PG 458 3Debug
Endbericht

Dieses Dokument wurde verfasst von
Boris Brodski, Boris Düdder, Carina Klar, Andrey Lysenko,
Daniel Maliga, Jonas Mathis, Antonio Pedicillo, Michael Striewe,
Sebastian Vastag, Daniel Vogtland, Dennis Weyland, Henning Zeller

Projektleitung: Alexander Fronk, Jens Schröder, Sören Blom

Inhaltsverzeichnis

Abbildungsverzeichnis xiii

1 Einführung in die Projektgruppe 3DEBUG

KAPITEL 1
Einleitung 2

KAPITEL 2
Geplantes Vorgehen 4
2.1 Einarbeitung, Seminarphase 4
2.2 Anforderungsanalyse 4
2.3 Konstruktion 5
2.4 Berichte 5
2.5 Fachgespräch 5

KAPITEL 3
Übersicht über den Abschlussbericht 6

2 Seminarphase

KAPITEL 4
Alternative Modellierungskonzepte in 3D 8
4.1 Einleitung 8
4.2 Geon-basierte Darstellungen 8
4.2.1 Menschliches Bild-Verstehen: Die Biederman Theorie 9
4.2.2 Geonen und UML 10
4.3 Graph-basierte Visualisierungen 13
4.3.1 Walrus: Allgemeine Graphdarstellungen 14
4.3.2 CrocoCosmos: Software-spezifische Graphen 16
4.4 Animiertes 3D UML 17
4.5 Ausblick 21

KAPITEL 5

Debugging-Techniken 22

5.1 Debugging-Techniken 22
5.1.1 Der Begriff Debugging 22
5.1.2 Heutiger Debugging-Begriff 22
5.1.3 Klassen von Bugs 24
5.1.4 Schädlingsbekämpfungsmittel 25
5.1.5 Der Debuggingprozess 25
5.1.6 Debugging-Techniken 26
5.1.7 Softwaretools 27
5.2 Integrated Comprehension Model 29
5.2.1 Top-Down Model 30
5.2.2 Program Model 31
5.2.3 Situation Model 31
5.2.4 Fallstudie 31
5.2.5 Ergebnisse der Studie 33
5.2.6 Informationsgewinnung 33
5.3 Grafische Tools 34
5.4 Schlüsse und Ausblick 34

KAPITEL 6

Debugging-Werkzeuge 36

6.1 Einleitung 36
6.2 Heisenberg - Prinzipien 37
6.2.1 Das erste Heisenberg - Prinzip 37
6.2.2 Das zweite Heisenberg - Prinzip 37
6.2.3 Das dritte Heisenberg - Prinzip 38
6.3 Der Kontext 38
6.3.1 Quelltextansicht (source view) 38
6.3.2 Stapelansicht (stack view) 38
6.3.3 Breakpointsansicht (breakpoint view) 39
6.3.4 Disassembleransicht (disassemble view) 39
6.3.5 CPU-Ansicht (CPU view) 40
6.3.6 Speicheransicht (memory view) 40
6.3.7 Variablenansicht (variable view) 40
6.3.8 Evaluatoransicht (evaluator view) 40
6.4 Bestandteile des Debuggers 41
6.4.1 Debugging Interface 41
6.4.2 Debug-Information 42
6.4.3 Breakpoints 43
6.4.4 Programmstapel 44
6.4.5 Disassemblierung 44
6.4.6 Gültigkeitsbereiche der Variablen 45
6.4.7 Evaluation der Ausdrücke 45
6.4.8 Aufrufe der Funktionen während Evaluation 45

6.5 Step-Algorithmen . . . 45
6.5.1 Single Step Algorithmus . . . 46
6.5.2 Step Into Algorithmus . . . 46
6.5.3 Step Over Algorithmus . . . 47
6.6 Debugging besonderer Arten von Applikationen . . . 47
6.6.1 Debugging von nebenläufigen Applikationen . . . 47
6.6.2 Debugging von GUI-Applikationen . . . 47
6.6.3 Debugging von optimiertem Code . . . 48
6.7 Fazit . . . 48

KAPITEL 7

Mathematische Grundlagen der 3D Modellierung . . . 49
7.1 Vektoren . . . 49
7.1.1 Vektor . . . 49
7.1.2 Norm . . . 49
7.1.3 Verknüpfungen . . . 50
7.1.4 Linearkombination, lineare Unabhängigkeit . . . 50
7.1.5 Basis . . . 51
7.2 Matrizen . . . 52
7.2.1 Matrix . . . 52
7.2.2 Multiplikation Matrix-Vektor . . . 52
7.2.3 Matrizenmultiplikation . . . 53
7.2.4 Determinante . . . 53
7.2.5 Spezielle Matrizen . . . 53
7.3 Objekte in 2D und 3D . . . 54
7.3.1 Linien . . . 54
7.3.2 Ebenen . . . 54
7.3.3 Schnitt Linie-Ebene . . . 55
7.3.4 Schnitt Ebene-Ebene . . . 55
7.4 Anwendungen . . . 56
7.4.1 Translation . . . 56
7.4.2 Skalierung . . . 56
7.4.3 Rotation . . . 57
7.4.4 Projektion . . . 58
7.4.5 Konkatenation von Transformation . . . 59
7.4.6 Inverse Transformation . . . 59

KAPITEL 8

Java3D - Eine Einführung . . . 61
8.1 Einleitung . . . 61
8.2 Grundlagen . . . 61
8.2.1 Begriffsklärung . . . 61
8.2.2 Die Java3D-Klassenhierarchie - ein kurzer Überblick . . . 62

8.3 Das Szenegraph-Konzept . . . 63
8.3.1 Überblick . . . 63
8.3.2 Elemente des Szenegraphen . . . 63
8.3.3 Zusammenhang zwischen Szenegraph und Java3D-Anwendungen . . . 65
8.4 Geometrische Strukturen . . . 66
8.4.1 Shape3D . . . 66
8.4.2 primitive Körper . . . 66
8.4.3 Appearance . . . 67
8.4.4 Beispiele . . . 67
8.5 Interaktion und Animation . . . 69
8.5.1 Behavior-Klassen . . . 69
8.5.2 Beispiel . . . 69
8.6 Fazit . . . 70

KAPITEL 9

Eclipse - Aufbau und Anwendung . . . 72
9.1 Geschichte . . . 72
9.2 Grundlagen . . . 72
9.2.1 Einführung . . . 72
9.2.2 Plug-Ins . . . 74
9.2.3 Workspace . . . 74
9.3 Aufbau . . . 74
9.3.1 Einführung . . . 74
9.3.2 Views . . . 74
9.3.3 Refresh . . . 76
9.3.4 Fehleranzeige . . . 77
9.4 Fazit . . . 77

KAPITEL 10

Das Eclipse Plug-In Modell . . . 79
10.1 Einleitung . . . 79
10.2 Die Workbench . . . 79
10.2.1 Ressourcen . . . 80
10.2.2 Editoren . . . 80
10.2.3 Views . . . 81
10.2.4 Perspektiven . . . 81
10.3 Das Plug-In Modell . . . 81
10.3.1 Initialisierungsvorgang . . . 82
10.3.2 Features . . . 83
10.4 Plug-In Manifest . . . 83
10.4.1 Deklaration von Erweiterungspunkten . . . 84
10.4.2 Beschreibung der Extension Points . . . 84
10.4.3 Fragment . . . 85

10.5 Deployment . . . 85
10.6 Effects . . . 85
10.7 Synopsis . . . 86

KAPITEL 11

Debugging APIs . . . 87
11.1 Einführung . . . 87
11.2 Grundlagen des Debuggings in Java . . . 87
11.2.1 Java Platform Debugging Architecture . . . 88
11.2.2 Java Debug Interface . . . 89
11.3 Grundlagen des Debuggings in Eclipse . . . 90
11.3.1 Launching . . . 90
11.3.2 Platform Debug Model . . . 92
11.3.3 User-Interface . . . 93

KAPITEL 12

eXtreme Programming . . . 95
12.1 Einleitung . . . 95
12.2 Warum XP? . . . 95
12.2.1 Grundprobleme der Softwareentwicklung . . . 95
12.2.2 XP als Lösung . . . 96
12.3 Werte von XP . . . 96
12.3.1 Kommunikation . . . 96
12.3.2 Einfachheit . . . 96
12.3.3 Feedback . . . 97
12.3.4 Mut . . . 97
12.4 Prinzipien von XP . . . 97
12.4.1 Unmittelbares Feedback . . . 97
12.4.2 Einfachheit anstreben . . . 98
12.4.3 Inkrementelle Veränderung . . . 98
12.4.4 Veränderung wollen . . . 98
12.4.5 Qualitätsarbeit . . . 98
12.4.6 Weitere Prinzipien . . . 98
12.5 Techniken von XP . . . 99
12.5.1 Metapher . . . 100
12.5.2 Planungsspiel . . . 100
12.5.3 Kunde vor Ort . . . 101
12.5.4 Einfaches Design (Simple Design) . . . 101
12.5.5 Kurze Releasezyklen (Short Releases) . . . 101
12.5.6 Fortlaufende Integration (Continuous Integration) . . . 101
12.5.7 Testen (Testing) . . . 102
12.5.8 Refactoring . . . 102
12.5.9 Programmieren in Paaren (Pair Programming) . . . 103
12.5.10 Gemeinsame Verantwortlichkeit (Collective Ownership) . . . 103

12.5.11 Programmierstandards (Coding Standards) . . . 104
12.5.12 40-Stunden-Woche (40 Hour Week) . . . 104
12.6 Rollen . . . 104
12.6.1 Kunde (Costumer) . . . 105
12.6.2 Programmierer (Programmer) . . . 105
12.6.3 Tester . . . 106
12.6.4 Verfolger (Tracker) . . . 107
12.6.5 XP-Trainer (XP-Coach) . . . 107
12.6.6 Berater (Consultant) . . . 108
12.6.7 Big Boss . . . 108
12.6.8 Zusatzrollen . . . 109
12.7 Anpassung der Rollen und Techniken . . . 110
12.7.1 Anpassung der Rollen . . . 110
12.7.2 Anpassung der Techniken . . . 111

3 Releasebeschreibungen

KAPITEL 13
Systemmetapher . . . 113

KAPITEL 14
Beschreibung des ersten Release . . . 114
14.1 Einleitung . . . 114
14.2 User-Stories . . . 114
14.3 Reflexion über die Tasks . . . 116
14.3.1 Aufbau der grundlegenden Infrastruktur . . . 117
14.3.2 Konzeptionelle Tasks . . . 117
14.3.3 Datenhaltung und -beschaffung . . . 119
14.3.4 Visualisierung . . . 122
14.4 Vorstellung der Architektur . . . 123
14.4.1 Beschreibung der geplanten Architektur . . . 123
14.4.2 Beschreibung der realisierten Architektur . . . 124
14.4.3 Vergleich von geplanter und realisierter Architektur . . . 128
14.5 Akzeptanztests . . . 128

KAPITEL 15
Beschreibung des zweiten Release . . . 133
15.1 Einleitung . . . 133
15.2 Machbarkeitsstudien . . . 133
15.2.1 Darstellung und Unterscheidung von Objektbeziehungen . . . 134
15.2.2 LinLog . . . 135

15.2.3 Force-Directed Layout . . . 135
15.2.4 Einfache Anordnungsalgorithmen . . . 136
15.3 User-Stories . . . 137
15.4 Reflexion über die Tasks . . . 140
15.4.1 Datenstruktur . . . 140
15.4.2 Benutzerinteraktion . . . 141
15.4.3 3D-Darstellung . . . 142
15.5 Vorstellung der Architektur . . . 144
15.5.1 Beschreibung der geplanten Architektur . . . 145
15.5.2 Beschreibung der realisierten Architektur . . . 146
15.5.3 Vergleich von geplanter und realisierter Architektur . . . 154
15.6 Akzeptanztests . . . 154

KAPITEL 16

Beschreibung des dritten Release . . . 158
16.1 Einleitung . . . 158
16.2 Reflexion über die Tasks . . . 159
16.2.1 Aufräumarbeiten . . . 159
16.2.2 Infobeschaffung . . . 160
16.2.3 Parametrisierung Layout-Algorithmen . . . 162
16.2.4 Animationen . . . 163
16.3 Erweiterung der Infobeschaffung . . . 164
16.3.1 Definition des Interface . . . 165
16.3.2 Einsatz des Java-Profilers . . . 166
16.3.3 Einsatz des Java Debug Interface . . . 168
16.3.4 Rekursive Snapshot-Beschaffung . . . 169
16.3.5 Kombination der Möglichkeiten . . . 169
16.4 Parametrisierung . . . 170
16.4.1 Definition des Interfaces . . . 171
16.4.2 Force-directed Layout . . . 172
16.4.3 Linlog . . . 173
16.5 Refactoring der Visualisierungen . . . 174
16.5.1 Struktur des Visualisierungsbereiches . . . 174
16.5.2 Ablauf einer Visualisierung . . . 175
16.5.3 Hintergrundbild . . . 179
16.6 Releaseabnahme . . . 182

KAPITEL 17

Beschreibung des vierten Release . . . 183
17.1 Einleitung . . . 183
17.2 Reflexion über die Tasks . . . 183
17.2.1 Bugfixes . . . 184
17.2.2 Usability (Benutzerinteraktion) . . . 185
17.2.3 Usability (Debugging-Funktionalität) . . . 187

17.2.4 Profiling . . . 188
17.2.5 Übrig gebliebene Arbeiten aus Release 3 . . . 190
17.3 Analyse einzelner Snapshots . . . 191
17.3.1 Mehrfachselektion . . . 191
17.3.2 Ausblendefunktion . . . 192
17.4 Snapshotplayer . . . 193
17.4.1 Funktionalitäten des Snapshotplayers . . . 193
17.4.2 Speichern und Laden der Snapshots . . . 195
17.4.3 Animation und Aufnahme . . . 195
17.5 Export in einen Film . . . 196
17.6 Informationsbeschaffung . . . 197
17.6.1 Java Debug Interface . . . 197
17.6.2 Integration von Informationsbeschaffungsalgorithmen . . . 198
17.7 Persistenz . . . 198
17.8 Profiling . . . 199
17.9 Releaseabnahme . . . 200

4 Reflexion

KAPITEL 18
Nutzen des Plug-Ins . . . 203
18.1 Top-Down-Modell . . . 203
18.2 Programmmodell . . . 204
18.3 Situationsmodell . . . 205
18.4 Beispiele aus der Praxis . . . 206
18.4.1 Auto . . . 206
18.4.2 Erkennen von Strukturen . . . 209
18.4.3 Algorithmus . . . 210

KAPITEL 19
Grenzen des Plug-Ins . . . 215
19.1 Kriterien . . . 215
19.2 Darstellung . . . 215
19.3 Ressourcen . . . 216
19.3.1 CPU . . . 216
19.3.2 Speicher . . . 216
19.3.3 Speicherung . . . 216
19.3.4 Suboptimale Ausnutzung der CPU . . . 217
19.3.5 Performanzverlust durch Netzwerkprotokoll (JDWP) . . . 217

19.4 Informationskollektoren . . . 217
19.4.1 Recursive Snapshot Collector . . . 217
19.4.2 Recursive Snapshot Collector (optimiert) . . . 218
19.4.3 NetDumper Collector . . . 218
19.4.4 JDI Collector . . . 219
19.5 Allgemeine Verwendbarkeit . . . 219

KAPITEL 20
Bedeutung der Anordnungsalgorithmen . . . 220
20.1 Bewertung des LinLog-Algorithmus . . . 221
20.2 Bewertung des Force-Directed-Layout . . . 223

KAPITEL 21
Bedeutung der Farb- und Formgebung der Objekte . . . 226
21.1 Bedeutung der Formen . . . 226
21.2 Bedeutung der Farben . . . 227
21.3 Ausblick . . . 229

KAPITEL 22
Erfahrungen mit Java3D . . . 230
22.1 Vorteile von Java3D . . . 230
22.2 Nachteile von Java3D . . . 231
22.3 Verbesserungsmöglichkeiten und Alternativen . . . 232

KAPITEL 23
Erfahrungen mit eXtreme Programming . . . 233
23.1 Umsetzung der Techniken . . . 233
23.2 Umsetzung der Rollen . . . 236
23.3 Fazit . . . 237

KAPITEL 24
Erfahrungen mit Eclipse . . . 239
24.1 Vorteile von Eclipse . . . 239
24.2 Nachteile von Eclipse . . . 240
24.3 Fazit . . . 241

5 Anhang

KAPITEL A

Handbuch, Quelltexte und Lizenz 243
Literaturverzeichnis 244

Abbildungsverzeichnis

4.1 Entwicklung von Geonen anhand *nonaccidental properties*, Quelle: Biederman (1987) . . . 10
4.2 Objektzerlegung in Komponenten (Geonen), Quelle: Irani und Ware (2003) . 10
4.3 von Testpersonen bevorzugte Realisierungen, Quelle: Irani u. a. (2001) . . . 12
4.4 geonbasiertes Diagramm und äqivalentes UML Diagramm, Quelle: Irani und Ware (2003) . . . 13
4.5 Walrus-Visualisierung einer Verzeichnisstruktur, Quelle: CAIDA (2004) . . . 14
4.6 Walrus-Visualisierung einer Netzinfektion durch den Code-Red Wurm, Quelle: CAIDA (2004) . . . 15
4.7 einfaches UML-Klassendiagramm (links) und zwei Walrus-Visualisierungen mit unterschiedlichen Spannbäumen. . . . 16
4.8 eine einfache CorocoCosmos-Visualisierung, Quelle: Lewerentz und Noack (2003) . . . 17
4.9 Repräsentation einer Programmlogik mit CrocoCosmos, suksessives Entfernen von unteren (detaillierteren) Schichten von oben nach unten, Quelle: Lewerentz und Noack (2003) . . . 18
4.10 ein einfaches dreidimensionales UML-Diagramm, Quelle: Radfelder und Gogolla (2000) . . . 19
4.11 UML-Sequenzdiagramm (oben 2D, unten 3D), Quelle: Radfelder und Gogolla (2000) . . . 20
4.12 Schnappschüsse einer Instanziierungsanimation, Quelle: Steimann u. a. (2002) 20

5.1 Breakpoints in Eclipse . . . 28
5.2 Ausführungskontrolle in Eclipse . . . 29
5.3 Variablendarstellung in Eclipse . . . 29
5.4 Grafische Darstellung des Integrated Comprehension Model nach von Mayrhauser & Vans. . . . 30
5.5 Grafische Darstellung des Wechsels zwischen den mentalen Modellen bei einer Testperson mit Erfahrung in der verwendeten Programmiersprache, etwas Vorwissen über den Code und fehlendem Domain-Wissen. . . . 32
5.6 Grafische Darstellung des Wechsels zwischen den mentalen Modellen bei einer Testperson ohne Erfahrung in der verwendeten Programmiersprache, ohne Vorwissen über den Code und mit umfassendem Domain-Wissen. . . . 32
5.7 Grafisches Frontend für gdb Gaylard und Zeller (2004) mit der Darstellung von Programmkonstrukten als Diagramm (links oben) und der Visualisierung von Variablenbelegungen in 2D und 3D (rechts). . . . 34

8.1 Ein beispielhafter Szenegraph . . . 64

8.2 aus Beispiel 2 resultierender Szenegraph . . . 68
8.3 aus Beispiel 3 resultierender Szenegraph . . . 70

9.1 Eclipse - Architektur . . . 73
9.2 Navigator und Package Explorer . . . 75
9.3 Java Browsing Perspective . . . 77

10.1 Die Eclipse Workbench mit Java-Editor und Views . . . 80
10.2 Eclipse Plug-In Architektur . . . 81

11.1 Schematische Darstellung der Komponenten . . . 88
11.2 Übersicht über Debug-Elemente . . . 91

14.1 Paketstruktur der realisierten Systemarchitektur (Release 1) . . . 124
14.2 Klassenstruktur des Control-Teils (Release 1) . . . 124
14.3 Klassenstruktur des Model-Teils (Release 1) . . . 125
14.4 Klassenstruktur des View-Teils (Release 1) . . . 126
14.5 Realisierte Systemarchitektur Release 1 . . . 127

15.1 Ein erster Entwurf für die Phasen des Visualisierungsprozesses (Release 2) 146
15.2 Paketstruktur der realisierten Systemarchitektur (Release 2) . . . 147
15.3 Klassen des Models (Release 2): statische Datenstruktur (links), dynamische Datenstruktur (rechts) . . . 148
15.4 Die Hauptklassen des Controlbereichs (Release 2) . . . 149
15.5 Die internen Hauptbestandteile der `ChooseClassesView` (Release 2) . . . 150
15.6 Die internen Hauptbestandteile der `Java3DView` (Release 2) . . . 151
15.7 Die gesamte Klassenstruktur mit Ausnahme der Pakete `actions`, `preferences` und `xml` (Release 2) . . . 153

16.1 Die Zustandsmaschine aus dem ISnapshotCollector . . . 165
16.2 `ArrangementAlgorithm` und `AlgorithmPanel` . . . 171
16.3 Das Panel für den Force-directed Algorithmus . . . 172
16.4 Das Panel für den Linlog Algorithmus . . . 173
16.5 Die Klassenstruktur des Visualisierungsbereiches . . . 174
16.6 Ablauf der Methode `setNextSnapshot` . . . 176
16.7 Beispiel: Zwei zu visualisierende Snapshots . . . 177
16.8 Beispiel: Objektdiagramm nach Setzen des ersten Snapshots . . . 178
16.9 Beispiel: Objektdiagramm während des Setzens des zweiten Snapshots . . . 178
16.10 Beispiel: Objektdiagramm nach Setzen des zweiten Snapshots . . . 180
16.11 Setzen eines Snapshots . . . 180
16.12 Animation eines Snapshot-Übergangs . . . 181

17.1 Der Snapshotplayer . . . 194

18.1 Auto-Beispiel 1 . . . 207
18.2 Auto-Beispiel 2 . . . 208

18.3 Auto-Beispiel 3 208
18.4 Beispielausgabe (textuell) zur Erkennung von Strukturen 209
18.5 Beispielausgabe (grafisch) zur Erkennung von Strukturen 210
18.6 Die Liste im textuellen Debugger 213
18.7 Fehlerhafte Verkettung 214
18.8 Korrekte Verkettung 214

20.1 Partitionierung durch den LinLog-Algorithmus 222
20.2 Partitionierung durch den Fruchterman-Reingold-Algorithmus 223
20.3 Force-Directed-Layout nach 1000 Iterationen 224
20.4 Force-Directed-Layout nach 100 Iterationen 225

TEIL 1

Einführung in die Projektgruppe 3DEBUG

KAPITEL 1

Einleitung

Alexander Fronk, Jens Schröder, Sören Blom

Visualisierungen sind in der Software-Technik weit verbreitet. In den meisten Fällen werden dabei zweidimensionale Grafiken eingesetzt, wie dies etwa in der UML üblich ist. Dreidimensionale Visualisierungen hingegen sind relativ selten. Das mag zum Einen an ihrer schwierigeren Handhabung liegen – dreidimensionale Grafiken lassen sich im Gegensatz zu zweidimensionalen nur schwerlich mit Papier und Bleistift umsetzen, Werkzeuge sind hier vonnöten. Zum Anderen mag dies im Fehlen einer sinnvollen Belegung der dritten Dimension im Rahmen der Erstellung von Software begründet sein – im Gegensatz zur Modellierung realer Gegenstände aus dem Alltagserleben, wie sie die Ingenieurswissenschaften kennen. Gleichwohl bieten dreidimensionale Visualisierungen eine Reihe von Vorteilen insbesondere dann, wenn man auf Interaktion mit der Visualisierung Wert legt:

- Komplexe strukturelle Zusammenhänge von Elementen – etwa das Enthaltensein in umgebenden Elementen bei gleichzeitiger Darstellung der Beziehungen zwischen diesen – sind in drei Dimensionen oft übersichtlich darstellbar und damit schnell erfassbar, ohne dabei den Kontext der umgebenden Elemente vernachlässigen zu müssen.

- In Kombination mit Bewegung im Raum können Detailinformationen ein- oder ausgeblendet werden, etwa in Abhängigkeit von der Entfernung des Betrachters zum betrachteten Element.

- Die Animation von Veränderungen betrachteter Elemente oder ihrer Beziehungen kann in drei Dimensionen plastisch dargestellt werden, was zu einem besseren Nachvollziehen dieser Veränderungen führen kann.

In einer Reihe von Arbeiten wurde am Lehrstuhl für Software-Technologie ein auf der Entwickunglungs- und Integrationsplattform Eclipse basierendes System zur dreidimensionalen Visualisierung und Manipulation von Klassenbeziehungen in Java-Programmen entwickelt. Dort wurden beispielsweise leicht voneinander zu unterscheidende Visualisierungstechniken wie Cone-Trees und Information-Cubes in Verbindung mit Federmodellen eingesetzt, um Informationen über Klassenhierarchien und Paketzugehörigkeiten darzustellen. Dabei hat sich gezeigt, dass die Dreidimensionalität einige ihrer Vorteilen auch bei der Visualisierung von Programminformationen ausspielt, insbesondere wenn man auf die Integration von Visualisierungstechniken zur Unterstützung des Programmverständnisses abzielt: So bietet sie etwa die Möglichkeit, an die UML angelehnte Klassen- und Paketdiagramme in einer einzigen Darstellung und dennoch übersichtlich zu vereinen.

Ferner hat sich gezeigt, dass dreidimensionale Repräsentationen platzsparender, übersichtlicher und einprägsamer sind. Gerade für die Entwicklung großer Programmsysteme scheint dies ein bedeutender Vorteil gegenüber üblichen zweidimensionalen Repräsentationen zu sein. Die Integration der Repräsentationen von verschiedenen statischen aber auch dynamischen Aspekten von Java-Programmen in einer einzigen Darstellung kann zudem zu einem besseren Programverständnis beitragen, da Informationen über die Laufzeit unmittelbar mit statischen Aspekten visuell in Verbindung gebracht werden können.

Üblicherweise werden Laufzeitinformationen in gängigen Entwicklungsumgebungen im Rahmen des Debugging angeboten. Hierbei wird normalerweise auf eine textuelle oder höchstens zweidimensionale Repräsentation zurückgegriffen. Für die Eclipse-Plattform existiert eine Debug-Schnittstelle. Es bietet sich daher an, diese mit einer dreidimensionalen Darstellung von zum Debugging benötigten Informationen zu ergänzen. So soll etwa eine dreidimensionale Visualisierung von den an einem Programmablauf beteiligten Objekten und Methoden sowie deren Aufrufbeziehungen angeboten werden.

Wir zielen mit unserem Ansatz langfristig auf die Anreicherung von IDEs mit dreidimensionalen Visualisierungstechniken ab, um damit Programmentwicklern Möglichkeiten anzubieten, ihr Verständnis des bearbeiteten Programmcodes und ihren Überblick über die darin verwendeten Strukturen zu erhöhen.

Die Projektgruppe 458 hat daher zum Ziel, eine für das Debugging geeignete dreidimensionale Darstellung von Laufzeitinformationen zu konzipieren und zu realisieren. Informationen sollen so in einem dreidimensionalen Raum visualisiert und angeordnet werden, dass das für ein erfolgreiches Debugging notwendige Erlangen des Programmverständnisses angemessen unterstützt wird. Dazu müssen verschiedene Techniken miteinander kombiniert werden: Zur dreidimensionalen Darstellung soll die Java3D-API Verwendung finden; zur Ermittlung der Laufzeitinformationen sollen die von Eclipse bereitgestellten JDT- und Debug-Plug-Ins verwendet werden.

KAPITEL 2

Geplantes Vorgehen

Alexander Fronk, Jens Schröder, Sören Blom

Die Projektgruppenarbeit kann grob in folgende Phasen aufgeteilt werden: Einarbeitungsphase, Anforderungsanalysephase, Konstruktionsphase mit eXtreme Programming. Folgende Abschnitte erläutern die einzelnen Phasen im Detail.

2.1 Einarbeitung, Seminarphase

In Form von Seminarvorträgen durch die Projektgruppenteilnehmer wird die Projektgruppe an die zu lösende Aufgabe herangeführt. Dies dient der Aneignung des nötigen Fachwissens. Die Einarbeitung erfolgt in folgende Themenbereiche und Problemfelder:

- Softwaretechnische Entwurfsnotationen, UML, Design Patterns
- eXtreme Programming (XP)
- Programmverständnis
- dreidimensionale Visualisierungstechniken
- Visuelle Debugger
- Visuelle Sprachen
- Eclipse
- Java3D-API

Neben einer inhaltlichen Einarbeitung hat die Projektgruppe die Gelegenheit, sich in die technische Arbeitsumgebung einzufinden und an ihre Bedürfnisse anzupassen.

2.2 Anforderungsanalyse

In dieser Phase wird die Projektgruppe die spezifischen Anforderungen an einen visuellen Debugger und die dreidimensionale Darstellung von Laufzeitinformationen herausarbeiten. Ein Ziel dabei ist es, eine geeignete Systemmetapher zu entwickeln, die als Voraussetzung für einen XP-basierten Konstruktionsprozess benötigt wird.

2.3 Konstruktion

Die Entwicklung des Debuggers folgt dem XP-Ansatz, der sich bereits bei der Durchführung der Projektgruppen 415 und 444 am Lehrstuhl für Software-Technologie bewährt hat. Auf Grund der ganzheitheitlichen Teamorientierung und der Eigenverantwortlichkeit der einzelnen Entwickler, die auch die selbstständige aber angeleitete Planung der auszuführenden Tätigkeiten umfasst, bietet sich XP als Entwicklungsprozess auch für diese Projektgruppe an.

Dem XP-Ansatz folgend ergibt sich eine inkrementelle Entwicklung mit vielen kleineren Releases, bei der die frühen Releases einen eher prototypischen Charakter haben, die dann zu einen vollständigen System führen.

2.4 Berichte

Die gesamten Arbeiten werden jeweils durch einen *Zwischen-* und einen *Abschlussbericht* dokumentiert.

Der Zwischenbericht dokumentiert die Ergebnisse des ersten Semesters, insbesondere die Anforderungsanalyse und die ersten Releases des zu erstellenden Systems.

Der Abschlussbericht wird den gesamten Projektverlauf festhalten. Die Ergebnisse der einzelnen Phasen werden vorgestellt und bewertet.

2.5 Fachgespräch

Den Abschluss der Projektgruppe bildet ein Fachgespräch, in dem die Projektgruppenteilnehmer den Fachbereich über den Ablauf und die Ergebnisse der Projektgruppe informieren. Dieses Fachgespräch wird im Rahmen des Diplomanden- und Doktorandenseminars des LS 10 stattfinden.

KAPITEL 3

Übersicht über den Abschlussbericht

Alexander Fronk, Jens Schröder, Sören Blom

Der vorliegende Bericht gliedert sich wie folgt:

- Teil 2 beinhaltet die schriftlichen Ausarbeitungen der Seminarthemen.
- Teil 3 umfasst kapitelweise die vier Entwicklungszyklen, die das entwickelte Produkt durchlaufen hat. Jeder Zyklus folgt dem in Kapitel 12 vorgestellten Prozess des eXtreme Programming und ist diesem Prozess folgend gegliedert.
- Der Bericht schließt in Teil 4 mit einer Reflexion, in der Inhalt und Ablauf der Projektgruppe kritisch beurteilt werden.
- Im Anhang sind die Inhalte der Webseite des Projekts beschrieben.

TEIL 2

Seminarphase

KAPITEL 4

Alternative Modellierungskonzepte in 3D

Daniel Vogtland

4.1 Einleitung

Zum Verständnis großer Software-Systeme werden geeignete Visualisierungsmöglichkeiten benötigt. Der eigentliche Quellcode wird schnell zu umfangreich und unübersichtlich, als dass er noch als Basis für Konzeptionen dienen könnte. Visualisierungstechniken, die den strukturellen Aufbau, Abhängigkeiten und letztendlich auch das (gewünschte) Verhalten der Software möglichst intuitiv beschreiben, werden benötigt. Nur so können Entwurf und Wartung derartiger Projekte effektiv durchgeführt werden.

Als Standard zur Visualisierung von Modellen im Umfeld der Softwaretechnologie gilt die *UML* (Unified Modeling Language). Sie stellt u. a. standardisierte Diagrammtypen, wie etwa Klassen- und Objektdiagramme, Sequenzdiagramme und Kollaborationsdiagramme, bereit. Allerdings können UML-Diagramme mit zunehmender Komplexität des dargestellten Modells schnell unübersichtlich werden, wodurch der Einblick in Struktureigenschaften des Modells verhindert werden kann. Verbesserungen können nur erfolgen, wenn die (visuelle) Informationsdichte bei gleicher Information vergrößert wird oder die visuelle Informationsverarbeitung durch eine günstigere Visualisierungsform beschleunigt wird.

Im Folgenden sollen kurz drei mögliche dreidimensionale Visualisierungen von (Aspekten von) Softwaresystemen vorgestellt werden. Dabei wird auf die Darstellung technischer Details und Implementierungsbereiche verzichtet, sondern nur die grundsätzlichen Ideen und beispielhafte Anwendungsmöglichkeiten werden vorgestellt. Zunächst wird auf die Grundzüge von Geon-basierten Visualiserungen eingegangen. Danach werden Graph-basierten Darstellungen anhand zweier Beispielsysteme dargestellt. Abschließend wird noch kurz eine einfache dreidimensionale Erweiterung von UML vorgestellt.

4.2 Geon-basierte Darstellungen

Visuelle Repräsentation sollte so gestaltet sein, dass sie menschliche Erkennungsprozesse nutzt. Eine Visualisierung zu „verstehen“ bedeutet, das dargebotene Bild (die aktuelle Sicht) in Elemente zu zerlegen und diese gegebenenfalls zueinander in Beziehung zu setzen. Um

mit den Elementen Informationen verknüpfen zu können, müssen diese eindeutig identifiziert bzw. klassifiziert werden. Anders ausgedrückt besteht die Aufgabe für den Betrachter darin, (bekannte) *Objekte* in dem Bild zu erkennen.

4.2.1 Menschliches Bild-Verstehen: Die Biederman Theorie

Biederman hat in seiner Arbeit Biederman (1987) die Grundsteine für die Konzeption eines dreidimensionalen Diagrammtyps gelegt, der diesem Ansatz nachgeht. Seiner Meinung nach sind Farbe, Intensität und Textur eines Objektes eher sekundäre Attribute bezüglich der Identifikation. Auschlaggebend für die Identifikation ist vor allem die Gestalt bzw. Form, dies ist also ein primäres Attribut. Objekte sollten möglichst unabhängig vom Standort des Beobachters sein. Also sollten sie durch relative Koordinaten und Größenverhältnisse definiert sein, statt durch absolute Werte. Probleme bei fehlender Informationen (z.B. wenn das Objekt teilweise verdeckt ist) werden durch Inferenzmechanismen gelöst. Dies bedeutet, dass das zu analysierende Element dem Objekt mit der größten Übereinstimmung zugeordnet wird. Zusätzliches Wissen und wahrnehmungspsychologische Gesetze fließen in den Erkennungsprozess mit ein. Auf diese Weise ist auch des Abstrahieren von Karrikaturen o. ä. möglich.

Biederman gibt schematisch einen klar strukturierten Ablauf für die Erkennung eines Objekts an. Zunächst werden Kanten extrahiert, beispielsweise durch Unterscheidung verschiedener Texturen. Falls genügend Informationen vorhanden sind, werden konkave Regionen ermittelt. Diesen werden bestimmten Komponenten (Teilobjekte) zugeordnet. Sind nicht genügend Informationen vorhanden, wird stattdessen auf so genannte *nonaccidental properties* geprüft. Es handelt sich dabei um fünf Eigenschaften, die im zweidimensionalen Bild wahrgenommen in die dreidimensionale Vorstellung übertragen werden. So führt z. B. die Erkennung einer geraden Linie zu der Annahme, dass die entsprechende/n Kante/n des erzeugenden dreidimensionalen Objektes ebenfalls einen geraden Verlauf hat/haben. Diese fünf Eigenschaften zeichnen sich durch relative Unempfindlichkeit gegenüber Rausch- und (leichten) Verzerrungseffekten so wie der Perspektive des Betrachters aus. Vielleicht können nun direkt Komponenten entdeckt werden. Es besteht aber auch die Möglichkeit, dass sich neue Kanteninformationen ergeben (z. B. beim Erkennen von Symmetrie), dann wird zum Stadium der Kantenextraktion zurückgekehrt. Sind schließlich Komponenten gefunden, werden sie und ihre Kombinationen mit bekannten Objekten abgeglichen. Das Objekt mit grösster Übereinstimmung (eventuell unter Zuhilfenahme von zusätzlichem Wissen) wird identifiziert.

Diese Komponenten wurden von Biederman als *Geonen* bezeichnet,- einfache dreidimensionale Gebilde aus denen jedes mögliche Objekt zusammengesetzt werden kann. Ausgehend von primitiven Grundkörpern entwickelte Biederman nach diesem Ansatz durch unterschiedliche Ausprägungen der *nonaccidental properties* einen Geonensatz von 36 Geonen (Abbildung 4.1).

Jedes Objekt wird bei der Verarbeitung lückenlos in Komponenten aus diesem Satz zerlegt (Abbildung 4.2). Allerdings wird ein Objekt nicht nur durch die enthaltenen Geonen bestimmt, auch die Anordnung und Größenverhältnisse sind entscheident. Sekundäre Attribute wie Textur oder Farbe werden zur Informationserweiterung genutzt, beispielsweise zur

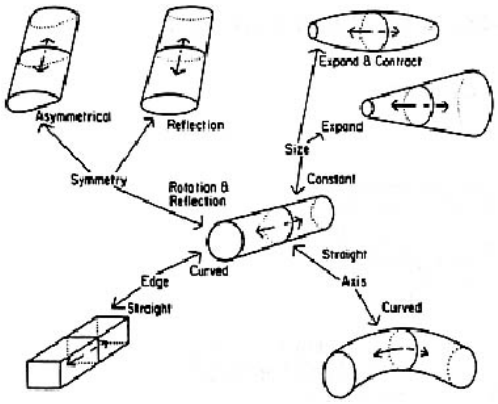


Abbildung 4.1.: Entwicklung von Geonen anhand *nonaccidental properties*, Quelle: Biederman (1987)

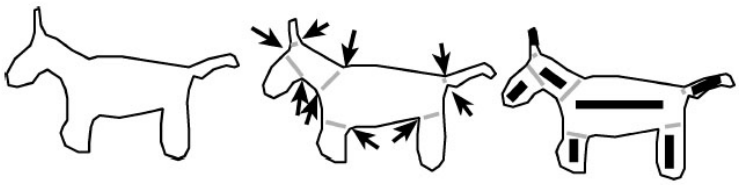


Abbildung 4.2.: Objektzerlegung in Komponenten (Geonen), Quelle: Irani und Ware (2003)

Unterscheidung von zwei Objekten mit gleicher Form.

4.2.2 Geonen und UML

In ihrer Arbeit Irani u. a. (2001) griffen die Autoren die Theorie von Biederman auf. Ihre Zielsetzung war eine verbesserte Visualisierung von Softwaresystemen, die jedoch genauso mächtig wie UML-Diagramme sein sollte. „Verbesserung“ bedeutet hier vor allem besseres intuitives Verständnis. Ausgangspunkt waren die bestehenden UML Notationen, wie sie in Klassen- und Objektdiagrammen vorkommen.

Basierend auf Biedermans Arbeit mit einem zusätzlichen Einbettungs- bzw. enthalten-sein-Aspekt (containment) formulierten die Autoren folgende Darstellungs-Regeln:

1. Farbe und Textur (Color and Texture) sind Oberflächeneigenschaften von Geonen, die nur eine sekundäre Rolle bei der Objektklassifikation spielen. Sie können beim kognitiven Prozeß eine Hilfe darstellen, spielen aber nur eine untergeordnete Rolle bei der ersten und automatisiertesten Erkennungsphase.

2. Vertikalität (Verticality): Geon A kann ÜBER, UNTER oder NEBEN Geon B sein.

3. Zentrierung (Centering): Objekte können zentralisiert oder dezentralisiert sein. Beispielsweise befinden sich menschliche Beine links und rechts an der Unterseite des Torsos (dezentralisiert), menschliche Arme sind links und rechts seitlich oben am Torso angeordnet (zentralisiert).

4. Verbindung relativ zu Verlängerung (Connection relative to elongation): Die meisten Geonen werden verlängert. Ob sich die Verbindung zu einem anderen Geon an einer langen oder kurzen Fassette befindet, hat wichtige Wahrnehmungs-relevante Bedeutung. Als Beispiel führen die Autoren die Unterscheidung von Menschen und vierbeinigen Tieren an.

5. Relative Größe (Relative Size): Geon A kann größer, gleich groß oder kleiner als Geon B sein.

6. Einbettung (Containment): Im Kontext eines Softwaresystems müssen auch Objekte, die von größeren Komponenten eingeschlossen sind, identifiziert werden können. Diese Enthalten-sein Beziehung ist hierarchisch. Solche Mechanismen benötigen bei der Darstellung die Nutzung von Transparenz.

Zunächst versuchten die Autoren verschiedene Konzepte der Modellierung darzustellen: Generalisierung (A ist-ein B), Abhängigkeit (A ist abhängig von B), Beziehungsstärke (manche Beziehungen sind stärker als andere), Multiziplitäten von Beziehungen / Ordinalität (ein A kann zu vielen B in Beziehung stehen) und Aggregationen (A besitzt-ein B). Für jedes Konzept wurden alternative Visualisierungsformen entwickelt. Testpersonen sollten anschließend die besten Darstellungsformen für diese Konzepte ermitteln. Der Testpersonenkreis bestand aus zwei Gruppen: Experten (Erfahrung mit Software-Diagramm-Notationen, teilweise UML) und Laien (keine Erfahrung mit Software-Visualisierung). Abbildung 4.3 fasst alle ausgewählten Visualisierungen zusammen. Nach diesen Versuchsreihen formulierten Inrani, Tingley & Ware unter Einbeziehung früherer Ergebnisse und externer Arbeiten folgende Liste von Regeln zur wahrnehmungsunterstützenden Diagrammgestaltung:

- Die Hauptelemente eines Systems sollten durch Geonen dargestellt werden.
- Verbindungen zwischen Elementen sind Verbindungen zwischen Geonen, Datenstrukturen werden durch ein Geonenskelett (Irani, Tingley & Ware bezeichnen mit Geonenskelett die Kombination von Geonen, die ein bestimmtes zusammengesetztes Element erzeugt) dargestellt.
- Untergeordnete Sub-Komponenten werden als Geonanhänge repräsentiert: kleinere Geonen werden an größeren angebracht.
- Geon Darstellungen sollten schattiert dargestellt werden, damit ihre 3D Gestalt besser zu erkennen oder überhaupt erst unterscheidbar ist (gleiche Silhouette).
- Sekundäre Attribute von Elementen und Beziehungen werden durch Farbe, Textur und Symbolen auf der Oberfläche eines Geons dargestellt.
- Alle Geonen sollten vom gewählten Beobachtungspunkt aus sichtbar sein.

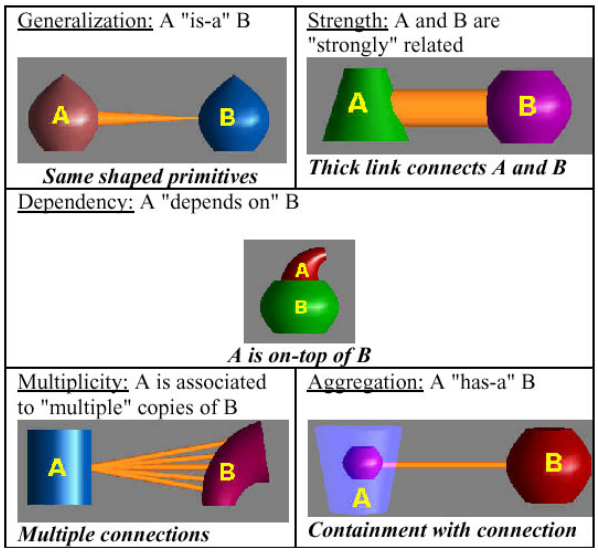


Abbildung 4.3.: von Testpersonen bevorzugte Realisierungen, Quelle: Irani u. a. (2001)

- Das Geondiagramm sollte sich initial an der orthogonal zur Blickrichtung liegenden Ebene orientieren.
- Verbindungspunkte von Geonen sollten deutlich sichtbar gemacht werden.
- Ähnlichkeit, Generalisierung: Geonen mit gleicher geometrischer Komposition oder Form können Elemente gleichen Typs ausdrücken.
- Gravitation: Wenn Geon A sich über Geon B befindet, suggeriert dies, dass A von B unterstützt wird. Außerdem beeinflusst Gravitation den Eindruck, ob Strukturen stabil oder instabil sind.
- Einbettung zeigt, dass Geon A in Geon B enthalten ist. Syntaktisch kann dies dargestellt werden, indem eine innere Komponente an dem gleichen Geon außerhalb festgemacht wird.
- Ordinalität: Mehrfache Assoziationen zwischen zwei Elementen können am besten durch mehrfache Befestigungen visualisiert werden.
- Kräftigere Verbindungen suggerieren höhere Verbindungsstärken.
- Sequenz: Geonen, die in einer Linie angeordnet sind, werden eine Metapher für eine Operationskette oder andere lineare Strukturen.
- Symmetrie: Manche Informationsstrukturen besitzen Symmetrie und symmetrische Geon-Anordnungen sollten dazu genutzt werden, dies zu zeigen.

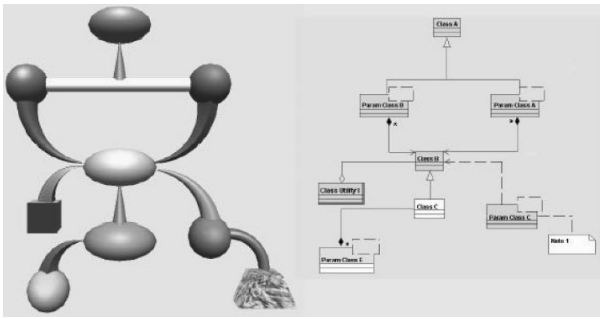


Abbildung 4.4.: geonbasiertes Diagramm und äqivalentes UML Diagramm, Quelle: Irani und Ware (2003)

- Zentralisierung: Hat eine Komponente zentrale Bedeutung für eine Struktur, so kann dies durch ihre Position und Anordnungen von Wechselbeziehungen verdeutlicht werden.
- Größe: Größere Komponenten können übergeordneten Bereichen zugeordnet werden.

In Irani und Ware (2003) beschreiben Irani & Ware Versuche, welche die Vorteile der geonbasierten Darstellung bestätigen. Der verwendete Geonenensatz bestand aus 24 Elementen. Es wurden Erinnerung an Strukturen und Auffinden von Teilstrukturen getestet (Abbildung 4.4). Dabei wurden Geondarstellungen neben UML-Diagrammen auch mit zweidimensionalen Geon-Silhouetten Diagrammen verglichen. Dieser Ansatz lieferte somit auch Vergleichsergebnisse zwischen zweidimensionalen und dreidimensionalen Darstellungen, losgelöst von der UML-Notation (andernfalls könnte man annehmen, die UML-Notation sei einfach nur „schlecht“). Geondiagramme führten durchweg zu den besten Ergebnissen.

In Casey und Exton (2003) wird eine in Java 3D implementierte Anwendung angesprochen. Diese unterstützt auch eine automatisierte Komposition der Elemente. Dies ist nach Dwyer eine notwendige Anforderung, da das Navigieren im dreidimensionalen Raum nicht so einfach ist wie in einem zweidimensionalen Editor, und der entsprechende Zeitaufwand bei der Modellierung ohne Automatisierung nicht vertretbar ist Dwyer (2001).

4.3 Graph-basierte Visualisierungen

Die Struktur und einzelne Momentaufnahmen eines Softwaresysteme können als *Graph* aufgefasst werden. Ein Graph besteht aus Knoten und (gerichteten oder ungerichteten) Verbindungen zwischen diesen Knoten. Diesen Knoten und Verbindungen (Kanten) können Eigenschaften zugeordnet werden, wie Farbe, Form oder auch Klassenzugehörigkeit. Im Kontext eines Softwaresystems können mögliche Knotentypen beispielsweise Klassen und Objekte, eventuell aber auch Methoden und Ressourcen sein. Kantentypen könnten „ist-ein“, „benutzt“, „führt aus“ oder „Instanz von“ sein.

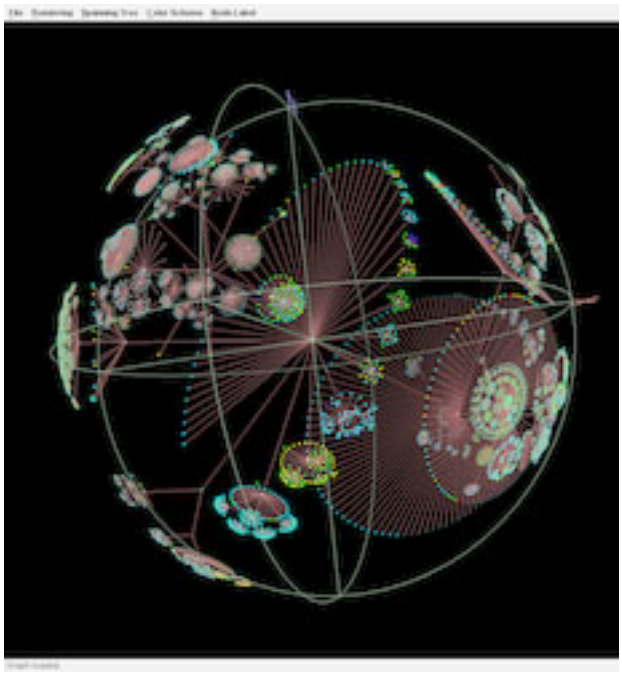


Abbildung 4.5.: Walrus-Visualisierung einer Verzeichnisstruktur, Quelle: CAIDA (2004)

Das Hauptziel besteht darin, die Darstellung so zu optimieren, dass sie für den Betrachter wichtige Beziehungen und strukturelle Merkmale in den Daten aufdeckt. Dies kann beispielsweise das *Clustering* beinhalten. Cluster sind hier Partitionen von Knoten, die z. B. dadurch gekennzeichnet sind, dass innerhalb eines Clusters alle Knoten stark miteinander (durch Kanten) verbunden sind, aber nur wenige Verbindungen hinausführen. Hierbei müssen Möglichkeiten gefunden werden, den Berechnungsaufwand zu reduzieren.

4.3.1 Walrus: Allgemeine Graphdarstellungen

Walrus ist ein in Java (mit Java 3D) programmiertes lauffähiges Visualisierungstool für Graphen. Es ist nicht auf Softwarevisualisierung spezialisiert, sondern dient der Darstellung von allgemeinen Graphen. Eigentlich handelt es sich um gerichtete Graphen, dies ist in der Visualisierung jedoch nicht erkennbar. Abbildungen 4.5 und 4.6 zeigen zwei Beispiele. Statt eines einfachen dreidimensionalen Raums wird *hyperbolische Geometrie* benutzt. In der hyperbolischen Geometrie entspricht ein Quadrat einer Kugel mit gleichem Mittelpunkt (vgl. Herman u. a. (2000); Munzner (2000)). Zusätzlich wird ein Fischauge-Effekt (vgl. Furnas (1986)) realisiert. Er bewirkt, dass die Detailstufe mit wachsendem Abstand vom Fokusmittelpunkt verringert wird. Der Nutzer kann die Darstellung interaktiv beeinflussen (z. B. Navigation und Anzeigeoptionen).

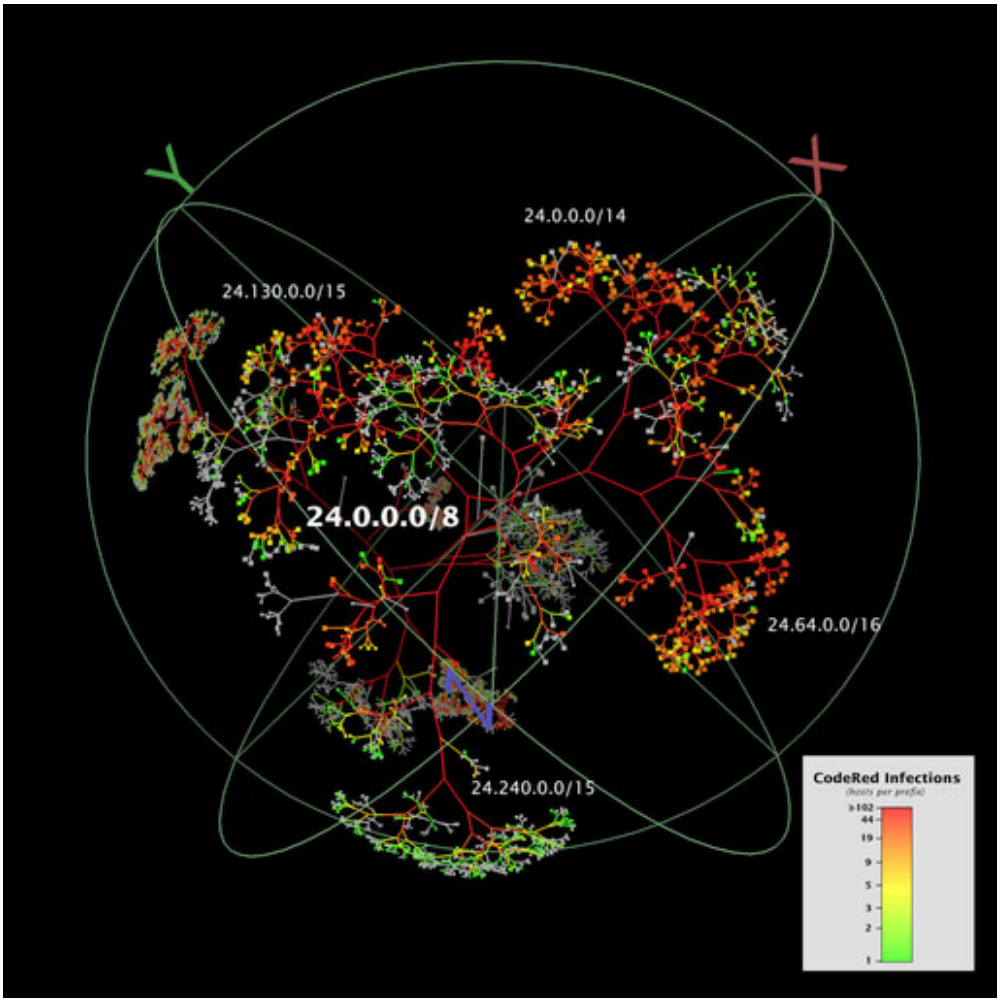


Abbildung 4.6.: Walrus-Visualisierung einer Netzinfektion durch den Code-Red Wurm, Quelle: CAIDA (2004)

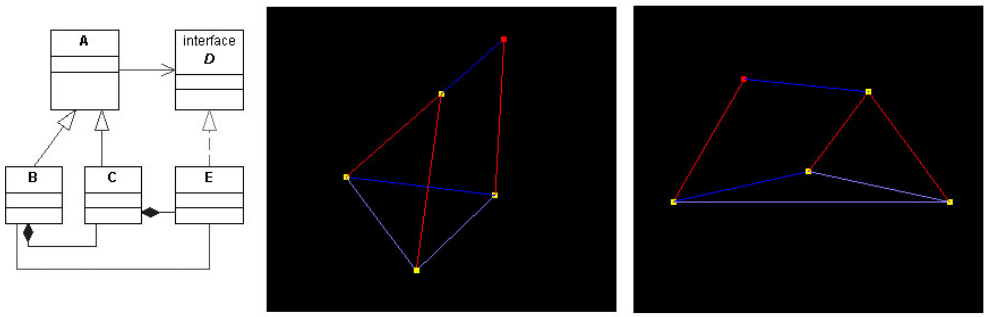


Abbildung 4.7.: einfaches UML-Klassendiagramm (links) und zwei Walrus-Visualisierungen mit unterschiedlichen Spannbäumen.

Neben der Definition von Knoten und gerichteten Kanten können in Walrus auch Attribute für diese individuell definiert werden, welche durch Walrus auslesbar sind. Diese Attribute haben in der Regel keine semantische Bedeutung, allerdings können sie zur Farbgebung benutzt werden. Auf diese Weise können Softwarevisualisierungsaspekte (welche ja beispielsweise in unterschiedlichen Kantentypen bestehen können; besitzt-ein vs. ist-ein) verwirklicht werden (Abbildung 4.7).

Um den Berechnungsaufwand zu reduzieren, wird ein *Spannbaum* (spanning tree) über die Eingabeknoten vorausgesetzt. Visualisierungen von Baumstrukturen haben erheblich geringere Komplexität Herman u. a. (2000). Ausgelassende Kanten des Originalgraphen werden anschließend einfach wieder eingefügt (hier liegt auch der Nachteil dieser Methode,- Fehleranfälligkeit). Weitere Informationen zu Walrus sind auf der CAIDA-Website CAIDA (2004) zu finden.

4.3.2 CrocoCosmos: Software-spezifische Graphen

Bei *CrocoCosmos* handelt es sich um eine Komponente eines experimentellen Tools zur Analyse eines Softwaresystems. Dieser Abschnitt stützt sich bei der Beschreibung des Systems auf den Artikel Lewerentz und Noack (2003).

CrocoCosmos dient der Visualisierung von Graphen, die zuvor aus Softwarestrukturen ermittelt wurden. Die Eingabe ist der gesamte Quellcode, der von der Applikation analysiert wird. Das Ergebnis ist ein hierarchischer Graph, der verschiedene Detailstufen ermöglicht. Die Programmentitäten wie Methoden, Attribute, Klassen, Dateien oder Subsysteme stellen Knoten dar. Die Containment-Hierarchie (z. B. Package - Klasse) wird durch die Graphstruktur behandelt, welche zusammenfassbare Untergraphen ermöglicht. Weitere Relationen wie „benutzt“, „ruft auf“ oder „ist-ein“ werden durch gerichtete Kanten erfasst. In den Knoten werden zusätzliche Informationen abgespeichert. Diese beruhen auf Metriken, welche Eigenschaften des Softwaresystems auf Zahlenwerte abbilden. Ein Beispiel für eine solche Eigenschaft besteht in der Anzahl von ein- und ausgehenden Kanten bezüglich der „besitzt-ein“-Beziehung. Für jede Hierarchhieebene sind dabei individuelle Metriken definierbar. Die in den Knoten

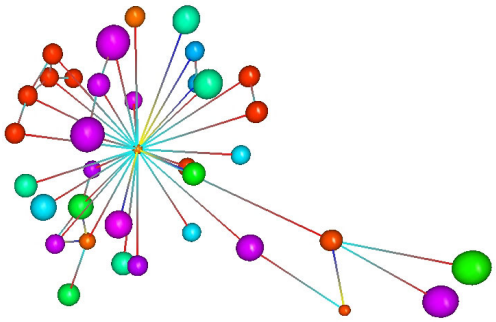


Abbildung 4.8.: eine einfache CorocoCosmos-Visualisierung, Quelle: Lewerentz und Noack (2003)

gespeicherten Metrikwerte werden bei der Visualisierung benutzt.

Knoten werden als Volumina (z. B. Quader oder Kugeln), Kanten als farbige Linien mit Farbverlauf entsprechend der Start- und Zielknoten dargestellt. Formzuordnungen (z. B. Kugeln für Klassen) und Knotenfärbungen (es können alle Subsysteme eines Systems die gleiche Farbe haben) sind durch den Nutzer beeinflussbar. Weitere visuelle Eigenschaften können mit Hilfe entsprechender Metriken von den Merkmalen der zu visualisierenden Softwareentität abhängig gemacht werden, beispielsweise kann der Durchmesser einer Klassen-Kugel durch die Anzahl der zu dieser Klasse gehörenden Methoden bestimmt sein. Der Nutzer hat weiterhin die Möglichkeit, Hierarchieebenen ein- oder auszublenden und frei im dreidimensionalen Raum zu navigieren.

Bei der Visualisierung eines solchen Graphen durch CrocoCosmos wird auf das Konzept der Energiemodelle zurück gegeriffen. Es handelt sich um Bewertungsfunktionen für Graphvisualisierungen, was zu einem Minimierungsproblem führt. In CrocoCosmos wird das *LinLog Energy Model* benutzt Lewerentz und Noack (2003). Auf dieses wird näher in Noack (2003a) eingegangen (weitere Energiemodelle werden beipielsweise in Dwyer (2001); Noack (2003b) beschrieben).

Ein Beispiel für eine schematische CrocoCosmos-Visualisierung ist in Abbildung 4.8 dargestellt. Abbildung 4.9 veranschaulicht die Möglichkeit der Hierarchieebenennutzung. Weitere Informationen zu CrocoCosmos sind auf Noack (2004) zu finden.

4.4 Animiertes 3D UML

Die UML hat sich als Visualisierungsnotation fest etabliert. Die vielleicht naheliegenste Idee, Softwarevisualisierungen zu verbessern, könnte also im Gegensatz zum Geonenansatz auch darin bestehen, UML „einfach“ um eine dritte Dimension zu erweitern (3D UML). Die zweidimensionalen Notationenselemente werden durch naheliegenden Transformationen zu drei-

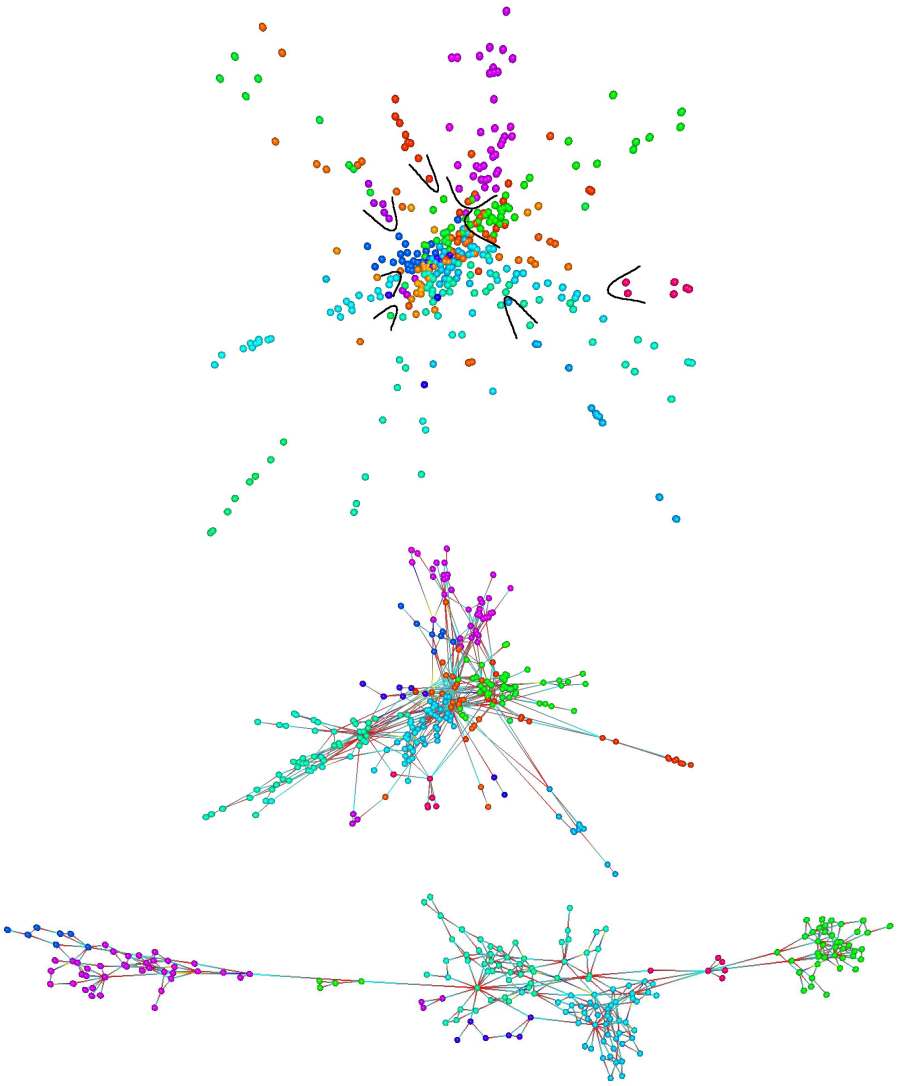


Abbildung 4.9.: Repräsentation einer Programmlogik mit CrocoCosmos, suksessives Entfernen von unteren (detaillierteren) Schichten von oben nach unten, Quelle: Lewerentz und Noack (2003)

dimensionalen Objekten: Assoziationslinien zu Röhren, Klassenrechtecke zu flachen Boxen

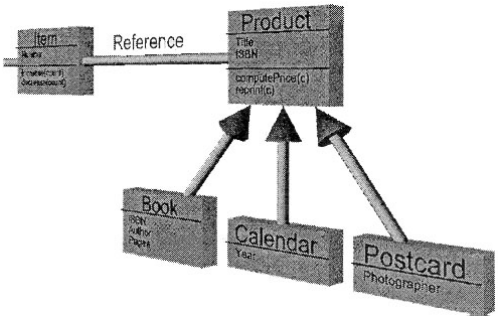


Abbildung 4.10.: ein einfaches dreidimensionales UML-Diagramm, Quelle: Radfelder und Gogolla (2000)

usw. (vgl. Abbildung 4.10).

Gogolla & Radfelder weisen in ihrer Arbeit Radfelder und Gogolla (2000) auf einen entscheidenen Vorteil hin, der durch die zusätzliche Dimension in Verbindung mit Nutzerinteraktion entsteht. So werden 2D UML-Diagramme schnell sehr komplex. Interssieren nur die Beziehungen zwischen zwei an entgegengesetzten Rändern gelegenen Objekten, so sind diese meist nicht mehr zu erfassen. Die dritte Dimension kann jedoch diesem Problem entgegenwirken. Dazu werden die entsprechenden Elemente in den Vordergrund geholt. Die gewünschten Beziehungen sind vergrößert und überlagern unwichtige Diagrammelemente, welche ihrerseits unscheinbar und nicht störend im Hintergrund verbleiben.

