrollfluss erzeugt. Hierfür wird die Methode `edgeException` der Klasse `DebuggingEvent` mit der ID des ausführenden Threads (`tId`), der Position der Instruktion, die die Exception verursacht hat (`$o$`), und der Position, an der die Exception gefangen wurde (57), aufgerufen.

Das nächste Ereignis dient zur Übermittlung der gefangenen Exception `t`. Hierzu wird eine Wertänderung der lokalen Variablen mit dem Index -2 übermittelt. Diese Position verwendet die Ereignisschnittstelle als Speicherplatz, wenn eine Methode nicht regulär beendet wird und eine Exception wirft. Zuletzt wird in Zeile 59 das Ende des Methodenaufrufs signalisiert und anschließend die zuvor gefangene Exception geworfen. Wie gezeigt, dient die lokale Variable `$o$` dazu, den Kontrollfluss im Falle einer geworfenen

wendung allerdings möglich. Die Benutzung des $-Zeichens in den Namen von Elementen, die durch die Instrumentierung einer Klasse hinzugefügt werden, verhindert, dass es zu Namenskonflikten kommt.

Exception zu rekonstruieren. Da die Variable zu jedem Zeitpunkt die Position der zuletzt ausgeführten Instruktion enthält, speichert sie zu Beginn eines `catch`-Blocks immer die Position der Instruktion, die die Exception ausgelöst hat. Auf die gleiche Weise funktioniert dieser Mechanismus auch für `catch`-Blöcke, die bereits im Quelltext einer Methode vorhanden sind. Hierbei entfallen die Zeilen 58–59 und es wird nur das `edgeException`-Ereignis gesendet, um den Kontrollfluss aufzuzeichnen.

Listing 7.4: Instrumentierte Methode `bar`.

```
 1 public int bar(int factor) {
 2   //  Prüfe ob ursprünglicher oder instrumentierter
        Bytecode auszuführen ist
 3   if (BootstrapLock.enabled
 4       || Thread.currentThread().debuggingEnabled ==
           false) {
 5     \\ ursprünglicher Bytecode
 6     int result = this.value * factor;
 7     if (this.next != null) {
 8       result += next.bar(factor);
 9     }
10     return result;
11   } else {
12     \\ instrumentierter Bytecode
13     int $o$ = 0;
14     long tId =
         ObjectIdMap.getObjectId(Thread.currentThread());
15     try {
16       \\ Start des Methodenaufrufs
17       DebuggingEvent.beginMethodCall(tId, "Foo#bar()I");
18
19       \\ Initialisiere lokale Variablen
20       DebuggingEvent.localVariableChange(this, tId, $o$
           = 20, 0);
21       DebuggingEvent.localVariableChange(factor, tId,
           $o$ = 21, 1);
22
23       \\ Beginn der Methodenausführung
24       DebuggingEvent.endMethodCallInit(tId);
25
26       \\ Zeile 6: int result = this.value * factor;
27       DebuggingEvent.localVariableUse(tId, $o$ = 27, 0);
```

```
28        DebuggingEvent.attributeUse(this, tId, $o$ = 28,
             "Foo#value");
29        DebuggingEvent.localVariableUse(tId, $o$ = 29, 1);
30        int result = this.value * factor;
31        DebuggingEvent.localVariableChange(result, tId,
             $o$ = 31, 2);
32
33        \\ Zeile 7: if (this.next != null) {
34        DebuggingEvent.localVariableUse(tId, $o$ = 34, 0);
35        DebuggingEvent.attributeUse(this, tId, $o$ = 35,
             "Foo#next");
36        if (this.next != null) {
37
38          \\ Zeile 8: result += this.next.bar(factor);
39          DebuggingEvent.conditionCovered(tId, $o$ = 39);
40          DebuggingEvent.localVariableUse(tId, $o$ = 40,
               2);
41          DebuggingEvent.localVariableUse(tId, $o$ = 41,
               0);
42          DebuggingEvent.attributeUse(this, tId, $o$ =
               42, "Foo#next");
43          DebuggingEvent.localVariableUse(tId, $o$ = 43,
               1);
44          DebuggingEvent.invokeMethod(tId, $o$ = 44);
45          result += next.bar(factor);
46          DebuggingEvent.localVariableChange(result, tId,
               $o$ = 46, 2);
47        } else {
48          DebuggingEvent.conditionCovered(tId, $o$ = 48);
49        }
50        \\ Zeile 10 return result;
51        DebuggingEvent.localVariableUse(tId, $o$ = 51, 2);
52        DebuggingEvent.localVariableChange(result, tId,
             $o$ = 52, -1);
53        DebuggingEvent.methodExit(tId, $o$ = 53);
54        return result;
55      } catch (Throwable t) {
56        // Fehlerbehandlung
57        DebuggingEvent.edgeException(tId, $o$, 57);
58        DebuggingEvent.localVariableChange(t, tId, $o$,
             -2);
59        DebuggingEvent.methodExit(tId, $o$);
60        throw t;
61      }
62    }}
```

Innerhalb des `try`-Blocks wird in Zeile 17 zunächst ein Ereignis über den Beginn des Aufrufs der Methode `bar` erzeugt. Dann werden in den Zeilen 20–21 die Werte der Argumente des Methodenaufrufs übermittelt. Das Ende der Initialisierung der Methodenargumente wird in der Zeile 24 gesendet.

In der Zeile 30 werden die Variablen `this`, `this.value` und `factor` benutzt bzw. ausgelesen. In den Zeilen 27–29 werden daher Ereignisse gesendet, die die Benutzung dieser Variablen signalisieren. Nach der Ausführung der ursprünglichen Anweisung in Zeile 30 wird die Änderung der lokalen Variablen `result` durch das Ereignis in Zeile 31 übermittelt.

Durch die `if`-Anweisung der Zeile 36 werden die lokale Variable `this` und das Attribut `this.next` benutzt. Für diese Anweisung müssen somit die Zeilen 34–35 eingefügt werden.

Die Zeile 39 ist Teil eines neuen Basisblocks im Kontrollflussgraphen. Die Überdeckung der Bedingung, die das Betreten des Basisblocks verursacht hat, wird durch das Ereignis in Zeile 39 signalisiert. Durch die Zeilen 40–43 wird die Benutzung der Variablen `result`, `this`, `this.next` und `factor` mitgeteilt. Das nachfolgende `invokeMethod`-Ereignis in Zeile 44 informiert über die Position des nachfolgenden Aufrufs der Methode `bar` im Programmcode. Anhand dieses Ereignisses kann die Benutzeroberfläche von JHyde später die Position im Quelltext ermitteln, an der ein Methodenaufruf aufgetreten ist. Anschließend folgt die ursprüngliche Quelltextzeile, durch die die Variable `result` verändert wird. Die Änderung von `result` wird anschließend in der Zeile 46 übermittelt.

Die `if`-Anweisung ist in den Zeilen 47–49 um einen `else`-Zweig erweitert worden. Innerhalb des `else`-Zweiges wird ein weiteres Bedingungsüberdeckungsereignis gesendet. Die beiden Fälle, in denen die Bedingung der `if`-Anweisung zu `true` bzw. `false` ausgewertet wird, können anhand der Position unterschieden werden, an der das Bedingungsüberdeckungsereignis aufgetreten ist. Die tatsächliche Instrumentierung von Bedingungsüberdeckungen lässt sich auf Quelltextebene nicht exakt wiedergeben; sie wird daher im nachfolgenden Abschnitt auf Bytecodeebene dargestellt.

Die letzte zu instrumentierende Zeile (54) beendet den Methodenaufruf und gibt den Wert der lokalen Variablen `result` zurück. Hierfür wird zunächst in Zeile 51 die Benutzung von `result` signalisiert. Das Ergebnis eines Methodenaufrufs wird in der lokalen Variablen mit dem Index -1 gespeichert

(Zeile 52). Anschließend wird in der Zeile 53 das Ende des Methodenaufrufs übermittelt und in der Zeile 54 die Variable `result` zurückgegeben.

Instrumentierung der Bedingungsüberdeckung

Die Instrumentierung von bedingten Sprunganweisungen im Bytecode ist in der Abbildung 7.10 dargestellt. Die Abbildung 7.10a zeigt den Quelltext der ursprünglichen If-Anweisung mit Kurzschlussauswertung. Durch den Java-Compiler wird diese in Bytecode (vgl. Abbildung 7.10b) übersetzt. Die Kurzschlussauswertung wird durch zwei bedingte Sprunganweisungen in den Zeilen 2 und 4 realisiert. Das Ergebnis der Instrumentierung ist in der Abbildung 7.10c dargestellt. An den Stellen im Bytecode, die mit „« `condition` *x* »“ bezeichnet sind, wird nach dem oben beschriebenen Schema ein entsprechendes Bedingungsüberdeckungsereignis gesendet. Tatsächlich werden für das Senden eines solchen Ereignisses 3 Bytecode-Instruktionen benötigt. Um die Lesbarkeit zu verbessern, sind die einzelnen Instruktionen hier nicht explizit aufgeführt. Neben den Instruktionen zum Senden der Bedingungsüberdeckungsereignisse werden in den Zeilen 7 und 10 zusätzliche Sprungbefehle eingefügt, um die Bedingungsüberdeckung exakt bestimmen zu können. Die Ausführung dieser `if`-Anweisung erzeugt in Abhängigkeit von der lokalen Variablen `i` drei Überdeckungsmengen: $\{8\}$, $\{3, 11\}$ und $\{3, 6\}$. Die Elemente der Überdeckungsmengen repräsentieren die entsprechenden Bedingungsüberdeckungsereignisse. Nach diesem Verfahren kann auch für Mehrfachverzweigungen und Schleifen die Bedingungsüberdeckung gemessen werden.

Registrierung einer unbekannten Klasse

Während der Ausführung des Prüflings wird die Struktur der verwendeten Klassen von speziellen Debugging-Ereignissen an die Debugger-VM gesendet. Dabei wird ausschließlich die Struktur derjenigen Klassen übermittelt, die während der Ausführung des Prüflings verwendet werden. Die Klassenstruktur einer Klasse wird immer unmittelbar vor dem ersten Zugriff auf ein Element der Klasse übermittelt. Zur Umsetzung des Registrierungsmechanismus wird jede Klasse durch den Instrumentierungsprozess um ein

Attribut und 3 Methoden erweitert. Die Erweiterungen sind in Listing 7.5 dargestellt.

Zunächst wir jede Klasse um das statische Attribut `$$$classRegistered$$$` ergänzt. Dieses Attribut wird nach der Registrierung einer Klasse auf `true` gesetzt, um zu verhindern, dass dieselbe Klasse mehrmals registriert wird. Die Registrierung einer Klasse kann durch die Instanz-Methode `$$$registerClass$$$` oder die statische Methode `$$$staticRegisterClass$$$` ausgelöst werden. Die erste Methode wird benötigt, wenn auf das Attribut eines Objekts o zugegriffen wird. Bevor das Ereignis über den Zugriff auf das Instanzattribut gesendet wird, muss sichergestellt werden, dass die Klasse des Objekts o bereits registriert ist. Hierzu wird die `$$$registerClass$$$`-Methode des Objekts o aufgerufen. Die Registrierungsmethode muss in diesem Fall dynamisch gebunden werden, da die Klasse des Objekts o erst zur Laufzeit bekannt ist. Die dynamisch gebundene Registrierungsmethode leitet den Aufruf an

```
1 if (i < 0 || i > 0) {
2   i++;
3 }
```

(a) If-Anweisung mit Kurzschlussauswertung.

```
1 ILOAD 0
2 IFLT  5
3 ILOAD 0
4 IFLE  6
5 IINC  0 1
6 ...
```

(b) Bytecode der Kurzschlussauswertung.

```
1  ILOAD 0
2  IFLT  8
3  << condition 3 >>
4  ILOAD 0
5  IFLE  11
6  << condition 6 >>
7  GOTO  9
8  << condition 8 >>
9  IINC  0 1
10 GOTO  12
11 << condition 11 >>
12 ...
```

(c) Instrumentierte Kurzschlussauswertung.

Abbildung 7.10: Instrumentierung einer `if`-Anweisung mit Kurzschlussauswertung.

Listing 7.5: Registrierungsmechanismus für die Klasse `Foo`.

```
1  private static boolean $$$classRegistered$$$ = false;
2
3  public void $$$registerClass$$$() {
4    Foo.$$$staticRegisterClass$$$();
5  }
6
7  public static void $$$staticRegisterClass$$$() {
8    if (Foo.$$$classRegistered$$$ == false) {
9      // Oberklasse registrieren
10     Object.$$$staticRegisterClass$$$();
11
12     // Klasse registrieren
13     DebuggingEvent.registerClass(Modifier.PUBLIC,
          "Foo", "java/lang/Object");
14
15     // Klassenattribute registrieren
16
17     // Instanzattribute registrieren
18     Foo.$$$registerInstanceAttributes$$$("Foo");
19
20     // Methoden registrieren
21     DebuggingEvent.registerMethod(Modifier.PUBLIC,
          "Foo", "bar", "()I", 4, 4);
22     DebuggingEvent.registerLocalVariable("Foo#bar()I",
          "Ljava/lang/Throwable;", "return", -2);
23     DebuggingEvent.registerLocalVariable("Foo#bar()I",
          "Ljava/lang/Throwable;", "return", -1);
24     DebuggingEvent.registerLocalVariable("Foo#bar()I",
          "LFoo;", "this", 0);
25     ...
26     DebuggingEvent.registerLineNumber("Foo#bar(I)I", 5,
          0);
27     DebuggingEvent.registerLineNumber("Foo#bar(I)I", 6,
          20);
28     ...
29     Foo.$$$classRegistered$$$ = true;
30   }
31 }
```

die statische Registrierungsmethode `$$$staticRegisterClass$$$` weiter.

Wenn ein Ereignis über den Zugriff auf ein Klassenattribut gesendet werden soll, dann wird die statische Registrierungsmethode direkt aufgerufen, da die betroffene Klasse bereits vor der Laufzeit bekannt ist. Innerhalb der statischen Registrierungsmethode wird mithilfe des Attributs `$$$classRegistered$$$` sichergestellt, dass die Klasse nur einmalig registriert wird. Falls eine Registrierung noch nicht stattgefunden hat, wird zunächst dafür gesorgt, dass die Oberklasse registriert ist (Zeile 10). Anschließend folgt in Zeile 13 die Registrierung der Klasse selbst. Die Registrierung der Klassenattribute entfällt in diesem Beispiel, da diese in der Klasse `Foo` nicht vorhanden sind. Das instrumentierte statische Attribut wird nicht registriert, weil es nicht Gegenstand des Debugging-Prozesses ist.

Die Registrierung der Instanzattribute wird in der Zeile 18 durch den Aufruf der Methode `$$$registerInstanceAttributes$$$` durchgeführt. Die Funktionsweise dieser Methode wird im nachfolgenden Abschnitt beschrieben.

Im Anschluss an die Übermittlung der Attribute folgt die Registrierung der Methoden. Für jede in einer Klasse definierten Methode werden zunächst die Modifizierer und die Signatur der Methode registriert (Zeile 21). Anschließend werden alle lokalen Variablen der Methode (Zeile 22 und folgende) übermittelt. Durch die Registrierung der Zeilennummern (Zeile 26 und folgende) können später Debugging-Ereignisse den Zeilen im Quelltext der Methode zugeordnet werden. Am Ende wird das `$$$classRegistered$$$`-Attribut auf `true` gesetzt, um die erneute Registrierung der Methode zu unterbinden.

Registrierung von Instanzattributen

Instanzattribute werden in Java entlang der Klassenhierarchie vererbt. Aus diesem Grund müssen auch die vererbten Instanzattribute der Oberklassen für die Klasse `Foo` registriert werden. Die Registrierung der deklarierten und geerbten Instanzattribute ist notwendig, da in der Debugger-VM alle Attribute einer Instanz der Klasse `Foo` bekannt sein müssen. Für die Registrierung der Instanzattribute wird die Methode `$$$registerInstanceAttributes$$$` (vgl. Listing 7.6) aufgerufen. Das Argument der Methode, `className`, legt fest, für welche Klasse die Instanzattribute registriert

werden. In der Zeile 2 wird zunächst die `$$$registerInstanceAttributes$$$`-Methode der Oberklasse aufgerufen. Auf diese Weise wird der Aufruf zur Registrierung der Instanzattribute entlang des gesamten Vererbungspfades bis zur Klasse `Object` nach oben gereicht. Nachdem der Aufruf zur Oberklasse weitergeleitet wurde, werden die Instanzattribute der Klasse registriert (Zeilen 3–4).

Listing 7.6: Registrierungsmechanismus für die Instanzattribute der Klasse `Foo`.

```
1 public static void
    $$$registerInstanceAttributes$$$(String className) {
2   Object.$$$registerInstanceAttributes$$$(className);
3   DebuggingEvent.registerAttribute(Modifiers.PRIVATE,
      className, "I", "value");
4   DebuggingEvent.registerAttribute(Modifiers.PRIVATE,
      className, "LFoo;", "next");
5 }
```

Registrierung einer unbekannten Instanz

Während der Ausführung des Prüflings wird die Erzeugung und Initialisierung neuer Objekte vollständig in Form von Debugging-Ereignissen protokolliert. Einige Objekte werden aber bereits in der Initialisierungsphase der Prüflings-VM erzeugt. Bei diesen handelt es sich im Wesentlichen um Instanzen der Bootstrapping-Klassen (siehe Abschnitt 7.3.2). Wenn die Ausführung des Prüflings auf ein bisher nicht protokolliertes Objekt trifft, dann wird die Übermittlung der Objektdaten an die Debugger-VM nachgeholt, indem die `$$$reportObject$$$`-Methode (vgl. Listing 7.7) des nicht protokollierten Objekts aufgerufen wird.

Während der Ausführung wird zunächst die ID des ausführenden Threads ermittelt (Zeile 2). Die nachträgliche Registrierung des Objekts soll später durch die Benutzeroberfläche von JHyde nicht in der Ereignisliste des ausführenden Threads angezeigt werden. Aus diesem Grund wird durch die

Listing 7.7: Erzeugte Methoden zur Übermittlung einer unbekannten Instanz.

```
1  public void $$$reportObject$$$() {
2    long tId =
        ObjectIdMap.getObjectId(Thread.currentThread());
3    DebuggingEvent.beginHide(tId);
4    DebuggingEvent.createObject(this, tId, "Foo", 0);
5    Foo.$$$reportObjectValues$$$(this, tId);
6    DebuggingEvent.endHide(tId);
7  }
8
9  public static void $$$reportObjectValues$$$(Object obj,
      long tId) {
10   Object.$$$reportObjectValues$$$(obj, tId);
11   Foo foo = (Foo) obj;
12   DebuggingEvent.attributeChange(foo.value, obj, tId,
        0, "Foo#value");
13   DebuggingEvent.attributeChange(foo.next, obj, tId, 0,
        "Foo#next");
14 }
```

Ereignisse in den Zeilen 3 und 6 ein unsichtbarer Abschnitt in der Ereignisfolge des ausführenden Threads definiert. Innerhalb des unsichtbaren Abschnitts wird zunächst die Erzeugung des Objekts protokolliert (Zeile 4) und anschließend die Methode `$$$reportObjectValues$$$` aufgerufen.

Die Methode `$$$reportObjectValues$$$` übermittelt die Werte der Instanzattribute. Hierfür wird der Aufruf zunächst an die Oberklasse delegiert, um die Werte der geerbten Instanzattribute zu übermitteln (Zeile 10). Im Anschluss werden die Werte der Instanzattribute der Instanz `obj` übermittelt (Zeilen 12–13).

Die Klasse DebuggingEvent

Die Klasse `DebuggingEvent` ist die Schnittstelle zwischen Prüfling und Transmitter. Sie delegiert die Aufrufe durch die instrumentierten Klassen des Prüflings an die Sendeeinheit des Transmitters. Darüber hinaus übernimmt sie die folgenden Aufgaben:

Aussetzung des Debugging-Modus - Zu Beginn des Aufrufs einer Ereignis-Methode der Klasse `DebuggingEvent` wird der Debugging-Modus für den aufrufenden Thread ausgeschaltet. Während der gesamten Übermittlung des Ereignisses durch die Klasse `DebuggingEvent` und den Transmitter werden somit keine ungewollten Debugging-Ereignisse erzeugt. Unmittelbar vor der Beendigung des Aufrufs der Ereignis-Methode wird der Debugging-Modus wieder aktiviert, so dass im weiteren Verlauf der Ausführung des Prüflings wieder Debugging-Ereignisse gesendet werden.

Dynamische Registrierung - Die Klasse `DebuggingEvent` löst die Registrierung aller Klassen aus, die während der Ausführung des Prüflings verwendet werden. Darüber hinaus sorgt sie für die nachträgliche Registrierung der Objekte, die in der Debugger-VM noch nicht bekannt sind. Zur Registrierung werden bei Bedarf die oben beschriebenen Registrierungsmethoden aufgerufen.

ID-Management - In den an die Debugger-VM gesendeten Ereignissen werden bestimmte Elemente, wie Klassen, Attribute und Methoden, über IDs referenziert. Die IDs dieser Elemente werden dynamisch zur Laufzeit vergeben werden. In den Aufrufen der Methoden der Klasse `DebuggingEvent`, mit denen die Klassen der Prüflings-VM instrumentiert sind, werden diese Elemente über `String`-Instanzen referenziert. Ein Beispiel hierfür ist der Aufruf `DebuggingEvent.attributeUse(this, tId, 42, "Foo#next");`, indem das betroffene Attribut über `"Foo#next"` referenziert wird. Die Klasse `DebuggingEvent` verwaltet mehrere Hashtabellen, mit denen sie die `String`-Referenzen in IDs umgewandelt, bevor sie die Ereignisse an den Transmitter weiterleitet. Durch die Umwandlung von String-Referenzen in IDs benötigen die Ereignisse erheblich weniger Speicherplatz und lassen sich daher wesentlich effizienter versenden. Ein weiterer Aspekt des ID-Managements ist die Verwaltung der Objekt-IDs. Die Klasse `DebuggingEvent` verwaltet die Hashtabelle `ObjectIdMap`, in der jeder Objektreferenz eine eindeutige ID zugeordnet wird.