Doch auch die zusätzliche Dimension allein ist gewinnbringend. Wird eine Dimension zur Darstellung von dynamischen Aspekten eines Softwaresystems benötigt, verbleibt beim 2D UML nur eine Dimension für verschiede Elemente, wie beispielsweise interagierende Objekte. Beim 3D UML können die Elemente dann statt nur nebeneinander auch hintereinander angeordnet sein. Ein gutes Beispiel sind Sequenzdiagramme (Abbildung 4.11).

Eine weitere Verbesserung der Visualisieriung besteht in der Nutzung von Animation. So können einerseits platzsparend dynamische Aspekte visualisiert werden (Abbildung 4.12), anderseits wird in Radfelder und Gogolla (2000) aber auch auf die Möglichkeit verwiesen, dynamische Aspekte vor statischen ablaufen zu lassen. So könnte beispielsweise im Hintergrund ein Klassendiagramm abgebildet sein, aus dem heraus im aktuellen Programmablauf neu instanziierte Objekte auf den Betrachter „zugeflogen“ kommen, um im Vordergrund zu interagieren.

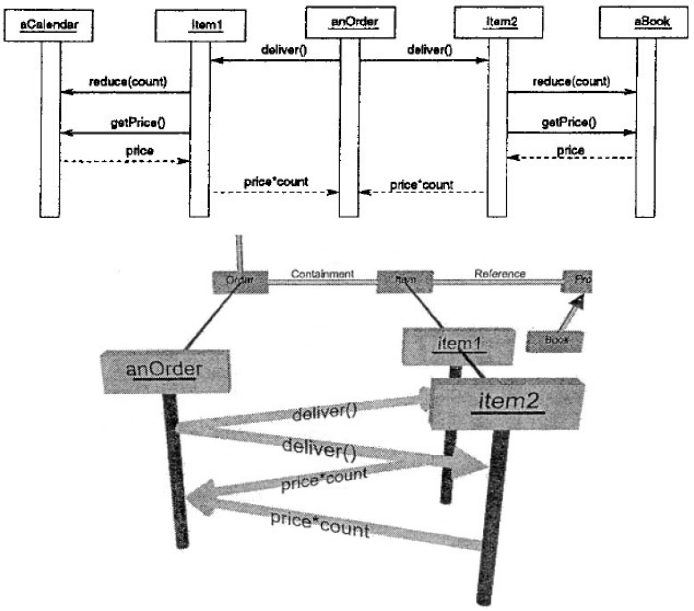


Abbildung 4.11.: UML-Sequenzdiagramm (oben 2D, unten 3D), Quelle: Radfelder und Gogolla (2000)

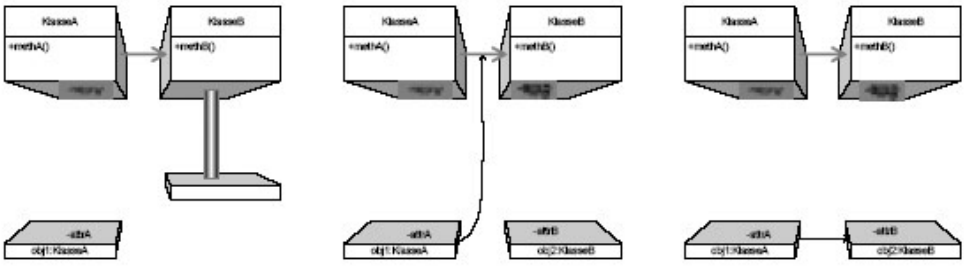


Abbildung 4.12.: Schnappschüsse einer Instanziierungsanimation, Quelle: Steimann u. a. (2002)

4.5 Ausblick

Alle drei vorgestellten Konzepte verdienen einer weiteren Betrachtung. Gerade im Bereich der Graph-basierten Darstellung gibt es eine Vielzahl von Veröffentlichungen. Doch auch Geondiagramme haben im Bereich der Softwarekonstruktion hohe Beachtung gefunden, was wohl u. a. auf ihre Nähe zu den klassischen UML-Diagrammen zurückzuführen ist. Es gibt sogar eine Überschneidung dieser beiden Visualisierungskonzepte. Schließlich kann man Geondarstellungen als eine Art Spezialfall von Graphvisualisierung auffassen. Man könnte den Knotentypen (z. B. Klassen) und Kantentypen (z. B. Generalisierung oder Assoziation) entsprechende Geonendarstellungen zuordnen, dann handelt es sich bei einem Geondiagramm um eine spezielle Darstellung eines (z. B. Klassenbeziehungs-) Graphen.

Graphvisualisierungen sind zwar nicht unbedingt so intuitiv wie Geondiagramme, doch ist meist eine kompaktere Darstellung als bei Geondiagrammen möglich. Allgemein scheinen die Vorteile von Graphdarstellungen im Erfassen von (oftmals globaleren) Zusammenhängen zu liegen, während die Vorteile der Geondarstellung eher im intuitivem Verständnis von Beziehungen bestimmter Elemente liegen. Graphvisualisierungen sind somit wohl hauptsächlich für die Softwareanalyse einzusetzen, während Geondiagramme eher der Entwicklung dienlich sind. Dies ist jedoch immer von der aktuellen Aufgabe abhängig, ein Pauschalurteil wird wohl kaum möglich sein.

Die Stärke von animiertem 3D UML liegt dagegen eher in der „Einfachheit“. Es wird auf Bekanntes gesetzt und somit ergibt sich ein Wiedererkennungseffekt. Sollte UML jedoch durch eines der ersten beiden Konzepte ersetzt werden, so wird sicherlich auch dieser Ansatz schnell verworfen werden. Hat man sich mit Geondiagrammen angefreundet, sind animierte Geonen animierten 3D UML-Elementen bestimmt ebenbürtig, wenn nicht sogar überlegen. Bis dahin stellt 3D UML allerdings eine sinnvolle Bereicherung dar.

Zur Darstellung dynamischer Softwareaspekte können in allen drei Fällen bei umfangreicheren Projekten allenfalls vorgefertigte Animationen, die Annahmen über oder Aufzeichnungen des Verhaltens der Software visualisieren, genutzt werden. Eine wirkliche Realzeit-Darstellung (wie sie beispielsweise beim Debugging nützlich sein könnte) ist aufgrund des hohen Rechenaufwands zur Zeit wohl nicht möglich.

KAPITEL 5

Debugging-Techniken

Michael Striewe, Sebastian Vastag

5.1 Debugging-Techniken

5.1.1 Der Begriff Debugging

> to debug (engl.) für entwanzen

Der Begriff Debugging ist eine moderne Legende aus den Tagen der ersten Computer. Die Geschichte soll sich auf den 9. Dezember 1947 zurückdatieren lassen. Am diesen Tag fanden Techniker einen Fehler innerhalb des Rechners Harvard Mark II. Der Verursacher war eine Motte, welche sich zwischen zwei Relais-Kontakten verfangen hatte und somit die Leitung auf logisch 0 legte. Diese Vorfälle wiederholten sich, weil der archaische Computer seine Halle regelmäßig so stark aufheizte, dass die Klimaanlagen versagten und die Fenster geöffnet werden mussten. Dadurch suchten sich insbesondere Nachts Motten und andere Insekten ein warmes Plätzchen. Die Techniker bezeichneten später das Aufsammeln der Insekten aus dem Innereien des Computers als „Debugging“. Wikipedia

5.1.2 Heutiger Debugging-Begriff

Mit der Zeit wurden die Computer immer kleiner (aber nicht kühler), Motten können sie nicht mehr gefährden. Der Begriff „Debugging“ hat jedoch die Jahrzehnte überlebt und bezeichnet heute immer noch das Suchen und Entfernen von Fehlern aus Systemen. Debugging und „Bug“ sind in der Informatik wegen dieser Historie kein fest definierter Begriff, eine mögliche Beschreibung für „Bug“ ist:

> Bug: Unerwünschte Verhaltensweise oder Zustandsänderung von Hard- oder Software.

An dieser Definition ist insbesondere „unerwünscht“ schwierig, da subjektiv. Oft genug hörte man aus Marketingabteilungen die Entschuldigung „It’s not a bug, it’s a feature!“
Unerwünschtes Verhalten eines Computersystems kann entstehen durch Telles und Hsieh (2001):

Speicher- oder Resourcenmangel : Durch falschen Umgang mit Alloziierung und Deallozііerung von Speicher und Resourcen kann ein System zur Verlangsamung oder zum Stillstand gebracht werden. Viele Sprachen unerstützen oder erzwingen die manuelle Verwaltung von Speicher, nicht freigegebener Speicher geht dem System bis zum Ende des Programmes verloren. Auch in Sprachen mit automatischer Speicherverwaltung (Fortran, Java, VB, etc.) können Speicherlecks entstehen. Speicher wird hier erst freigegeben, wenn ein Objekt nicht mehr referenziert wird. Durch ungeschickten Programmierstil mit Halten aller Referenzen lässt sich auch hier die Katastrophe herbeiführen.

logische Fehler sind die häufigste Form eines Bugs, entstanden durch Denkfehler des Programmierers selbst. Der erdachte Algorithmus löst nicht das Problem bzw. überhaupt kein Problem, oder es werden einfach die falschen Methoden ausgeführt. Leider schützt vor Fehlern im sematischen Bereich kein Compiler oder Debugger, einer Maschine können sie nicht auffallen.

Fehler im Quelltext : Ein Programm kann syntaktisch korrekt sein und (abgesehen von logischen Fehlern) trotzdem etwas Unerwartetes tun. Der Programmierer ist dann in eine der syntaktischen Fallen der verwendeten Sprache getreten. Dies können Typfehler bei Variablenzuweisungen in schwach getypten Sprachen sein. Manche Sprachkonstrukte sind auch noch anfällig gegen Tippfehler, beliebte Fehler in Java sind z.B.

- die Verwechselung von Zuweisung „=“ und boolschen Vergleich „==“ innerhalb von boolschen Ausdrücken. Eine erfolgreiche Variablenzuweisung ist leider auch `true`.
- oder das falsche Zuweisen von Werten an globale Variablen innerhalb von Konstruktoren:

```
String name;

public MyClass(String name) {
  name = name;  //falsch
  this.name = name;  //richtig
}
```

Pointerfehler entstehen durch Zeiger auf falsche oder nicht initialisierte Objekte und Speicherbereiche. Entweder wurde das Objekt selbst zerstört oder der Pointer wurde durch ein anderes Missgeschick überschrieben. Es gibt drei Kategorien von Pointerfehlern. Zeiger auf nicht initialisierte Objekte sind Nullpointer. Andererseits kann der Speicherbereich, auf den der Pointer zeigt, wieder zur andersweitigen Verwendung freigegeben worden sein. Vielleicht existiert in diesen Speicher noch das ursprüngliche Objekt, vielleicht aber auch nicht. Drittens wären da noch die ungültigen Pointer. Im Gegensatz zu den beiden ersten Typen zeigen sie auf ein initialisiertes Objekt im dafür reservierten Speicherbereich. Nur ist dies nicht das erwartete Objekt. Unter Umständen lassen sich sogar so Funktionen aus dem Objekt aufrufen, was später zur Katastrophe führt. Diese letzte Gruppe von Pointerfehlern ist am schwierigsten zu finden, insbesondere wenn vom falschen Objekt weitere Pointer ausgehen.

Fehler durch Nebenläufigkeit durch gleichzeitigen Zugriff auf eine gemeinsame Variable/Resource. Diese Fehler können sporadisch auftreten, da viele Betriebssysteme und Laufzeitumgebungen mit nichtdeterministischen Schedulern arbeiten.

und unzählig viele weitere Gründe.

Neben diesen Programmierfehlern haben Entwicklungsteams auch mit ganz anderen Problemen zu kämpfen:

Designbugs resultieren aus schlechter Planung des Zusammenspiels einzelner Softwarekomponenten. Diese Fehler lassen nicht das Programm abstürzen, sie werfen aber den ganzen Programmierstil und den Einsatz der (falschen) Designpatterns über den Haufen. Zum Beispiel kann eine Referenz auf ein bestimmtes Objekt benötigt werden, die Kapselung der gesamten Softwarekomponente verhindert aber den direkten Zugriff. Hässlicher, unsauberer Programmcode mit „Quickhacks“ ist die Folge.

Planungsbugs : Das gesamte Projekt entspricht überhaupt nicht dem, was sich der Kunde vorgestellt hat. Offensichtlich war die Kommunikation zwischen Auftragsgeber und Dienstleister mangelhaft.

Dokumentationsfehler sind eine Art von Fehlern, an denen man völlig unschuldig sein kann, da sie oft von außerhalb einstreuen. Fremder Code und Bibliotheken sind oft falsch dokumentiert und erfüllen so die Erwartungen in ihre Funktion nicht.

Im Folgenden lassen wir diese Projektfehler außen vor und beschränken uns auf reine Softwarefehler.

5.1.3 Klassen von Bugs

Fehler in Software sind recht individuell, daher möchten wir hier nur eine grobe Klassifikation vorstellen. Sie wird uns später bei der Betrachtung der Eigenschaften von Debuggingtechniken eine genauere Bewertung ermöglichen.

Bohrbugs

Einfach zu reproduzierende und/oder bei jedem Programmdurchlauf wiederkehrende Fehler fallen in die Kategorie Bohrbugs [1]. Durch gründliches Testen am System sollten sie im fertigen Produkt nicht mehr auftauchenHeisenbugs & Bohrbugs (2003).

Heisenbugs

Besonders gemein sind Fehler, die durch den Debugger selbst verdeckt werden, d.h. der Debugger verändert das System derart, dass der Fehler ausbleibt. Tritt der Fehler auch ohne Debugger nicht regelmäßig auf, so überlebt er oft den Auslieferungstermin der Software und wird erst vom Kunden entdeckt. [2]

[1] Namensgeber ist das Bohrsche Atommodell, welches recht einfache, kugelförmige und in sich geschlossene Atome annimmt

[2] Der Name ist von der Unschärferelation Werner Heisenbergs abgeleitet. Eine ihrer Aussagen ist, dass Ort und Geschwindigkeit eines Quantenteilchens nicht gleichzeitig gemessenen werden können, da eine Messung jeweils die andere Größe verfälscht.

Regressionsbugs

Bereits als behoben geglaubter Bug, der durch mysteriöse Umstände in einer späteren Programmversion wieder zu Tage tritt.

5.1.4 Schädlingsbekämpfungsmittel

Statt Bugs zu beheben sollte man sich zuerst darüber Gedanken machen, wie man sie vermeiden kann. Dies lässt sich schon durch Planung und Entwurf der Software unterstützen, vor allem bei Teamarbeit. Eine saubere Projektdokumentation verhindert Missverständnisse und fördert die Koordination in der Gruppe.

Auch innerhalb des Quellcodes sind Kommentare und Dokumentationen nicht nur Hilfe für fremde Betrachter. Sie sind auch eine willkommene Gelegenheit für den Entwickler selbst, sich noch einmal mit seinem Code zu beschäftigen und so eventuelle Bugs zu erkennen. Immer beliebter wird das gemeisame Programmieren in Zweierteams vor einem gemeinsamem Rechner. Durch das Mitdenken des zweiten Entwicklers lassen sich insbesondere logische und Syntaxfehler vermeiden (Vier-Augen-Prinzip).

Falls trotz aller guten Vorsätze sich Fehler in der Software einschleichen, ist das noch keine Katastrophe (eher der Nomalfall). Diese Fehler sollten nur rechtzeitig vor der Auslieferung der Software gefunden und behoben werden. Die effektivsten (und teuersten) Methoden sind Verifizierung und formale Beweise. Sie lassen sich nicht vollständig automatisieren und werden deshalb nur bei extrem wichtigen Softwarekomponenten durchgeführt. Sehr beliebt (und manchmal auch mit großer Wirkung in den Medien) sind Test- und Betaversionen. So lässt sich ein Programm schon im Einsatz testen ohne im Fehlerfall Garantie geben zu müssen.

Als gute Praxis hat sich der Einsatz von automatisierten Testumgebungen erwiesen, für Java hat sich JUnit etabliert. So können schon während der Entwicklung, z.B. bei jedem vollständigen nächtlichen Build, Funktionen auf ihr korrektes Verhalten getestet werden.

Die mit obigen Mitteln gefundenen Bugs sollten schnell und möglichst effizient aus dem Quellcode verschwinden. Dabei haben sich einige Debugging-Strategien und Softwaretools bewährt, die wir im folgenden vorstellen möchten.

5.1.5 Der Debuggingprozess

Das Entfernen eines Fehlers lässt sich im Allgemeinen in folgende Schritte aufteilen:

1. Das Problem identifizieren, z.B. durch eigene Beobachtung oder Feedback vom Kunden.
   - Dabei muss geklärt werden, ob sich das Programm wirklich fehlerhaft verhält oder nur eine falsche Vorstellung vom korrekten Verhalten existiert.
   - Falls es ein Fehler ist, was sollte das Programm tun und was tut es wirklich?
2. Die zur Fehlersituation zugehörigen Informationen sammeln
   - Problembeschreibung vom Anwender
   - Logfiles durchsehen und nach ersten Ursachen forschen

   - Eigene Beobachtungen und Erfahrungen miteinbringen
3. Hypothese über den Fehler aufstellen, d.h. die defekte Stelle im Quellcode finden
4. Hypothese testen, Debugger zur Hilfe nehmen
5. bestätigt sich die Hypothese nicht, so zurück zu 3.
6. Erarbeiten einer Lösung passend zur Hypothese
7. Testen der Lösung, evtl. wieder Debugger zur Hilfe nehmen:
   - Korrektur der defekten Stelle im Quelltext.
   - Regression: Vielleicht verursacht man durch die Korrektur an anderer Stelle neue Fehler. Auf diese ist zu achten.
8. wenn Lösung nicht hilft, zurück zu 6.

Dabei ist das Aufstellen der Hypothese ein wichtiger Schritt im Prozess. Umso besser die Hypothese, umso leichter wird auch die Korrektur werden.

5.1.6 Debugging-Techniken

Für das Aufstellen der Debug-Hypothese lassen sich verschiedene Techniken unterscheiden. Bei manchen Vorgehensweisen wird der Quelltext manipuliert, sie werden unter dem Begriff „eingreifendes Debugging“ zusammengefasst. Diese Änderungen können das Programm ungewollt beeinflussen und so eine effektive Fehlersuche erschweren. Denkbar wären zusätzliche Bildschirmausgaben oder das öffnen einer Verbindung zu einen anderen Rechner mit einem Debugging-Client.

„Nichteingreifendes Debugging“ beobachtet nur ein laufendes Programm von außen. Dieser Ansatz wird bevorzugt, weil so die Wahrscheinlichkeit für Heisenbugs gegen Null geht.

Eine Klassifizierung auf anderer Ebene ist die Unterscheidung in Kurz- und Langzeitdebugging. Beim Langzeitdebugging wird das System im produktiven Betrieb (also im harten Einsatz) beobachtet und auftretendes Fehlverhalten notiert. Beim kurzfristigen Debugging wird das System ausschließlich zum Debuggen gestartet und dabei permanent vom Entwickler kontrolliert.

Als konkrete Beispiele lassen sich nennen:

- Der Vergleich des Programmverhaltens mit der Dokumentation, Kommentaren im Code, womöglich verfügbarem ähnlichem Code und den generierten Logdateien. Das Programm muss hierfür nicht manipuliert werden, folglich ist es nichteingreifendes Debugging.
- Als massiv eingreifendes Debugging ist das Vereinfachen von Code zu bezeichnen. Der Programmierer geht dabei fremden oder älteren eigenen Code durch und schreibt ihn neu, in einer Form, die ihm mehr zusagt. Am Ende sollte der Entwickler den Code zu 100% verstanden haben (sonst hätte er nicht ein semantisch gleiches Programm reproduzieren können). Dabei sollte der Fehler aufgefallen und behoben worden sein. Hier ist die Nachkontrolle wegen eigenen neuen Fehlern immens wichtig.

- Nicht ganz so radikal, aber immer noch eingreifend ist das absichtliche Einschleusen von Fehlern in das Programm („Error seeding“). Durch den fast spielerischen Vergleich von Ursache und Wirkung bekommt man einen besseren Einblick in den Kontrollfluss.
- Als letzter und verzweifelter Ausweg bleibt der Wechsel des Compilers oder der Laufzeitumgebung, da auch diese Softwarekomponenten Fehler enthalten können. Womöglich läuft so das eigene Programm auf einmal fehlerfrei. Diese Technik ist noch knapp als „Nichteingreifendes Debugging“ zu bezeichen.
- Der Mensch erkennt Zusammenhänge in Bildern viel schneller als in Text. Diese Errungenschaft der Evolution kann man sich auch beim Debugging durch geeignete Visualisierung zu Nutze machen. In Kapitel 5.3 möchten wir dazu einige Beispiele bringen.

5.1.7 Softwaretools

Fast alle oben genannten Debuggingtechniken lassen sich durch Softwaretools unterstützen. Insbesondere für das Betrachten von Variablenbelegungen ist Debugging-Software eine große Hilfe. Wir möchten einen kurzen Überblick über drei Tools geben, ihre Reihenfolge entspricht auch ihrer historischen Entwicklung.

Standardausgabe

Ein sehr einfacher Trick ist die Ausgabe von Variablenwerten und Ereignissen als Text auf die Konsole. Eigentlich braucht man hier kein zusätzliches Softwaretool, die Möglichkeiten hierfür sind in fast jeder Programmiersprache enthalten. In Java geschieht dies für die Variable `xyz` durch `System.out.println(xyz)`. Dies ist recht unkompliziert und mit etwas Übung gelangt man sehr schnell zu interessanten Einsichten in die Programmstrukturen. Insbesondere lässt sich durch die Reihenfolge der Ausgabe viel über den Ablauf des Programmes lernen.
Der gravierende Nachteil ist der Eingriff in den Quellcode. Oft lassen Programmierer die Ausgabebefehle nach der Fehlerkorrektur stehen, der Quellcode wird mit der Zeit sehr unübersichtlich. Auch kann das simple Schreiben von ASCII-Zeichen auf eine Textkonsole erstaunlich viel Rechenzeit beanspruchen, womit sich das Laufzeitverhalten des Programmes (zum schlechteren) ändert. Eher ein kosmetisches Ärgernis sind vergessene Ausgaben, die später beim verwunderten Kunden den Bildschirm besiedeln.

Es sollte beachtet werden, dass echte Fehlermeldungen in die Standardfehlerausgabe `System.err` gehören.

Speicherdumps

Eine sehr genaue Einsicht in die Fehlersituation bietet die Ausgabe aller Speicherzellen, die zum Zeitpunkt des Fehler relevant oder in der Nähe waren. Die Speicheradressen sowie die enthaltenen Bytewerte erscheinen oft in hexadezimaler Schreibweise. Das berühmt-berüchtigte Beispiel eines Dumps waren die Bluescreens der alten Windows-Versionen.

Leider ist so eine Wüste aus Hexadezimalzahlen nicht besonders intuitiv zu interpretieren. Eine praktikable Lösung für dieses Problem sind „symbolische Debugger“: Die vom Compiler erzeugte Symboltabelle mit der Zuordnung Variablenname $\leftrightarrow$ Speicheradresse wird hier weiterverwendet. Statt der Speicheradresse kann so der Name der Variable ausgegeben werden.

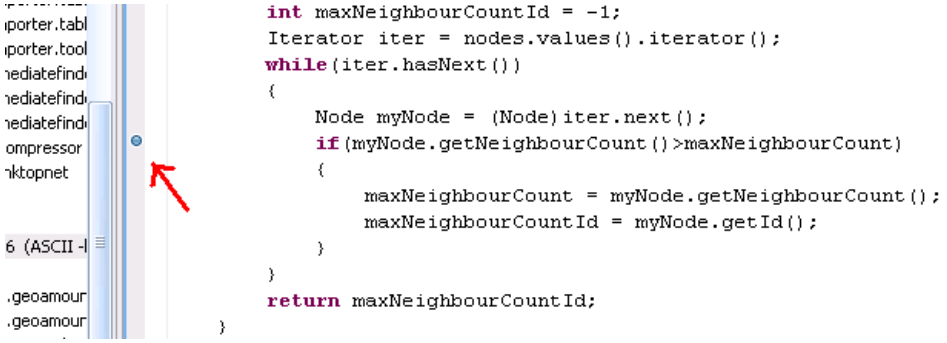


Abbildung 5.1.: Breakpoints in Eclipse

Debugging zur Laufzeit

In den letzen Jahren haben sich immer mehr integrierte Entwicklungsumgebungen (IDEs) durchgesetzt. Sie lösen den Texteditor und den Compiler an der Kommandozeile ab, sehr oft ist auch ein Debugger mit eingebaut.

Durch das enge Zusammenspiel der Komponenten ergibt sich eine neue Dimension von Komfort beim Debuggen. Der Benutzer der IDE kann Zeilen im Quelltext des zu entwickelnden Programmes markieren. Kompiliert und startet man das Programm aus der IDE heraus, so pausiert es beim Erreichen dieser Breakpoints (Haltepunkte). Innerhalb des eingefrorenen Programmes kann der Entwickler einige Details einsehen:

- Die Variablenbelegung aller im Kontext verwendeten Variablen
- Nachvollziehen des nächsten Programmschritts, die gerade aktuelle Codezeile wird markiert
- Manche Debugger erlauben sogar das Ändern von Code, dieser wird sofort in das noch laufende Programm eingesetzt. Solch ein Vorgehen fordert Heisenbugs förmlich heraus.

Dem Debugger stehen auch alle anderen Annehmlichkeiten der IDE zur verfgung, wie Codevervollständigung, Auszeichnung der Syntax, integrierte Referenz, etc.

Ein kleines Manko können Breakpoints innerhalb von Schleifen sein. Ist man an der Variablenbelegung beim genau fünftausendsten Durchlauf einer Schleife interessiert, so bleibt nur das fünftausendfache Klicken auf „weiter“ oder man ändert doch wieder etwas am Quellcode und fügt einen bedingten Sprung ein.
Deswegen bieten einige Debugger auch „bedingte“ Breakpoints an, diese werden erst bei Erreichen einer Bedingung scharfgeschaltet.

Wird ein Programm an einer Quellcodezeile angehalten, so ist der Befehl am Breakpoint noch nicht ausgeführt. Der Quelltext mit dem Breakpoint sei die initiale „Quellcodeebene“. Zur Steuerung des weiteren Kontrollflusses bieten Debugger mindestens folgende Funktionen (Abbildung 5.2):

Step into : wenn der nächste Befehl eine komplexe Funktion ist (also nicht nur einfache Variablenzuweisung oder Bedingung), so springt der Debugger „tiefer“ in die zugehörige

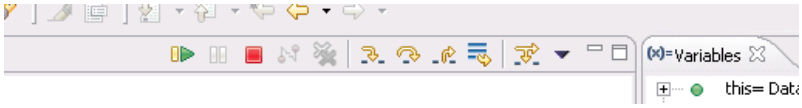


Abbildung 5.2.: Ausführungskontrolle in Eclipse

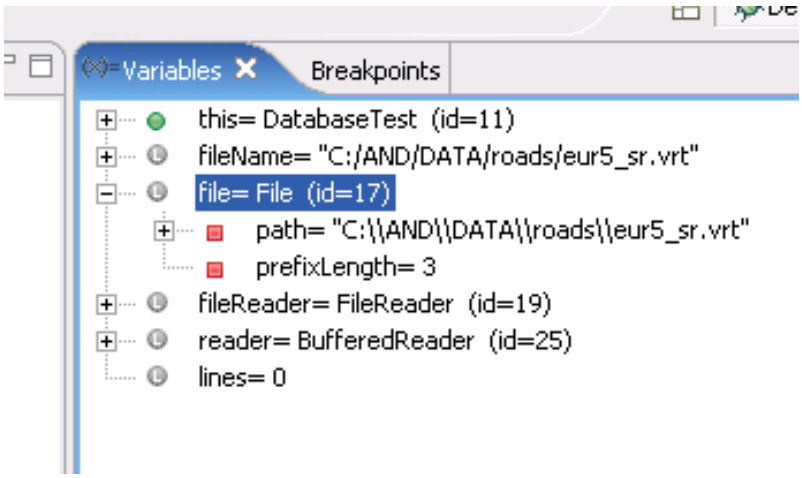


Abbildung 5.3.: Variablendarstellung in Eclipse

Quellcodeebene.

Step over : der nächste Befehl in der aktuellen Quellcodeebene wird ausgeführt.

Step return : die Funktion in der aktuellen Quellcodebene wird vollständig beendet, danach springt man eine Quellcodebene höher.

Ähnlich dem zuvor beschriebenen symbolischen Debugger ist der aktuelle Zustand aller Variablen am Breakpoint einsehbar (Abbildung 5.3). Falls eine Variable vom Typ Objekt ist, so bietet der Debugger eine hierarchische Ansicht aller darin enthaltenen primitiven Variablen und Unterobjekte.

5.2 Integrated Comprehension Model

Neben den technischen Aspekten von Debugging-Strategien und dem allgemeinen Vorgehen im Debugging-Prozess muss außerdem betrachtet werden, wie ein Programmierer den Code während des Debuggens geistig verarbeitet und speichert. Die Beschränktheit des menschlichen Gedächtnisses verhindert grundsätzlich, dass ein Programmierer den gesamten Code im Kopf behalten kann. Daher wird er verschieden große Teile des Codes in verschiedenen Abstraktionsebenen mental speichern müssen. Ausgehend von älteren eigenen und fremden

Studien, die sowohl Top-Down- als auch Bottom-Up-Modelle zur Code-Repräsentation vorstellten, entwarfen von Mayrhauser und Vans das Integrated Comprehension Model, das die mentale Repräsentation in die folgenden drei Ebenen zerlegt von Mayrhauser und Vans (1997) (Abbildung 5.4):

- Top-Down Model
- Situation Model
- Program Model

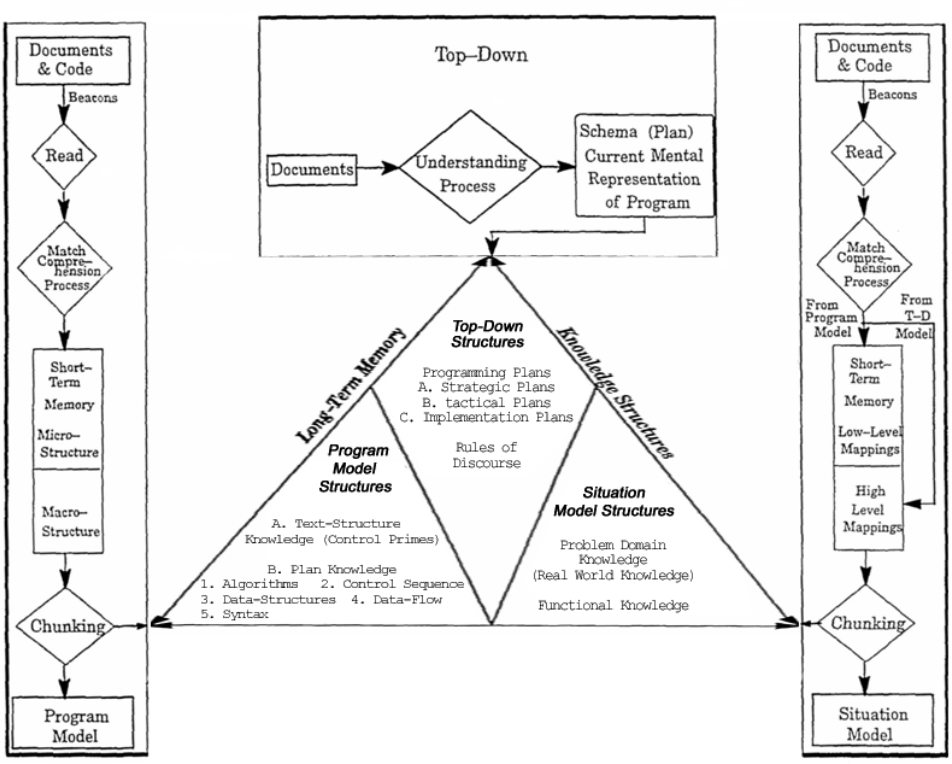


Abbildung 5.4.: Grafische Darstellung des Integrated Comprehension Model nach von Mayrhauser & Vans.

5.2.1 Top-Down Model

Das Top-Down Model entsteht aus der Betrachtung der Programmdokumentation und beinhaltet generelles Wissen über das Anwendungsgebiet (Domainwissen), Wissen über die Aufgabe des Programms, Wissen über die Komponenten und ihrer Zusammenarbeit sowie einen Plan der Implementierung. Mit diesem Modell erfasst der Programmierer die Komponenten

des Gesamtsystems, ihre grundsätzliche Funktion und ihre Zusammenarbeit. Er gewinnt und speichert Informationen darüber, welche Aufgabe des Programmes von welcher Komponente übernommen wird, ohne detailierte Angaben über die Funktion der Kompontente oder die genaue Umsetzung der Kommunikation zwischen Komponenten zu haben.

5.2.2 Program Model

Das Program Model entsteht aus der Betrachtung eines kleinen Codeabschnitts und beinhaltet Informationen zur Struktur des Codes, zu Algorithmen und Kontrollfluss sowie zu Datenstrukturen und Datenfluss. Es wird Bottom-Up hergeleitet, indem der Programmierer eine oder einige wenige Zeilen Code liest und ihren Sinn erfasst. Im Kurzzeitgedächtnis vergleicht er sie dann mit den kurz zuvor gelesenen Zeilen und entwickelt so eine Mikrostruktur für die gerade gelesenen Codezeilen und entwickelt unter Rückgriff auf sein Gedächtnis eine Makrostruktur für den gesamten gerade betrachteten Code-Abschnitt.

Für große Programme ist ein Program Model kaum vollständig aufzubauen. Da die Informationen aus dem Program Model in komprimierter Form in das Gesamtwissen des Programmierers eingehen, ist dies aber auch nicht erforderlich.

5.2.3 Situation Model

Das Situation Model kombiniert Domain-Wissen und Funktions-Wissen, indem es sowohl das Program Model als auch das Top-Down Model benutzt. Es abstrahiert den Datenfluss und die Funktionalität, d.h. es komprimiert die Erkenntnisse aus dem Program Model. Gleichzeitig bringt es diese Informationen mit den Plänen aus dem Top-Down Model in Verbindung und konkretisiert somit das allgemeine Wissen über das Programm.

Grundsätzlich wird das Situation-Model wie das Program Model Bottom-Up hergeleitet.

5.2.4 Fallstudie

Zur Überprüfung ihres Modells haben von Mayrhauser und Vans eine Fallstudie durchgeführt, in der vier Programmierer mit unterschiedlicher Erfahrung und Vorkenntnissen mit einem größeren Softwaresystem (> 40000 Zeilen Code) konfrontiert wurden. Ihre Aufgabe war es, einen bekannten Bug zu lokalisieren und zu beheben. Dabei wurde das Verhalten der Testpersonen protokolliert.

Die Testpersonen lassen sich wie folgt charakterisieren:

Der Sprachexperte hat Erfahrung in der verwendeten Programmiersprache, aber nicht in der Anwendungsdomäne. In der Studie gab es eine Testperson dieser Art, die zusätzlich über etwas Vorwissen über den zu bearbeitenden Programmcode verfügte.

Der Domain-Experte hat keine Erfahrung in der verwendeten Programmiersprache und kein Vorwissen über den Quellcode, aber Erfahrung in der Anwendungsdomäne. In der Studie gab es drei Testpersonen dieser Art, die entweder über kein Vorwissen, wenig Vorwissen oder viel Vorwissen über den zu bearbeitenden Programmcode verfgten.

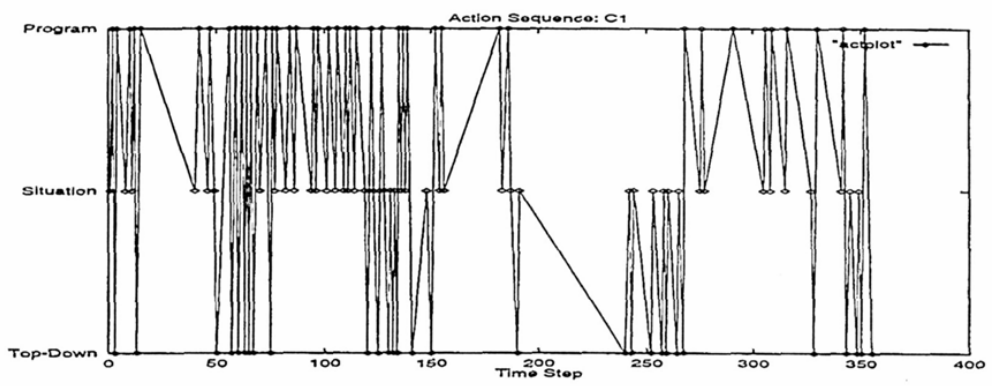


Abbildung 5.5.: Grafische Darstellung des Wechsels zwischen den mentalen Modellen bei einer Testperson mit Erfahrung in der verwendeten Programmiersprache, etwas Vorwissen über den Code und fehlendem Domain-Wissen.

Der Sprachexperte

Der Sprachexperte wechselt häufig zwischen dem Program Model und dem Situation Model und nutzt nur selten das Top-Down Model (Abbildung 5.5). Bei den seltenen Sprüngen ins Top-Down Model verbringt er dort auch nur eine sehr kurze Zeitspanne, während er sehr lange im Situation Model arbeitet. Diese Beobachtung ist schlüssig, da ihm aufgrund des fehlenden Domain-Wissens das Top-Down Model nicht helfen kann. Aufgrund seiner Spracherfahrung kann er aber den Code schnell erfassen und gut abstrahieren, so dass das Situation Model für ihn die Hauptebene bei der Arbeit darstellt.

Der Domain-Experte

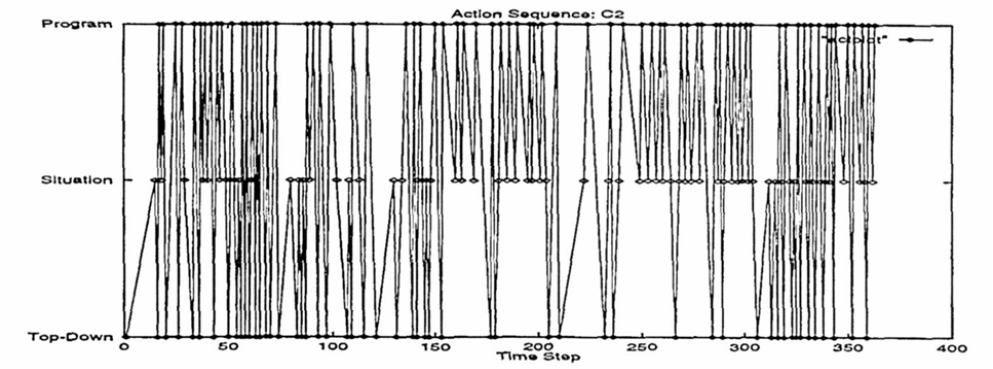


Abbildung 5.6.: Grafische Darstellung des Wechsels zwischen den mentalen Modellen bei einer Testperson ohne Erfahrung in der verwendeten Programmiersprache, ohne Vorwissen über den Code und mit umfassendem Domain-Wissen.

Domain-Experten wechseln deutlich häufiger zwischen dem Top-Down Model und den tieferen Ebenen. Abbildung 5.6 verdeutlicht dies am Beispiel des Programmierers ohne Vorwissen über den Code, der von allen Testpersonen die höchste Frequenz des Wechsels zwischen den Modellen aufweist. Er versucht, aus seinem Domain-Wissen heraus den unbekannten Code zu verstehen, d.h. er vergleicht jede gewonnen Informationen über einen Programmteil mit seinem Wissen über die Aufgaben des Programms, um sich von der Korrektheit seiner Annahmen über den Codeabschnitt zu überzeugen. Das Situation Model dient ihm dabei als Brücke, um längere und komplexere Codepassagen zu verstehen.

Mit steigendem Vorwissen über den Code sinkt die Häufigkeit der Wechsel ab, während die Aufenthaltsdauer im Top-Down Model sogar ansteigt. Durch das vorhandene Vorwissen braucht der Programmierer nicht mehr jede Codezeile zu lesen und erspart sich somit für Teile des Programmes den Aufbau eines Program Models. Die Nutzung des Situation Models geht dagegen nicht zurück.

5.2.5 Ergebnisse der Studie

Die Autorinnen der Studie kommen zu folgenden Schlüssen:

Programmierer mit hohem Codewissen und wenig Domainwissen arbeiten Bottom-Up aus dem Code und nutzen das Situation Model als Brücke zum Domainverständnis. Sie suchen in der Dokumentation Erklärung für die Informationen, die sie aus dem Code gewonnen haben.

Programmierer mit wenig Codewissen und hohem Domainwissen arbeiten Top-Down aus der Dokumentation und bauen das Situation Model in vielen kleinen Schritten auf. Sie nutzen das Situation Model als Brücke zum Codeverständnis und suchen im Code nach der Umsetzung des allgemeinen Plans aus dem Top-Down Model.

Programmierer mit hohem Code- und Domainwissen brauchen keine Brücke zwischen den Modellen. Sie nutzen die Wechsel zwischen den Modellen lediglich zur Vervollständigung ihrer bereits umfassenden mentalen Repräsentation des Codes.

5.2.6 Informationsgewinnung

Zur Erstellung der verschiedenen Modelle sind jeweils unterschiedliche Informationsquellen relevant, die zudem auf unterschiedliche Art und Weise erschlossen werden können.

Das Program Model steht sehr nah am Code und ist daher relativ leicht zu gewinnen. Das Lesen und Erfassen des Codes wird durch klassische Debugging-Tools, die z.B. Breakpoints oder die Überwachung der Variablenbelegung anbieten, gut unterstützt.

Das Top-Down Model ist in Teilen unabhängig vom Programm (Domainwissen) und grundsätzlich unabhängig vom konkreten Code. Die Informationen sind normalerweise in der Programm-Dokumentationen schriftlich und ggf. auch grafisch vorhanden und bereits geeignet strukturiert.

Der Aufbau eines Situation Model ist dem Programmierer weitgehend alleine überlassen. Das Strukturieren und Zusammenfassen von Informationen aus dem Code wird von Tools wenig unterstützt, während eine vollständige schriftliche Dokumentation aller Programmabläufe in der Detailiertheit des Situation Model die Nutzbarkeit der Programmdokumentation alleine aufgrund des Umfangs stark einschränken würde.

Insbesondere im Debugging ist beim Aufstellen von Hypothesen und Testen von Lösungen das Situation Model am wichtigsten und somit eine stärkere Unterstützung des Programmierers sehr wünschenswert.

5.3 Grafische Tools

Grafische Ansätze zur Darstellung von Programminformationen könnten die Lücke für das Situation Model füllen. Grafiken werden in der Regel schneller aufgenommen, verarbeitet und besser im Gedächtnis gespeichert als rein textuelle Darstellungen und bieten eine natürliche Abstraktion des Codes, ohne die Struktur zu verdecken. Zudem sind Diagramme oft auch intuitiver zu bedienen und zu manimulieren sowie von sprachlichen Grenzen unabhnngiger. Allerdings erfordert die grafische Informationsdarstellung meist schon in einfachen Formen einen deutlich gesteigerten Ressourceneinsatz, der z.B. die Darstellung zumfassender Debugging-Informationen in Echtzeit zur Laufzeit des Programmes nahezu unmöglich erscheinen lässt.

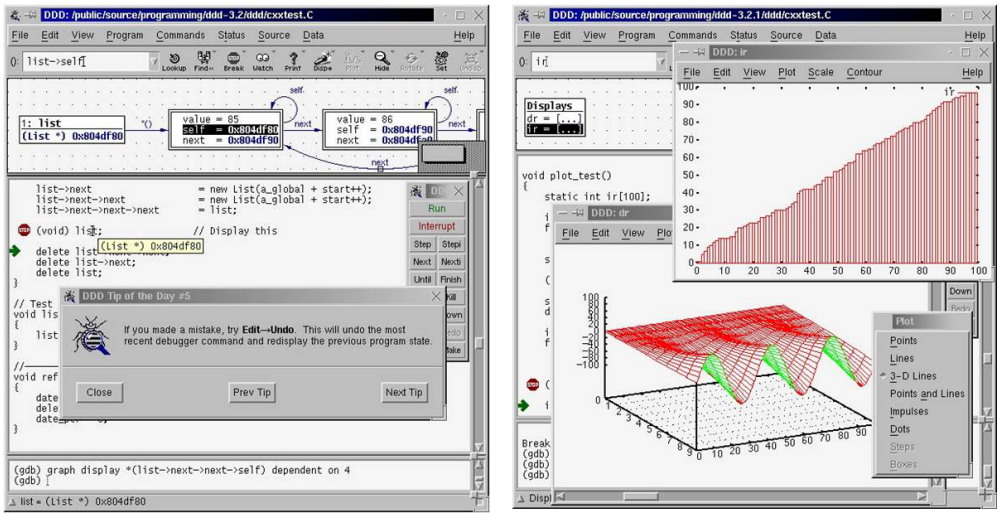


Abbildung 5.7.: Grafisches Frontend für gdb Gaylard und Zeller (2004) mit der Darstellung von Programmkonstrukten als Diagramm (links oben) und der Visualisierung von Variablenbelegungen in 2D und 3D (rechts).

5.4 Schlüsse und Ausblick

Debugging ist ein komplexer Prozess, der die Kenntnis von Konzepten, die gezielte Anwendung verschiedener Techniken und die Fähigkeit zur geistigen Repräsentation von Code auf verschiedenen Ebenen voraussetzt. Vorhandene Tools können diesen Prozess unterstützen,

bieten aber nicht immer die gerade benötigten Informationen in der geeigneten Form an. In unserer PG möchten wir neue (z.B. bessere oder effizientere) Darstellungskonzepte für Debugging-Informationen erproben, um neue Antworten auf die Frage zu finden, welche Informationen zu welcher Zeit in welcher Form dem Entwicklern am meisten helfen. Dabei könnte die 3-dimensionalen Visualisierung von Laufzeitinformationen zu einer solchen Antwort führen.

KAPITEL 6

Debugging-Werkzeuge

Boris Brodski

6.1 Einleitung

Dies ist eine Seminarausarbeitung der PG 458, die wichtige Prinzipien und Konzepte eines Debuggers vorstellt. Für den Vortrag und die Ausarbeitung wurde ausschließlich das Buch Rosenberg (1996) benutzt.

Ein Debugger ist ein Programm, das hilft, das Verhalten von anderen Programmen zu verstehen und zu analysieren. Meistens werden Debugger benötigt, um Programmfehler zu suchen, jedoch sind sie auch sehr hilfreich zum Verständnis des Programmablaufs. Debugger werden oft von Softwareentwicklern und Testern benutzt. Die Idee des Debuggers besteht darin, dass ein Debugger während der Ausführung von Programm die Kontrolle über das Programm übernimmt, um das Programm stoppen und schrittweise ausführen zu können. Man kann den Kontext des Programms an jede Stelle überprüfen. Der Kontext fast die gesamte Information über den Ablauf des Programms zusammen. Er beinhaltet zum Beispiel die Belegung aller Variablen, den Zustand des Prozessors, die Zeile im Programm, die gerade ausgeführt wird und viele andere wichtige Informationen, die in weiteren Kapiteln ausführlich vorgestellt werden. Außerdem kann der Debugger eine Absturzstelle und den Grund des Absturzes ermitteln. Einige Debugger sind sogar in der Lage, nach dem Absturz einer Applikation anhand des Protokolls oder des so genanten “Memory Dump” (die Speicherung des gesamten Applikationsspeichers), den Kontext der abgestürzten Applikation darzustellen.

Der Debugvorgang wird so ausgeführt: eine kompilierte Applikation, deren Quelltext verfügbar ist, wird in einen Debugger geladen. Danach werden interessante Stellen im Quelltext mit Hilfe der so genannten Breakpoints markiert und die Applikation gestartet. Der Debugger stoppt die Applikation, falls die Ausführung einen der gesetzten Breakpoints erreicht hat. Der Nutzer hat dann die Möglichkeit, den Kontext der Applikation zu prüfen, die Applikation schrittweise auszuführen oder sie weiter laufen zu lassen.

Meistens sind die Debugger programmiersprachen- oder sogar compilerspezifisch. Das heißt, dass man einen solchen Debugger nur mit den Programmen benutzen kann, die in einer bestimmten Sprache geschrieben sind oder mit einem bestimmten Compiler kompiliert sind. So eine Spezialisierung erlaubt Implementierung von vielen Diensten, die ein universaler Debugger nicht anbieten kann. Einer dieser Dienste ist zum Beispiel, dass ein programmiersprachenspezifischer Debugger eine Illusion erschaffen kann, dass der Programmquelltext direkt aus-

geführt wird. Die tatsächlich durchgeführten Prozessorschritte werden hinter dem Programmquelltext versteckt.

Im folgenden Kapitel werden Heisenberg Prinzipien, die bei der Erstellung einer Debugger zu beachten sind, vorgestellt. In Kapitel 6.3 wird der Kontext präsentiert. In Kapitel 6.4 werden wichtige Bestandteile eines Debuggers erläutert. In Kapitel 6.5 werden drei wichtige Algorithmen vorgestellt. Die besondere Arten von Debugging werden in Kapitel 6.6 besprochen, gefolgt von einer Zusammenfassung.

6.2 Heisenberg - Prinzipien

Die Heisenberg-Prinzipien sind die wichtigsten Prinzipien für die Entwicklung eines Debuggers. Die bestehen aus 3 Thesen, die folgend vorgestellt werden.

6.2.1 Das erste Heisenberg - Prinzip

Das erste Prinzip besagt, dass die Umgebung der Applikation nicht vom Debugger beeinflusst werden darf. Würde der Debugger die Umgebung der Applikation beeinflussen, könnten entweder neue Fehler entstehen, die es erschweren, den ursprünglichen Fehler zu suchen, oder auch einige ursprünglichen Fehler können verschwinden, was es unmöglich macht, sie mit Hilfe des Debuggers zu finden.

Es besteht jedoch die Gefahr, dass die Applikation auf vielen Weisen beeinflusst wird. Selbst die Tatsache, dass der Debugger ein Programm ist, das auf demselben Betriebssystem ausgeführt wird, macht die Implementierung eines Debuggers gemäßdieses Prinzips fast unmöglich. Der Debugger ist ein Programm, das selbstverständlich Speicher und anderen Ressourcen braucht. Es kann passieren, dass spezielle Fehler in der Applikation nur dann auftreten, wenn der Specherzeiger einen bestimmten Wert annimmt, der sich bei Anwesenheit des Debuggers ändert.

Moderne Prozessoren und Betriebssysteme, die den sogenannten virtuellen Speicher unterstützen, ermöglichen einige solcher Effekte zu verhindern, jedoch tritt das Problem beim Debuggen von eingebetteten Systemen nach wie vor auf.

Falls der Debugger und die aktuelle Applikation beide GUI Applikationen sind, wird es selbst bei neuen Prozessoren schwierig, dafür zu sorgen, dass der Debugger die GUI Applikation nicht beeinflusst. (siehe Abschnitt 6.6.2)

6.2.2 Das zweite Heisenberg - Prinzip

Der zweite Prinzip von Heisenberg lautet: Die Information, die vom Debugger dargestellt wird, muss immer glaubwürdig sein. Die Entwickler eines Debuggers müssen immer dafür sorgen, dass die Nutzer dem Debugger vertrauen können. Vom Debugger fehlerhaft präsentierte Information verwirrt den Nutzer und macht die Suche nach Fehler schwieriger.

Da die Kompilierung des Programms ohne hin ein komplizierter Prozess ist, ist das Einhalten dieses Prinzips eine mühsame Angelegenheit, da man viel Schritte, die beim Kompilieren gemacht wurden, rückgängig machen muss. Die meisten Schwierigkeiten entstehen

beim Debuggen vom optimierten Code, weil die Umkehrung von Optimierung eine besonders schwierige Aufgabe ist (siehe Kapitel 6.6.3).

6.2.3 Das dritte Heisenberg - Prinzip

Das dritte Prinzip von Heisenberg lautet: Der Kontext der Applikation muss so umfangreich wie möglich dargestellt werden. Um mit Hilfe des Debuggers Fehler zu suchen, muss der Nutzer in der Lage sein, die nötige Information über den Zustand der Applikation zu erhalten. Wenn der Debugger die Applikation stoppt, soll der Nutzer den Grund erfahren können. Es kann entweder ein Programmabsturz oder ein Breakpoint den Grund für das Anhalten des Programms sein. Weiter stellt sich die Frage, wo man sich aktuell in dem Programmquelltext befindet. Danach soll es für den Nutzer möglich sein, die Belegung der Variablen abzufragen. Die weiteren Bestandteile des Kontexts werden in Kapitel 6.3 ausführlich beschrieben.

6.3 Der Kontext

Der Kontext fasst die Informationen über den Ablauf eines Programms zusammen. Seine Repräsentation ist eine der wichtigste Aufgaben der Debuggers. Der Kontext wird durch die sogenannten Ansichten (views) dargestellt. Jede Ansicht präsentiert Information zu einem bestimmten Thema. Um die Arbeit mit einem Debugger zu beschleunigen, ist es besser, die Ansichten so zu erstellen, dass, da wo es Sinn macht, die direkte Modifikation der Daten möglich ist.

6.3.1 Quelltextansicht (source view)

Die Quelltextansicht antwortet auf die Frage “In welchem Stadium der Ausführung befindet sich das Programm? In welchem Modul wird welche Zeile ausgeführt?”. Es wird angezeigt, welche Zeile bzw. welcher Befehl auszuführen war, als das Programm angehalten wurde. Außerdem wird in dieser Ansicht andere für das Debuggen spezifische Information angezeigt, wie zum Beispiel die Stellen im Programmquellcode, an denen Breakpoints gesetzt sind.

Mit Hilfe dieser Ansicht wird auch die oben genannte Illusion geschaffen, dass der Programmquellcode direkt ausgeführt wird. Dafür werden spezielle Algorithmen benutzt, die zum Beispiel die Applikation zeilenweise ausführen, was in der Quelltextansicht direkt angezeigt wird. Diese Algorithmen werden ausführlich in Kapitel 5 beschrieben.

6.3.2 Stapelansicht (stack view)

Jede Programmiersprache, mit der der Benutzer Funktionen definieren kann, nutzt den Stapel, um lokale Variablen und andere funktionenspezifische Information zu speichern. Ein Stapel ist ein Speicherbereich, der sequenziell aufgefüllt wird. Bei jedem Funktionsaufruf wird ein so genannten Frame auf dem Stapel gespeichert. Ein Frame ist ein zusammenhängendes Abschnitt des Stapel, der die nötige Information enthält, um den Zustand der Applikation vor dem Funktionsaufruf wiederherzustellen. Außerdem enthält der Frame die Belegung der Parameter und lokalen Variablen der aufgerufenen Funktion. Dieses Prinzip erlaubt den konfliktfreien

rekursiven Aufruf von Funktionen.

In großen Programmen werden oft gleiche Funktionen von verschiedenen Stellen aufgerufen. Es bedeutet, dass es nicht ausreicht zu wissen, welche Funktionen gerade ausgeführt wird. Um eine vollständige Übersicht über das Programm zu erhalten, muss man auch wissen, welche Funktion die grade betrachtete Funktion aufgerufen hat. Die so genannte Geschichte der Aufrufe (call history) wird in der Stapelansicht angezeigt.

In der Stapelansicht werden also alle nicht beendeten Funktion, angefangen mit der Startfunktion, angezeigt. Um eine bessere Übersicht zu schaffen, ist es wichtig, die Namen der Funktionen und ihre Parameterliste anzuzeigen. Anhand des Stapelframes ist es möglich, die Belegung der Parameter abzuleiten. Es ist aber nicht immer sinnvoll, die Parameterbelegung direkt in der Stapelansicht anzuzeigen, da die Parameterwerte während der Ausführung der Funktion bereits abgeändert sein können. Der Eintrag in der Stapelansicht, wie etwa `zeigeFenster(3)` bedeutet intuitiv, dass die Funktion `zeigeFenster(int i)` mit dem Parameter `i = 3` aufgerufen wurde. Das ist allgemein falsch, wie folgendes Beispiel zeigt:

```
function zeigeFenster(int i)
{
    i = 3;
    // ...
}
```

Ab der Stelle nach der Zuweisung `i = 3` wird es immer `zeigeFenster(3)` in Stapelansicht angezeigt, obwohl die Funktion `zeigeFenster()` mit beliebigen Parameter aufgerufen werden kann.

6.3.3 Breakpointsansicht (breakpoint view)

Die Breakpointsansicht zeigt an, welche Breakpoints im Programmquellcode gesetzt sind. Mit Hilfe dieser Ansicht soll es außerdem möglich sein, neue Breakpoints einzufügen, bereits existierende Breakpoints anzupassen oder zu löschen. Die zugehörige Eigenschaften von Breakpoints werden in Abschnitt 6.4.3 beschrieben.

6.3.4 Disassembleransicht (disassemble view)

Disassemblierung ist eine teilweise Umkehrung der Compilierung. Der Binärecode des Programms wird zurück in die Maschinensprache (assembler) konvertiert. Das ermöglicht es dem Nutzer, der über die nötigen Kenntnisse verfügt, genau zu sehen, wie das Programm vom Compiler übersetzt wurde und wie es auf dem Prozessor ausgeführt wird. Da die Disassembleransicht die Flexibilität des Debuggers erhöht, ist es für viele Sprachen sinnvoll, eine Disassembleransicht im Debugger zu realisieren. Diese Ansicht ermöglicht außerdem die Suche nach Compilierfehlern.

Um einen vollständigen Überblick über den Maschinencode zu erschaffen, soll folgende Information in Disassembleransicht präsentiert werden:

- Hexadezimale Darstellung vom Binärecode des Programms
- Darstellung des Binärecodes des Programms in Maschinensprache (Assembler)
- Abbildung des ursprünglichen Programmquellcodes auf Maschinensprache
- Darstellung der im Programm gesetzten Breakpoints

Zusätzlich ist es für das Debuggen sehr hilfreich, falls alle vorkommende Adressen, so wie Variablenadressen und Funktionenadressen aufgelöst und mit den Namen aus dem Programmquellcode versehen werden.

6.3.5 CPU-Ansicht (CPU view)

Um den Ablauf des Programms in der Maschinensprache nachvollziehen zu können, braucht man Informationen über den Prozessorzustand. Diese stellt die CPU-Ansicht bereit. Hier werden die Werte aller CPU-Register dargestellt. Die Modifikation der Register sollte durch direkte Änderung in der CPU-Ansicht möglich sein. Damit der Benutzer besser den Ablauf des Programms nachvollziehen kann, ist es hilfreich, wenn die in einem Schritt verursachten Änderungen farblich hervorgehoben werden.

6.3.6 Speicheransicht (memory view)

Die Speicheransicht ermöglicht das Prüfen und Modifizieren des Prozessspeichers. Es soll auch hier verschiedene Ansichtsmöglichkeiten, wir etwa Byteweise oder 4-Byteweise, zur Verfügung gestellt werden. Die Möglichkeit, direkt über diese Ansicht den Speicherinhalt zu ändern, sorgt auch hier für schnellere Arbeit der Nutzer.

6.3.7 Variablenansicht (variable view)

Die Variablenansicht dient dazu, die Werte von ausgewählten Variablen zu überwachen bzw. zu ändern. Man kann außerdem eine automatische Liste der lokalen Variablen zu Verfügung stellen. Die Werte der Variablen sollen entsprechend dem Variablentyp dargestellt werden. Für eine bessere Kontrolle über die Ausführung der Applikation ist es wiederum hilfreich, die von einem Schritt verursachten Änderungen farblich hervorzuheben.

In der Variablenansicht ist es sehr wichtig, die Gültigkeitsbereiche (scopes) der Variablen zu beachten. Das heißt, man muss immer überprüfen, ob die aktuelle Variable überhaupt noch im Programm existiert. Sonst muss der Debugger eine entsprechende Fehlermeldung anstatt des Variabenwertes anzeigen.

6.3.8 Evaluatoransicht (evaluator view)

Die Evaluatoransicht wird von vielen Debuggern zur Verfügung gestellt. In dieser Ansicht ist es möglich, beliebige mathematische Ausdrücke auszuwerten. Die Ausdrücke werden mit der einer Programmiersprache ähnlichen Syntax interpretiert und können die Variablen aus dem Programm, sowie Aufrufe von im Programm definierten Funktionen enthalten. Die Funktionsweise dieser Ansicht wird in den Abschnitten 6.4.7 und 6.4.8 beschrieben.

6.4 Bestandteile des Debuggers

In Kapitel 6.3 wurde verschiedene Ansichten präsentiert, die ein Debugger zur Verfügung stellen soll. In diesem Kapitell werden die wichtigste Bestandteile des Debuggers beschrieben, die die Implementierung dieser Ansichten erlauben.

6.4.1 Debugging Interface

Um alle Anforderungen, die wie in Kapitel 6.3 beschrieben, an einem Debugger gestellt werden, implementieren zu können, muss der Debugger tief in das System eingreifen. Da dies meistens sehr schwer und manchmal unmöglich ist, stellen fast alle Betriebssysteme extra eine Schnittstelle für Debugger zu Verfügung, das so genannte Debugging Interface.

Es gibt auch Bedingungen, die die Hardware, also den Prozessor, erfühlen muss, um das Debuggen überhaupt möglich zu machen. Hier werden erstmal minimale Anforderungen an den Prozessor und das Betriebssystem vorgestellt, danach werden weitere Dienste vorgestellt, die von Betriebssystem oder Prozessor angeboten werden können.

Als minimale Anforderungen an des System muss es die Möglichkeit geben, ein Breakpoint im Programmcode zu setzen und verschiedene Benachrichtigungen vom System zu abonnieren. Die Breakpoints sind das wichtigste Hilfsmittel, um den Ablauf der Applikation zu kontrollieren. In Kapitel 6.5 werden die entsprechenden Algorithmen vorgestellt, die Debugger verwenden, um das Programm schrittweise auszuführen. Es soll also möglich sein, einen Marker in den Programmcode zu setzen. Falls die entsprechende Stelle ausgeführt wird, soll das Betriebssystem die Applikation stoppen und eine Benachrichtigung über das Breakpoint-Ereignis an den Debugger schicken. Es gibt mehrere Möglichkeiten, dies zu realisieren. Es gibt Prozessoren, die über einen speziellen Breakpointbefehl verfügen. Falls es diesen Befehl nicht gibt, kann man eine unzulässige Sequenz von Bytes verwenden, die den gleichen Effekt wie der Breakpointbefehl erzeugt.

Um einen Breakpoint im Programmcode zu setzen, muss den Debugger den Ausschnitt des Programmcodes sichern und mit dem Breakpointcode überschreiben. Wenn ein Breakpoint aktiviert wird, kann der Debugger den ursprünglichen Programmcode wiederherstellen und dann entscheiden, ob er den Stopp beim Nutzer meldet oder das Programm weiterlaufen lässt.

Um, wie schon oben erwähnt wurde, auf Ereignisse reagieren zu können, braucht der Debugger eine Möglichkeit, verschiedene Benachrichtigungen des Betriebssystems zu abonnieren. Es werden folgende Benachrichtigungen benötigt:

- Benachrichtigung über ein Breakpoint-Ereignis. Diese Benachrichtigung wird benötigt, um auf Breakpoints zu reagieren.
- Benachrichtigung über ein Programmabsturz bzw. eine Ausnahmesituation (Exception). Diese Benachrichtigung wird benötigt, um die Fehlersituation an den Nutzer melden. Es soll die Stelle im Programmquellcode angezeigt werden, an der der Fehler auftrat.
- Benachrichtigung über einen Programmstart bzw. ein Programmende. Diese werden gebraucht, um die Breakpoints in den Programmcode einzubringen und alle Datenstrukturen zu initialisieren. Beim Programmstart werden auch weitere Benachrichtigungen

abonniert. Die Programmstoppbenachrichtigung wird benötigt, um die Datenstrukturen freizugeben und den Nutzer das Programmenden zu melden. Benachrichtigung über die Erzeugung bzw. Zerstörung eines Thread. Diese Benachrichtigungen werden benötigt, um Datenstrukturen anzupassen, falls man es mit einer so genannte nebenläufigen Applikation zu tun hat (siehe Abschnitt 6.6.1). Diese

- Benachrichtigung kann gebraucht werden, um zum Beispiel, das Auftreten eines Deadlocks von Threads zu erkennen.

Es ist möglich, dass das Betriebssystem noch weitere Dienste für den Debugger anbietet. Ein möglicher Dienst ist zum Beispiel die Unterstützung von Breakpoints auf Betriebssystemebene. Ein weiteres Beispiel für solche Dienste ist ein so genannter Single Step Befehl. Hier geht es darum, bei einer gestoppten Applikation genau einen Prozessorbefehl auszuführen und wieder zu stoppen. Falls dieser Befehl nicht zu Verfügung steht, kann er mit Hilfe der Breakpoints realisiert werden (siehe Abschnitt 6.5.1).

Es ist für den Debugger sehr hilfreich, falls ein sogenannter Databreakpoint von Prozessor unterstützt wird. Ein Databreakpoint ist ein Marker im Speicherbereich oder an einer Variablen. Die Ausführung von dem Programm wird gestoppt, falls der markierte Speicherbereich geändert wird. Es ist besonders hilfreich, falls man mit dem Arrayüberlauf-Fehler zu tun hat. Die Implementierung des Databreakpoints ohne Prozessorunterstütung ist sehr schwierig (siehe Abschnitt 6.4.3).

Ein Beispiel für einen weiteren Dienst, der vom Betriebssystem zu Verfügen gestellt werden kann, ist eine Benachrichtigung bei Systemaufrufen. Das bedeutet, dass der Debugger benachrichtigt werden kann, wann welche Systemaufrufe von der Applikation angestoßen werden. Man bekommt eine Benachrichtigung, bevor der Systemaufruf stattfindet und eine Benachrichtigung direkt danach. Der Debugger hat im ersten Fall die Möglichkeit, die Parameter, die an den Systemaufruf übergeben werden, und im zweiten Fall das Ergebnis des Systemaufrufes auszulesen und zu verändern. Diesen Dienst kann der Debugger nutzen, um die Umgebung der Applikation gezielt zu verändern. Damit kann etwa das Verhalten der Applikation in einer künstlich erzeugten Umgebung getestet werden.

6.4.2 Debug-Information

Die Debug-Information ist eine zusätzliche Information, die während der Übersetzung vom Compiler optional erzeugt und gespeichert wird. Um die Debug-Information nutzen zu können, muss der Debugger das Format, in welchem diese Information vorliegt, erkennen und die gespeicherte Information auf dem Datenträger finden. Die zweite Aufgabe wird oft vereinfacht, indem man die Debug-Information direkt in der ausführbare Datei speichert. Der Debugger muss aber auch korrekt funktionieren, falls keine Debug-Information gefunden werden konnte. Es ist selbstverständlich, dass ohne der dazu nötigen Information nicht alle Funktionen von Debugger zu Verfügung stehen können.

Die Debug-Information beinhaltet normalerweise folgendes:

- Eine Abbildung vom Programmquelltext auf Programmbinärecode. Diese Abbildung wird benötigt, um oben beschriebene Illusion der Ausführung des Programmquelltex-

tes darzustellen. Hier wird beschrieben, welche Abschnitte von Binärcode für welche Zeilen im Programmquellcode stehen.

- Eine Liste aller Variablen. Die Liste aller Variablen wird genutzt, um den Kontext der Applikation darzustellen. Außerdem braucht man die Liste aller Variablen, um die Ausdrücke in der Evaluatoransicht richtig zu interpretieren.

- Eine Abbildung von Variablen auf den Datenspeicher oder Stapel. Diese Abbildung wird benötigt, um Werte der Variablen auszulesen oder zu ändern. Der Speicherort jeder Variable kann zeitabhängig sein, sich also mit der Zeit ändern. Falls etwa die Variable in einem Programmabschnitt häufig zugegriffen wird, kann den Kompiler sich dazu entscheiden, innerhalb dieses Abschnittes die Variable in einem Prozessorregister zu speichern.

- Die Information über im Programm definierte Typen. Diese Information wird von Debuggern benutzt, um die Belegung der Variablen richtig zu interpretieren. Außerdem kann man aus dieser Information berechnen, wieviel Speicher die Variable benötigt. Konstanten.

- Die Konstanten werden hauptsächlich für die Auswertung der Ausdrücke in der Evaluatoransicht benutzt.

6.4.3 Breakpoints

Breakpoints sind das wichtigste und manchmal das einzige Hilfsmittel eines Debuggers, um das Programm zu steuern. Sie sind aber auch ein sehr wichtiges Hilfsmittel für den Nutzer, um in dem Programm Fehler zu suchen. Debugger und Benutzer nutzen oft verschiedene Typen von Breakpoints. Der Debugger nutzt temporäre Breakpoints, die beim Ausführen eines Assemblerbefehls ausgelöst werden. Die Nutzer sind öfter an dauerhafte Breakpoints interessiert, die sich beim Ausführen einer Programmquelltextzeile auslösen. Die Debugger unterstützen oft verschiedene Optionen für die Breakpoints. Daraus entstehen folgende Typen von Breakpoints:

- Breakpoint auf einer Zeile des Programmquelltextes. Stoppt die Ausführung des Programms direkt bevor die Zeile mit dem Breakpoint ausgeführt wird.

- Breakpoint mit Schwelle (threshold). Die Ausführung des Programms wird gestoppt, wenn die Zeile mit solch einem Breakpoint eine bestimmte Anzahl von Durchläufen erreicht hat.

- Breakpoint mit einer Bedingung. Stoppt die Ausführung des Programms direkt bevor die Zeile mit dem Breakpoint ausgeführt wird, wenn die Bedingung erfüllt ist.

- Databreakpoint. Stoppt die Ausführung des Programms, wenn eine markierte Variable oder ein markierter Speicherbereich modifiziert wird.

Einige Typen von Breakpoints können auch kombiniert werden. Es ist auch erlaubt, mehrere Breakpoints auf die gleiche Programmquellcodezeile zu setzen. Das macht Sinn, wenn die Breakpoints von unterschiedlicher Art sind oder verschiedene Bedingungen bzw. Threasholdwerte haben.

Falls eine Unterstützung für Databreakpoints vom Prozessor nicht vorhanden ist, verlangsamt sich die Ausführung einer Applikation bei einigen Debuggern drastisch, wenn man einen Databreakpoint setzt. Das liegt daran, dass der Debugger nach jedem Schritt der Applikation überprüfen muss, ob sich die markierte Variable oder der markierte Speicherbereich verändert hat.

6.4.4 Programmstapel

Wie schon ober diskutiert wurde, ist der Programmstapel ein wichtiger Bestandteil des Kontexts. Für den Debugger ist es deswegen sehr wichtig, den Programmstapel richtig interpretieren (parsen) zu können, um ihn darstellen zu können. Allgemein ist dies eine schwierige Aufgabe. Wie bereits oben beschrieben wurde, besteht der Stapel aus Frames. Jeder Frame entspricht einer Instanz einer Funktion. Ein Frame enthält viele Informationen, die beim Debuggen für den Nutzer interessant sein können, zum Beispiel die Belegung der lokalen Variablen. Der Aufbau eines Frames hängt von Prozessortyp und Compiler ab. Zusätzlich kann der Aufbau eines Frames für verschiedene Funktionen verschieden sein. Wie der Frame für eine bestimmte Funktion aufgebaut ist, wird in der Debug-Information beschrieben.

Beim Interpretieren vom Programmstapel muss der Debugger davon ausgehen, dass der Stapel wegen eines Programmfehlers fehlerhaft oder komplett zerstört sein kann. Es gibt verschiedene Verfahren, wie man in solchen Fällen die Information, zu mindest aber den Anfang und das Ende des Stapels, wiederherstellen kann.

Es sei nur bemerkt, dass in Java Debug API diese Aufgabe, den Programmstapel zu untersuchen, glücklicherweise von der Java Virtual Maschine erledigt wird.

6.4.5 Disassemblierung

In der Disassembleransicht soll es möglich sein, das Programm auf Maschinenspracheebene zu debuggen. Es sollte zu sehen sein, wie das Programm vom Compiler übersetzt wurde und wie das Programm in Wirklichkeit ausgeführt wird. Um das zu realisieren ist, es unbedingt erforderlich, den Programmbinärcode zu interpretieren.

Diese Aufgabe ist stark vom Prozessortyp abhängig. Die RISC-Prozessoren haben einen kürzeren Befehlssatz mit fester Länge, was den Umgang mit dem Programmcode für den RISC-Prozessoren erleichtert. Insbesondere ist es möglich, für einen gegebenen Befehl den Anfang eines vorherigen Befehls zu ermitteln, was in der Assembleransicht benötigt wird.

Der Umgang mit einem Programmcode für den CISC-Prozessor (zum Beispiel, Intel x86) ist es im Gegensatz schwerer, da die Befehle verschiedene Länge haben und es Suffixe, die die Bedeutung des nachfolgenden Befehls verändern, gibt.

6.4.6 Gültigkeitsbereiche der Variablen

Um glaubwürdige Information anzeigen zu können, muss der Debugger die Gültigkeitsbereiche der Variablen (variable scope) richtig erkennen. Es soll auch berücksichtigt werden, dass die Variablen „sterben" und wieder „geboren" werden können. Das heißt, dass es einen bestimmten Bereich im Programmcode geben kann, an den Variablen nicht existieren, obwohl sie vor und nach diesem Bereich einen Wert haben. Die Information über dir Gültigkeitsbereiche der Variablen ist in den Debug-Information enthalten.

Der Debugger sollte auch in der Lage sein, zwischen verschiedenen Gültigkeitsbereichen bzw. Kontexten wechseln zu können. Das ist sehr nützlich, falls man den Stapel eine oder mehrere Ebenen tief durchsuchen will, um die Werte der Variables zu überprüfen.

6.4.7 Evaluation der Ausdrücke

Um oben beschriebenen Evaluatoransicht implementieren zu können, muss der Debugger einen Ausdruck evaluieren können. Der Ausdruck soll in der gleichen Programmiersprache vorliegen, in der der Applikation geschrieben wurde, und kann manchmal mit Hilfe des Compilers interpretiert werden. Danach kann der Debugger die in dem Ausdruck vorkommenden Variablen durch ihre aktuellen Werte ersetzen und dem Compiler den fertigen Ausdruck evaluieren lassen. Diese Methode ist besser, als in Debugger den Evaluation neu zu implementieren, da es immer Unterschiede zwischen dem originalen Interpreter von dem Compiler und dem Interpreter vom Debugger geben kann, was zu zusätzlichen Fehlern führen kann.

Bei der Evaluation der Ausdrücke muss immer beachtet werden, dass alle in dem Ausdruck enthaltene Variablen in dem aktuell gewählten Gültigkeitsbereich existieren und bezüglich dieses Gültigkeitsbereichs evaluiert werden. Bei der Implementierung des Evaluators muss auch berücksichtigt werden, dass bei jedem Stopp mehrere Ausdrücke evaluiert werden müssen. Sie soll also effizient implementiert werden.

6.4.8 Aufrufe der Funktionen während Evaluation

Es ist oft sehr hilfreich, wenn der Debugger auch Funktionen aus dem Programmquelltext im aktuellen Kontext während der Evaluation ausführen kann. Statt den Code der Funktion zu interpretieren wird häufig der Programmbinärcode einfach mit voller Geschwindigkeit auf dem Prozessor gestartet. Dafür wird ein neuer Stapelframe aufgebaut, in dem die Parameter für die Funktion gespeichert werden. Dann wird ein temporärer Breakpoint ans Ende der Funktion gesetzt und die Funktion wird gestartet. Nach dem Stopp wird das Ergebnis zwischengespeichert und der ursprüngliche Zustand der Applikation wiederhergestellt. Der erzielte Wert wird für die weitere Evaluierung verwendet.

6.5 Step-Algorithmen

Es werden hier wichtige Algorithmen vorgestellt, die von Debuggern benutzt werden, um der Applikation schrittweise auszuführen. Viele davon sind sehr schwer zu implementieren, selbst wenn die Beschreibung einfach aussieht. Das liegt daran, dass man, um einen solchen

Algorithmus auszuführen, mehrmals auf verschiedene Benachrichtigungen warten muss. Es muss also bei einem Stopp überprüft werden, in welchem Zustand sich der Debugger befindet und welcher Algorithmus gerade ausgeführt wird. Falls der Stopp wegen eines Absturzes oder eines benutzerdefinierten Breakpoints stattfand, muss man die Ausführung des Algorithmus abbrechen, alle angeforderte Ressourcen freigeben und alle temporäre Breakpoints entfernen.

6.5.1 Single Step Algorithmus

Single Step Algorithmus bedeutet „Führe einen CPU Schritt aus und stoppe". Das ist ein grundlegender Algorithmus, der für viele andere komplexere Algorithmen benutzt wird. Es sollte möglich sein, aus der Disassemblieransicht diesen Algorithmus direkt anstoßen zu können.

Falls der Prozessor und das Betriebssystem diesen Befehl nicht direkt anbieten, muss der Debugger ihn selbstständig realisieren. Er kann folgendermaßen mit Hilfe von Breakpoints umgesetzt werden. Es wird der nachkommende Maschinenbefehl untersucht, um die Stelle herauszufinden, wo der nächsten Befehl sein kann. Es ist entweder direkt der nächste Befehl (bei nicht Sprung-Befehlen), oder ein woanders liegender Befehl (bei Sprung-Befehlen) oder beides (bei bedingten Sprung-Befehlen). Es wird auf die nächste mögliche Befehle Breakpoints gesetzt und die Applikation wird gestartet. Falls die nachfolgenden Befehle richtig ermittelt wurden und der aktuelle Befehl fehlerfrei ausgeführt wurde, stoppt der Prozessor an einem der gesetzten Breakpoints, nachdem er genau einen Befehl ausgeführt hat. Am Ende sollte der Debugger die benutzten Breakpoints entfernen.

6.5.2 Step Into Algorithmus

Step Into bedeutet „Führe nächste Zeile des Programmquelltexts aus und stoppe. Falls eine über den Programmquellcode verfügbare Funktion aufgerufen wird, stoppe an der ersten Zeile dieser Funktion". Dieser Algorithmus wird sehr häufig von Benutzern verwendet und ist ein Bestandteil jedes Debuggers. Hier geht es darum, eine Illusion zu schaffen, dass der Programmquelltext direkt ausgeführt wird.

Eine naive Methode ihn zu implementieren wäre folgende: Führe den Single Step Algorithmus aus. Prüfe an jede Stelle im Binärcode, ob man sich am Anfang irgendeiner Zeile im Programmquelltext befindet. Falls dies der Fall ist, stoppe, sonst fange von vorne an. Es ist klar, dass diese Methode nicht effizient sein kann.

Eine andere Idee besteht darin, mehrere Breakpoints zu setzen und das Programm mit voller Geschwindigkeit laufen zu lassen. Es reicht, wenn man einen Breakpoint auf jede Zeile in der aktuellen Funktion und auf die erste Zeile jeder Funktion, für die Programmquellcode existiert, setzt. Man muss ein Breakpoint auf die erste Zeile jeder Funktion setzen, da man analytisch nicht herausfinden kann, welche Funktion aufgerufen wird. Das hängt damit zusammen, dass es vorkommen kann, dass man einen Systemaufruf macht, der selbst über einen Funktionszeiger eine über den Programmquelltext verfügbare Funktion aufruft. Der Algorithmus ist immer noch nicht ganz optimal, da es sein kann, dass man sehr viele Funktionen im Programm oder sehr viele Programmzeilen in der aktuellen Funktion hat.

Allgemein ist es sehr schwierig, einen optimalen Algorithmus für „Step Into" zu entwerfen. Einige Debugger verzichten auf die korrekte Implementierung des Step Into Algorithmus.

6.5.3 Step Over Algorithmus

Step Over bedeutet „Führe nächste Zeile des Programmquelltext aus und stoppe. Dabei sollen alle inzwischen aufgerufenen Funktionen mit volle Geschwindigkeit ausgeführt werden".

In Vergleich mit dem Step Into Algorithmus ist der Step Over Algorithmus einfacher zu realisieren. Es gibt mehrere Strategien, die man verfolgen kann. Man kann jeden einzelnen Befehl ausführen und die Situation analysieren, was nach wie vor sehr ineffizient ist. Man kann aber auch auf jede Zeile der aktuellen Funktion einen Breakpoint setzen und die Applikation ausführen lassen.

6.6 Debugging besonderer Arten von Applikationen

Hier werden einige spezielle Einsätze von Debuggern beschrieben, die für besondere Arten von Anwendungen hilfreich sind: nebenläufige Applikationen, GUI-Anwendungen und Applikation mit optimiertem Code.

6.6.1 Debugging von nebenläufigen Applikationen

Eine nebenläufige Applikation ist eine Applikation mit mehreren Threads (Ausführungsfaden), die zeitgemäss parallel ausgeführt werden. Jedes Programm hat am Anfang ein Thread. Falls das Programm ein weiteres Thread erzeugt, wird ein neuer Ausführungspfad gestartet, der parallel bzw. gleichzeitig mit dem ursprünglichen Thread läuft. Alle Threads teilen alle globale Variablen, haben aber eigene lokale Variablen.

Eine der Aufgaben eines Debuggers beim Debuggen von nebenläufigen Applikationen ist, das Programm theadsweise zu analysieren und zu steuern. Der Debugger muss in der Lage sein, mehrere Threads zu verwalten, Thread zu stoppen, zu starten und den Kontext des Threads anzuzeigen.

Dabei müssen mehrere Probleme berücksichtigt werden: So müssen etwa beim Stoppen einer Applikation alle Threads gestoppt werden. Es können komplizierte Beziehungen zwischen den Threads existieren. Die Threads können auf andere Threads warten. Da der Debugger immer davon ausgehen muss, dass die Applikation fehlerhaft sein kann, muss er auch fehlerfrei weiter funktionieren, falls die Applikation in einen Deadlockzustand gerät. Es ist auch sicherlich hilfreich, wenn der Debugger die Beziehungen zwischen Threads analysieren und anzeigen kann. Dann kann die Deadlockerkennung, die die Deadlocksituation dem Nutzer meldet, in den Debugger eingebaut werden.

6.6.2 Debugging von GUI-Applikationen

Beim Debugging von GUI-Applikationen entstehen zusätzliche Probleme und Anforderungen an den Debugger. Es ist schwierig, eine GUI-Applikation beim Debuggen nicht zu beeinflussen, da der Debugger die gleiche GUI-Schnittstelle vom Betriebssystem nutzt und womit zwangsläufig die Umgebung der Applikation ändert. Wenn etwa ein Breakpoint aktiviert wird,

bekommt der Debugger den „Fokus", er wird also aktiv. In diesem Fall schickt das System eine Benachrichtigung an die zu debuggende Applikation, dass sie nicht mehr aktiv ist. Ohne Einsatz eines Debuggers bekäme die Applikation keine solche Benachrichtigung. Ein Ausweg hier ist es, einen Debugger mit der Fernoption (remote) zu realisieren. Es soll also möglich sein, auf dem Rechner, wo die Applikation läuft, ein kleines Programm ohne GUI laufen zu lassen, das mit dem GUI Debugger auf einem anderen Rechner eine Verbindung aufbaut und das Debugging ermöglicht.

Anderseits kann der Debugger für GUI-Applikationen zusätzliches Hilfsmittel anbieten, zum Beispiel Eventbreakpoints. Das sind Breakpoints, die sich beim Empfang einer bestimmten Benachrichtigung aktivieren.

6.6.3 Debugging von optimiertem Code

Manchmal ist es sehr interessant, optimierten Code zu debuggen, da es zum Beispiel passieren kann, dass ein Fehler nur im optimierten Code eintritt. Einen optimierten Code zu debuggen ist eine große Herausforderung für den Debugger. Je nach Compiler und Optimierungsstrategien können verschiedene schwierige Probleme für den Debugger entstehen. In Java ist das Debuggen von optimiertem Code jedoch gar nicht nötig oder sogar möglich, da das Debuggen auf Programmquellcode oder höchstens auf Bytecode-Ebene passiert. Die Optimierung wird aber von JVM JIT Compiler auf einer darunterliegende Ebene durchgeführt, auf die der Debugger nicht zugreifen kann.

6.7 Fazit

Ein Debugger ist ein sehr nützliches und erforderliches Hilfsmittel für den Softwareentwickler. Es stellt aber andererseits eine große Herausforderung für den Debuggerentwickler dar, da man mit dem Betriebssystem, dem Prozessor, der fehlerhaften Applikation und mit dem Nutzer gleichzeitig arbeiten muss. Außerdem darf der Debugger nicht viel Speicher und Prozessorzeit beanspruchen, da dadurch das Debuggen stark verlangsamt werden kann. Der Debugger darf auch selbst keine Fehler enthalten, da es die Suche nach Fehlern erschwert. Es müssen teilweise schwierige algorithmische Probleme gelöst werden. Außerdem, um Information übersichtlich repräsentieren zu können, soll der Debugger über eine übersichtliche GUI-Schnittstelle verfügen.

KAPITEL 7

Mathematische Grundlagen der 3D Modellierung

Carina Klar

7.1 Vektoren

In der Computergraphik sind Vektoren von Bedeutung, da alle Objektmengen des dreidimensionalen Raums durch Punktmengen repräsentiert werden. Diese wiederum entsprechen Vektoren, die den Gesetzen der linearen Algebra unterliegen.

7.1.1 Vektor

Ein Vektor v ist ein Element des n-dimensionalen Euklidischen Zahlenraums $\mathbb{R}^n$. Also

$$v \in \mathbb{R}^n \Leftrightarrow \text{v} = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} \text{ mit } x_i \in \mathbb{R}, i = 0, \cdots, n-1.$$

Ein Vektor kann sowohl als Spaltenvektor als auch als Zeilenvektor geschrieben werden.

Um die Länge eines Vektors berechnen zu können, benutzt man die Norm.

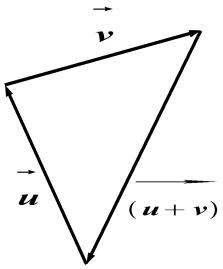
7.1.2 Norm

Die Norm $||v||$ eines Vektors v ist eine nichtnegative Zahl definiert gemäß

$$||v|| = \sqrt{v \cdot v} = \sqrt{\sum_{i=0}^{n-1} v_i^2}$$

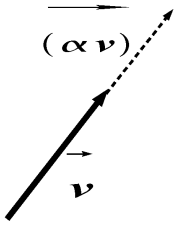
7.1.3 Verknüpfungen

Fr Vektoren gibt es zwei wichtige Verknüpfungen, eine ist die Vektoraddition, die andere die Skalarmultiplikation. Sollen die beiden Vektoren $u, v \in \mathbb{R}^n$ addiert werden, ergibt sich



$$u+v = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-1} \end{pmatrix} + \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} = \begin{pmatrix} u_0+v_0 \\ u_1+v_1 \\ \vdots \\ u_{n-1}+v_{n-1} \end{pmatrix} \in \mathbb{R}^n.$$

Zusätzlich kann ein Vektor $v \in \mathbb{R}^n$ mit einer skalaren Größe $\alpha \in \mathbb{R}$ multipliziert werden. Der entstehende Vektor setzt sich durch



$$\alpha \cdot v = \begin{pmatrix} \alpha v_0 \\ \alpha v_1 \\ \vdots \\ \alpha v_{n-1} \end{pmatrix} \in \mathbb{R}^n$$

zusammen. Je nach Wert der skalaren Größe α wird der Vektor v gestreckt oder gestaucht.
Schon durch diese beiden elementaren Operationen auf Vektoren lassen sich grundlegende Begriffe der linearen Algebra definieren.

7.1.4 Linearkombination, lineare Unabhängigkeit

Ein Vektor $\xi \in \mathbb{R}^n$ heißt Linearkombination der Vektoren $x, y, z \in \mathbb{R}^n$, falls

$$\xi = \alpha x + \beta y + \gamma z$$

gilt, also wenn sich also ξ durch die drei Vektoren x, y, z darstellen lässt. Man sagt dann, dass die Menge der Vektoren ξ, x, y, z linear abhängig ist.

Eine Menge linear unabhängiger Vektoren hat hingegen die Eigenschaft, dass sich keines der Elemente durch die anderen zusammensetzen lässt. Um formal eine Menge auf lineare Unabhängigkeit zu untersuchen, muss die folgende Implikation auf Erfüllbarkeit getestet werden:

$$\alpha x + \beta y + \gamma z + \delta \xi = 0 \Rightarrow \alpha = \beta = \gamma = \delta = 0.$$

7.1.5 Basis

Eine Menge linear unabhängiger Vektoren ist eine Basis $b_1, \cdots, b_n \in \mathbb{R}^n$ eines Vektorraums, falls jeder andere Vektor des Vektorraums eine Linearkombination der Basisvektoren ist. Das heißt also, dass die Basisvektoren den gesamten Vektorraum aufspannen.

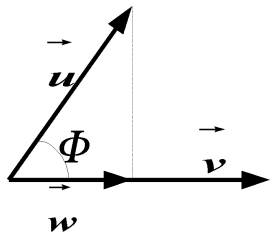
Stehen zudem noch die Basisvektoren senkrecht aufeinander und haben sie alle die Länge eins, so wird diese Basis Orthonormalbasis genannt.

Natürlich können zwei Vektoren auch miteinander multipliziert werden. Hierzu gibt es zwei Möglichkeiten: die Skalarmultiplikation und das Kreuzprodukt.

Skalarmultiplikation Bei der Skalarmultiplikation ist das Produkt eine skalare Größe. Sie berechnet sich durch

$u \cdot v = \sum_{i=1}^{n} u_i \cdot v_i = ||u|| \cdot ||v|| \cdot \cos\varphi.$

Am letzten Term der Gleichung kann die geometrische Bedeutung des Skalarprodukts erkannt werden. Die Länge des auf den Vektor v projizierten Anteils von u multipliziert mit der Länge von v resultiert im Skalarprodukt. So kann man auch einsehen, dass das Skalarprodukt zweier senkrecht aufeinander stehenden Vektoren gleich Null ist. Dann ist nämlich die Länge des auf den Vektor v projizierten Anteils von u und somit der

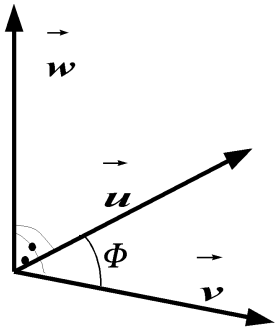


ganze Ausdruck gleich Null.

Kreuzprodukt Werden zwei Vektoren mit Hilfe des Kreuzprodukts miteinander multipliziert, erhält man einen auf beiden Vektoren senkrecht stehenden dritten Vektor. Dieser reprï¿½entiert den Flächeninhalt des von den beiden ursprünglichen Vektoren aufgespannten Parallelogramms. Formal definiert sich das Kreuzprodukt wie folgt:

$$w = u \times v = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix},$$

wenn $u, v, w \in \mathbb{R}^n$ drei Vektoren sind.



7.2 Matrizen

In der 3D-Modellierung müssen oft Objekte verschoben, rotiert oder skaliert werden. Man verwendet Matrizen, um diese Operationen durchzufhren. Das bedeutet, es können sowohl die Objekte in 3D als auch Transformationen auf diesen Objekten durch Matrizen beschrieben werden. Begriffe und Operationen, die dafür benötigt werden, werden hier vorgestellt.

7.2.1 Matrix

Eine $m \times n$-Matrix ist wie folgt aufgebaut: $\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \ddots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m1} & \cdots & \cdots & a_{mn} \end{pmatrix}$.

Dabei bezeichnet A_{i*} die i-te Zeile und A_{*j} die j-te Spalte der Matrix A.

7.2.2 Multiplikation Matrix-Vektor

Um eine $m \times n$ Matrix A mit einem Spaltenvektor x zu multiplizieren geht man wie folgt vor:

$A \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix}$, wobei $r_i = A_{i*} \cdot x = \sum_{k=1}^{n} a_{ik} \cdot x_k$.

Diese Operation kann als Spezialfall des Produkts zweier Matrizen angesehen werden.

7.2.3 Matrizenmultiplikation

Damit eine Multiplikation von zwei Matrizen definiert ist, muss die Anzahl der Zeilen der ersten Matrix gleich der Anzahl der Spalten der zweite Matrix sein. Die einzelnen Komponenten der entstehenden Matrix setzen sich aus

$c_{ij} = A_{i*} \cdot B_{*j} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$

zusammen.

7.2.4 Determinante

Um mit Matrizen arbeiten zu können, werden noch Determinanten benötigt. Da wir uns nur im zwei-dimensionalen und drei-dimensionalen Raum bewegen, genügt es, auch nur Determinanten fr 2×2 und 3×3 Matrizen einzuführen. Diese berechnen sich wie folgt:

$$|M| = \begin{vmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{vmatrix} = m_{00}m_{11} - m_{01}m_{10}$$

und

$$|M| = \begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{vmatrix}$$

$$= m_{00}m_{11}m_{22} - m_{01}m_{12}m_{20} + m_{02}m_{10}m_{21} - m_{02}m_{11}m_{20} - m_{01}m_{10}m_{22} - m_{00}m_{12}m_{21}$$

7.2.5 Spezielle Matrizen

Oft tauchen Matrizen mit bestimmten Eigenschaften immer wieder auf.

So gibt es zum Beispiel die Einheitsmatrix

$$I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 1 \end{pmatrix}.$$

Wendet man diese Matrix auf ein Objekt an, so repräsentiert die Einheitsmatrix die identische Abbildung. Die Inverse einer Matrix hat die Eigenschaft, dass sie multipliziert mit der ursprünglichen Matrix Eins ergibt. Die Inverse einer Matrix lässt sich entweder aufgrund dieser Eigenschaft durch Lösen eines Gleichungssystems berechnen oder, indem man den Kehrwert der Determinante mit der Selbstadjungierten multipliziert. Das sieht für 2×2 und 3×3 Matrizen wie folgt aus:

$$A^{-1} = \frac{1}{|A|} \begin{pmatrix} m_{11} & -m_{01} \\ -m_{10} & m_{11} \end{pmatrix}$$

$$A^{-1} = \frac{1}{|A|} \begin{pmatrix} det(A_{11}) & -det(A_{21}) & det(A_{31}) \\ -det(A_{12}) & det(A_{22}) & -det(A_{32}) \\ det(A_{13}) & -det(A_{23}) & det(A_{33}) \end{pmatrix}$$

A_{ij} entsteht aus A durch Streichen der i-ten Zeile und j-ten Spalte.

7.3 Objekte in 2D und 3D

7.3.1 Linien

Mit Linearkombinationen von zwei Vektoren kann man eine unendlich ausgedehnte Linie darstellen. Sei v ein Ortsvektor und d ein Richtungsvektor. Dann wird eine Linie durch

$l = v + \lambda \cdot d$, mit $-\infty < \lambda < +\infty$

definiert.

Um eine begrenzte Linie darzustellen, braucht man zwei Ortsvektoren. Sie kann dann durch

$l = v + \lambda(w - v)$, mit $0 \leq \lambda \leq 1$

ausgedrckt werden.

7.3.2 Ebenen

Eine Ebene kann man sich als von zwei Linien aufgespannt vorstellen. Analog zur Linie ist dann die Parameterform einer Ebene

$p = u + \alpha \cdot v + \beta \cdot w$,

mit $-\infty < \alpha, \beta < +\infty$.

Allerdings ist hier zu beachten, dass die beiden Richtungsvektoren voneinander linear unabhängig sind, da ansonsten keine Ebene aufgespannt wird. Eine für weitere Rechnungen geeignetere Darstellungsform von Ebenen ist die implizite Form. Hier wird der Ortsvektor u aus der Parameterform und ein beliebiger Punkt q der Ebene benutzt. Der Vektor zwischen diesen beiden Punkten liegt offensichtlich in der Ebene. Multipliziert man diesen Vektor $(q - u)$ mit

dem Normalenvektor der Ebene, ist das Ergebnis Null. Das liegt daran, dass der Normalenvektor senkrecht auf dem anderen Vektor steht und das Skalarprodukt von senkrechten Vektoren Null liefert. Die implizite Form einer Ebene lautet also

$(q-u)\cdot n = 0 \Leftrightarrow q\cdot n - u\cdot n = 0.$

Man kann diese Gleichung noch mit $|n| = 1$ normieren. Dann entspricht die Konstante $d = u\cdot n$ dem Abstand der Ebene zum Nullpunkt.

Im dreidimensionalen Raum interessiert man sich oft für Durchstoßpunkte und Kollisionen von Objekten. Dazu ist es nötig, den Schnitt einer Linie mit einer Ebene beziehungsweise zweier Ebenen berechnen zu können.

7.3.3 Schnitt Linie-Ebene

Im ersten Fall substituiert man die Punkte der Linie $l = v + \lambda\cdot w$ in der Definition der Ebene in impliziter Form $q\cdot n = d$. Dann ergibt sich aufgelöst nach λ ein Schnittpunktsparameter

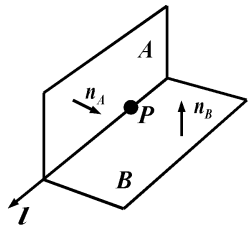
$\lambda_s = \frac{d-n\cdot v}{n\cdot w}$.

Den Schnittpunkt p_s erhält man durch Einsetzen von λ_s in die Liniengleichung:

$p_s = v + \lambda_s\cdot w.$

7.3.4 Schnitt Ebene-Ebene

Das zweite Problem ist etwas aufwendiger zu lösen. Wieder seien die beiden Ebenen in impliziter Form gegeben: $E_1 = n_1\cdot p - d_1 = 0$ und $E_2 = n_2\cdot p - d_2 = 0$. Die Schnittlinie dieser beiden Ebenen liegt natürlich in beiden Ebenen. Damit stehen die beiden Normalenvektoren n_1 und n_2 senkrecht auf der Schnittlinie. Der Richtungsvektor der Schnittlinie kann damit durch das Kreuzprodukt der beiden Normalenvektoren ausgedrückt werden, das heißtï¿½ $\frac{1}{2}\ l = n_1 \times n_2$, denn das Kreuzprodukt steht senkrecht auf den beiden Vektoren. Um nun die gesamte Gleichung der Schnittlinie aufstellen zu können, muss noch ein weiterer Punkt auf der Schnittlinie gefunden werden. Dazu definiert man eine dritte Ebene, die senkrecht zum Richtungsvektor der Schnittlinie steht und den Nullpunkt schneidet. Das bedeutet, dass diese Ebene parallel zum Richtungsvektor l der Schnittlinie steht. Die Ebenengleichung der dritten Ebene lautet dann
$l_x\cdot x + l_y\cdot y + l_z\cdot z = 0$
Löst man noch das Gleichungssystem bestehend aus den drei Ebenengleichungen, ergibt sich ein Punkt $P = (x,y,z)$ auf der Schnittlinie. Damit ist die Gleichung der Schnittlinie eindeutig bestimmt.

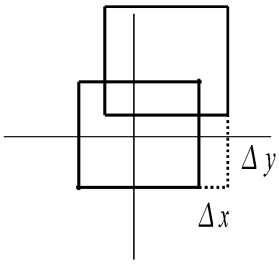


7.4 Anwendungen

Es gibt viele Operationen, die auf Objekten im dreidimensionalen Raum durchgefhrt werden können. Einige davon sollen hier beschrieben werden. Ziel ist es, jede beliebige Transformation durch Hintereinanderausführung elementarer Operationen realisieren zu können. Jede einzelne Transformation wird durch eine Matrix ausgedrückt.

7.4.1 Translation

Bei einer Verschiebung eines Objekts werden die Koordinaten der Punkte dieses Objekts mit einem Richtungsvektor addiert. Das führt zu diesem Gleichungssystem:



$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+\Delta x \\ y+\Delta y \\ z+\Delta z \\ 1 \end{pmatrix}$$

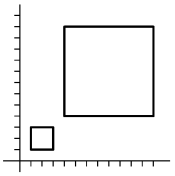
und damit zu der Translationsmatrix

$$T = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

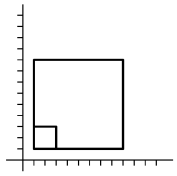
7.4.2 Skalierung

Um ein Objekt zu stauchen oder zu strecken, wird der Ortsvektor mit dem Skalierungsfaktor multipliziert. Das wird durch diese Matrix erreicht:

$$\begin{pmatrix} F_x & 0 & 0 & 0 \\ 0 & F_y & 0 & 0 \\ 0 & 0 & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

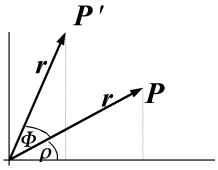

Uniforme Skalierung

Es gibt zwei verschiedenen Arten von Skalierung. Im ersten Fall sind alle Skalierungsfaktoren gleich. Diese wird uniforme Skalierung genannt. Das Resultat ist, dass mit der Skalierung auch gleichzeitig eine Verschiebung stattfindet. Bei der nicht-uniformen Skalierung entsteht keine Verschiebung, wenn einer der Skalierungsfaktoren Eins ist. Allgemein bedeutet die nicht-uniforme Skalierung, dass die Skalierungsfaktoren unterscheidliche Werte haben.
Durch die Skalierung kann auch eine Spiegelung erreicht werden. Dazu muss nur mindestens einer der Skalierungsfaktoren negativ sein. Setzt man einen Faktor auf Null, so erhält man eine Projektion auf eine Koordinatenachse. Darauf wird später noch genauer eingegangen.


Nicht-uniforme Skalierung

7.4.3 Rotation

Die nächste interessante Transformation ist die Rotation. Hierzu bietet sich die Darstellung der Punkte durch Polarkoordinaten an. Die Darstellung des Punktes $P = (x,y)$ durch Polarkoordinaten zum Beispiel lautet $x = r \cdot \cos\varphi$ und $y = r \cdot \sin\varphi$. Die Variable r steht für den Abstand des Punktes vom Ursprung. Der Winkel φ ist der Winkel, den der Vektor $\overline{OP}$ zwischen Ursprung und dem Punkt P mit der x-Achse einschließt. Dann lässt sich ein um ϕ gedrehter Punkt $P' = (x',y')$ darstellen durch $x' = r \cdot \cos(\varphi + \phi)$ und $y' = r \cdot \sin(\varphi + \phi)$.
Durch Anwendung trigonometrischer Sätze ergibt sich daraus $x' = x \cdot \cos\phi - y \cdot \sin\phi$ und $y' = x \cdot \cos\phi + y \cdot \sin\phi$.



Aus den hier errechneten Koeffizienten kann man die Rotationsmatrizen für die Rotation im dreidimensionalen Raum aufstellen.

Die Drehung um die x-Achse wird reprßentiert durch

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

die Drehung um die y-Achse durch

$$\begin{pmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und die Drehung um die z-Achse durch

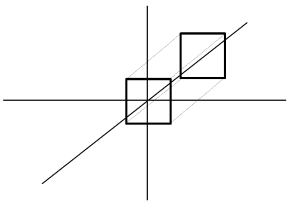
$$\begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

7.4.4 Projektion

Als letzte Transformation soll die Projektion vorgestellt werden. Es sollen hier zwei verschiedene Arten besprochen werden.

Zum einen die orthogonale Projektion. Wie schon bereits erwähnt, ist dies eine spezielle Skalierung, bei der ein Skalierungsfaktor auf Null gesetzt wird. Die Matrix

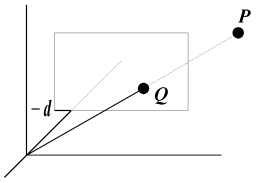
$$P_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



erreicht zum Beispiel eine Projektion in die x/y-Ebene. Eine wichtige Eigenschaft der orthogonalen Projektion ist, dass parallele Linien nach der Projektion parallel bleiben.

Die andere Art der Projektion ist die perspektivische Projektion. Der Augenpunkt befindet sich in diesem Fall im Ursprung des Koordinatensystems. Um das Prinzip der perspektivischen Projektion besser erklären zu können, wird angenommen, dass ein Punkt P in die Ebene $z = -d$ projiziert werden soll. Durch diese Transformation entsteht ein neuer Punkt Q mit den Koordinaten $Q = (q_x, q_y, -d)$. Um explizit die Koordinaten des Punktes Q aus den Koordinaten des Punktes P zu berechnen, werden die Gesetze des Strahlensatzes benutzt. Dann gilt $\frac{q_x}{p_x} = \frac{-d}{p_z} \Leftrightarrow q_x = -d\frac{p_x}{p_z}$. Auf diese Weise lassen sich auch die übrigen Koordinaten des Punktes Q berechnen. Zusammengefasst erhält man für dieses Beispiel die Projektionsmatrix

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 1 \end{pmatrix}.$$



7.4.5 Konkatenation von Transformation

Nun können an einem Objekt mehrere Transformationen durchgeführt werden. Dazu werden die Transformationsmatrizen nacheinander von links multipliziert. Das resultierende Matrizenprodukt wird von links auf das zu transformierende Objekt angewendet. Zu beachten ist, dass die Matrizenmultiplikation nicht kommutativ ist. ï¿½dert man die Reihenfolge der Transformationsmatrizen, so ändert sich auch die resultierende Transformation.

7.4.6 Inverse Transformation

Inverse Transformationen erhält man durch Berechnung der Inversen der entsprechenden Transformationsmatrix T. Es gilt $T^{-1} \cdot T = 1$. Wie genau die Inverse berechnet wird, wurde am Anfang erklärt. Wichtig ist, dass die inverse Skalierung nur für Faktoren $\neq 0$ definiert ist. Deshalb gibt es für die Projektion keine eindeutige Umkehrung.

Um verknüpfte Transformationen zu invertieren, werden die Inversen der entsprechenden

Transformationsmatrizen in umgekehrter Reihenfolge angewendet, etwa

$$(T_4 \cdot T_3 \cdot T_2 \cdot T_1)^{-1} = T_1^{-1} \cdot T_2^{-1} \cdot T_3^{-1} \cdot T_4^{-1}$$

KAPITEL 8

Java3D - Eine Einführung

Henning Zeller

8.1 Einleitung

Ziel des vorliegenden Textes ist es, unter dem vorgegebenen Rahmen (Seminarausarbeitung) einen Einblick in die wichtigsten Konzepte von Java3D, einer Grafikbibliothek zur Darstellung und Manipulation von dreidimensionalen Szenen, zu ermöglichen. Die hier vorgestellten Inhalte basieren im wesentlichen auf dem Java3D-Tutorial von Sun (Bouvier (2001)) sowie dem Buch von Selman Selman (2002). Wir wollen zunächst in Kapitel 8.2 einige grundlegende Begriffe klären, um anschließend in knapper, überblicksartiger Form auf die Java3D-Klassenhierarchie einzugehen. In Kapitel 8.3 widmen wir uns dem fundamentalen Konzept von Java3D, dem sogenannten Szenegraphen. Kapitel **??** stellt einige Möglichkeiten vor, visuelle, geometrische Inhalte zu erzeugen. Animationen und interaktive Konzepte sind Gegenstand von Kapitel 8.5. Schließlich wollen wir in Kapitel 8.6 mögliche Stärken und Schwächen von Java3D reflektieren. Die vorgestellten Konzepte und Techniken sollen zudem durch möglichst anschauliche Beispiele vertieft werden.

8.2 Grundlagen

In diesem Kapitel werden wir uns zunächst mit der Frage beschäftigen, worum es sich bei Java3D überhaupt handelt. Abschließend werden wir kurz Bezug auf die Java3D-Klassenhierarchie nehmen.

8.2.1 Begriffsklärung

Java3D ist eine von Sun entwickelte optionale Erweiterung der Java2-Plattform, welche die Darstellung und Manipulation von dreidimensionalen, interaktiven Szenen ermöglicht. Dem

Entwickler werden dabei auf Basis einer High-Level-API Werkzeuge zur Arbeit mit dreidimensionalen, geometrischen Objekten zur Verfügung gestellt. Diese Objekte befinden sich innerhalb eines sogenannten virtuellen Universums, das gerendert werden soll. Das Rendering geschieht automatisch, wahlweise mit den Grafikbibliotheken OpenGL oder DirectX, die von Java3D beide unterstützt werden.

Mit Java3D verfolgte Sun hauptsächlich die Idee, objektorientierten Prinzipien Einzug in die Grafikprogrammierung zu gewähren. Während beispielsweise unter OpenGL eine 3D-Szene durch Ansammlungen von Punkten, Linien oder Dreiecken beschrieben wird, können wir diese unter Java3D im Sinne der Objektorientierung als eine hierarchische Anordnung von 3D-Objekten (z.B. Kugeln, Würfel oder Cylinder) betrachten. Hierdurch wird ein höheres Abstraktionsniveau erreicht, welches auch bei der Szenenoptimierung hilfreich sein kann.

8.2.2 Die Java3D-Klassenhierarchie - ein kurzer Überblick

Anstatt explizit einzelne Klassen der Java3D-API herauszugreifen, wollen wir hier im Sinne eines besseren Überblicks eine strukturelle Sichtweise angeben. Dabei lassen sich die Klassen der API folgendermaßen kategorisieren:

- **(primitive) 3D-Objekte bzw. Körper**

  Hierunter fallen Klassen zur Realisierung geometrischer Körper, wie sie z.B. in `javax.media.j3d.Geometry` zu finden sind. Dabei kann zwischen primitiven Körpern (unter Java3D sind dies Würfel, Kegel, Cylinder und Kugel) und komplexeren, zusammengesetzten Strukturen, die beispielsweise über Rasterdaten definiert werden können, unterschieden werden.

- **Mathematik**

  In der Grafikprogrammierung müssen unter anderem oft Operationen auf Vektoren oder Matrizen durchgeführt werden. Das Paket `javax.vecmath` stellt die hierfür benötigten Klassen zur Verfügung.

- **Interaktion und Animationen**

  Da es unter Java3D möglich ist, Szenen interaktiv zu gestalten (dem Benutzer also zum Beispiel durch Maus- und Tastaturkommandos eine Manipulation der Szene zu erlauben), sind entsprechende Werkzeuge und Routinen nötig. Diese finden sich hauptsächlich im Paket `j3d.utils.behaviors`.

- **Bestandteile des Szenegraphen bzw. des virtuellen Universums**

  Der Szenegraph ist das grundlegende Konzept zur dreidimensionalen Szenenbeschreibung unter Java3D, welchem wir uns im nun folgenden Kapitel etwas eingehender widmen wollen.

8.3 Das Szenegraph-Konzept

Wir wollen zunächst überlegen, wie sich ein Szenegraph charakterisieren lässt und welche Eigenschaften er erfüllen muss. Anschließend werden wir die konkreten Elemente eines Szenegraphen vorstellen, um darauf aufbauend den Zusammenhang zwischen Szenegraph und konkreten Java3D-Anwendungen zu erläutern.

8.3.1 Überblick

Ein Szenegraph ist ein gerichteter, kreisfreier Graph in hierarchischer Anordnung. Er besteht aus Instanzen von Java3D-Klassen, deren Zusammenspiel die Beschaffenheit der zu rendernden Szene beschreibt. Hier sind z.B. die Anordnung der geometrischen Objekte, Beleuchtung, Geräusche oder Verhalten (Interaktion mit dem Benutzer) wichtig. Wie bei herkömmlichen Graphen werden diese Informationen in Form von Knoten (diese repräsentieren bei einem Szenegraphen Instanzen von Java3D-Klassen) und Kanten (modellieren die Beziehungen zwischen diesen Instanzen) repräsentiert.

8.3.2 Elemente des Szenegraphen

Abbildung 1 zeigt einen beispielhaften Szenegraphen, auf dessen Bestandteile wir nun näher eingehen wollen. Eine virtuelle, dreidimensionale Welt wird in Java3D als `VirtualUniverse` bezeichnet. Ein solches Objekt bildet die Wurzel eines jeden Szenegraphen. Daran wird nun mindestens ein sogenanntes `Locale`-Objekt angehängt, das unter anderem die Aufgabe hat, über dem virtuellen Universum ein Koordinatensystem aufzuspannen; das `Locale`-Objekt ist also gewissermaßen der Ursprung dieses Koordinatensystems. Weiterhin können an das `Locale`-Objekt beliebig viele Teilgraphen angehängt werden. Hiervon sind mindestens zwei nötig: der sogenannte Inhaltsgraph (*Content Branch Graph*), von dem es auch mehrere geben kann und der Sichtgraph (*View Branch Graph*), welcher die Sichtparameter des virtuellen Universums, wie z.B. den Standpunkt des Betrachters oder die Blickrichtung spezifiziert. Dies geschieht mit Hilfe einer sogenannten `ViewPlatform`, auf der sich der Betrachter befindet und auf der er sich im virtuellen Sinne bewegen kann. Soll nun z.B. die Blickrichtung geändert werden, so wird dies über die der `ViewPlatform` übergeordneten `TransformGroup` (s.u.) realisiert. Im `View`-Objekt werden alle nötigen Informationen zusammengefasst und die so gerenderte Szene über ein `Canvas3D`-Objekt auf dem Bildschirm ausgegeben.

Der Inhaltsgraph enthält die eigentlichen, 'greifbaren' Bestandteile des Universums und beschreibt deren Beschaffenheit bzw. Zusammenspiel. Diese Elemente wollen wir im folgenden vorstellen. Wir unterscheiden:

GroupNodes (innere Knoten) `GroupNodes` haben genau einen Elternknoten und beliebig viele Kinder, die entweder wieder `GroupNodes` oder Blätter (s.u.) sein können. Dabei

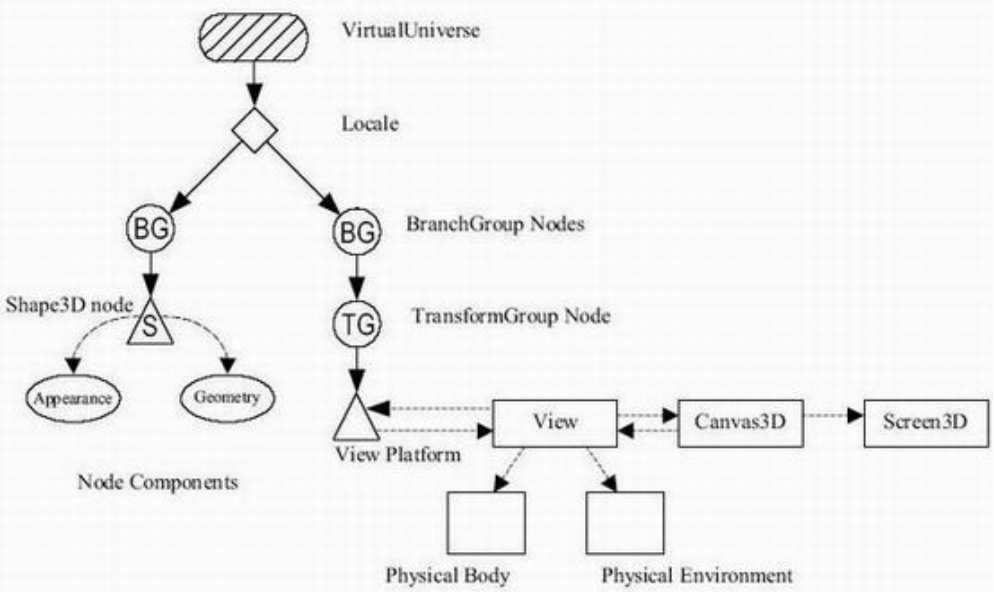


Abbildung 8.1.: Ein beispielhafter Szenegraph

ist `GroupNode` eine abstrakte Klasse, deren Implementierungen die inneren Knoten des Szenegraphen darstellen. Die zwei wichtigsten Implementierungen wollen wir uns näher ansehen, nämlich `BranchGroup` und `TransformGroup`. `BranchGroups` haben hauptsächlich organisatorische Funktion, sie stellen nämlich die Wurzel weiterer, eigenständiger Teilgraphen dar. Dazu muss für jeden Teilgraphen eine `BranchGroup` unter das entsprechende `Locale`-Objekt gehängt werden. Zusätzlich können innere Knoten eines Teilgraphen nur über seine zugeordnete `BranchGroup` entfernt werden. Schließlich können `BranchGroups` auch ineinander geschachtelt werden, um eine weitere Verzweigung von Teilgraphen zu erhalten. Über `TransformGroups` hingegen können affine Abbildungen wie Rotation, Skalierung oder Verschiebung realisiert werden. Hierzu legt man die gewünschte Operation (oder auch mehrere) fest, indem man die Attribute der betreffenden `TransformGroup` mittels eines `Transform3D`-Objektes entsprechend setzt. Diese Operation wird nun auf alle Kinder der `TransformGroup` (also auf diejenigen geometrischen Objekte, die an diese `TransformGroup` angehängt wurden) angewendet.

LeafNodes (Blätter) `LeafNodes` haben genau einen Elternknoten, aber keine Kinder. Alle gegenständlichen Objekte, wie geometrische Körper, aber auch Beleuchtung, Geräusche und interaktive Elemente einer Szene bilden die Blätter eines Szenegraphen. Man kann hier zwei Klassen von `LeafNodes` unterscheiden, nämlich sogenannte `ShapeNodes`, die Informationen über dreidimensionale Objekte beinhalten und `EnvironmentNodes`, welche Informationen über Beleuchtung, Hintergrund, Verhalten von Objekten,

etc. repräsentieren.

NodeComponents `NodeComponents` sind Knoten, die Eigenschaften von 3D-Objekten beschreiben. Sie werden von `Shape3D`-Objekten referenziert und sind grob gesehen keine Elemente des Szenegraphen, werden aber der Übersichtlichkeit halber oft dennoch in solchen dargestellt. Für eine geometrische Struktur müssen zweierlei Dinge bekannt sein. Zum einen ihre sogenannte `Geometry`, also beispielsweise die Anordnung ihrer Eckpunkte und die sogenannte `Appearance`, hierzu zählen z.B. die Farbe oder Transparenzeigenschaften. Diese Informationen werden in `NodeComponents` gespeichert.

8.3.3 Zusammenhang zwischen Szenegraph und Java3D-Anwendungen

Welcher Zusammenhang besteht nun zwischen Szenegraph und konkreten Java3D-Anwendungen oder anders gefragt: Wie lässt sich aus einem Szenegraph-Modell eine Java3D-Anwendung gewinnen? Wir wollen in einer Art „Kochrezept“ die hierfür nötigen Schritte angeben:

1. Erzeuge ein `Canvas3D`-Objekt
2. Erzeuge ein `VirtualUniverse`-Objekt
3. Erzeuge ein `Locale`-Objekt und weise es dem `VirtualUniverse`-Objekt zu
4. Konstruiere einen Sichtgraphen
   a) Erzeuge ein `View`-Objekt
   b) Erzeuge ein `ViewPlatform`-Objekt
   c) Weise das `Canvas3D`- und `ViewPlatform`-Objekt dem `View`-Objekt zu
5. Erzeuge mindestens einen Inhaltsgraphen
6. Kompiliere die Branch Graphen
7. Fge die Branch Graphen in das `Locale`-Objekt ein

Schritt 6 mag zunächst etwas exotisch erscheinen. Wir wollen uns an dieser Stelle jedoch mit der Erklärung begnügen, dass die Kompilierung im wesentlichen aus Performance-Gründen durchgeführt wird. Hier wird also der Szenegraph als solcher optimiert, um das anschließende Rendering zu beschleunigen. Wir können uns nun das Leben erleichtern, indem wir statt eines `VirtualUniverse`-Objektes ein sogenanntes `SimpleUniverse` verwenden. Dies befreit uns von der Aufgabe, den Sichtgraphen spezifizieren zu müssen. Von dieser Möglichkeit werden wir bei einigen später im Text folgenden Beispielen (Abs. 4.4) noch Gebrauch machen.

8.4 Geometrische Strukturen

In diesem Kapitel werden einige Möglichkeiten vorgestellt, visuelle Objekte bzw. Inhalte zu erstellen. Dabei werden wir zunächst auf Shape3D-Objekte eingehen, die für das Erschaffen visueller Inhalte essentiell sind. Primitive Objekte sind eine weitere Möglichkeit, geometrische, visuelle Objekte zu erstellen. Die vorgestellten Konzepte sollen abschließend durch einige Anwendungsbeispiele veranschaulicht werden.

8.4.1 Shape3D

`Shape3D`-Objekte sind `LeafNodes`, die jeweils zwei `NodeComponents` referenzieren. Dabei legen sie zum einen die geometrische Struktur eines sichtbaren Objektes (`Geometry`) fest, zum anderen beinhalten sie Informationen darüber, wie dieses Objekt gerendert werden soll (`Appearance`). Diese Rendering-Attribute werden in Kapitel 4.3 kurz erläutert. Zusätzlich enthält ein `Shape3D`-Objekt ein `Bounds`-Objekt, das für Kollisionserkennung mit anderen Objekten im virtuellen Universum oder für das Picking eine Rolle spielt. `Shape3D`-Objekte sind die gebräuchlichste Form, um geometrische, sichtbare Objekte zu erstellen. Die Geometrie-Informationen eines solchen Objekts können dabei über Primitive Körper (Abs. 4.2) oder über ein sogenanntes `GeometryArray` definiert werden. Letzteres ermöglicht es, für jeden Punkt des Objektes eigene Koordinaten, Farben, etc. zu verwalten.

8.4.2 primitive Körper

Die von Java3D zur Verfügung gestellten primitiven Körper sind eine weitere Möglichkeit, in einem virtuellen Universum sichtbaren Inhalt zu erstellen. Primitive Objekte zeichnen sich dadurch aus, dass ihre geometrische Struktur (`Geometry`) vorgegeben ist, nicht aber die übrigen Rendering-Attribute (`Appearance`). Java3D stellt folgende primitive Grundkörper zur Verfügung:

- `Box` (Würfel)

  Definiert ber Höhe, Breite und Tiefe
- `Sphere` (Kugel)

  Definiert über den Radius
- `Cylinder`

  Definiert über Radius und Höhe
- `Cone` (Kegel)

  Definiert über Radius und Höhe

8.4.3 Appearance

Das `Appearance`-Objekt eines `Shape3D`-Elementes enthält wie bereits erwähnt die Rendering-Information für das entsprechende Objekt. Diese können vor oder während der Szenegraph-Erstellung gesetzt oder auch dynamisch zur Laufzeit manipuliert werden. Mögliche Attribute beziehen sich z.B. auf die Farbe, Transparenz oder das Material des betroffenen Objektes. Um die Rendering-Attribute eines Objektes zu setzen, erstellt man ein `Appearance`-Objekt, setzt dessen Attribute entsprechend (dies kann wiederum die Erzeugung neuer Objekte erfordern) und fügt es abschließend zum zugehörigen `Shape3D`-Objekt hinzu.

8.4.4 Beispiele

Wir wollen als erstes einen sogenannten `ColorCube` erschaffen und ausgeben (Beispiel 1). Die Klasse `ColorCube` erbt von der Klasse `Shape3D`, daher sind Instanzen von `ColorCube` also auch `Shape3D`-Objekte. Wie in Kapitel 3.3 bereits angekündigt, werden wir hierzu ein `SimpleUniverse` verwenden, um uns die Arbeit ein wenig zu erleichtern. Es mag Sinn machen, sich das vorgestellte „Kochrezept“ für Java3D-Anwendungen noch einmal vor Augen zu führen (Abs. 3.3).

```
public class Wuerfel extends Applet {
    public Wuerfel() {
        setLayout(new BorderLayout());
        GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();
        Canvas3D canvas3D = new Canvas3D(config);
        add("Center", canvas3D);
        SimpleUniverse su = new SimpleUniverse(canvas3D);
        su.getViewingPlatform().setNominalViewingTransform();

        BranchGroup bg = new BranchGroup();
        ColorCube cc = new ColorCube(0.4);
        bg.addChild(cc);
        bg.compile();
        su.addBranchGraph(bg);
    }
}
```

Zunächst wird ein `Canvas3D`-Objekt namens `canvas3D` erzeugt (siehe Abs. 3.3 Schritt 1) und dem `SimpleUniverse`-Objekt `su` (Abs. 3.3 Schritt 2) zugeordnet. Da wir ein `SimpleUniverse`-Objekt verwenden, entfallen wie schon erwähnt die Schritte 3 und 4 aus dem in

Kapitel 3.3 vorgestellten Kochrezept. Wir erzeugen nun eine neue `BranchGroup bg`, um einen Inhaltsgraphen anlegen zu können (Abs. 3.3 Schritt 5). Als nächstes erzeugen wir einen `ColorCube cc`, den wir dem Inhaltsgraphen hinzufügen. Schließlich werden die Branch-Graphen kompiliert (Abs. 3.3 Schritt 6) und in das `Locale`-Objekt des virtuellen Universums eingehängt (Abs. 3.3 Schritt 7).

Wir wollen nun auf dem so erzeugten Würfel noch eine Rotation entlang seiner x-Achse durchführen (Beispiel 2). Dies können wir recht schnell bewerkstelligen, indem wir zu obigem Beispiel die folgenden Zeilen hinzufügen:

```
Transform3D rotate = new Transform3D();
rotate.rotX(Math.PI/4.0d);
TransformGroup wuerfelRotate = new TransformGroup(rotate);
wuerfelRotate.addChild(cc);
bg.addChild(wuerfelRotate);
```

Abbildung 2 zeigt den mit diesem Beispiel korrespondierenden Szenegraphen. Über das `Transform3D`-Objekt legen wir also die gewünschte Operation (`rotX`) fest und initialisieren mit ihr eine `TransformGroup` namens `wuerfelRotate`. Anschließend wird das zu rotierende Objekt (`cc`) unter `wuerfelRotate` gehängt und diese selbst schließlich unter die zu diesem Inhaltsgraphen gehörige `BranchGroup` (`bg`).

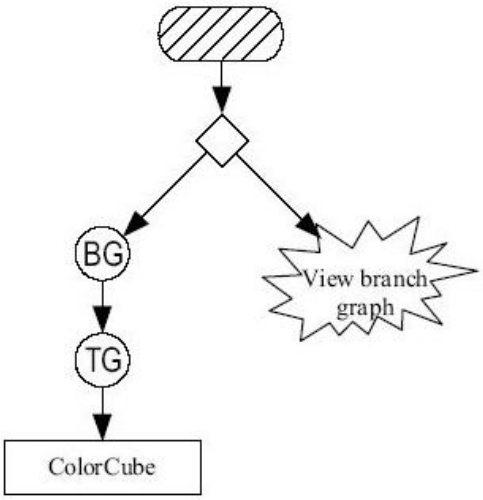


Abbildung 8.2.: aus Beispiel 2 resultierender Szenegraph

8.5 Interaktion und Animation

Gegenstand dieses Kapitels sind Animationen und interaktive, den Benutzer einbeziehende Techniken. Wir stellen zunächst die benötigten Werkzeuge und Klassen vor, um abschließend das Gesamtkonzept an einem kleinen Beispiel zu erläutern.

8.5.1 Behavior-Klassen

Behavior-Klassen dienen in Java3D dazu, Animationen von Objekten bzw. die Interaktion mit selbigen zu definieren. Ein solches Behavior kann dann im Prinzip auf sämtliche Attribute eines sichtbaren Objektes zugreifen und diese manipulieren.

Um nicht unnötig Rechenzeit zu verschwenden, kann der Entwickler eine sogenannte *Scheduling Region* festlegen, in der die spezifizierte Animation oder Interaktion stattfinden soll. Diese wird dann erst ausgelöst, wenn die scheduling-Grenze von der `ViewPlatform` aus überschritten wird. Wenn also der Benutzer die Animation nicht sehen kann, wird sie auch nicht ausgelöst.

Zu den vordefinierten Behaviors in Java3D gehören die sogenannten Interpolator-Klassen. Diese basieren auf einer Zeitfunktion und sind in der Lage, Parameter von Szenegraph-Objekten zu manipulieren. Mit solchen Interpolator-Klassen ist es dann möglich, Position (`PositionInterpolator`), Orientierung (`RotationInterpolator`), Farbe (`ColorInterpolator`), Größe (`ScaleInterpolator`) oder Transparenz (`TransparencyInterpolator`) zu verändern. Interpolator-Objekte arbeiten mit sogenannten `Alpha`-Objekten zusammen. Diese realisieren eine zeitlich veränderliche Funktion, deren Wert von der verstrichenen Zeit und eventuell übergebenen Parametern abhängt. Das wohl einfachste `Alpha`-Objekt realisiert eine Endlos-Schleife.

8.5.2 Beispiel

Wir wollen uns noch einmal an den in Kapitel 4.4 entwickelten `ColorCube` erinnern und diesen jetzt kontinuierlich rotieren lassen. Dazu erstellen wir zuerst eine `TransformGroup`, für die wir schreibenden Zugriff erlauben. Als nächstes erzeugen wir ein `Alpha`-Objekt mit den Parametern -1 und 4000. Dies bedeutet, dass alle 4 Sekunden eine Rotation durchgeführt werden soll und dies durchgängig (-1). Nun benötigen wir einen `RotationInterpolator`, da wir den Würfel ja kontinuierlich rotieren lassen wollen. Der Interpolator erhält Referenzen auf die `TransformGroup`, auf der er arbeiten soll, sowie auf das `Alpha`-Objekt. Schließlich müssen wir noch die Scheduling Region festlegen. Dies tun wir mithilfe einer `BoundingSphere`; dadurch sind die betroffenen Objekte also von einer kugelförmigen *Scheduling Region* umgeben. Abbildung 3 zeigt den resultierenden Szenegraphen. Der zu ergänzende Code sieht folgendermaßen aus:

```
TransformGroup objSpin = new TransformGroup();
objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
bg.addChild(objSpin);
objSpin.addChild(cc);

Alpha rotationAlpha = new Alpha(-1, 4000);
RotationInterpolator rotator =
   new RotationInterpolator(rotationAlpha, objSpin);

BoundingSphere bounds = new BoundingSphere();
rotator.setSchedulingBounds(bounds);
objSpin.addChild(rotator);
```

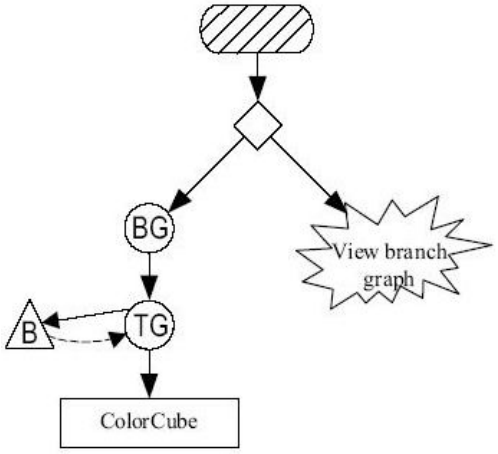


Abbildung 8.3.: aus Beispiel 3 resultierender Szenegraph

8.6 Fazit

Den Themenkomplex Java3D unter dem vorgegeben Rahmen umfassend zu behandeln, war natürlich von vornherein unmöglich. So wurden zum Beispiel manche Aspekte überhaupt nicht erörtert (Beleuchtung, Loader, etc.), andere hingegen nur oberflächlich behandelt (Interaktion und Animation). Es lässt sich aber festhalten, dass mit der Java3D-API ein durchaus mächtiges Werkzeug für die dreidimensionale Grafikentwicklung zur Verfügung steht. Einer der größten Vorteile von Java3D dürfte dabei in seiner Einfachheit liegen; so man denn die

zugrunde liegenden Konzepte einmal verinnerlicht hat. Andererseits kann die mangelnde Erfahrung mit Java3D durchaus zu Frustrationen und Mehrarbeit führen. Abzuwarten bleibt, wie sich Java3D hinsichtlich einiger Forderungen im Bereich der Performance bewähren wird.