7.3.4 Architektur

Die wichtigsten Klassen des dynamischen Instrumentierungsprozesses sind im Klassendiagramm der Abbildung 7.11 dargestellt. Durch die farbliche Hinterlegung werden zwei Gruppen von Klassen unterschieden: Die weiß hinterlegten Klassen sind Teil der Implementierung des Instrumentierers. Die grau hinterlegten Klassen sind Teil der Java-API oder des ASM-Frameworks.

Für die Steuerung des Instrumentierungsprozesses ist die Klasse `ByteCodeTransformer` zuständig. Sie implementiert das Interface `ClassFileTransformer` der Java-Instrumentation-API und kann somit als Instrumentierer für dynamisch geladene Klassen registriert werden. Zur Instrumentierung der Java-Klassen greift der `ByteCodeTransformer` auf die Klassen des ASM-Frameworks zurück.

Das ASM-Framework basiert auf dem Entwurfsmuster des Besuchers [55, S. 331–344]. Die wichtigsten Besucherschnittstellen für den Instrumentierungsprozess sind das `ClassVisitor`-Interface und das `MethodVisitor`-Interface. Durch die Implementierung dieser Schnittstellen ist es möglich, benutzerdefinierte Besucher zu erzeugen, die die gewünschte Instru-

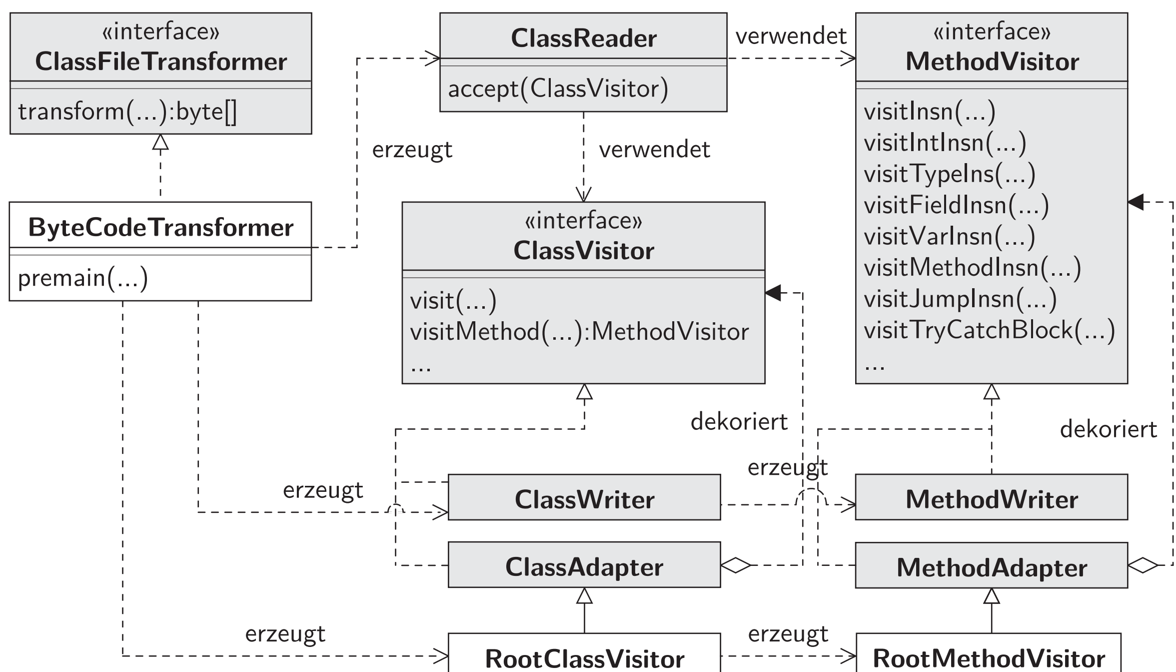


Abbildung 7.11: Klassendiagramm der wichtigsten Instrumentierungsklassen.

mentierung durchführen. Während des Instrumentierungsprozesses sorgen die Klassen `RootClassVisitor` und `RootMethodVisitor` für die im vorangegangenen Abschnitt beschriebene Instrumentierung der Klassen.

Die Klasse `RootClassVisitor` erweitert die Klasse `ClassAdapter`, die eine Basisimplementierung des `ClassVisitor`-Interfaces bereitstellt. Ein `ClassAdapter` besitzt eine Referenz auf eine `ClassVisitor`-Instanz und delegiert in allen `ClassVisitor`-Methoden den Aufruf an diese Instanz. Die Klasse `ClassAdapter` setzt somit das Dekorierermuster [55, S. 177–184] um und wird dazu verwendet, eine Kette von `ClassVisitor`-Instanzen zu erzeugen. Das ASM-Framework stellt mit der `ClassWriter`-Klasse eine weitere Implementierung des `ClassVisitor`-Interfaces bereit. Der `ClassWriter` ist für die Serialisierung zuständig. Er bildet das Ende einer Kette von `ClassVisitor`-Instanzen und erzeugt aus den empfangenen Methodenaufrufen den Bytecode der instrumentierten Java-Klasse.

Die Klasse `ClassReader` ist für die Deserialisierung einer Java-Klasse verantwortlich. Hierfür liest sie eine Java-Klasse ein und übermittelt deren Struktur in Form von `visit`-Aufrufen an den `ClassVisitor`, der durch den Aufruf der `accept`-Methode übergeben wird. Der `ClassReader` simuliert auf diese Weise die Objektstruktur der Java-Klasse, die durch den `ClassVisitor` besucht wird.

Das `MethodVisitor`-Interface definiert die Schnittstelle zum Besuch des Bytecodes einer Java-Methode. Das ASM-Framework stellt auch für das `MethodVisitor`-Interface zwei Implementierungen bereit. Der `MethodWriter` wandelt die besuchten Instruktionen in Bytecode um und der `MethodAdapter` stellt wiederum die Grundfunktionen des Dekorierermusters bereit. Der `RootMethodVisitor` erweitert die Klasse `MethodAdapter` und ist für die Instrumentierung der Instruktionsliste der besuchten Methode verantwortlich.

7.3.5 Instrumentierungsprozess

In Folgenden wird der Instrumentierungsprozess für Java-Klassen in der Prüflings-VM erläutert. Zunächst wird dargestellt, wie die Instanzen der

`Visitor`-Klassen für die Instrumentierung angeordnet werden. Anschließend werden die dynamische Instrumentierung, die statische Instrumentierung und die Instrumentierung der Registrierungsmechanismen beschrieben.

7.3.5.1 Komposition der `Visitor`-Instanzen

Das Zusammenspiel zwischen dem `ClassReader` und den `Visitor`-Instanzen ist im Kommunikationsdiagramm der Abbildung 7.12 dargestellt. Der `ClassReader` liest den Bytecode einer nicht instrumentierten Klasse ein und nimmt den `RootClassVisitor` als Besucher entgegen. Der `RootClassVisitor` nimmt die Aufrufe der `visit`-Methoden durch den `ClassReader` entgegen und leitet diese mit ggf. ergänzten `visit`-Aufrufen an den `ClassWriter` weiter.

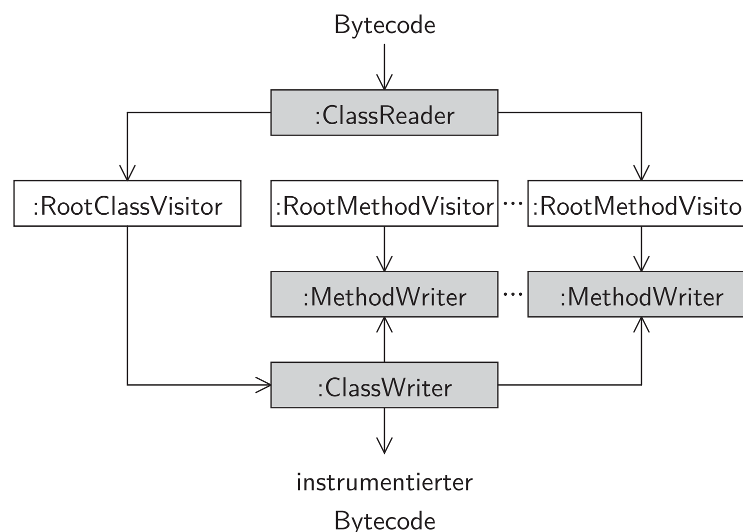


Abbildung 7.12: Kommunikationsdiagramm des Instrumentierers.

Wenn die `visitMethod`-Methode des `ClassWriter` aufgerufen wird, dann erzeugt dieser einen neuen `MethodWriter` für die Instruktionen der Methode und gibt diesen an den `RootClassVisitor` zurück. Der `RootClassVisitor` dekoriert diesen mit einem `RootMethodVisitor`, den er an den `ClassReader` zurückgibt. Anschließend übermittelt der `ClassReader` die Instruktionsliste an diesen `RootMethodVisitor`. Der `RootMethodVisitor` leitet die empfangenen Aufrufe an den

MethodWriter weiter und ergänzt sie um weitere visit-Aufrufe, so dass die Instruktionsliste nach dem im vorausgegangenen Abschnitt beschriebenen Schema erweitert wird. Der MethodWriter wandelt die empfangenen Methodenaufrufe schließlich in Bytecode um.

Wie durch das Kommunikationsdiagramm angedeutet, wird für jeden Methodenaufruf eine neue Visitor-Kette aus RootMethodVisitor und MethodWriter erzeugt. Der ClassWriter speichert während des Instrumentierungsprozesses alle MethodWriter-Instanzen. Erst unmittelbar vor der Erzeugung des instrumentierten Bytecodes der Java-Klasse werden die instrumentierten Instruktionslisten der MethodWriter-Instanzen ausgelesen und zum vollständigen Bytecode der Java-Klasse zusammengefügt. Aufgrund dieses Verfahrens muss die Erzeugung der instrumentierten Java-Klasse nicht strikt dem im Abschnitt 2.3.6 gezeigten Aufbau einer Java-Klasse folgen. Somit können zum Beispiel auch während der Instrumentierung einer Methode noch weitere Einträge zum Konstantenpool hinzugefügt werden. Wenn die instrumentierten Methoden direkt in den Bytecode der instrumentierten Klasse geschrieben würden, dann ließen sich Elemente, die, wie der Konstantenpool, in der Struktur der Klassendatei vor den Methodendefinitionen stehen, nicht mehr verändern. Da die Zusammenstellung der instrumentierten Klasse erst ganz am Ende der Instrumentierung stattfindet, können alle Elemente der Klassendatei während des gesamten Instrumentierungsprozesses manipuliert werden.

7.3.5.2 Dynamische Instrumentierung

Der Ablauf des Instrumentierungsprozesses ist im Sequenzdiagramm der Abbildung 7.13 dargestellt. Die Instrumentierung wird durch den Aufruf der Methode transform der ByteCodeTransformer-Instanz angestoßen. Über diesen Methodenaufruf wird der Bytecode der zu instrumentierenden Java-Klasse in Form des byte-Arrays src übergeben. Für die Deserialisierung des Bytecodes wird ein neuer ClassReader und für die Serialisierung der instrumentierten Java-Klasse ein neuer ClassWriter erzeugt. Der ClassWriter wird zusätzlich durch einen neuen RootClassVisitor dekoriert.

Der Instrumentierungsprozess wird durch den Aufruf der accept-Methode gestartet. Der ClassReader übermittelt die Struktur der Java-Klasse

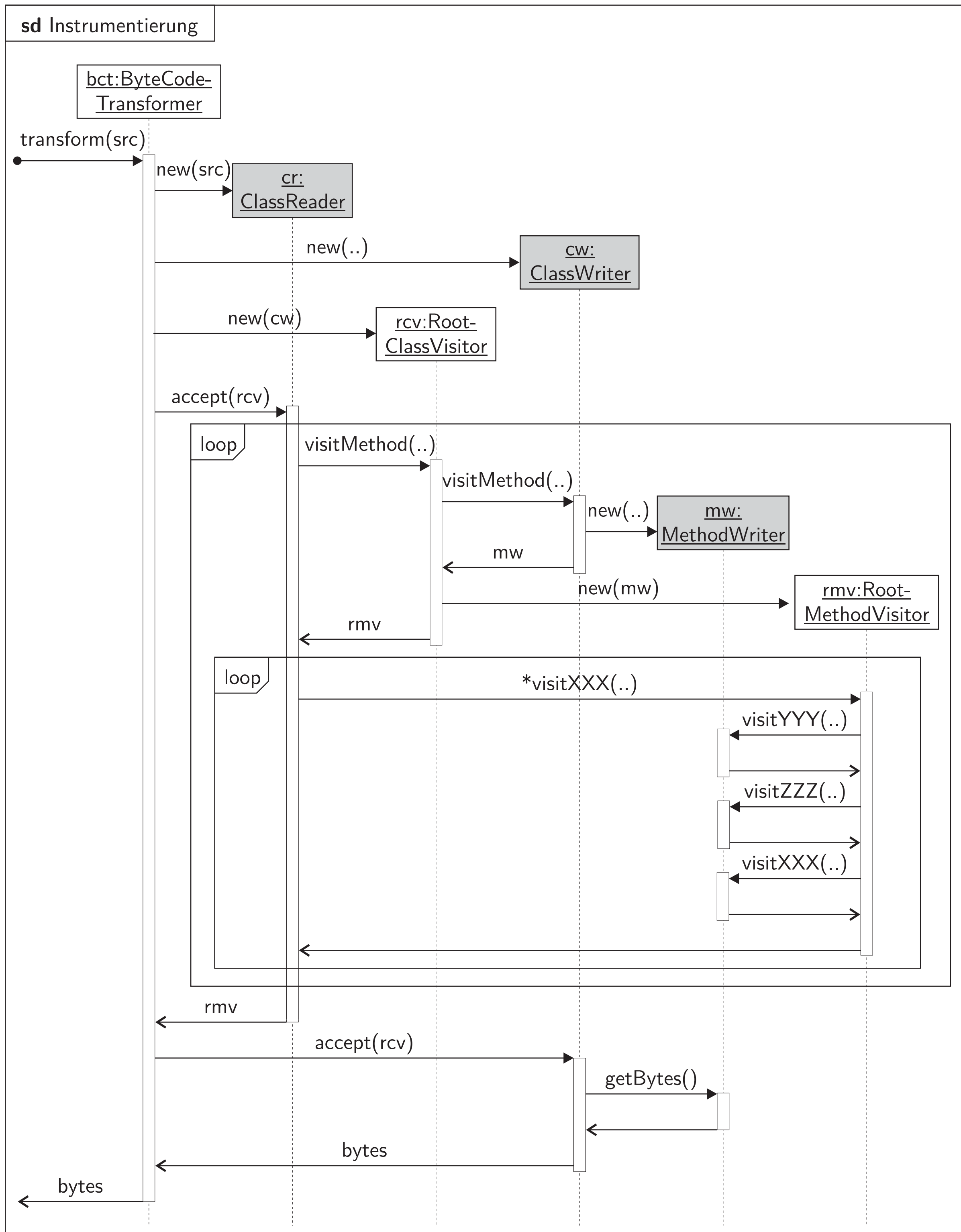


Abbildung 7.13: Sequenzdiagramm des Instrumentierungsprozesses.

durch den Aufruf der Besuchermethoden des übergebenen RootClass-Visitor. Im Sequenzdiagramm ist exemplarisch die Instrumentierung einer Methode angedeutet. Die Instrumentierung der Methode ist von einer Schleife umgeben, um anzudeuten, dass im Verlauf der Instrumentierung alle Methoden der Klasse nach demselben Schema instrumentiert werden. Die Instrumentierung einer Methode beginnt mit dem Aufruf der Methode visitMethod der RootClassVisitor-Instanz. Diese leitet den Aufruf zunächst an den ClassWriter weiter. Der ClassWriter erzeugt für die besuchte Methode einen neuen MethodWriter, den er an den RootClassVisitor zurückgibt. Der RootClassVisitor dekoriert den MethodWriter mit einem neuen RootMethodVisitor, den er anschließend an den ClassReader zurückgibt.

Während der Ausführung der inneren Schleife sendet der ClassReader die Instruktionsliste der besuchten Methode in Form von Aufrufen der Besuchermethoden des MethodVisitor-Interfaces an den RootMethod-Visitor. Der Aufruf von visitXXX repräsentiert den Aufruf einer beliebigen Methode des MethodVisitor-Interfaces. Der RootMethodVisitor ergänzt die empfangene Instruktion bei Bedarf um weitere Instruktionen und leitet beide an den MethodWriter weiter. Die instrumentierten Instruktionen werden hier durch die Aufrufe visitYYY und visitZZZ repräsentiert. Im Anschluss wird die Kontrolle an den ClassReader zurückgegeben und die Ausführung der inneren Schleife mit der nächsten Anweisung in der Instruktionsliste fortgesetzt. Wenn auf diese Weise die gesamte Instruktionsliste instrumentiert wurde, setzt die äußere Schleife die Instrumentierung mit der nächsten Methode fort.

Nachdem der ClassReader die gesamte Struktur der Java-Klasse übermittelt hat, gibt er die Kontrolle zurück an den ByteCodeTransformer. Dieser ermittelt daraufhin den instrumentierten Bytecode durch den Aufruf der Methode toByteArray der ClassWriter-Instanz. Der Class-Writer ruft auf allen MethodWriter-Instanzen, die er während des Instrumentierungsprozesses der Klasse erzeugt hat, die Methode getBytes auf. Auf diese Weise sammelt er für jede Methode den Bytecode der Instruktionsliste ein. Der ClassWriter fügt die gesammelten Bytecodes zur instrumentierten Java-Klasse zusammen und gibt diese in Form eines byte-Arrays an den ByteCodeTransformer zurück. Der ByteCode-Transformer übergibt den instrumentierten Bytecode wiederum an

die Java-Instrumentation-API. Die Instrumentation-API sorgt dann dafür, dass die instrumentierte Klasse geladen wird.

7.3.5.3 Statische Instrumentierung

Die statische Instrumentierung ist im Wesentlichen identisch zur dynamischen Instrumentierung. Der einzige Unterschied besteht darin, dass die `transform`-Methode der `ByteCodeTransformer`-Instanz nicht beim Laden einer Klasse durch die Java-Instrumentation-API aufgerufen wird.

Im Abschnitt 7.3.2 wurde gezeigt, dass alle Bootstrapping-Klassen und alle Klassen der Java-API, die durch den Instrumentierer oder den Transmitter benutzt werden, statisch instrumentiert werden müssen. Diese Klassen befinden sich in der Datei `./lib/rt.jar` des JRE. Durch den statischen Instrumentierungsprozess werden diese Klassen instrumentiert und in der Datei `bootstrapping.jar` gespeichert.

Die Position der Datei `rt.jar` ist im sogenannten `bootclasspath` der Java-VM gespeichert. Der `bootclasspath` enthält eine Folge von Verzeichnissen und Archiv-Dateien, die für die Suche nach den Klassen der Java-API verwendet wird. Wenn eine Klasse in die Java-VM geladen werden soll, dann werden zuerst die Einträge des `bootclasspath` nach der entsprechenden Klassendatei durchsucht. Wird eine Klassendatei an mehreren Stellen im `bootclasspath` gefunden, dann wird sie von der Stelle geladen, die am weitesten vorne im `bootclasspath` definiert ist. Über die Kommandozeilenoption `-Xbootclasspath:/p:bootstrapping.jar` wird der ursprünglichen `bootclasspath`-Folge der Eintrag der Datei `bootstrapping.jar` vorangestellt. Dies hat zur Folge, dass für alle Klassen, die in der Datei `bootstrapping.jar` definiert sind, nicht die ursprüngliche Version aus der Datei `rt.jar`, sondern die instrumentierte Version aus der Datei `bootstrapping.jar` geladen wird.

7.3.5.4 Instrumentierung der Registrierungsmechanismen

Um die Registrierungsmechanismen (vgl. Abschnitt 7.3.3) zu instrumentieren, müssen der Klasse weitere Methoden hinzugefügt werden. Wie in der

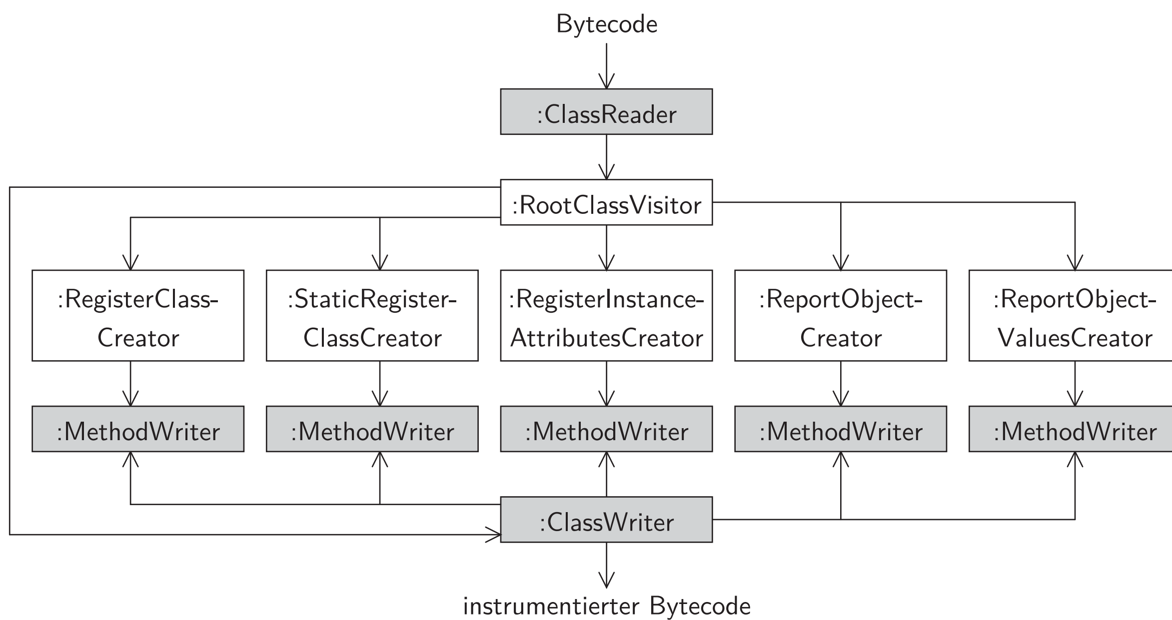


Abbildung 7.14: Instrumentierung der Registrierungsmechanismen.