KAPITEL 9

Eclipse - Aufbau und Anwendung

Andrey Lysenko

9.1 Geschichte

Eclipse ist ein Open-Source-Projekt unter der Leitung der Firma IBM, die dafür ca. 40 Mio. Dollar ausgegeben hat. Entwickelt wird es aber von dem Konsortium ECLIPSE.ORG. Dieses Konsortium besteht aus der IBM-Tochterfirma OBJECT TECHNOLOGIES INTERNATIONAL (OTI), die eine entscheidende Rolle in der Entwicklung von Eclipse gespielt hat, und vielen anderen Firmen: BORLAND, MERANT, QNX SOFTWARE SYSTEMS, RATIONAL SOFTWARE3, RED HAT, SUSE, TOGETHERSOFT3 und WEBGAIN2. Heute hat das Konsortium mehr als 45 Mitglieder, unter anderem auch SYBASE, HITACHI, ORACLE, HEWLETT-PACKARD usw.

OTI (1996 gekauft von IBM) hat viel früher ein Java-Tool "Visual Age for Java" (VA4J) in Smalltalk geschrieben. Eclipse ist also ein mehr oder weniger VA4J in Java übernommen und ergänzt und hat daher schon eine solide Geschichte.

9.2 Grundlagen

9.2.1 Einführung

Eclipse ist ein Rahmenwerk zur Integration verschiedenster Anwendungen. Eine solche Anwendung ist z.B. die mitgelieferte Java Entwicklungsumgebung JDT (Java Development Toolkit). Diese Anwendungen werden in Form sogenannter Plug-Ins (Abbildung 9.1) zur Verfügung gestellt und von der Eclipse-Plattform automatisch erkannt und integriert.

Eclipse bietet weiterhin Funktionalität zum Verwalten von Ressourcen (normalerweise Dateien) an. Diese befinden sich im sogenannten **Workspace**, einem speziellen Verzeichnis im

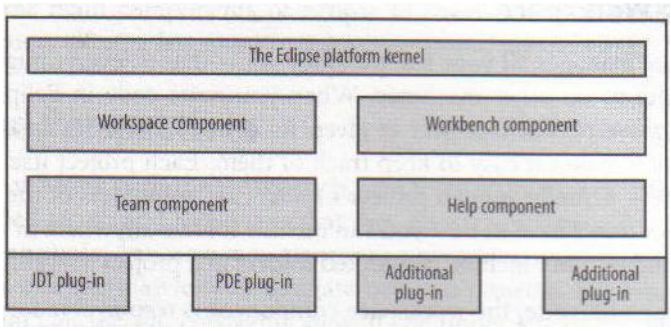


Abbildung 9.1.: Eclipse - Architektur

Dateisystem. Verändert eine Anwendung eine solche Ressource aus Eclipse heraus, werden auch die anderen Anwendungen - über entsprechende Benachrichtigungsmechanismen - davon erfahren.

Der Benutzer arbeitet dabei immer in der **Workbench**, dem grafisch sichtbaren Teil der Eclipse-Plattform. Das konkrete Aussehen der Workbench wird von der gerade ausgewählten **Perspektive** bestimmt. Diese fasst jeweils eine Menge von sogenannten Views und Editoren zusammen und stellt spezielle Befehle für die Menü- und Werkzeugleisten zur Verfügung. Z.B. können in der Debug-Perspektive beliebige Programme getestet und geprüft werden, indem man zu jedem Zeitpunkt der Programmausführung eine beliebige Variable oder sogar Ausdrücke wie z.B. $x+y/5$ abfragen kann. Solche Abfragung kann man sowohl schrittweise, als auch durch das Setzen sogenannter Breakpoints an verschiedenen Stellen eines Programms durchführen.

Ein **View** bietet meist eine Sicht auf die vorhandenen Ressourcen. Je nach View werden nur Teile oder auch innere Zusammenhänge der Ressourcen angezeigt.

Ein **Editor** dient zum Bearbeiten einer Ressource. Hierbei wird ein strikter Laden-Verändern-Speichern Lebenszyklus eingehalten. Erst wenn eine im Editor veränderte Ressource auch gespeichert wird, konnen alle oben erwähnten Benachrichtigungsmechanismen greifen. Spezielle Views können auch direkt mit einem Editor (und nicht mit der eigentlichen Ressource) gekoppelt werden. Zum Beispiel ist der Outline View der Java-Perspektive direkt an den Java-Quelltexteditor gekoppelt.

Eine Besonderheit bei Eclipse ist nun die äußerste Flexibilität, mit der vorhandene Views und Editoren kombiniert werden können. Nicht nur die Anordnung in der Workbench ist frei wählbar. Man kann in eine geöffnete Perspektive jeden anderen View und jeden Editor hinzufügen - auch wenn diese in einem ganz anderen Plug-In definiert wurden. Dies ermöglicht es dem Benutzer, eine auf ihn zugeschnittene Entwicklungsumgebung zusammenzustellen.

9.2.2 Plug-Ins

Um neue Anwendungen zu installieren, werden entsprechende Plug-Ins einfach in das Verzeichnis $ECLIPSE_HOME/plugins kopiert. Die Plug-Ins können allerdings erst nach einem Neustart von Eclipse verwendet werden. Je nach installiertem Plug-In sind jetzt neue Perspektiven auswählbar, neue Einträge in Menü- und Werkzeugleiste vorhanden oder in untergeordneten Strukturen neue Funktionalitäten eingetragen.

9.2.3 Workspace

Die Grundeinstellung sieht als Wurzelverzeichnis für den Workspace das Verzeichnis $ECLIPSE_HOME/workspace vor. Gerade wenn mehrere Benutzer auf die Installation zugreifen wollen oder sich einen Rechner teilen, ist eine Trennung der Arbeitsbereiche sinnvoll. Man kann beim Starten von Eclipse ein beliebiges Verzeichnis zum Workspace-Wurzelverzeichnis machen.

9.3 Aufbau

9.3.1 Einführung

Die wichtigsten Perspektiven von Eclipse, mit denen ein Benutzer sich zuerst auseinandersetzen muss, wenn er mit Eclipse programmieren möchte, sind die Java- und Debug-Perspektiven. Diese Perspektiven enthalten folgende Views:

- *Java-Perspektive:* Package Explorer, Hierarchy, Navigator, Outline, Problems, Console,
- *Debug-Perspektive:* Debug, Variables, Breakpoints, Expressions, Console, Outline,

wobei einige in beiden Perspektiven enthalten sind. Alle Views kann man natürlich jeder Perspektive hinzufügen oder aus ihr entfernen und danach auch abspeichern. Hier werden die wichtigsten Views näher beschrieben.

9.3.2 Views

Package Explorer

Im Package Explorer sind alle Projekte mit darin enthaltenen Pakete, Klassen und Methoden zu sehen. Pakete sind keine realen Ressourcen, sondern virtuelle Objekte. Die Paket-Struktur

einer Projekts ergibt sich aus den Paket-Deklarationen am Beginn jeder Java-Quelldatei. Die Java-Spezifikation verlangt allerdings, dass sich die Paket-Struktur isomorph (s. Abbildung 9.2) auf eine Verzeichnisstruktur abbilden lässt.

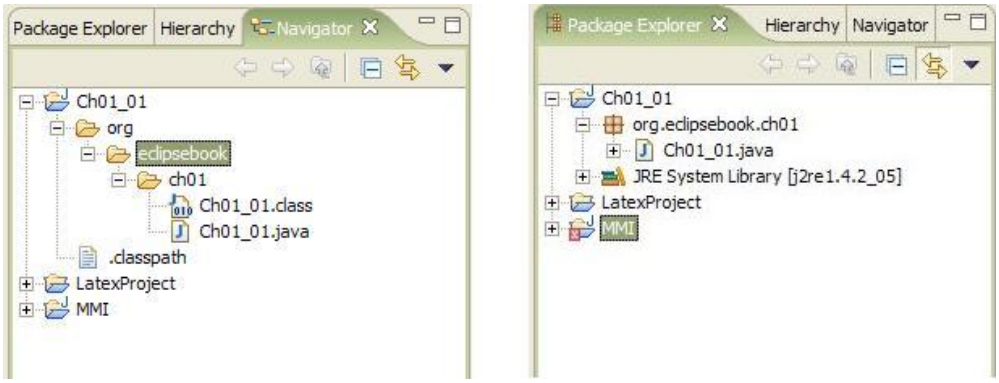


Abbildung 9.2.: Navigator und Package Explorer

Navigator

Der Navigator (Abbildung 9.2) zeigt Projekte, Verzeichnisse und Dateien und vermittelt somit einen Überblick über die von der Eclipse-Workbench verwalteten Ressourcen und gestattet die Navigation in der Menge der Ressourcen.

Hierarchie

Die Hierarchie zeigt die Super- und Subtypen für Klassen und Interfaces an. Dabei besteht die Möglichkeit, entweder die Sicht nur auf Super- oder Subtypen zu beschränken oder die komplette Hierarchie anzuzeigen. Mit der History-Funktion kann man rasch zwischen den verschiedenen Sichten wechseln oder vorher angezeigte Hierarchien noch einmal anzeigen. Der Hierarchie-Browser besteht aus zwei Fenstern. Im oberen Fenster wird die Typenhierarchie angezeigt, im unteren Fenster werden die Felder und Methoden des selektierten Typs angezeigt.

Outline

Der Outline-View gestattet die Navigation innerhalb einer Quelldatei. Generell ist der Outline nicht auf Java-Programme beschränkt, sondern steht - je nach installierten Plug-ins - auch für andere Dateitypen zur Verfügung.

Bei Java-Programmen stellt der Outline-View sowohl die Felder und Methoden als auch die import-Anweisungen dar. Sind innere Klassen definiert, werden auch diese Klassen dargestellt: die Hauptklasse und die inneren Klassen bilden eine Baumstruktur.

Durch einen einfachen Klick auf ein im Outline dargestelltes Feld oder eine Methode wird der Editor auf die Definition des Feldes bzw. der Methode positioniert. Die verschiedenen Buttons der Outline-Toolbar erlauben es, bestimmte Elemente aus dem Outline-View herauszufiltern. Mit dem Sortieren-Button z.B. können Felder und Methoden alphabetisch geordnet werden (andernfalls gilt die Reihenfolge in der Quelldatei).

Problems und Console

Im View Problems erscheinen alle Probleme, die beim Kompilieren entstehen und in Console wird der in die Standardausgabe (d.h. System.out.print()) umgeleitete Text gedruckt.

Debug-, Breakpoint-, Variables-, und Expression-View

Diese Views sind ein Bestandteil der Debug-Perspective. Diese Perspektive stellt in Eclipse umfangreiche Debugging-Funktionalität zur Verfügung und ermöglicht eine gute übersicht üben den aktuellen Zustand der virtuellen Maschine. Durch die Evaluierung von Einzelstatements oder eine überprüfung von Attributen lassen sich viele weitere Informationen über die VM erhalten. Eclipse bietet, wie fast jede andere IDE, natürlich auch eine Möglichkeit zum Remote Debugging, die besonders bei Server-Anwendungen hilfreich sein kann.

Im Debug-View sieht man beim Debugging die aktuell ausführende Zeile. Gleichzeitig sieht man im Breakpoints-View alle Breakpointes und im Variables-View die aktuelle Variablenbelegung. Außerdem lässt sich ein beliebiger Ausdruck im Expressions-View beobachten.

Java Browsing Perspective

Eine weitere sehr nützliche Perspektive ist Java Browsing Perspective, über die man recht schnell an beliebige Stellen im Code gelangen kann. Wie

man in Abbildung 9.3 sehen kann, kann man zwischen Projekten, Paketen, Klassen und Elementen innerhalb der ausgewählten Klasse umschalten, wobei im Editor im unteren Bereich sofort auf die entsprechende Stelle gesprungen wird.

9.3.3 Refresh

Gelegentlich kommt es vor, dass der Zustand der Dateien im Workspace nicht dem Kenntnisstand von Eclipse entspricht, denn Eclipse speichert Metadaten über die Ressourcen der Workbench in einem versteckten Verzeichnis ab. Wurde jedoch eine Datei der Workbench an Eclipse vorbei verändert, z.B. durch die Anwendung eines externen Werkzeugs, so kann es passieren, dass die Metadaten nicht mehr mit dem aktuellen Zustand einer Ressource übereinstimmen. Das ist nicht weiter schlimm. In diesem Fall führt man eine Synchronisierung auf den betroffenen Ressourcen durch. Man selektiert die betroffene Ressource im Navigator und wendet die Kontextfunktion Refresh an. Diese Funktion lässt sich nicht nur auf einzelne Dateien, sondern auch auf ganze Verzeichnisse oder Projekte anwenden.

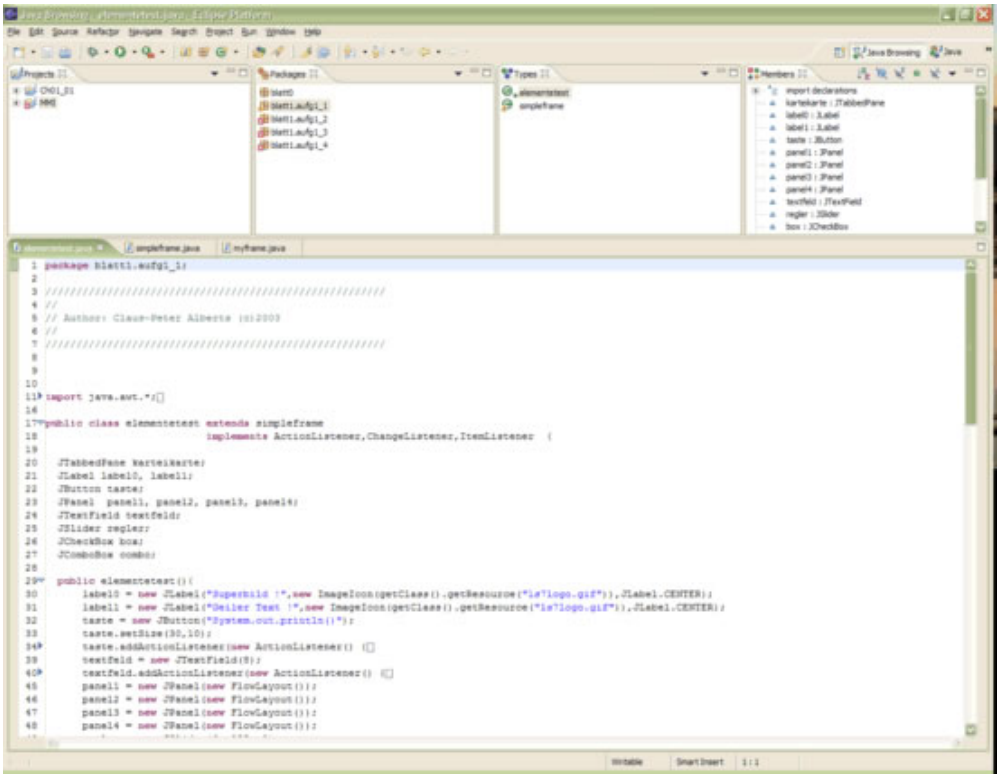


Abbildung 9.3.: Java Browsing Perspective

9.3.4 Fehleranzeige

Der gerade editierte Quelltext wird in Eclipse während des Speichervorgangs übersetzt und auftretende Fehler werden im Quelltext sofort markiert. Sie können auf diese Weise schnell lokalisiert und verbessert werden. Sämtliche Fehler werden zusätzlich in einer ürsichtstabelle dargestellt und es kann schnell an die jeweilige Stelle gesprungen werden. Für jeden markierten Fehler können Tool Tips mit der Beschreibung des Fehlers angezeigt werden.

9.4 Fazit

Eclipse ist eine sehr leistungsfähige IDE. Immer mehr Firmen verwenden es als Basis für ihre eigenen Produkte. Es hat sich mittlerweile einen festen Platz in der Riege der etablierten Entwicklungsumgebungen sichern können. Durch den modularen Aufbau und die freie

Lizenz hat es den großen Vorteil, dass es als universelles Tool einsetzbar ist. Dem Eclipse-Entwicklungsteam ist es besonders gut gelungen, eine hervorragende Plattform für verschiedenste Erweiterungen zu kreieren.

KAPITEL 10

Das Eclipse Plug-In Modell

Boris Düdder

10.1 Einleitung

Diese im Rahmen eines Projektgruppenseminars erstellte Ausarbeitung beschäftigt sich mit Eclipse. Eclipse ist ein, von der IBM Corp. im Stiftungsmantel unterstütztes Softwareentwicklungssystem, welches als eine offene, erweiterbare Architektur implementiert wurde (Eclipse Foundation, 2003). Plug-Ins sind die Umsetzung der Forderung der Erweiterbarkeit. Plug-Ins sind Erweiterungsmodule für Eclipse, die zusätzliche Funktionalität zu Verfügung stellen. Wozu benötigt man bei einer so ausgereiften Softwareentwicklungsumgebung wie Eclipse berhaupt Plug-Ins? - Die Frage ist einfach zu beantworten. - Eclipse ist bis auf einen kleinen Kernel eine Sammlung von Plug-Ins. Diese Plug-Ins dienen der Erweiterung und Ergänzung der Funktionalität von Eclipse. Dabei ist die Erweiterbarkeit und Komponentenorierentierung eines der Basiselemente von Eclipse.

Die Ausarbeitung wird in Kapitel 10.2 einen kurzen Überblick über die Benutzeroberfläche (Workbench) und deren Elemente verschaffen. Woran sich in Kapitel 10.3 die Betrachtung des Plug-In Modells von Eclipse anschließt, wobei die Bereiche der Erstellung und Deklaration von Plug-Ins in Kapitel 10.4 sowie deren Initialisierung in Eclipse vorgestellt werden und die Betrachtung der Installation von Plug-Ins in Kapitel 10.5 diese Ausarbeitung schließt.

10.2 Die Workbench

Die `Workbench` (siehe Abbildung 10.1 auf der nächsten Seite) stellt den lauffähigen Applikationsrahmen dar und dient der Interaktion zwischen dem Benutzer und Eclipse. Sie verwaltet sowohl die `Ressourcen` als auch die Steuerelemente, die die Funktionalität der Plug-Ins steuern. Jede Workbench in Eclipse besteht zusätzlich aus den Komponenten `Perspektiven`, `Views` und `Editoren`, die im Folgenden vorgestellt werden.

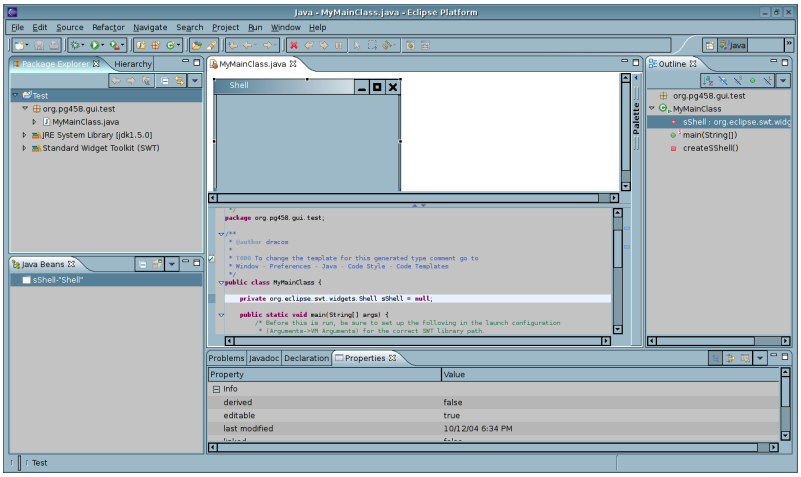


Abbildung 10.1.: Die Eclipse Workbench mit Java-Editor und Views

10.2.1 Ressourcen

Die Workbench verwaltet drei Arten von Ressourcen:

1. Dateien,
2. Verzeichnisse und
3. Projekte

Dateien und Verzeichnisse korrespondieren mit den physikalischen oder logischen Verzeichnissen und Dateien des Dateisystems. Projekte selbst sind ausschließlich die Wurzel eines Projektbaums und können nur Verzeichnisse oder Dateien beinhalten. Einem Projekt wird, ähnlich wie bei Verzeichnissen, beim Anlegen ein Verzeichnis des Dateisystems zugeordnet.

10.2.2 Editoren

Ein Editor ist der Teil einer Workbench, der die Möglichkeit bietet, Dateien eines bestimmten Dateityps (Java, Wars, ...) zu bearbeiten. Sollte hingegen einem Dateityp kein Editor zugeordnet werden, so wird versucht, einen externen Editor zu starten.

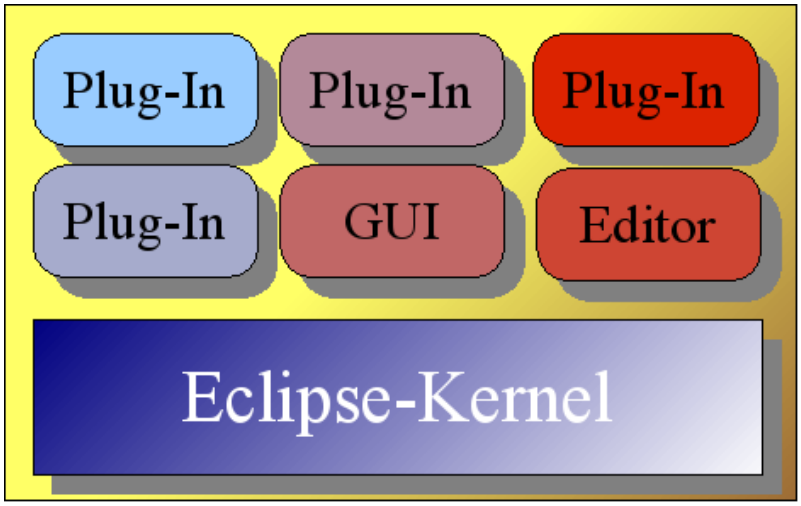


Abbildung 10.2.: Eclipse Plug-In Architektur

10.2.3 Views

Ein View unterstützt Editoren indem er alternative Darstellungen anbietet, oder bei der Navigation und der Verwaltung von Workbenchinformationen hilft. Beispielsweise bietet die Navigator-View Informationen ber die Ressourcen der Projekte, während die Properties-View editierbare Objekteigenschaften, z.B. von aktuell editierten GUI-Elementen, darbietet.

10.2.4 Perspektiven

Eine Perspektive definiert eine Anordnung der Editoren und Views in der Workbench, sowie die zur Verfügung stehenden Werkzeugleisten und Menüs. Mit der Perspektive wird eine von der Aufgabe abhängige, optimale Anordnung von Editoren und Views bereitgestellt. Die Workbench besitzt mehrere Perspektiven, u.a. für Debugging, Editieren von Sourcecode sowie Teamarbeit.

10.3 Das Plug-In Modell

Wie in der Einfhrung erwähnt, ist Eclipse als eine offene, erweiterbare Plattform entwickelt worden. Dies ist realisiert worden, in dem man einen kleinen Plattformkern geschaffen hat, der dazu dient, Plug-Ins zu starten. Dieser Vorgang ähnnelt dem Bootstrapping in Betriebssystemen. Die gesamte Funktionalität von Eclipse basiert vollständig auf den Plug-Ins (siehe

Abbildung 10.2 auf der vorherigen Seite). Somit ist die Workbench nur ein Plug-In der Eclipse Plattform. Bestandteile einer Eclipse Standardinstallation sind neben dem Plattformkern Plug-Ins für Ressourcenverwaltung, der graphischen Benutzeroberfläche, dem Hilfesystem sowie der Teamarbeit mittels CVS. Als optionale Plug-Ins existieren unter anderem die Java Entwicklungsumgebung (JDT), sowie die Plug-In Entwicklungsumgebung (PDE). Jede Eclipse Installation besitzt ein `plugins`-Verzeichnis, in dem sich die installierten Plug-Ins befinden. Die einzelnen Plug-Ins besitzen dort ein Unterverzeichnis, in dem sich die Ressourcen und die das Plug-In beschreibende `Manifest`datei `plugin.xml` befinden.

10.3.1 Initialisierungsvorgang

Wird Eclipse gestartet, werden vom Plattformkern die Informationen aus den im Plug-Ins Verzeichnis befindlichen Manifestdateien ausgelesen und anschließend in einem Teil des Plattformkerns, der `Plug-In-Registry`, registriert. Die Plug-Ins werden erst bei Bedarf aus diesem Repository geladen. Diese Methode des Ladens bezeichnet man als Lazy Loading, was eine Verbesserung der Ladezeit der Entwicklungsumgebung gewährleistet, aber jedoch auch einen gravierender Nachteil besitzt. Da die Manifestdateien ausschließlich während des Startvorgangs gelesen werden, können die Plug-Ins nicht während der Laufzeit installiert werden. Daher ist ein Plug-In Entwickler gezwungen, immer eine zweite Eclipseinstanz zu starten, in der das Plug-In getestet werden kann.
Eine grundlegende Philosophie von Eclipse ist, dass Plug-Ins nicht isoliert existieren. Sie bauen aufeinander auf und ergänzen ihre Fähigkeiten. Dies bedingt die Kommunikation der Plug-Ins untereinander. Dabei muss die Kommunikation folgendes beinhalten,

- um welche Funktionalität sie die Plattform erweitern
- um welche Funktionalität andere das Plug-In erweiter können.

Um diese Anforderungen zu erfüllen, ist das Kernkonzept der Plug-In Architektur, die `Extension Points`, entstanden. Zur Zusammenarbeit ist es notwendig, die Informationen über diese Extension Points zu kommunizieren.
Ein Extension Point definiert für ein Plug-In, um welche Funktionalität das Plug-In von anderen Plug-Ins erweitert werden kann. Da Plug-Ins Extension Points verwenden und selbst welche publizieren, entsteht ein komplexes Netz von Abhängigkeiten untereinander. Ein Plug-In A kann mit einem anderen Plug-In B in den folgenden Relationen stehen:

- Abhängigkeit: A ist das vorausgesetzte Plug-In und B das Abhängige. A liefert die Funktionalität, die B benötigt, um korrekt arbeiten zu können.
- Erweiterung: A ist das *Basis-Plug-In* und B das erweiternde Plug-In. B erweitert die Funktionalität von A.

10.3.2 Features

Ein Feature stellt eine logische Gruppierung von Plug-Ins dar. Diese Gruppierung ist meistens Aufgaben- oder Themengebunden, wie z.B. ein Feature für die Java-Entwicklung. Der Name des Features korrespondiert mit dem Namen eines Plug-Ins, dem sogenannten Feature-Plug-In. Zusätzlich bietet das Feature-Konzept für die Plug-Ins noch einige wichtige Bonbons. Ein Feature kann eine Begrüssungsseite, eine anfängliche Perspektive sowie Cheat-Sheets, d.h. eine Tipps&Tricks-Seite für das Feature, welches dem Endbenutzer den Einstieg in jenes erleichtert soll, besitzen.

10.4 Plug-In Manifest

Das Plug-In Manifest ist ein XML-Dokument, welches die Informationen ber den strukturellen Aufbau des Plug-Ins fr die Plattform beinhaltet. Einer der notwendigen Bestandteile eines Plug-Ins ist die Manifestdatei `plugin.xml` im Verzeichnis des Plug-Ins. Üblicherweise beginnt die Entwicklung eines Plug-Ins mit der Erstellung dieses XML-Dokuments. Diese Datei beinhaltet neben einer allgemeinen Beschreibung des Plug-Ins auch die zuvor erwähnten Relationen zu anderen Plug-Ins. Mindestens muss die Manifestdatei die folgenden Einträge besitzen:

- einen Namen fr das Plug-In,
- einen eindeutigen Identifikator (ID),
- die Versionsnummer des Plug-Ins.

Damit die ID plattformweit eindeutig ist, wird oft der komplette Paketname verwendet. Im Folgenden werden wir uns mit der Deklaration und den Einträgen in der XML-Datei beschäftigen. Der folgende Programmcode (Listing 10.1) ist exemplarisch und deklariert ein minimales Plug-In. In den Zeilen 3-5 werden die o.g. Informationen bereitgestellt.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <plugin
3      name="mein minimales plugin"
4      id="mein_minimales_plugin.Plug"
5      version="1.0.0"
6      provider-name="PG458.edu">
7      <runtime>
8          <library name="mein_minimales_plugin.jar"/>
9      </runtime>
10     <requires>
11         <import plugin="org.eclipse.ui"/>
12     </requires>
13 </plugin>
```

Listing 10.1: Minimales Plug-In

10.4.1 Deklaration von Erweiterungspunkten

Ein sogenanntes Basis-Plug-In stellt der Plattform neue Extension Points zur Verfügung und wird von anderen Plug-Ins erweitert. Um dieses Plug-In erweitern zu können, muss das Basis-Plug-In seine Extension Points publizieren. Die Extension Points werden in der Manifestdatei mithilfe des XML-Elements „extension-point“ deklariert. Der Extension Point `org.eclipse.ui actionSets` ermöglicht dem Endanwender in seiner GUI, z.B. durch Menüeinträgen oder Werkzeugleisteknöpfen, die Verwendung der Funktionalität des Plug-Ins. Der folgende Ausschnitt aus der plugin.xml (Listing 10.2) beschreibt die Deklaration des Extension Points „`actionSet`“.