Abbildung 7.14 dargestellt, werden für die zu erzeugenden Methoden Instanzen der Klassen `RegisterClassCreator`, `StaticRegisterClassCreator`, `RegisterInstanceAttributes`, `ReportObjectCreator` und `ReportObjectValuesCreator` erzeugt, die alle das Interface `ClassVisitor` implementieren. Die `RootClassVisitor`-Instanz leitet die vom `ClassReader` empfangenen Methodenaufrufe an alle `Creator`-Instanzen weiter. Über die empfangenen Aufrufe der Besuchermethoden werden die `Creator`-Instanzen über alle Attribute und Methoden der zu instrumentierenden Klasse informiert. Diese Informationen werden verwendet, um mit den zugeordneten `MethodWriter`-Instanzen die Registrierungsmethoden zu erzeugen.

7.3.6 Konfiguration

Der Instrumentierer wird über die Konfigurationsdatei `InstrumentationConfig.properties` (vgl. Listing 7.8) konfiguriert. Diese Datei befindet sich im Paket (Package) `de.wwu.jhyde.instrumentation.config`. In der Datei können zwei Properties gesetzt werden.

Listing 7.8: Die Konfigurationsdatei `InstrumentationConfig.properties`.

```
1 de.wwu.jhyde.instrumentation.log4j.properties =
     log4j.properties
2 de.wwu.jhyde.instrumentation.static.list =
     static-instrumentation.txt
```

Der Wert des ersten Property ist eine relative Pfadangabe für die log4J-Properties-Datei, mit der für Pakete und Klassen des Instrumentierers individuelle Logger-Klassen und Log-Level definiert werden.

Das zweite Property enthält die relative Pfadangabe einer Textdatei, die die vollqualifizierten Namen aller statisch zu instrumentierenden Klassen enthält. Während des statischen Instrumentierungsprozesses werden die in der Textdatei angegebenen Klassen instrumentiert und in der `bootstrapping.jar`-Datei zusammengefasst.

7.4 Rekorder

In der Debugger-VM ist der Rekorder dafür zuständig, aus den empfangenen Ereignissen ein Modell zu erzeugen, welches den Programmablauf des Prüflings repräsentiert. Das Modell wird später dazu verwendet, den hybriden Debugging-Prozess durchzuführen. Im nachfolgenden Abschnitt (7.4.1) wird zunächst das Modell zur Repräsentation von Programmabläufen vorgestellt. Im Anschluss wird im Abschnitt 7.4.2 beschrieben, wie die im Kapitel 5 beschriebenen Debugging-Strategien, implementiert sind.

7.4.1 Modell des Programmablaufs

Das Modell, mit dem der Programmablauf von Prüflingen repräsentiert wird, sollte im Wesentlichen zwei Eigenschaften erfüllen:

- Die für den Debugging-Prozess benötigten Daten sollen sich schnell und effizient aus dem Modell auslesen lassen.

- Das Modell sollte kompakt sein und möglichst wenig Speicherplatz verbrauchen.

Die Anforderungen an das Modell sind nicht konfliktfrei, da für den effizienten Zugriff der Programmablauf in strukturierter Weise gespeichert werden muss. Die strukturierte Speicherung ermöglicht auf der einen Seite zwar ein effizientes Durchsuchen des Programmablaufs, benötigt auf der anderen Seite jedoch zusätzlichen Speicherplatz.

Das Modell, welches JHyde für die Repräsentation von Programmabläufen verwendet, ist ein Kompromiss zwischen effizientem Zugriff und Speicherplatzbedarf. Wie in der Abbildung 7.15 dargestellt, wird der aufgezeichnete Programmablauf sowohl im Arbeitsspeicher als auch auf einem externen Speicher abgelegt.

Die Erzeugung des Modells übernimmt die Klasse `Computation`. Diese Klasse implementiert das Interface `IJHydeEvents`. Über diese Schnittstelle empfängt eine `Computation`-Instanz vom Transmitter alle Ereignisse über den Programmablauf des Prüflings. Eine Instanz der Klasse `Computation` repräsentiert die vollständige Ausführung des Prüflings und aggregiert sämtliche Elemente des aufgezeichneten Programmablaufs.

Die `Computation`-Instanz ordnet jedem Ereignisse einen eindeutigen Zeitstempel (`Timestamp`) zu und leitet es zunächst an die `EventList-File`-Instanz weiter, um es in einer Datei auf einem externen Speicher

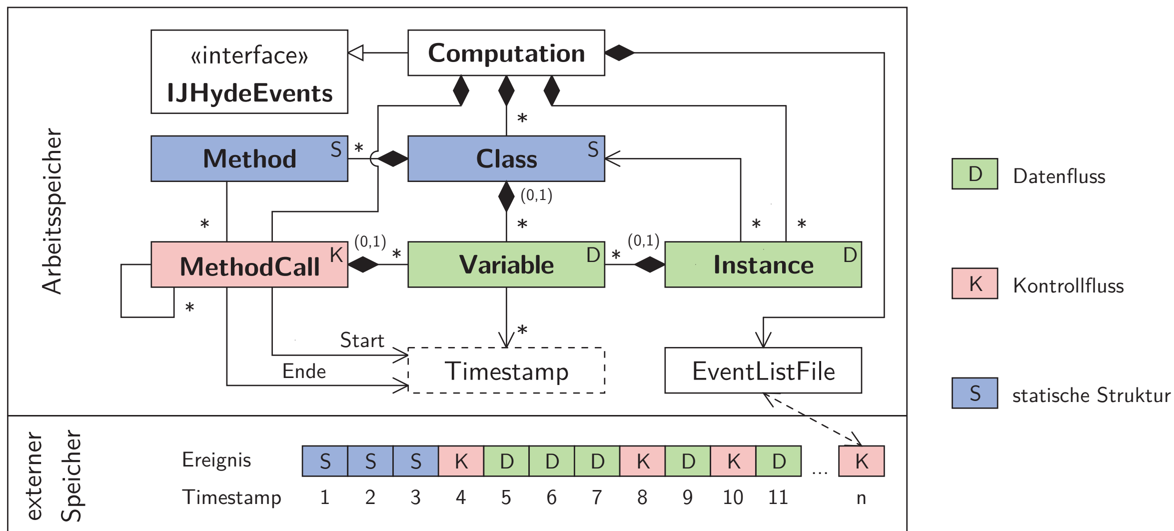


Abbildung 7.15: Datenmodell des Programmablaufs.

zu sichern. Der Zeitstempel ist ein `long`-Wert, durch den die eintreffenden Ereignisse sortiert werden. Ein Ereignis e_1, welches zeitlich vor einem e_2 aufgetreten ist, besitzt einen kleineren Zeitstempel als e_2. Die `EventListFile`-Instanz speichert alle Ereignisse der Reihenfolge nach, d. h. geordnet nach ihrem Zeitstempel. Die Position eines Ereignisses in der externen Ereignisdatei lässt sich direkt aus dem Zeitstempel des Ereignisses bestimmen. Da die Ereignisdatei wahlfreien Zugriff erlaubt, kann die `EventListFile` das Ereignis zu einem beliebigen Zeitstempel in $O(1)$ ermitteln.

Während der zweiten Phase des Debugging-Prozesses müssen mithilfe der Ereignisdatei Programmzustände rekonstruiert werden. Um einen Programmzustand in angemessener Zeit aus der Ereignisdatei rekonstruieren zu können, sind weitere Informationen über die Struktur der Ereignisdatei notwendig. Ohne diese Strukturinformationen würde der Aufwand für die Zustandsrekonstruktion mit der Anzahl der aufgezeichneten Ereignisse n mindestens in der Größenordnung $\Omega(n)$ wachsen. Für große Ereignisdateien ließe sich der Programmzustand dann nicht mehr in akzeptabler Laufzeit rekonstruieren.

Aus diesem Grund erzeugt die `Computation`-Instanz im Arbeitsspeicher der Debugging-VM weitere Datenstrukturen, die Informationen über die Struktur des Programmablaufs speichern. Diese Strukturen beschleunigen die Suche in der Ereignisdatei und ermöglichen es, Programmzustände schneller zu rekonstruieren. Die Datenstrukturen enthalten Informationen über den Datenfluss, den Kontrollfluss und die statische Struktur des aufgezeichneten Programms.

Die statische Struktur eines Prüflings wird durch `Class`- und `Method`-Instanzen repräsentiert. Für jede Klasse, die während der Ausführung des Prüflings registriert wird, erzeugt die `Computation` eine neue `Class`-Instanz. Die `Class`-Instanz speichert den Namen der Klasse sowie die Namen und Datentypen aller Klassen- und Instanzattribute. Darüber hinaus sind in einer Klasse alle definierten Methoden aggregiert. Eine `Method`-Instanz speichert die Signatur der Methode und die Namen und Datentypen aller lokalen Variablen.

Der Datenfluss des Prüflings wird durch `Instance`- und `Variable`-Instanzen repräsentiert. Eine `Instance`-Instanz stellt ein einzelnes Objekt

des Programmablaufs dar. Jede `Instance` speichert ihren Typ in Form einer Referenz auf die entsprechende `Class`-Instanz. Darüber hinaus aggregiert sie eine Menge von `Variable`-Instanzen, die die Instanzattribute repräsentieren. Alle `Instance`-Objekte, die während der Programmausführung erzeugt werden, sind unter ihrer Objekt-ID in einer Hashtabelle in der `Computation`-Instanz gespeichert. Die Hashtabelle ermöglicht es somit, zu einer Objekt-ID das entsprechende `Instance`-Objekt zu ermitteln.

Die Variablen des Prüflings werden durch `Variable`-Instanzen abgebildet. Eine `Variable` kann entweder eine lokale Variable, ein Klassenattribut oder ein Instanzattribut repräsentieren. Jede `Variable`-Instanz gehört somit entweder zu einem `MethodCall`, einer `Class` oder einer `Instance`. Im Wesentlichen besteht eine `Variable` aus einer aufsteigend sortierten Folge von Zeitstempeln (`Timestamp`). Jeder Zeitstempel dieser Folge referenziert ein Ereignis, das den Wert der Variablen geändert hat. Um den Wert einer Variablen zum Zeitpunkt t zu bestimmen, wird zunächst durch eine binäre Suche in der Folge von Zeitstempeln der größte Zeitstempel ermittelt, der kleiner oder gleich t ist. Anhand des ermittelten Zeitstempels kann über die `EventListFile` das zugehörige Ereignis und damit der Wert der Variablen zum Zeitpunkt t ermittelt werden.

Der Kontrollfluss eines Programms wird in Form von `MethodCall`-Instanzen aufgezeichnet. Ein `MethodCall` repräsentiert einen Methodenaufruf des Programmablaufs. Jeder `MethodCall` speichert die aufgerufene Methode anhand einer Referenz auf die entsprechende `Method`-Instanz. Des Weiteren aggregiert er eine Menge von `Variable`-Instanzen, die die lokalen Variablen des Aufrufs repräsentieren. Zwischen den Methodenaufrufen einer Berechnung bestehen Eltern-Kind-Beziehungen. Aufgrund dieser Beziehungen bilden sie eine Baumstruktur, den sogenannten Berechnungsbaum. Um den Berechnungsbaum einer Programmausführung rekonstruieren zu können, referenziert die `Computation`-Instanz die Wurzel des Berechnungsbaums. Zudem besitzt jeder `MethodCall` Referenzen auf seinen Elternknoten und alle Kindknoten.

Jeder `MethodCall` speichert zwei Zeitstempel, die den Beginn und das Ende des Methodenaufrufs markieren. Anhand dieser Zeitstempel ist es möglich, die Werte aller Variablen zu Beginn und am Ende des Methodenaufrufs nach dem oben beschriebenen Verfahren zu bestimmen. Der Wert

einer Variablen v zu einem beliebigen Zeitpunkt lässt sich durch binäre Suche mit einem Aufwand von $O\big(\log(w)\big)$ bestimmen, wobei w der Anzahl der Werte entspricht, die der Variablen v während des Programmablauf zugewiesen wurden.

Das verwendete Modell zur Aufzeichnung von Programmabläufen ist einen Kompromiss zwischen Geschwindigkeit und Speicherplatzbedarf. Da alle Ereignisse in einer Datei auf einem externen Speicher abgelegt sind, verringert sich der Arbeitsspeicherverbrauch des Modells, während die Zeit zur Rekonstruktion von Programmzuständen aufgrund des langsameren Zugriffs auf den externen Speicher steigt. Für die Durchführung der Defektsuche ist der zusätzliche Aufwand akzeptabel, da die Benutzeroberfläche von JHyde nur eine begrenzte Anzahl von Werten gleichzeitig darstellen kann. Für diese begrenzte Anzahl an Variablen lassen sich die Werte nahezu in Echtzeit bestimmen.

Der Arbeitsspeicherverbrauch dieses Modells ist trotz der Auslagerung der Ereignisse auf einen externen Speicher groß. Der meiste Speicherplatz wird für die `Variable`-Instanzen benötigt. Für jede Variable des Prüflings wird in diesem Modell eine `Variable`-Instanz erzeugt und jede dieser `Variable`-Instanzen speichert wiederum die Zeitstempel aller Wertänderungen. Der Arbeitsspeicherverbrauch der Debugger-VM ist daher um ein Vielfaches größer als der Arbeitsspeicherverbrauch der Prüflings-VM.

In Kapitel 8 werden einige Verfahren aus dem Bereich des Offline-Debugging vorgestellt, die interessante Methoden präsentieren, mit denen sich der Speicherverbrauch der Offline-Debugger reduzieren lässt. Einige dieser Ansätze ließen sich auch auf das Datenmodell von JHyde übertragen, um den Arbeitsspeicherverbrauch weiter zu reduzieren.

7.4.2 Debugging-Strategien

Im folgenden Abschnitt wird erläutert, wie die im Kapitel 5 beschriebenen deklarativen Debugging-Strategien implementiert sind. Eine deklarative Debugging-Strategie definiert auf den unklassifizierten Methodenaufrufen des Suchraums eine Reihenfolge, in der die Methodenaufrufe zu klassifizieren sind.

Die Abbildung 7.16 zeigt die Implementierung der Debugging-Strategien anhand eines Klassendiagramms. Die Schnittstelle `IDebuggingStrategy` definiert für alle Debugging-Strategien die zu implementierende Methode `next`. Die Methode erwartet als Argumente die `MethodCall`-Instanz `m` und eine `StateEvaluator`-Instanz `s`. Der `MethodCall` `m` entspricht dem gegenwärtig selektierten Methodenaufruf und der `StateEvaluator` `s` entspricht einer Hashtabelle, in der für jeden `MethodCall` die Klassifikation (`State`) gespeichert ist. Anhand des Methodenaufrufs `m` und den Klassifikationen der Methodenaufrufe `s` muss die Methode `next` den nächsten zu klassifizierenden Aufruf bestimmen und zurückgeben.

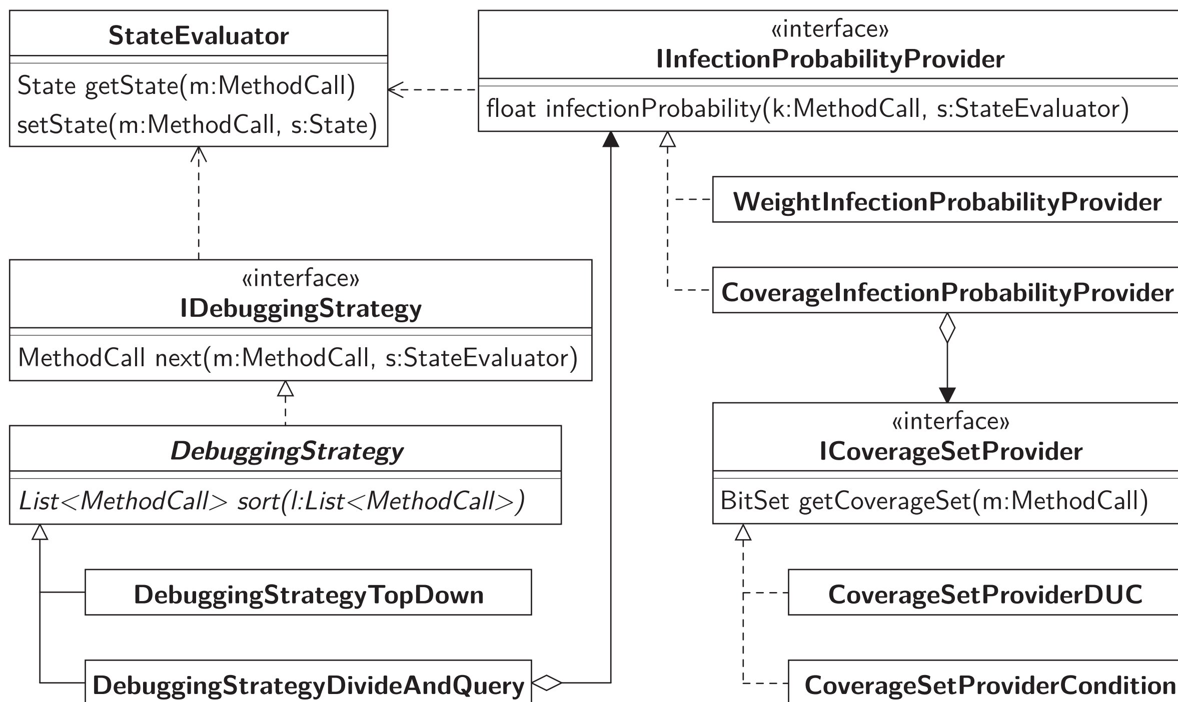


Abbildung 7.16: Klassendiagramm der Debugging-Strategien.

Die Klasse `DebuggingStrategy` implementiert die Schnittstelle `IDebuggingStrategy` und stellt die Grundfunktionalität für alle Debugging-Strategien bereit. Zu diesem Zweck definiert sie die abstrakte Schablonenmethode (Template Method) `sort` [55, S. 325–330]. Der Methode `sort` wird als Argument eine Liste aller unklassifizierten Methodenaufrufe des Suchbereichs übergeben. Die Aufgabe der Methode ist es, die Metho-

denaufrufe entsprechend der implementierten Debugging-Strategie zu sortieren und zurückzugeben. Die Methode `sort` übernimmt somit für alle Debugging-Strategien die Bestimmung der unklassifizierten Methoden des Suchbereichs und die Traversierung der sortierten und zurückgegebenen Methoden.

Die Klasse `DebuggingStrategyTopDown` implementiert die Methode `sort`, so dass alle unklassifizierten Methoden aufsteigend anhand des Zeitstempels des Methodenaufrufs sortiert werden. Dieses Vorgehen entspricht der Top-Down-Strategie.

Die Divide-and-Query-Strategie wird durch die Klasse `DebuggingStrategyDivideAndQuery` implementiert. Wie in den Abschnitt 5.2 und 5.3 beschrieben, wird bei dieser Strategie nach dem i-ten Klassifikationsschritt immer die Methode gewählt, deren Klassifikation den Erwartungswert der Größe des verbleibenden Suchraums minimiert. Dieser berechnet sich nach der Formel:

$$E(W_{i+1}^k) = w_i^k \cdot p_i^k + (W_i - w_i^k) \cdot (1 - p_i^k).$$

Für die Berechnung der Knotengewichte w_i^k werden Informationen über die strukturellen Beziehungen der `MethodCall`-Instanzen benötigt. Diese Informationen können aus den Eltern-Kind-Beziehungen, die als Referenzen in jedem `MethodCall` gespeichert sind, gewonnen werden. Das Gewicht eines Knotens k nach dem i-ten Klassifizierungsschritt ist definiert als $w_i^k = |S_i \cap T_k|$. Es entspricht somit der Anzahl der Methodenaufrufe, die sowohl Element des Suchbereichs S_i als auch Element des Teilbaums mit der Wurzel k, T_k, sind. Für die Berechnung der Infektionswahrscheinlichkeiten p_i^k ist der `DebuggingStrategyDivideAndQuery`-Instanz eine Instanz des Typs `IInfectionProbabilityProvider` zugeordnet.

Das Interface `IInfectionProbabilityProvider` definiert die Methode `infectionProbability(m:MethodCall, s:StateEvaluator)` zur Berechnung der Infektionswahrscheinlichkeit des Methodenaufrufs `m`. Die für die Berechnung benötigten Klassifikationen der Methodenaufrufe werden in Form der `StateEvaluator`-Instanz `s` als zweites Argument übergeben.

Die Schnittstelle `IInfectionProbabilityProvider` wird durch zwei Klassen implementiert. Die Klasse `WeightInfectionProbabil-`

ityProvider gibt nach dem i-ten Klassifikationsschritt für einen Methodenaufruf k die Infektionswahrscheinlichkeit $p_i^k = \frac{w_i^k}{W_i}$ zurück. Die Infektionswahrscheinlichkeit ist in diesem Fall proportional zum Gewicht w_i^k des Knotens k. Eine DebuggingStrategyDivideAndQuery-Instanz, der ein WeightInfectionProbabilityProvider zugeordnet ist, sortiert die Methoden des Suchbereichs somit exakt nach der in Abschnitt 5.2 beschriebenen einfachen D&Q-Strategie mit gewichtsproportionalen Infektionswahrscheinlichkeiten.

Der CoverageInfectionProbabilityProvider berechnet Infektionswahrscheinlichkeiten, die nicht auf dem Gewicht, sondern auf den Überdeckungsinformationen der Methodenaufrufe basieren. Während der Aufzeichnung des Programmablaufs eines Prüflings werden Ereignisse über die überdeckten Bedingungen sowie Ereignisse über die Lese- und Schreibzugriffe auf Variablen an die Computation-Instanz gesendet. Aus den Informationen über Lese- und Schreibzugriffe werden für jeden Methodenaufruf die überdeckten DU-Ketten ermittelt. Die Menge der überdeckten DU-Ketten und die Menge der überdeckten Bedingungen eines Methodenaufrufs werden in der entsprechenden MethodCall-Instanz gespeichert. Um zu entscheiden, welche Überdeckungsinformationen für die Berechnung von Infektionswahrscheinlichkeiten verwendet werden, ist dem CoverageInfectionProbabilityProvider eine Instanz des Typs ICoverageSetProvider zugeordnet.

Das Interface ICoverageSetProvider definiert die Methoden getCoverageSet(m:MethodCall), die für den Methodenaufruf m Überdeckungsinformationen in Form eines BitSet zurückgibt. Die Klasse BitSet ist Teil der Java-API und repräsentiert eine Menge in Form eines Bitvektors. Jedes Element des Vektors repräsentiert genau eine Überdeckungsentität. Wenn die Entität durch den Methodenaufruf überdeckt wurde, dann ist das entsprechende Element des Vektors true, andernfalls ist das entsprechende Element false.