```
1  ...
2  <extension point="org.eclipse.ui.actionSets">
3      <actionSet
4          id="org.eclipse.contribution.hello.actionSet"
5          label="Hello Action Set">
6          <action
7              id="org.eclipse.contribution.hello.HelloAction"
8              label="Hallo">
9          </action>
10     </actionSet>
11 </extension>
12 ...
```

Listing 10.2: Plug-In actionSet

Die ID muss auch in diesem Fall eindeutig sein. Um diese Eigenschaft auch global zu gewährleisten, wird der Extension Point ID (in Zeile 2 in Listing 10.2) die eindeutige Plug-In ID (in Zeile 4 in Listing 10.2) vorangestellt, sodass der Extension Point von anderen Plug-Ins über `org.eclipse.ui. actionSet` aufgerufen werden kann.

10.4.2 Beschreibung der Extension Points

Das Extension Point Schema ist ein XML-Dokument, welches für einen Extension Point definiert, welche Informationen das Plug-In in welcher Weise zur Verfügung stellen muss, damit die erweiternden Funktionen vom Basis-Plug-In angesprochen und verwendet werden können. Jedem Extension Point wird eindeutig ein Schema zu geordnet. Die Plug-Ins, welche die Funktionalität des Plug-Ins verwenden wollen, müssen wissen, wie die Erweiterung exakt aussieht oder exakter, welche Informationen benötigt werden, um die Klasse dieser Erweiterung zu instanziieren. Beispielsweise verwendet das Listing 10.2 implizit das Schema des Extension Points `org.eclipse.ui.actionSet`, um die Verbindung zu diesem Extension Point zu ermöglichen. Die Aufgabe des Extension Point Schemas ist es, den Extension Point detaillierter zu beschreiben. Ein Extension Point Schema ist ein XML-Dokument und muss explizit deklariert werden, falls das Schema:

- einen anderen Namen als die ID des Extension Points besitzt, oder

- in einem Unterverzeichnis des Plug-In-Hauptverzeichnis liegt

10.4.3 Fragment

Ein Fragment ist ein spezielles Plug-In, welches einem vorhandenen Plug-In, zur bereits vorhandenen Funktionalität, zusätzliche bereit stellt. Das Plug-In ist ohne das Fragment funktionsfähig, umgekehrt gilt dies nicht.

Ein typisches Beispiel für Fragmente tritt bei der Internationalisierung der Plug-Ins auf. Die sprachspezifischen Elemente, wie etwas Strings oder Bilder können in speziellen Dateien hinterlegt werden. Dabei entsteht für jede Sprache ein Fragment.

10.5 Deployment

Nach der Erstellung eines eigenen Plug-Ins gibt es verschiedene Möglichkeiten, es in eine existierende Eclipse-Installation zu integrieren. Dazu mssen die Jar-Dateien und die anderen Ressourcen des Plug-Ins kopiert werden. Dies ist auf verschiedene Arten möglich:

1. Das automatische Upgrade über einen Updateserver. Dabei wird eine Beschreibungsdatei vom Server geladen, Informationen, wie etwa die Lizenz, angezeigt und die Installationsdateien in die jeweiligen Verzeichnisse `feature` und `plugin` kopiert.
2. Manuelles kopieren der Plug-In-Ressourcen in die jeweilgen Verzeichnisse. Üblicherweise reicht es aus, die Dateien des Plug-Ins in das `plugin`-Unterverzeichnis zu kopieren.

Nun muss Eclipse nur neu gestartet werden, da die Plug-In-Manifeste ausschließlich beim Programmstart gelesen werden.

10.6 Effects

Effects ist ein Editorframework für die dreidimensionale Darstellung und Manipulation von Softwarestrukturen, [1]. Die vorgegangene Projektgruppe (PG 444) des Lehrstuhls X mit dem Thema, Eclipse Framework for Editing Complex Three-Dimensional Software Visualizations, und der Softwarelösung Effects, hat extensiv Java3D und das Plug-In Konzept von Eclipse verwendet.
Dabei wurden die Kernkomponenten als Eclipse Plug-Ins realisiert, wohingegen die Erweiterungen, wie neue 3D-Diagrammtypen (eigene Diagramme wie ein spezielles Sequenzdiagram,

3D-Diagramme, etc.), durch Fragmente realisiert werden können. Die erfassten Informationen können in einem XML-Dokument gespeichert werden. Der Benutzer wird durch eine eigene Effects-Perspektive, Views und eine Integration von Effects in das Eclipse-Hilfesystem unterstützt. Dieser Ansatz würde auch für die im Rahmen der PG 458 gestellten Anforderung anbieten.

10.7 Synopsis

Die hier vorgestellten Aspekte des Plug-In Mechanismus von Eclipse werden durch eine Vielzahl von weiteren technischen Betrachtungen ergänzt. Diese würden aber den Rahmen dieser Ausarbeitung bei weitem sprengen. Erwähnenswert, zukünftig betrachtenswert und wertvoll im Rahmen dieser Projektgruppe sind.

- `Naturen`: Sie sind eine Spezialisierung von allgemeinen Projekten in Eclipse. Beispielsweise wird aus einem allgemeinen Projekt z.B. ein Java- oder C++-Projekt.
- `Builder`: Sie sind Module, die auf bestehenden Sourcecode operieren und daraus etwas anderes machen, z.B. der Java-Compiler oder Codemetrik-Tools. Im Rahmen dieser PG könnte ihre Aufgabe eine Vorbereitung des Debug-Prozesses sein.
- `Markierungen`: Sie sind ein Mechanismus von Eclipse, benutzerdefinierte Informationen mit Ressourcen zu verbinden. Ein Beispiel sind die Fehlermarkierungen im Sourcecodeeditor nach einem fehlgeschlagenen Compilerlauf oder benutzerdefinierte Breakpoints, die zum Debuggen verwendet werden.

Eclipse und speziell die besprochenen Plug-In Mechanismen bieten eine gute Möglichkeit für die Realisierung der im Rahmen des Seminars erarbeiteten Systemmetapher der Projektgruppe.

KAPITEL 11

Debugging APIs

Jonas Mathis

11.1 Einführung

Die Ausarbeitung zum PG-Seminar gibt eine kurze Einführung in die internen Abläufe beim Debugging einer Java-Applikation. Zum einen werden dabei Konzepte der Java-API, wie auch zum anderen Konzepte der Debugging-API der Eclipse-Entwicklungsumgebung angesprochen. Der Text beruht insbesondere auf den Online-Artikeln Szurszewski (2003) und Wright und Freeman-Benson (2004), sowie den Seiten JPDA und Eclipse JDT.

Das Debugging einer Applikation erfolgt aus Entwicklersicht auf verschiedenen Ebenen. Es ist dabei möglich, sowohl über ein grafisches Benutzer-Interface (GUI) als auch z.B. den Assemblercode auf das Programm zur Laufzeit zuzugreifen. Bei der Entwicklung eines Debuggers muß deshalb die Möglichkeit geschaffen werden, alle verfügbaren Informationen über den laufenden Prozeß zu sammeln und darzustellen. Beim Debugging einer Java-Anwendung werden daher die Informationen und Inhalte der Virtual Maschine (VM) betrachtet, die das Ziel-Programm ausführt.
Intern stehen mehrere Schichten für das Debugging einer Java-VM, bzw. der darauf laufenden Applikation zur Verfügung. Das Grundkonzept bildet hierbei die Java Platform Debugging Architecture (JPDA). Auf der JPDA aufbauend bieten die Java Development Tools (JDT) der Entwicklungsumgebung Eclipse eine „high-level“- Zugriffsmöglichkeit auf Debuginformationen. Um einen Debugger für Eclipse selber entwickeln zu können, ist die Kenntnis der internen Konzepte von JPDA und JDT unerläßlich, da sie die Programmierschnittstelle (API) bereitstellen.

11.2 Grundlagen des Debuggings in Java

Die Java Platform Debugging Architecture (JPDA) ist der Ausgangspunkt für die Betrachtung der internen Abläufe einer VM und so auch für Betrachtungen der Objektebene des zu untersuchenden Programms.

11.2.1 Java Platform Debugging Architecture

Durch die Java Platform Debugging Architecture (JPDA) wird ermöglicht, auf verschiedenen Ebenen auf Virtuelle Maschinen und Applikationen Einfluß zu nehmen. Zu diesem Zweck stellt sie mehrere Komponenten bereit, deren Zusammenspiel in Abbildung 11.1 dargestellt wird.
Generell ist festzuhalten, daß größtenteils Interfaces zum Abfragen der Debuginformationen dienen. Auf der Ebene der Virtuellen Maschine wird das Java VM Debug Interface (JVMDI) verwendet, um Informationen auszulesen, die dann wiederum über das Java Debug Interface (JDI) dem Debugging-Prozeß bereitgestellt werden. Weiterhin wird eine Art Zwischenschicht verwendet, indem auf der VM-Seite das Back-end und auf der Seite des Debuggers das Front-end miteinander kommunizieren. Die Kommunikation zwischen Front- und Back-end wird durch das Java Debug Wire Protocol (JDWP) festgelegt, welches das Layout der zu übertragenen Pakete bzw. Informationseinheiten definiert.

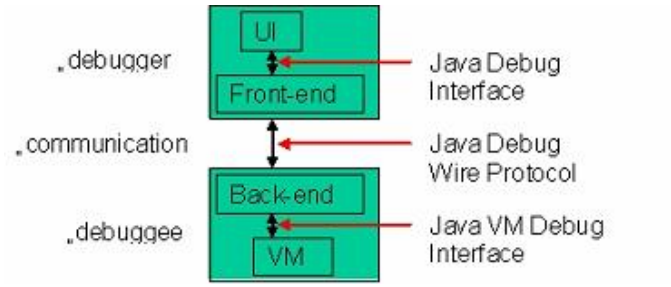


Abbildung 11.1.: Schematische Darstellung der Komponenten

Im folgenden sollen die einzelnen Komponenten genauer betrachtet werden.

Debuggee

Unter Debuggee ist der zu untersuchende Prozeß zu verstehen. Dieser läßt sich weiter unterteilen in die laufende Anwendung (nicht eingezeichnet), die entsprechende VM, die die Anwendung ausführt und das Back-end der JPDA.

VM Die Virtuelle Maschine führt die Anwendung aus. - Detaillierte Kenntnisse über das Prinzip der Virtuellen Maschine sind für das Verständnis des Debugging-Konzeptes nicht relevant, weshalb darauf nicht näher eingegangen wird.
Die VM muß, um das Debugging zu unterstützen, das Java VM Debug Interface implementieren. Eine Refrenzimplementierung liefert die im Java SDK enthaltene HotSpot Virtual Machine. Jedoch existieren auch Virtuelle Maschinen, welche das JVMDI nicht implementieren und somit keine Debug-Funktionalität bereitstellen.

Back-end Das in einer „nativen“ Sprache geschriebene Back-end leitet Anfragen vom Front-end des Debuggers an die VM weiter, indem es die JVMDI Schnittstelle nutzt. Des weiteren leitet es die Antworten der VM an das Front-end weiter. Dazu wird der Kommunikationskanal mit dem Java Debug Wire Protocol (JDWP) genutzt.

Kommunikation

Sämtliche Kommunikation, das Senden von Anfragen („requests“) und Antworten („responses“), geschieht durch Benutzung des Java Debug Wire Protocol. Dieses definiert das Format und Layout der zu übertragenden Daten, jedoch nicht die Art des Transports. So kann der Datenaustausch z.B. sowohl über Sockets, als auch durch Shared Memory-Zugriffe geschehen. Dadurch kann das Debugging der Java Applikation lokal, auf demselben Rechner, oder per Remote-Zugriff erfolgen.

Debugger

Ebenso wie der Debuggee besteht der Debugger aus zwei integralen Teilen, in diesem Fall aus User-Interface und Front-end. Das User-Interface entspricht der sichtbaren Debugging-Applikation, mit der der Nutzer interagiert. Intern wird vom vermeintlichen „Debugger“ das JDI des Front-end genutzt.
Es ist anzumerken, daß der Debugger-Prozeß und der zu untersuchende Prozeß aus Sicherheitsgründen (z.B. zur Vermeidung von Heisenbugs Telles und Hsieh (2001)) in zwei verschiedenen Virtual Machines laufen.
Der typische Ablauf der Kommunikation stellt sich wie folgt dar: der Nutzer arbeitet mit dem User-Interface. Dieses stellt Anfragen an das Front-end, indem es das dort implementierte Java Debug Interface nutzt. Die Anfrage wird via JDWP an das auf der Ziel-VM laufende Back-end geschickt. Die benötigten Informationen erhält das Back-end dann durch die Aufrufe des JVMDI der VM. Die generierte Antwort nimmt den umgekehrten Weg zurück.

11.2.2 Java Debug Interface

Das JDI regelt den Zugriff auf Debuginformationen. Ein Debugger kann direkt unter Rückgriff auf JDI geschrieben werden. Die Schnittstelle liefert dem Klienten Informationen über den Zustand einer VM, und somit über die Objekte eines laufenden Programms. Funktionsaufrufe können verfolgt werden. Die Werte von Variablen können ausgelesen werden, wobei sog. „Mirrors“ Variablen, Arrays und Werte kapseln. Des weiteren erlaubt das JDI die Kontrolle über die gewünschte VM. So können Threads angehalten und wieder gestartet, Breakpoints gesetzt und Benachrichtigung bei Exceptions, dem (Nach-)Laden einer Klasse, oder der Erzeugung eines neuen Threads empfangen werden.
Das JDI wird durch mehrere Pakete in *com.sun.jdi.** implementiert. Einige der Klassen der API werden im folgenden beschrieben:
com.sun.jdi liefert den Hauptbestandteil von JDI, indem es „Mirrors“ für Werte, Typen und die

Ziel-VM (`VirtualMachine`) definiert. Die Verbindung zur VM wird üblicherweise durch einen `VirtualMachineManager` erzeugt. Die Verbindung (per `Connector`) zwischen der VM, auf der der Debugger läuft und der Ziel-VM des zu untersuchenden Prozesses wird üblicherweise mit Hilfe der Klassen in *com.sun.jdi.connect* geschaffen. Sie ermöglichen das Starten von Virtuellen Maschinen (`LaunchingConnector`). Außerdem ist es auch möglich, auf Verbindungen zu warten. Um den internen Ablauf kontrollieren und steuern zu können wird in *com.sun.jdi.event* ein Ereignisbehandlungsmechanismus definiert, samt möglicher Ereignisse, z.B. BreakpointEvent, ExceptionEvent, VMDeathEvent, WatchpointEvent und andere. Solche Ereignisse werden stets zusammenfaßt als `EventSet`, welche wiederum aus der `EventQueue` entnommen werden. Eine Filterung für das Versenden von Ereignissen kann durch die in *com.sun.jdi.request* enthaltenen Klassen erreicht werden. Sie legen fest, für welche Ereignisse tatsächlich Events gesendet werden sollen.
Die Gesamtheit aller für das Debugging eines Programms benötigten Klassen und Interfaces wird als „Debug Model“ bezeichnet.

Zusammenfassend ist festzuhalten, daß man für das Debugging mit Hilfe der JPDA ein komplettes (auf das zu untersuchende Programm abgestimmtes) „Debug Model“ entwerfen muß, welches auch die Verbindung zur VM beinhalten muß.

11.3 Grundlagen des Debuggings in Eclipse

Eclipse stellt seine eigene API bereit, die Java Development Toolkit, welche als Plug-In realisiert sind. In den Paketen *org.eclipse.debug.** und darauf aufbauend *org.eclipse.jdt.debug.** findet sich das Framework einer Implementierung des „Debug Models“ auf der Grundlage der Java Platform Debug Architecture. Dieser Implementierungs-Rahmen wird als „Platform Debug Model“ bezeichnet und bildet das Herzstück der Debugging-Konzepte von Eclipse.

11.3.1 Launching

Bevor ein laufender Prozeß überhaupt untersucht werden kann, muß er erst einmal gestartet werden. In Eclipse muß dazu das zu untersuchende Programm in einer „Debug Configuration“ gestartet werden, welche besondere Attribute für das Debugging festlegt. Dazu wird ein Objekt erzeugt, das die Schnittstelle `ILaunchConfiguration` implementiert und somit eine Menge von Attribut-Wert-Paaren enthält, die den Programmaufruf und -ablauf beeinflussen. Aus dem Startvorgang der Applikation resultiert ein `ILaunch`-Objekt, über welches auf die Informationen während des Debuggings zugegriffen werden kann. Die Typen von Informationen, um die es sich dabei handeln kann, wird zusätzlich im folgenden erläutert.

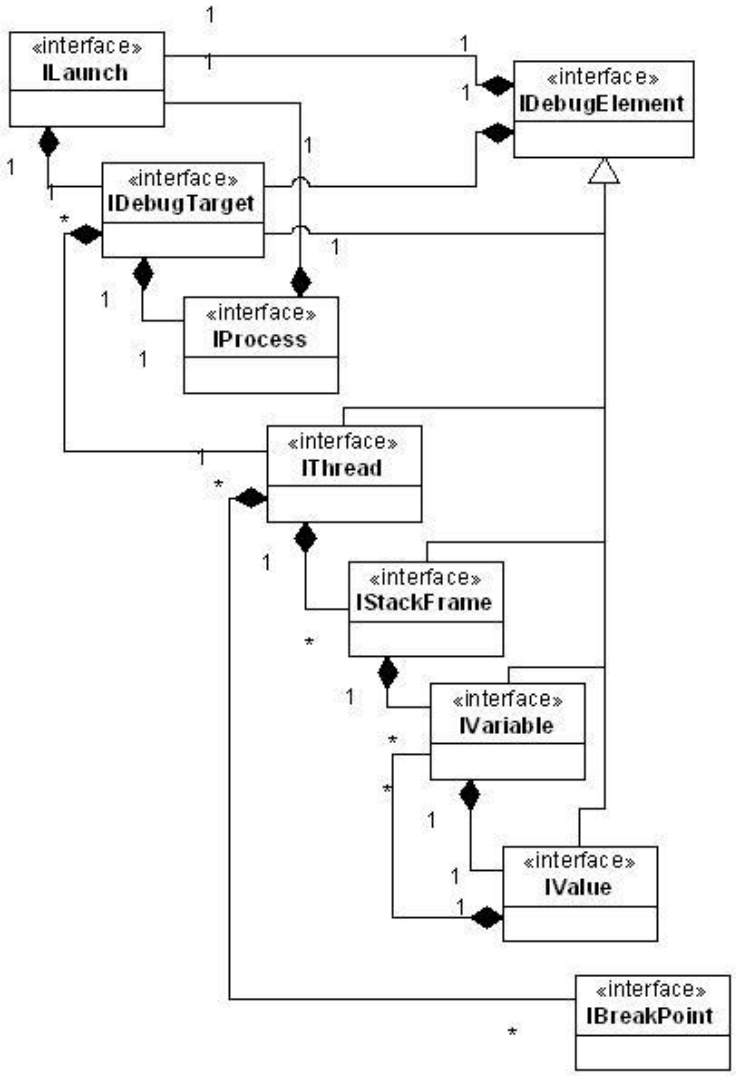


Abbildung 11.2.: Übersicht über Debug-Elemente

11.3.2 Platform Debug Model

Das Eclipse-eigene Platform Debug Model ist eine Sammlung von Klassen und insbesondere Schnittstellen, die implementiert und erweitert werden, um das Grundgerüst für Laufzeituntersuchungen von Applikationen zu bilden. Schnittstellen existieren für verschiedene Aspekte des Debugging-Prozesses. So unterscheidet man zwischen Artefakten („artifacts“), also irgendwie gearteten Teilen des Programms, Aktionen („actions“) und Ereignissen („Events“).

Artefakte

Alle Artefakte, die während des Debuggings benötigt werden, implementieren die Schnittstelle `IDebugElement`. Diese dient der Bereitstellung von Informationen über den zu debuggenden Ziel-Prozeß. Das Konzept der Debug-Elemente ist eine abstrakte Sicht, weshalb noch viele weitere, näher spezifizierte und von `IDebugElement` abgeleitete Schnittstellen zur Verfügung stehen. Eine Übersicht liefert Abbildung 11.2 auf der vorherigen Seite.
Die zu untersuchende VM, bzw. der zu verfolgende Prozeß, wird in `IDebugTarget` gekapselt. Die Schnittstelle `IThread` stellt einen Thread des Prozesses dar. Dieser kann durch Methodenaufrufe angehalten, fortgesetzt oder beendet werden. Außerdem besteht die Möglichkeit, die Ausführung des Programms schrittweise fortzusetzten. Darüber hinaus beinhaltet ein angehaltener Thread eine Menge von sogenannten „Stack Frames“. Diese, durch `IStackFrame` gekapselten Entitäten stellen den internen Zustand des Threads dar. Sie enthalten u. a. lokale Variablen und ermöglichen ebenfalls die Ausführungskontrolle des enthaltenden Threads. Variablen, bzw. komplexe Datenstrukturen implementieren die Schnittstelle `IVariable`. Auf den Wert einer Variablen kann über die Schnittstelle `IValue` zugegriffen werden. Auf Werte eines von der VM verwalteten Registers läßt sich mit Hilfe von `IRegister` zugreifen.
Es stehen des weiteren Schnittstellen für Expressions bereit, also für Codeausschnitte, die sich zu einem Wert auswerten lassen, und auch für Breakpoints innerhalb des Programms, d.h. vorbestimmte Stellen im Quelltext, zu denen die Ausführung des Programms angehalten werden soll.

Aktionen

Das Verhalten von Debug-Elementen, wie zum Beispiel die Möglichkeit, einen Thread anzuhalten, wird durch eine Menge von Schnittstellen implementiert, welche die übergreifende Bezeichnung „actions“ trägt.
Hier sind insbesondere die Schnittstellen `IDisconnect`, `IStep`, `ISuspendResume` und `IValueModification` zu nennen. Bei einer Virtuellen Maschine gibt `IDisconnect` an, daß für den Debugger die Möglichkeit besteht, sich vom Zielprogramm, bzw der laufenden VM zu lösen. `IStep` und `ISuspendResume` sind hauptsächlich für Prozesse, Threads und Stack Frames vorgesehen. `IStep` bietet dabei die Möglichkeit, die Ausführung des Programms schrittweise fortschreiten zu lassen.
`ISuspendResume` beschreibt die Fähigkeit, die Ausführung des Programms anzuhalten und gegebenenfalls wieder fortzusetzen.

Ereignisse

Es handelt sich hierbei um die Ereignisse, die während der Ausführung eines Programms auftreten können. Neben speziellen Ereignissen für einzelne Debug-Elemente gibt es auch „globale“ Ereignisse wie `STATE` (ein Element, jedoch nicht seine Sub-Elemente, hat sich geändert) und `CONTENT` (ein Element, bzw in ihm enthaltene Elemente haben sich geändert). Am Beispiel des Breakpoints soll das Konzept der Benachrichtigung über Ereignisse erläutert werden.

Breakpoints Ein Breakpoint wird im Quellcode gesetzt, und der gesetzte und aktive Breakpoint wird dann beim Start der VM in dieser „installiert“, d.h. intern in der VM registriert. Es muß zusätzlich im Debugger ein `IBreakpointListener` implementiert werden, der seine Informationen wiederum von einem `IBreakpointManager` bezieht und sich dort registriert. Gelangt das Programm nun an den Breakpoint, so wird es an dieser Stelle angehalten und ein `SUSPEND`-Event wird gefeuert. In diesem Event-Objekt enthält dann das Feld für den Grund des Stops den Hinweis „Breakpoint“.

Expressions Ähnlich den Breakpoints werden auch Expressions behandelt: unter Zuhilfenahme eines `IExpressionManager` werden `IExpressions` verwaltet und ausgewertet.

Durch die drei Aspekte Artifacts, Actions und Events wird also das Debug Model beschrieben. Um diese Interna vor dem Benutzer des Debuggers zu verstecken und ein einfaches Arbeiten zu ermöglichen, ist eine sogenannte „Debug Model Representation“ nötig. Außerdem muß ein auf das Debugging ausgerichtetes Benutzer-Interface zu Verfügung gestellt werden.

11.3.3 User-Interface

Unter Eclipse wird das User-Interface durch das Konzept der *DebugUITools* bereitgestellt. Es wird ein Rahmen für das Benutzer-Interface vorgegeben, sowie einzelne visuelle Elemente aus dem Bereich des Debuggings, wie z.B. TreeViews für Datenstrukturen oder auch Watchlists für Variablen. Den *DebugUITools* zugrundeliegend muß eine „Debug Model Representation“ festgelegt werden.

Debug Model Representation

Die Debug Model Representation (DMR) wird durch die Schnittstelle `IDebugModelRepresentation` modelliert. Sie ermöglicht den Zugriff auf Labels, Grafiken und insbesondere Editoren für die einzelnen Debug-Elemente und bildet somit den Ausgangspunkt für das grafische Debugging einer Applikation. Ein Grundgerüst dazu wird von `IDebugView` geliefert.

Neben der Darstellung von Informationen über einzelne Teile (Objekte) des Programms ermöglicht die DMR auch die Verknüpfung von aktuellem Zustand der VM zu Codeabschnitten. Dieses sog. „Source code lookup“ ermöglicht, daß der Benutzer z.B. den Quelltext zeilenweise durchlaufen kann und sich gleichzeitig die Veränderungen durch und nach den einzelnen Schritten ansehen kann. Außer der (zumeist schwierigen) Implementierung der Schnittstelle `IDebugView` , um erweiterte Funktionalität, wie vorgefertigte Dialoge, Ansichten und Behandlungsroutinen für bestimmte Mausklicks zu erhalten, besteht auch die Möglichkeit, die Klasse `AbstractDebugView` zu erweitern.

Fazit

Wie die Betrachtung von JPDA bzw. Eclipse JDT zeigt, ist der Entwurf eines Debuggers nicht ohne gute Kenntnisse der zugrundeliegenden Konzepte möglich. In beiden Fällen muß eine große Anzahl Klassen und unter Eclipse insbesondere Schnittstellen für das zu lösende Problem implementiert werden. Selbst ein - sicherlich wünschenswertes - kleines Beispiel würde daher den Rahmen dieser Ausarbeitung sprengen.

KAPITEL 12

eXtreme Programming

Daniel Maliga, Antonio Pedicillo, Dennis Weyland

12.1 Einleitung

Dieses Kapitel dient der Vorstellung des eXtreme Programming (XP), einer Methode zur Software-Entwicklung, um diese in der PG458 einsetzen zu können.
Zu Beginn werden wir anhand der grundlegenden Werte und Prinzipien von XP die konzeptionellen Unterschiede dieser Methode gegenüber herkömmlichen Ansätzen der Software-Entwicklung erläutern. Anschließend werden wir näher auf die zwölf Techniken von XP eingehen, mit deren Einsatz die Prinzipien von XP verwirklicht werden sollen.

12.2 Warum XP?

12.2.1 Grundprobleme der Softwareentwicklung

Die Entwicklung von Software hat mit ein paar wenigen Grundrisiken zu kämpfen.
Ein Risiko bei der Verwirklichung eines Software-Projektes besteht zum Beispiel darin, den Termin für die Fertigstellung nicht einhalten zu können oder nur Software ausliefern zu können, die fehlerbehaftet ist oder keine ausreichende Funktionalität bereitstellt. Daraus resultieren die Risiken der Nichtbenutzung der Software oder eines Auftragsverlustes. Weitere Risiken sind mangelnde Wartbarkeit des Programmes, bedingt durch immer weiter fortschreitende Aufblähung oder schlechte Dokumentation, sowie der Einbau überflüssiger Funktionalitäten oder gar völlig falscher Features aufgrund von Kommunikationsproblemen zwischen Auftraggeber und Entwicklern. Diese Probleme können auch durch die oft große Zeitspanne zwischen der Erteilung eines Entwicklungsauftrags und der Fertigstellung der Software verursacht werden.

12.2.2 XP als Lösung

XP hilft dabei, diese Risiken zu minimieren. Dies wird durch die Kombination von verschiedenen Techniken erreicht. So werden zum Beispiel die wichtigsten Funktionalitäten des Programms zuerst implementiert und es existiert zu jedem Zeitpunkt eine lauffähige Version der bis dahin verwirklichten Programmteile (mögen diese auch noch so rudimentär sein), so dass selbst bei einer Verspätung die Software schon zum vorgesehenen Zeitpunkt der Auslieferung mit den wichtigsten Funktionen benutzbar ist. Die ständige Lauffähigkeit wird durch extrem häufige, automatisierte Tests sichergestellt. Diese Tests sorgen zusammen mit der Prämisse, immer möglichst simple Lösungen zu suchen, auch für gute Wartbarkeit. Der Fokus auf Einfachheit erleichtert außerdem auch die Erweiterung des Codes sowie eine beschleunigte Fertigstellung des Produktes. Weiterhin stellt die Tatsache, dass der Auftraggeber eng in den Entwicklungsprozess eingebunden ist, eine große Hilfe im Bemühen dar, keine überflüssigen, unwichtigen oder falschen Funktionen einzubauen. Der Kunde ist gewissermaßen der „Fahrer“ der Entwicklung, der vom Team ständig Rückmeldung darüber bekommt, ob der Prozess noch in der richtigen Bahn läuft.

12.3 Werte von XP

Die Philosophie von XP basiert auf vier zunächst recht abstrakten Werten: *Kommunikation*, *Einfachheit*, *Feedback* und *Mut*. Dies sind die Grundpfeiler von XP, die jedes Mitglied eines XP-Teams ähnlich den sozialen Werten einer Gesellschaft verinnerlicht haben muss, damit die Teamarbeit funktionieren kann. Im Folgenden werden wir die vier Werte näher erläutern.

12.3.1 Kommunikation

Software wird im Allgemeinen in Teamarbeit entwickelt. Probleme entstehen in den allermeisten Fällen durch mangelnde Kommunikation, zum Beispiel weil einer der Programmierer es versäumt, seine Kollegen über eine wichtige Designänderung zu informieren. Schlechte Kommunikation kann auch durch ein schlechtes Betriebsklima bedingt sein, in dem wichtige Informationen beispielsweise aus Angst vor Bestrafung nur unwillig mitgeteilt werden.
XP stellt intensive Kommunikation durch eine Reihe von Techniken sicher, die ein hohes Maß an persönlicher Kommunikation erfordern, wie etwa das Programmieren in Paaren. Desweiteren kann ein XP-Coach Probleme zwischen den Teammitgliedern erkennen und entsprechend darauf reagieren.

12.3.2 Einfachheit

Bei XP wird immer eine möglichst einfache Lösung gesucht, die die gesuchte Funktionalität gerade noch realisiert. Falls sich später herausstellt, dass die Funktionalität erweitert werden

muss, dann geschieht dies erst dann, wenn es notwendig wird. Dies ist besser, als von Vornherein zeitaufwändig an einer komplexen Lösung zu arbeiten, die hinterher überhaupt nicht gebraucht wird.

12.3.3 Feedback

Ein essentieller Baustein von XP ist Feedback, welches auf verschiedenen Ebenen existiert. Das zu implementierende Softwaresystem gibt über Tests bzw. deren Ergebnisse Feedback an die Programmierer. Die Programmierer geben Feedback über den Systemstatus und über die User-Stories (in etwa: Anwendungsfälle) an den Kunden weiter. Der Kunde wiederum gibt Feedback über den schon ausgelieferten Teil der Software an die Programmierer.
Feedback steht im engen Zusammenhang mit Kommunikation und ist besonders wichtig für das Gelingen eines XP-Projektes, da vom Wissen über Programmieraufwand und Mängel der Software das weitere Vorgehen abhängt.

12.3.4 Mut

XP verfolgt eine offensive Arbeitsphilosophie. Um diese Philosophie umzusetzen, wird den Beteiligten Mut abverlangt: Mut, Fehler einzugestehen; Mut, mühsam geschriebenen Code zu löschen, wenn man mit ihm der Problemlösung nicht mehr näherkommt; Mut, auch mal Ansätze auszuprobieren, die auf den ersten Blick abwegig oder nicht realisierbar erscheinen. Beim Arbeiten mit XP darf man sich nicht scheuen, Fehler zu machen, und man darf sich ebenso nicht scheuen, diese Fehler dann wieder zu korrigieren.

12.4 Prinzipien von XP

Die Prinzipien stellen Handlungsleitfäden dar, mit deren Hilfe ein Entwicklerteam die im vorigen Abschnitt vorgestellten Werte erreichen kann. Nach Beck (2000) gibt es fünf zentrale Prinzipien (*unmittelbares Feedback*, *Einfachheit anstreben*, *inkrementelle Veränderung*, *Veränderungen wollen*, *Qualitätsarbeit*) sowie noch ein paar weitere, weniger wichtige Prinzipien. In diesem Abschnitt werden wir die Prinzipien näher betrachten.

12.4.1 Unmittelbares Feedback

Aus der Lernpsychologie ist bekannt, dass ein Lerneffekt umso größer ist, je schneller die Rückmeldung über eine Handlung kommt. Dementsprechend wird Feedback bei XP möglichst unmittelbar gegeben, einerseits durch Komponententests, andererseits durch engen Kontakt mit dem Kunden und kurze Releasezyklen.

12.4.2 Einfachheit anstreben

Jedes Problem wird prinzipiell angegangen, als ob es eine einfache Lösung dafür gäbe. Nach Beck (2000) trifft dies nämlich in 98% der Fälle zu. Die durch dieses Vorgehen erzielte Zeitersparnis ist höher als der zusätzliche Aufwand für die Korrektur der restlichen 2%.

12.4.3 Inkrementelle Veränderung

Die Erfahrung zeigt, dass große, umfassende Veränderungen, die gleichzeitig vorgenommen werden, nur in den seltensten Fällen funktionieren. Daher ist es besser, Modifikationen in kleine Schritte aufzuteilen und nach jedem Schritt zu testen, ob das Gesamtsystem noch funktioniert.

12.4.4 Veränderung wollen

Ein XP-Projekt ist hochgradig dynamisch. Sowohl die Anforderungen an das Produkt als auch die Programmtexte selbst verändern sich im Laufe der Zeit dramatisch. Deshalb müssen die Mitglieder eines XP-Teams Veränderungen nicht nur akzeptieren, sondern sogar als angenehm und vorteilhaft willkommen heißen. Die beste Strategie ist daher immer diejenige, die das aktuelle Problem löst und gleichzeitig die meisten Optionen für die Zukunft offenlässt.

12.4.5 Qualitätsarbeit

Niemandem macht es Spaß, schlechte Resultate zu liefern. Programmierer wollen Programme schreiben, die qualitativ hochwertig sind, so dass sie auf die geleistete Arbeit stolz sein können. Wird das Team in diesem Bestreben nicht unterstützt, so wirkt sich das negativ auf die Motivation der Mitarbeiter aus, worunter wiederum die Produktqualität leidet.

12.4.6 Weitere Prinzipien

Neben den oben erwähnten zentralen Prinzipien gibt es nach Beck (2000) noch eine Reihe weiterer Prinzipien, von denen wir einen Teil hier aufführen.

Geringe Anfangsinvestition: Durch einen anfangs niedrigen Kapitalaufwand findet sowohl eine Risikominimierung als auch eine Fokussierung auf die wichtigsten Aspekte des Projektes statt.

Auf Sieg spielen: Im Gegensatz zu vielen „herkömmlichen“ Teams, die bei der Entwicklung eines Produktes vor allem darauf bedacht sind, keine Fehler zu begehen und nach Vorschrift zu arbeiten, ist ein XP-Team darauf aus, die ihm gestellte Aufgabe möglichst gut zu erledigen.

Gezielte Experimente: Entscheidungen sollen immer durch konkrete Tests abgesichert werden, um Unsicherheiten über die Richtigkeit zu vermeiden.

Offene, ehrliche Kommunikation: Das Betriebsklima muss es erlauben, auch schlechte Nachrichten zu überbringen und Kritikpunkte anzusprechen.

Instinkte berücksichtigen, nicht dagegen arbeiten: Der Arbeitsprozess zielt zwar auf ein langfristiges, gemeinsames Ziel, muss aber auch die kurzfristigen Eigeninteressen der einzelnen Mitarbeiter einbeziehen, wie etwa Lernen, Interaktion, Kontrolle.

Verantwortlichkeit annehmen: Aufgaben dürfen nicht einfach einem Teammitglied zugewiesen werden, sondern müssen freiwillig von jemandem übernommen werden.

Lokale Anpassung: XP stellt kein universell gültiges Konzept dar. Daher sind Anpassungen an die örtlichen Verhältnisse nicht nur erlaubt, sondern durchaus erwünscht und eventuell notwendig

Leichtes Gepäck: Bei einem XP-Projekt muss man immer bereit sein, zusammen mit dem Projekt in veränderte Bedingungen zu „reisen“. Dafür sind viele Artefakte, die „mitgeschleppt“ werden müssten, nur hinderlich.

Ehrliche Einschätzung: Beim Schätzen von Zeitaufwänden u.ä. dürfen keine zu detaillierten Angaben, die jeder Wissensgrundlage entbehren, gemacht werden.

12.5 Techniken von XP

Unter Techniken versteht man im Zusammenhang mit XP die zwölf Verfahren, mit denen unter XP gearbeitet wird. Diese Techniken sind konkrete Umsetzungen der Werte und Prinzipien. Sie greifen ineinander und unterstützen sich gegenseitig. Dementsprechend kann man nach Beck (2000) zwar auch nur einen Teil der Techniken verwenden, es stellt sich jedoch ein weitaus größerer Vorteil ein, wenn man alle Techniken anwendet.
In diesem Abschnitt werden wir die einzelnen Techniken vorstellen.

12.5.1 Metapher

Die Metapher dient dem Verständnis des zu realisierenden Systems durch alle Beteiligten und als gemeinsame Sprachgrundlage. Es ist demnach wichtig, dass alle Teammitglieder - auch und gerade der Kunde - die Metapher verinnerlichen und als sinnvoll ansehen. Ein Beispiel für eine Metapher ist die Bezeichnung einer Bildschirmoberfläche als Schreibtisch.
Sowohl das System als Ganzes, als auch die einzelnen Teile und ihre Verbindungen untereinander können von der Metapher erfasst werden, so dass man letztendlich nur das Vokabular der Metapher braucht, um das System zu beschreiben. Dies hilft bei der Gestaltung einer Systemarchitektur (Lippert u. a. (2002)). Beck (Beck (2000)) geht sogar soweit zu sagen, die Metapher könne die Architektur in weiten Teilen ersetzen.
Wichtig beim Einsatz einer Metapher ist eine zügige Feststellung, ob die konkrete Metapher zum letztendlich geschriebenen Code und den Tests passt. Ist dies nicht der Fall, behindert das Festhalten an einer untauglichen Metapher nur den weiteren Entwicklungsprozess. Damit zusammenhängend ist auch unmittelbar klar, dass die Metapher unter Umständen im Laufe des Projektes verändert werden muss.

12.5.2 Planungsspiel

Das Planungsspiel ist, wie der Name schon sagt, die Planungsphase in einem XP-Projekt, die wiederholt stattfindet. Im Planungsspiel existieren zwei Parteien, die Geschäftsseite und die Entwicklerseite. Erstere beschreibt die verlangten Funktionalitäten der Software auf sogenannten Story-Cards und entscheidet über deren Umfang und Priorisierung. Letztere schätzt für jede Story-Card den Implementierungsaufwand bzw. die dafür benötigte Zeit ab, klärt über technische Konsequenzen auf und entscheidet über den Entwicklungsprozess innerhalb eines Releases (dem Kunden zur Verfügung gestellte Version der Software). Die letztendliche Entscheidung, welche Funktionalitäten realisiert werden, trifft die Geschäftsseite aufgrund der Informationen, die die Entwicklerseite gegeben hat. Dabei werden die Anforderungen in Form der Story-Cards nach Wichtigkeit sortiert, um den Entwicklern die Planung innerhalb des Releases zu vereinfachen.
Ein wichtiger Aspekt des Planungsspiels ist die Zusammenarbeit zwischen Entwickler- und Geschäftsseite. Änderungen am Plan werden immer zusammen vorgenommen. Die Zusammenarbeit zwischen Auftraggeber und -nehmer ist beim Planungsspiel am intensivsten; hierbei hat der Kunde am ehesten die Möglichkeit, Probleme und Potentiale im Projektverlauf zu erkennen und entsprechend darauf zu reagieren.
Ein Release wird, sobald der Umfang feststeht, von den Entwicklern in sogenannte Iterationen aufgeteilt, die im Iterationsplanungsspiel behandelt werden. An dieser Variante des Planungsspiels sind nur die Entwickler beteiligt. Die User-Stories werden in Tasks aufgeteilt. Tasks stellen die technische Umsetzung einer Teilfunktionalität dar; sie können mehrere User-Stories betreffen und müssen keine komplett abdecken. Tasks werden von einzelnen Entwicklern übernommen, die dann den nötigen Aufwand abschätzen und die Tasks letztendlich implementieren. Im Rahmen des Iterationsplanungsspiels wird der Aufwand möglichst gerecht auf die Entwickler verteilt.

12.5.3 Kunde vor Ort

Der Kunde vor Ort ist Teil des Teams und wird im Idealfall durch einen der künftigen Nutzer des zu implementierenden Systems dargestellt. Ist dies nicht möglich, so kann diese Funktion auch durch einen Vertreter des Managements oder des Marketings ausgefüllt werden. Der Kunde besetzt im Planungsspiel die Geschäftsseite, schreibt also User-Stories und entscheidet über deren Priorität. Darüber hinaus steht er als Ansprechpartner für Fachfragen für die übrigen Entwickler zur Verfügung und schreibt Funktionstests, um die Umsetzung der Story-Cards auf Korrektheit zu prüfen.

12.5.4 Einfaches Design (Simple Design)

XP legt ein anderes Verständnis von Design zu Grunde. Es wird nicht zu Beginn des Projektes ein großes allumfassendes Design ausgearbeitet. Stattdessen wächst das Design mit dem Projekt. Somit sollte das Projekt zu jedem Zeitpunkt auf dem einfachsten möglichen Design basieren. Neue Problemstellungen erfordern ein verändertes Design. Dies wird durch ein jederzeit leicht und unkompliziert veränderbares System unterstützt. Teile des Designs, die im Vorraus programmiert werden, später aber aufgrund der sich schnell ändernden Produktanforderungen unnötig sind und somit nie zum Einsatz kommen und dadurch unnötige Kosten verursachen, wird man in einem XP Projekt selten oder im Idealfall gar nicht finden.
Die benötigten schnellen und unkomplizierten Änderungen lassen sich durch geeignetes Refactoring erzielen. Dabei muss die Funktionsweise des Systems durch Tests abgesichert sein. Programmierstandards sowie Programmieren in Paaren erleichtern diesen Vorgang und mit Hilfe von kurzen Releasezyklen kann der Erfolg des Designs sofort überprüft werden.

12.5.5 Kurze Releasezyklen (Short Releases)

Releasezyklen in XP sind verhältnismäßig kurz und betragen oftmals nur wenige Monate. Für jedes Release werden die darin vorkommenden Leistungsmerkmale immer komplett implementiert. Das System ist also mit jedem Release komplett einsatzfähig, was die implementierten Leistungsmerkmale angeht.
Auf der einen Seite erhält der Kunde mit jedem Releasezyklus ein einsatzfähiges System mit den wertvollsten Geschäftsanforderungen. Auf der anderen Seite erhalten die Programmierer auf diese Art und Weise sehr schnell Feedback vom Kunden und können darauf flexibel reagieren.

12.5.6 Fortlaufende Integration (Continuous Integration)

Der Sourcecode wird bei XP fortlaufend, d.h. nach einigen Stunden, mindestens aber nach einem Tag Entwicklungsarbeit, integriert und getestet. Fehler werden sofort korrigiert, die Integration ist erst beendet wenn alle Tests wieder fehlerfrei ausgefhürt werden können. Kann

dies nicht garantiert werden, so müssen die Änderungen ersteinmal verworfen werden.
Um Konflikte beim Zusammenführen zu vermeiden, sollten die Integrationen häufig und schon nach kleineren Änderungen erfolgen. In diesem Fall sind auch die neu implementierten Funktionalitäten dem gesamten Team schnell verfügbar.
Bewährt hat sich in solchen Fällen zur Unterstützung der fortlaufenden Integration ein sogenannter Integrationsrechner, der dem Team physisch zur Verfügung steht und an dem die Integrationen durchgeführt werden können.

12.5.7 Testen (Testing)

Tests spielen in XP eine wichtige Rolle. Dabei wird zwischen Komponententests, die von den Programmierern während ihrer Arbeit geschrieben werden, und Akzeptanztests, die vom Kunden geschrieben oder in Auftrag gegeben werden, unterschieden. Praktisch von großem Vorteil sind dabei Testumgebungen, die automatisierte Tests unterstützen, und so den Testvorgang erheblich vereinfachen und beschleunigen.
Tests erhöhen einerseits das Vertrauen des Kunden in das System und damit auch in das Projekt und andererseits das Vertrauen der Programmierer in ihren Sourcecode. Zu jedem Zeitpunkt sollte also ein System vorliegen, das die bis dahin erstellten Komponententests erfolgreich besteht. Eine Integration ist wie oben beschrieben erst dann abgeschlossen, wenn dies der Fall ist.
Ein weiterer wichtiger Punkt in Bezug auf das Testen in XP ist das Prinzip des „Test First“. Man beginnt die Implementierung einer bestimmten Funktionalität zuerst mit dem Schreiben geeigneter Tests. So wird unter anderem auch sichergestellt, dass ein einfaches Design erstellt wird, das gerade dazu geeignet ist, die geforderten Funktionalitäten umzusetzen.

12.5.8 Refactoring

Refactoring ist die Umstrukturierung eines Softwaresystems unter Beibehaltung seiner nach außen sichtbaren Funktionalität. Ziel dabei ist eine verbesserte Struktur, ein möglichst einfaches Design. Dies sorgt kurzfristig gesehen zuerst einmal für einen Mehraufwand, der sich mittel- und langfristig allerdings auszahlen wird.
Durch richtig eingesetzes Refactoring erhöht sich die Qualität des Sourcecodes enorm. Z.B. kann durch Refactoring das Design erheblich vereinfacht werden. Auf diese Weise können die Programmierer flexibler auf neue oder sich verändernde Anforderungen reagieren. Änderungen am System sind dadurch schnell und einfach, selbst in fortgeschrittenen Projekten, möglich.
Refactoring kann allerdings nur funktionieren, wenn die Funktionalität des Systems durch geeignete Tests sichergestellt wird. So kann effizient überprüft werden, ob die vorgenommenen Änderungen auch die gewünschte Funktionalität beibehalten. Um bei größeren Änderungen unnötige Probleme zu verhindern, sollte man diese in kleinere „Refactorings“ aufteilen und diese dann nach und nach abarbeiten.

12.5.9 Programmieren in Paaren (Pair Programming)

Programmiert wird in XP immer zu zweit in einem Paar. Dabei nimmt jeder aus dem Paar eine andere Rolle ein. Ein Partner arbeitet mit Tastatur und Maus und übernimmt die Implementierungsarbeit. Dabei wird er vom anderen Partner unterstützt. Dieser erkennt kleinere Fehler, die immer wieder beim Programmieren auftreten, und denkt darüber hinaus strategischer. Er beschäftigt sich also nicht nur mit der gerade implementierten Methode an sich, sondern macht sich z.B. auch Gedanken über den Ansatz im Allgemeinen oder über weitere Testfälle, die noch nicht funktionieren. Er schaut also aus einer anderen, weiter entfernten, Perspektive auf das Problem. Die Rollen werden dabei sehr oft getauscht, genau wie die Zusammensetzung der Paare, die meist sogar täglich wechselt.
Durch das Programmieren in Paaren, insbesondere das häufige Wechseln der Partner, wird das Wissen eines Einzelnen über das ganze Team verteilt. Der Ausfall eines Programmieres (Krankheit, Verlassen des Projektes) hat dadurch nicht die fatalen Auswirkungen, die in anderen Projekten teilweise beobachtbar sind.
Testen und Refactoring wird durch Programmieren in Paaren vereinfacht. Die Einhaltung von Programmierstandards sowie einfaches Design und fortlaufende Integrationen werden gefördert.

12.5.10 Gemeinsame Verantwortlichkeit (Collective Ownership)

Es gibt grob gesehen drei Möglichkeiten, die Verantwortlichkeit in einem Projekt zu verteilen. Die erste Mölichkeit besteht darin, dass niemand für einen bestimmten Teil des Sourcecodes verantwortlich ist. Jeder kann ungeachtet von irgendwelchen Nebeneffekten Teile des Codes ändern. Chaos ist dabei vorprogrammiert. Der Sourcecode wächst zwar schnell an, mit ihm aber auch seine Instabilität.
Die zweite Mölichkeit, individuelle Verantwortlichkeit, überträgt die Verantwortlichkeit für bestimmte Codesegmente auf den dafür zuständigen Programmierer. Möchte jemand eine bestimmte Stelle im Sourcecode ändern, so muss er diese Änderung von dem dafür verantwortlichen Programmierer anfordern. Das System bleibt zwar stabil, entwickelt sich aber nicht optimal weiter.
Die dritte Möglichkeit ist eine Verteilung der Verantwortlichkeit auf das gesamte Team. Nicht jeder kennt dabei die einzelnen Teile des Systems gleich gut, weiß aber genug, um damit arbeiten zu können. Im Prinzip kann daher jeder Programmierer Änderungen an allen Teilen des Systems durchführen.
Diese dritte Möglichkeit ist Bestandteil von XP. Jeder kennt sich also mit dem gesamten System aus und ist in der Lage, Verbesserungen sofort durchzuführen, wenn er eine entsprechende Möglichkeit sieht. Der Ausfall eines Teammitglieds kann bei gemeinsamer Verantwortlichkeit leichter kompensiert werden.

12.5.11 Programmierstandards (Coding Standards)

Während eines XP-Projektes wechselt man ständig seine Partner und bearbeitet laufend andere Teile des Sourcecodes. Wird in einem solchen Projekt kein einheitlicher Programmierstil verwendet, dann führt dies jedesmal wieder zu unnötigen Einarbeitungszeiten, Fehlern durch schlechte Lesbarkeit des Sourcecodes, und ähnlichen Problemen. Abhilfe schaffen da Programmierstandards. Hierdurch sollte man nach einer gewissen Vorlaufzeit im Projekt nicht mehr erkennen können, wer aus dem Team welchen Teil Code geschrieben hat. Außerdem sollte durch Programmierstandards die Kommunikation im Sourcecode selbst gefördert werden.
Mit gemeinsamen Programmierstandards wird darüberhinaus ein einfacheres und schnelleres Refactoring möglich und gemeinsame Verantwortlichkeit unterstützt.

12.5.12 40-Stunden-Woche (40 Hour Week)

Wichtig für das XP ist die Einhaltung der 40-Stunden-Woche. Damit ist allerdings nicht gemeint, daß man jede Woche stur 40 Stunden am Projekt arbeiten soll. Es soll nur vermieden werden, daß man die Arbeit Tag für Tag unausgeruht und unmotiviert angeht. Darunter leidet nicht nur die Produktivität an sich, sondern auch die Qualität, in der man programmiert sowie das Verhältnis zu den Teamkollegen, um nur einige Auswirkungen zu nennen.
Ein ernsthaftes Problem liegt bereits vor, wenn man zwei Wochen hintereinander Überstunden macht. Viele Probleme lassen sich einfach nicht durch Überstunden lösen. Und Überstunden in zwei aufeinanderfolgenden Wochen wirken sich langfristig nur ungünstig auf den Projektfortschritt aus.
Ideal wäre es, wenn man Tag für Tag frisch und ausgeruht zur Arbeit kommt, um das Büro dann nach ein paar Stunden Arbeit, mit sich selbst und der eigenen Leistung zufrieden, zu verlassen. Das Wochenende sollte dazu genutzt werden, mal zwei Tage mit etwas anderem als Arbeit zu verbringen, um dann zu Wochenbeginn wieder erholt und voller neuer Ideen die Arbeit aufnehmen zu können.

12.6 Rollen

XP besteht nicht nur aus den Werten und Techniken, sondern wird auch durch die verschiedenen Rollen repräsentiert. Dabei muss nicht zwangsläufig jede Rolle einer bestimmten Person zugeschrieben sein. Es kann vorkommen, dass mehrere Rollen von einer Person übernommen werden. Kent Beck stellt in Beck (2000) die verschiedenen Rollen vor. Diese werden kurz aufgeführt. In den nachfolgenden Abschnitten werden wir im Detail die XP-Rollen beschreiben.

12.6.1 Kunde (Costumer)

Der Kunde verkörpert die Seite des XP-Projektes, die die genauesten Vorstellungen des späteren System besitzt. Ihm fi¿½lt die Aufgabe zu, die Story-Cards für die Entwickler anzufertigen. Hat der Kunde noch keine XP-Erfahrungen aufzuweisen, ist die Zusammenarbeit zwischen Entwickler und Kunden stärker. Der Kunde muss sich erst mit dem Erstellen der Story-Cards vertraut machen. Die ersten Story-Cards werden noch nicht optimal ausgearbeitet sein. Die Entwickler weisen durch Feedback den Kunden darauf hin, ob die Story-Cards vollständig sind oder noch ausgearbeitet werden müssen. Während des XP-Prozesses lernt der Kunde, wie umfangreich die Beschreibung sein soll und welche Informationen darin aufzunehmen oder auszuschließen sind. Somit befindet sich der Kunde in einem ständigen Lernprozess.
Wenn keine Test-Tools von der Entwicklerseite zur Verfügung gestellt werden, muss der Kunde lernen, Funktionstests (Akzeptanztests) zu schreiben. Das ist keine leichte Aufgabe, wenn es sich um komplexe Systeme handelt. Hier ist wieder das Feedback von den Entwicklern gefragt. Welche Testfälle sind wünschenswert? Und welche Testdaten wählt man dafür? Diese Fragen müssen geklärt werden, um einen reibungslosen XP-Prozess zu gewährleisten.

12.6.2 Programmierer (Programmer)

Die Programmierer erstellen aus den Story-Cards ein lauffähiges System. Ein Programmierer dokumentiert seinen geschriebenen Code und erstellt dazugehörige Komponententests. Somit verkörpert der Prorammierer auch die Rolle eines Testers.
Ein wichtiger Bestandteil ist die Kommunikation zwischen den Programmierern im XP-Prozess. Die Technik „Programmieren in Paaren“ setzt dies voraus. Für ein XP-Projekt wäre es kontraproduktiv, wenn es mehrere Prorammierer gäbe, die nicht teamfähig sind. Das Verhalten des Programmierers ist wichtig im XP-Prozess. Als XP-Programmierer ist Mut erforderlich und Ängste sollten eingestanden werden. Kent Beck hebt vier Ängste hervor (Beck (2000)):
Ein Programmierer hat Angst

- für dumm gehalten zu werden,
- für nutzlos gehalten zu werden,
- überflüssig zu werden,
- und nicht gut genug zu sein.

Es gehört Mut dazu, sich als Programmierer Ängste einzugestehen. XP lebt vom Mut. Das wissen auch alle Teilnehmer im XP-Projekt. Somit können Progammierer befreiter ihrer Arbeit nachgehen.

12.6.3 Tester

In XP sind die Komponententests so stark mit der Programmierung verkoppelt, dass die Programmierer diese selbst ausführen. Allein durch das „Test-First-Prinzip“ ließe sich eine personelle Trennung nicht durchführen. Daher konzentriert sich die Rolle des Testers in XP auf die für den Anwender relevanten Akzeptanztests.
Die Aufgabe des Testers ist es, den Anwender bei der Erstellung von Akzeptanztests, deren Formulierung und Umsetzung zur Seite zu stehen. Dies kann durch Hilfe von Test-Tools erleichert werden. Unabhängig davon, ob die Akzeptanztests mit Tools ausgeführt werden oder nicht, ist der Tester dafür zuständig, die Akzeptanztests regelmäßig durchzuführen. Auf dieser Basis gewinnt der Tester Einblick in den Fortschritt des Projektes. In der Literatur wird immer wieder empfohlen, dass man den eigenen Code nicht selbst testen soll. Ein Argument sind die Missverständnisse beim Interpretieren der Spezifikation der Anforderungen. Die Programmierer würden beim Testen – wie beim Programmieren – das funktionale Verhalten des Programms als richtig sehen, obwohl es nicht der Spezifikation entspricht. Es werden verschiedene Arten der Täuschung genannt:

- Bei der *Trick- oder Ablauftäuschung* weicht der Programmtext von dem ab, was normale Programmierer denken, d.h. ein Programmierer wendet bei seiner Programmierung ein Trick an, um das entsprechende Ziel zu erreichen, z.B. indem er in einer Funktion eine Abfrage einbaut, die eine Fehlerfreiheit des Moduls gewährleistet. Jedoch kann zu einem späteren Zeitpunkt nicht ersichtlich werden, ob ein Fehler in diesem Modul lag. Dies erschwert die Lesbarkeit und das Verständnis eines Programms für einen Tester.

- Eine *Erwartungstäuschung* entsteht durch Prägung oder Prädisposition, d.h. durch eine bestimmte Erwartung, mit der eine Person an die Aufgabe herangeht.
  Z.B. liest man als Tester den Kommentar einer Methode und schaut flüchtig oder oberflächlich über den Programmcode, da man die Erwartung hat, dass die Methode das bezweckt, was im Kommentar angegeben ist.

- Zu einer *Hemmungstäuschung* kommt es, wenn das zuvor erlernte Wissen falsch interpretiert wurde.

- Bei einer *Überdeckungstäuschung* überdeckt ein großer, starker und überwiegender Eindruck einen kleinen, schwachen oder unbedeutenden Eindruck.
  Z.B. „Testmethodon“ wird wie „Testmethoden“ gelesen. Das Gehirn erkennt das Wort und liest es als richtig, weil nicht auf jeden Buchstaben sondern auf das ganze Wort geachtet wird. Ein(e) Leseanfänger(in) liest jeden einzelnen Buchstaben und reproduziert das Wort. Somit würde eine Täuschung vermieden werden.

- Eine *Wiederholungstäuschung* entsteht durch den Analogieschluss, da scheinbar gleiche Vorgänge auch gleiche oder ähnliche Ergebnisse haben.
  Daher werden z.B. ähnliche Programmteile nicht mehr (oder flüchtig) geprüft, weil ein Programmteil schon als fehlerfrei beurteilt wurde.

Durch *Paar-Programmierung* können *Erwartungstäuschung* und *Überdeckungstäuschung* vermieden werden, da bekanntlich vier Augen mehr sehen als zwei Augen. Das *Refactoring* schließt eine *Wiederholungstäuschung* aus, da man bei XP besonderes darauf bedacht ist, aufkommende Redunanzen im Quellcode zu vermeiden.

12.6.4 Verfolger (Tracker)

Der Verfolger (unter Kent Beck: Terminmanager) stellt das „Gewissen“ des Teams dar. Er beobachtet den Projektfortschritt und sammelt dabei Informationen (Daten). Diese Daten dienen dem Verfolger als Feedback für die Entwickler. Er kann mit diesen Informationen erkennen, ob eine Iteration termingerecht abgeschlossen werden konnte. Somit können sich die Entwickler selbst ein Bild machen, wie gut oder schlecht sie in der Entwicklung des Systems liegen. Kommt es in einem Projekt vor, dass das System zu dem festgelegten Termin nicht fertiggestellt werden kann, muss durch Rücksprache mit den Kunden über eine mögliche Funktionalitätsreduktion nachgedacht werden.
Der Verfolger sollte über den gesamten Projektverlauf nicht dieselben Daten erheben und die dazugehörigen Statistiken erstellen. In Lippert u. a. (2002) wird ein Beispiel aufgeführt, dass dies in einem XP-Projekt wenig sinnvoll ist. In einem ihrer Projekte hatten die Entwickler einen hohen *Load-Factor*. Unter einem *Load-Factor* ist das Verhältnis zwischen den tatsächlichen erbrachten Aufwänden (Idealzeit) und der tatsächlichen verbrauchten Zeit (Realzeit) zu verstehen. Diesen haben die Entwickler versucht mit Hilfe von *Sprints* zu minimieren. Nach wenigen *Sprints* sank er zwischen 1,0 und 1,2. Daraufhin haben sie die Erhebung des *Load-Factors* eingestellt, denn dieser hatte sich wieder auf ein normales Niveau eingependelt. Sprints sind kurze Zeiträume von maximal 30 Tagen, an denen die Entwickler eines Teams so konzentriert wie möglich an der Entwicklung des zu erstellenden Softwaresystem arbeiten.
Aus diesem Beispiel geht hervor, dass der Verfolger gezielt wissen muss, welche Daten für den momentanen Projektverlauf relevant sind.

12.6.5 XP-Trainer (XP-Coach)

Ein XP-Trainer ist vergleichbar mit einem Fußballtrainer. Er kennt die Stärken und Schwächen seines Teams. Wenn seine Mannschaft in Rückstand gerät und nicht mehr selbst weiß, wie sie sich aus der Situation befreien sollen, greift der Trainer ein und lenkt das Team, bis es wieder auf eigenen Füßen steht. Er erkennt das Problem und versucht dieses durch gezielte Schritte zu lösen.
Ein XP-Trainer greift nur in das Projekt ein, wenn die Entwickler vom eigentlichen Weg abgekommen sind. Wenn das Team zum ersten Mal mit XP in Berührung kommt, dient der XP-Trainer als Mentor. Er besitzt das tiefgründige Wissen über XP und muss dieses vermitteln. Ohne einen XP-Trainer kann es vorkommen, dass die Disziplin zum Schreiben von *Komponententests* bei einigen Entwicklern fehlt. Oder man vernachlässigt das *Refactoring* und somit kann kein *einfaches Design* durchgeführt werden. Das kann dazu führen, dass der ganze XP-Prozess zusammenbricht. Ein XP-Trainer ist dazu da, diese Probleme aufzudecken und das

Team darauf hinzuweisen.
Der XP-Trainer sollte darauf bedacht sein zu wissen, wann er in das Geschehen eingreifen soll. Die Entwickler können keinen „Aufpasser“ leiden, der ihnen ständig im Nacken sitzt. Wenn ein XP-Trainer zu oft eingreift, kann sich das negativ auf die Motivation der Entwickler auswirken. Sie verlieren an Selbstbewusstsein und werden unsicher. Dies ist für ein XP-Projekt unpassend, da es von Mut, der aus Selbstbewusstsein gewonnen wird, lebt.
Entwicklerteams, die XP-Erfahrungen aus vorherigen XP-Projekten haben, sind so gut eingespielt, dass es unnötig ist, permanent die Rolle des XP-Trainer zu besetzen. Hier greift der XP-Trainer in Extremsituation ein, wenn sich das Team nicht mehr selbst zu helfen weiß.

12.6.6 Berater (Consultant)

Die Rolle des Beraters ist nicht fest in das XP-Team integriert. Das resultiert daraus, dass im Team Spezialisten bei technologischen Fragen fehlen. Durch *Paar-Programmierung* versucht man, jedes einzelne Teammitglied auf das gleiche Level zu bringen, d.h. durch ständiges Wechseln der Partner kann jeder sein eigenes Wissen vermitteln und erweitern. Dennoch kann es im Verlauf des Projekts vorkommen, dass das Team an seine Grenzen gerät, da das tiefgründige Wissen über das Spezialgebiet fehlt. Hier kommt der Berater ins Spiel.
Ein Berater tritt ins Spiel ein, wenn ein Problem gelöst werden soll. Es besteht keine feste Integration in das Projekt. Somit wird der Berater durch eine Person aus einem anderem hauseigenen Projekt oder einem externen Unternehmen besetzt. In XP besitzt der Berater die Charakteristiken eines Ausbilders. Das Team lässt sich durch den Berater eine Einführung in die Technologie geben. Sollten im Anschluss Fragen aufkommen, werden diese mit den unterschiedlichen Programmierer in kleinen Gruppen geklärt. Somit wird verhindert, dass im weiteren Verlauf des Projektes noch einmal auf den Berater zurückgegriffen werden muss.

12.6.7 Big Boss

Die Rolle des Big Boss wird von Kent Beck nicht eindeutig beschrieben. Zum einen soll sie dazu dienen, dem Team Mut und Zuversicht zu geben und zum anderen die organisatorische und finanzielle Seite abzudecken. Als Beispiel führt er ein, dass das Team einen Tester anfordern könnte. Wird dieser nicht besorgt, erklärt das Team, wie sich das auf den Terminplan auswirken könnte. Ist man als Big Boss mit der Antwort des Teams nicht zufrieden, kann das Team ihn auffordern, den Umfang des Projekts zu reduzieren.
Nach Kent Becks Aussagen steht der Big Boss also hinter dem Team und lenkt es. Er unterstüzt das Team während des XP-Prozesses. Die personellen und finzaniellen Ressourcen werden von ihm kontrolliert. Dennoch ist unklar von welcher Seite er agiert. Ist es die Anwender- oder Entwicklerseite? Stammt er aus der Anwenderseite, stehen die finanziellen Ressourcen im Vordergrund. Diese Person handelt nach geschäftspolitischen Überlegungen. Er versucht das Budget so gering wie möglich zu halten, dafür aber eine hohe Funktionalität in das System zu integrieren. Aus Entwicklerseite versucht er das funktionstüchtiges System termingerecht

abzuliefern. Dabei achtet er darauf, dass das Team nicht überfordert wird, da dies kontraproduktiv sein könnte.

12.6.8 Zusatzrollen

In Lippert u. a. (2002) werden drei weitere Rollen eingeführt. Diese drei Rollen resultieren daraus, dass Kent Beck keine präzise Beschreibung zu der Rolle des Big Boss formuliert. Die Autoren teilen die Rolle des Kunden in die Rolle des Auftraggebers und Anwenders auf. Der Auftraggeber ist für die Zielsetzungen und Entscheidungen im Projekt zuständig. Hingegen stellt der Anwender das Fachwissen für das Projekt bereit. Die Rolle des Kunden aufzuteilen, hat den Vorteil, dass somit eine bessere Kontrolle besteht. Zwei Personen, die zwei verschiedene Rollen ausüben, können besser koordinieren, als wenn eine Person beide Rollen übernehemen müsste. Hat man Entscheidungskraft auf beiden Seiten (Anwender/Entwickler), können alle Entscheidungen richtig abgewogen werden. Somit wäre es aus der Sicht der Autoren vorteilhafter die Rolle aufzuteilen, um Konflikte zu vermeiden. Die dritte Rolle ist der Projektverantwortliche. Er fällt die Entscheidungen über den Entwicklerprozess. Diesen drei Rollen ersetzen den Big Boss.

Auftraggeber (Client)

Die wesentliche Aufgabe des Auftraggebers ist es, die Projektziele zu definieren und die finanziellen Mitteln zur Verfügung zu stellen. Die Ziele werden aus geschäftspolitischen Überlegungen definiert. Somit stellt er auch die Zielerreichung fest.

Anwender (User)

Der Anwender stellt das anwendungsfachliche Wissen für die Programmierer zur Verfügng. Wenn Fragen zum Anwendungsbereich vorliegen, werden diese vom Anwender beantwortet. Die Autoren differenzieren aus ihrer eigenen Erfahrung das anwendungsfachliche Wissen in zwei Bereiche. Zum einem in die anwendungsfachliche Logik und zum anderem in die konkreten Aufgaben am Arbeitsplatz des Anwenders.