Die Schnittstelle ICoverageSetProvider wird durch die Klassen CoverageSetProviderDUC und CoverageSetProviderCondition implementiert. Diese geben die Menge der überdeckten DU-Ketten bzw. die Menge der überdeckten Bedingungen des Methodenaufrufs m zurück.

Die Berechnung der Infektionswahrscheinlichkeiten eines Methodenaufrufs geschieht nach der Formel

$$p_i^k = \frac{1}{\sum_{a \in S_i} g_i(a)} \cdot \sum_{a \in (S_i \cap T_k)} g_i(a).$$

Sie berechnet sich somit aus der Summe der Infektionswerte g_i der Knoten, die sowohl Element des Suchraums S_i als auch Element des Teilbaums T_k sind. Um einen zulässigen Wahrscheinlichkeitswert aus dem Intervall $[0, 1]$ zu erhalten, wird diese Summe durch die Summe der Infektionswerte aller Knoten des Suchbereichs dividiert. Der Infektionswert eines Methodenaufrufs ist abhängig von den überdeckten Entitäten des Aufrufs und den bisherigen Klassifikationen. Die genaue Berechnung der Infektionswerte wurde bereits im Abschnitt 5.3.5 beschrieben.

Die Implementierung der Debugging-Strategien macht umfangreichen Gebrauch vom Strategie-Entwurfsmuster [55, S. 315]. Dies erleichtert die Erweiterung von JHyde um weitere Debugging-Strategien. Das Strategiemuster wird an drei Stellen durch die definierten Schnittstellen `IDebuggingStrategy`, `IInfectionProbabilityProvider` und `ICoverageSetProvider` umgesetzt. Die Schnittstelle `IDebuggingStrategy` erleichtert das Ergänzen neuer Debugging-Strategien. Die Schnittstelle `IInfectionProbabilityProvider` ermöglicht es, weitere Strategien für die Berechnung der Infektionswahrscheinlichkeit eines Methodenaufrufs zu implementieren. Zuletzt bietet die Schnittstelle `ICoverageSetProvider` die Möglichkeit, weitere Überdeckungskriterien für die Berechnung von Infektionswahrscheinlichkeiten hinzuzufügen.

7.5 Benutzeroberfläche

Die Benutzeroberfläche von JHyde ist ein Plugin für die Eclipse-IDE. Eclipse [40, 48] ist eine universelle, erweiterbare Umgebung zur Entwicklung von Software-Applikationen (IDE), die auf dem OSGi-Framework [122, 167] basiert. Das OSGi-Framework ist eine Spezifikation einer leichtgewichtigen, modul- und serviceorientierten Plattform für die Programmiersprache Java. Die durch die Spezifikation definierte Laufzeitumgebung stellt für Module,

welche als Bundles bezeichnet werden, ein vollständiges Lebenszyklusmanagement zur Verfügung. Bundles können während der Laufzeit installiert, gestartet, gestoppt und deinstalliert werden.

Um Namenskonflikte zwischen unterschiedlichen Bundles zu vermeiden, sieht die OSGi-Spezifikation für jedes Bundle einen separaten Klassenpfad vor. Während der Laufzeit verfügt jedes Bundle über einen eigenen Class-Loader, für den ausschließlich die Klassen und Ressourcen innerhalb des Bundles sichtbar sind. Um die Kooperation zwischen Bundles zu ermöglichen, kann ein Class-Loader die Anfrage zum Laden einer Klasse an assoziierte Class-Loader delegieren. Die Class-Loader des OSGi-Frameworks sind somit in Form eines Graphen angeordnet. Dieser Graph ermöglicht es die Interdependenzen zwischen den Bundles exakt abzubilden und vermeidet daher Namenskonflikte zwischen den Bundles. Im Vergleich zu einer hierarchischen Anordnung der Class-Loader ist die Anordnung in Form eines Graphen ein entscheidender Vorteil. In der Java EE, sind die Class-Loader zum Beispiel in Form eines Baumes angeordnet. Um eine Programmbibliothek für eine Menge A von Knoten des Baumes zugänglich zu machen, muss diese an dem Class-Loader bzw. Knoten v registriert werden, der gemeinsamer Vorfahre aller Knoten aus A ist. Damit wird die Verwendung der Bibliothek aber auch für alle anderen Nachkommen von v erzwungen, die nicht Element von A sind. Da die Abhängigkeiten zwischen Modulen keiner Hierarchie mit Eltern-Kind-Beziehungen, sondern einem Netzwerk aus Clients und Servern entsprechen, können diese durch eine Baumstruktur nur unzureichend abgebildet werden. Das OSGi-Framework ermöglicht es, Abhängigkeiten der Bundles durch ein Netzwerk von Class-Loader-Instanzen exakt abzubilden.

Die Eclipse IDE lässt sich über Bundles, die als Plugins bezeichnet werden, dynamisch erweitern. Zu diesem Zweck können Plugins sogenannte Extension-Points definieren. Über einen Extension-Point können andere Plugins von außen zu der Funktionalität eines Plugins beitragen. Diese Erweiterungen werden Extensions genannt. Ein Plugin kann durch die Menge der definierten Extension-Points exakt definieren, welche Teile des Moduls öffentlich sind und somit erweitert werden können. Über die Menge der definierten Extensions kann ein Plugin die Extension-Points anderer Plugins erweitern. Die Plugins des Eclipse-Frameworks basieren auf dem Plugin-Entwurfsmuster [53, S. 544–548]. Dieses ermöglicht durch eine zen-

tralisierte Konfiguration die Komposition einer Menge von Plugins für die Laufzeitumgebung. Die Komposition der Plugins wird in der Regel durch eine zentrale Konfigurationsdatei gesteuert. Da die Komposition nicht statisch durch den Programmcode festgelegt ist, müssen die Programmteile bei einer Änderung der Komposition nicht erneut kompiliert werden.

Die Benutzeroberfläche besteht aus vier Ansichten: der Berechnungsbaumansicht, der Knotenansicht, der Ereignisansicht und der Variablenansicht. Jede dieser Ansichten bietet eine unterschiedliche Sicht auf den aufgezeichneten Programmablauf eines Prüflings. Die vier Ansichten verwenden die Klasse `TreeModelViewer` des Eclipse-Frameworks, um Elemente des Programmablaufs in einer Baumstruktur abzubilden.

Für die Darstellung der Elemente des Programmablaufmodells eines Prüflings werden mehrere Adapter benötigt, über die der `TreeModelViewer` auf die Inhalte und die Struktur des Modells des Programmablaufs zugreifen kann. Die Verwendung dieser Adapter ist notwendig, da der `TreeModelViewer` des Eclipse-Frameworks keine Kenntnis über die Typen des Programmablaufmodells besitzt. Die Adapterklassen transformieren das Programmablaufmodell in ein Modell, dessen Strukturen der `TreeModelViewer` darstellen kann.

Die Funktionsweise der Adapterklassen ist in der Abbildung 7.17 veranschaulicht. In der Abbildung sind die Klassen, die Teil des Eclipse-Frameworks sind, grau hinterlegt. Der `TreeModelViewer` verwendet die folgenden Adapter-Klassen.

`ElementLabelProvider` - Die abstrakte Klasse `ElementLabelProvider` definiert die abstrakte Methode `getLabel`. Diese Methode gibt für den `TreePath t` der `TreeModelViewer`-Instanz einen Bezeichner in Form eines `String`-Objekts zurück. Der `TreePath` beschreibt einen Pfad in der Baumstruktur, die der `TreeModelViewer` darstellt. Der Beginn des Pfades ist die Wurzel der Baumstruktur und das Ende des Pfades bildet das Objekt, dessen Bezeichner ermittelt werden soll. Der durch `getLabel` zurückgegebene Bezeichner wird dazu verwendet, das entsprechende Objekt in der Ansicht des `TreeModelViewer` zu repräsentieren.

`ElementContentProvider` - Die abstrakte Klasse `ElementContentProvider` definiert zwei abstrakte Methoden, mit denen

der `TreeModelViewer` die darzustellende Baumstruktur ermittelt. Die abstrakte Methode `getChildCount` gibt die Anzahl der Kinder zurück, die das Objekt `o` in der Baumdarstellung des `TreeModelViewer` besitzt. Die Kinder des Objekts `o` werden mithilfe der Methode `getChildren` ermittelt.

`AbstractModelProxy` - Die Klasse `AbstractModelProxy` definiert ein generisches Modell für die dargestellte Baumstruktur. Das `AbstractModelProxy` implementiert die Funktionalitäten eines Subjekts im Beobachter-Entwurfsmuster (vgl. [55, S. 293–303]). Der `TreeModelViewer` registriert sich über die Methode `addModelChangedListener` beim `AbstractModelProxy` als Beobachter und kann somit auf Änderungen im Modell der dargestellten Baumstruktur reagieren. Um die registrierten Beobachter über Änderungen am Modell zu informieren, stellt das `AbstractModelProxy` die Methode `fireModelChanged` bereit.

Für die Umsetzung der vier Ansichten der Benutzeroberfläche von JHyde werden die drei Adapterklassen, d. h. die beiden `Provider`-Klassen und das `AbstractModelProxy`, erweitert. Für jede Ansicht werden alle drei abstrakten Adapter durch jeweils eine konkrete Adapterklasse erweitert. Für die Berechnungsbaumansicht werden zum Beispiel die Klassen `ComputationTreeLabelProvider`, `ComputationTreeContentProvider` und `ComputationTreeModelProxy` erzeugt. Über die konkreten Implementierungen der abstrakten Adapter kann für jede Ansicht von JHyde exakt gesteuert werden, welche Teile des Programmablaufmodells dargestellt werden und wie diese Teile dargestellt werden.

Der `TreeModelViewer` einer JHyde-Ansicht wird mit den entsprechenden konkreten Adapterinstanzen initialisiert. Um eine Ansicht mit Inhalten zu füllen, wird dem `TreeModelViewer` ein Element des Programmablaufmodells übergeben. Dieses Element ist die Wurzel der Baumstruktur, die der `TreeViewer` darstellen soll. Für jedes darzustellende Element ermittelt der `TreeModelViewer` zunächst die Beschriftung über den entsprechenden `ElementLabelProvider` und die Anzahl der Kinder über den `ElementContentProvider`. Die Kinder eines Elements werden nicht sofort, sondern erst bei Bedarf ermittelt. Für ein Element e wird der Aufruf der `getChildren`-Methode erst ausgeführt, wenn der Teilbaum von e in der Benutzeroberfläche der Ansicht ausgeklappt wird.

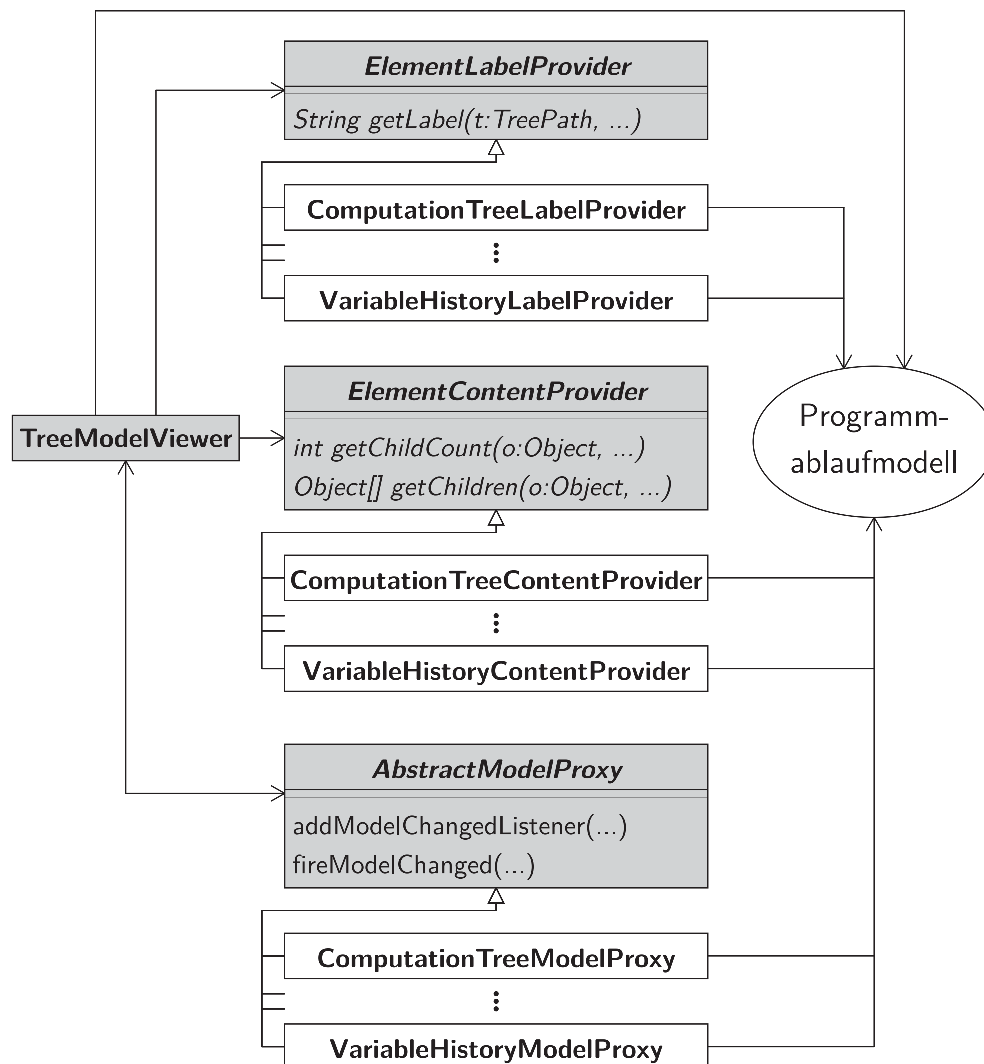


Abbildung 7.17: Klassendiagramm der Benutzeroberfläche.

Während des Debugging-Prozesses kann sich der Zustand des Programmablaufmodells ändern. Dies ist zum Beispiel der Fall, wenn sich die Klassifikation einer Methode ändert. Die Änderungen am Programmablaufmodell werden dem `TreeModelViewer` durch Aufrufe der `fireModelChanged`-Methode der entsprechenden `ModelProxy`-Instanz mitgeteilt.

7.6 Fazit

In diesem Kapitel wurden die wichtigsten Aspekte der Implementierung von JHyde beschrieben. Zuerst wurde die Architektur von JHyde vorgestellt. Die zentralen Eigenschaften der Architektur sind der modulare Aufbau von

JHyde und die Verteilung über zwei separate Java-VMs. Die Trennung von Prüfling und Debugger hat den Vorteil, dass gegenseitige Interferenzen zwischen Prüfling und Debugger minimiert werden.

Das erste vorgestellte Modul von JHyde war der Transmitter. Dieser ist für den Transport der Debugging-Ereignisse von der Prüflings-VM zur Debugger-VM zuständig. Der Transmitter definiert eine Schnittstelle für alle Ereignisse, die während eines Programmablaufs auftreten können. Des Weiteren wandelt er die Ereignisse vor der Übermittlung in ein kompaktes binäres Format. Durch die nebenläufigen Sende- und Empfangsprozesse wird sichergestellt, dass weder die Ausführung des Prüflings noch die Benutzeroberfläche des Debuggers während der Datenübertragung blockieren. Gegenwärtig verwendet der Transmitter eine Socket-Verbindung für die Übermittlung der Ereignisse. Wie gezeigt wurde, lassen sich jedoch leicht weitere Übertragungswege ergänzen.

Der Instrumentierer dient dazu, die Klassen in der Prüflings-VM zu erweitern, so dass diese während der Ausführung Debugging-Ereignisse senden. Im Rahmen der Beschreibung des Transmitters wurde zunächst begründet, warum das ASM-Framework zur Instrumentierung eingesetzt wurde. Anschließend wurde dargelegt, dass die Methoden einiger Klassen in der Prüflings-VM grundsätzlich in zwei unterschiedlichen Modi ausgeführt werden müssen: im Debugging-Modus, wenn sie durch den Prüfling aufgerufen werden, und im normalen Modus, wenn sie durch den Transmitter oder den Instrumentierer aufgerufen werden. Auf der Grundlage dieser Anforderungen wurde ein allgemeines Instrumentierungsschema für alle Klassen in der Prüflings-VM entwickelt. Im Anschluss wurde die Architektur des Instrumentierers beschrieben und schließlich der Instrumentierungsprozess selbst dargestellt.

Der Rekorder erzeugt in der Debugger-VM aus den empfangenen Ereignissen ein Modell des Programmablaufs des Prüflings. Die wichtigsten Eigenschaften des Modells sind die Verteilung auf Arbeitsspeicher und externen Speicher sowie die Erzeugung von Indexstrukturen, mit denen sich Programmzustände effizient rekonstruieren lassen. Neben dem Modell des Programmablaufs wurde auch die Umsetzung der Suchstrategien beschrieben. Da diese Umsetzung dem Strategie-Entwurfsmuster folgt, lässt sich JHyde leicht um neue Strategien erweitern.

Die Benutzeroberfläche von JHyde ist ein Eclipse-Plugin, das aus insgesamt vier Ansichten besteht. Zur Darstellung des Programmablaufmodells kann in weiten Teilen auf die Funktionen der Eclipse-Plattform zurückgegriffen werden. Um es den Komponenten der Benutzeroberfläche von Eclipse zu ermöglichen, das Programmablaufmodell darzustellen, wurden mehrere Adapter implementiert, über die gewünschte Aspekte das Programmablaufmodells für die Darstellung durch die Komponenten des Eclipse-Frameworks adaptiert werden.

Kapitel 8

Stand der Forschung

Inhalt

8.1 Abfragebasiertes Debugging **218**
8.2 Record-Replay-Techniken **221**
8.3 Omniscient-Debugging **227**
8.4 Deklaratives Debugging **229**
8.5 Debugging-Strategien **232**
8.6 Fazit **235**

Der Ansatz für das hybride Debugging von Java-Programmen basiert auf dem Konzept, den Programmablauf des untersuchten Programms aufzuzeichnen. Damit unterscheidet sich JHyde in entscheidender Weise von den sogenannten Online-Debuggern, die die Ausführung des untersuchten Programms unterbrechen, um den gegenwärtigen Programmzustand darzustellen. Die Online-Debugger zeichnen sich durch geringen Ressourcenverbrauch aus, besitzen aber den Nachteil, dass sie immer nur einen Ausschnitt des Programmablaufs darstellen können. Die Suche nach Defekten ist bei Online-Debuggern sehr zeitaufwendig, da der Benutzer die Ausführung nur entlang der Ausführungsrichtung des Programms nachvollziehen kann.

Der Ressourcenbedarf eines Offline-Debuggers ist wesentlich größer, da er den Programmablauf vor Beginn der Defektsuche zunächst aufzeichnet. Der zusätzliche Zeit- und Ressourcenbedarf für die Programmaufzeichnung wird jedoch in Kauf genommen, da mit den gesammelten Informationen die Defektsuche entscheidend beschleunigt werden kann. Die Überzeugung, dass

der Aufwand für das Offline-Debugging gerechtfertigt ist, beruht auf der Beobachtung, dass die Defektsuche eine komplexe und zeitaufwendige Aufgabe ist. Der zusätzliche Aufwand für die Programmaufzeichnung wird durch die Beschleunigung der Defektsuche in der Regel mehr als ausgeglichen.

JHyde basiert ebenfalls auf dem Konzept des Offline-Debuggers. Die Entwicklung von Methoden und Werkzeugen, die den Programmablauf aufzeichnen, um die Defektsuche zu erleichtern, ist seit den späten 60er Jahren Gegenstand der Forschung. Eine der ersten Arbeiten in diesem Bereich ist das 1969 veröffentlichte EXDAMS Projekt [15]. Das System, welches nie vollständig in Betrieb genommen wurde, sollte Teile des Programmzustands aufzeichnen, um diese nach der Programmausführung zu untersuchen und anzuzeigen. Die Möglichkeiten zur Anzeige des Programmzustands waren jedoch aufgrund der ausschließlich textbasierten Darstellung stark eingeschränkt.

In den folgenden Anschnitten werden die Gemeinsamkeiten und Unterschiede zu den wichtigsten Techniken und Methoden aus dem Bereich des Offline-Debugging herausgearbeitet. Zunächst wird JHyde im Abschnitt 8.1 mit den abfragebasierten Debugging-Methoden verglichen. Anschließend wird im Abschnitt 8.2 ein Vergleich zu den Record-Replay-Techniken gezogen. Diese Debugging-Verfahren zeichnen nichtdeterministische Teile des Programmablaufs auf, um bei einer erneuten Ausführung das Programmverhalten möglichst exakt zu rekonstruieren. Die Gemeinsamkeiten, die JHyde zu den Methoden des Omniscient-Debugging aufweist, werden im Abschnitt 8.3 diskutiert. Der Abschnitt 8.4 vergleicht JHyde mit den Methoden und Ansätzen des deklarativen Debugging. Im Abschnitt 8.5 werden die deklarativen Debugging-Strategien diskutiert und zu bereits vorhandenen Strategien in Bezug gesetzt. Dieses Kapitel schließt mit einer Zusammenfassung der Ergebnisse im Abschnitt 8.6

8.1 Abfragebasiertes Debugging

Die abfragebasierten Debugging-Methoden ermöglichen es dem Benutzer, mithilfe von Abfragen nach Ereignissen im Programmablauf zu suchen, die

einen Hinweis auf den gesuchten Defekt geben. Die Abfragen werden entweder vor (*a priori*) oder nach (*a posteriori*) dem Programmablauf des Prüflings in einer durch die entsprechende Methode definierten Abfragesprache formuliert. Im Folgenden werden einige wichtige Methoden und Werkzeuge des abfragebasiertes Debugging kurz vorgestellt. Im Anschluss werden diese Ansätze mit JHyde verglichen.

Hy+/GraphLog - Das von Consens, Hasan und Mendelzon [35] entwickelte System unterstützt die Visualisierung des Ablaufs von verteilten Programmen und ermöglicht es, a posteriori deklarative Abfragen in einer grafischen Abfragesprache zu formulieren.

PQL - Die Program Query Language [106] ist eine abstrakte Abfragesprache, mit der sich a priori Ereignismuster auf Objekten beschreiben lassen. Durch dynamische Programmanalyse können diese Muster während der Ausführung des Programms entdeckt werden. PQL ermöglicht es zudem, Aktionen zu definieren, die beim Auffinden eines Musters ausgeführt werden.

Whyline - Whyline [79, 80] ist ein Debugger für die funktionale Programmierumgebung Alice [36], der es a posteriori ermöglicht, Fragen über den Programmablauf zu stellen. Diese Fragen beziehen sich vor allem auf grafische Ausgaben in der Alice-Umgebung. Die formulierten Fragen lauten zum Beispiel „Warum ist diese Linie rot“ oder „Warum ist der Radius nicht 0.5“. Die Fragen, die der Benutzer über den Programmablauf stellen kann, werden durch Whyline nach der Aufzeichnung des Programmablaufs, u. a. durch den Einsatz von Programm-Slicing-Techniken, automatisch generiert. Im Unterschied zu den anderen Ansätzen muss der Benutzer die Frage nicht selbst formulieren, sondern bekommt durch Whyline die Menge möglicher Fragen vorgegeben.

TQuel - Die Temporal Query Language (TQuel) [134, 135] ist eine Abfragesprache, die explizite Sprachkonstrukte für zeitliche Anfragen unterstützt. Ereignisse über den Programmablauf werden in einer temporalen Datenbank gespeichert. Mithilfe von TQuel können für die Defektsuche a posteriori Abfragen auf den Daten durchgeführt werden.

YODA - YODA [93] ist ein Debugger für die Programmiersprache Ada, der Ereignisse über den Programmablauf als Fakten in einer Prolog-Datenbank speichert. An die Datenbank können im Anschluss mithilfe der Sprache Prolog Anfragen gestellt werden.

Opium - Opium ist ein erweiterbares Debugging-System für die Programmiersprache Prolog [43, 45]. Das System ermöglicht es, Prolog-Abfragen für das Debugging zu stellen und unterstützt zudem Trace-Debugging sowie das Setzten von Breakpoints.

Coca - Der Debugger Coca [44] für die Programmiersprache C ermöglicht es, vor der Programmausführung komplexe Haltebedingungen bzw. Breakpoints mithilfe der Sprache Prolog zu formulieren. Eine Haltebedingung definiert Bedingungen auf den Ereignissen, die während des Programmablaufs aufgezeichnet werden. Wenn diese Bedingungen erfüllt sind, dann wird die Programmausführung wie bei einem Breakpoint unterbrochen. Der Benutzer hat dann die Möglichkeit, den Programmzustand zu inspizieren.

Dynamic Query-based Debugging - Lencevicius, Hölzle und Singh [94] übertragen das Konzept der erweiterten Breakpoint-Bedingungen von Coca in ihrem Ansatz auf die objektorientierte Programmiersprache Java.

JavaDD - Girgis und Jayaraman haben den deklarativen Debugger JavaDD [57] für die Programmiersprache Java entwickelt. JavaDD speichert Ereignisse über den Programmablauf in einer Prolog-basierten Datenbank. Durch Anfragen an die Datenbank können anschließend Fakten über den Programmablauf untersucht werden. Die Autoren beschreiben zudem ein visuelles Interface, das die Erzeugung deklarativer Anfragen erleichtert.

Abgrenzung zu JHyde

Auch JHyde unterstützt die implizite Formulierung von Anfragen an den Programmablauf. Zum Beispiel wird mithilfe der Variablenansicht implizit die Frage beantwortet, zu welchem Zeitpunkt sich der Wert einer Variablen während des Programmablaufs geändert hat. Im Vergleich zu den hier

vorgestellten abfragebasierten Methoden ist die Möglichkeit, Anfragen zu formulieren, bei JHyde auf einige sehr spezielle Fragen beschränkt. Die dedizierten Abfragesprachen sind um ein Vielfaches ausdrucksstärker als die spezialisierten Abfragemöglichkeiten, die über die vier Ansichten von JHyde bereitgestellt werden.

Die abfragebasierten Methoden verfolgen eine grundsätzlich andere Herangehensweise an den Debugging-Prozess. Sie erfordern einen Benutzer, der mit den Interna des Programmablaufs in hohem Maße vertraut ist und in der Lage ist sinnvolle und unter Umständen komplexe Abfragen zu formulieren. Beim Omniscient-Debugging besteht der Prozess der Defektsuche nicht in der Formulierung komplexer Abfragen, sondern in der Navigation des Zustandsraums des untersuchten Programms. Die Philosophien, die diesen beiden Ansätzen zugrunde liegen, sind somit grundsätzlich verschieden.

Die deklarative Debugging-Komponente von JHyde basiert ebenfalls auf der Formulierung von Fragen. Im Vergleich zu den abfragebasierten Methoden sind hier die Rollen jedoch vertauscht, da nicht der Benutzer die Fragen über die Validität von Methodenaufrufen stellt, sondern der Debugger. Auch diese beiden Verfahren verfolgen daher gegensätzlich unterschiedliche Herangehensweisen an die Defektsuche.

Grundsätzlich lässt sich festhalten, dass der hybride Debugging-Prozess von JHyde aus Sicht des Benutzers sehr viel einfacher ist und ein höheres Maß an Automatisierung bietet. Die Formulierung geeigneter Fragen ist eine komplexe Aufgabe, die häufig eine gute Kenntnis über den Programmablauf und auch die Ursache des Fehlers voraussetzt. Wenn dieses Wissen nicht vorhanden ist, dann ist es mitunter sehr schwierig, geeignete Abfragen für die Defektsuche zu formulieren. Bei JHyde ist dieses Vorwissen nicht erforderlich. Hier wird der Ansatz verfolgt, dass sich der Benutzer das nötige Wissen zum Auffinden des Defekts *während* des Debugging-Prozesses durch die Unterstützung von JHyde aneignet.

8.2 Record-Replay-Techniken

Die Record-Replay-Techniken dienen dazu, ein Programm mehrmals hintereinander mit exakt demselben Programmverhalten auszuführen. Für de-

terministische Programme ist dies trivial. Sobald ein Programm allerdings nichtdeterministisches Verhalten aufweist, müssen während der ersten Ausführung des Programms die Ergebnisse nichtdeterministischer Befehle aufgezeichnet werden. Das Programm wird in diesem Fall in einem sogenannten Record-Modus ausgeführt. Bei allen weiteren Ausführungen des Programms können die Aufzeichnungen dazu verwendet werden, dasselbe Verhalten wie bei der ersten Programmausführung zu erzwingen. Hierfür wird das Programm im sogenannten Replay-Modus ausgeführt.

Mithilfe dieser Technik kann ein Programm prinzipiell nicht nur vorwärts, sondern rückwärts ausgeführt werden. Ein rückwärtsgerichteter Sprung zu einem früheren Zeitpunkt t in der Ausführungshistorie eines Programms lässt sich simulieren, indem das Programm im Replay-Modus erneut ausgeführt und die Ausführung zum Zeitpunkt t gestoppt wird.

Die Record-Replay-Debugger stellen einen Kompromiss zwischen den reinen Online-Debuggern und den reinen Offline-Debuggern dar. Statt den Programmablauf gar nicht (Online-Debugger) oder vollständig (Offline-Debugger) aufzuzeichnen, zeichnen sie nur diejenigen Teile des Programmablaufs auf, die benötigt werden, um bei einer erneuten Ausführung des Programms identisches Programmverhalten zu erzeugen.

Das nichtdeterministische Verhalten eines Programms wird im Wesentlichen durch die beiden folgenden externen Einflüsse auf den Programmablauf verursacht: *Eingabebefehle* und *Nebenläufigkeit*. Für jede Klasse von externen Einflüssen gibt es unterschiedliche Methoden und Ansätze das Programmverhalten über mehrere Programmdurchläufe hinweg zu reproduzieren. Im Folgenden werden für jede dieser Klassen unterschiedliche Lösungsansätze vorgestellt.

Eingabebefehle

Betriebssysteme besitzen explizite Befehle, um mit Peripheriegeräten, wie der Maus oder dem Keyboard, zu kommunizieren. Durch diese Befehle werden Daten von den Peripheriegeräten übermittelt, die das Verhalten eines Programms beeinflussen können. Die folgenden Record-Replay-Techniken befassen sich mit der Reproduktion von Eingabebefehlen.

TORNADO - Das TORNADO-System [108] läuft auf einer Linux-Plattform und fängt während der Laufzeit des aufgezeichneten Programms alle Systemaufrufe ab. Für jeden Systemaufruf protokolliert TORNADO alle Änderungen im Speicherbereich des aufgezeichneten Programms. In der Replay-Phase ist TORNADO durch die aufgezeichneten Informationen in der Lage das Verhalten jedes Systemaufrufs exakt zu reproduzieren.

jRapture - Das jRapture-System [139] protokolliert die Interaktionen eines Java-Programms mit der Außenwelt der Java-VM, wie der Benutzerschnittstelle des Betriebssystems, dem Dateisystem und der Tastatur. Zu diesem Zweck manipuliert jRapture die entsprechenden Methoden der Java-API, die mit der Außenwelt der Java-VM kommunizieren.

Nebenläufigkeit

In Betriebssystemen mit präemptivem Multitasking wird durch einen sogenannten Scheduling-Mechanismus gesteuert, wie die Rechenzeit der zur Verfügung stehenden CPUs auf die aktiven Prozesse und Threads verteilt wird. Aus der Sicht eines Prozesses oder Threads ist die Zuteilung der Rechenzeit nichtdeterministisch. Der Ablauf eines nebenläufigen Programms ist somit durch die Beeinflussung des Scheduling-Mechanismus ebenfalls nichtdeterministisch. Um das Verhalten eines nebenläufigen Programms reproduzieren zu können, muss die durch den Scheduling-Mechanismus erzeugte Zuteilung von Rechenzeit reproduziert werden.

Einen Spezialfall von Nebenläufigkeit stellen Interrupts dar. Durch einen Interrupt kann ein Programm durch Unterbrechung des Programmablaufs über ein externes Ereignis informiert werden. In der Regel ändert ein Interrupt den Inhalt des Programmzählers, um den Kontrollfluss des Programms zu einem sogenannten Interrupt-Handler zu transferieren. Das Problem bei der Reproduktion eines Interrupts ist ähnlich wie beim Scheduling von nebenläufigen Programmen. Der Interrupt muss in der Replay-Phase an exakt derselben Position im Programmablauf ausgelöst werden.

Die folgenden Record-Replay-Techniken befassen sich mit Problemen, die in nebenläufigen Programmen auftreten.

Instant Replay - Die Instant Replay Technik [92] kontrolliert alle Zugriffe auf gemeinsam genutzten Speicher (Shared Memory) durch ein sogenanntes Concurrent-Read-Exclusive-Write-Protokoll (CREW-Protokoll). Während der Record-Phase wird für jeden Schreib- und Lesezugriff die Versionsnummer des betroffenen Objekts protokolliert. In der Replay-Phase werden die Versionsnummern benutzt, um die Schreib- und Lesezugriffe solange zu blockieren, bis das Objekt, auf das zugegriffen wird, die richtige Versionsnummer besitzt.

Interrupt Replay - Audenaert und Levrouw [13] beschreiben eine Erweiterung von Instant Replay. Interrupt Replay ermöglicht es, das Auftreten von Interrupts aufzuzeichnen und in der Replay-Phase zu reproduzieren. Um den genauen Zeitpunkt des Auftretens eines Interrupts zu bestimmen, liest Instant Replay den Wert des Programmzählers beim Auftreten eines Interrupts aus. Das Iterrupt Replay System ist für die Programmiersprache Modula-2 [166] umgesetzt worden.

Repeatable Scheduling - Der von Russinovich und Cogswell [128] entwickelte Repeatable Scheduling Algorithmus (RSA) kann die Scheduling-Reihenfolge von Threads auf einem Einprozessorsystem aufzeichnen. Während der Replay-Phase wird den Threads das aufgezeichnete Scheduling aufgezwungen.

DejaVu - Das Deterministic Java Replay Utility [7, 34, 81] ist Teil der von IBM entwickelten Jalapeño-VM [23]. Der Ansatz von DejaVu basiert im Wesentlichen, wie bei Repeatable Scheduling, auf der Protokollierung des Thread-Scheduling. In der Record- und Replay-Phase wird das Thread-Scheduling durch einen logischen Thread-Scheduler übernommen. Durch diese Maßnahme werden alle Java-Threads auf einen einzigen System-Thread abgebildet. Der logische Thread-Scheduler von DejaVu kann auf diese Weise das Scheduling in der Record-Phase protokollieren und in der Replay-Phase vollständig rekonstruieren.

JaRec - JaRec [56] dient dazu, das Verhalten der expliziten Synchronisierungsoperationen in Java aufzuzeichnen und in der Replay-Phase zu rekonstruieren. Das Verhalten eines nebenläufigen Java-Programms kann daher in der Replay-Phase nur dann exakt wiedergegeben werden, wenn das Programm keine Wettlaufsituationen (race conditions) [115] enthält.

RecPlay - RecPlay [127] funktioniert in ähnlicher Weise wie JaRec. Es verfügt allerdings über einen Mechanismus, mit dem Wettlaufsituationen während der Replay-Phase erkannt werden können. Falls eine Wettlaufsituation gefunden wurde, kann der Benutzer diese mithilfe von RecPlay suchen und entfernen. Sobald keine Wettlaufsituationen mehr gefunden werden, wird die Erkennung von Wettlaufsituationen deaktiviert. Dadurch wird die Ausführungsgeschwindigkeit in der Replay-Phase erhöht.

Igor - Das Igor-System [51] speichert den gesamten Zustand des Systems in periodischen Abständen. Die Zeitpunkte, zu denen der Systemzustand gespeichert wird, werden Checkpoints genannt. Abgesehen von den Checkpoints besitzt Igor keine weiteren Mechanismen zur Handhabung von nichtdeterministischem Verhalten. Zwischen zwei Checkpoints kann das Verhalten des Programms in der Replay-Phase somit von dem ursprünglichen Verhalten in der Record-Phase abweichen.

Recap - Recap [123] verfolgt einen ähnlichen Ansatz wie Igor. Neben den periodischen Checkpoints speichert es allerdings auch noch die Auswirkungen von Betriebssystemaufrufen und protokolliert Zugriffe auf den gemeinsamen Speicher. Recap weist daher einen relativ hohen Speicherplatzverbrauch auf. Im Gegenzug ist die Genauigkeit der Replay-Phase wesentlich höher als bei Igor, da Recap die wichtigsten Arten von Nichtdeterminismus handhaben kann.

Netzers Ansatz - Der von NETZER [114] beschriebene Ansatz basiert auf einem Algorithmus, der während des Zugriffs auf den gemeinsamen Speicher mögliche Wettlaufsituationen erkennt. In der Record-Phase müssen dann nur die kritischen Speicherzugriffe, die eine Wettlaufsituation verursachen können, aufgezeichnet werden. Der Vorteil dieser Methode ist der wesentlich geringere Speicherplatzverbrauch.

Abgrenzung zu JHyde

Die vollständige Aufzeichnung eines Programmablaufs verbraucht eine erhebliche Menge an Rechenzeit und Speicherplatz. Die Record-Replay-Techniken adressieren dieses Problem, indem sie Methoden entwickeln, die

es erlauben den Ressourcenbedarf für die Aufzeichnung des Programmablaufs deutlich zu reduzieren. Um den Ressourcenverbrauch bei der Programmaufzeichnung zu senken, wird ein Programm in der Replay-Phase erneut ausgeführt. Aufzeichnungen müssen in der Record-Phase nur an den Stellen gemacht werden, an denen sich das Programm nichtdeterministisch verhält.

Die Record-Replay-Techniken verfolgen das Ziel, einen einmal aufgezeichneten Programmablauf reproduzierbar zu machen. Dieser Ansatz zielt vor allem auf das Debugging von nichtdeterministischen Programmen. Ein aufgezeichnetes Programm kann auf diese Weise in mehreren aufeinanderfolgenden Debugging-Zyklen untersucht werden.

Aufgrund dieser Zielsetzung sind die Record-Replay-Techniken nur bedingt für den Einsatz in JHyde geeignet. Die Record-Replay-Techniken eignen sich zur Durchführung eines Trace-Debugging-Prozesses, indem der Programmablauf schrittweise und vorwärtsgerichtet nachvollzogen wird. Während des hybriden Debugging-Prozesses wird der Zustandsraum des untersuchten Programms nicht ausschließlich in Richtung des Programmablaufs durchsucht. Die Suche entspricht vielmehr einer Folge von vorwärts- und rückwärtsgerichteten Sprüngen durch den Zustandsraum. Für diese Art der Suche sind die Record-Replay-Techniken eher ungeeignet, da sie besonders bei rückwärtsgerichteten Sprüngen den Programmzustand, ausgehend vom Beginn des Programmablaufs, ständig neu berechnen müssten.

Darüber hinaus ist keine der Record-Replay-Techniken in der Lage, sämtliche Formen von nichtdeterministischem Verhalten zu handhaben. Für ein fehlerfreies Debugging-Verfahren müssten jedoch alle Formen des Nichtdeterminismus abgedeckt sein.

Einige Techniken, wie zum Beispiel die Systeme Igor und Recap, enthalten jedoch interessante Ansätze. Der Checkpoint-Mechanismus dieser Systeme beschleunigt Sprünge im Programmablauf. Unter Umständen ließen sich die von diesen Methoden eingesetzten Komprimierungsverfahren zur Speicherung der Checkpoints auch dazu verwenden, den Ressourcenverbrauch von JHyde zu reduzieren.

8.3 Omniscient-Debugging

Das Omniscient-Debugging ist Bestandteil des hybriden Debugging-Prozesses von JHyde. Daher steht JHyde mit den Arbeiten aus diesem Forschungsfeld in engem Zusammenhang. Nachfolgend werden die wichtigsten Beiträge und Entwicklungen im Bereich des Omniscient-Debugging vorgestellt.

ZStep95 - Der Debugger ZStep [31, 97, 159] für die Programmiersprache Lisp [138] beschreibt erste Ansätze und Grundlagen des Omniscient-Debugging. Der Fokus von ZStep liegt auf der grafischen Visualisierung von Aufrufgraphen. ZStep ermöglicht es, das untersuchte Programm schrittweise vorwärts und rückwärts auszuführen und dabei die Änderungen des Aufrufgraphen in Echtzeit zu verfolgen. Darüber hinaus verfügt ZStep jedoch nicht über die Möglichkeit, die Ursache von Infektionen zurückzuverfolgen. Der Schwerpunkt liegt hier eindeutig auf der Visualisierung des Programmablaufs.

ODB - Die Umsetzung des Omniscient-Debugging-Prozesses in JHyde ist inspiriert durch die Arbeiten von Lewis [95, 96]. JHyde bietet im Wesentlichen dieselben Funktionen wie der Omniscient DeBugger (ODB). JHyde verwendet für die Umsetzung des Omniscient-Debugging jedoch zwei separate Java-VMs. Die Trennung zwischen dem eigentlichen Debugger und dem Prüfling ist ein entscheidender Unterschied zwischen beiden Ansätzen. Da beim ODB beide in derselben Java-VM ausgeführt werden, kann es zu unerwünschten Interferenzen zwischen Debugger und Prüfling kommen. Durch die Trennung der beiden Komponenten wird dies verhindert. Zusätzlich lagert JHyde Teile des Programmablaufs in einer Ereignisdatei aus und reduziert damit den Arbeitsspeicherverbrauch des Verfahrens.

Unstuck - Unstuck [67] ist ein Omniscient-Debugger für die Programmiersprache Smalltalk [75]. Die Funktionalitäten sind in weiten Teilen identisch zu denen von ODB.

TOD - Pothier, Tanter und Piquer haben das Konzept von ODB in ihrem Trace-Oriented Debugger (TOD) [126] weiterentwickelt. Der Fokus dieser Arbeit liegt auf der Verbesserung der Skalierbarkeit des Omniscient-Debugging. TOD besitzt zu diesem Zweck einen Mechanismus, der es erlaubt, die Aufzeichnung von Ereignissen für einzelne

Teile des Prüflings zu deaktivieren, um die Gesamtzahl der aufgezeichneten Ereignisse zu reduzieren. Des Weiteren enthält TOD eine speziell für die Anforderungen des Omniscient-Debugging entworfene Datenbank, die es ermöglicht, die Aufzeichnungen des Programmablaufs aus dem Arbeitsspeicher auszulagern.

CodeGuide - Die kommerzielle Entwicklungsumgebung CodeGuide [121] enthält einen erweiterten Trace-Debugger, der während der Ausführung die Ergebnisse der letzten Ausführungsschritte zwischenspeichert. Daher kann der Debugger während der Ausführung einzelne Schritte in der Programmausführung rückgängig machen. Die Anzahl der gespeicherten Ereignisse ist jedoch auf einige Tausend begrenzt. Daher ist die Möglichkeit, die Programmausführung rückwärts gerichtet zu durchsuchen, stark eingeschränkt. Zudem bietet der Debugger keine Möglichkeiten, Fehlerwirkungen gezielt zurückzuverfolgen.

Abgrenzung zu JHyde

Das Omniscient-Debugging ist eine interessante und vielversprechende Erweiterung des Debugging-Prozesses. Es bietet im Vergleich zum Trace-Debugging entscheidende Verbesserungen, die den zeitaufwendigen und komplexen Prozess der Defektsuche entscheidend vereinfachen. Kommerzielle Entwicklungen in diese Richtung zeigen, dass der Ansatz auch für die Praxis interessant ist. Ein entscheidender Nachteil dieser Verfahren ist der enorme Ressourcenverbrauch. In diesem Bereich besteht nach wie vor Forschungsbedarf. Der TOD [126] zeigt in diesem Zusammenhang einige interessante und vielversprechende Ansatzpunkte für weitere Verbesserungsmöglichkeiten.

Im Bezug auf das Omniscient-Debugging weist JHyde eine große Übereinstimmung mit ODB und TOD auf. Im Unterschied zu diesen Ansätzen unterstützt JHyde jedoch zusätzlich das deklarative Debugging von Java-Programmen. Wie in den vorangegangenen Kapiteln beschrieben, haben beide Verfahren unterschiedliche Stärken und Schwächen. Im Vergleich zu den hier beschriebenen Verfahren bietet JHyde vor allem in den Fällen, in denen das Omniscient-Debugging weniger effizient ist, eine Vereinfachung

des Debugging-Prozesses durch das deklarative Debugging an. Der Benutzer hat während der gesamten Defektsuche die Möglichkeit, jeweils diejenige Methode zu wählen, die für die weitere Suche am geeignetsten erscheint.