Bei der anwendungsfachlichen Logik handelt es sich um standardisierte Prozesse wie Lohnabrechnung oder Steuerrecht. Geschäftsprozesse, die nicht allgemein standardisierten Anteilen der anwendungsfachlichen Logik entsprechen, werden durch einen Vertreter des unteren oder mittleren Management der Anwenderorganisation besetzt.
Der zweite Bereich kann nicht aus der Literatur erschlossen werden, sondern lebt vom idealisierten Bild des Vorgesetzten (indirekt: Anwender), wie seine Mitarbeiter ihre Arbeit erledigen. Diese Aufgabenerledigung konkret als Bild zu haben ist notwendig, um ein nützliches System zu entwicklen. Damit trägt der Anwender zur Gestaltung bei.
Der Anwender definiert die Anforderungen des Systems, indem er alleine oder mit den Programmierern die Story-Cards schreibt. Er muss seinen Anwendungsbereich kennen. Zudem erstellt er die Akzeptanztests für einzelne Funktionalitäten des Systems.

Projektverantwortlicher

Der Projektverantwortliche entspricht nicht einer Rolle aus XP. Das liegt daran, dass ein XP-Team eigenverantwortlich handelt. Sie treffen die Entscheidungen im XP-Prozess selbst. Entweder man diskutiert es mit dem Team aus oder man trägt die alleinige Verantwortung über die Entscheidung. Somit ist ein Projektverantwortlicher im XP-Projekt unsinnig. Dennoch entschließen sich die Autoren aus Lippert u. a. (2002), diese Rolle ins Leben zu rufen, da es eine Reihe von Aufgaben gibt, die nicht gleichzeitig wahrgenommen werden können. Dazu gehört der gesamte Bereich der Personalplanung. Der Projektverantwortliche ist auch der primäre Ansprechpartner für seinen Vorgesetzten und seine Kunden.

12.7 Anpassung der Rollen und Techniken

In diesem Kapitel beschäftigen wir uns mit der Adaption der Rollen und Techniken an unsere Projektgruppe. Nicht alle Rollen und Techniken können ohne jegliche Anpassung übernommen werden. Manche entfallen, da es keine Verwendung für sie gibt.

12.7.1 Anpassung der Rollen

Die Rollen, die im vorherigen Kapitel vorgestellt wurden, können bis auf einige ohne Anpassung übernommen werden. Das eigentliche Problem liegt bei der Rolle des Kunden. In unserer Projektgruppe wird es keinen realen Kunden geben. Diese Rolle ist für den eigentlichen XP-Prozess sehr wichtig, da der Kunde die genauen Vorstellungen über das zukünftige System besitzt. Ohne einen richtigen Kunden könnte die Erstellung der Story-Cards im Planungsspiel nicht zu Stande kommen. Den Programmierern würde auch das Feedback des Kunden fehlen. Diese könnten nicht überprüfen, ob die Spezifikationen den Anforderungen des Kunden entsprechen.
Die Rolle des Kunden, die in Lippert u. a. (2002) beschrieben wird, kann in unserem Projekt nicht einwandfrei überommen werden. Die Autoren splitten die Rolle in zwei Rollen. Sie unterscheiden zwischen dem Auftraggeber und dem Anwender. Beide Rollen können nicht einer Person aus unserem Projekt zugeordnet werden. Die Rolle des Anwenders kann von jeden einzelnem Teammitglied übernommen werden. Als Informatiker wird man sich mit dem Thema „Debugging“ zum ersten Mal ernsthaft auseinandersetzen, wenn man an einem größeren Projekt teilnimmt. Somit wird man im Verlaufe der Entwicklung des Eclipse-Plug-In selbst auf Werkzeuge wie dem Debugmodus von Eclipse zurückgreifen. Das verschafft den nötigen Einblick in die Materie, die man als Programmierer nutzen kann. Diese können sich auch einen Einblick von der Rolle des Kunden verschaffen.
Einen wirklichen Auftraggeber wird es in unserem Projekt nicht geben. Als Stichwort sollte man das „Non-Profit-Marketing“ nennen, d.h. bei unserem Projekt steht nicht die finanzielle Seite im Vordergrund. Einen Gewinn zu erzielen ist auch nicht unser Ziel. Das Projekt entspricht mehr einen Forschungsprojekt, das vom Lehrstuhl für Software-Technologie der

Universität Dortmund geleitet wird.
Die Rolle des Big Boss wird nicht direkt in den Vordergrund geraten, da eigentlich die Projektgruppe das selbstständige Arbeiten im Team erlernen soll. Die Rolle des Big Boss kann indirekt den Betreuern oder dem Professor übertragen werden. Diese greifen nur in Extremsituationen ein, wenn das Projekt nicht voranschreitet oder sogar zum Scheitern verurteilt ist. Die Projektgruppe bekommt den Big Boss nicht zu spüren, wenn sie einen reibungslosen XP-Prozess vollzieht.
Die Rolle des Beraters kann zum Beginn des Projektes nicht bestimmt werden. Diese ergibt sich im XP-Prozess. Fehlt dem Team das Spezialwissen über eine Technologie, muss diese entweder von einigen Teammitgliedern erlernt werden oder über die Projektgrupe hinaus müssen Personen gefunden werden, die über dieses Spezialwissen verfügen.

12.7.2 Anpassung der Techniken

Aus dem vorherigen Abschnitt geht hervor, dass die Techniken „Planungsspiel“ und „Kunde vor Ort“ angepasst werden müssen. Beim Planungsspiel müssen die Storycards von den PG-Mitgliedern erstellt werden. „Der Kunde vor Ort“ sollte abwechselnd von verschiedenen PG-Mitgliedern gespielt werden.
„Programmierstandards“ sind wichtig und sollten eingeführt werden. Um Ärger im Vorfeld zu vermeiden, sollte man zu einem bekannten Programmierstandard greifen. Die Firma Sun bietet einen solchen.
Die „40-Stunden-Woche“ ist in unserem Projekt nicht realisierbar, denn es handelt sich um eine Lehrveranstaltung, die über zwei Semester geht. Im Idealfall sind 20 Semesterstunden pro Woche angesetzt. Diese Zahl kann je nach Aufwand und den Fähigkeiten eines jeden einzelnen Mitglied der Projektgruppe variieren.
Die restlichen Techniken können ohne Anpassung übernommen werden. Wenn sich im Verlauf des Projektes herausstellt, dass eine Technik nicht nach der Vorgabe genutzt werden kann, wird man eine Adaption dieser Technik vornehmen müssen.

TEIL 3

Releasebeschreibungen

KAPITEL 13

Systemmetapher

Carina Klar, Antonio Pedicillo

Das Ziel der Projektgruppe ist die Erstellung eines PlugIns für die Entwicklungsumgebung Eclipse, das es ermöglicht, Laufzeitinformationen in Form eines Films zu präsentieren, den sich der Benutzer ansehen und in dessen Ablauf er nach Wunsch eingreifen kann. Dies soll in einer 3D-Ansicht geschehen. Die darzustellenden Objekte aus den Laufzeitinformationen werden dabei durch geometrische Formen repräsentiert, beispielsweise „normale“ Klassen als Kugeln oder Würfel. Einzelne Objekte, die in wie auch immer gearteter Verbindung zueinander stehen, werden dann z.B. durch einen Pfeil oder eine Linie miteinander verbunden. So eine graphische Darstellung soll vom Benutzer aus verschiedenen Blickwinkeln betrachtet werden können. Weiterhin soll es möglich sein, die Veränderungen der Laufzeitinformationen zwischen zwei beliebigen Zeitpunkten nachzuvollziehen. Zu diesem Zweck sollen Objekte und Verbindungen innnerhalb der 3D-Ansicht umherwandern, ausgeblendet oder eingeblendet weren. Außerdem soll herausgefunden werden, inwieweit diese Art der Darstellung von Quellcode das Debuggen eines Programms erleichtert. Hieraus ergeben sich konzeptionelle und technische Fragen:

Auf der konzeptionellen Seite müssen die anzuzeigenden Laufzeitinformationen und deren dreidimensionale Darstellung ausgewählt werden, die dem Benutzer ein sinnvolles Debugging ermöglichen. Daneben müssen sich beim Design Gedanken dazu gemacht werden, welche Interaktionen für den Benutzer aller Voraussicht nach sinnvoll sind.

Auf der technischen Seite muss die Frage geklärt werden, wie die Laufzeitinformationen in Eclipse beschafft werden können. Außerdem muss die 3D-Darstellung realisiert und in Eclipse integriert werden. Schließlich muss die 3D-Sicht mit den Laufzeitinformationen gekoppelt werden, um dadurch die aktuellen Informationen darstellen zu können.

KAPITEL 14

Beschreibung des ersten Release

14.1 Einleitung

Dennis Weyland

Mit diesem Release sollen grundlegende Techniken erlernt und ihre Verwendung eingeübt werden. Insbesondere steht dabei die Auseinandersetzung mit weitgehend unbekannten Technologien wie Java-3D, Debugging in Java und die Eclipse-Plug-In-Struktur im Vordergrund.

Als Ziel des ersten Releases steht die dreidimensionale Darstellung aller instanziierten Objekte eines im Java-Quelltext vorliegenden Programmes zum Ende der „main-Methode“. Vorgaben zur Art der Visualisierung gibt es dabei nicht. Eine Interaktion mit dem Anwender ist auch nicht Bestandteil der Anforderungen.

Diese relativ groben Anforderungen wurden mit der Ausgabe der User-Stories durch die Kunden präzisiert bzw. erweitert. Auch die einzelnen Bestandteile des Releases werden im Folgenden näher beschrieben.

14.2 User-Stories

Carina Klar, Antonio Pedicillo

Das Ziel des ersten Release ist die Erstellung eines Eclipse-Plug-In zur Visualisierung von Laufzeitiformationen eines Java-Programms. Die Grundlage für die Umsetzung dieses Ziels sind die von den Kunden formulierten User-Stories. In diesem Abschnitt werden die User-Stories nach ihrer Wichtigkeit geordnet einzeln vorgestellt.

Snapshot-Anzeige an einem Breakpoint
Beschreibung: „Bei Erreichen einer bestimmten Programmzeile (vorher festgelegt) soll die

Anzeige der aktuelle Objekte ausgegeben werden können.“ (Kategorie ***must-be*** - *Story 1*)
Die wichtigste Story beinhaltet die Umsetzung des Hauptziels dieses Releases. Bei Erreichen einer bestimmten Programmzeile (nicht mehr nur wie oben beschrieben am Ende der „main-Methode“) sollen die aktuell existierenden Objekte graphisch dreidimensional angezeigt werden können. Objekte, die bis zu diesem Punkt des Programms erzeugt, aber auch wieder zerstört worden sind, sollen nicht dargestellt werden. Die zu betrachtende Programmzeile soll durch einen Breakpoint im Sinne des herkömmlichen Debuggens in Eclipse ausgewählt werden können. Die Debug-Informationen, die ohnehin von Eclipse geliefert werden, sollen weiterhin verfügbar sein. Das bedeutet gleichzeitig, dass die bisher von Eclipse bereitgestellten Breakpoints weiterhin gesetzt werden können.

Objekt-Visualisierung durch Kugeln
Beschreibung: „Objekte sollen als Kugeln visualisiert werden.“ (Kategorie ***must-be*** - *Story 2*)
Alle anzuzeigenden Objekte sollen als Kugeln dargestellt werden. Dafür soll eine spezielle Sicht erstellt werden, in der der Benutzer das 3D-Debugging durchführt.

Textuelle Darstellung von Debug-Informationen
Beschreibung: „Die Debug-Information soll gleichzeitig als Text dargestellt werden. Wichtig ist der innere Zustand.“ (Kategorie ***must-be*** - *Story 3*)
Die Werte der Objektattribute mit primitiven Datentypen und Strings sollen in einem separaten Fenster angezeigt werden. Dazu muss eine dargestellte Kugel selektiert werden können, woraufhin die Attributwerte in dem speziellen Fenster erscheinen.

Auswahl zu beobachtender Klassen
Beschreibung: „Es sollen Klassen festgelegt werden können, deren Objekte visualisiert werden (Einschränkung).“ (Kategorie ***costly-to-lose*** - *Story 1*)
Damit die Darstellung der Laufzeitinformationen auch bei größeren Programmen übersichtlich bleibt und der Fokus auf spezielle Klassen gesetzt werden kann, soll der Benutzer vor dem Debugging in einer Baumstruktur Klassen selektieren können, deren Objekte dargestellt werden.

Anzeige von Attributwerten
Beschreibung: „In der Visualisierung sollen die Werte der Attribute mit primitiven Datentypen (und String) angezeigt werden.“ (Kategorie ***costly-to-lose*** - *Story 2*)

Erkennbarkeit von Objekten derselben Klasse
Beschreibung: „Objekte derselben Klasse sollen dieselbe Farbe haben.“ (Kategorie ***costly-to-lose*** - *Story 3*)

Um auch bei vielen Objekten den Überblick behalten zu können und die Zuordnung von Objekten zu Klassen einfacher zu gestalten, erhalten alle Objekte eines bestimmten Typ dieselbe eindeutig festgelegte Farbe.

Timerfunktion für Snapshot-Ausgabe
Beschreibung: „Nach bestimmter, vorher festgelegter Zeit soll die Anzeige der Objekte ausgegeben werden.“ (Kategorie ***costly-to-lose*** - *Story 4*)
Zusätzlich zur Auswahl der Programmzeile durch einen Breakpoint soll die Ausgabe der Objekte nach einer vorher festgelegten Zeit erfolgen können. Dies kann zum Beispiel bei einem Programm, das eine graphische Oberfläche beinhaltet, sehr nützlich sein.

Einbindung der Online-Hilfe
Beschreibung: „Hilfedatei in die vorhandene Hilfefunktion von Eclipse einbinden.“ (Kategorie ***nice-to-have*** - *Story 1*)
Dem Benutzer soll eine Hilfedatei für dieses Plug-In zur Verfügung stehen. Diese soll in die Hilfe von Eclipse integriert werden.

Einfache Installation
Beschreibung: „Die Software soll einfach zu installieren sein.“ (Kategorie ***nice-to-have*** - *Story 2*)
Es wird eine benutzerfreundliche Installation gewünscht, z.B. über die von Eclipse bereitgestellten Mechanismen.

Tool-Tips
Beschreibung: „Wird der Cursor über ein Element des Fensters gelegt, soll eine kurze Erklärung erscheinen.“ (Kategorie ***nice-to-have*** - *Story 3*)
Wird der Cursor über bestimmte Elemente der 3D-Debug-Sicht geführt, sollen kurze Erläuterungen (Tooltips) erscheinen.

14.3 Reflexion über die Tasks

Henning Zeller

Im Folgenden wird über die Tasks des ersten Release reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die signifikanten Probleme und die Differenzen zwischen Zeitabschätzungen und tatsächlich benötigter Zeit eingegangen. Außerdem werden die Tasks aus Gründen der übersichtlichkeit in drei wesentliche Kategorien eingeteilt.

14.3.1 Aufbau der grundlegenden Infrastruktur

Einrichtung der Debug-Pakete

Beschreibung: Es sollen die benötigten Pakete der JDT-API (org.eclipse.jdt.debug, org.eclipse.jdt.launching, etc.) eingerichtet werden.

geplante Zeit: Veranschlagt wurde hierfür 1 Tag sowie ein halber Tag, um eine entsprechende Schulung durchzuführen.

reale Zeit: Der Task erwies sich als komplexer als erwartet. Benötigt wurden 2 Tage, die angedachte Schulung entfiel.

Integrieren des Launchers

Beschreibung: Der Launcher soll die Interaktion zwischen dem Benutzer und Eclipse ermöglichen. Wird ein entsprechender Button in der Toolbar angeklickt, wird zum einen die Datenstruktur angestoßen, die die statischen Informationen des Projekts, das debugged werden soll, aufnimmt und zum anderen die 3D-Debug-Perspektive initialisiert.

geplante Zeit: 2 Tage

reale Zeit: 3 Tage wurden investiert, bis man sich schließlich entschied, auf einen bereits vorhandenen Launcher zurckzugreifen.

14.3.2 Konzeptionelle Tasks

Schreiben der Hilfe

Beschreibung: Verfassen einer Hilfe zur Plug-In-Funktionalität

geplante Zeit: 2 Tage

reale Zeit: Mit der Umsetzung dieses Tasks wurde sehr spät begonnen, so dass nur ein geringer Umfang der Hilfe, realisiert in einem halben Tag, zustande kam.

Einfache Installation des Plug-Ins

Beschreibung: Hier spielt auch die Berücksichtigung verschiedener Plattformen (Linux, Win32, etc.) eine Rolle.

geplante Zeit: 1 Tag

reale Zeit: 2 Tage; Es gab zeitweise Probleme bei der Umstellung von Java 1.4 auf Java 1.5.

Einbinden eines Hilfe-Plug-Ins

Beschreibung: Die Hilfe zum entwickelten Plug-In sollte in die bereits vorhandene Eclipse-Hilfe integriert werden.
geplante Zeit: 1 Tag
reale Zeit: Es traten keine Probleme auf, benötigt wurde etwa ein halber Tag.

3D-View ins Plug-In integrieren

Beschreibung: Eine View zur Darstellung der 3D-Visualisierung muss implementiert und in eine Perspektive integriert werden.
geplante Zeit: 1.5 Tage
reale Zeit: Das Neuzeichnen bei Änderung der Fenstergröße funktionierte anfangs nicht, außerdem wurde bei geteilter Geometrie der Java3D Objekte das Universum nicht gelöscht. Im Endeffekt wurden deshalb 3 Tage benötigt.

Klassenauswahl-View integrieren

Beschreibung: Integration eines Views, der es dem Benutzer ermöglicht, diejenigen Klassen auszuwählen, deren Laufzeitinformationen visualisiert werden sollen.
geplante Zeit: 1 Tag
reale Zeit: 3 Tage; Der Task wurde ursprünglich zu oberflächlich formuliert; es ergaben sich dann im Rahmen der Implementierung weitere zu realisierende Anforderungen, die während des Planspiels nicht berücksichtigt worden waren.

PropertyView zur Anzeige der Objektattribute entwickeln

Beschreibung: Für Objekte, die im Java3D-View angeklickt werden, sollen im PropertyView entsprechende Attribute angezeigt werden.
geplante Zeit: 1 Tag
reale Zeit: 5 Tage; es gab unter anderem erhebliche Probleme beim Layout des PropertyViews (Tabelle).

Anzeige von Tooltips

Beschreibung: Realisierung von Tooltips, um die Benutzerführung zu vereinfachen.
geplante Zeit: 1 Tag
reale Zeit: 2 Tage; Es wurden weniger Tooltips als geplant realisiert, da sie sich an vielen Stellen entweder als unnötig oder in der Realisierung zu aufwändig erwiesen.

Inhalte der Tooltips

Beschreibung: siehe oben
geplante Zeit: 1 Tag
reale Zeit: Aufgrund der geringen Anzahl von Tooltips wurde weniger als ein halber Tag benötigt.

Automatischer Breakpoint nach bestimmter Zeit

Beschreibung: Nach einer bestimmten Zeit soll automatisch ein Breakpoint gesetzt werden und die zu diesem Zeitpunkt im Laufzeitsystem befindlichen Objekte erfasst und visualisiert werden.
geplante Zeit: 1 Tag
reale Zeit: 1 Tag; Es gab keine nennenswerten Probleme.

Integration von Breakpoints ins Plug-In

Beschreibung: Es sollen nutzerdefinierte Breakpoints genutzt werden können.
geplante Zeit: 2 Tage
reale Zeit: 3 Tage; Aufgrund der mangelnden Dokumentation von JDT fanden wir zunächst keinen Ansatzpunkt, an dem eigene Breakpoints implementiert oder die vorhandenen um zusätzliche Funktionen erweitert werden können. Letztendlich wurde ohne Veränderung auf die Eclipse-eigenen Breakpoints zurückgriffen.

14.3.3 Datenhaltung und -beschaffung

Beschaffung der Laufzeitdaten

Beschreibung: Für die vom Benutzer für den Snapshot ausgewählten Klassen müssen die Laufzeitinformationen beschafft werden.
geplante Zeit: 3 Tage
reale Zeit: 5 Tage; Die Einarbeitung erwies sich als schwierig, die eigentliche Datenbeschaffung konnte dann recht einfach bewerkstelligt werden.

Datenstruktur zur Repräsentation der Laufzeit-Informationen

Beschreibung: Die Laufzeitinformationen werden in einer entsprechenden Datenstruktur gespeichert, um später visualisiert werden zu können.
geplante Zeit: 2 Tage
reale Zeit: 2 Tage; Es gab keine größeren Probleme.

Beschaffung der ausgewählten Klassen

Beschreibung: Die vom Benutzer für den Snapshot ausgewählten Klassen müssen beschafft werden.
geplante Zeit: 3 Tage
reale Zeit: 2 Tage; Keine Probleme

Datenstruktur für die ausgewählten Klassen

Beschreibung: Die für den Snapshot ausgewählten Klassen sollen in einer entsprechenden Datenstruktur repräsentiert werden.
geplante Zeit: 1 Tag
reale Zeit: 1 Tag; Es gab keine größeren Probleme. Allerdings stellte sich heraus, dass die Datenstruktur teilweise nicht benötigt wurde. So wurde beim Filtern ein Mechanismus verwendet, der nicht die komplette Datenstruktur benutzte, sondern sich viele Informationen selbst besorgte.

Beschaffung der Attribute der ausgewählten Klassen

Beschreibung: Attribute sollen ebenfalls dargestellt werden.
geplante Zeit: 2 Tage
reale Zeit: 0.5 Tage; Keinerlei Probleme

Erweiterung der Datenstruktur um Attribute

Beschreibung:	Dieser Task ergibt sich unmittelbar aus den Anforderungen des vorangegangenen Tasks.
geplante Zeit:	1 Tag
reale Zeit:	0.5 Tage; Keine Probleme

14.3.4 Visualisierung

3D-Objekte entwerfen und erzeugen

Beschreibung: Die Laufzeitinformationen sollen (in Form von Kugeln) visualisiert werden.
geplante Zeit: 2.5 Tage
reale Zeit: 3 Tage; Probleme: mangelnde Erfahrung mit Java3D.

3D-Objekte anordnen, grafische Datenstruktur

Beschreibung: Die visualisierten Objekte sollen angeordnet werden (dies geschieht in Form eines würfelförmigen Gittermodelles); hierzu wird eine entsprechende Datenstruktur benötigt.
geplante Zeit: 2 Tage
reale Zeit: 2 Tage; Keine größeren Probleme

Graphische Elemente in 3D-View integrieren

Beschreibung: Die visualisierten Objekte sollen in einem 3D-View dargestellt werden.
geplante Zeit: 1 Tag
reale Zeit: 0.5 Tage; Keine Probleme

Realisierung eines Picking-Behaviors zur Ausgabe von textuellen Informationen (Objektattribute, etc.)

Beschreibung: Sobald der Benutzer ein Objekt im 3D-View anklickt, sollen Informationen im PropertyView angezeigt werden.
geplante Zeit: 2 Tage
reale Zeit: 3 Tage; Beim Raytracing wurde zunächst die Reihenfolge der Betrachtung vertauscht, so dass einige (zumeist verdeckte) Kugeln ignoriert wurden.

Gleiche Farbe für Objekte derselben Klasse

Beschreibung: Zur selben Klasse gehörige Objekte sollen in derselben Farbe dargestellt werden.
geplante Zeit: 3 Tage
reale Zeit: 2 Tage; Keine Probleme, es mussten allerdings einige Änderungen an der grafischen Datenstruktur vorgenommen werden.

14.4 Vorstellung der Architektur

Daniel Maliga, Jonas Mathis, Michael Striewe

Im Nachfolgenden wird die in diesem Release implementierte Architektur vorgestellt. Dazu wird zunächst die geplante Architektur beschrieben, dann die realisierte Architektur festgehalten und abschließend beide verglichen.

14.4.1 Beschreibung der geplanten Architektur

Für die gestellte Aufgabe bietet sich die Verwendung des Model-View-Control-Konzeptes (MVC) an, das insbesondere bei graphischen Benutzeroberflächen Verwendung findet. Bei diesem Konzept werden die zugrundeliegende Datenstruktur (*Model*) und die Darstellung der Daten (*View*) explizit voneinander getrennt und von spezialisierten Programmteilen übernommen. Dabei sind auch unterschiedliche Darstellungen zur selben Zeit möglich. Zwischen diesen beiden Teilen liegt der *Control*, der Koordinierungsaufgaben sowie die Kernfunktionen des Programms beinhaltet. Veränderungen im *Model* werden dabei ähnlich wie beim Entwurfsmuster des *Beobachters* (Gamma u. a. (1997)) an die *Views* weitergegeben.

Auf den vorliegenden Fall lässt sich dies wie folgt übertragen: die von Eclipse bzw. dem Eclipse-Debug-Plug-In bereitgestellten Informationen sollen durch einen `Collector`, der zusammen mit Filter- und Kontrollklassen den *Controller*-Teil des MVC-Modells bildet, abgefragt und im `Model` gespeichert werden. Die Objekte der ausgewählten Klassen sollen dann in einer dreidimensionalen Ansicht (`Java3DView`) dargestellt werden. Die Attribute eines selektierten Objekts sollen innerhalb der `Java3DView` als Text eingeblendet werden.

Das Auslösen der Datensammlung durch den `Collector` übernimmt ein auf selbst definierte Snapshot-Breakpoints reagierender `Trigger`. Aus den gesammelten Daten werden über die Klasse `Filter` die relevanten Informationen ausgewählt und danach im `DataStore` gespeichert, welches als zentrales `Repository` dient. Die Datenübergabe soll im Sinn einer *Pipes-and-Filters*-Architektur umgesetzt werden.

Der `Core` stellt zentrale Hilfsfunktionalitäten bereit und nimmt die Anbindung des Plug-Ins an Eclipse vor. Die Online-Hilfe für das Plug-In wird durch ein separates Plug-In realisiert. Zur späteren Lokalisierung ist die Erstellung von Fragmenten vorgesehen.

Das `Model` ist in zwei Teile gegliedert, von denen einer die statische Klassenstruktur des zu debuggenden Programms repräsentiert und der andere dessen dynamische Laufzeitinformation. Innerhalb der statischen Klassenstruktur wird dabei gespeichert, ob die zugehörigen Laufzeitobjekte bei der Datensammlung berücksichtigt werden sollen. Auf diese Weise wird eine Vorfilterung der darzustellenden Information ermöglicht.

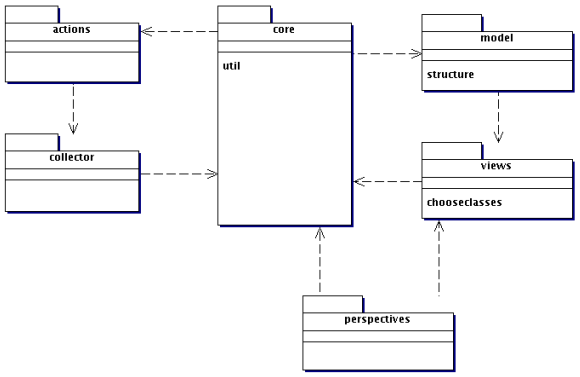


Abbildung 14.1.: Paketstruktur der realisierten Systemarchitektur (Release 1)

14.4.2 Beschreibung der realisierten Architektur

Abb. 14.1 zeigt die Umsetzung der Planung in der Paketstruktur. Das MVC-Modell findet seine Entsprechung in den Paketen `Model`, `Views` und dem Zusammenspiel aus `Core`, `Actions` und `Collector`. Die Filter-Funktion ist dabei im Paket `Collector` integriert, während die im `Core` liegende Klasse `DataStore` die Funktion des Repository übernimmt. Das zusätzliche Paket `Perspectives` initialisiert die Views und das Repository.

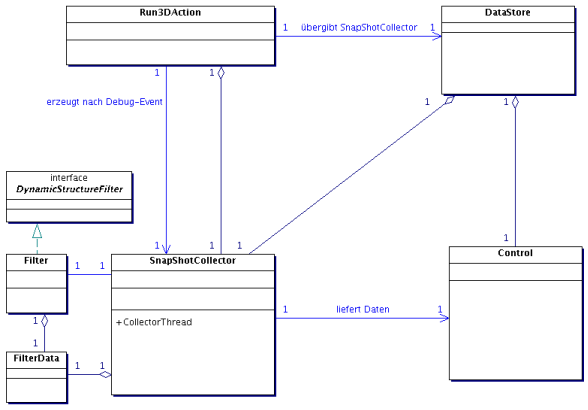


Abbildung 14.2.: Klassenstruktur des Control-Teils (Release 1)

Die wichtigsten Klassen des Control-Teils sind in Abb. 14.2 dargestellt. Dieser Teil ist für die Erzeugung eines DebugSnapshots zuständig. Ausgangspunkt ist die Klasse `Run3DAction`, die zunächst eine Instanz der Klasse `SnapShotCollector` erzeugt und zur späteren Weiterverwendung der statischen Klasse `DataStore` übergibt. Der `SnapShotCollector` sammelt dann nach jedem Debug-Event (Breakpoint oder Timer) über die Eclipse-eigenen

Methoden Daten aus der Virtual Machine und nutzt die Klasse `Filter`, um die vom Nutzer gewünschten Daten auszuwählen. Danach übergibt er den erzeugten Snapshot an die Klasse `Control`, die ihn unter anderem im `DataStore` speichert.

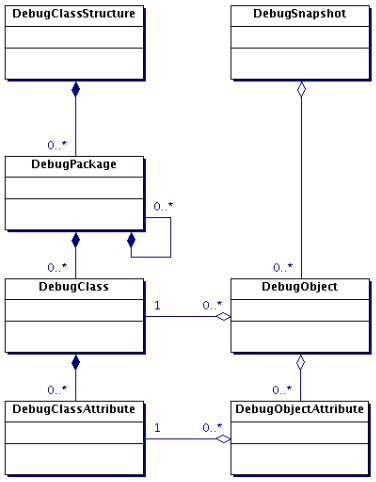


Abbildung 14.3.: Klassenstruktur des Model-Teils (Release 1)

Der Aufbau eines Snapshots ist aus den Details des Pakets `Model` in Abb. 14.3 ersichtlich. Die eingesetzte Datenstruktur ist in der Lage, auf Quelltextebene Pakete, Klassen sowie Attribute und auf Objektebene die einzelnen Objekte sowie deren Attribute mit ihren konkreten Werten abzubilden. Die linke Hälfte des Diagramms stellt die statischen Aspekte des zu debuggenden Programms dar, berücksichtigt dabei aber stets nur tatsächlich genutzte Klassen. Die rechte Hälfte bildet den dynamischen Zustand des Programms zum Zeitpunkt des Snapshots ab. Für jedes zu diesem Zeitpunkt existierende und zu beobachtende Objekt wird eine Instanz der Klasse `DebugObject` erzeugt, mit Attributen ausgestattet und dem Snapshot hinzugefügt.

Die drei einzelnen Teile des Pakets `View` sind in Abb. 14.4 dargestellt. Im oberen Teil befindet sich die Klassenstruktur für die `ChooseClassesView`. Für die einzelnen Komponenten des zu debuggenden Programms stehen Klassen zu deren Darstellung in einem Auswahlbaum bereit. Diese beinhalten jeweils Kontrollelemente, um dem Nutzer die Auswahl von Teilbäumen zu ermöglichen.

Links unten ist der Aufbau der 3D-Ansicht dargestellt. Die Klasse `Java3DView` repräsentiert die Hauptsicht auf die erzeugte 3D-Visualisierung. Sie basiert auf der Java3D-Technologie. Dementsprechend bilden die dazugehörigen Klassen Java3D-Funktionalitäten ab: Das `GraphicalObject` kapselt die Debug-Informationen eines `DebugObject` und repräsentiert diese als primitives 3D-Objekt (farbige Kugel). Das `PickBehavior` reagiert für jede Kugel auf Mausklicks und führt dazu, dass in der separaten `PropertyView` Informationen (Name, Attribute) zum angeklickten Objekt angezeigt werden. Die Verknüpfung zwischen den beiden Views liegt außerhalb des Pakets in der Klasse `Control`, die das `PickListener`-Interface implementiert.

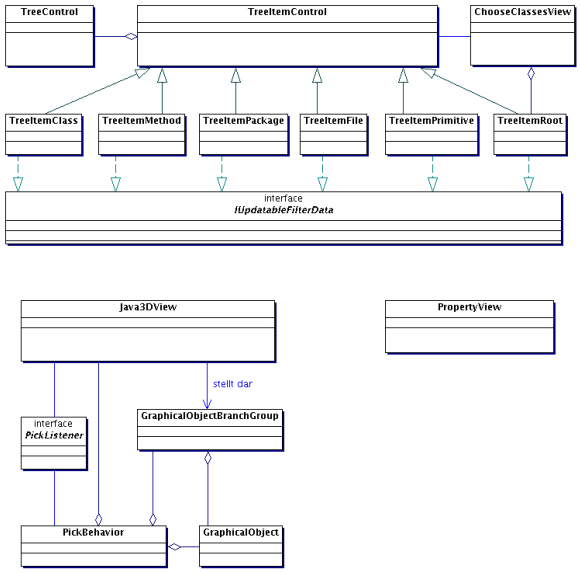


Abbildung 14.4.: Klassenstruktur des View-Teils (Release 1)

Den Gesamtzusammenhang der Klassen zeigt Abb. 14.5. Die Klasse `PG458Plugin` hat keine direkten Beziehungen zu anderen Klassen, da sie nur für Eclipse-interne Zwecke benötigt wird und die Ressourcen festlegt. Zentralen Zugriff auf die Views und auf den `DebugSnapShot` bietet die `Control`-Klasse aus dem Paket `Core`. Insbesondere werden im `DataStore` Verweise auf die Views sowie auf den `DebugSnapShot` und die Datenstruktur gespeichert.

Datenübergabe zwischen den Paketen findet an vier Stellen statt. Nach dem Aufruf der `Run3DAction` wird die vom Nutzer in der `ChooseClassesView` getroffene Auswahl ausgelesen und in die HashMap von `FilterData` übertragen, welche vom `SnapShotCollector` beim Auslesen der Debug-Daten berücksichtigt wird. Der `SnapShotCollector` schreibt die gewonnenen Daten in die dynamische Datenstruktur aus dem Paket `Model` und übergibt den fertigen Snapshot der Klasse `Control`. Diese reicht ihn sowohl ans `DataStore` zur Speicherung als auch an die `Java3DView` zur Darstellung weiter.

Weiterhin stellt das Paket `Views` ein Interface zur Verfügung, welches die Behandlung von Mausklicks auf der `Java3DView` ermöglicht und von der Klasse `Control` implementiert wird. Diese liest dann das `DebugObject` eines angeklickten `GraphicalObjects` aus und gibt es an die `PropertyView` weiter, die dann die Attribute des gewählten Objekts anzeigt.

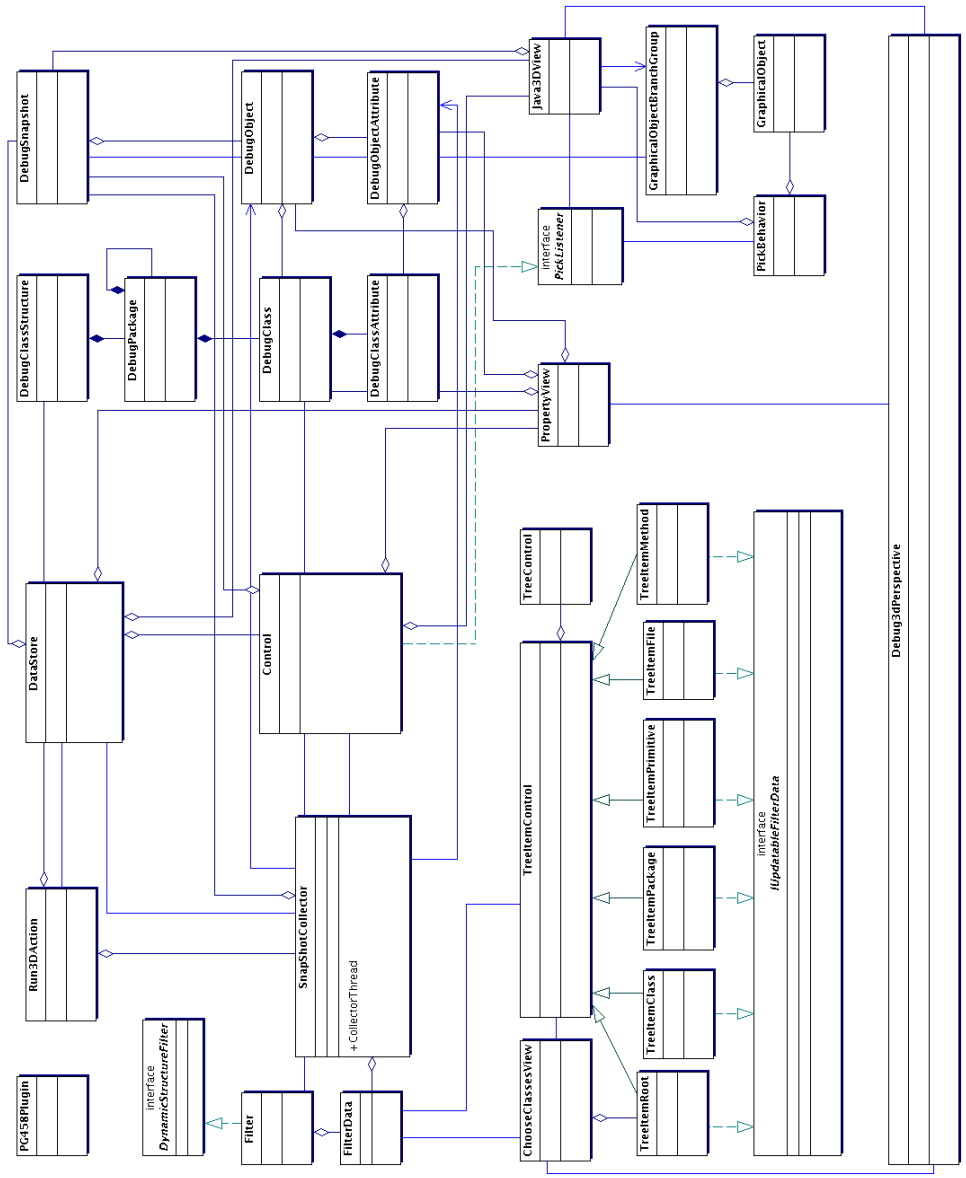


Abbildung 14.5.: Realisierte Systemarchitektur Release 1

14.4.3 Vergleich von geplanter und realisierter Architektur

Entgegen der Planung wird die statische Datenstruktur nicht als erstes aufgebaut und dann als Grundlage für die Filterung der Daten im `SnapShotCollector` genutzt, sondern sie entsteht erst später parallel mit der Erzeugung des Snapshots, während die separaten Filterdaten (`FilterData`) bereits vorher erzeugt werden. Desweiteren wird keine *Pipes-and-Filters*-Architektur genutzt, sondern die Klasse `SnapShotCollector` ist zentrale Instanz für den Aufbau eines Snapshots, indem sie Daten sammelt, zur Filterung gibt, aus den gefilterten Daten den Snapshot erstellt und diesen weitergibt.

Die geplante Trennung von `Collector` und `Trigger` erwies sich als nicht notwendig, da der `SnapShotCollector` bereits selber als Listener auf Debug-Events implementiert werden konnte. Ebenso entfiel auch die Notwendigkeit für selbst definierte Breakpoints, da das Plug-In bereits auf die normalen Breakpoints reagiert.

Die letztendlich implementierte Bereitstellung einer View zur Auswahl der zu beobachtenden Klassen wurde in der ursprüglichen Planung nicht berücksichtigt. Da sich eine Einschränkung der darzustellenden Klassen aber als dringend notwendig erwies, um unnötigen Ressourcenverbrauch zu vermeiden und die Übersichtlichkeit des Snapshots zu erhöhen, wurde diese Funktionalität durch die `ChooseClassesView` realisiert.

Die Kontrollfunktionalität im Paket `Core` übernimmt wie geplant die Klasse `Control`. Die Rolle des *Repository* wird durch die Klasse `DataStore` realisiert, in der auch die `Views` angemeldet sind und die auch Rückgabemethoden für die verwalteten Objekte enthält. Ebenfalls entsprechend der Planung wird die `Java3DView` mit ihren Objekten aufgebaut. Die Einführung der zusätzlichen `PropertyView` zur Anzeige der Objektattribute ergab sich aus technischen Schwierigkeiten mit der Anzeige von Textinformationen innerhalb der `Java3DView`. Außerdem entspricht diese Trennung eher der bei Eclipse üblichen Aufteilung der Benutzeroberfläche.

14.5 Akzeptanztests

Carina Klar

In der Projektgruppensitzung am 06.12.2004 wurden die Akzeptanztests des ersten Release durchgeführt. Das bedeutet, dass die Kunden anhand der von ihnen zu Beginn aufgestellten User-Stories das von den Entwicklern ausgelieferte Programm auf deren korrekte Umsetzung überprüfen. Dazu erstellten die Kunden Varianten von Programmdurchläufen, die die verschiedenen aus den User-Stories hervorgehenden Anforderungen an das Programm abdecken. Hauptsächlich diente das Programm „Dave“ als zu debuggendes Programm. Um spezielle Tests durchführen zu können, erstellten die Kunden aber auch eigene kleinere Programme. Im Nachfolgenden werden die verschiedenen Tests und deren Ergebnisse aufgeführt.

Das Plug-In soll unter Linux ohne Anleitung installiert werden können.
Testergebnis: nachträglich bestanden
Anmerkung: Der Benutzer wird durch geeignete Hinweise in den Fenstern durch die Installation geleitet.

Das Plug-In soll unter Windows ohne Anleitung installiert werden können.
Testergebnis: nachträglich bestanden
Anmerkung: Der Benutzer wird durch geeignete Hinweise in den Fenstern durch die Installation geleitet.

Allgemeiner Programmdurchlauf: Eclipse wird gestartet. – Das zu debuggende Programm wird geöffnet. – Es werden mehrere Breakpoints gesetzt. – In der Auswahl der anzuzeigenden Klassen wird ein ganzes Package ausgewählt. – Der 3D-Debug-Button wird betätigt.
Testergebnis: bestanden
Anmerkung: Dieser Test prüft die Grundfunktionalität der Software und stellt die Ausgangsbasis für die weiteren Tests dar. Ein erfolgloser Test an dieser Stelle würde auch alle folgenden Tests hinfällig werden lassen.

Es soll möglich sein, ganze Hierarchien von Klassen auszuwählen.
Testergebnis: bestanden
Anmerkung: Die Klassen werden baumartig aufgelistet und eine Klasse kann durch Setzen eines Häkchens selektiert werden. Automatisch werden dadurch alle untergeordneten Zweige selektiert.

Die zum Zeitpunkt des zuletzt erreichten Breakpoints existierenden Objekte sollen so als Kugeln dargestellt werden, dass alle für den Benutzer sichtbar sind.
Testergebnis: bestanden
Anmerkung: Die Kugeln werden auf den Außenflächen eines Würfels angeordnet. Die Sichtbarkeit aller Kugeln ist durch die Möglichkeit des Zoomens und Drehens der Kugeln gegeben.

Objekte gleicher Klassen sollen die gleiche Farbe haben. Objekte unterschiedlicher Klassen sollen nicht die gleiche Farbe haben.
Testergebnis: bestanden
Anmerkung: keine

Ein Mausklick auf eine bestimmte Kugel soll die Objektinformationen in der PropertyView anzeigen (zugehörige Klasse, Werte innerer Variablen).
Testergebnis: bestanden
Anmerkung: In einem separaten Fenster werden diese Informationen in einer Tabelle aufgelistet. Diese aktualisiert sich nach jeder neuen Auswahl einer Kugel.

Die selektierte Kugel soll gut zu erkennen sein.
Testergebnis: bestanden
Anmerkung: Die selektierte Kugel wird heller.

Ein Mausklick ins „Leere“ und nicht auf eine Kugel soll die Objekteigenschaften der vorher selektierten Kugel löschen.
Testergebnis: bestanden
Anmerkung: In der Tabelle der Objekteigenschaften wird die Information “Kein Objekt ausgewählt“angezeigt.

Im Programmcode wird ein neuer Breakpoint gesetzt. Die 3D-Ansicht soll sich aktualisieren.
Testergebnis: nachträglich bestanden
Anmerkung: Die 3D-Ansicht aktualisierte sich anfangs nicht. Wenn zum Zeitpunkt des neuen Breakpoints keine Objekte existierten, hätte die 3D-Ansicht entsprechend leer sein müssen.

Die klassische vorher in Eclipse verfügbare Debug-Perspektive zur Informationsanzeige soll weiterhin verfügbar sein.
Testergebnis: bestanden
Anmerkung: Das Plug-In bietet eine eigene Perspektive. Durch Umschalten der Perspektive kann die ursprüngliche Debugperspektive von Eclipse weiterhin verwendet werden.

Wird in der Auswahl der anzuzeigenden Klassen keine Klasse ausgewählt, so soll die 3D-Ansicht leer bleiben.
Testergebnis: bestanden
Anmerkung: keine

Der zu debuggende Sourcecode beinhaltet unter anderem zwei Packages, in denen je zwei Klassen mit gleichem Namen existieren. Beide Klassen werden als anzuzeigend ausgewählt. Die entsprechenden Kugeln der Objekte, die zu diesen Klassen gehören, sollen dieselbe Farbe haben.
Testergebnis: bestanden
Anmerkung: keine

Der zu debuggende Sourcecode erzeugt und zerstört ein Objekt bevor der gesetzte Breakpoint erreicht wird. Dieses Objekt wird in der 3D-View nicht als Kugel visualisiert.
Testergebnis: bestanden
Anmerkung: keine

Eine Hilfe zum Plug-In ist in der Eclipse-Umgebung integriert.
Testergebnis: bestanden
Anmerkung: Der Rahmen für die Hilfe ist vorhanden. Eine geeignete Benutzerhilfe wird in der weiteren Entwicklung des Programms ergänzt.

Die Maus wird über einige Elemente der 3D-Debug-Umgebung geführt. An geeigneten Stellen erscheinen Tooltips.
Testergebnis: bestanden
Anmerkung: Die Tooltips im Fenster zur Auswahl der anzuzeigenden Klassen erscheinen, falls die Klassennamen nicht vollständig in das Fenster passen und zeigen dann den Namen an.

Für das Plug-In gibt es die Möglichkeit eines einfachen Webupdates.
Testergebnis: nachträglich bestanden
Anmerkung: keine

Statt den Zeitpunkt für die Objektanzeige mit Breakpoints anzugeben, kann auch eine bestimmte Zeit festegelegt werden, zu der die aktuellen Objekte betrachtet werden sollen.
Testergebnis: bestanden
Anmerkung: Wird der 3D-Debug-Button betätigt, erscheint ein Fenster, in dem die Zeit des Timers in Millisekunden festgelegt werden kann. Um den Timer nicht zu benutzen, sondern eine Programmzeile durch einen Breakpoint zu bestimmen, muss der Wert 0 eingetragen werden.

KAPITEL 15

Beschreibung des zweiten Release

15.1 Einleitung

Henning Zeller

Im Rahmen des ersten Releases geschah die Visualisierung der im Laufzeitsystem befindlichen Objekte relativ willkürlich und Beziehungen zwischen diesen wurden bis auf die Klassenzugehörigkeit nicht weiter berücksichtigt. Daher war das Hauptanliegen des zweiten Release, eine sinnvolle Anordnung der visualisierten Objekte unter Berücksichtigung gegenseitiger Beziehungen wie z.B. Assoziationen, Vererbungen, etc. zu finden. Hierzu wurden in der ersten Phase Machbarkeitsstudien zu verschiedenen Anordnungsalgorithmen durchgeführt, auf die im folgenden Abschnitt näher eingegangen wird.

In der zweiten Phase wurden die gefundenen Ansätze konkret implementiert und an die übrigen Systemkomponenten angebunden. Außerdem wurden weitere, aus den User-Stories abgeleitete Anforderungen realisiert. Hier wären z.B. die Realisierung von Tooltips im dreidimensionalen Raum, ein Serialisierungsmechanismus, der über den von Java zur Verfügung gestellten Standardmechanismus hinausgeht oder die Möglichkeit, auf ein einzelnes Objekt im 3D-View zu fokussieren, zu nennen. Zudem musste die Datenstruktur an die neuen Anforderungen angepasst werden. Schließlich soll es dem Benutzer erlaubt werden, verschiedene grundlegende Einstellungen wie Speicherort, Farbgebung, Wahl des Anordnungsalgorithmus, etc. über eine Preference-Page vorzunehmen.

15.2 Machbarkeitsstudien

Carina Klar, Jonas Mathis, Michael Striewe, Dennis Weyland

Um die Möglichkeiten zur Visualisierung in diesem Release besser einschätzen zu können, wurden verschiedene Machbarkeitsstudien durchgeführt. Die Praktikabilität verschiedener Formen der Darstellung von Relationen (unterschiedliche Pfeilspitzen, etc.) wurde ausführlich

getestet. Zur Anordnung der Objekte in der dreidimensionalen Ansicht wurden verschiedene Verfahren erprobt. Zum einen das aus der Physik bekannte Verfahren des force-directed Layout, welches im wesentlichen auf der Vereinbarung beruht, dass sich ähnliche Objekte stärker anziehen (also später im Raum näher beieinander liegen) als Objekte, die wenig gemeinsam haben. Es zeigte sich, dass sich dieses Prinzip gewinnbringend für unsere Problemstellung einsetzen lässt. Zum anderen wurde ein Algorithmus zur Energieverteilung getestet, dessen erzeugte Anordnungen vor allem im Sinne des Clusterings sinnvoll erschienen. Desweiteren wurde ein evolutionäres Verfahren zur Anordnung der Objekte auf der Oberfläche einer Kugel getestet. Damit sollte der Tatsache Rechnung getragen werden, dass bei einer großen Anzahl visualisierter Objekte im Raum gewisse Objekte nur schwer zu identifizieren sind bzw. verdeckt sein können.

15.2.1 Darstellung und Unterscheidung von Objektbeziehungen

Um verschiedene Beziehungen zwischen Objekten übersichtlich darzustellen, ist die Verwendung eindeutiger grafischer Notationen notwendig. Um geeignete Möglichkeiten zu finden, wurden verschiedene Ansätze für die Beziehungsrepräsentation durch Pfeile und von spezifischen Darstellungen für Arrays erprobt.

Bei der Darstellung von Beziehungen ist es grundsätzlich sinnvoll, durch das Setzen von Pfeilspitzen zwischen gerichteten und ungerichteten Beziehungen zu unterscheiden. Für die weitere Unterscheidung von Pfeilen gibt es zwei verschiedene Möglichkeiten: Veränderung der Pfeilspitzen im Bezug auf ihre Form, Größe und Farbe sowie Veränderung von Verbindungslinien im Bezug auf Dicke und Farbe. In Anlehung an die UML-Notation wurden zunächst vier verschiedene Arten von Pfeilspitzen getestet: einfache, durch vier Linien gezeichnete Spitzen, kegelförmige Spitzen sowie leere und gefüllte Pastillen. Liniendicke und -farbe waren in den Prototypen frei parametrisierbar.

Die Erprobung in verschieden komplexen Zusammenhängen ergab, dass eine stark auf Farben basierende Pfeilsemantik wenig hilfreich ist, da aufgrund der Farbigkeit der Kugeln zu viele weitere Farben negative Auswirkungen auf die Übersichtlichkeit haben. Zudem sind Farben bei sehr dünnen Pfeilen sowie den Linienspitzen und leeren Pastillen kaum zu erkennen. Außerdem erweisen sich pyramidenförmige Spitzen und gefüllte Pastillen als besser erkennbar im Vergleich zu Linienspitzen und leeren Pastillen, was sich insbesondere beim Zoomen eines Snapshots positiv bemerkbar macht.

Im zweiten Release werden zwei Unterscheidungsmerkmale eingesetzt: zwischen einer „hat ein“-Beziehung und einer „erzeugt“-Beziehung wird durch das setzen einer blauen oder roten, jeweils pyramidenförmigen Spitze unterschieden. Die Pfeillinie ist grundsätzlich blau und gibt durch ihre Stärke die Vielfachheit einer Beziehung an, d.h. mit wachsender Zahl von Referenzen von einem Objekt auf ein anderes wächst die Dicke der Linie.

Für die Darstellung von Arrays wurde zusätzlich eine gesonderte Anordungsform erprobt, in der die einzelnen Elemente mit festen Abständen zueinander angeordnet werden. Bei dieser

Möglichkeit zeigten sich drei Nachteile: Erstens ist die Anordnung nur bis zu maximal drei Array-Dimensionen intuitiv, danach muss auf Verschachtelung von Objekten zurückgegriffen werden. Zweitens ergibt sich eine lückenhafte und damit unübersichtliche Darstellung, wenn Arraydimensionen verschiedene Größen haben bzw. Teile des Arrays nicht besetzt sind. Drittens ist bei voll besetzten dreidimensionalen Arrays eine sichere Navigation zu einem in der Mitte liegenden Element nahezu unmöglich. Aus diesen Gründen wurde auf eine besondere Anordnung für Arrayelemente verzichtet. Eine Auszeichnung von Array-Beziehungen durch die Verwendung einer andersfarbigen Pfeilspitze oder -farbe oder einer anderen Spitzenart ist im Rahmen der oben beschriebenen Pfeilsemantik denkbar, wurde im zweiten Release aber nicht weiter erpobt.

15.2.2 LinLog

Ein Ansatz zur Anordnung der Objekte in der 3D-View beruht auf dem Algorithmus LinLog (Noack (2003b)).

Ein Energiemodell legt dabei fest, nach welchem Maße sich die Objekte jeweils anziehen sollen. Der Algorithmus berechnet daraufhin eine Anordnung im Raum durch Minimierung der Gesamtenergie zwischen den einzelnen Elementen.

LinLog berechnet eine Anordnung, bei der interpretierbare Entfernungen zwischen den einzelnen Elementen entstehen. Diese Entfernungen können durch ein Energiemodell auf verschiedene Anwendungsfälle angepasst werden.

In dieser speziellen Anwendung war die Idee, das Energiemodell nach den folgenden Kriterien aufzubauen: Die kleinste Energie besteht zwischen Objekten, die nicht zur selben Klasse gehören und keine Relationen untereinander besitzen. Auf der nächsten Stufe befinden sich wiederum Objekte unterschiedlicher Klassen, die nun aber in Relation zuenander stehen. Es folgen Objekte derselben Klasse ohne Relationen. Objekte derselben Klasse, die durch eine Relation verbunden sind, besitzen die meiste Energie untereinander.

Mit dem oben beschriebenen Energiemodell erzeugt LinLog ein Layout, bei dem Objekte gleicher Klassen in Clustern angeordnet sind. Gleichzeitig fließen existierende Relationen in die Berechnung mit ein. Da diese Anordnung möglicherweise eine gute Darstellung einer Programmsituation sein könnte, wurde LinLog als ein Anordnungsalgorithmus für Eclipse-PlugIn der Projektgruppe eingesetzt.

15.2.3 Force-Directed Layout

Ziel dieser Machbarkeitsstudie war zu prüfen, ob eine sinnvolle Anordnung der visuellen Objekte (Arrays, Debug-Objekte, Relationen) auf Basis eines sogenannten Force-Directed-Algorithmus möglich ist.

Ein Force-Directed-Algorithmus verläuft in der Regel iterativ. In diesem Fall werden in jedem Iterationsschritt zuerst Kräfte berechnet, die auf die einzelnen Objekte wirken. Diese werden dann in einem weiteren Schritt dazu benutzt die Position der Objekte zu verändern. Man erhofft auf diese Weise nach einigen Iterationen ein Kräftegleichgewicht zu erhalten, in diesem Fall wäre die Summe der wirkenden Kräfte auf ein Objekt 0 bzw. läge unter einem bestimmten Schwellenwert, der eine Art Abbruchkriterium darstellt.

In unserem Fall würden z.B. Kräfte auf die einzelnen Debug-Objekte berechnet. Dabei könnte es sich z.B. um Kräfte handeln, die zwischen Objekten wirken, die in Relation zueinander stehen oder es könnte sich ganz allgemein um eine Kraft handeln, die Objekte, deren Abstand unter einem konstanten Schwellenwert liegt, voneinander wegbewegt.

Dieser Ansatz ist vom Prinzip her sehr flexibel. Unterschiedliche Kombinationen von Kräften mit verschiedenen Parametern sind denkbar. Auch die Abbruchbedingungen können unterschiedlichster Natur sein.

Getestet wurde der Algorithmus zunächst einmal an einem einfachen Beispiel. Es wurden einige Objekte (Kugeln) mit zufälliger Farbe erzeugt. Objekte, deren Farben ähnlich waren, standen automatisch in Relation zueinander. Es wurden nun zwei Kräfte definiert. Die erste sorgte dafür, dass zwei Objekte, die nah beieinander liegen, voneinander abgestoßen werden. Die zweite Kraft wirkte zwischen Objekten, die in Relation zueinander stehen, und zog diese Objekte gegenseitig an.

Man konnte in diesem Test sehr schön erkennen, wie sich ein FFarbwürfel“bildete. Dies war auch zu erwarten, da die Relation auf Basis der 3-Komponenten des RGB-Farbmodells basierte, und somit eine regelmäßige 3-dimensionale Struktur eine Folge der Definition dieser Relation war.

Von der erzielten Anordnung her kann man den Test als erfolgreich bezeichnen, lediglich die Laufzeit war nicht ganz so überzeugend und könnte Probleme bereiten bzw. müsste zu einer Anpassung des Ansatzes führen. Um die Qualität des Algorithmus im realen Einsatz besser testen zu können, wurde er in zwei Varianten in das PlugIn der Projektgruppe integriert.

15.2.4 Einfache Anordnungsalgorithmen

Unter dem Begriff der einfachen Anordnungsalgorithmen wurden in den Machbarkeitsstudien verschiedene Methoden getestet, die die Objekte in festen Strukturen und weitgehend unabhängig von den Objektbeziehungen anordnen.

Ein erster Ansatz war die Anordnung aller Objekte auf einem Kreisring und mit festem Abstand zwischen zwei benachbarten Objekten. Bei einem ausreichen groß gewählten Abstand bzw. Kreisradius garantiert eine solche Anordnung, dass kein Verbindungspfeil durch eine Kugel hindurch läuft. Aufgrund der Tatsache, dass diese Methode nur zwei Dimensionen des 3D-Raumes ausnutzt, ergeben sich allerdings zwangsweise viele Schnittpunkte zwischen Pfeilen. Außerdem wird die Darstellung schnell unübersichtlich, da in Beziehung stehende Objekte

sehr weit weg von einander angeordnet sein könnten. Zudem wird bereits bei wenigen Objekten der Radius des Kreises schon sehr groß.

Eine Weiterentwicklung dieses Ansatzes stellt die Anordnung der Objekte auf einer Kugeloberfläche dar. Die Objektpositionen auf der Kugeloberfläche werden deterministisch und weiterhin unabhängig von Objektbeziehungen berechnet. Auch diese Anordnung garantiert Überschneidungsfreiheit zwischen Pfeilen und Kugeln und vermeidet zudem viele Schnittpunkte zwischen Pfeilen. Die Ausnutzung der dritten Dimension reduziert zudem den Radius der Kugel erheblich. Vorteilhaft wirkt sich aus, dass jedes Objekt besonders effizient in den Vordergrund der Darstellung rotiert werden kann; nachteilig wirkt sich aus, dass alle Pfeile im Inneren der Kugel verlaufen und daher schwerer zu verfolgen sind.

Ein weiterer Schritt zur Verbesserung der Anordnung ist die Berücksichtigung von Objektbeziehungen, die zur Optimierung von Objektpositionen auf der Kugeloberfläche genutzt werden sollen. Es wurden zwei Verfahren getestet: Beim ersten wird eine zufällige Anordnung erzeugt und durch Verschieben und Vertauschen von Objektpositionen versucht, die Gesamtlänge aller Pfeile zu reduzieren. Beim zweiten werden Objekte nacheinander eingefügt und auf die jeweils bestmögliche Position gesetzt. Beide Verfahren erwiesen sich in nicht-trivialen Fällen als wenig effizient. Das erste Verfahren benötigte zu viele Iterationen, um weit entfernte Objekte zusammen zu führen; das zweite Verfahren kann bei ungeeigneter Einfügereihenfolge keine guten Lösungen garantieren. Aus diesen Gründen wurde die Weiterentwicklung beider Verfahren zugunsten anderer Verfahren eingestellt.

Die grundsätzlichen Vorteile einer Anordnung von Objekten auf einer Kugeloberfläche sind davon aber nicht betroffen, so dass eine Umsetzung dieser Strategie über ein geeignetes Force-Directed Layout für ein späteres Release denkbar ist.

15.3 User-Stories

Sebastian Vastag, Antonio Pedicillo

Im zweiten Release sollten die Erkenntnisse aus dem ersten Release angewendet und verbessert werden. Zielsetzung der Userstories war die Schaffung der Basisfunktionalität für eine komplette Version in einen späteren Release.
Die Userstories wurden in die Kategorien unterteilt (Darstellung, Technik und Usability).

Beziehung zwischen Objekten darstellen.
Beschreibung: „Die Beziehungen zwischen den Objekten sollen durch Pfeile visualisiert sein.“ (Kategorie ***Darstellung*** - *must-be*)
Story 3 Diese mit der höchsten Priorität bewertete Story beinhaltet die Umsetzung des Hauptziels dieses Releases. Beziehungen sollen durch Pfeile zwischen den dreidimensionalen Objekte darstellt werden. Für dieses Release kommen als Beziehungen in Frage:

- Objekt A steht mit Objekt B in Beziehung (Assioziation)
- Objekt A erzeugt Objekt B (Aufruf)

Sinnvolle Anordnung der Objekte
Beschreibung: „Eine sinnvolle Anordnung der 3D-Objekte unter Berücksichtigung der Beziehungen“ (Kategorie ***Darstellung*** - *must-be*)
Story 4 In der dreidimensionalen Ansicht ist algorithmisch eine „sinnvolle“ Anordnung der Objekte im Raum zu erreichen, sodass der Benutzer die Ähnlichkeiten und/oder die Beziehungen schnell erkennen kann.
Womöglich lassen sich Anwendungsfälle für spezialisierte Anordnungsalgorithmen finden.

Spezielle Darstellungsform für Arrays
Beschreibung: „Die Array sollen eine gesonderte Darstellungsform haben, damit sie visuell von anderen Objekten unterscheidbar sind.“ (Kategorie ***Darstellung*** - *must-be*)
Story 2 Als Darstellungsform für das Array wird ein Kubus benutzt. Von ihm gehen Assoziationspfeile zu den separat dargestellten Elemeneten des Arrays.

Anzeige und Wechseln von mehreren Snapshots
Beschreibung: „Angeben von mehreren Breakpoints, Aufzeichnung eines Snapshots bei jedem Breakpoint, später Wechsel der Ansicht zwischen den Snapshots.“ (Kategorie ***Technik*** - *costly-to-lose*)
Story 8 Der Benutzer kann vor dem Debugging mehrere Breakspoints im Quellcode setzen. In der 3D-View wird immer ein Snapshots des jeweiligen aktuellen Breakspoints angezeigt. Nachdem das zu testende Programm alle Breakpionts erreicht hat, läßt sich noch nachträglich zwischen den Snapshots wechseln.

Exceptions markieren
Beschreibung: „Herkunft von nicht gefangenen Exceptions markieren, z. B. als zerbrochene Kugeln“ (Kategorie ***Darstellung*** - *costly-to-lose*)
Story 5 Die Exceptions sollen auch eine spezielle Darstelllung ähnlich den Arrays erhalten, somit erhält der Benutzer einen besseren Überblick in der 3D-View.

„Tooltip“ auf Objekten in der 3D-View
Beschreibung: „Schnelle Anzeige von Informationen zu Objekten in der 3D-View durch Tooltips.“ (Kategorie ***Darstellung*** - *costly-to-lose*)
Story 6 Jedes Objekt in der 3D-Ansicht erhält einen Tooltip, er erscheint nach kurzer Wartezeit wenn man die den Mauszeiger über dem Objekt verweilt. Es sollen ähnliche Informationen wie in der PropertyView zur Verfügung gestellt werden. Somit kann sich der Benutzer in der 3D-Welt besser zurecht finden.

Timer: Aufzeichnung von Start bis Ende des Timers
Beschreibung: „Es kann ein Start- und Endzeitpunkt festgelegt werden, in diesem Intervall werden in kurzen Zeitabständen die zur Erstellung eines Snapshots benötigten Informationen gesammelt.“ (Kategorie ***Technik*** - *costly-to-lose*)
Story 7 Diese Userstory wurde im aktuellen Release nicht mehr berücksichigt, da keine sinnvolle Anwendung für eine solche Funktion erkennbar war. Zudem zeichneten sich Probleme beim Umgang mit den entstehenden großen Datenmengen ab.

Verfolgung von Veränderungen
Beschreibung: „Für ein speziell markiertes 3D-Objekt: Anzeige der Veränderungen zwischen verschiedenen Snapshots.“ (Kategorie ***Technik*** - *nice-to-have*)
Story 9 Der Benutzer markiert ein Objekt. Falls in einem aufgezeichneten späteren Snapshot dieses Objekt aus dem Speicher entfernt wird, also „null“ ist, sollte immer noch ein Schatten des Objektes in Form einer halbtransparenten Kugel oder ähnlichen sichtbar sein.

Hinweisfenster bei nicht erfolgter Auswahl in ChooseClassesView.
Beschreibung: „Falls ein Programm gestartet wird und nichts im ChooseClassesView ausgewählt ist soll ein Hinweis oder Fehler ausgeben.“ (Kategorie ***Usability*** - *nice-to-have*)
Story 10 Wenn der Benutzer in der 3Debug-Perspektive keine Klassen in der ChooseClassesView gewählt hat und trotzdem das Programm zum debuggen starten möchte, wird ein Hinweisfenster angezeigt. Der Benutzer hat die Wahl, ob er mit der leeren Auswahl auf eigene Verantwortung fortfahren möchte.

Perspektivenwechsel
Beschreibung: „Einstellbarer automatischer Wechsel in die 3Debug-Perspektive.“ (Kategorie ***Usability*** - *nice-to-have*)
Story 11 Wird der 3D-Debugger nicht in der 3Debug-Perspektive gestartet, soll ein Hinweisfenster mit der Möglichkeit zum Wechsel in die 3Debug-Perspektive erscheinen oder der Wechsel automatisch geschehen.

Stoppen des aktuellen laufenden Programms
Beschreibung: „Stoppen (terminieren) des zu untersuchenden Programmes auch außerhalb der originalen Debug-Perspektive.“ (Kategorie ***Usability*** - *nice-to-have*)
Story 12 Es ist ein Stoppschalter in die 3Debug-Perspektive einzubauen, der die gleiche Funktionalität wie der Stoppschalter (rotes Quadrat) der Debug-Perspektive besitzt.

15.4 Reflexion über die Tasks

Carina Klar, Michael Striewe

Im Folgenden wird über die Tasks des zweiten Release reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die signifikanten Probleme und die Differenzen zwischen Zeitabschätzungen und tatsächlich benötigter Zeit eingegangen. Außerdem werden die Tasks aus Gründen der Übersichtlichkeit in drei wesentliche Kategorien eingeteilt.

15.4.1 Datenstruktur

Statische Datenstruktur erweitern/überarbeiten

Beschreibung:	Die statische Datenstruktur wurde im letzten Release nicht wie geplant bzw. effizient genutzt, da sie nicht als Grundlage für den Filter verwendet wurde, sondern erst später erzeugt wurde und eine zusätzliche Klasse FilterData existierte. Deshalb soll die Struktur zur Verbesserung der Performanz überarbeitet werden. Im Zuge dieser Maßnahme soll die Klasse FilterData wieder verschwinden.
geplante Zeit:	3 Tage
reale Zeit:	Es traten keine Probleme auf und die Zeit konnte eingehalten werden.

Dynamische Datenstruktur erweitern/überarbeiten

Beschreibung:	Die dynamische Datenstruktur soll zur Repräsentation von Relationen zwischen Objekten erweitert werden.
geplante Zeit:	3 Tage
reale Zeit:	Die geschätzte Zeit war zutreffend. Allerdings wurden bereits im vorhinein konzeptionelle Ideen gesammelt.

Differenzbildung und Keyframes auf Snapshot-Ebene

Beschreibung:	Um mehrere Snapshots effizienter speichern zu können, sollen nur die Differenzen zwischen den einzelnen Snapshots oder einzelnen Keyframes im Speicher gehalten werden.
geplante Zeit:	3 Tage
reale Zeit:	Die Task wurde nicht umgesetzt, da die zu erwartende Reduktion des Speicherplatzverbrauchs den Performanzverlust durch den höheren Berechnungsaufwand nicht rechtfertigt.

15.4.2 Benutzerinteraktion

Einbau einer Funktion zur Auswahl eines Snapshots

Beschreibung: Es soll ermöglicht werden, einen von mehreren Snapshots zur Anzeige in der 3D-View auszuwählen und zwischen den Snapshots zu blättern.
geplante Zeit: 2 Tage
reale Zeit: Benötigt wurde nur ein Tag, wobei zusätzliche Optimierung notwendig ist, wie zum Beispiel die Namensgebung der einzelnen Snapshots.

Allgemeine Preferences für Speicherort, Darstellung, etc.

Beschreibung: Die Preferences von Eclipse sollen um eine weitere Seite mit Einstellungen für das Plug-In erweitert werden.
geplante Zeit: 2 Tage
reale Zeit: Es wurden 2 Tage benötigt und es traten keine Probleme auf.

Preferences für Launch

Beschreibung: Im LaunchManager soll ein zusätzlicher LaunchTyp angeboten werden, der Einstellmöglichkeiten für einen 3D-Debug-Vorgang anbietet.
geplante Zeit: 3 Tage
reale Zeit: Es wurden 3 Tage benötigt. Aufgrund fehlender Dokumentation über das Einhängen und Aktivieren der Tabs mußte viel experimentiert werden. Ein weiteres Problem war das Schreiben eines eigenen Launch-Delegates.

Stop-Button für das laufende Programm

Beschreibung: Ein laufender Debug-Vorgang soll über einen Button in der 3D-View gestoppt werden können.
geplante Zeit: 1 Tag
reale Zeit: Es traten keine Probleme auf und die Zeit konnte eingehalten werden.

Spezifische Breakpoints

Beschreibung: Für das 3D-Debugging soll ein zusätzlicher Breakpoint angeboten werden, so dass ein 3D-Debug-Vorgang durchlaufen kann, ohne das zu debuggende Programm anhalten zu müssen.

geplante Zeit: 5 Tage

reale Zeit: Aufgrund von Zeitmangel wurde diese Task nicht umgesetzt.

Hinweisfenster bei falsche Auswahl im ChooseClassesView

Beschreibung: Sollte in der ChooseClassesView keine Klasse oder alle Klassen ausgewählt sein, soll ein Hinweisfenster erscheinen, das die Richtigkeit der Auswahl abfragt.

geplante Zeit: 0.5 Tage

reale Zeit: Die Zeit konnte ohne Probleme eingehalten werden.

Automatischer Perspektiven-Wechsel

Beschreibung: Nach einem Klick auf den 3D-Debug-Button soll automatisch in die 3D-Debug-Perspektive gewechselt werden.

geplante Zeit: 3 Tage

reale Zeit: Es wurde nur ein Tag benötigt, da diese Task gleichzeitig mit der Task für die Einrichtung einer Preferences für den Launch umgesetzt werden konnte.

15.4.3 3D-Darstellung

Darstellung eines normalen Objektes