8.4 Deklaratives Debugging

Deklaratives Debugging, auch bekannt als algorithmisches Debugging, wurde ursprünglich von Shapiro für die logische Programmiersprache Prolog entwickelt [130]. Durch den komplexen Ausführungsmechanismus ist es äußerst schwierig und wenig intuitiv, den Programmablauf einer logischen Programmiersprache schrittweise nachzuvollziehen. Das Debugging einer logischen Programmiersprache mithilfe eines Trace-Debuggers ist daher sehr mühsam und zeitaufwendig. In ähnlicher Weise wie bei den deklarativen Programmiersprachen steht beim deklarativen Debugging die Frage „*Was* wurde berechnet?“ und nicht die Frage „*Wie* wurde es berechnet?“ im Mittelpunkt. Das deklarative Debugging ist für viele unterschiedliche Programmierparadigmen [160] umgesetzt worden.

Logisch - Wie bereits erwähnt, wurde das deklarative Debugging ursprünglich für Prolog entwickelt [130]. Darüber hinaus existiert zum Beispiel für NU-Prolog, eine Erweiterung von Prolog, das NU-Prolog Debugging Environment (Nude) [113].

Funktional - Nielsson und Fritzon [119] haben eine deklarative Debugging-Methode für funktionale Sprachen entwickelt. Diese wurde später durch Nielsson [117, 118] weiterentwickelt. Für die Programmiersprache Haskell [73] gibt es unter anderem die deklarativen Debugger Budda [124, 125] und Hat-Delta [41].

Funktional-logisch - Erste Ansätze zur Erweiterung des deklarativen Debugging auf funktional-logische Programmiersprachen kommen von Naish [112]. Diese Ansätze wurden später durch Caballero und Rodríguez-Artalejo erweitert [26]. Darüber hinaus hat MacLarty [103] einen deklarativen Debugger für die funktional-logische Sprache Mercury [136] entwickelt.

CLP - Die Constraintprogrammierung (Constraint Logic Programming) [70] ist eine Erweiterung des logischen Programmierparadigmas um

Constraints. Für das CLP-Paradigma haben unter anderem Tessier [154] und später Tessier und Ferrand [155] deklarative Debugging-Techniken entwickelt.

CFLP - CFLP (Constraint Functional-Logic Programming) verbindet Elemente Constraint-basierter, funktionaler und logischer Programmierung. Für die CFLP-Sprache TOY [11, 101] hat Caballero [24] einen deklarativen Debugger entwickelt. Die Funktionalitäten wurde anschließend in Zusammenarbeit mit Rodríguez-Artalejo und Del Vado Vírseda [27, 28] erweitert.

Imperativ - Auch außerhalb der Domäne deklarativer Programmiersprachen sind Methoden für deklaratives Debugging entwickelt worden. Shahmehri und Fritzson [129] haben deklaratives Debugging erstmalig auf die imperative Programmiersprache Pascal [165] angewendet. Dieser Ansatz wurde später in Zusammenarbeit mit Kamkar und Gyimothy erweitert [54].

Objektorientiert - Im Bereich der objektorientierten Programmiersprachen haben Kouh und Yoo eine Methode für das deklarative Debugging von Java-Programmen [85, 86] entworfen. Auf Grundlage dieses Ansatzes wurde später der deklarative Debugger HDTS [83, 84] entwickelt. HDTS ist eine Kombination aus deklarativem Debugging und Trace-Debugging. Der Debugger verwendet zudem Slicing-Techniken [82], um den Berechnungsbaum während des deklarativen Debugging-Prozesses zu beschneiden und so die Anzahl der zu klassifizierenden Methodenaufrufe zu reduzieren. Um das Slicing durchführen zu können, muss der Benutzer diejenige Variable auswählen, deren Wert infiziert ist. HDTS entfernt daraufhin alle Teile des Berechnungsbaums, die den Wert der Variablen nicht beeinflusst haben.

Abgrenzung zu JHyde

Die Untersuchung zeigt, dass das deklarative Debugging für eine Vielzahl von Sprachen und Paradigmen umgesetzt wurde. Der Schwerpunkt der Anwendung liegt eindeutig im Bereich der deklarativen Programmiersprachen, aber auch für imperative und objektorientierte Sprachen gibt es interessante Umsetzungen dieser Technik.

Die deklarative Debugging-Methode von JHyde grenzt sich am deutlichsten von den Ansätzen im Bereich der deklarativen Sprachen ab. Im Unterschied zu den deklarativen Sprachen besitzt Java Seiteneffekte. Für die Korrektheit des Verfahrens ist es daher wichtig, dass der Benutzer während des Debugging-Prozesses über diese Seiteneffekte in Kenntnis gesetzt wird. Beim Debugging von deklarativen Sprachen tritt diese Problematik nicht auf, da diese in der Regel keine Seiteneffekte besitzen.

Im Vergleich zur deklarativen Debugging-Methode von Shahmehri und Fritzson liegt der größte Unterschied in der Programmiersprache. Java ist eine objektorientierte Programmiersprache und wesentlich komplexer als die von Shahmehri und Fritzson adressierte imperative Sprache Pascal.

Im Bezug auf das deklarative Debugging besitzt JHyde die meisten Gemeinsamkeiten mit HDTS, welches ebenfalls die Sprache Java adressiert. Beim genaueren Vergleich der beiden Methoden ergeben sich jedoch folgende entscheidende Unterschiede:

- Um die exakte Position eines Defekts innerhalb eines Methodenaufrufs zu identifizieren, verwendet HDTS eine Art Trace-Debugging, während JHyde zu diesem Zweck das Omniscient-Debugging benutzt. Wie die vorangegangenen Untersuchungen gezeigt haben, ist das Omniscient-Debugging im Bezug auf die Defektsuche wesentlich effizienter als das Trace-Debugging.

- Zur Beschleunigung des deklarativen Debugging setzt HDTS Slicing-Verfahren ein. JHyde benutzt hierfür eine verbesserte D&Q-Strategie, die die Infektionswahrscheinlichkeiten von Methodenaufrufen auf der Grundlage von vorangegangenen Klassifikationen und Überdeckungsinformationen bestimmt. In der Tat ist die Verwendung von Slicing-Verfahren ein interessanter und vielversprechender Ansatz. Es besteht jedoch die Gefahr, dass gerade diejenigen Teile des Berechnungsbaums weggeschnitten werden, die den defekten Methodenaufruf enthalten. Dies ist der Fall, wenn eine Infektion nicht durch die Zuweisung eines *falschen* Wertes verursacht wurde, sondern durch eine *fehlende* Wertzuweisung. Es besteht dann die Möglichkeit, dass die Stelle, an der die Wertzuweisung *hätte* stattfinden müssen, aus dem Berechnungsbaum entfernt wird. Dies ist möglich, da sie dem Slicing zufolge

keinen Einfluss auf den infizierten Wert hat. Die Methode von JHyde garantiert in jedem Fall die Korrektheit des Verfahrens, da zur Effizienzsteigerung ausschließlich die Reihenfolge der Klassifikationen optimiert wird.

8.5 Debugging-Strategien

JHyde unterstützt für das deklarative Debugging im Wesentlichen drei Debugging-Strategien.

Top-Down - Die Top-Down Strategie von JHyde (vgl. Abschnitt 5.1) wurde ursprünglich von Av-Ron [14] entwickelt. Für diese Strategie wurden unterschiedliche Modifikationen entwickelt. Maeji und Kanamori [105] schlagen vor, bei der Top-Down-Suche Kindknoten zu bevorzugen, die rekursive Aufrufe repräsentieren, da diese einen engen semantischen Bezug haben. Der Benutzer kann eine Folge rekursiver Kindknoten daher schneller klassifizieren. MacLarty [104] empfiehlt diejenigen Kinder zuerst zu klassifizieren, die eine Exception geworfen haben, da diese eine höhere Infektionswahrscheinlichkeit aufweisen. In ähnlicher Weise schlägt Brinks [20] vor, immer den schwersten Kindknoten zuerst zu klassifizieren, da dieser die höchste Infektionswahrscheinlichkeit besitzt. Dieser Ansatz wurde von Silva [132] erweitert, indem er bei der Gewichtsberechnung nicht die Knoten eines Teilbaumes, sondern die *verschiedenen* aufgerufenen Funktionen bzw. Methoden zählt. Diese Strategie wählt immer den Kindknoten, in dessen Teilbaum die meisten unterschiedlichen Funktionen aufgerufen wurden. Unter den Voraussetzungen, dass alle Funktionen die gleiche Defektwahrscheinlichkeit besitzen und eine defekte Funktion immer eine Infektion produziert, stellt diese Strategie eine Verbesserung gegenüber der von Brinks vorgeschlagenen Strategie dar. Die von Davie und Chitil vorgeschlagene Erweiterung der Top-Down-Strategie [41] berechnet Infektionswahrscheinlichkeiten für Kindknoten mithilfe vorangegangener Klassifikationen. Die Infektionswahrscheinlichkeit einer Funktion steigt, je öfter diese als infiziert klassifiziert wurde und sie sinkt, je öfter sie als valide klassifiziert wurde. Die Strategie wählt den Kindknoten mit der höchsten Infektionswahrscheinlichkeit.

Divide-and-Query - Zusammen mit der Entwicklung des deklarativen Debugging hat Shapiro die D&Q-Strategie beschrieben [130]. Diese Strategie wird auch von JHyde umgesetzt.

Erweitertes Divide-and-Query - Für die D&Q-Strategie sind mehrere Verbesserungen entwickelt worden. Silva hat seinen Ansatz, bei der Gewichtsberechnung nur die unterschiedlichen aufgerufenen Funktionen eines Teilbaumes zu zählen, auch auf das D&Q-Verfahren angewandt [132]. MacLarty [104] schlägt vor, bei der Gewichtsberechnung die Typen der Knoten eines Teilbaumes zu berücksichtigen. Er hat 13 Typen von Knoten identifiziert, denen er unterschiedliche Einflüsse auf das Gewicht eines Teilbaumes zugeordnet hat. Auf diese Weise können den Typen unterschiedliche Infektionswahrscheinlichkeiten zugeordnet werden. Weitere Verbesserungsvorschläge von MacLarty [104] und Silva [132] basieren auf einer zusätzlichen Klassifikationsmöglichkeit der Knoten. Ein Knoten kann für diese Strategien zusätzlich als „unzulässig" (inadmissible) klassifiziert werden. Dies bedeutet, dass der Aufruf des Knotens nicht zulässig ist bzw. nicht dem intendierten Verhalten des Programms entspricht. Hierdurch erhält der Debugger die Information, dass der gesuchte Defekt im Programmablauf *vor* dem unzulässigen Knoten aufgetreten sein muss. Ohne diese Klassifikationsmöglichkeit würde ein entsprechender Knoten als valide klassifiziert. Durch diese weitere Klassifikationsmöglichkeit kann der Suchbereich somit zusätzlich eingeschränkt werden. Die Erweiterungen von MacLarty und Silva ermöglichen es darüber hinaus, auch einen bestimmten Parameter einer Funktion als unzulässig zu klassifizieren. Mithilfe von Slicing-Techniken kann der Debugger dann diejenigen Teile des Programmablaufs identifizieren, die den unzulässigen Parameter verursacht haben. Die so gewonnenen Informationen verwendet MacLarty, um den Suchbereich weiter einzuschränken, und Silva, um die Klassifikationsreihenfolge zu optimieren.

Abgrenzung zu JHyde

Wie bereits erwähnt, sind die in JHyde umgesetzten Strategien Top-Down und D&Q direkte Umsetzungen der von Av-Ron bzw. Shapiro entwickelten Methoden. Die auf gewichtsunabhängigen Infektionswahrscheinlichkei-

ten basierende erweiterte D&Q-Strategie (vgl. Abschnitt 5.3) unterscheidet sich in den folgenden Punkten von den hier beschriebenen Strategien:

Überdeckungsinformationen - Die Strategie von JHyde besitzt einen Mechanismus, bei dem die Knoten unterschiedliche Beiträge zur Infektionswahrscheinlichkeit eines Teilbaumes liefern. Hier weist die JHyde-Strategie Ähnlichkeiten mit dem Ansatz von MacLarty auf, indem 13 Typen von Knoten mit unterschiedlichen Gewichten festgelegt werden. Bei der JHyde-Strategie werden die Beiträge zur Infektionswahrscheinlichkeit allerdings nicht a priori festgelegt. Sie werden vielmehr dynamisch aufgrund der aufgezeichneten Überdeckungsinformationen ermittelt. Dieser Ansatz ist sehr viel flexibler und spiegelt die tatsächliche Komplexität und damit die Infektionswahrscheinlichkeit der Knoten genauer wieder.

Vorangegangene Klassifikationen - Zusätzlich berücksichtigt die JHyde-Strategie auch die Historie der klassifizierten Knoten. Dies geschieht in Verbindung mit den aufgezeichneten Überdeckungsinformationen. Je öfter eine Überdeckungsentität als valide klassifiziert wurde, desto geringer ist ihr Beitrag zur Infektionswahrscheinlichkeit. Die Berücksichtigung vorangegangener Klassifikationen findet sich auch in der Strategie von Davie und Chitil [41]. Allerdings hat hier eine Klassifikation auf alle Knoten, die einen Aufruf derselben Funktion bzw. Methode repräsentieren, identischen Einfluss. Die JHyde-Strategie ist an dieser Stelle präziser, da die Änderung der Infektionswahrscheinlichkeiten nicht auf Grundlage der aufgerufenen Methode, sondern auf Grundlage der durch die aufgerufene Methode erzeugten Überdeckung berechnet wird. In abgewandelter Form berücksichtigt auch die Strategie von Silva [132] vorangegangene Klassifikationen bei der Gewichtsberechnung. Wenn ein Knoten als unzulässig klassifiziert wird, dann werden die Gewichte der Knoten erhöht, die den unzulässigen Aufruf bzw. den unzulässigen Parameter verursacht haben können. Die Möglichkeit, einen Knoten als unzulässig zu klassifizieren, bietet JHyde zurzeit nicht. Dieses Verfahren bietet daher einen interessanten Ansatzpunkt, die JHyde-Strategie zu erweitern.

Minimierung des Erwartungswertes - JHyde verbindet die Informationen über das Gewicht und die Infektionswahrscheinlichkeit, um für

jeden Knoten einen Erwartungswert zu berechnen. Der Erwartungswert eines Knotens k entspricht der erwarteten verbleibenden Größe des Suchraumes für den Fall, dass k klassifiziert wird. Dem Ablauf eines Greedy-Algorithmus folgend wird immer der Knoten mit dem geringsten Erwartungswert zuerst klassifiziert. Der Ansatz, Gewicht und Infektionswahrscheinlichkeit getrennt voneinander zu bestimmen und zu einem Erwartungswert zu verdichten, ist neu. Die anderen hier vorgestellten Verfahren, die ebenfalls auf Infektionswahrscheinlichkeiten basieren, wählen immer den Knoten mit der größten Infektionswahrscheinlichkeit. Der Unterschied dieser Ansätze zeigt sich am Beispiel zweier Knoten k_1 und k_2, die dieselbe Infektionswahrscheinlichkeit p aber unterschiedliche Gewichte g_1 und g_2 besitzen. Wenn die Auswahl ausschließlich auf Grundlage der Infektionswahrscheinlichkeit durchgeführt wird, gibt es keine Präferenz für k_1 oder k_2. Da die Strategie von JHyde zusätzlich die Gewichte der Knoten berücksichtigt, wird für $p > 0,5$ der leichtere der beiden Knoten und für $p < 0,5$ der schwerere der beiden Knoten klassifiziert. Nur für den Fall, dass $g_1 = g_2$ gilt, gibt es keine Präferenz für einen der Knoten. Im Bezug auf die erwartete Größe des verbleibenden Suchraums ist dieser Strategie somit den anderen Verfahren überlegen.

8.6 Fazit

In diesem Kapitel wurden die für JHyde entwickelten und implementierten Methoden mit anderen Offline-Debugging-Techniken verglichen. Zunächst wurde die neu entwickelte hybride Debugging-Methode von den abfragebasierten Debugging-Techniken und den Record-Replay-Techniken abgegrenzt. Im Anschluss wurden die in JHyde kombinierten Debugging-Methoden, das Omniscient-Debugging und das deklarative Debugging, näher betrachtet. Für beide Methoden wurden Gemeinsamkeiten und Unterschiede zu bestehenden Ansätzen und Verfahren herausgearbeitet. Beim Vergleich zu den vorhandenen Methoden ergeben sich folgende wesentliche Unterschiede und Neuerungen:

- Der Entwurf und die Umsetzung eines hybriden Debugging-Prozesses für Java-Programme. Insbesondere die Kombination von deklarativem Debugging und Omniscient-Debugging.
- Die Entwicklung einer neuen Debugging-Strategie, die Gewichte *und* Infektionswahrscheinlichkeiten berücksichtigt und zu einem Erwartungswert verdichtet.
- Die Berechnung der Infektionswahrscheinlichkeiten auf der Grundlage von vorangegangenen Klassifikationen in Verbindung mit Überdeckungsinformationen. Die Verwendung von Überdeckungsinformationen zur Schätzung von Infektionswahrscheinlichkeiten stellt dabei eine besondere Neuerung dar.

Kapitel 9

Schlussbetrachtungen

Inhalt

9.1 Zusammenfassung 237
9.2 Ausblick 240

Dieses Kapitel bildet den Abschluss der vorliegenden Dissertation. Im Abschnitt 9.1 werden die Resultate der vorangegangenen Kapitel zusammengefasst und die wesentlichen Ergebnisse dieser Arbeit dargestellt. Der Abschnitt 9.2 beschreibt, aufbauend auf diesen Ergebnissen, Ansatzpunkte für weitere Arbeiten und weitere Forschung.

9.1 Zusammenfassung

Im ersten Kapitel wurden das Thema, die Zielsetzung und der Aufbau dieser Arbeit formuliert. Zunächst wurde festgestellt, dass das Debugging von Java-Programmen ein komplexer und arbeitsintensiver Prozess ist. Seit der Veröffentlichung der Programmiersprache Java im Jahr 1995 wird das Debugging von Java-Programmen im Wesentlichen mit der Hilfe von Trace-Debuggern durchgeführt. Im Bereich der Forschung wurden in der Zwischenzeit viele Debugging-Techniken entwickelt, deren Funktionalitäten weit über die eines Trace-Debuggers hinaus gehen. Ausgehend von dieser Lücke zwischen Praxis und Forschung wurde die Zielsetzung dieser Arbeit formuliert. Es sollte ein Debugger für Java entwickelt werden, der durch den Einsatz

neuerer Debugging-Methoden den Debugging-Prozess vereinfacht und beschleunigt. Der Debugger sollte deklaratives Debugging und Omniscient-Debugging in einem einzigen Werkzeug vereinen.

Im zweiten Kapitel wurden die für diese Arbeit relevanten Grundlagen der Java-Technik beschrieben. Zuerst wurden die wichtigsten Eigenschaften der Programmiersprache Java vorgestellt und im Anschluss die wichtigsten Strukturen der Java-VM erläutert.

Im dritten Kapitel wurde der Debugging-Prozess dargestellt. Neben einer Erläuterung der Wirkungskette von Defekt, Infektion und Fehlerwirkung wurden die wichtigsten Infektionstypen, die während eines Programmablaufs auftreten können, identifiziert. Anschließend wurde die Zielsetzung des Debugging-Prozesses aus theoretischer Sicht dargestellt. Für die erfolgreiche Identifikation eines Defekts muss auf der Grundlage des Programmablaufs ausgehend von der Fehlerwirkung über die infizierten Programmzustände auf den verursachenden Defekt geschlossen werden. Im Bezug auf den Programmablauf ist dieser Prozess rückwärtsgerichtet.

Im vierten Kapitel wurden zunächst drei unterschiedliche Debugging-Techniken vorgestellt: Trace-Debugging, Omniscient-Debugging und deklaratives Debugging. Für jede dieser Techniken wurde untersucht, inwiefern diese für die Lösung der im dritten Kapitel beschriebenen Problemstellung geeignet ist. Es hat sich gezeigt, dass die Eignung der Verfahren stark durch die Art der verursachten Infektionen beeinflusst wird. Während das Trace-Debugging den übrigen Verfahren in allen wesentlichen Punkten unterlegen ist, ergibt sich aus dem Vergleich von Omniscient-Debugging und deklarativem Debugging kein eindeutiger Gewinner. Diese Erkenntnisse haben zur Entwicklung der hybriden Debugging-Methode geführt, die die Vorteile des Omniscient-Debugging und des deklarativen Debugging in einem einzigen Verfahren vereint.

Im fünften Kapitel wurden die deklarativen Debugging-Strategien beschrieben, die für den deklarativen Debugging-Prozess von JHyde umgesetzt wurden. Top-Down und D&Q zählen zu den bekanntesten und verbreitetsten Strategien. Eine Neuerung stellt die im Rahmen dieser Arbeit entwickelte Erweiterung der D&Q-Strategie dar. Diese berücksichtigt vorangegangene Klassifikationen, Knotengewichte und Infektionswahrscheinlichkeiten von Knoten. Aus diesen Informationen berechnet sie Erwartungswerte für

die Knoten des Suchraums. Der Erwartungswert eines Knotens k entspricht der erwarteten Größe des verbleibenden Suchraums für den Fall, dass k klassifiziert wird. Die Berechnung der Infektionswahrscheinlichkeiten eines Knotens geschieht auf Grundlage von Überdeckungsinformationen, die während der Aufzeichnung des Prüflings gesammelt werden. Die Verwendung von Überdeckungsinformationen zur Messung der Komplexität bzw. der Infektionswahrscheinlichkeit eines Knotens unterscheidet dieses Verfahren in besonderer Weise von anderen Ansätzen zur Bestimmung der Infektionswahrscheinlichkeit eines Knotens.