Beschreibung: Die Datenstruktur zur Darstellung normaler Objekte muss für die Erfordernisse der Darstellung von Relationen und zur Durchführung von Animationen erweitert werden.

geplante Zeit: 2 Tage

reale Zeit: Es traten keine Probleme auf und die Zeit wurde eingehalten.

Darstellung eines Arrays

Beschreibung: Es muss eine geeignete Darstellungsweise für Arrays gefunden und implementiert werden.
geplante Zeit: 2 Tage
reale Zeit: Es wurde nur ein Tag benötigt, da eine sehr einfache Darstellungsweise umgesetzt wurde.

Darstellung von Exceptions

Beschreibung: Es muss eine geeignete Darstellungsweise für Exceptions gefunden und implementiert werden. Zusätzlich muss der SnapshotCollector diese zuvor erkennen.
geplante Zeit: 3 Tage
reale Zeit: Es wurde nur ein Teil der Task umgesetzt. Dafür wurde ein Tag benötigt. Es fehlt die Erkennung auch geerbter Exceptiontypen, sowie die Eintragung einer Relation zum werfenden Objekt.

Darstellung von Beziehungen zwischen Objekten

Beschreibung: Es muss eine geeignete Darstellungsweise für Relationen zwischen Objekten gefunden und implementiert werden.
geplante Zeit: 5 Tage
reale Zeit: Diese Task konnte bereits nach 4 Tagen umgesetzt werden.

Anordnung der 3D-Objekte in der 3D-View

Beschreibung: Es muss ein geeigneter Algorithmus gefunden bzw. implementiert werden, der die Objekte sinnvoll in der 3D-View anordnet.
geplante Zeit: 12 Tage
reale Zeit: In der geschätzten Zeit konnten zwei verschiedene Algorithmen umgesetzt werden.

3D-Tooltips

Beschreibung: In der 3D-View sollen Tooltips mit Objektinformationen angezeigt werden, nachdem der Mauszeiger zwei Sekunden auf dem entsprechenden Objekt ruht.
geplante Zeit: 3 Tage
reale Zeit: In drei Tagen konnte die Task erledigt werden.

Eigenes KeyBehavior schreiben

Beschreibung:	Mit bestimmten Tastenkombination soll die Navigation in der 3D-Szene ermöglicht werden.
geplante Zeit:	1 Tag
reale Zeit:	Es wurden 3 Tage benötigt, denn es gab Probleme bei der Eventauslösung und mit der korrekten Rotation.

PickingBehaviors effizienter implementieren

Beschreibung:	Das PickingBehavior soll auf die neue visuelle Datenstruktur umgestellt und nach Möglichkeit effizienter implementiert werden.
geplante Zeit:	3 Tage
reale Zeit:	Da nicht viele Effizienzverbesserungen durchgeführt werden konnten, wurde nur ein Tag benötigt.

Darstellung der Fokussierung

Beschreibung:	Der Nutzer soll in der 3D-View einen Rahmen ziehen können, um die innerhalb liegenden Objekte zu fokussieren. Außerdem sollen ein selektiertes Objekt durch Rapid-Zooming in Mittelpunkt der View gerückt werden.
geplante Zeit:	7 Tage
reale Zeit:	Diese Task wurde aufgrund von Zeitmangel ins nächste Release verschoben.

15.5 Vorstellung der Architektur

Daniel Vogtland, Henning Zeller, Dennis Weyland

Im Nachfolgenden wird die in diesem Release implementierte Architektur vorgestellt. Dazu wird zunächst die geplante Architektur beschrieben, dann die realisierte Architektur festgehalten und abschließend beide verglichen. Auf Implementationsbereiche, die sich seit dem letzten Release nicht oder nur unwesentlich verändert haben, wird nur so weit es als notwendig erscheint eingegangen.

15.5.1 Beschreibung der geplanten Architektur

Den Ausgangspunkt für die Architektur dieses Releases stellt die bestehende Architektur des letzten Releases dar. Sie basiert auf dem Konzept *Model-View-Control* (MVC). Die von Eclipse gelieferten Debug-Informationen werden durch einen `Collector` gesammelt und im `Model` gespeichert. Zusätzlich wird in diesem Release durch die `Arrangements` Visualisierungsinformation berechnet und bei den zu visualisierenden Daten im `Model` abgelegt. Angestoßen wird dieser Prozess durch den `Action`-Bereich des Systems. Sämtliche visuelle Darstellung (basierend auf `Model`-Inhalten) wird schließlich durch den `Views`-Bereich realisiert.

Das `Model` soll mehrere Debugger-Schnappschüsse (Snapshots) verwalten. Serialisierung für diese soll unterstützt werden. Zur Reduktion des Speicherverbrauchs soll auf Differenzbildung und Keyframing zurückgegriffen werden. Ein Snapshot enthält die gefundenen Objekte sowie Relationen zwischen diesen. Eine Objektinformation enthält die eigentliche Debuginformation (Klasse, Objektidentität, Attributbelegungen) und Visualisierungsinformationen. Eine Relationsinformation beinhaltet die entsprechenden Objektinformationen und ist von einem bestimmten Typ (Aggregation bzw. Creation). Neben diesen dynamischen Strukturen beinhaltet das Model auch wieder die statische Datenstruktur des letzten Releases, die jedoch wieder in der ursprünglich geplanten Form verwendet werden soll.

Die `Arrangements` stellen (i.d.R. Force-Directed-) Layout-Algorithmen dar, die Visualisierungsinformationen zum `Model` hinzufügen. Dabei soll dieser Bereich durch Interface-Nutzung dynamisch gestaltet werden, also eine leicht erweiterbare Mehrzahl von Algorithmen unterstützen. Ihre Eingabe stellt Information aus dem `Model` dar, welche zuvor durch den `Collector` ermittelt wurde. Auch visuelle Informationen, die aus einem vorherigen Berechnungsschritt resultieren, können eine zusätzliche Eingabe darstellen.

Der Bereich `Views` visualisiert die im `Model` enthaltenen Daten. Eine eigene grafische Datenstruktur kapselt Java3D-Objekte und dient

- der Darstellung von Objekten mit Koordinaten und Erscheinung
- der Darstellung von Verbindungspfeilen zwischen diesen Objekten inklusive Erscheinung
- als Listener bezüglich grafischer Objekte und Pfeile.

Außerdem sollen 3D-Tooltips verbunden mit grafischen Objekten möglich sein. Eine Verbindung zwischen verschiedenen Snapshots soll als Animation mit Hilfe der grafischen Datenstruktur visualisiert werden.

Abb. 15.1 zeigt einen geplanten Pipeline-ähnlichen Ablauf des Programms in Form eines Aktivitätsdiagrammes mit Bezug auf die Systemarchitektur.

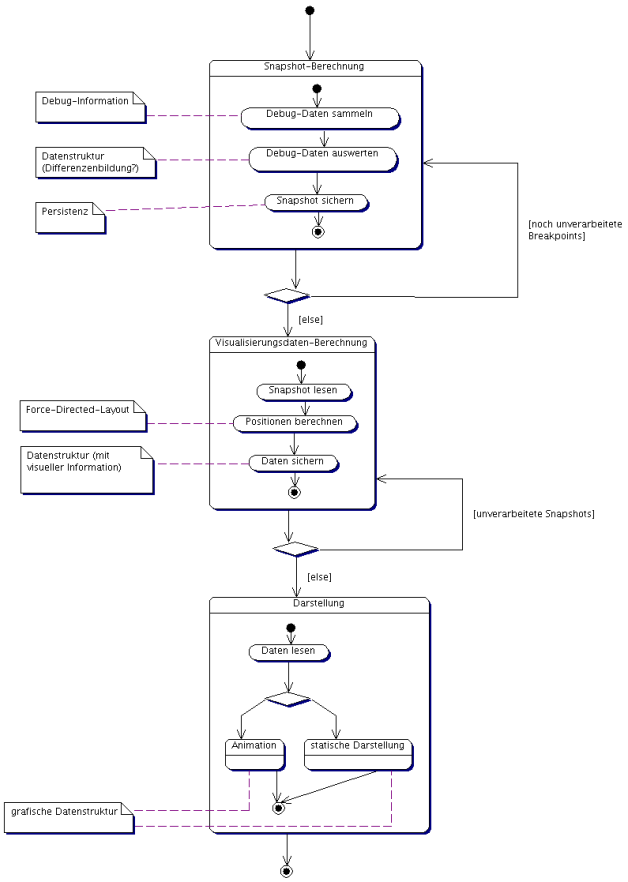


Abbildung 15.1.: Ein erster Entwurf für die Phasen des Visualisierungsprozesses (Release 2)

15.5.2 Beschreibung der realisierten Architektur

Abb. 15.2 zeigt die Umsetzung der Planung in der Paketstruktur. Das MVC-Modell findet ähnlich wie im Release 1 seine Entsprechung in den Paketen `model`, `views` (ohne das Paket `arrangement`) und dem Zusammenspiel aus `core`, `actions` und `collector`. Im `core` befinden sich auch die Serialisierungsklassen `SnapshotInfoXML` mit dem zugehörigen `xml`-Paket (Persistenz per XML-In-/Export) und `SnapshotinfoSerialize` (Persistenz per Objektserialisierung; zugunsten der XML Variante wird dieser Mechanismus nicht mehr verwendet). Weiterhin existieren zu der XML-Persistenz zwei weitere Pakete, diese wurden durch den XML-Compiler der verwendeten JAXB-API automatisch erstellt und besitzen nur unterstützenden Funktionalität. Das Paket `preferences` stellt diverse Klassen zur Unterstützung eines eigenen Launch-Typs (u.a. Auswahl des Arrangement-Algorithmus) und globa-

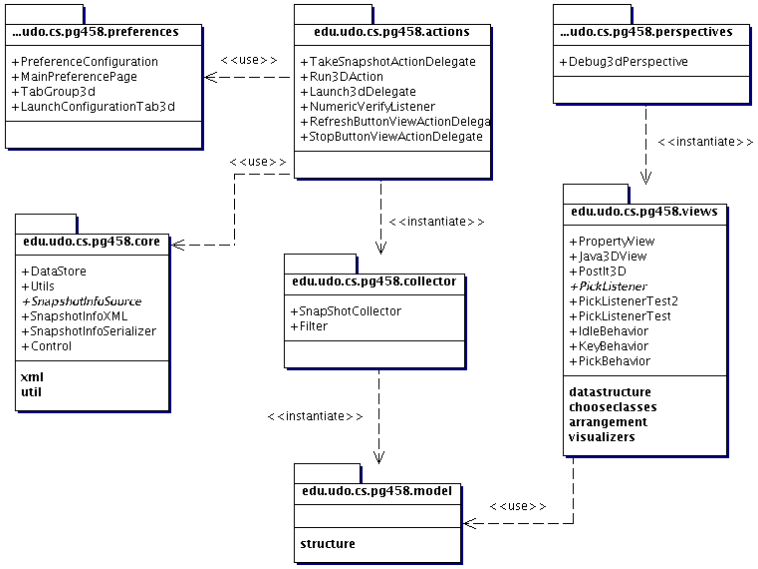


Abbildung 15.2.: Paketstruktur der realisierten Systemarchitektur (Release 2)

ler Einstellungen zur Verfügung. Die Pakete `pg458` (ohne unterliegende) und `perpectives` sind notwendige Implementierungen zur Integration des Plug-Ins in Eclipse und zur Initialisierung der Views.
Im Folgenden wird auf einige Bereiche der Architektur näher eingegangen.

Abb. 15.3 zeigt die wichtigsten Klassen bezüglich des Models. Sie gliedern sich in zwei Bereiche: statische Datenstrukur und dynamische Datenstruktur. Erstere entspricht in ihrer Funktionalität der statischen Datenstruktur aus dem ersten Release, wurde jedoch wieder besser der schon ursprünglich für Release 1 geplanten Anforderungen angepasst. Sie beschreibt die statischen Aspekte (Pakete, Klassen, Attribute) für ein zu debuggenden Programms. Die dynamische Datenstruktur beschreibt einen Snapshot (Momentaufnahme) des laufenden zu debuggenden Programms. Ein `SnapshotInfo`-Objekt beinhaltet eine bestimmte Anzahl von `ObjectInfo`-Objekten. Diese repräsentieren die Objekte des zu debuggenden Programms und sind von einem der Typen „standard“, „array“ oder „exception“. Sie beinhalten neben der Typangabe auch direkte visuelle Informationen (Farbe, Größe, Koordinaten) und einen Verweis auf ein `DebugObject` (beinhaltet die eigentliche Debugger-Information, wurde aus Release 1 übernommen). Daneben besitzt ein `SnapshotInfo`-Objekt eine bestimmte Anzahl von `RelationInfo`-Objekten. Eine `RelationInfo` repräsentiert eine gerichtete Relation von einer `ObjectInfo` zu einer anderen und ist vom Typ „standard“(entspricht besitzt-ein) oder „creation“ (entspricht hat-erzeugt). Die beiden Klassen `ObjectInfoComparator` und `IdentityComparator` dienen dem ordnenden Vergleich zweier `ObjectInfo` Objekte anhand der vom Debugger gelieferten IDs.
Wie schon in Release 1 übernimmt die im `core` liegende Klasse `DataStore` die Funktion

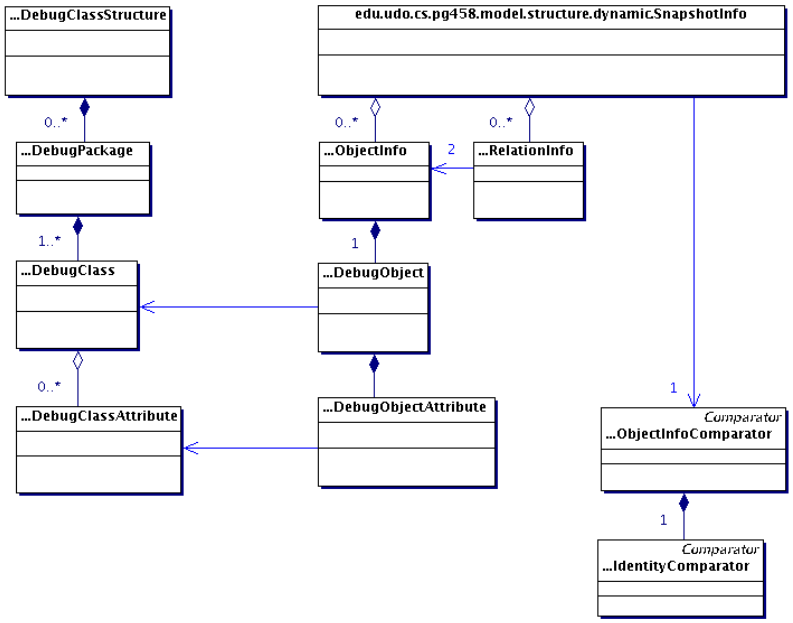


Abbildung 15.3.: Klassen des Models (Release 2): statische Datenstruktur (links), dynamische Datenstruktur (rechts)

eines Repository. Allerdings können diesmal mehrere Snapshots verwaltet werden. Es wird jedoch weiterhin nur eine `DebugClassStructure` verwaltet.

Abb. 15.4 zeigt die Hauptklassen des Controlbereichs. Die statische Datenstruktur wird durch Aufruf einer der Klassen `Run3DAction` (Plug-In Mainbutton) oder `Launch3dDelegate` (Start eines eigenen Launchtyps) gestartet. Der `Filter` bestimmt, welche Klassen beim Debugging-Prozess betrachtet werden. Der Debugger wird gestartet und benachrichtigt bei einem gefundenen Haltepunkt den `SnapshotCollector`. Dieser erzeugt unter Beachtung der durch den `Filter` gegebenen Beschränkungen eine `SnapshotInfo` (vorerst ohne visuelle Information). Anschließend wird diese `SnapshotInfo` an den ausgewählten Arrangement-Algorithmus weitergericht, der die Koordinaten der `ObjectInfo` Objekte berechnet. Die Klasse `DeterministicColoring` bestimmt die Farben und berücksichtigt gegebenenfalls die Farben des letzten Snapshots um gleiche Farben für gleiche Klassen zu gewährleisten. Die `SnapshotInfo` wird anschließend an die Klasse `DataStore` weiter gegeben, welche die Datenstrukturen verwaltet. Sie kennt außerdem alle Sichten. Die `Control`-Klasse kennt ebenfalls alle Sichten, implementiert das *Singleton*-Pattern und koordiniert als `PickListener` die Kommunikation zwischen der `Java3Dview` und der `PropertyView` bezüglich eines ausgewählten Objekts. Wird über den entsprechenden Button ein Snapshot manuell ausgelöst, so erzeugt `takeSnapshotActionDelegate` dynamisch einen neuen Haltepunkt und die Anwendung reagiert wie gerade beschrieben.

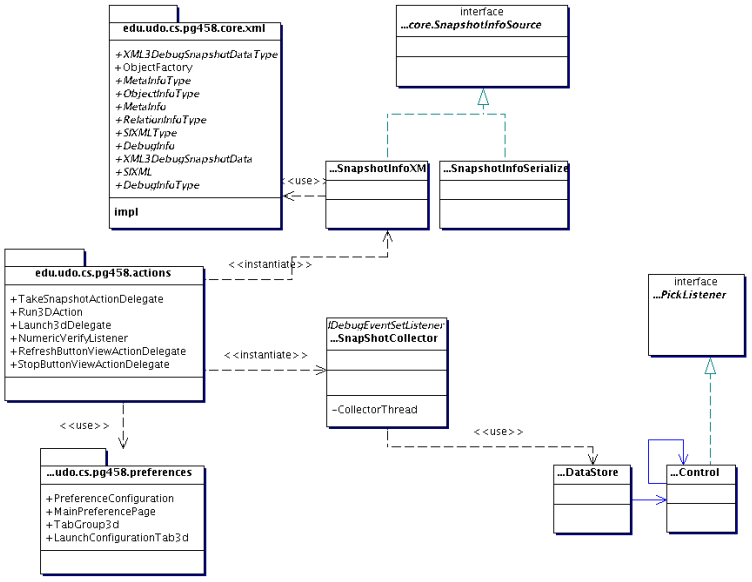


Abbildung 15.4.: Die Hauptklassen des Controlbereichs (Release 2)

Das Paket `views` stellt Sichten auf verschiedene Bereiche des Models zur Verfügung. Es handelt sich um drei Sichtklassen:

- `ChooseClassesView`: Diese Klasse mit ihren untergeordneten Klassen (s. Abb. 15.5) stellt die statische Datenstruktur dar und erlaubt eine Auswahl von Teilbereichen dieser für die Betrachtung beim Debugging-Prozess mit Hilfe der `Filter`. Die Implementierung hat sich bis auf einen Punkt seit Release 1 nicht wesentlich verändert, die statische Datenstruktur wird nun (dem MVC-Modell eher entsprechend) außerhalb der `ChooseClassesView` erzeugt.
- `PropertyView`: Diese Klasse zeigt die Belegung der Attribute und untergeordnete Objekte eines in der `Java3DView` ausgewählten Objektes an. Die Implementierung entspricht der des ersten Releases.
- `Java3DView`: Diese Klasse hat die meisten Änderungen erfahren. Sie ist im Kontext anderer zugehöriger Klassen in Abb. 15.6 dargestellt. Ihre Aufgabe besteht in der Visualisierung eines Snapshots (vorhandene Objekte und ihre Relationen untereinander), so wie dem animierten Übergang zwischen zwei Snapshots. Ein eigenes `KeyBehavior` wirkt einem Bug in der Windows-Implementierung von Java3D entgegen. Das schon im ersten Release verwendete `PickBehavior` dient der Synchronisation mit der `PropertyView` beim Auswählen eines Objektes. Beim `IdleBehavior` handelt es sich um einen Mechanismus, der die Anzeige eines dreidimensionalen Tooltips mit Hilfe von

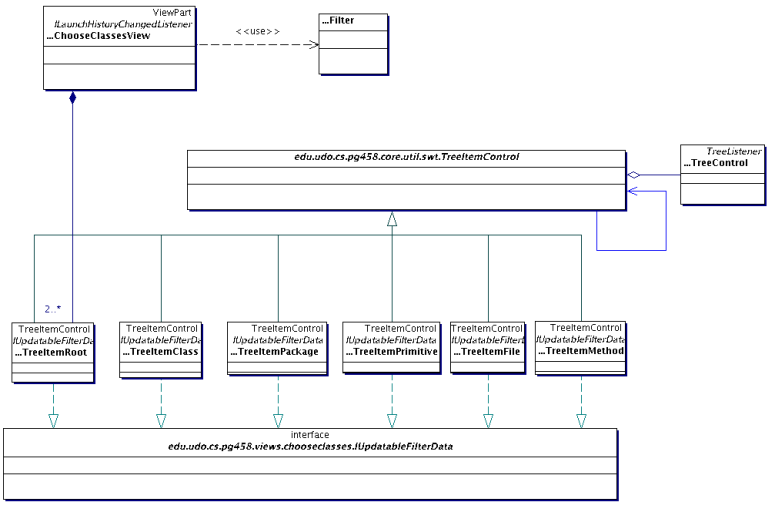


Abbildung 15.5.: Die internen Hauptbestandteile der `ChooseClassesView` (Release 2)

`PostIt3D` für ein Objekt, auf dem der Mauszeiger einige Zeit ruht, ermöglicht. Es sind einige neue Steuerelemente integriert worden: Auswahl eines Snapshots, Übergang zum nächsten oder vorherigen Snapshot, eigener Stop-Button für den Debug Prozess, manueller Snapshot, Zurücksetzung der Sicht in den initialen Zustand und Fokussierung des ausgewählten Objekts in der Sicht.
Die eigentliche Visualisierung nehmen die Klassen `SnapshotVisualizer` (einzelner Snapshot) und `SnapshotTransitionVisualizer` (Animation eines Übergangs) vor. Beide nutzen eine grafische Datenstruktur, die Java3D-Objekte kapselt. Der `SnapshotVisualizer` wird mit einem `SnapshotInfo`-Objekt initialisiert. Zu jeder `ObjectInfo` wird ein `PrimitiveContainer` erzeugt, dessen Implementierung von der Objektinformation abhängig ist. Farbe und Koordinaten des Mittelpunktes werden aus der dynamischen Datenstruktur ausgelesen. Falls mindestens eine gerichtete Relation zwischen den Objekten vorhanden ist, wird ein `ArrowContainer` erzeugt, der die entsprechenden `PrimitiveContainer` visuell verbindet. Abhängig von Anzahl und Typen der bestehenden Relationen wird eine entsprechende Implementierung (momentan nur `DirectedPrimitiveArrow` oder `BiDirected-PrimitiveArrow`) mit variabler Linienstärke und unterschiedlichen Pfeilspitzenfarben gewählt. Der `SnapshotTransitionVisualizer` arbeitet ähnlich. Allerdings wird hier für zwei Objekte mit gleichen IDs nur ein `PrimitiveContainer` verwendet (Positionsänderungen werden jedoch berücksichtigt). Die Animation verläuft über einen definierten Zeitraum. Koordinatenänderungen werden bei einem `Primitive-Container` direkt, bei einem `ArrowContainer` mit Hilfe der Listener-Funktionalität (`CoordinatesChangeListener`) realisiert. Die Erzeugung/Zerstörung von Objekten bzw. Relaionen wird mit Ein-/Ausblenden visualisiert. Der `SnapshotTransi-`

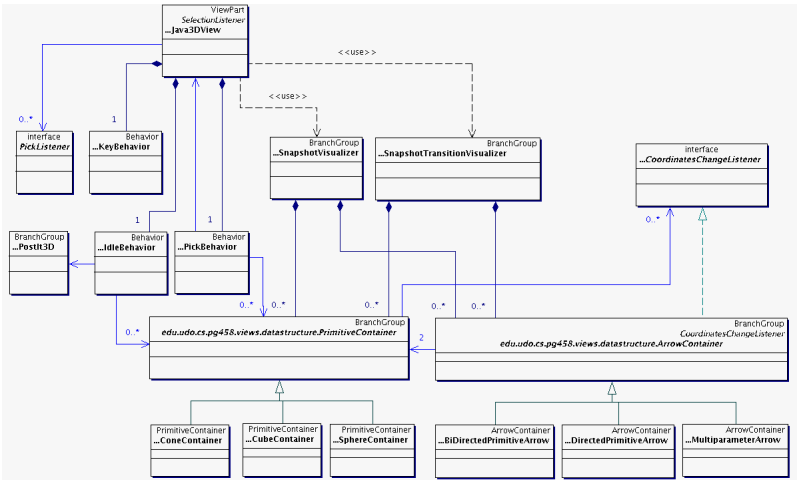


Abbildung 15.6.: Die internen Hauptbestandteile der `Java3DView` (Release 2)

`tionVisualizer` unterstützt derzeit weder das `IdleBehavior` noch das `PickBehavior`.

Die Anordnungsalgorithmen setzen und verändern die Positionen der Objekte innerhalb eines Snapshots (`SnapshotInfo`). Das Interface `ISnapshotArrangement` ist die Schnittstelle dieser Anordnungsalgorithmen und besteht lediglich aus der Methode `arrange`.
Bisher gibt es folgende Anordnungsalgorithmen:

- `ForceDirectedArrangement`: Ein flexibler, aber dafür recht rechenintensiver Force-Directed-Anordnungsalgorithmus.
- `MinimizerPolyLogBarnesHut`: Ein sehr schneller Force-Directed-Anordnungsalgorithmus.
- `SimpleArrangement`: Dieser Algorithmus wird vorwiegend zum Testen benutzt.
- `InitializeSnapshotArrangement`: Ein simpler Algorithmus, der die Positionen der einzelnen Objekte deterministisch initialisiert.
- `FocusArrangement`: Dieser Algorithmus verschiebt die Objekte so, dass sich alle im sichtbaren Bereich der 3D-Ansicht befinden.
- `ComposedSnapshotArrangement`: Dies ist kein wirklicher Anordnungsalgorithmus. `ComposedSnapshotArrangement` stellt eine Möglichkeit zur Kombination mehrerer Anordnungsalgorithmen zur Verfügung, die nacheinander in einer Pipes & Filters Struktur durchlaufen werden. Dies eignet sich z.B. zur Kombination der Anordnungsalgorithmen `InitializeSnapshotArrangement`, `ForceDirectedArrangement` und `FocusArrangement`. Hierbei werden zuerst die Positionen der

Objekte initialisiert, danach werden die Objekte basierend auf der Initialisierung über den Force-Directed-Anordnungsalgorithmus angeordnet und zuletzt werden die Objekte so verschoben, dass sich alle im sichtbaren Bereich der 3D-Ansicht befinden.

Um die Anordnungsalgorithmen möglichst flexibel zu halten wurde außerdem noch eine abstrakte Klasse für Abbruchbedingungen, `AbortCondition`, definiert. Momentan sind folgende Abbruchbedingungen implementiert:

- `IterativeAbortCondition`: Diese Abbruchbedingung tritt nach einer vorher festgelegten Anzahl an Iterationen ein.
- `CoordinateChangeAbortCondition`: Diese Implementierung von `AbortCondition` stoppt den Algorithmus sobald die Positionsveränderungen der Objekte einen vorher festgelegten Wert unterschreiten und somit eine Konvergenz des Anordnungsalgorithmus andeuten.
- `ComposedOrAbortCondition`: Dies ist eine zusammengesetzte Abbruchbedingung, die den Algorithmus stoppt, sobald eine der in ihr enthaltenen Abbruchbedingungen ein Terminieren signalisiert. Eine Verwendung von `CoordinateChangeAbortCondition` zusammen mit `IterativeAbortCondition` in einer `ComposedOrAbortCondition` führen zu folgendem Verhalten: Der Algorithmus stoppt sobald eine Konvergenz zu erkennen ist oder sobald die maximale Iterationszahl erreicht wird. Dadurch wird auf einfache Weise eine obere Schranke für die Laufzeit der Anordnungsalgorithmen definiert, die allerdings bei Konvergenz nicht erreicht wird.

Zuletzt sollten noch die elementaren Bestandteile eines `ForceDirectedArrangements` erwähnt werden. Dabei handelt es sich um Kräfte, die zwischen Objekten wirken. Bisher gibt es die folgenden zwei Kräfte, die auf der Schnittstelle `IForce` aufbauen:

- `GlobalRejectingForce`: Diese Kraft stößt Objekte, deren Abstand unter einem vorher festgelegten Wert liegtm voneinander ab. Dies dient dazu um Ansammlungen von Objekten auf engem Raum (evtl. sogar mit Überlagerungen) zu vermeiden und insgesamt eine übersichtlichere Anordnungs zu erzeugen.
- `RelationalForce`: Diese Kraft wirkt nur zwischen Objekten, die in einer Beziehung zueinander stehen (z.B. Assoziation) und zieht diese an. Dadurch werden die Objekte entsprechend ihres Kontexts angeordnet. Ob diese Anordnung im Endeffekt einen Sinn hat wird sich erst im späteren Verlauf des Projekts zeigen.

Abb. 15.7 zeigt alle Klassen (mit Ausnahme der Pakete `actions`, `preferences` und `xml`) und ihre Beziehungen untereinander. Die Anordnung entspricht im Wesentlichen der der Ausschnittsdiagramme. Man findet grob die Bereiche Control ohne Arrangement (links oben; mittig, leicht rechts), Model (mittig oben), Views (unten) und Arrangement-Algorithmen (rechts oben). Die Klasse `Utils` im `core` Paket wurde vernachlässigt, da sie nur statische Methoden für Binäre Suche und Einfügen mittels Binärer Suche in `Vector`-Listen bereitstellt.

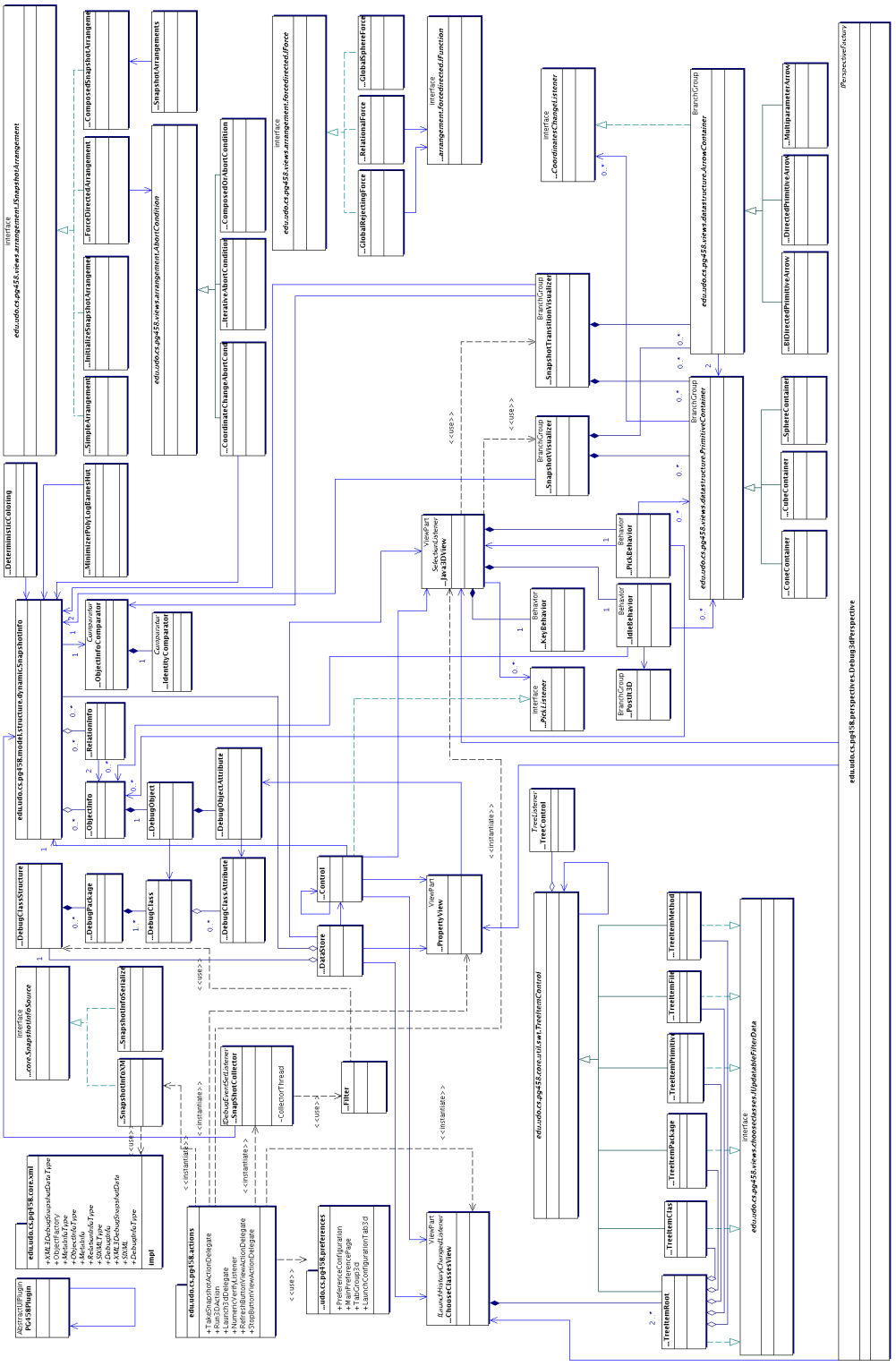


Abbildung 15.7.: Die gesamte Klassenstruktur mit Ausnahme der Pakete `actions`, `preferences` und `xml` (Release 2)

15.5.3 Vergleich von geplanter und realisierter Architektur

Die Umsetzung des angestrebten MVC-Modells gelang diesmal noch besser als beim ersten Release. Es fällt allerdings auf, dass die Paketstruktur unter diesem Gesichtspunkt nicht immer sinnvoll gewählt wurde. Das Paket `arrangements` sollte nicht dem Paket `views` untergeordnet sein, da es zwar für die Berechnung von Visualisierungsdaten zuständig ist, jedoch nicht selbst visualisiert (als View agiert). Das Paket `util.swt` wäre dagegen besser unterhalb des Pakets `views` angesiedelt, da es sich um Hilfsklassen für die visualisierende `ChooseClassesView` handelt.
Die Klassen `Control` und `DataStore` weisen in Hinblick auf Sichtenverwaltung ähnliche Funktionalität auf. Dieser Punkt sollte im nächsten Release überdacht werden.
Die geplante Differenzbildung wurde letztendlich doch verworfen, da sie nach Meinung der Entwicklerteams keine nennenswerten Vorteile bietet. Selbst unter Ausschluß der Visualisierungsinformationen ändert sich ein Großteil der Daten von Snapshot zu Snapshot. Unter Umständen entstünden jedoch einige Nachteile durch mehr Berechnungsaufwand, Performanceverlust und Informationsverlust.
Ein großes Manko stellt die Realisierung der Arrangement-Algorithmen dar. In der Planung sollte eine Interface-Struktur für alle Algorithmen entworfen werden. Diese Struktur wurde zwar entwickelt, jedoch wird sie von `MinimizerPolyLogBarnesHut` nicht verwendet. Dieses Manko sollte recht einfach behoben werden können, da auch dieser Algorithmus intern eine ähnliche Struktur benutzt.
Momentan werden keine Selbstrelationen visualisiert, die Datenstruktur unterstützt diese jedoch bereits. Die Animation berücksichtigt momentan noch keine Änderung der visuellen Relation (Typ, Richtung, Stärke) zwischen zwei Objekten. Auch die Änderung eines Objekttyps (casting) wird nicht unterstützt. Dies wurde jedoch in der Planung auch nicht explizit betrachtet.
Der geplante Pipeline-Ablauf wurde dagegen recht genau umgesetzt und scheint auch für weitere Releases einen guten Ausgangspunkt darzustellen. Auch die Animation von Snapshotübergängen und die Möglichkeit der Speicherung (was prinzipiell auch schon eine nachträgliche Neuberechnung der Visualisierungsdaten mit einem anderen Arrangement-Algorithmus ermöglicht) stellt einen Schritt in Richtung Verwirklichung der Systemmetapher dar.

15.6 Akzeptanztests

Antonio Pedicillo, Sebastian Vastag

In der Projektgruppensitzung am 27.02.2005 wurden die Akzeptanztests des zweiten Release durchgeführt. Das bedeutet, dass die Kunden anhand der von ihnen zu Beginn aufgestellten User-Stories das von den Entwicklern ausgelieferte Programm auf deren korrekte Umsetzung überprüfen. Dazu erstellten die Kunden Varianten von Programmdurchläufen, die die verschiedenen aus den User-Stories hervorgehenden Anforderungen an das Programm abdecken.

Hauptsächlich diente das Programm „Dave“ als zu debuggendes Programm. Um spezielle Tests durchführen zu können, erstellten die Kunden aber auch eigene kleinere Programme. Im Nachfolgenden werden die verschiedenen Tests und deren Ergebnisse aufgeführt.

Es soll ein Objekt angewählt werden und danach eine vergrößerte Darstellung in der Mitte der Ansicht angezeigt werden.
Testergebnis: nachträglich bestanden
Anmerkung: Der Benutzer wählt mit der Maus in der 3D-View ein Objekt aus und drückt dann auf die Fokussierschalter.

Tooltip soll über einen Objekt erscheinen, wenn die Maus länger darüber verweilt.
Testergebnis: bestanden
Anmerkung: Der Benutzer geht mit der maus auf ein Objekt und wartet zwei Sekunden. Danbnach erscheint ein Tooltip, indem diverse Informationen des Objektes angezeigt werden.

Freies Bewegen soll in der Ansicht (x,y-Achse, Drehung) möglich sein.
Testergebnis: bestanden
Anmerkung: Der Benutzer kann nach belieben mit der Maus oder mit der Tastatur sich in der 3D-Welt bewegen.

Arrays sollen in der Ansicht kenntlich gemacht/unterschieden werden.
Testergebnis: bestanden
Anmerkung: Ein Array wird als Kubus dargestellt. Enthaltene Objekte werden mit Assoziationspfeilen dargesetllt.

Beziehungen zwischen Objekten sollen erkennbar sein.
Testergebnis: bestanden
Anmerkung: Die Beziehungen werden durch dreidimensionale Pfeile darstellt. Die Assoziation wird durch einen blauen Pfeil und der Aufruf wird durch einen roten Pfeil gekennzeichnet.

Objekte, die im Programm in einer Beziehung stehen, sollen auch in erkennbarer visueller Beziehung stehen.

Testergebnis: bestanden

Anmerkung: Durch ein Force-Directed-Layout-Algorithmus werden die Objekte, die in Beziehung stehen, sinnvoll angeordnet. Dabei kann der Benutzer auf verschiedene Algorithmen zurückgreifen. Ein Algorithmus ordnet die Objekte nach einem bestimmten Cluster-Prinzip und der andere Algorithmus beinhaltet verschiedene Kraftauswirkungen.

Anzeigen mehrerer Snapshots von verschiedenen Breakpoints sowie Wechsel zwischen den Snapshots soll möglich sein.

Testergebnis: bestanden

Anmerkung: Der Benutzer kann in der 3Debug-Perspektive mit einer Combobox zwischen den einzelnen Snapshots wechseln.

Ein Programm wirft eine Exception. Das Objekt welches die Exception geworfen hat, soll kenntlich gemacht werden.

Testergebnis: bestanden

Anmerkung: Für Expections wurde eine als Darstellungsform ein Kegel gewählt.

Es soll die Veränderung eines gewählten Objektes zwischen zwei Snapshots angezeigt werden.

Testergebnis: bestanden

Anmerkung: Zur Anzeige der Veränderung eines Objektes werden zwei Snapshots gebraucht. Es wird eine Differenz zwischen zwei Snapshots berechnet und amimiert in der 3D-View angezeigt.

Keine Klassen in der ChooseClassesView sollen auswählt werden und der 3D-Debugger wird gestartet. Ein Hinweisfenster sollte erscheinen.

Testergebnis: bestanden

Anmerkung: Wird mit einer leeren Auswahl an Klassen und Javapaketen gestartet, erscheint ein Auswahlfenster, indem Hinweis mit der Aufschrift “You have checked either all or no items in ChooseClassesView. Continue anyway?äusgegeben wird.

Wenn Eclipse in der Java-Perspektive ist, sollte beim Starten des 3D-Debuggers die Perspektive automatisch zur 3D-Debug-Perspektive wechseln oder danach gefragt werden.
Testergebnis: bestanden
Anmerkung: keine

Anhalten eines beliebigen gestarteten Programmes soll aus der 3Debug-Perspektive möglich sein.
Testergebnis: bestanden
Anmerkung: Ein Stoppknopf ist in der 3Debug-Perspektive vorhanden.

KAPITEL 16

Beschreibung des dritten Release

16.1 Einleitung

Jonas Mathis

Nachdem im zweiten Release viele Erweiterungen für die 3D-Visualisierung in das Plug-In eingefügt wurden, wobei insbesondere die Anordnungsalgorithmen zu nennen sind, wurde die durch die vorlesungsfreie Zeit entstandene Pause vor allem für Bereinigungsarbeiten am Code, sowie für die Reflexion des Debugging-Prozesses genutzt. Dazu wurden in der PG zwei Untergruppen gebildet, wobei die erste für das „Code-Review“ zuständig war, also für Wartungsarbeiten am Code, um die Struktur der Klassen zu verbessern und evtl. auch logische Fehler zu beheben, und die zweite Gruppe noch einmal die Konzepte des Debuggings für alle verständlich aufarbeiten sollte. Desweiteren hat man sich überlegt, wie das Plug-In zu diesem Zeitpunkt in das übliche Vorgehen beim Debugging mit einbezogen werden könnte. Um die Ergebnisse der gesamten Gruppe vorzustellen, wurde daher vor Beginn des zweiten Semesters ein Workshop abgehalten, in dessen Rahmen die Veränderungen bis dahin vorgenommenen Änderungen am Code vorgestellt und die Vor- und Nachteile beim Einsatz des Plug-Ins herausgearbeitet wurden. Bei einem Brainstorming im Laufe des Workshops wurde die „Marschroute“ für dieses dritte Release festgelegt und tiefergehende, d.h. klassenübergreifende Änderungen mit allen Projektteilnehmern abgesprochen. Als Ziele für dieses Release wurden nur wenige neue Funktionalitäten festgelegt. Stattdessen lag der Fokus auf der Konsolidierung des bisherigen Projektfortschritts und technischer Optimierung der bereits vorhandenen Funktionen. Aufgrund dieser Tatsache wurde darauf verzichtet, die im Rahmen des *eXtreme programming* sonst üblichen Rolle der Kunden zu vergeben.

Im Vordergrund des dritten Releases sollte besonders die Beschaffung der Debug-Informationen stehen, da die bisher verwendete Variante zu viele Schwachstellen (hoher Platzverbrauch und Laufzeit bzw. Unvollständigkeit) an den Tag legte. Zu diesem Zweck sind zu verschiedenen Ansätzen Machbarkeitsstudien durchgeführt worden, und die Beschaffungsalgorithmen wurden implementiert. Um dem Ziel eines „Laufzeit-Films“ (oder in abgeschwächter Form: eine wie auch immer bewegte Abfolge von Snapshots) näher zu kommen, sollte im Laufe des Releases ein Rahmen für die Animation von Snapshots geschaffen werden, also die graphische Überführung eines Snapshots in einen anderen.

16.2 Reflexion über die Tasks

Antonio Pedicillo, Andrey Lysenko

Im Folgenden wird über die Tasks des dritten Release reflektiert. Hierzu werden die einzelnen Tasks kurz vorgestellt und insbesondere auf die signifikanten Probleme und die Differenzen zwischen Zeitabschätzungen und tatsächlich benötigter Zeit eingegangen.

Die Tasks wurden in vier Gruppen gegliedert, die thematisch als Schwerpunkte aus der Releaseplanung hervor gingen. Die Kategorie „Aufräumarbeiten“ beinhaltet Tasks, die aus einem Code-Review des bisherigen Projektverlaufs entstanden. Die weiteren drei Kategorien, „Parametrisierung Layout-Algorithmen“, „Infobeschaffung“ und „Animationen“ repräsentieren die drei Hauptziele dieses Release, die in den folgenden Kapiteln noch näher erläutert werden.

16.2.1 Aufräumarbeiten

Refactoring der Klasse Java3DView

Beschreibung:	Die Klasse Java3DView gehört zu den mächtigsten Klassen unseres Plug-Ins. Es sollten Aufgaben, die diese Klasse erledigt, gegebenfalls ausgelagert werden.
geplante Zeit:	2 Tage
reale Zeit:	Aufwand erheblich geringer, da ursprünglich geschätzte, wesentliche Änderungen nicht sinnvoll erschienen.

Switch-Statements in der Behaviorklassen entfernen

Beschreibung:	Die KeyBehavior-, PickBehavior-, und Idlebehavior-Klassen beinhalteten lange switch-statements. Diese sollen zur Übersichtlichkeit verkürzt werden.
geplante Zeit:	1 Tag je Untertask
reale Zeit:	Aufwand erheblich geringer, da die switch-statements in privaten Methoden ausgelagert wurden.

Pfeilspitzen an Cones und Cube anpassen

Beschreibung: Dieser Task besteht aus Untertasks. Es soll eine bessere Methoden herausgefunden werden, da die Pfeilspitzen den Kegel und Kubus nicht berühren.

geplante Zeit: 2 Tage je Untertask

reale Zeit: Aufwand hat sich beim Kegel bestätigt. Es wurde auf eine mathematische Formel zurückgegriffen, die von einem Halbkreis, in dem ein rechtwinkliges Dreieck liegt, den Innenkreisradius berechnet. Der Aufwand für den Kubus war erheblich weniger.

16.2.2 Infobeschaffung

Überprüfen der möglichen Alternativen für die Infobeschaffung

Beschreibung: Dieser Task wurde wegen der zwei existierenden Infobeschaffungsalternativen in zwei Untertasks eingeteilt.

geplante Zeit: 6 Tage

reale Zeit: Der Aufwand hat sich als erheblich schwieriger erwiesen.

Alternative 1: „-Xrunhprof Profiler“

Beschreibung: In diesem Task soll überprüft werden, ob und ggf. wie die Anbindung des Profilers zur Erzeugung und des Parsers zur Analyse des Heap-Dumps an das Plug-In möglich ist. Insbesondere war das Erzeugen von Sockets, Dateien und das Setzen von Startparametern für die VM zu testen.

geplante Zeit: 3 Tage

reale Zeit: Die Zeit wurde vollständig benötigt, aber alle auftretenden Probleme konnten in diesem Zeitrahmen gelöst werden.

Alternative 2: „Java Constructor Breakpoint“

Beschreibung: In diesem Task soll überprüft werden, ob und ggf. wie das *Java Debugging Interface* (JDI) zur Beschaffung von Debuginformationen genutzt werden kann.

geplante Zeit: 3 Tage

reale Zeit: Die geschätzte Zeit reichte nicht aus, da es an vielen Stellen Unklarheiten und Probleme gab, die beseitigt werden mussten. Insgesamt wurden daher 5 Tage benötigt. Dabei wurde die ursprüngliche Idee von Breakpoints in Konstruktoren verworfen. Stattdessen werden Ereignisse, insbesondere Methodenaufrufe, und da vor allem Aufrufe von Konstruktoren, abgefangen.

Implementieren der möglichen Alternativen für die Infobeschaffung

Beschreibung: Der Task hat sich ebenfalls wegen der zwei Infobeschaffungsalternativen in zwei unten beschriebenen Untertasks gesplittet.

geplante Zeit: 6 Tage

reale Zeit: Der Aufwand hat sich als erheblich schwieriger erwiesen.

Alternative 1: „-Xrunhprof Profiler“

Beschreibung: Die getestete Möglichkeit des Profiler-Einsatzes soll vollständig implementiert und an die Schnittstelle des Plug-Ins angebunden werden.

geplante Zeit: 3 Tage

reale Zeit: Der Aufwand war größer als eingeschätzt, da zusätzliche Probleme mit dem Verhalten des Dumpers unter Eclipse auftraten, für die zum Teil trotz intensiver Suche keine Lösung gefunden werden konnte (fehlende Variablenwerte im Dump). Daher wurden 2 zusätzliche Tage benötigt.

Alternative 2: „Java Constructor Breakpoint“

Beschreibung: Die Möglichkeiten der Informationsbeschaffung Ã¼ber das *Java Debugging Interface* (JDI) sollen vollständig implementiert und an die Schnittstelle des Plug-Ins angebunden werden.

geplante Zeit: 3 Tage

reale Zeit: Der Task war erheblich aufwändiger und einige Arbeiten konnten nicht abgeschlossen werden. Diese werden auf das vierte Release verschoben. Die bisher investierte Zeit liegt bei 5 Tagen.

16.2.3 Parametrisierung Layout-Algorithmen

Layout-Algorithmen „LinLog“ und „Dennis“ anpassen incl. „Control“

Beschreibung: Beide Algorithmen sollten angepasst werden, indem feste und frei wählbare Parameter festgelegt werden können. In der Control-Klasse sollte dann der Algorithmus übernommen werden, der ausgewählt wurde.

geplante Zeit: 1 Tag pro Algorithmus

reale Zeit: Zeitlich gut geschätzt. Es haben sich keine Schwierigkeiten ergeben.

GUI: für jeden Algorithmus ein Panel anbinden

Beschreibung: Es sollte für beide Algorithmen jeweils ein eigenes Control-Fenster mit frei wählbaren Parametern entwickelt werden, die eine Speicherung der Parameter erlaubt.

geplante Zeit: 2 Tage

reale Zeit: Der Aufwand war erheblich größer. 5 Tage wurden insgesamt gebraucht, da eigene Klassen(-entwürfe)/Interfaces undokumentiert waren. Es haben sich auch Probleme bei der Speicherung der WorkingCopy ergeben. Die Anbindung war generell sehr viel aufwändiger als vorher gedacht.

16.2.4 Animationen

Grafische Datenstruktur dynamischer gestalten

Beschreibung: Die Datenstruktur für Pfeile und Objekt-Visualisierungen soll überarbeitet werden, um dynamische Änderungen von grafischen Eigenschaften wie Größenänderungen zu ermöglichen. Hierzu müssen die Interfaces der Container geändert und die entsprechenden Implementierungen angepasst werden.

geplante Zeit: 3 Tage

reale Zeit: Der Aufwand hat sich als realistisch erwiesen. Problematisch waren Skalierungen auf 0, die bei gerichteten Pfeilen für unbenutzte Spitzen nötig waren. Als Lösung wurden die betroffenen Pfeilspitzen genügend klein skaliert, um in der Ansicht nicht sichtbar zu sein.

SnapshotTransitionVisualizer komplett neu implementieren

Beschreibung: Die Visualisierung von Snapshots soll effizienter und flexibler gestaltet werden. Aufgrund des großen Umfangs wurde dieser Task in die folgenden drei Tasks aufgeteilt.

geplante Zeit: 4 Tage

reale Zeit: Aus diesem Task, der die folgenden Tasks zusammenfasst, resultierte der endgültige Visualizer. Der Aufwandsanteil bezüglich des SnapshotTransitionVisualizers kann vom Rest der Arbeit nicht klar abgegrenzt werden.

Architektur und Funktionalität der Visualisierung überdenken

Beschreibung: Die genauen Ziele einer Verbesserung der Visualisierungs-Infrastruktur sollen erarbeitet werden.

geplante Zeit: 1 Tag

reale Zeit: Der Task konnte in der vorgesehenen Zeit bearbeitet werden.

Neue Hilfs-Datenstruktur für Visualisierung entwerfen und implementieren

Beschreibung: Es soll eine Datenstruktur geschaffen werden, die die im vorherigen Task formulierten Ziele effektiv unterstützt.

geplante Zeit: 3 Tage

reale Zeit: Der Task konnte in der vorgesehenen Zeit bearbeitet werden. Es traten keine Schwierigkeiten auf.

3D-Tooltips integrieren und Behaviours überarbeiten

Beschreibung: Die Tooltips und Behaviours in der Java3DView müssen an den neuen Visualizer angepasst werden.

geplante Zeit: 2 Tage

reale Zeit: Der Aufwand war geringer als geplant. Es stellte sich heraus, dass die Verwendung der Tooltips zu überdenken ist, da sie ein sehr schlechtes Größenverhalten aufweisen.

GUI-Elemente für Kontrolle der Animation

Beschreibung: Es sollen weitere Buttons zur Steuerung der Animation in der Java3DView angelegt werden.

geplante Zeit: 1 Tag

reale Zeit: Dieser Taks wurde nicht bearbeitet, da die technischen Voraussetzungen, um weitere Kontrollelemente sinnvoll einsetzen zu können, noch nicht realisiert sind.

Vom Benutzer wählbare Hintergrundgrafik

Beschreibung: Der Benutzer soll ein frei wählbares Bild als Hintergrundgrafik für die Java3DView einstellen können. Dadurch soll die Orientierung in dieser Ansicht erleichtert werden.

geplante Zeit: 2 Tage

reale Zeit: Der Aufwand war geringer, die benötigte Zeit lag bei 1 Tag. Allerdings wurde noch kein geeignetes Hintergrundbild, das wirkliche Vorteile bei der Orientierung bringt, gefunden.

16.3 Erweiterung der Infobeschaffung

Boris Brodski, Boris Düdder, Carina Klar, Michael Striewe

Bei den Arbeiten an den ersten beiden Releases hatte sich das Erzeugen des Debug-Snapshots als besonders zeitintensiver Teil des Programmablaufs herausgestellt. Deshalb wurde nach alternativen Möglichkeiten gesucht, um die komplexe Kapselungsstruktur des Eclipse-Debug-Interfaces zu umgehen und damit die Arbeit des Plug-Ins zu beschleunigen und mehr Informationen zu gewinnen.

Bei der Suche nach Alternativen erwiesen sich zwei Möglichkeiten als vielversprechend: der direkte Zugriff auf Debug-Informationen über das *Java Debug Interface* (JDI) sowie die Ana-

lyse von Heap-Dumps der Virtual Machine. Beide Möglichkeiten boten unterschiedliche Vor- und Nachteile in Bezug auf Laufzeit und Informationsgewinn, die im Folgenden näher erläutert werden. Daher wurden beide Ansätze parallel verfolgt. Zusätzlich wurde auch überprüft, inwieweit der bisherige *Recursive Snapshot Collector* optimiert werden kann und ob eine Kombination verschiedener Collectortechniken möglich ist.

Um eine schnelle Auswechslung der verwendeten Collectortypen zu gewährleisten, wurde ein Interface definiert, das die Einbindung verschiedener Beschaffungsalgorithmen ermöglicht. Die Einbindung weiterer Algorithmen wird über die Bereitstellung eines Extension-Points ermöglicht.

16.3.1 Definition des Interface

Das Interface bestimmt die möglichen Aufrufe an den Snapshot-Collector und definiert im Prinzip eine Zustandsmaschine mit fünf Zuständen (Abb. 16.1). Die Zustandsübergänge werden durch Methoden ausgelöst, die aufgrund von Nutzereingaben (z.B. Starten oder Beenden eines Launches) oder Debug-Events (z.B. am Breakpoint) von den zuständigen kontrollierenden Klassen aufgerufen werden.

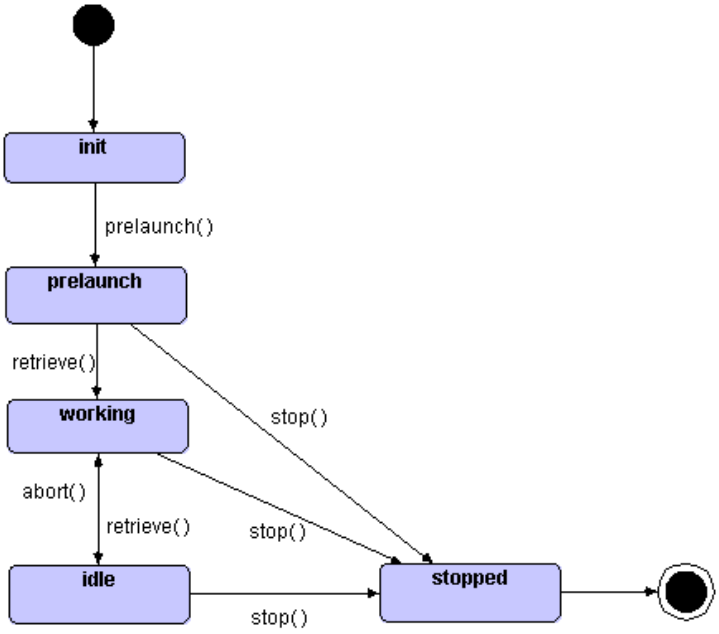


Abbildung 16.1.: Die Zustandsmaschine aus dem ISnapshotCollector

Ein Snapshot-Collector befindet sich nach der Instanziierung und der Ausführung seines Konstruktors im Zustand *init* und hat bisher nur interne Operationen ohne Auswirkung auf das

Plug-In ausgeführt. Über den Aufruf von `prelaunch()` wird er in den gleichnamigen Zustand versetzt und muss alle Operationen durchführen, die vor dem Start der Virtual Machine notwendig sind. Dazu gehören z.B. die Öffnung von Ports, das Setzen von Startparametern für die Virtual Machine oder das Erstellen von Dateien für Ausgaben. Die Methode bekommt den zu startenden Launch übergeben, um ggf. Änderungen an der Konfiguration vornehmen zu können.

Bei laufender Virtual Machine werden Snapshots über die Methode `retrieve()` angefordert, welche den Collector in den Zustand *working* versetzt. In diesem Zustand kommuniziert der Collector mit der Virtual Machine, um die gewünschten Informationen zu gewinnen, zu verarbeiten und als SnapshotInfo dem Plug-In zur Verfügung zu stellen. Die Methode sollte die beschafften Informationen entsprechend der Vorgaben der ChooseClassesView filtern und kann dazu auf die statische Methode `getClassByName()` der Klasse `Filter` zurückgreifen. Diese liefert als Rückgabe die benötigte DebugClass, falls die Klasse ausgewählt wurde oder `null`, falls die Klasse nicht ausgewählt wurde.

Nach Beendigung der Infobeschaffung und Rückgabe der SnapshotInfo oder durch explizites Aufrufen der Methode `abort()` wechselt der Collector in den Zustand *idle*. Jeder Collector muss die Methode `wantsToSuspend()` mit dem Rückgabewert true oder false implementieren, damit die steuernde Klasse feststellen kann, ob die Virtual Machine vor dem Aufruf von `retrieve()` angehalten und danach wieder gestartet werden muss oder nicht.

Aus jedem der drei Zustände *prelaunch*, *working* und *idle* heraus kann die Methode `stop()` aufgerufen werden, die den Collector beendet. Dazu gehört insbesondere, dass alle im Zustand *prelaunch* vorgenommenen spezifischen Einstellungen des Collector wieder rückgängig gemacht werden.

16.3.2 Einsatz des Java-Profilers

Der Einsatz des Java-Profilers (HPROF (2001)) wurde gewählt, da auf diese Weise ein sehr einfacher Zugriff auf alle erzeugten Objekte innerhalb des Java-Heaps möglich ist.

Über Start-Parameter der Virtual Machine wird eine Socket-Verbindung zum Plug-In eingerichtet, über die die Anforderung eines Heap-Dumps und die Entgegennahme der Rückgabe möglich ist. Dies erfolgt auf Basis des von SUN unter Public License bereitgestellten sogenannte NETDUMPERS (NetDumper (2004)). Der Heap-Dump enthält umfangreiche Informationen über Stacks und Stackframes, Klassen und Objekte sowie Felder, Variablenwerte und Objektbeziehungen enthält. Das Plug-In nutzt das binäre Dump-Format, welches von der Virtual Machine auf Anforderung als Stream über den Socket geschickt wird und vom NETDUMPER in einer temporären Datei gespeichert wird. Die Verbindung bleibt grundsätzlich während der gesamten Laufzeit der Virtual Machine bestehen und am Ende des Datenstroms des ersten Dumps wird kein Endzeichen gesendet. Das bedeutet, dass bei der Anforderung eines weiteren Dumps die Datei nur um entstandene Änderungen des Java-Heaps erweitert wird. Die Kodierung des Dumps ist leider nicht dokumentiert, aber der Dump kann über einen Hexeditor

betrachtet und in kleinen Ausschnitten durchaus manuell gelesen sowie durch den von SUN entwickelten Parser HAT (Foote (2004)) vollständig analysiert und ausgewertet werden.

Die grundsätzliche Aufgabe bei dieser von SUN zur Verfügung gestellten Technik bestand also nur darin, den NETDUMPER an die bestehenden Infrastruktur des Plug-Ins anzubinden und aus den Analysedaten von HAT einen Snapshot zu erzeugen.

Der Originalcode konnte tatsächlich weitgehend unverändert übernommen werden. Änderungen waren nur in den folgenden Bereichen nötig:

- `NetDumper` Um das Abarbeiten mehrerer Breakpoints zu ermöglichen, musste der Dumper so umgestaltet werden, dass Ausgaben der Virtual Machine in eine Datei geschrieben werden, die zur Erzeugung des Snapshots sofort ausgewertet werden kann, an die aber auch für spätere Snapshots noch Daten angehängt werden können. Bei ordnungsgemäßer Beendigung des Collectors wird diese Datei automatisch wieder entfernt. Bei Programmfehlern kann diese Datei allerdings unerwünschter Weise ungelöscht bleiben und zu Fehlern führen.
  Da ein Dump kein Endzeichen besitzt, interpretiert das Plug-In nach dem Anfordern eines Dumps eine Zeitspanne von 10 Sekunden ohne Datenempfang als Zeichen dafür, dass alle Daten des Dumps gesendet wurden. Dies kann allerdings nicht garantiert werden, wenngleich bisher während der Übertragung maximal Lücken von 3 Sekunden ausgetreten sind. Während dieser 10-sekündigen Wartephase kann das Plug-In keinen weiteren Dump anfordern, der eine Wiederaufnahme des Datenstroms bedeuten würde.
- `Parser` Das Auslesen und Verarbeiten mehrerer aufeinanderfolgender Snapshots wurde vom Parser zwar theoretisch, aber nicht tatsächlich unterstützt, da wichtige Daten übersprungen wurden. Zwei kleine Korrekturen im Programmcode konnten diesen Fehler leicht beheben.

Die Virtual Machine wird bei Nutzung dieses Snapshot-Collectors mit den Parametern `-Xrunhprof:net=localhost:7000,format=b,depth=100,doe=n` gestartet, nachdem der NetDumper auf Port 7000 den benötigten Socket geöffnet hat. Bei ordnungsgemäßer Beendigung des Collectors werden diese Parameter automatisch wieder entfernt, um die Lauffähigkeit anderer Collectortypen nicht zu behindern. Bei Programmfehlern können die Parameter allerdings unerwünschter Weise ungelöscht bleiben und zu Fehlern führen.

Es stellte sich allerdings heraus, dass zwei zentrale Arten von Informationen nicht über den Dump gewonnen werden können. Aus den Daten der Stacks und Stackframes lassen sich zwar zu jedem Objekt Informationen über die aufrufende Klasse, Methode und Programmzeile gewinnen, jedoch nicht über die konkrete Instanz dieser Klasse. Damit ist die Darstellung von Erzeugungsbeziehungen nicht möglich.

Ferner werden Arrays aus benutzerdefinierten Klassen keine korrekten Objekttypen zugeordnet. Die benötigten Informationen sind zwar im Dump enthalten und über einen Hex-Editor auffindbar, jedoch scheint bereits die Virtual Machine die Zuordnung von Klassen-IDs falsch

vorzunehmen. Dieses Problem wurde teilweise dadurch überbrückt, dass bei nicht-leeren Arrays der Objekttyp aufgrund der enthaltenen Elemente bestimmt werden kann. Diese Methode ist allerdings nicht zuverlässig und schlägt bei leeren Arrays fehl. Nicht identifizierte Arrays werden dann als „Instance of Unknown class“ behandelt und vom Filter entfernt.

Während diese beiden Fehler immer beim Einsatz des Profilers und von HAT auftreten und daher vermutlich auf Format und Inhalt des erzeugten Dumps zurückzuführen sind, tritt ein weiteres gravierendes Problem nur bei Erzeugung des Dumps über Eclipse auf: sämtliche Werte primitiver Variablen sind in diesen Dumps leer bzw. 0. Beim Aufruf über die Konsole wird dagegen für identischen Programmcode bei identischen Startparametern der Virtual Machine ein vollständiger Dump ausgegeben. Dieser Fehler tritt unabhängig davon auf, ob mehrere Virtual Machines aktiv sind sowie unabhängig von der eingesetzten Virtual Machine. Getestet wurden die Standard-VMs von JDK 1.5, SunJava 1.4.2, SunJava 1.5 sowie die Java HotSpot Client VM (build 1.4.2_03-b02). Während die beiden 1.5-Maschinen wie oben beschrieben keine korrekten Werte lieferte, erlaubte Eclipse bei den 1.4-Maschinen gar keinen Start, da keine Debug-Connection eingerichtet werden konnte. Vermutlich verwendet Eclipse an dieser Stelle einen nicht konfigurierbaren Aufruf der Virtual Machine, denn die selbe Virtual Machine lässt sich über die Konsole problemlos starten und liefert korrekte Ergebnisse. Dies wird auch deshalb wahrscheinlich, weil alle Maschinen beim Start über die Konsole einige (unerhebliche) Fehlermeldungen des Profilers ausgeben, während 1.5-Maschinen unter Eclipse keinerlei Ausgabe erzeugen. Bei 1.4-Maschinen ist die Fehlerausgabe unter Eclipse dagegen länger.

16.3.3 Einsatz des Java Debug Interface

Das *Java Debug Interface* (JDI) ist das native Interface, welches auch von Eclipse zum Debuggen von Java-Code verwendet.

Die Integration und Verwendung dieser Komponente erfolgt über die Implementation von Listenern. Diese werden bei zuvor definierten Events aufgerufen. Die Eventschleife befindet sich auf der Seite von Eclipse, somit muss eine Debugging-Komponente alle Listener bei Eclipse anmelden.

Das Protokoll beinhaltet das Setzen von Breakpoints beim Methodenaufruf. Dies erlaubt die Verfolgung von Objekterzeugung innerhalb der Runtime. Da dieses Feature aber nur über einen Klassenfilter verfügt, muss man beim Aufruf eines entsprechenden Listeners überprüfen, ob es sich um den Aufruf eines Konstruktors, in Java Terminologie $< init >$, handelt.

Diese Methode erlaubt die Überwachung von Objekterzeugungen mit jeweiligen Beteiligten: Klasse sowie Objekt, Methode, Programmzeile wo das Objekt erzeugt wurde. Zusätzlich erhält man eine Auflistung der Attribute sowie derer Werte.

Leider hat dieses Verfahren folgende gravierende Nachteile:

1. Die Verfolgung aller Aufrufe benötigt enorm viel Zeit verglichen mit der reinen Ausführungszeit. Die Verwendung eines geschickt gewählten Filters könnte bei kleiner Auswahlmenge von überwachten Klassen die Verfolgungszeit in akzeptable Dimensionen bringen.
2. Es scheint keine Möglichkeit zu existieren die Erzeugung von Array-Objekten zu überwachen. Die Snapshots besitzen ohne Arrays keinen hohen Informationsgehalt und können zu Verwirrungen führen.

Diese Nachteile haben uns dazu bewogen, die JDI Methode nicht für den Snapshot Collector zu verwenden. Sie ist nur sinnvoll einsetzbar zur zusätzlichen Informationsgewinnung.

16.3.4 Rekursive Snapshot-Beschaffung

Die in den ersten beiden Releases eingesetzte Art der Infobeschaffung wurde für dieses Release mit der Bezeichnung „Recursive-Snapshot-Collector“ versehen, da sie rekursiv auf dem von Eclipse zur Verfügung gestellten Objektbaum arbeitet. In diesem Baum kapselt Eclipse Debug-Informationen über Threads, Stacks, Stackframes und die darin enthaltenen Objekte mit ihren Variablenwerten. Für alle Threads und alle darin enthaltenen Stacks und Stackframes werden die gefundenen Objekte betrachtet und in den Snapshot eingefügt, sofern sie zu einer vom Nutzer ausgewählten Klasse gehören. Für alle von einem Objekt referenzierten weiteren Objekte wird dieses Verfahren rekursiv aufgerufen. Um eine Mehrfachbehandlung von Objekten zu verhindern, werden alle IDs von gefundenen Objekten gespeichert. Durch die Größe des Baumes und die offensichtlich nicht sehr effiziente Kapselung durch Eclipse ist ein vollständiger Baumdurchlauf für große Programme allerdings praktisch nicht durchführbar. Deshalb bricht die Rekursion ab, sobald sie auf ein Objekt trifft, welches zu einer vom Nutzer nicht ausgewählten Klasse gehört. Dies führt insbesondere dann zu Problemen, wenn eine häufig auftretende Klasse nicht gewählt wurde oder ein großer Teilbaum von einer nicht gewählten Klasse abhängt.

Doch selbst bei Auswahl aller Klassen durch den Nutzer und der entsprechend langen Laufzeit kann das Verfahren nicht die Abbildung aller instanziierten Objekte garantieren, da nicht alle Objekte im Objektbaum zu finden sind, was z.B. insbesondere die Erkennung von Exceptions unmöglich macht. Zudem stehen auch keine Informationen über Vererbung zur Verfügung, so dass abgeleitete Exceptionklassen ohnehin nicht erkannt werden können.

Ferner sind außer Assoziationen keinen anderen Objektbeziehungen erkennbar, so dass z.B. Erzeugungsbeziehungen nicht abgerufen werden können. Damit stellt diese Art der Infobeschaffung nur einen Teil der insgesamt gewünschten Informationen zur Verfügung.

16.3.5 Kombination der Möglichkeiten

Um die Schwachstellen der einzelnen Collectortypen zu kompensieren, wurde darüber nachgedacht, mehrere Ansätze zu kombinieren. Tabelle 16.1 zeigt, welche Funktion die einzelnen

Collectoren erfüllen. Wie zu erkennen ist, würde die Kombination des `DumpSnapshotCollector` mit dem `JDISnapshotCollector` praktisch allen Anforderungen genügen, wobei der Nutzer auf eine Übereinstimmung der Objekt-IDs mit den Bezeichnungen von Eclipse verzichten müsste, da beide Collectortypen für Objekte eigene IDs vergeben, die nicht auf die Bezeichnungen des Eclipse-Debuggers abgebildet werden können.

Funktion	`RecursiveSnapshotCollector`	`DumpSnapshotCollector`	`JDISnapshotCollector`
alle Objekte	×	√	√
„created“-Beziehung für Objekte	×	×	√
Arrayerkennung	√	(√)	×
Exceptionerkennung (auch abgeleitete)	×	√	?
Ursprung von Exceptions	×	×	√
primitve Variablenwerte	√	(×)	√
IDs analog zum Eclipse-Debugger	√	×	×

Tabelle 16.1.: Vergleich der Snapshot-Collectoren

Die tatsächliche Zusammensetzung eines neuen Collectors aus mehreren Typen wurde in diesem Release aus Zeitgründen nicht realisiert. Nach einer Aufwandsabschätzung und genaueren Studie zur praktischen Durchführbarkeit wird dies möglicherweise im Endrelease umgesetzt. Es steht noch nicht fest, ob über die IDs die Äquivalenz von Objekten aus verschiedenen Collectoren festgestellt werden kann und somit Informationen überhaupt kombiniert werden können.

16.4 Parametrisierung

Jonas Mathis

Um dem Nutzer Freiheiten bei der Anordnung der Objekte in der 3D-Ansicht zu geben, sollen einige Parameter der verschiedenen Anordnungsalgorithmen individuell einstellbar sein. Zu diesem Zweck ist in den Launch-Optionen des Plug-Ins ein Panel vorgesehen, über das die entsprechenden Einstellungen vorgenommen werden können.

Zunächst musste eine gemeinsame Schnittstelle geschaffen werden, auf die die vorhandenen Algorithmen dann angepasst wurden. Zudem sind weitere Algorithmen auf diese Weise leicht integrierbar.

16.4.1 Definition des Interfaces

Aus dem letzten Release stammen die Algorithmen „Linlog“ und „Force-directed Layout“. Diese wurden damals ohne Festlegung einer gemeinsamen Schnittstelle implementiert, was aber nun aus Gründen der Benutzbarkeit und vor allem der Erweiterbarkeit geändert wurde. In der Planungsphase ist eine neue Struktur für die Layout-Algorithmen entworfen worden, wobei insbesondere die Angabe von Parametern berücksichtig wurde.

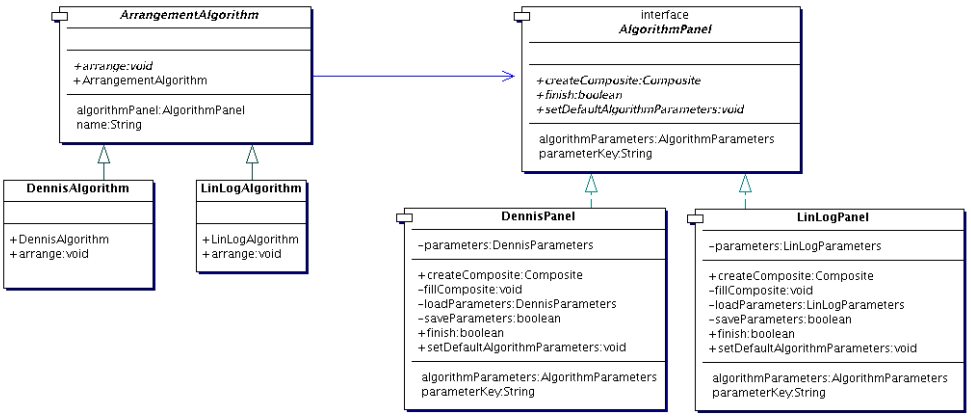


Abbildung 16.2.: `ArrangementAlgorithm` und `AlgorithmPanel`

Wie aus Abbildung 16.2 ersichtlich, wurde für die Layout-Algorithmen die abstrakte Oberklasse `ArrangementAlgorithm` geschaffen, von der die beiden Algorithmen `DennisAlgorithm` und `LinLogAlgorithm`, sowie alle möglichen weiteren Entwicklungen, erben. Den Festlegungen in der Oberklasse zufolge, verfügt ein Algorithmus über einen Namen und ein eigenes Panel, um ein Dialogfeld für die Eingabe der Parameter bereitzustellen. Die ursprüngliche Idee, für die „AbortConditions“ ebenfalls ein Interface bereitzustellen, wurde nach einer Aufwands- und Nutzenschätzung doch nicht in die Tat umgesetzt. Stattdessen wurden die Abbruchbedingungen für die Algorithmen in den jeweiligen „arrange“-Methoden implementiert. So werden die bereits in Release 2 vorhandenen `AbortConditions` weiterverwendet und den Anforderungen des Algorithmus gemäß verschachtelt, bevor die eigentliche „arrange“-Methode des Algorihmus aufgerufen wird. Um diese Abbruchbedingungen dennoch variabel zu gestalten, wurden kritische Werte (z.B. Anzahl Iterationen) als Parameter für die Algorithmen übernommen.

Das Interface `AlgorithmPanel` stellt die Schnittstelle für ein Dialogfeld bereit, so daß ein jeder Algorithmus über einen Mechanismus verfügt, mit dessen Hilfe er die benötigten Parameter einlesen kann. Eine Eingabe der Parameter kann erfolgen, wenn im Tab „3Debug“ des „Run“-Dialoges der Button „Settings...“ gedrückt wird. Diese Parameter werden in der „LauchConfiguration“ gespeichert, damit sie für weitere Läufe des Algorithmus zur Verfügung stehen. Die Methode „createComposite“ sorgt jeweils dafür, daß die Dialogelemente gezeichnet und mit den bis dato verwendeten Werten gefüllt werden. Sollten noch keine Werte

feststehen, so werden Default-Werte verwendet. In der Methode „finish“ sollten die Werte für die Parameter in die aktuelle „LaunchConfiguration“ übertragen werden. Bei der Implementierung dieses Verhaltens kam es allerdings zu Schwierigkeiten, da im Dialogfeld geänderte Werte zuerst von Eclipse nicht in der „LaunchConfiguration“ gespeichert wurden. Der Workaround bestand darin, in der Methode „performApply(LaunchConfigurationWorkingCopy)“ des „LaunchConfigurationTab3d“ die Parameter noch einmal aus der „WorkingCopy“ auszulesen und erneut hineinzuschreiben. Nun werden die Parameter korrekt gespeichert und stehen beim nächsten Aufruf zur Verfügung. Das Panel verfügt neben einem „Ok“- und „Cancel“-Button auch über einen „Default“-Button, um die Standardwerte für den Algorithmus wiederherzustellen.

16.4.2 Force-directed Layout

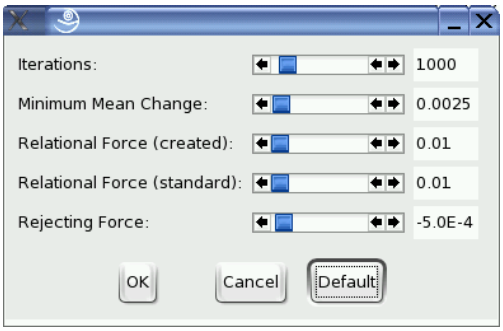


Abbildung 16.3.: Das Panel für den Force-directed Algorithmus

In Abbildung 16.3 wird das Panel zur Eingabe der Parameter für den Force-directed Algorithmus dargestellt. Durch die ersten beiden Werte, „Iterations“ und „Minimum Mean Change“ wird das Abbruch-Kriterium vorgegeben, d.h. der Anordnungsalgorithmus stoppt, sobald eine bestimmte Anzahl Iterationen erreicht ist, oder der über alle Objekte gemittelte Abstand sich zwischen zwei Iterationen kaum mehr verändert hat. Die anderen drei Parameter geben die eigentlichen Kräfte an, die bei der Berechnung verwendet werden sollen. Informationen zu Wertebereichen und Standard-Werten finden sich in Tabelle 16.2. Die Default-Werte wurden durch Ausprobieren gefunden.

Bezeichnung	min.	max.	default
Iterations	0	10000	1000
Minimum Mean Change	0,0	1,0	0,0025
Relational Force (created)	0,0	1,0	0,01
Relational Force (standard)	0,0	1,0	0,01
Rejecting Force	0,0	-0,01	-0,0005

Tabelle 16.2.: Die Parameter für „Force-directed“

16.4.3 Linlog

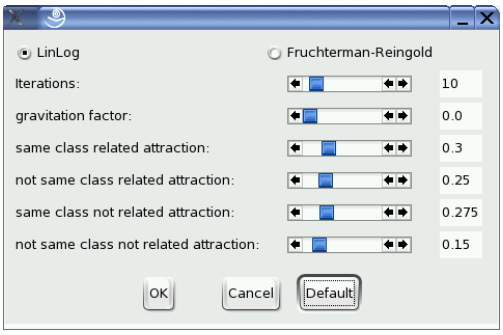


Abbildung 16.4.: Das Panel für den Linlog Algorithmus

Das bei der Verwendung von LinLog erscheinende Panel zur Eingabe der Parameter ist in Abbildung 16.4 dargestellt. Per Radio-Button erhält der Nutzer die Wahl zwischen den beiden Varianten „LinLog“ und „Fruchtermann-Reingold“. Das einzige Abbruchkriterium für den Algorithmus ist die Anzahl der Iterationen, so daß man durch die Festlegung eines Wertes maßgeblich die Laufzeit der Anordnung beeinflussen kann. Außerdem gibt es noch die Möglichkeit, Werte für die Gravitation der anzuordnenden Objekte, sowie verschiedene Anziehungskräfte zwischen anders- und gleichartigen Objekten festzulegen. Dies kann durch die direkte Eingabe des gewünschten Wertes oder mittels der „Slider“ geschehen, für deren korrektes Arbeiten intern auch der `SliderListener` verwendet wird. Die Default-Werte entsprechen den von den Entwicklern des Algorithmus empfohlenen Werten, siehe Tabelle 16.3.

Bezeichnung	min.	max.	default
Iterations	0	100	10
gravitation factor	0,0	1,0	0,0
same class related	0,0	1,0	0,3
not same class related	0,0	1,0	0,25
same class not related	0,0	1,0	0,275
not same class not related	0,0	1,0	0,15

Tabelle 16.3.: Die Parameter für „LinLog“

16.5 Refactoring der Visualisierungen

Daniel Maliga, Daniel Vogtland, Sebastian Vastag

Im vorherigen Release wurden alle grafischen Objekte für jeden Visualisierungsvorgang neu erzeugt, wie auch die Visualizer-Klassen selbst. Die Visualisierungen wurden also vollständig unabhängig voneinander behandelt. Dadurch bedingt wurde in der `Java3DView` bei Übergängen zwischen Snapshot-Visualisierungen, z.B. bei Animationen, stets eine Zeit lang nur eine leere Fläche angezeigt. Außerdem wurden die Visualisierung einzelner Snapshots und die Animation von Snapshot-Übergängen in zwei verschiedenen Klassen (`SnapshotVisualizer` und `SnapshotTransitionVisualizer`) realisiert. Darüber hinaus erscheint eine Zusammenlegung beider aufgrund Ähnlichkeiten der Funktionalität wünschenswert. Daher wurde die Snapshot-Visualisierung für das dritte Release neu konzipiert und implementiert. Sie wird nun von der Klasse `Visualizer` realisiert.