Um die Leistungsfähigkeit der verwendeten Verfahren zu vergleichen, wurde eine empirische Studie durchgeführt. Jede der Strategien wurde in insgesamt 32 Testfällen zur Defektsuche eingesetzt. Im Vergleich zur Top-Down-Strategie konnten mit der einfachen D&Q-Strategie 22 Prozent und mit der erweiterten D&Q-Strategie bis zu 40 Prozent der Klassifikationen eingespart werden. Durch die Erweiterungen konnten die Einsparungen der D&Q-Strategie somit fast verdoppelt werden.

Im sechsten Kapitel wurde JHyde, ein Debugger, der das hybride Debugging von Java-Programmen ermöglicht, vorgestellt. Zunächst wurde die Benutzeroberfläche von JHyde, insbesondere der Aufbau und die Funktionsweise der unterschiedlichen Ansichten, erläutert. Im Anschluss wurde der Debugging-Prozess mit JHyde am Beispiel eines defekten Java-Programms durchgeführt. Dabei wurde sowohl das deklarative Debugging als auch das Omniscient-Debugging verwendet, um den Defekt zu lokalisieren. Anhand dieses Beispiels wurde zudem gezeigt, dass keine der beiden Debugging-Methoden der anderen in allen Fällen überlegen ist. Die Eignung der Verfahren ist von der Art der Infektion abhängig.

Im siebten Kapitel wurden wichtige Aspekte des Entwurfs und der Implementierung von JHyde erläutert. Zuerst wurde ein Überblick über die Architektur von JHyde gegeben. Anschließend wurden der Aufbau und die Funktionsweise der vier Funktionseinheiten von JHyde erläutert. Der Transmitter ist für die Übermittlung der Ereignisse von der Prüflings-VM zur Debugger-VM verantwortlich. Der Instrumentierer instrumentiert alle Klassen, die in die Prüflings-VM geladen werden. Die instrumentierten Methoden übermitteln während der Ausführung Ereignisse über den Programmablauf an den Transmitter. In der Debugger-VM nimmt der Rekorder diese Ereignisse in Empfang und speichert sie in einem Modell, welches den Programmablauf

des Prüflings repräsentiert. Die Benutzeroberfläche präsentiert das aufgezeichnete Modell in vier Ansichten dem Benutzer und ermöglicht es, den hybriden Debugging-Prozess durchzuführen.

Im achten Kapitel wurden die theoretischen und praktischen Ergebnisse dieser Dissertation mit anderen Arbeiten aus verwandten Forschungsgebieten verglichen. Zu den betrachteten Gebieten zählen abfragebasiertes Debugging, Record-Replay-Techniken, Omniscient-Debugging, deklaratives Debugging und deklarative Debugging-Strategien. Diese Arbeit wurde von den bestehenden Methoden und Verfahren aus diesen Gebieten abgegrenzt. Zudem wurden in diesem Zusammenhang die Neuerungen und der Forschungsbeitrag dieser Arbeit herausgearbeitet.

9.2 Ausblick

Die im Rahmen dieser Arbeit entwickelten Methoden und Verfahren für das hybride Debugging von Java-Programmen bieten interessante und vielversprechende Ansatzpunkte für Verbesserungen und Erweiterungen. Im Folgenden werden die wichtigsten Erweiterungsmöglichkeiten kurz beschrieben.

Threads - Zum gegenwärtigen Zeitpunkt ist das Debugging nebenläufiger Programme mit JHyde nicht möglich. Besteht ein Programmablauf nicht nur aus einem, sondern aus mehreren parallel ausgeführten Threads so erhöht dies die Komplexität des Debugging-Prozesses in der Regel erheblich. Die Komplexitätssteigerung ergibt sich, wenn mehrere Threads auf gemeinsamen Speicher zugreifen und dadurch ihr Verhalten gegenseitig beeinflussen. Die Ursache einer Fehlerwirkung ist sehr viel schwieriger zu ermitteln, wenn sich die durch einen Defekt verursachte Infektion über den gemeinsam genutzten Speicher vom verursachenden Thread auf einen anderen Thread überträgt. In der Programmausführung muss die Wirkungskette vom Defekt über die infizierten Zustände zur Fehlerwirkung dann über mehrere ineinander verzahnte Ausführungsstränge zurückverfolgt werden. Das Omniscient-Debugging lässt sich vergleichsweise einfach auf nebenläufige Programme erweitern. Problematisch ist hingegen jedoch die Erweiterung des deklarativen Debugging. Wenn bei einer nebenläufigen

Programmausführung der Programmzustand während eines Methodenaufrufs c von einem validen in einen infizierten Zustand überführt wurde, dann ist es schwierig zu beurteilen, ob der Methodenaufruf c tatsächlich infiziert ist. Es besteht nämlich die Möglichkeit, dass die Infektion gar nicht durch den Methodenaufruf c, sondern einen anderen, parallel ausgeführten Methodenaufruf eines weiteren Threads verursacht wurde. Um den Benutzer bei der Klassifikation der Knoten nicht zu überfordern, müssten geeignete Hilfsmittel entwickelt werden, mit denen sich die Wechselwirkungen zwischen den Threads in geeigneter Form darstellen lassen.

Ressourcenbedarf - Der enorme Ressourcenbedarf der Offline-Debugger ist mitunter ein entscheidender Grund, der dazu geführt hat, dass diese Werkzeuge in der praktischen Softwareentwicklung bisher wenig Verbreitung gefunden haben. Obwohl JHyde bereits Teile des aufgezeichneten Programmablaufs in einer Datei auf einem externen Speicher auslagert, ist der Arbeitsspeicherverbrauch immer noch hoch. Wie im Abschnitt 7.4.1 bereits angedeutet, ließen sich weitere Teile des aufgezeichneten Programmablaufs auslagern. Ansätze hierfür können im Bereich der Record-Replay-Techniken und bei dem von Pothier, Tanter und Piquer entwickelten TOD [126] gefunden werden.

Empirische Evaluation - Die Offline-Debugger bieten im Vergleich zu einem Trace-Debugger einen klaren Mehrwert, da sie den Benutzer durch zusätzliche Funktionen und Informationen über den Programmablauf bei der Defektsuche unterstützen. Aufgrund dieser Tatsache ist es unmittelbar einsichtig und plausibel, dass die Defektsuche durch den Einsatz eines Offline-Debuggers beschleunigt werden kann. Dennoch gibt es keine ausführlichen empirischen Untersuchungen, die den durch die Offline-Debugger generierten Mehrwert quantifizieren. Eine fundierte empirische Untersuchung, die die tatsächliche Beschleunigung des Debugging-Prozesses misst, könnte die Verbreitung dieser Techniken in der praktischen Softwareentwicklung begünstigen. Es ist jedoch zu beachten, dass eine solche Untersuchung nicht unproblematisch ist, da die meisten Programmierer bereits umfassende Kenntnisse im Umgang mit Trace-Debuggern haben, während vor allem der deklarative Debugging-Prozess gerade im Bereich der objektorientierten Programmierung relativ unbekannt ist.

Integration von Slicing - Ein Defekt verursacht eine Ursache-Wirkungskette, die zu einer Fehlerwirkung führt. Die Zielsetzung des Debugging-Prozesses ist es, diese Ursache-Wirkungskette ausgehend von der Fehlerwirkung bis zum Defekt zurückzuverfolgen. Slicing-Techniken dienen dazu, Ursache-Wirkungsketten in Programmabläufen zu identifizieren. Sie lassen sich daher sowohl beim deklarativen Debugging als auch beim Omniscient-Debugging einsetzen, um die Defektsuche zu beschleunigen. Beim deklarativen Debugging können Slicing-Techniken, wie von Silva [132] vorgeschlagen, dazu dienen, die geschätzten Infektionswahrscheinlichkeiten der Knoten zu erhöhen, die Teil einer identifizierten Ursache-Wirkungskette sind. Beim Omniscient-Debugging kann die Hervorhebung von Ursache-Wirkungsketten die rückwärtsgerichtete Defektsuche erleichtern und beschleunigen. Das Slicing bietet daher interessante Verbesserungsansätze für beide Teile des hybriden Debugging-Prozesses.

Kombination mit Testverfahren - Auch die Integration von JHyde mit Testtools bietet vielversprechende Verbesserungsmöglichkeiten. Zum Beispiel könnten die aus Testfällen gewonnenen Informationen dazu verwendet werden, die Infektionswahrscheinlichkeiten der Knoten anzupassen. Erfolgreich getestete Methoden könnten bereits vor dem Beginn der Defektsuche mit einer geringeren Infektionswahrscheinlichkeit bewertet werden. Durch diesen Ansatz ließe sich vor allem das deklarative Debugging optimieren.

Andere Programmiersprachen - JHyde zeichnet den in der Prüflings-VM ausgeführten Bytecode auf. Neben der Programmiersprache Java existieren viele weitere Programmiersprachen, wie Clojure [63], Jython [74] und Scala [161], die zur Ausführung ebenfalls in Java-Bytecode übersetzt werden. Daher lassen sich auch diese Sprachen prinzipiell mithilfe von JHyde debuggen. Es wäre interessant zu untersuchen, welche Anpassungen von JHyde vorzunehmen wären und inwieweit der Debugging-Prozess für diese Sprachen durch den Einsatz von JHyde verbessert werden könnte.

Literaturverzeichnis

[1] Adl-Tabatabai, Ali-Reza ; Cierniak, Michał; Lueh, Guei-Yuan ; Parikh, Vishesh M. ; Stichnoth, James M.: Fast, effective code generation in a just-in-time Java compiler. In: *ACM SIGPLAN Notices* 33 (1998), Nr. 5, S. 280–290 (Zitiert auf den Seiten 11 und 23)

[2] Adobe Systems: *PostScript language reference*. 3. Addison-Wesley, 1999 (Zitiert auf Seite 18)

[3] Agrawal, Hiralal ; Horgan, Joseph R.: *Dynamic Program Slicing*. 1990 (Zitiert auf Seite 45)

[4] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2007 (Zitiert auf Seite 16)

[5] Aho, Alfred V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986 (Zitiert auf den Seiten 101 und 102)

[6] Alpern, Bowen ; Cocchi, Anthony ; Fink, Stephen ; Grove, David ; Lieber, Derek: Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In: *In Proc. 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2001, S. 108–124 (Zitiert auf Seite 30)

[7] Alpern, Bowen ; Ngo, Ton ; Choi, Jong-Deok ; Sridharan, Manu: DejaVu: deterministic Java replay debugger for Jalape no Java virtual machine. In: *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, ACM, 2000. – ISBN 1–58113–307–3, S. 165–166 (Zitiert auf Seite 224)

[8] Apache Jakarta: *Byte Code Engineering Library*. `http://jakarta.apache.org/bcel/`, 2010 (Zitiert auf Seite 173)

[9] Apache Software Foundation: *Apache log4cxx*. `http://logging.apache.org/log4cxx/index.html`, 2010 (Zitiert auf Seite 47)

[10] Apache Software Foundation: *Apache log4j*. `http://logging.apache.org/log4j/`, 2010 (Zitiert auf Seite 169)

[11] Arenas, Puricación ; Fernández, Antonio ; Gil, Ana ; López, Francisco.J. ; Rodríguez, Mario ; Sáenz, Fernando: TOY: A Multiparadigm Declarative Language, Version 2.2.3, July 2006 / UCM. 2006 (TR-SIP). – Forschungsbericht (Zitiert auf Seite 230)

[12] Arnold, Matthew ; Fink, Stephen J. ; Grove, David ; Hind, Michael ; Sweeney, Peter F.: A Survey of Adaptive Optimization in Virtual Machines. In: *Special issue on Program Generation, Optimization, and Adaptation* Bd. 93, 2005 (Proceedings of the IEEE 2), S. 449–466 (Zitiert auf Seite 23)

[13] Audenaert, Koenraad M. R. ; Levrouw, Luk J.: Interrupt replay: a debugging method for parallel programs with interrupts. In: *Microprocessors and Microsystems* 18 (1994), Nr. 10, S. 601–612 (Zitiert auf Seite 224)

[14] Av-Ron, Evyatar: Top-down diagnosis of Prolog programs / Weizmann Institute Department of Computer Science and Applied Mathematics. 1984. – Forschungsbericht (Zitiert auf den Seiten 85 und 232)

[15] Balzer, Robert M.: EXDAMS: extendable debugging and monitoring system. In: *AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, ACM, 1969, S. 567–580 (Zitiert auf Seite 218)

[16] Balzert, Helmut: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. 2. Spektrum Akademischer Verlag, 1998 (Zitiert auf den Seiten 96, 98 und 102)

[17] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Basiskonzepte und requirements Engineering*. Springer, 2009 (Zitiert auf Seite 63)

[18] Bauer, Bernhard ; Höllerer, Riitta: *Übersetzung objektorientierter Programmiersprachen: Konzepte, abstrakte Maschinen und Praktikum „Java-Compiler“*. Springer, 1998 (Zitiert auf den Seiten 23 und 29)

[19] Böhm, Oliver: *Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP*. dpunkt.verlag, 2006 (Zitiert auf Seite 174)

[20] Brinks, Dominic F. J.: *Declarative Debugging in Gödel*, University of Bristol, Diss., 1995 (Zitiert auf Seite 232)

[21] Brodie, Leo: *Starting FORTH: an introduction to the FORTH language and operating system for beginners and professionals*. Prentice-Hall, 1987 (Zitiert auf Seite 18)

[22] Bruneton, Eric ; Lenglet, Romain ; Coupaye, Thierry: ASM: A code manipulation tool to implement adaptable systems. In: *Adaptable and extensible component systems*. Grenoble, Frankreich, 2002 (Zitiert auf Seite 174)

[23] Burke, Michael G. ; Choi, Jong-Deok ; Fink, Stephen ; Grove, David ; Hind, Michael ; Sarkar, Vivek ; Serrano, Mauricio J. ; Sreedhar, V. C. ; Srinivasan, Harini ; Whaley, John: *The Jalapeño Dynamic Optimizing Compiler for Java*. 1999 (Zitiert auf den Seiten 23 und 224)

[24] Caballero, Rafael: A declarative debugger of incorrect answers for constraint functional-logic programs. In: *WCFLP ’05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, ACM, 2005, S. 8–13 (Zitiert auf Seite 230)

[25] Caballero, Rafael ; Hermanns, Christian ; Kuchen, Herbert: Algorithmic Debugging of Java Programs. In: *Electronic Notes in Theoretical Computer Science* 177 (2007), S. 75–89 (Zitiert auf Seite 3)

[26] Caballero, Rafael ; Rodríguez-Artalejo, Mario: A Declarative Debugging System for Lazy Functional Logic Programs. In: *Electronic Notes in Theoretical Computer Science* 64 (2002), S. 113 – 175. – WFLP 2001, International Workshop on Functional and

(Constraint) Logic Programming, Selected Papers (Zitiert auf Seite 229)

[27] Caballero, Rafael ; Rodríguez-Artalejo, Mario ; Del Vado Vírseda, Rafael: Declarative diagnosis of missing answers in constraint functional-logic programming. In: *FLOPS'08: Proceedings of the 9th international conference on Functional and logic programming*, Springer-Verlag, 2008, S. 305–321 (Zitiert auf Seite 230)

[28] Caballero, Rafael ; Rodríguez-Artalejo, Mario ; Del Vado Vírseda, Rafael: Declarative Diagnosis of Wrong Answers in Constraint Functional-Logic Programming. In: Etalle, Sandro (Hrsg.) ; Truszczynski, Miroslaw (Hrsg.): *Logic Programming* Bd. 4079. Springer, 2006, S. 421–422 (Zitiert auf Seite 230)

[29] Caballero, Roldan ; Rodríguez-Artalejo, Mario: A Declarative Debugging System for Lazy Functional Logic Programs. In: *Electronic Notes in Theoretical Computer Science* 64 (2002) (Zitiert auf Seite 57)

[30] Calvert, Kenneth L. ; Donahoo, Michael J.: *TCP/IP Sockets in Java: Practical Guide for Programmers*. Elsevier, 2008 (Zitiert auf Seite 156)

[31] Chang, Bay-Wei W.: *Objective reality for self: concreteness and animation in the seity user interface*, Stanford University, USA, Diss., 1996 (Zitiert auf Seite 227)

[32] Chiba, Shigeru: Javassist - A Reflection-based Programming Wizard for Java. In: *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998 (Zitiert auf Seite 174)

[33] Chiba, Shigeru ; Nishizawa, Muga: An easy-to-use toolkit for efficient Java bytecode translators. In: *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, Springer, 2003, S. 364–376 (Zitiert auf Seite 174)

[34] Choi, Jong-Deok ; Srinivasan, Harini: Deterministic replay of Java multithreaded applications. In: *SPDT '98: Proceedings of the SIG-*

METRICS symposium on Parallel and distributed tools, ACM, 1998, S. 48–59 (Zitiert auf Seite 224)

[35] Consens, Mariano ; Hasan, Masum ; Mendelzon, Alberto: Visualizing and querying distributed event traces with Hy+. In: Litwin, Witold (Hrsg.) ; Risch, Tore (Hrsg.): *Applications of Databases* Bd. 819. Springer, 1994, S. 123–141 (Zitiert auf Seite 219)

[36] Cooper, Stephen ; Dann, Wanda ; Pausch, Randy: Alice: a 3-D tool for introductory programming concepts. In: *CCSC '00: Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges*, Consortium for Computing Sciences in Colleges, 2000, S. 107–116 (Zitiert auf Seite 219)

[37] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms*. 2nd. The MIT Press, 2001 (Zitiert auf den Seiten 18, 58, 89 und 128)

[38] Coulouris, George F. ; Dollimore, Jean ; Kindberg, Tim: *Distributed systems: concepts and design*. Pearson Education, 2005 (Zitiert auf Seite 10)

[39] Dahm, Markus: Byte Code Engineering. In: *Java-Informations-Tage*, Springer-Verlag, 1999, S. 267–277 (Zitiert auf Seite 173)

[40] Daum, Berthold: *Java-Entwicklung mit Eclipse 3: Anwendungen, Plugins und Rich Clients*. dpunkt.verlag, 2004 (Zitiert auf Seite 209)

[41] Davie, Thomas ; Chitil, Olaf: One right does make a wrong. In: *Seventh Symposium on Trends in Functional Programming*, 2006, S. 27–40 (Zitiert auf den Seiten 229, 232 und 234)

[42] Dowson, Mark: The Ariane 5 software failure. In: *SIGSOFT Software Engineering Notes* 22 (1997), Nr. 2, S. 84 (Zitiert auf Seite 2)

[43] Ducassé, Mireille: Opium: an extendable trace analyzer for Prolog. In: *The Journal of Logic Programming* 39 (1999), Nr. 1-3, S. 177–223 (Zitiert auf Seite 220)

[44] Ducassé, Mireille: Coca: an automated debugger for C. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*, ACM, 1999, S. 504–513 (Zitiert auf Seite 220)

[45] Ducassé, Mireille ; Emde, Anna-Maria: OPIUM: a debugging environment for Prolog development and debugging research. In: *SIGSOFT Software Engineering Notes* 16 (1991), Nr. 1, S. 54–59 (Zitiert auf Seite 220)

[46] Dustin, Elfride: *Effective software testing: 50 specific ways to improve your testing*. Addison-Wesley, 2002 (Zitiert auf Seite 47)

[47] Eckel, Bruce: *Thinking in Java*. 4. Prentice Hall, 2006 (Zitiert auf Seite 13)

[48] Eclipse Foundation: *Eclipse IDE*. `http://www.eclipse.org/`, 2010 (Zitiert auf den Seiten 46, 146 und 209)

[49] Eisenstadt, Marc: My hairiest bug war stories. In: *Communications of the ACM* 40 (1997), Nr. 4, S. 30–37 (Zitiert auf Seite 3)

[50] Elrad, Tzilla ; Filman, Robert E. ; Bader, Atef: Aspect-oriented programming: Introduction. In: *Communications of the ACM* 44 (2001), Nr. 10, S. 29–32 (Zitiert auf Seite 174)

[51] Feldman, Stuart I. ; Brown, Channing B.: IGOR: a system for program debugging via reversible execution. In: *SIGPLAN Notices* 24 (1989), Nr. 1, S. 112–123 (Zitiert auf Seite 225)

[52] Forman, Ira R. ; Forman, Nate: *Java reflection in action*. Manning, 2004 (Zitiert auf den Seiten 162 und 164)

[53] Fowler, Martin: *Patterns für Enterprise Application-Architekturen*. Addison-Wesley Professional, 2003 (Zitiert auf Seite 210)

[54] Fritzson, Peter ; Shahmehri, Nahid ; Kamkar, Mariam ; Gyimothy, Tibor: Generalized algorithmic debugging and testing. In: *ACM Letters on Programming Languages and Systems* 1 (1992), Nr. 4, S. 303–322 (Zitiert auf Seite 230)

[55] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Boston, MA : Addison-Wesley, 1995 (Zitiert auf den Seiten 174, 193, 194, 206, 209 und 212)

[56] Georges, Andy ; Christiaens, Mark ; Ronsse, Michiel ; De Bosschere, Koen: JaRec: a portable record/replay environment for multi-threaded Java applications. In: *Software - Practice & Experience* 34 (2004), Nr. 6, S. 523–547 (Zitiert auf Seite 224)

[57] Girgis, Hani Z. ; Jayaraman, Bharat: JavaDD: a Declarative Debugger for Java / Department of Computer Science and Engineering, University at Buffalo. 2006 (7). – Forschungsbericht (Zitiert auf Seite 220)

[58] Girgis, Moheb R. ; Woodward, Martin R.: An experimental comparison of the error exposing ability of program testing criteria. In: Banff (Hrsg.): *Proceedings: Workshop on Software Testing*, IEEE Computer Society Press, 1986, S. 64–73 (Zitiert auf den Seiten 96 und 102)

[59] Gosling, James ; Joy, Bill ; Bracha, Gilad: *The Java language specification*. 3. Addison-Wesley, 2005 (Zitiert auf Seite 7)

[60] Grosso, William: *Java RMI*. O'Reilly, 2002 (Zitiert auf Seite 156)

[61] Gupta, Samudra: *Pro Apache log4j*. 2. Apress, 2005 (Zitiert auf den Seiten 47 und 169)

[62] Hailpern, B. ; Santhanam, P.: Software debugging, testing, and verification. In: *IBM Systems Journal* 41 (2002), Nr. 1, S. 4–12 (Zitiert auf Seite 3)

[63] Halloway, Stuart: *Programming Clojure*. Pragmatic Bookshelf, 2009 (Zitiert auf Seite 242)

[64] Hermanns, Christian ; Kuchen, Herbert: Implementation and Evaluation of a Declarative Debugger for Java. In: Mariño, Julio (Hrsg.): *Functional and (Constraint) Logic Programming: 19th International Workshop, WFLP 2010 Madrid, Spain, 17. Jannuar, 2010. Informal Proceedings*, 2010, S. 157–171 (Zitiert auf Seite 3)

[65] Hitchens, Ron: *Java NIO*. O'Reilly, 2002 (Zitiert auf Seite 158)

[66] Hoare, Charles A. R.: Monitors: an operating system structuring concept. In: *Communications of the ACM* 17 (1974), Nr. 10, S. 549–557 (Zitiert auf Seite 11)

[67] Hofer, Christoph ; Denker, Marcus ; Ducasse, Stéphane: Design and Implementation of a Backward-In-Time Debugger. In: *Proceedings of NODE'06* Bd. P-88, 2006 (Lecture Notes in Informatics), S. 17–32 (Zitiert auf Seite 227)

[68] Hoffmann, Erwin: *Technik der IP-Netze - TCP/IP incl. IPv6 : Funktionsweise, Protokolle und Dienste.* 2. Hanser Verlag, 2007 (Zitiert auf Seite 156)

[69] Hughes, Thomas P.: *American genesis: a century of invention and technological enthusiasm, 1870-1970.* University of Chicago Press, 2004 (Zitiert auf Seite 1)

[70] Jaffar, Joxan ; Maher, Michael J.: Constraint logic programming: a survey. In: *The Journal of Logic Programming* 19–20 (1994), Nr. 1, S. 503–581 (Zitiert auf Seite 229)

[71] JetBrains: *IntelliJ IDEA.* `http://www.jetbrains.com/idea/`, 2010 (Zitiert auf Seite 146)

[72] Jones, Richard: *Garbage Collection.* Wiley, 1996 (Zitiert auf den Seiten 10 und 17)

[73] Jones, Simon P. (Hrsg.): *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003 (Zitiert auf Seite 229)

[74] Juneau, Josh ; Baker, Jim ; Wierzbicki, Frank ; Soto, Leo ; Ng, Victor: *The Definitive Guide to Jython: Python for the Java Platform.* Apress, 2010 (Zitiert auf Seite 242)

[75] Kay, Alan C.: The early history of Smalltalk. In: *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, ACM, 1993, S. 69–95 (Zitiert auf Seite 227)

[76] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An Overview of AspectJ. In: *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, Springer-Verlag, 2001, S. 327–353 (Zitiert auf Seite 174)

[77] Kniesel, Günter: *JMangler*. `http://roots.iai.uni-bonn.de/research/jmangler/`, 2010 (Zitiert auf Seite 175)

[78] Kniesel, Günter ; Costanza, Pascal ; Austermann, Michael: JMangler - A Framework for Load-Time Transformation of Java Class Files. In: *SCAM 2001, Proceedings of IEEE Workshop on Source Code Analysis and Manipulation*, IEEE Computer Society Press, 2001, S. 100–110 (Zitiert auf Seite 175)

[79] Ko, Andrew J. ; Myers, Brad A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 2004, S. 151–158 (Zitiert auf Seite 219)

[80] Ko, Andrew J. ; Myers, Brad A.: Finding causes of program output with the Java Whyline. In: *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, ACM, 2009, S. 1569–1578 (Zitiert auf Seite 219)

[81] Konuru, Ravi ; Srinivasan, Harini ; Choi, Jong-Deok: Deterministic Replay of Distributed Java Applications. In: *Parallel and Distributed Processing Symposium, International* (2000), S. 219 (Zitiert auf Seite 224)

[82] Korel, Bogdan ; Laski, Janusz W.: Dynamic program slicing. In: *Information Processing Letters* Bd. 29, 1988, S. 155–163 (Zitiert auf Seite 230)

[83] Kouh, Hoon-Joon ; Jo, Sun-Moon ; Yoo, Weon-Hee: Design of HDTS System for Locating Logical Errors in Java Programs. In: *ICIS '05: Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science*, IEEE Computer Society, 2005, S. 418–423 (Zitiert auf Seite 230)

[84] *Kapitel* Debugging of Java Programs Using HDT with Program Slicing. In: Kouh, Hoon-Joon ; Kim, Ki-Tae ; Jo, Sun-Moon ; Yoo, Weon-Hee: *Lecture Notes in Computer Science*. Bd. 3046: *Computational Science and Its Applications - ICCSA 2004*. Springer, 2004, S. 524–533 (Zitiert auf Seite 230)

[85] Kouh, Hoon-Joon ; Yoo, Weon-Hee: Hybrid Debugging Method: Algorithmic + Step-wise Debugging. In: Kumar, Vipin (Hrsg.) ; Gavrilova, Marina (Hrsg.) ; Tan, Chih (Hrsg.) ; L'Ecuyer, Pierre (Hrsg.): *Proceedings of the 2002 International Conference on Software Engineering Research and Practice*, 2002, S. 342–347 (Zitiert auf Seite 230)

[86] Kouh, Hoon-Joon ; Yoo, Weon-Hee: The Efficient Debugging System for Locating Logical Errors in Java Programs. In: Kumar, Vipin (Hrsg.) ; Gavrilova, Marina (Hrsg.) ; Tan, Chih (Hrsg.) ; L'Ecuyer, Pierre (Hrsg.): *Computational Science and Its Applications — ICCSA 2003* Bd. 2667. Springer, 2003, S. 961–961 (Zitiert auf Seite 230)

[87] Kowalski, Robert: Algorithm = logic + control. In: *Communications of the ACM* 22 (1979), Nr. 7, S. 424–436 (Zitiert auf Seite 75)

[88] Krall, Andreas: Efficient JavaVM Just-in-Time Compilation. In: *International Conference on Parallel Architectures and Compilation Techniques*, 1998, S. 205–212 (Zitiert auf den Seiten 11 und 23)

[89] Krüger, Guido: *Handbuch der Java-Programmierung*. 4. Addison-Wesley, 2006 (Zitiert auf den Seiten 13, 50 und 63)

[90] Landi, William: Undecidability of Static Analysis. In: *ACM Letters on Programming Languages and Systems* 1 (1992), S. 323–337 (Zitiert auf Seite 46)

[91] langpop.com: *Programming Language Popularity*. `http://www.langpop.com/`, 2010 (Zitiert auf Seite 7)

[92] LeBlanc, Thomas J. ; Mellor-Crummey, John M.: Debugging Parallel Programs with Instant Replay. In: *IEEE Transactions on Computers* 36 (1987), Nr. 4, S. 471–482 (Zitiert auf Seite 224)

[93] LeDoux, Carol H. ; Parker, D. S. Jr.: Saving traces for Ada debugging. In: *SIGAda '85: Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, Cambridge University Press, 1985, S. 97–108 (Zitiert auf Seite 220)

[94] Lencevicius, Raimondas ; Hölzle, Urs ; Singh, Ambuj: Dynamic Query-Based Debugging of Object-Oriented Programs. In: *Automated Software Engineering* 10 (2003), S. 39–74 (Zitiert auf Seite 220)

[95] Lewis, Bil: Debugging Backwards in Time. In: *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, 2003 (Zitiert auf den Seiten 54, 56 und 227)

[96] Lewis, Bil ; Ducasse, Mireille: Using events to debug Java programs backwards in time. In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, 2003, S. 96–97 (Zitiert auf Seite 227)

[97] Lieberman, Henry ; Fry, Christopher: Bridging the gulf between code and behavior in programming. In: *CHI'95: Human Factors in Computing Systems*, ACM Press, 1995, S. 480–486 (Zitiert auf Seite 227)

[98] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002 (Zitiert auf den Seiten 95, 97, 98, 99, 101 und 102)

[99] Lippman, Stanley B. ; Lajoie, Josée ; Moo, Barbara E.: *C++Primer*. 4. Addison-Wesley, 2006 (Zitiert auf Seite 50)

[100] Lloyd, John W.: Practical advantages of declarative programming. In: *Joint Conference on Declarative Programming: GULP-PRODE* Bd. 2, 1994 (Zitiert auf Seite 75)

[101] López Fraguas, Francisco ; Sánchez Hernández, José: TOY: A Multiparadigm Declarative System. In: Narendran, Paliath (Hrsg.) ; Rusinowitch, Michael (Hrsg.): *Rewriting Techniques and Applications* Bd. 1631. Springer, 1999, S. 244–247 (Zitiert auf Seite 230)

[102] Luk, Chi keung ; Cohn, Robert ; Muth, Robert ; Patil, Harish ; Klauser, Artur ; Lowney, Geoff ; Wallace, Steven ; Janapa, Vijay ; Hazelwood, Reddi K.: Pin: building customized program analysis tools with dynamic instrumentation. In: *In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming*

language design and implementation, ACM Press, 2005, S. 190–200 (Zitiert auf Seite 47)

[103] MacLarty, Ian: *Practical Declarative Debugging of Mercury Programs*, The University of Melbourne, Diss., 2005 (Zitiert auf Seite 229)

[104] MacLarty, Ian: *Practical Declarative Debugging of Mercury Programs*, Department of Computer Science and Software Engineering, The University of Melbourne, Diss., 2005 (Zitiert auf den Seiten 232 und 233)

[105] Maeji, Machi ; Kanamori, Tadashi: Top-Down Zooming Diagnosis of Logical Programs / ICOT, Japan. 1987 (TR-290). – Forschungsbericht (Zitiert auf Seite 232)

[106] Martin, Michael ; Livshits, Benjamin ; Lam, Monica S.: Finding application errors and security flaws using PQL: a program query language. In: *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, 2005, S. 365–383 (Zitiert auf Seite 219)

[107] Marwedel, Peter: *Embedded System Design.* Birkhäuser, 2006 (Zitiert auf Seite 8)

[108] Michiel, Frank C. ; Cornelis, Frank ; Ronsse, Michiel ; Bosschere, Koen D.: TORNADO: A Novel Input Replay Tool. In: *In Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03*, CSREA Press, 2003, S. 1598–1604 (Zitiert auf Seite 223)

[109] Mills, Bruce: *Theoretical introduction to programming.* Birkhäuser, 2006 (Zitiert auf Seite 18)

[110] Müller, Thomas ; Graham, Dorothy ; Friedenberg, Debra ; Veendendal, Erik van: Certified Tester Foundation Level Syllabus / International Software Testing Qualifications Board. 2007 (1). – Forschungsbericht (Zitiert auf Seite 38)

[111] Myers, Glenford J.: *The Art of Software Testing, Second Edition.* 2. John Wiley & Sons, 2004 (Zitiert auf Seite 45)

[112] Naish, Lee: A Declarative Debugging Scheme. In: *Journal of Functional and Logic Programming* 1997 (1997), April, Nr. 3 (Zitiert auf Seite 229)

[113] Naish, Lee ; W., Philip ; Zobel, Justin: The NU-Prolog debugging environment. In: Porto, Antonio (Hrsg.): *Proceedings of the Sixth International Conference on Logic Programming*. Lisboa, Portugal, 1989, S. 521–536 (Zitiert auf Seite 229)

[114] Netzer, Robert H. B.: Optimal tracing and replay for debugging shared-memory parallel programs. In: *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, ACM, 1993, S. 1–11 (Zitiert auf Seite 225)

[115] Netzer, Robert H. B. ; Miller, Barton P.: What are race conditions?: Some issues and formalizations. In: *ACM Letters on Programming Languages and Systems* 1 (1992), Nr. 1, S. 74–88 (Zitiert auf Seite 224)

[116] Nielson, Flemming ; Nielson, Hanne R. ; Hankin, Chris: *Principles of Program Analysis*. Springer, 1999 (Zitiert auf Seite 101)

[117] Nilsson, Henrik: *Declarative Debugging for Lazy Functional Languages*, Linköping, Sweden, Diss., 1998 (Zitiert auf Seite 229)

[118] Nilsson, Henrik: How to look busy while being as lazy as ever: the Implementation of a lazy functional debugger. In: *Journal of Functional Programming* 11 (2001), Nr. 6, S. 629–671 (Zitiert auf den Seiten 57 und 229)

[119] Nilsson, Henrik ; Fritzon, Peter: Algorithmic Debugging for Lazy Functional Languages. In: *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, Springer-Verlag, 1992, S. 385–399 (Zitiert auf Seite 229)

[120] Object Web: *ASM*. `http://asm.ow2.org/`, 2009 (Zitiert auf Seite 174)

[121] Omnicore: *CodeGuide*. `http://www.omnicore.com/de/codeguide.htm`, 8 2010 (Zitiert auf Seite 228)

[122] OSGi Alliance: *OSGi - The Dynamic Module System for Java*. http://www.osgi.org/Main/HomePage, 2010 (Zitiert auf Seite 209)

[123] Pan, Douglas Z. ; Linton, Mark A.: Supporting reverse execution for parallel programs. In: *SIGPLAN Not.* 24 (1989), Nr. 1, S. 124–129 (Zitiert auf Seite 225)

[124] Pope, Bernard: *A Declarative Debugger for Haskell*, The University of Melbourne, Australia, Diss., 2006 (Zitiert auf Seite 229)

[125] Pope, Bernard ; Naish, Lee: Practical aspects of declarative debugging in Haskell 98. In: *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, ACM, 2003, S. 230–240 (Zitiert auf Seite 229)

[126] Pothier, Guillaume ; Tanter, Éric ; Piquer, José: Scalable omniscient debugging. In: *SIGPLAN Notices* 42 (2007), Nr. 10, S. 535–552 (Zitiert auf den Seiten 227, 228 und 241)

[127] Ronsse, Michiel ; De Bosschere, Koen: RecPlay: a fully integrated practical record/replay system. In: *ACM Transactions on Computer Systems* 17 (1999), Nr. 2, S. 133–152 (Zitiert auf Seite 225)

[128] Russinovich, Mark ; Cogswell, Bryce: Replay for concurrent non-deterministic shared-memory applications. In: *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, ACM, 1996, S. 258–266 (Zitiert auf Seite 224)

[129] Shahmehri, Nahid ; Fritzson, Peter: Algorithmic debugging for imperative languages with side-effects. In: Hammer, Dieter (Hrsg.): *Compiler Compilers* Bd. 477. Springer, 1991, S. 226–227 (Zitiert auf Seite 230)

[130] Shapiro, Ehud Y.: *Algorithmic Program DeBugging*. MIT Press, 1983 (Zitiert auf den Seiten 57, 86, 229 und 233)

[131] Shapiro, Fred R.: Etymology of the Computer Bug: History and Folklore. In: *American Speech* 62 (1987), Nr. 4, S. 376–378 (Zitiert auf Seite 1)

[132] *Kapitel* A Comparative Study of Algorithmic Debugging Strategies. In: Silva, Josep: *Lecture Notes in Computer Science*. Bd. 4407: *Logic-Based Program Synthesis and Transformation*. Springer, 2007, S. 143–159 (Zitiert auf den Seiten 232, 233, 234 und 242)

[133] Søndergaard, Harald ; Sestoft, Peter: Referential transparency, definiteness and unfoldability. In: *Acta Informatica* 27 (1990), Nr. 6, S. 505–517 (Zitiert auf Seite 66)

[134] Snodgrass, Richard T.: Monitoring in a software development environment: A relational approach. In: *SIGSOFT Software Engineering Notes* 9 (1984), Nr. 3, S. 124–131 (Zitiert auf Seite 219)

[135] Snodgrass, Richard T. ; Gomez, Santiago ; Mckenzie, L. E.: Aggregates in the Temporary Query Language TQuel. In: *IEEE Transactions on Knowledge and Data Engineering* 5 (1993) (Zitiert auf Seite 219)

[136] Somogyi, Zoltan ; Henderson, Fergus ; Conway, Thomas: The execution algorithm of mercury, an efficient purely declarative logic programming language. In: *The Journal of Logic Programming* 29 (1996), Nr. 1-3, S. 17 – 64 (Zitiert auf Seite 229)

[137] Srivastava, Amitabh ; Eustace, Alan: Atom: A system for building customized program analysis tools / Western Research Lab, Compaq. 1994 (94/2). – Forschungsbericht (Zitiert auf Seite 47)

[138] Steele, Jr. Guy L. ; Gabriel, Richard P.: The evolution of Lisp. In: *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, ACM, 1993, S. 231–270 (Zitiert auf Seite 227)

[139] Steven, John ; Chandra, Pravir ; Fleck, Bob ; Podgurski, Andy: jRapture: A Capture/Replay tool for observation-based testing. In: *SIGSOFT Software Engineering Notes* 25 (2000), Nr. 5, S. 158–167 (Zitiert auf Seite 223)

[140] Sun Microsystems: *Java Instrumentation API.* http://java.sun.com/javase/6/docs/api/java/lang/instrument/Instrumentation.html, 2010 (Zitiert auf Seite 170)

[141] Sun Microsystems: *Java Platform Debugger Architecture.* http://java.sun.com/javase/technologies/core/toolsapis/jpda/, 2010 (Zitiert auf Seite 114)

[142] Sun Microsystems: *Java Platform, Enterprise Edition.* http://java.sun.com/j2ee/, 2010 (Zitiert auf Seite 8)

[143] Sun Microsystems: *Java Platform, Micro Edition.* http://java.sun.com/j2me/, 2010 (Zitiert auf Seite 8)

[144] Sun Microsystems: *Java Platform, Standard Edition.* http://java.sun.com/j2se/, 2010 (Zitiert auf Seite 8)

[145] Sun Microsystems: *Java Platform, Standard Edition 6 API Specification.* http://java.sun.com/javase/6/docs/api/, 2010 (Zitiert auf den Seiten 8, 14, 109, 146 und 156)

[146] Sun Microsystems: *Java SE HotSpot at a Glance.* http://java.sun.com/javase/technologies/hotspot/index.jsp, 2010 (Zitiert auf Seite 23)

[147] Sun Microsystems: *NetBeans IDE.* http://netbeans.org/, 2010 (Zitiert auf Seite 146)

[148] Sun Microsystems: *The Java Language: An Overview.* http://java.sun.com/docs/overviews/java/java-overview-1.html, 2010 (Zitiert auf Seite 9)

[149] Sun Microsystems: *The Java Language Environment.* http://java.sun.com/docs/white/langenv/Intro.doc2.html, 2010 (Zitiert auf Seite 9)

[150] Sun Microsystems: *The Java Language Specification.* http://java.sun.com/docs/books/jls/, 2010 (Zitiert auf den Seiten 11 und 13)

[151] Sun Microsystems: *The Java Virtual Machine Specification.* http://java.sun.com/docs/books/jvms/, 2010 (Zitiert auf den Seiten 14 und 151)

[152] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme: Grundlagen und Paradigmen.* Pearson Studium, 2003 (Zitiert auf Seite 10)

[153] Tassey, Gregory: The economic impacts of inadequate infrastructure for software testing / National Institute of Standards and Technology. 2002 (3). – Forschungsbericht (Zitiert auf Seite 2)

[154] Tessier, Alexandre: Declarative Debugging in Constraint Logic Programming. In: *Asian Computing Science Conference*, Springer-Verlag, 1996, S. 64–73 (Zitiert auf Seite 230)

[155] Tessier, Alexandre ; Ferrand, Gérard: Declarative Diagnosis in the CLP Scheme. In: *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, Springer-Verlag, 2000, S. 151–174 (Zitiert auf Seite 230)

[156] TIOBE Software: *TIOBE Programming Community Index for August 2010.* http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2010 (Zitiert auf Seite 7)

[157] Tip, Frank: A Survey of Program Slicing Techniques. In: *Journal of Programming Languages* 3 (1995), S. 121–189 (Zitiert auf Seite 46)

[158] Troelsen, Andrew: *Pro C# 2010 and the .NET 4 Platform, Fifth Edition.* 5. Apress, 2010 (Zitiert auf den Seiten 50 und 63)

[159] Ungar, David ; Lieberman, Henry ; Fry, Christopher: Debugging and the experience of immediacy. In: *Communications of the ACM* 40 (1997), Nr. 4, S. 38–43 (Zitiert auf Seite 227)

[160] Van-Roy, Peter ; Haridi, Seif: *Concepts, techniques, and models of computer programming.* MIT Press, 2004 (Zitiert auf Seite 229)

[161] Wampler, Dean ; Payne, Alex: *Programming Scala: Scalability = Functional Programming + Objects.* O'Reilly Media, 2009 (Zitiert auf Seite 242)

[162] Wegner, Peter: Concepts and paradigms of object-oriented programming. In: *SIGPLAN OOPS Mess.* 1 (1990), Nr. 1, S. 7–87 (Zitiert auf den Seiten 9 und 63)

[163] White, Abe: *Serp*. `http://serp.sourceforge.net/`, 2010 (Zitiert auf Seite 173)

[164] Wilhelm, Reinhard ; Maurer, Dieter: *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer, 1997 (Zitiert auf Seite 29)

[165] Wirth, Niklaus: The Programming Language Pascal. In: *Acta Informatica* 1 (1971), S. 35–63 (Zitiert auf Seite 230)

[166] Wirth, Niklaus: *Texts and monographs in computer science*. Bd. 4: *Programming in Modula-2*. Springer, 1988 (Zitiert auf Seite 224)

[167] Wütherich, Gerd ; Hartmann, Nils ; Kolb, Bernd ; Lübken, Matthias: *Die OSGI Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt.verlag, 2008 (Zitiert auf Seite 209)

[168] Zeller, Andreas: *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005 (Zitiert auf den Seiten 38 und 54)

Entwicklung und Implementierung eines hybriden Debuggers für Java

Christian Hermanns

Das Debugging ist ein komplexer und arbeitsintensiver Prozess in der Softwareentwicklung. Für das Debugging von Java-Programmen werden bis heute vor allem sogenannte Trace-Debugger verwendet. Diese unterstützen die Fehlersuche, indem sie es ermöglichen, ein untersuchtes Programm schrittweise auszuführen. Im Bereich der Forschung sind viele neue Methoden und Werkzeuge entwickelt worden, die im Vergleich zum Trace-Debugging eine erhebliche Verbesserung und Vereinfachung des Debugging-Prozesses versprechen. Auf die in der Praxis eingesetzten Verfahren hatten diese Entwicklungen bisher nur einen äußert geringen Einfluss. In der vorliegenden Arbeit wird die Entwicklung und Implementierung einer neuen hybriden Debugging-Methode für Java-Programme beschrieben. Die Methode kombiniert deklaratives Debugging und Omniscient-Debugging.