Im folgenden wird diese Neuimplementierung beschrieben und der Ablauf einer Visualisierung anhand eines Beispiels erläutert.

Außerdem wurde zwecks Verbesserung der Orientierung im 3D-Fenster ein Hintergrund eingefügt. Darauf wird am Schluss dieses Abschnitts näher eingegangen.

16.5.1 Struktur des Visualisierungsbereiches

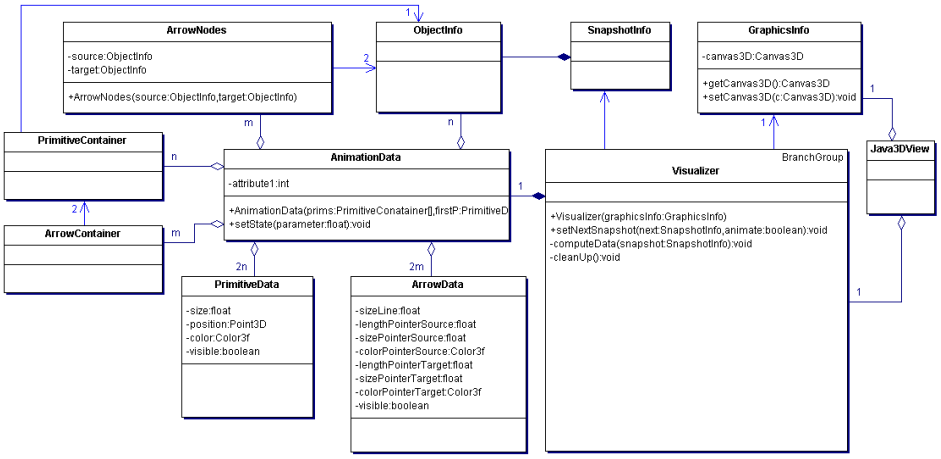


Abbildung 16.5.: Die Klassenstruktur des Visualisierungsbereiches

Das Klassendiagramm in Abb. 16.5 stellt die Struktur des Visualisierungsbereiches dar. Die `Java3DView` erzeugt während ihrer Initialisierung eine Instanz der neuen Klasse `Graphics-`

`Info`, in der Referenzen auf wichtige Java3D-Objekte wie das verwendete `Canvas3D` gehalten werden. Anschließend wird ein `Visualizer`-Objekt erzeugt, wobei ihm das `GraphicsInfo`-Objekt übergeben wird. Die `Java3DView` verwendet während ihrer gesamten Lebenszeit dasselbe `Visualizer`-Objekt.

Der `Visualizer` arbeitet ähnlich einem *Stream*. Initialisiert mit einem leeren Snapshot, wird bei jedem Aufruf der Methode `setNextSnapshot` der jeweils übergebene Snapshot visualisiert, wobei auch ein animierter Übergang von der aktuellen Visualisierung möglich ist.

Zu jedem Zeitpunkt wird vom `Visualizer` genau ein Objekt der Klasse `AnimationData` gehalten. Diese stellt in erster Linie die benötigten Informationsobjekte zur grafischen Darstellung zweier Snapshots bereit. Im einzelnen sind dies die `ObjectInfo`-Objekte (aus `SnapshotInfo`-Objekten extrahiert), `ArrowNodes`-Objekte, die eine grafische Verbindung zwischen zwei Objekten darstellen (sie fassen also ggf. mehrere `RelationInfo`-Objekte zusammen), `PrimitiveContainer`-Objekte zur Visualisierung je einer `ObjectInfo`, `ArrowContainer`-Objekte zur Darstellung je eines `ArrowNodes`, zu jedem `PrimitiveContainer` zwei `PrimitiveData`-Objekte, sowie zu jedem `ArrowNodes`-Objekt zwei `ArrowData`-Objekte. Ein `PrimitiveData`-Objekt umfasst die Ausprägung der grafischen Attribute des entsprechenden `PrimitiveContainer`-Objekts und außerdem ein `visible`-Flag. Pro `PrimitiveContainer` werden ein Startzustand (aktuelle Visualisierung) und ein Endzustand (zu visualisierender neuer Snapshot) in je einem `PrimitiveData` gespeichert. Die Behandlung der `ArrowData`-Objekte erfolgt analog dazu.

Alle notwendigen Daten für ein `AnimationData` werden in dessen Konstruktor übergeben und können auch nicht mehr mittels add-/remove-Methoden aktualisiert werden. Wie leicht im Klassendiagramm (Abb. 16.5) zu erkennen ist, werden Listen von Objekten/Relationen und deren grafische Umsetzungen verwendet. Diese werden intern unabhängig voneinander verwaltet, jedoch werden globale Indexpositionen benutzt. Eine nachträgliche Veränderung einzelner Daten wäre also mit relativ hohem Aufwand verbunden, der nicht gerechtfertigt wäre, da es sich bei `AnimationData` in erster Linie um eine Hilfsklasse zur Zusammenfassung mehrerer Datentypen handelt. Aufgrund dessen findet auch jegliche Validierung der Daten im `Visualizer` statt. Neben der Funktionalität als „Datenspeicher“ stellt `AnimationData` außerdem die Methode `setState` bereit, die die grafischen Eigenschaften der Container abhängig vom Wert des Parameters zwischen Start- und Endzustand interpoliert.

16.5.2 Ablauf einer Visualisierung

Die Kernmethode der Klasse `Visualizer` ist `setNextSnapshot`. Um den jeweils nächsten Snapshot bzw. den Übergang zu diesem zu visualieren, werden zunächst Informationen über Objekte und Relationen aus der übergebenen `SnapshotInfo` extrahiert und mit den Informationen aus dem aktuellen `AnimationData` verglichen. Grafische Elemente (`Primitive-` und `ArrowContainer`) werden ggf. neu erzeugt, oder falls bestehend übernommen.

Anschließend wird ein neues `AnimationData`-Objekt erzeugt, welches Objekte und Relationen, deren entsprechende Container- und Data-Objekte des neuen Snapshots wie auch der

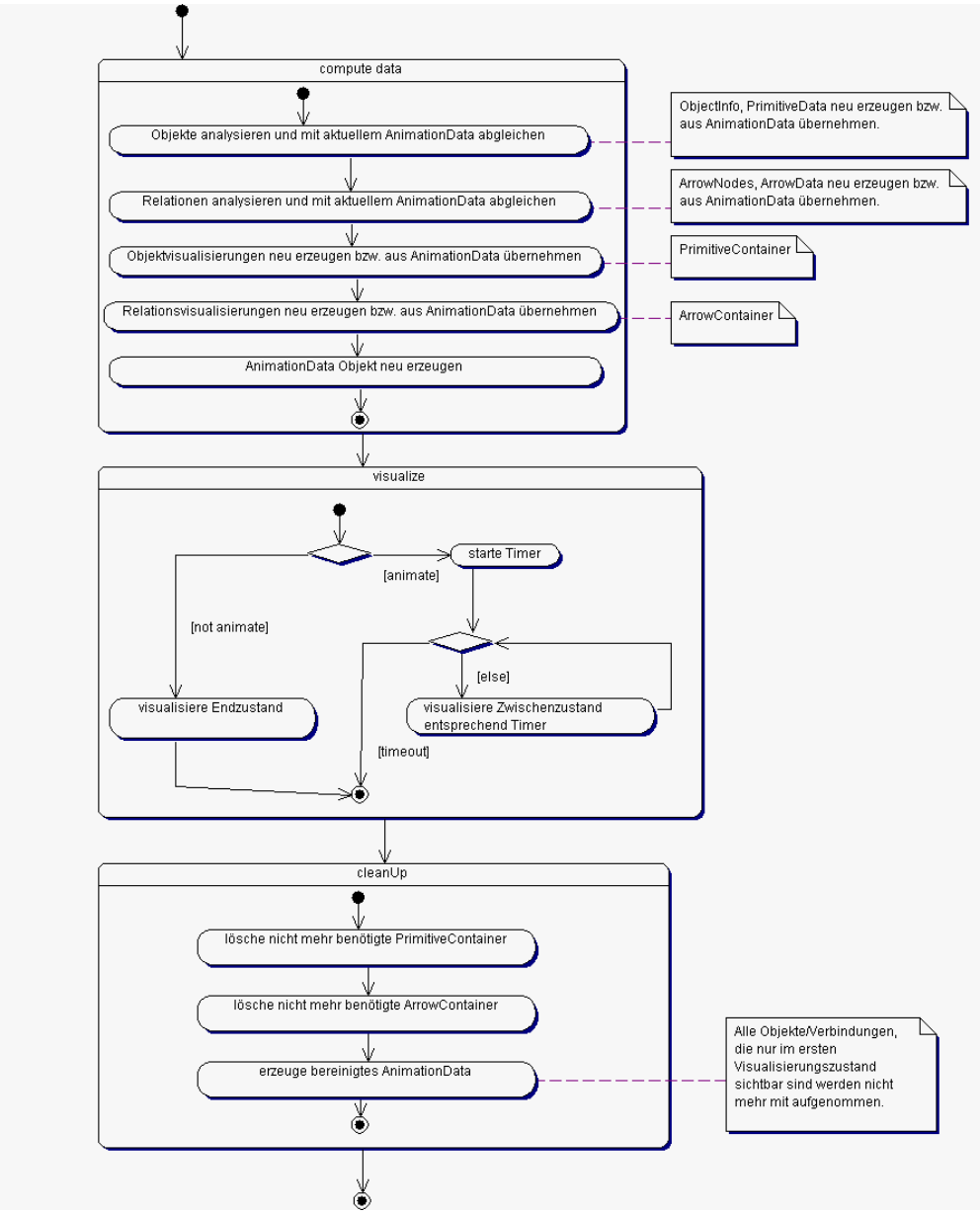


Abbildung 16.6.: Ablauf der Methode `setNextSnapshot`

vorherigen Visualisierung beinhaltet. All diese Funktionalität wird durch die private Methode `computeData` realisiert.

Nun wird mit Hilfe des `AnimationData` der neue Snapshot visualisiert – entweder als animierter Übergang mit festgelegter Zeitdauer vom alten Snapshot oder direkt. Im ersten Fall wird die Animation durch Darstellung von Zwischenzuständen, die mittels Interpolation berechnet werden, realisiert. Ansonsten wird auf die Animation verzichtet und der neue Snapshot sofort dargestellt.

Nach Abschluss der Visualisierung wird das `AnimationData` durch eine „bereinigte“ Version ausgetauscht, in der Objekte und Relationen, die im jetzt aktuellen Snapshot nicht mehr enthalten sind, entfernt wurden. Der genaue Ablauf ist in Abb. 16.6 als Aktivitätsdiagramm dargestellt.

Um das Verständnis des Ablaufs zu erleichtern, wird im Folgenden die Veränderung eines `AnimationData`-Objekts im Verlauf der Visualisierung anhand des in Abb. 16.7 gezeigten Beispiels erläutert.

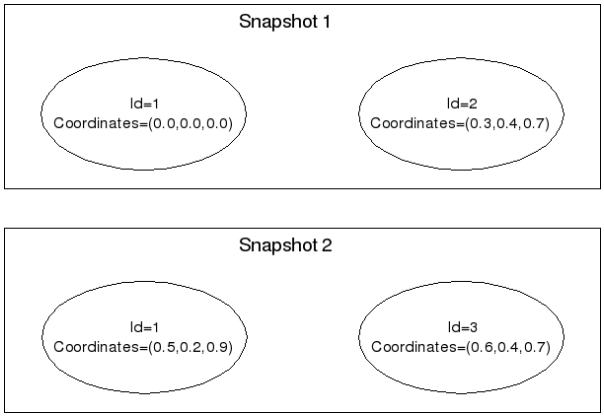


Abbildung 16.7.: Beispiel: Zwei zu visualisierende Snapshots

Dieses Beispiel stellt zwei Snapshots dar. Der Einfachheit halber werden wir uns auf Koordinaten und ID als Attribute eines Objekts beschränken. Im ersten Snapshot sind zwei Objekte vorhanden, von denen nur eines auch im zweiten Snapshot vorhanden ist; allerdings kommt ein neues Objekt hinzu. Das verbliebene Objekt (id=1) hat jedoch veränderte Koordinaten.

Nehmen wir an, der `Visualizer` wäre gerade erst initialisiert worden. Dies entspricht dem Setzen eines leeren Snapshots. Nun rufen wir `setNextSnapshot` mit dem ersten Beispiel-Snapshot auf. Nach dem vollständigen Durchlauf der Methode resultiert das im Objektdiagramm 16.8 dargestellte `AnimationData`.

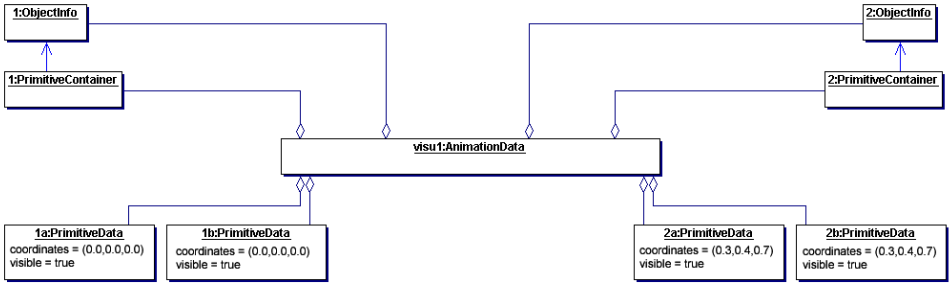


Abbildung 16.8.: Beispiel: Objektdiagramm nach Setzen des ersten Snapshots

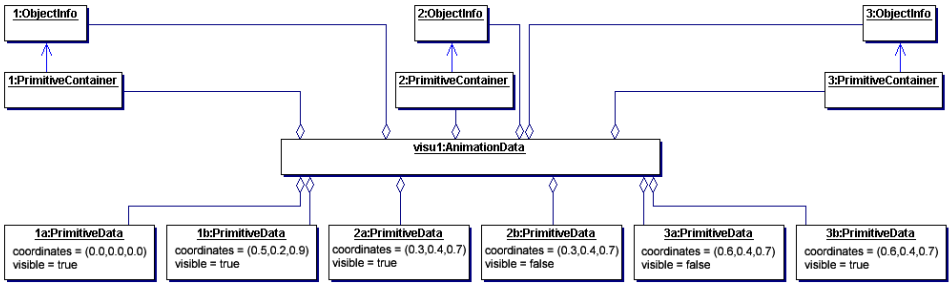


Abbildung 16.9.: Beispiel: Objektdiagramm während des Setzens des zweiten Snapshots

Nun rufen wir nochmals `setNextSnapshot` auf, diesmal mit dem zweiten Snapshot als Argument. Nachdem die Untermethode `computeData` abgeschlossen ist, sind die neuen Objekte vereint mit den Informationen aus dem vorherigen `AnimationData` sowie Koordinatenänderungen in einem neuen `AnimationData` enthalten, wie in Abb. 16.9 zu sehen.

Nach erfolgter Visualisierung werden die Objekte und zugehörigen Informationen, die nicht Bestandteil des übergebenen Snapshots sind, entfernt und so ein „bereinigtes“ `AnimationData` erzeugt, das in Abb. 16.10 dargestellt ist. Dies wird verwirklicht, indem im zweiten `Primitive-` bzw. `ArrowData` nach auf `false` gesetzten `visible`-Flags gesucht wird (linearer Listendurchlauf).

Die Methode `setNextSnapshot` wird durch die `Java3DView` aufgerufen. Um den Ablauf der Methodenaufrufe zu verdeutlichen, folgen zwei Sequenzdiagramme (aus Platzgründen wurden die grafischen Container-Objekte vernachlässigt). Das erste Sequenzdiagramm (Abb. 16.11) zeigt die Initialisierung des `Visualizer`-Objekts sowie das Setzen eines einzelnen Snapshots ohne Animation.

Im zweiten Diagramm (Abb. 16.12) wird das Setzen eines Snapshots ohne Animation, gefolgt vom Setzen eines zweiten Snapshots mit animiertem Übergang von der vorherigen Visualisierung dargestellt.

16.5.3 Hintergrundbild

Zur besseren Orientierung und vor allem zur optischen Aufwertung des Debuggers wurde die Möglichkeit geschaffen, einen Hintergrund in die Java3DView einzublenden.
Um die eigentliche Szene herum, bestehend aus Kugelobjekten und Pfeilen, wird dazu eine Sphere als spezielles Background-Objekt gelegt. Auf die Innenseite der an sich transparenten Kugel wird eine Textur in Form eines Bildes gelegt, welches der Benutzer in der LaunchConfiguration frei wählen kann. Dieses Bild sollte jedoch einige technische Randbedingungen erfüllen:

- JPEG-Format
- Das Bild sollte die Projektion eines Kugelpanoramas auf eine rechteckige Fläche sein (ca. 3:4-Format). Dies sind z.B. Landkarten in Mercatorprojektion (die Polarregionen der Erde erscheinen langgestreckt). Andere Bildformate werden ebenfalls akzeptiert, sehen nur nicht besonders gut aus.
- die Bildgröße in Pixeldimensionenen sollte nicht zu groß sein, da das JPEG entpackt und komplett in den Texturspeicher der Grafikkarte geladen werden muss. Eine Größe von 640*480 Pixel hat sich als praktikabel erwiesen. Um schwache Rechner zu entlasten muss der Hintergrund explizit aktivert werden.

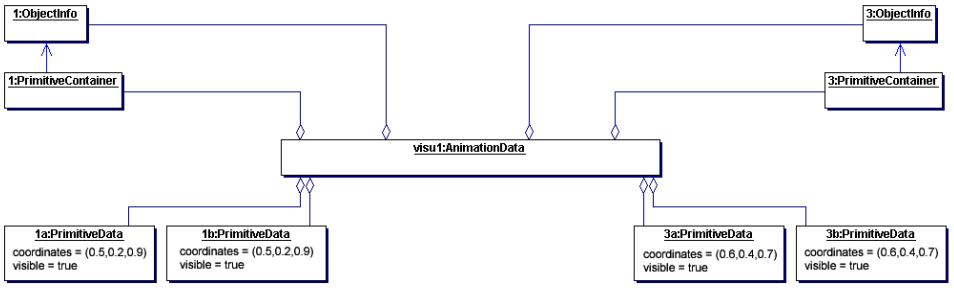


Abbildung 16.10.: Beispiel: Objektdiagramm nach Setzen des zweiten Snapshots

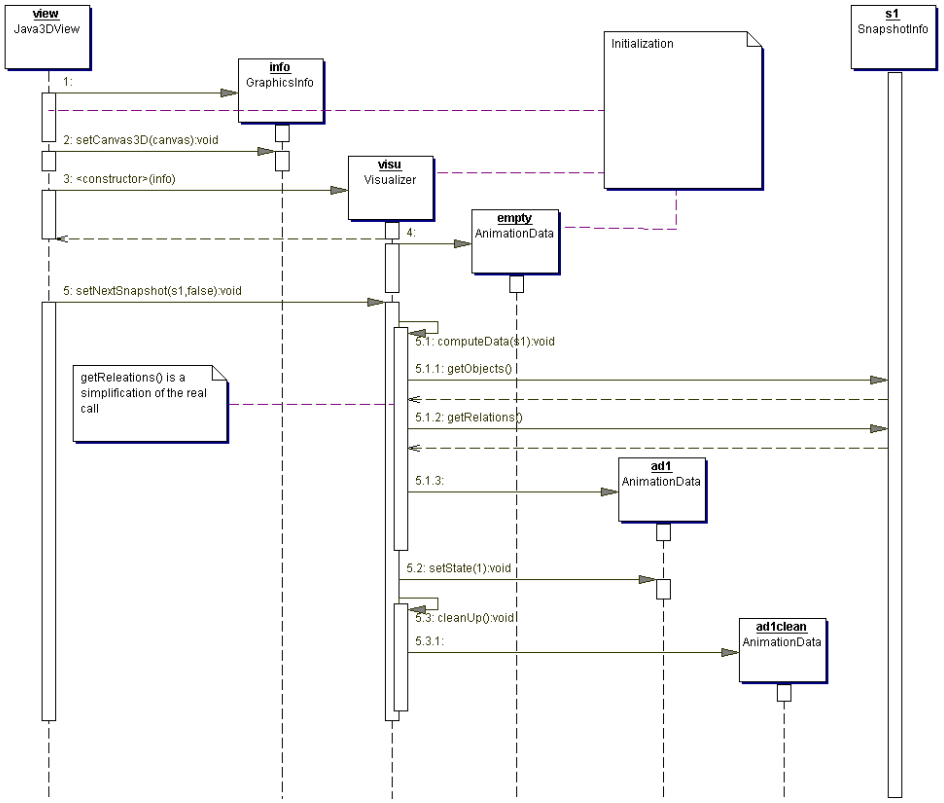


Abbildung 16.11.: Setzen eines Snapshots

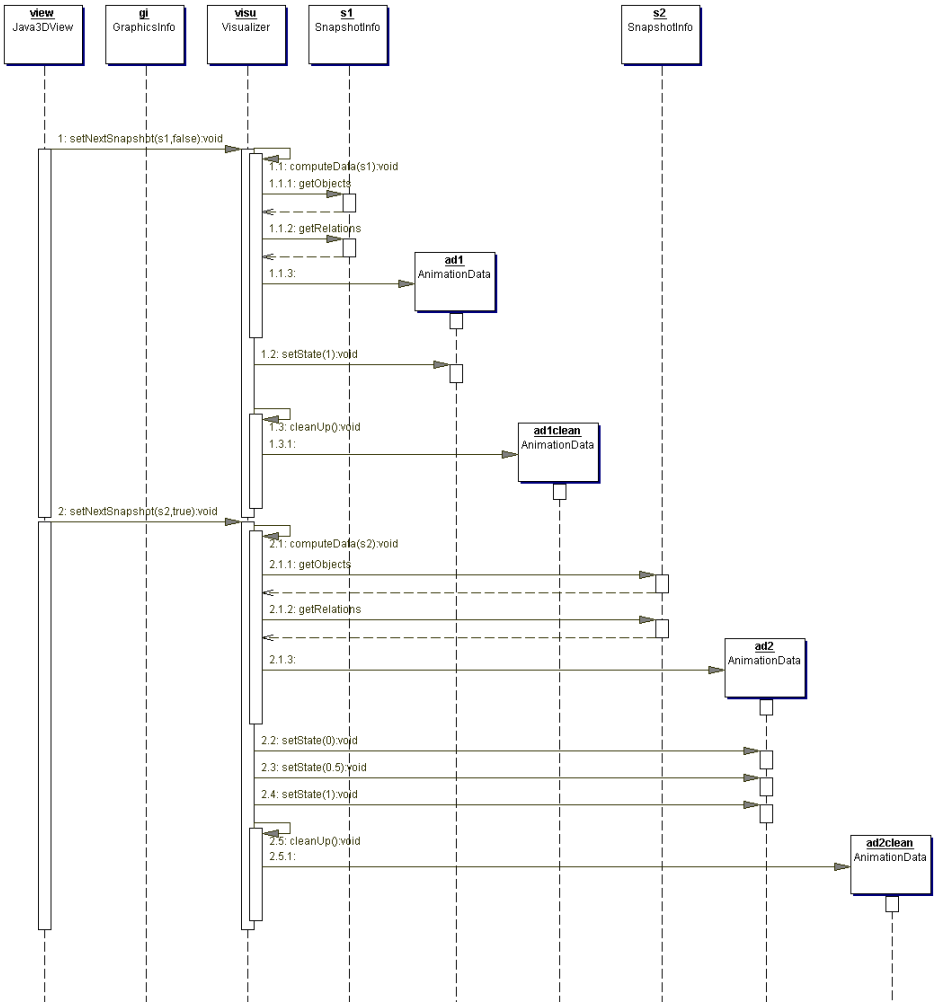


Abbildung 16.12.: Animation eines Snapshot-Übergangs

Die Kugel im Hintergrund ist an die TransformGroup der Szene gebunden. Drehen sich die Objekte der Szene vor dem Bildauschnitt des Betrachters, so dreht sich auch die Sphere mit. Dadurch findet der Benutzer Objekte immer an derselben Stelle des Hintergrundes wieder, was die Orientierung erleichtert. Ungeeignete Hintergrundbilder können allerdings die Orientierung wiederum erschweren, wenn ein bei Rotation oder Bewegung stark variierender Hintergrund die Aufmerksamkeit des Benutzers bindet. Besonders gut unterstützt wird die räumliche Orientierung durch Bilder, die mittels eines Horizontes eine eindeutige Teilung des 3D-Raumes angeben.

16.6 Releaseabnahme

Carina Klar, Daniel Maliga, Michael Striewe

Da in diesem Release die Kundenrolle nicht besetzt war, konnten auch keine auf User-Stories aufbauende Akzeptanztests durchgeführt werden. Stattdessen wurde in der Projektsitzung vom 12.05.2005 anhand der Tasks überprüft, ob alle gesetzten Ziele erreicht wurden. Dabei konnte festgestellt werden, dass alle geplanten Tasks der Kategorie „Aufräumarbeiten“ erfolgreich durchgeführt werden konnten. In die Kategorie „Parametrisierung“ wurde nachträglich die geplante Kategorie „Abbruchkriterien“ integriert, da sich dies im Laufe der Bearbeitung als sinnvoll erwies, so dass auch hier alle Ziele vollständig und zügig erreicht werden konnten.

Im Gegensatz dazu konnten in der Kategorie „Infobeschaffung“ die gesetzten Ziele nicht vollständig umgesetzt werden. Sowohl der Einsatz des Java-Profilers zum Erzeugen von Heap-Dumps als auch die Anknüpfung an das Java-Debugging-Interface erwiesen sich im Laufe der Zeit als arbeitsintensiver als geplant, und trotz sorgfältiger Durchführung der Studien zu diesen Alternativen traten unvorhergesehene Schwierigkeiten auf (z.B. fehlende Daten im Heap-Dump), so dass beide Alternativen nicht alle gewünschten Informationen liefern konnten. Daher konnte das Ziel, einen leistungsfähigeren Ersatz für den bisherigen Algorithmus zu finden, nicht erreicht werden, weshalb daran im vierten Release weiter gearbeitet wird.

Mit der Überarbeitung der Animationsinfrastruktur sollten Vorarbeiten für einen flexibleren und effizienteren Einsatz von Animationen im Endrelease durchgeführt werden. Obwohl große Strukturveränderungen vorgenommen wurden, konnten die Arbeiten planmäßig durchgeführt und abgeschlossen werden.

KAPITEL 17

Beschreibung des vierten Release

17.1 Einleitung

Jonas Mathis, Daniel Vogtland

Nachdem mit Release 3 die grundlegenden Funktionalitäten des 3D-Debuggers gegeben sind – insbesondere sind hier die verschiedenen Möglichkeiten zur Informationsbeschaffung und die Visualisierungstechniken zu nennen – soll nun das Hauptaugenmerk auf der Praxistauglichkeit und Nutzbarkeit des Plug-Ins liegen. Dies bedeutet auf der einen Seite die Überarbeitung bestehender Klassen mit der Zielsetzung, die Effizienz und somit die Ausführungsgeschwindigkeit des Programms zu erhöhen. Auf der anderen Seite mussten Interaktionsmöglickeiten für den Benutzer gefunden und implementiert werden, damit das Plug-In auch eine wirkliche Unterstützung im Debugging-Prozess darstellt. Außerdem sollen die Informationsbeschaffungsmechanismen weiterentwickelt werden, um Schwachstellen der ursprünglichen Varianten zu beheben. Die zu Beginn des Projektes entworfene Systemmetapher soll durch dieses Release verwirklicht werden.

17.2 Reflexion über die Tasks

Henning Zeller

Die im Rahmen des vierten und letzten Releases zu erledigenden Aufgaben lassen sich in vier wesentliche Bereiche einteilen. Erstens sollten einige, im Laufe des bisherigen Entwicklungsprozesses (oder auch im aktuellen Release) bekannt gewordene Bugs entfernt werden. Zweitens sollte im Hinblick auf das Endprodukt die Benutzbarkeit unseres Plug-Ins erhöht werden. Hier sollten insbesondere die Interaktionsmöglichkeiten und die bereitgestellten Debugging-Funktionalitäten erweitert und intuitiver gestaltet werden. Drittens wurde ein Profiler verwendet, um performanzkritische Bereiche unseres Produktes zu ermitteln und Verbesserungsmöglichkeiten zu prüfen. Viertens war es nun an der Zeit, Tasks, die aus früheren Releases übrig geblieben waren, zu bearbeiten.

17.2.1 Bugfixes

Hohe CPU Auslastung auf Windows Rechnern aufspüren und beseitigen

Beschreibung: Die Ausführung des Plug-Ins auf Windows-Rechnern lässt die CPU-Last regelmäßig auf 99 Prozent ansteigen.

geplante Zeit: 4 Tage

reale Zeit: 0,5 Tage. Es wurde eine Endlosschleife im Netdumper (einem der Informationskollektoren) entfernt, die offensichtlich für oben beschriebenes Programmverhalten verantwortlich war.

Laden und Speichern von Snapshots ermöglichen

Beschreibung: Dieser Task wurde bereits in einem früheren Release implementiert, es gab jedoch auf einigen Maschinen immer wieder Probleme bei der Nutzung dieser Funktionalität.

geplante Zeit: 0,5 Tage

reale Zeit: Das Problem konnte umgehend behoben werden, da lediglich ein Pfad lokal gesetzt war.

3DView so anpassen, dass sie nach Minimieren und Wiederherstellen weiterhin nutzbar ist.

Beschreibung: Bei Minimieren der Java3D-View bleibt diese auf Windows-Rechnern einfach weiss. Dieses Problem besteht seit Beginn der Entwicklung.

geplante Zeit: 4 Tage

reale Zeit: 1 Tag; Es wurde ein Teilnehmer unserer Vorgänger-PG (PG Effects) zu diesem Phänomen befragt und seine daraufhin gemachten Vorschläge umgesetzt. Diese beruhten im Wesentlichen auf einem Überschreiben der `setFocus()`-Methode der Java3D-View, so dass nach jeder Größenänderung der View alle Inhalte entfernt, neu eingesetzt und neu gezeichnet wurden.

Fehlerhaftes Merken von selektierten Objekten bei Wechsel zwischen Snapshots beheben

Beschreibung: Beim Wechsel zwischen Snapshots wurden ursprünglich nicht markierte Objekte markiert.

geplante Zeit: 1 Tag

reale Zeit: 0,5 Tage; Problem konnte mit geringem Aufwand gelöst werden, es handelte sich um einen Schleifen-bezogenen Logikfehler.

17.2.2 Usability (Benutzerinteraktion)

Export der Animation in einen 2D-Film ermöglichen

Beschreibung: Aus einer Sequenz Snapshot-Bildern im JPEG-Format sollte ein Film generiert werden, welcher dann in einem gängigen Format vorliegt.

geplante Zeit: 3 Tage

reale Zeit: 4 Tage; Hier wurde auf das von SUN zu diesem Zweck als Package bereit gestellte JAVA MEDIA FRAMEWORK (JMF) zurückgegriffen. Die Verzögerung um einen Tag resultierte aus den Schwierigkeiten mit dem vom JMF nicht implementierten MPEG-Format. Letztendlich wurde daher das QUICKTIME-Format verwendet.

Mehrfache Selektion ermöglichen

Beschreibung: In der Java3D-View sollten statt bisher nur eines Objektes mehrere (im Prinzip beliebig viele) selektiert werden können.

geplante Zeit: 2 Tage

reale Zeit: 0,5 Tage; Dieser Task stellte kein Problem dar, es mussten lediglich kleinere Änderungen an der zugrunde liegenden Datenstruktur vorgenommen werden.

Property-View an die Möglichkeit der mehrfachen Selektion anpassen

Beschreibung: Es sollte für jedes selektierte Objekt ein Tab im Property-View hinzugefügt werden.

geplante Zeit: 1 Tag

reale Zeit: Der Task konnte in der vorgegebenen Zeit erledigt werden, allerdings machte das SWT-Layout zur Verschachtelung von Tabs und Tabellen sowie die Zuschaltung der Scrollbalken an den richtigen Stellen Schwierigkeiten. Es wurde keine Möglichkeit gefunden, die Attribut-Tabelle innerhalb eines Tabs stets auf die volle Breite zu maximieren.

Erhalt der Selektion beim Rotieren/Zoomen

Beschreibung: Bisher ging die Markierung eines Objektes verloren, wenn der Benutzer die Szene rotierte oder den Zoom betätigte.

geplante Zeit: 2 Tage

reale Zeit: 0,5 Tage; Der Task wurden zusammen mit der Realisierung der mehrfachen Selektion erledigt.

Fokussierung auf die mehrfache Selektion erweitern

Beschreibung: In der Java3D-View konnte bisher nur ein Objekt fokussiert werden, so dass nun aufgrund der Möglichkeit der mehrfachen Selektion eine Anpassung stattfinden musste, damit mehrere Objekte fokussiert werden können.

geplante Zeit: 2 Tage

reale Zeit: 2 Tage; Die veranschlagte Zeit reichte aus.

Objekte, die als Attribut in der Property-View angezeigt werden, beim Anklicken auch in der Java3D-View markieren

Beschreibung: Bei einem Doppelklick auf ein als Attribut angezeigtes Objekt in der Property-View sollte selbiges auch in der Java3D-View wie bei der einfachen Selektion markiert werden.

geplante Zeit: 3 Tage

reale Zeit: 3 Tage; Der Task bereitete keine größeren Probleme.

DebugObjectAttribute-IDs für Objekte in Kollektoren setzen

Beschreibung: Diese Funktionalität ermöglichte die Identifizierung von Objekten in der Property-View, wodurch die „Doppelklick“-Funktion der Property-View erst realisiert werden konnte.

geplante Zeit: 2 Tage

reale Zeit: 2 Tage; Keine Probleme.

Wenn ein Objekt markiert ist, sollen alle nicht mit ihm verbundenen Objekte ausgeblendet/abgedunkelt werden können. Die Pfadlänge ist dabei vom Benutzer wählbar.

Beschreibung: Der Benutzer kann so Abhängigkeiten im Bezug auf ein gewähltes Objekt ermitteln und sich zudem bei komplexeren Anwendungen sukzessive vorarbeiten.

geplante Zeit: 2 Tage

reale Zeit: Der geschätzte Aufwand bestätigte sich, jedoch arbeitet die Funktion zuweilen nicht zuverlässig. Die Probleme scheinen bei Java3D zu liegen, da Debugger-Ausgaben richtige Daten bei nicht-deterministischer falscher Darstellung vorlagen. Eine im Release verwendete Hilfslösung bestand darin, die selben Java3D-Befehle immer wieder aufzurufen. Eine sinnvollere Alternative konnte in der vorhandenen Zeit nicht mehr gefunden werden.

Einstellung der Animationsdauer über die Preferences ermöglichen

Beschreibung: Über die `PreferencePage` kann ein Default-Wert für die Animationsdauer gesetzt werden. Zudem kann diese in der `LaunchConfiguration` angepasst werden.

geplante Zeit: 2 Tage

reale Zeit: 0,5 Tage; Keine Probleme. Die Animationsdauer wird in der finalen Version allerdings im sogenannten Snapshot-Player geregelt.

Snapshotlisten laden, zusammensetzen, teilen und einzelne Snapshots entfernen können

Beschreibung: Diese Funktionalitäten sind z.B. für verteilt arbeitende Entwicklerteams interessant oder aber für den Fall, dass Debugging-Sessions über längere Zeiträume hinweg durchgeführt werden müssen.

geplante Zeit: 3 Tage

reale Zeit: Dieser Task wurde durch die Realisierung des Snapshot-Players abgedeckt.

17.2.3 Usability (Debugging-Funktionalität)

Zyklische Animation (Repeat) ermöglichen

Beschreibung: Ist diese Funktion aktiviert, wird die zuletzt visualisierte Transitionsfolge zyklisch wiederholt.

geplante Zeit: 2 Tage

reale Zeit: 2 Tage; Es traten keine größeren Probleme auf.

Animation über mehrere Snapshots hintereinander ermöglichen (Start- und Endpunkt festlegbar)

Beschreibung: Bisher konnte lediglich die Transition zwischen zwei benachbarten Snapshots visualisiert werden, dies sollte nun entsprechend erweitert werden.

geplante Zeit: 1 Tag

reale Zeit: 1 Tag; Keine Probleme.

Automatisierte Snapshots ermöglichen

Beschreibung: Diese Funktionalität soll es ermöglichen, über einen festgelegten Zeitraum in regelmäßigen Abständen Snapshots zu erstellen.

geplante Zeit: 3 Tag

reale Zeit: 0,5 Tage; Die Bearbeitung dieses Tasks erwies sich deutlich einfacher als erwartet.

Automatische Abarbeitung aller gewählten Breakpoints ermöglichen

Beschreibung: Hierbei soll es dem Benutzer auf Wunsch erspart werden, zwischen den einzelnen Breakpoints den Resume-Button betätigen zu müssen.

geplante Zeit: 1 Tag

reale Zeit: 1 Tag; Es gab lediglich kleinere Positionierungs- und Größenprobleme bei der Einbindung der nötigen SWT-Widgets in der Layout des Launch-Dialogs auf.

Automatische StepIntos zwischen zwei Breakpoints ermöglichen

Beschreibung: Automatische Generierung von Snapshots für Anweisungen, die sich zwischen zwei Breakpoints befinden.

geplante Zeit: 5 Tage, falls überhaupt machbar

reale Zeit: Dieser Task erwies sich im Rahmen der zur Verfügung stehenden Zeit und des vorhandenen Wissens über das Debugging-Interface von Eclipse als nicht umsetzbar.

17.2.4 Profiling

Untersuchung von Verbesserungsmöglichkeiten

Beschreibung: Hier mussten zunächst einmal performanzkritische Bereiche unserers Plug-Ins mithilfe des Profilers identifiziert werden.

geplante Zeit: 3 Tage

reale Zeit: 3 Tage; Im vorgegebenen Zeitrahmen konnten einige solcher Bereiche ausfindig gemacht werden, deren Verbesserung die nötige Leistungssteigerung bewirkte.

Umsetzung der Verbesserungen

Beschreibung:	Effizientere Implementierung der durch den Profiler ermittelten performanzmindernde Code-Bereiche
geplante Zeit:	2 Tage
reale Zeit:	2 Tage; In der veranschlagten Zeit konnten vor allem im Bereich der von uns verwendeten Datenstrukturen deutliche Verbesserungen erzielt werden.

17.2.5 Übrig gebliebene Arbeiten aus Release 3

Infobeschaffung über JDI implementieren

Beschreibung: Es sollte ein weiterer Kollektor realisiert werden, der sich zur Beschaffung der Laufzeitinformationen der JDI bedient.

geplante Zeit: 2 Tage

reale Zeit: Der Task konnte in der vorgesehenen Zeit bearbeitet werden. Die Darstellung von Arrays ist allerdings technisch nicht möglich, da das JDI die dazu benötigten Informationen nicht zur Verfügung stellt.

Kombination von Informationsbeschaffungsalgorithmen

Beschreibung: Da alle bisher implementierten Kollektoren spezifische Stärken und Schwächen aufweisen, sollte eine Kombination dieser Kollektoren entwickelt werden, umso möglichst viele vorhandene Stärken abzudecken.

geplante Zeit: 5 Tage, falls überhaupt machbar

reale Zeit: Der Task erwies sich als nicht durchführbar, da die einzelnen Kollektoren unterschiedliche ObjectIDs verwenden.

GUI-Elemente für die Animation überarbeiten/ergänzen

Beschreibung: Die Benutzerführung sollte vereinfacht werden, indem über einen sogenannten Snapshot-Player Snapshots für die Animation oder die statische Visualisierung ausgewählt und bearbeitet werden können.

geplante Zeit: 4 Tage

reale Zeit: 5 Tage; kleinere Probleme bei der Synchronisation der einzelnen Views: Instanziierung des Players, gegenseitiger Ausschluss von `Snapshotplayer` und `Java3DView` bei `Visualizer`-Zugriff.

Optimierung des RecursiveSnapshotCollectors

Beschreibung: Der RecursiveSnapshotCollector sollte hinsichtlich seiner Performance optimiert werden.

geplante Zeit: 3 Tage

reale Zeit: Der Zeitaufwand bestätigte sich.

Einsatz von 3D-Tooltips verbessern/überdenken

Beschreibung:	Die bisher verwendeten 3D-Tooltips hatten den Nachteil, dass ihre Sichtbarkeit vom Abstand der Szene zum Betrachter abhing, d.h. je weiter die Szene herausgezoomt wurde, desto mehr verkleinerten sich die Tooltips bis hin zur Unlesbarkeit.
geplante Zeit:	2 Tage
reale Zeit:	1,5 Tage; Die Idee von 3D-Tooltips wurde fallen gelassen und durch die Nutzung von 2D-Tooltips außerhalb der 3D-Darstellung ersetzt.

17.3 Analyse einzelner Snapshots

Carina Klar, Jonas Mathis, Michael Striewe, Daniel Vogtland

In diesem Abschnitt werden diejenigen Funktionen erläutert, die bei der Analyse eines einzelnen Snapshots ohne Betrachtung von zeitlichen Veränderungen relevant sind. Sie sollen insbesondere die Analyse von Zusammenhängen zwischen einzelnen Objekten ermöglichen und dem Nutzer erlauben, die für ihn relevanten Informationen in den Vordergrund zu stellen. Einige dieser Aspekte bleiben auch während der Animation erhalten, während andere bei Snapshotübergängen nicht erhalten bleiben.

17.3.1 Mehrfachselektion

Der bisherige Selektionsmechanismus für Objekte in der `Java3DView` stellte sich in vielen Belangen als unzureichend heraus. Zum Einen können für den Benutzer auch mehrere Objekte gleichzeitig von Interesse sein, die er beispielsweise durch Fokussierung in den Mittelpunkt der Darstellung bewegen oder die er als Ausgangspunkt für die Ausblendefunktion (Abschnitt 17.3.2) nutzen möchte. Zum Anderen ging bislang bei jeder mausbezogenen Navigation innerhalb der dreidimensionalen Szene die Selektion verloren. Diese Probleme sind durch die Einführung einer Mehrfachselektion behoben worden. Im Folgenden sind alle wichtigen Änderungen aufgelistet.

- *Änderungen im `PickBehavior`*: Das `PickBehavior` enthält nun, anstelle einer Refernz auf das selektierte Objekt, ein Array von Referenzen auf mehrere selektierte Objekte. Ein Mausklick ins „Leere“ hat nun keine Auswirkungen mehr auf die aktuelle Selektion. Ein Klick auf ein Objekt nimmt dieses in die Selektion auf, falls dieses noch nicht selektiert ist. Andernfalls wird es wieder deselektiert.

- *Änderungen an der* `Java3DView`: Bestehende und neue Funktionen mussten auf die Mehrfachselektion zugeschnitten werden. Die Fokussierung selektierter Objekte wurde komplett neu implementiert. Um die selektierten Objekte herum wird eine `BoundingBox` gelegt und diese bezüglich der aktuellen Sicht maximiert. Im Vergleich zur früheren Implementierung bleibt die davon unabhängige Nutzernavigation, wie beispielsweise die Rotation der Szene, vollständig erhalten.
  Die zweite wichtige Änderung betraf das Ausblenden von nicht relevanten Objekt- und Relationsvisualisierungen. In ihrer Einführungsphase war die Methode für die einfache Selektion programmiert worden und musste dementsprechend auf Mehrfachselektion erweitert werden, was jedoch keine Probleme darstellte.

- *Änderungen an der* `PropertyView`: Für jedes selektierte Objekt wird ein Reiter erzeugt und hinter die bestehenden Reiter eingefügt. Wird ein Objekt deselektiert, wird der entsprechende Reiter entfernt und - falls vorhanden - der zuletzt hinzugefügte angezeigt.
  Die Reiterüberschrift besteht aus der ID des Objektes. Der Inhalt der Attributtabelle und das Titellabel der Tabelle, bestehend aus ID, Name und Klassenzugehörigkeit des Objektes, entsprechen weitestgehend der Vorgängerversion. Falls es sich bei einem aufgelisteten Attribut selbst wieder um ein Objekt handelt, wird dieses Objekt mit einem Doppelklick auf den Namen in der Attributliste selektiert (sofern es noch nicht selektiert ist) und der entsprechende Reiter angezeigt. Zu diesem Zweck wurde in der Klasse `DebugObjectAttribute` ein optionales Attribut eingeführt, welches auf die ID eines Objektes verweist.
  Jeder Reiter verfügt über zwei zusätzliche Buttons. Mit „deselect“ wird das angezeigte Objekt deselektiert, was dieselbe Wirkung wie eine Deselektion per Maus hat. Bei einem Auslösen des „highlight“ Buttons blinkt das Objekt kurz auf, was eine große Hilfe bei Benutzung der Doppelklickfunktion für Objektsprünge darstellt, da oftmals ein auf diese Weise zusätzlich hervorgehobenes Objekt nicht sofort ausgemacht werden kann.

17.3.2 Ausblendefunktion

Die Ausblendefunktion ist dazu gedacht, dem Nutzer auch in großen Snapshots die Möglichkeit zu geben, die Übersicht über einen ausgewählten, zusammenhängenden Bereich zu gewinnen. Dafür steht in der GUI ein Button sowie ein Texteingabefeld zur Verfügung. Nach Auswahl eines oder mehrerer Objekte und Angabe einer Tiefe im Textfeld werden nach Betätigen des Buttons alle Objekte ausgeblendet, die von den gewählten Objekten um mehr als die eingegebene Anzahl von Zwischenstationen entfernt sind. Ausgeblendete Objekte sind nur noch als fast transparente 3D-Objekte sichtbar. Alle Kanten, die zu ausgeblendeten Objekten führen oder von ihnen ausgehen, werden ebenfalls ausgeblendet. Diese Funktion ermöglicht die Analyse eines besonders interessanten Bereichs des Snapshots, ohne dass der Benutzer dafür einen neuen Snapshot mit reduzierter Klassenauswahl erzeugen müßte.

Während der Animation bleibt dieser Zustand allerdings nicht erhalten. Dies liegt daran, dass aus Gründen der Einfachheit und zur Vermeidung von Seiteneffekten im `Visualizer` das

Ausblenden direkt über die Veränderung der Transparenzattribute einzelner 3D-Objekte in der `Java3DView` umgesetzt wird. Dadurch wird die zur Animation genutzte Datenstruktur nicht angetastet und somit bei jedem Animationslauf wieder zurückgesetzt.

Die Identifizierung der auszublendenden Objekte ist über eine Breitensuche realisiert, in der über die wachsende Tiefe des Baumes iteriert wird. Danach werden alle betroffenen Relationspfeile ausgeblendet. Das Neuzeichnen der Grafik in der `Java3DView` ist allerdings fehlerbehaftet, da Java3D nicht jeden Zeichenbefehl sofort umsetzt. Dies kann dazu führen, dass Objekte oder Relationen nicht oder nur teilweise ausgeblendet werden, obwohl ihre Transparenz korrekt gesetzt wurde. Um die irritierenden Auswirkungen für den Nutzer zu minimieren, wird nach dem Klicken auf den Button ein Thread gestartet, der den Befehl zum Ausblenden wiederholt aufruft. Das führt dazu, dass das Ausblenden von Objekten und Pfeilen für den Benutzer schrittweise geschieht. Dieses Verfahren garantiert allerdings nicht, dass damit tatsächlich alle Objekte und Pfeile korrekt ausgeblendet werden. Eine bessere und zuverlässigere Methode zur Lösung dieses Problems konnte in der gegebenen Zeit nicht gefunden werden.

17.4 Snapshotplayer

Antonio Pedicillo

Der `SnapshotPlayer` wurde eingeführt, um die `Java3DView` zu entlasten, die im Release 3 noch sämtliche Nutzer-Interaktion beinhaltete, wodurch in dieser View zu viele Buttons vorhanden waren. Durch die Auslagerung eines Teils der Funktionen wie Laden, Speichern und Abspielen mehrerer Snapshots in den `SnapshotPlayer` wird die Übersichtlichkeit verbessert.

17.4.1 Funktionalitäten des Snapshotplayers

Die Oberfläche des `SnapshotPlayers` (Abb. 17.1) ähnelt denen üblicher Media-Player. Zur Orientierung, welcher Snapshot gerade in der `Java3DView` angezeigt wird, existieren eine Anzeige, ein Schieberegler und zwei Listen. In der Liste auf der linken Seite werden die aktuell im Speicher befindlichen Snapshots aufgeführt. Auf der rechten Seite können frühere gespeicherte Snapshots geladen werden. Der Schieberegler zeigt die aktuelle Position im Animationsverlauf und kann zum schnellen Vor- und Zurückblättern in der Snapshotliste genutzt werden. Zwischen dem Schieberegler und den Listen gibt es die Schaltflächen zur Steuerung der Animation. Im Folgenden werden die einzelnen Schaltflächen vorgestellt.

Zurück-Schaltfläche: Bei der Zurück-Schaltfläche wird in der `Java3DView` auf einen existierenden vorherigen Snapshot übergegangen. Dabei werden die Anzeige, der Schieberegler und die Selektion der Liste auf der linken Seite aktualisiert.

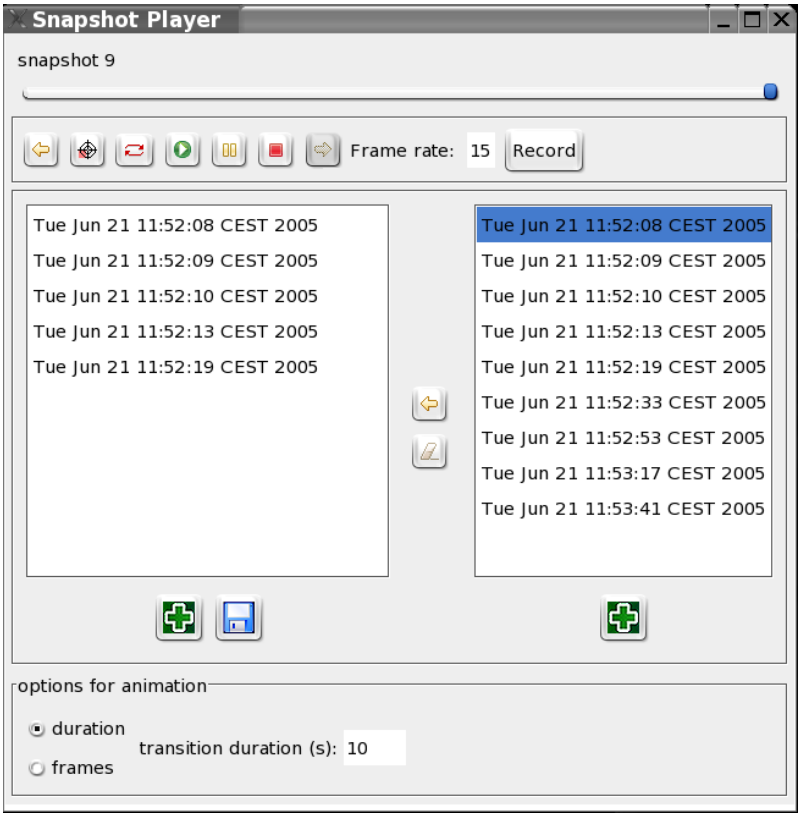


Abbildung 17.1.: Der Snapshotplayer

Fokussierungs-Schaltfläche: Während der Animation kann die Fokussierung der Objekte in der der `Java3DView` beibehalten werden. Dies ist nicht bei Objekten möglich, die in späteren Snapshots entstehen.

Wiederholungs-Schaltfläche: Bei der Wiederholungsschaltfläche handelt es sich um einen Toggle-Button. Ist dieser eingeschaltet, werden die selektierten Snapshots in der Liste auf der linken Seite endlos wiederholt.

Abspiel-Schaltfläche: Die Abspiel-Schaltfläche startet die Animation. Dabei werden die selektierten Snapshots der Liste auf der linken Seite animiert.

Pause-Schaltfläche: Die Pause-Schaltfläche unterbricht die laufende Animation bzw. setzt sie fort. Das Pausieren unterbricht die Animation immer mit Erreichen eines vollständigen Snapshots, aber nicht während der Animation eines Snapshot-Übergangs. Somit wird verhindert, dass der Benutzer durch halb eingeblendete Objekte irritiert wird.

Stop-Schaltfläche: Durch die Stop-Schaltfläche kann die laufende Animation gestoppt werden. Die Anzeige springt zum nächsten vollständigen Snapshot.

Vorwärts-Schaltfläche: Bei der Vorwärts-Schaltfläche wird in der `Java3DView` auf einen nächsten Snapshot übergegangen. Dabei werden die Anzeige, der Schieberegler und die Selektion der Liste auf der linken Seite aktualisiert.

17.4.2 Speichern und Laden der Snapshots

Zum Speichern und Laden der Snapshots gibt es zwei Listen. In der links dargestellten Liste werden die Snapshots eines aktuellen 3Debug-Laufs angezeigt, während rechts gespeicherte Snapshots geladen werden können. Dem Benutzer wird die Möglichkeit angeboten, Snapshots in der Liste auf der linken Seite zu löschen. Dazu werden die jeweiligen Snapshots in der Liste selektiert und durch einen Klick auf die Löschen-Schaltfläche entfernt. Dem Benutzer ist es freigestellt, ob er noch zusätzlich gespeicherte Snapshot aus früheren Debug-Läufen in die Liste auf der linken Seite einfügen möchte. Um dies zu ermöglichen, müssen alle gewünschten Snapshots auf der rechten Seite selektiert werden. Durch einen Klick auf die Hinzufügen-Schaltfläche werden alle selektierten Snapshots in die Liste auf der linken Seite kopiert. Dies ist für den Fall gedacht, dass man Snapshots aus verschiedenen Debugläufen, die verschiedene Ebenen des Programms repräsentieren, zusammenstellen möchte. Dazu fügt man in die Liste auf der linken Seite nur Snapshots ein, die relevant erscheinen. Somit kann der Benutzer nach freier Wahl eigene Abläufe der Snapshots zusammenstellen, die ihm ein besseres Verständnis des Programms geben können. Es kann nur auf der linken Seite gespeichert werden, da auf der rechte Seite keine Veränderungen vorgenommen werden können.

17.4.3 Animation und Aufnahme

Der Benutzer kann bei der Animation zwischen „Zeit in Sekunden“ und „Framerate“ wählen. Ist „Zeit in Sekunden“ eingestellt, wird innerhalb dieser eingestellten Sekunden zwischen zwei Snapshots animiert. Bei eingestellter Framerate wird der Übergang zwischen zwei Snapshots durch die Anzahl der Frames bestimmt. Eine gewählte Animation lässt sich auch aufzeichnen. Der Benutzer kann zwischen einer Framerate von 1 bis 35 wählen. Als Voreinstellung ist eine Framerate von 15 eingestellt. Wenn der Benutzer auf die Aufnahme-Schaltfläche drückt, werden im Hintergrund, abhängig von der Framerate, JPEG-Bilder von der Animation erstellt. Nach Beendigung dieses Vorgangs wird aus den JPEG-Bildern mittels der von SUN bereitgestellten Klasse `JpegImagesToMovie` ein QUICKTIME-Film erstellt. Dieser kann mit allen Playern abgespielt werden, die das QUICKTIME-Format unterstützen.

17.5 Export in einen Film

Andrey Lysenko

Als Ziel hat sich unsere Projektgruppe die Erstellung eines Debuggingfilms gesetzt, der als exportierte Video-Datei in einem gängigen Format zur Verfügung gestellt werden soll. Dieser Debuggingfilm soll den Programmierer beim Erwerb eines allgemeinen Programmverständnis und bei der Struktur- oder Fehleranalyse möglichst effektiv unterstützen. Daher wurde versucht, möglichst viele Features, die für einzelne Snapshots zur Verfügung stehen oder die der Snapshot-Player für Animationen anbietet, auch für einen Film zu erhalten. Bedingt durch den Export des Films als Video-Datei gehen allerdings auch einige Informationen verloren, wie etwa die Angaben aus der Property-View, die im Film nicht berücksichtigt werden können. Ein Debuggingfilm in Form eines extern abspielbaren Videos kann also nicht mehr alle Vorteile des dreidimensionalen Debuggings zur Verfügung stellen und ist somit unter den folgenden Bedingungen vorteilhaft.

Aufgrund der Ansprüche, die Java3D bei der Erzeugung von Animationen an die Computerressourcen, insbesondere an die Grafikleistung stellt, ist es in manchen Fällen sehr nützlich, die Animation als Film zu exportieren. Ein solcher von unserem Werkzeug erstellter Film lässt sich beliebig oft ohne ressourcenbedingte Verzögerungen abspielen, die bei den aufwändigen mathematischen Berechnungen zwischen den einzelnen Snapshots entstehen und selbst auf sehr leistungsfähigen Rechnern zu starken Beeinträchtigungen führen. Somit kann die Analyse des Programms anhand eines Films schneller als mit herkömmlichen Methoden ablaufen. Der Beschleunigungsgrad hängt sehr stark vom Umfang des zu analysierenden Projektes und von der Rechenleistung des eingesetzten Rechners ab. So kann z.B. eine umfangreiche Animation eines großen Projektes unangemessen lange dauern.

Ein aus dem oben beschriebenen Vorteil resultierender Anwendungsfall wäre das mehrfache Starten ein und desselben Animationsprozesses zur Auswertung eines Programms und schrittweisen Einarbeitung in ein komplexes Problem. Es ist in diesem Fall ressourcenschonend und vor allem zeitsparend, dazu einen Film zu erzeugen. Der aus dieser Animation einmalig erstellte Film lässt sich mit einem geeigneten Player nicht nur vor- und zurückspulen, sondern auch stoppen und pausieren. Einzelne Frames können zwecks Analyse je nach Bedarf sogar vergrößert oder verkleinert werden. Allerdings stehen dann alle anderen Funktionen, die von Java3D angeboten werden, wie Drehen, Fokussierung und ausführliche Informationen über die Objekte nicht mehr zur Verfügung.

Das andere große Anwendungsgebiet für einen aus einer Animation generierten Film besteht darin, den Ablauf eines Programms bei Präsentationen vorzuführen. Im Allgemeinen eignet sich solch ein Film besonders gut für das verzögerungsfreie Präsentieren von Programmabläufen. Wenn man bei einer Vorführung nicht zu sehr auf Details wie z.B. Ablesen der Objektattribute, sondern vielmehr aus das grobe Verständnis zielen will, dann ist es wesentlich angenehmer und leichter, einen durchgängigen und flüssigen Film als eine Animation mit unvorhersehbar langen Pausen nach jedem Snapshot nachzuvollziehen.

Die Erzeugung eines Films und die Einstellung der Feinheitsstufe der Übergänge zwischen einzelnen Frames sind im Kapitel 17.4.3 „Animation und Aufnahme“ beschrieben.

Bei der Implementierung der Exportfunktion sind wir auf Probleme bei der Kodierung ins MPEG-Format gestoßen, die die Arbeit verzögert haben. Die API des JAVA MEDIA FRAMEWORK (JMF) sieht die Erstellung von Filmen in folgenden Formaten vor: AVI, MPEG und QUICKTIME. Es wurde aus Portabilitätsgründen zunächst versucht, die Videos im AVI- oder im MPEG-Format zu kreieren, wobei Letzteres aber nicht funktionierte. Es hat sich herausgestellt, dass das Kodieren von Filmen im MPEG-Format zwar vorgesehen, aber nicht implementiert ist. Deshalb haben wir uns darauf geeinigt, das AVI- und das QUICKTIME-Format zu benutzen.

17.6 Informationsbeschaffung

Boris Brodski, Boris Düdder, Daniel Maliga

Die im Laufe des dritten Release verworfene Idee zur Nutzung des *Java Debug Interface* (JDI) zur Beschaffung von Laufzeitinformationen wurde im vierten Release doch noch einmal aufgegriffen, um eine Alternative zu den mit einigen Nachteilen behafteten `RecursiveSnapshotCollector` und `DumpSnapshotCollector` anbieten zu können. Es konnte eine lauffähige Version implementiert werden. Somit stehen dem Benutzer insgesamt drei verschiedene Verfahren zur Informationsbeschaffung zur Verfügung.

17.6.1 Java Debug Interface

Das *Java Debug Interface* (JDI) wird als allgemeine Schnittstelle von SUN für die Implementierungen von Debugging-Clients zur Verfügung gestellt und ist Bestandteil der größeren Debug-Architektur *Java Platform Debugger Architecture* (JPDA). Diese ist modular aufgebaut und bietet einen verbindungspunktorientierten Zugriff auf die Debug-Informationen. Als Basis für die Datenübertragung und Steuerung des Debuggees dient das *Java Debug Wire Protocol* (JDWP).

Vorteile dieser Lösung sind:

- Die als „Creator-Createe-Relationen“ bezeichneten Informationen über die Objekterzeugungen und die daran beteiligten Objekte können inklusive der Position im Quellcode gewonnen werden. Diese Informationen unterscheiden diese Schnittstelle stark von den anderen.

- Das „Stepping“ beim Debugging lässt sich beeinflussen. Damit ist ein automatisiertes „Step Into“ (führe den nächsten Programmbefehl aus und stoppe) oder „Step Over“ (führe den nächsten Programmbefehl in dieser Methode aus und stoppe) realisierbar. Diese Funktion bietet sich für die spätere Kontrolle und Ergänzung bei der Kombination mit anderen Informationsbeschaffungsalgorithmen an.

Nachteile dieser Lösung sind:

- Die Debugging-Schnittstelle liefert kontinuierlich Debuginformation, was den Programmablauf stark abbremst.
- Mit Hilfe dieser Schnittstelle kann keine Erzeugung von Arrays überwacht werden.

17.6.2 Integration von Informationsbeschaffungsalgorithmen

Bei der Integration der Informationsbeschaffungsalgorithmen ging es darum, die Schwächen der einzelnen Algorithmen zu eleminieren, indem sie zu einem Algorithmus kombiniert werden sollten. Dieser Projektteil wurde aufgrund des hohen Arbeitsaufwands und ungewissen Nutzens verworfen. Die verschiedenartigen Schnittstellen sind nur aufwendig miteinander zu kombinieren. Zum Beispiel sind die ObjectIDs des `JDISnapshotCollector` nicht mit ObjectIDs von `DumpSnapshotCollector` identisch und eine einfach Zusammenführung von Informationen somit nicht möglich. Weiterhin ist zu bemerken, dass der Ressourcenbedarf sich durch die verschiedenen Beschaffungsalgorithmen erhöht. Damit werden der Benutzbarkeit des Systems starke Beschränkungen auferlegt.

17.7 Persistenz

Jonas Mathis, Daniel Vogtland

Die Möglichkeit, Daten zu speichern ist ein wichtiges Feature, denn der Prozess der Datenerzeugung kann recht zeitaufwändig werden, und zur späteren visuellen Analyse ist somit eine Speichermöglichkeit für Debugging- und Visualisierungsinformation unumgänglich.
Leider traten mit der in Release 2 und 3 verwendeten XML-basierten Lösung immer wieder Probleme auf. Deren Analyse stellte sich jedoch als schwierig heraus, da ein Großteil der Hilfsklassen automatisch durch ein Tool erzeugt wurde und Fehler oftmals nicht reproduzierbar waren.

Als erste Lösung wurde das Serialisierungskonzept von Java verwendet. Hier war schon zum zweiten Release eine Implementierung entstanden, welche jedoch zugunsten der XML-Variante

nicht mehr benutzt wurde. Diese Implementierung konnte mit leichten Änderungen an die aktuelle Datenstruktur angepasst werden. Sie funktioniert zuverlässig, ist jedoch aufgrund des Dateiformats nicht auf externe Weiterverarbeitung ausgelegt.

Zum Ende der Entwicklungszeit für dieses Release wurde deshalb doch noch einmal eine XML-basierte Variante für die Persistenz entwickelt. Die Implementierung erfolgte gänzlich ohne automatische Unterstützung, was eine bessere Fehlersuche und -korrektur ermöglichte. Die Laufzeitinformation wird beim Einladen einer Datei komplett rekonstruiert, allerdings wird zu jedem `DebugObject` eine eigene `DebugClass` erzeugt, die nur den Klassenpfad enthält. Dies ist die einzige Information, die nach der Informationsbeschaffung weiterhin verwendet wird.
Die Ausgabe liefert eine XML-Datei. Eingelesen werden kann jede XML-Datei, die dem verwendeten XML-Format entspricht. Die gültige DTD ist leicht aus dem Code oder erzeugten Dateien abzuleiten.

Schließlich fanden beide Lösungen Eingang in das Release. Die Serialisierungsvariante bietet dabei den Vorteil, in der Regel Dateien zu erzeugen, deren Größe weniger als die Hälfte des Speicherplatzes der entsprechenden XML-Dateien bemisst. Die XML-Variante dagegen erzeugt „verständliche“ Dateien und ermöglicht so eine einfache Weiterverarbeitung.
Das gewählte Dateiformat wird sowohl beim Laden als auch Speichern anhand der Dateiendung entschieden. Serialisierungsdateien werden durch die Endung „3db“ gekennzeichnet, bei jeder anderen Endung wird die zu lesende oder zu schreibende Datei als XML-Inhalt interpretiert. Unter Windows kann der Typ auch per Auswahlliste gewählt werden.

17.8 Profiling

Daniel Maliga, Sebastian Vastag

Am Ende des dritten Release war ein großer Teil der geplanten Funktionalität des Plug-Ins bereits realisiert. Aufwändigere Programmteile wie z.B. die Anordnungsalgorithmen hatten bei normalgroßen zu debuggenden Programmen allerdings eine Laufzeit im Minutenbereich, was die Benutzbarkeit des Plug-Ins wesentlich herabsetzte. Daher wurde im Rahmen von Release 4 untersucht, an welchen Stellen der Programmcode zu verbessern war, um akzeptable Laufzeiten zu erreichen.

Zu diesem Zweck installierten wir auf einem Rechner das HYADES-Plug-In für Eclipse, welches einen Profiler für Java-Programme zur Verfügung stellt. Ein typischer Durchlauf des Debuggers wurde gestartet und nebenher mit HYADES der Speicherverbrauch, die Anzahl der Methodenaufrufe sowie die Laufzeiten gemessen.

Dabei wurden folgende Erkenntnisse gewonnen:

- Die Anordnungsalgorithmen griffen sehr häufig auf die Datenstrukturen zurück, welche ihrerseits extrem oft Sortier- und Vergleichsoperationen ausführten. Zum Beispiel wurden beim Debugging eines Testprogramms, welches nur 17 Objekte erstellte, eine halbe Million Vergleiche durchgeführt.
- Eine oft benötigte Information war die zu einem Objekt gehörende Menge von Assoziationen. Die bisherige dynamische Datenstruktur war über drei Listen realisiert, aus denen zunächst die verbundenen Objekte herausgesucht werden mussten.
- Beim Suchen in den Listen wurden für einen Durchlauf mehrere hundert Vergleichsoperationen ausgeführt, die alle das Comparable-Interface implementieren. Diese hatten pro Aufruf zwar eine sehr kurze Laufzeit, durch die häufige Nutzung summierten sie sich dennoch bis in den Sekundenbereich.

Um die Laufzeit an den entsprechenden Stellen zu optimieren, wurden folgende Änderungen vorgenommen:

- Die über drei Listen implementierte dynamische Datenstruktur wurde verworfen und die Relationen direkt in den zugehörigen Objekten gespeichert. Dadurch wurde die Laufzeit für eine Suchoperation von $O(log(n))$ auf $O(1)$ reduziert. Gleichzeitig verringerte sich damit die Anzahl der Vergleichsoperationen. Mit diesem Ansatz steigt zwar der Aufwand für das Ändern und Löschen von Objekten aus Snapshots, dies wurde aber faktisch nicht im Projekt genutzt.
- Die Implementierungen des Comparable-Interface waren recht allgemein angelegt. Durch Spezialisierung des Codes und Zusammenfassung in eine Klasse sank die Laufzeit um den Faktor 5, was im praktischen Einsatz auf den zur Verfügung stehenden Rechnern wieder 6-7 Sekunden pro angezeigtem Snapshot sparte.
- Die quadratische Laufzeit der Anordnungsalgorithmen konnte zwar nicht mehr reduziert werden, allerdings ließ sich auch hier durch Zwischenspeicherung von abgefragten Relationen wieder ein konstanter Faktor 5 herausholen.

17.9 Releaseabnahme

Carina Klar, Daniel Maliga, Michael Striewe

Da in diesem Release keine expliziten User-Stories durch die Kunden entworfen wurden, sondern eine allgemeine Releaseplanung aufgestellt wurde, wurde die Releaseabnahme in der Projektsitzung vom 16.06.2005 auf der Basis dieser Planung durchgeführt. Als Vorbereitung für die anstehende Präsentation auf dem Campus-Fest wurde ein Testprogramm entworfen, anhand dessen die Funktionalität des Plug-Ins bei der Abnahme demonstriert werden sollte.

Zunächst konnte festgestellt werden, dass alle wichtigen Bugfixes, die sich aus der Abnahme von Release 3 ergeben hatten, umgesetzt werden konnten. Danach wurden ausführlich die Funktionen des Snapshotplayers vorgeführt. Dabei wurde gezeigt, dass alle Usability-Ziele in Bezug auf Animationen erreicht werden konnten. Ebenfalls erfolgreich präsentiert werden konnten die Features zur statischen Usability. Lediglich beim Wechsel zwischen Animationen und statischen Aspekten wurden geringfügige Fehler sichtbar, die als Nacharbeit zum Release noch behoben werden sollen.

Im Bereich der Debugging-Funktionalität wurde nur das automatisierte Abarbeiten mehrerer Breakpoints vorgeführt, jedoch nicht die ebenfalls lauffähige Timer-Funktion. Nicht realisiert wurde die ursprünglich geplante Funktion automatischer „StepIntos“ zwischen zwei Breakpoints. Sie ist nur über den JDI-Snapshot-Collector realisierbar, nach dessen Fertigstellung nicht mehr genug Zeit für die Implementierung dieser Funktion verblieb. Ihr Fehlen reduziert den Nutzen des Plug-Ins für die detailierte Programmanalyse.

Auf eine ausführliche Überprüfung der verschiedenen Informationsbeschaffungsalgorithmen wurde ebenfalls verzichtet, wenngleich der im dritten Release noch nicht verfügbare JDI-Collector nun vollständig realisiert war. Die geplante Kombination der drei Beschaffungsalgorithmen erwies sich als nicht durchführbar und wurde daher fallen gelassen.

Allgemein machten sich in der Abnahme die positiven Ergebnisse des Profilings bemerkbar, da beispielsweise die Informationsbeschaffung und die Anordnungsalgorithmen deutlich schneller liefen als im vorherigen Release.

TEIL 4

Reflexion

KAPITEL 18

Nutzen des Plug-Ins

Daniel Maliga, Jonas Mathis, Daniel Vogtland

In diesem Kapitel wird erläutert, inwieweit das 3Debug-Plug-In im Rahmen eines Debugging-Prozesses von Nutzen sein kann. In Kapitel 5.2 wurde das *Integrated Comprehension Model* vorgestellt, das drei Betrachtungsebenen eines zu debuggenden Programms einführt: das *Top-Down-Modell*, das *Programmmodell* und das *Situationsmodell*. Wir werden im Folgenden beschreiben, welche Unterstützung unser Plug-In für die jeweilige Ebene bietet.

18.1 Top-Down-Modell

Auf dieser Ebene werden die Komponenten des Gesamtsystems, ihre grundsätzlichen Funktionen und ihre Zusammenarbeit erfasst. Die debuggende Person verfügt über generelles Wissen bezüglich des Anwendungsgebietes (Domainwissen).

In der Regel sind die Komponenten eines Systems aus mehreren Klassen zusammengesetzt, wobei die Zugehörigkeit zu einer Komponente nicht fest definiert ist, sondern durch die Beziehungen zu den anderen Klassen in der Komponente charakterisiert wird. Nach dieser „Richtschnur“ werden Komponenten durch den implementierten Force-Directed-Layout-Algorithmus (siehe Kapitel 20.2) intuitiv visualisiert, da dieser Objekte entsprechend der Relationen untereinander partitioniert. Hier sind auch andere Layout-Algorithmen denkbar (z.B. LinLog mit Clustering nach Package-Zugehörigkeit und schwächer gewichteter Einfluss der Relationen), welche entwickelt und von unserem Plug-In verwendet werden können.

Mit Unterstützung durch unser Plug-In wird so mitunter die Einarbeitung in ein fremdes System vereinfacht. Ist die Initialisierung des zu untersuchenden Systems abgeschlossen, so lässt sich durch einen gesetzten Breakpoint (falls eine entsprechende Codestelle bekannt ist) oder manuell (z.B. nach erfolgreichem Aufbau einer grafischen Benutzerschnittstelle) ein Snapshot erzeugen. Ein Blick auf die Visualisierung dieses Snapshots unter Verwendung des Force-Directed-Layout-Algorithmus kann beispielsweise Aufschluss über die Gewichtung der einzelnen Komponenten (Anzahl der instanziierten Objekte pro Komponente) und das Zusammenspiel von Komponenten (Abstand zwischen den Komponenten, zentral positionierte Komponenten, Instanziierungspfeile bei Nutzung der JDI-basierten Informationsbeschaffung) liefern.

Alternativ kann der Benutzer auch automatisch Snapshots nach einem festen Zeitintervall generieren lassen. So kann man das Verhalten des Systems über einen längeren Zeitraum hinweg analysieren. Dies wird durch eine animierte Visualisierung erleichtert. Dabei ist durch die Navigationsmöglichkeiten der *3D Debug View* eine freie Wahl der Perspektive möglich. Durch die Option, Snapshots speichern zu können, kann die Betrachtung auch zu späteren Zeitpunkten ohne erneute Sammlung der Debuginformationen und Berechnung der Visualierungsdaten erfolgen. Soll das Systemverhalten in einem Vortrag demonstriert werden (z.B. während der Einarbeitung eines neuen Teams), kann eine Animation auch in einen 2D-Film exportiert und so leicht in die Präsentation eingearbeitet werden.

18.2 Programmmodell

Auf dieser Ebene werden kurze Quelltextabschnitte analysiert. Der Benutzer entwickelt dabei genaue Vorstellungen über den (korrekten) Ablauf von Algorithmen, den Datenfluss im betrachteten Programm und die verwendeten Datenstrukturen. Dies alles wird unter dem Begriff „Funktionswissen“ zusammengefasst. Die Programmanalyse erfolgt hierbei vollständig auf Quelltextebene.

Dies ist das klassische Anwendungungsgebiet eines textuellen Debuggers, allerdings bietet auch unser Plug-In Unterstützung auf dieser Ebene. Die *Property View* bietet Informationen über aktuelle Attribute selektierter Objekte im betrachteten Snapshot. Abhängig von der gewählten Informationsbeschaffung umfasst dies Attributbezeichnungen, Attributwerte und Verweise auf andere Objekte, die ihrerseits wieder durch die Doppelklick-Funktion der *Property View* in die Selektion aufgenommen werden können.

Der Nutzer wird auf der *Programmmodell*-Ebene in der Regel Snapshots durch von ihm gesetzte Breakpoints erzeugen lassen, um einen kritischen Quelltextabschnitt zu prüfen. Das automatische Abarbeiten aller Breakpoints stellt hier eine hilfreiche Erleichterung für ihn dar. Eine nützliche Ergänzung hätte in der angestrebten automatischen StepInto-Funktion zwischen zwei festgelegten Breakpoints bestanden. Dies wäre jedoch nach unseren Erkenntnissen nur mit dem JDI-Interface möglich gewesen und ließe sich somit aufgrund der geforderten Austauschbarkeit von Informationsbeschaffungs-Algorithmen nicht in das Plug-In integrieren.

Die betrachteten Informationen lassen sich durch eine Auswahl von relevanten Klassen in der *Choose Classes View* auf interessante Aspekte reduzieren. Dies stellt einen Vorteil gegenüber vielen textuellen Debuggern (wie dem ECLIPSE-Debugger) dar.

Da Objekte zuerst in der *3D Debug View* selektiert werden müssen, bevor ihre textuellen Attribut-Informationen in der *Property View* betrachtet werden können, sind die klassenabhängige Färbung (Eingrenzung möglicher Objekte) und die Tooltips (schnelle Objektidentifikation) nützliche Hilfestellungen zum schnelleren Auffinden interessanter Objekte. Da allerdings auf der Ebene des *Program Models* kleinere Konstrukte wie z.B. Schleifen mit Zählvariablen und Abbruchbedingungen im Mittelpunkt stehen, wird wohl in der Mehrzahl der

Fälle ein gewohntes, klassisches textuelles Debugging-Tool Verwendung finden, und nur für oben genannte Situationen, wie die Einschränkung der zu betrachtenden Klassen, auf die 3D-Visualisierung zurückgegriffen.

18.3 Situationsmodell

Diese Ebene bringt das *Top-Down-Modell* und das *Programmmodell* zusammen. Der Programmablauf wird nicht mehr wie im *Programmmodell* auf Quelltextebene nachvollzogen, sondern eher auf algorithmischer Ebene. Dabei fließen auch globalere Kenntnisse über Zusammenhänge aus dem *Top-Down-Modell* mit ein.

Ein Snapshot entspricht einer Momentaufnahme des zu untersuchenden Systems und umfasst zusätzlich entsprechende Visualisierungsdaten für die *3D Debug View*. Die im Snapshot enthaltenen Informationen können, wie bereits erwähnt, auf relevante Aspekte reduziert werden. Die dreidimensionale, graphbasierte Visualisierung mit gesonderter Array- und Exception--Darstellung unterstützt den Benutzter beim Erkennen von Strukturen und Unregelmäßigkeiten innerhalb von diesen (vgl. Kapitel 4). Bei Benutzung der JDI-basierten Informationsbeschaffung können außerdem Instanziierungen gesondert (durch eine besondere Art der Relations-Pfeile) visualisiert werden. Diese Beziehungen sind für den Debugging-Prozess von großer Wichtigkeit. Somit lassen sich beispielsweise Instanziierungsbeziehungen gut erkennen und Fehlinstanziierungen leicht erkennen.

Die Möglichkeit, Objekte zu selektieren und (bis zu einer festgelegten Tiefe) mit diesen unverbundenen Objekte auszublenden, ermöglicht es auch, relevante Teilstrukturen visuell hervorzuheben. Dies erleichtert die Fehlersuche, da gezielt unterschiedliche Teile desselben Snapshots untersucht werden können.

Doch nicht nur die Betrachtung eines einzelnen Snapshots bietet Unterstützung im Debugging-Prozeß, auch der Vergleich mehrerer Snapshots kann erheblich zur Analyse eines fehlerhaften Programm-(Teil-)Ablaufs beitragen, beispielsweise bei der Implementierung eines Algorithmus, die zu ungültigen Resultaten führt. Mehrere Snapshots, die in aufeinanderfolgenden Codeabschnitten erzeugt wurden, und die Veränderungen zueinander geben dabei Aufschluß über die von eben diesen Programmteilen ausgeführten Aktionen auf den Objekten, wie z.B. Erzeugung, Entfernung oder Änderung von Referenzen. Die durch unser Plug-In bereitgestellte Animation erleichtert dabei das Verfolgen von Veränderungen. Werden interessante Objekte selektiert (hier helfen wieder Tooltips und Farbgebung), so kann eine Selektions-Fokussierung während der Animation diese Änderungsverfolgung noch weiter erleichtern.

Manchmal sind nur bestimmte Teilintervalle einer solchen Snapshot-Sequenz interessant, oder einzelne Snapshots redundant. Mit Hilfe des Auswahl-Mechanismus des *Snapshot Players* ist so eine Beschränkung der Animation auf relevante Teile möglich, ohne Snapshots (die vielleicht doch noch interessant für den Debugging-Prozess werden könnten) zu verwerfen.

Ist eine Animation zu komplex, um sie sofort vollständig zu erfassen, kann die Repeat-Funktion des *Snapshot Players* von Nutzen sein. So kann der betrachtete Ablauf (mit einer für den Betrachter als angenehm empfundenen Abspielgeschwindigkeit) immer wieder zyklisch abgespielt werden, was auch ein nur phasenweise einsetzendes Erfassen der Gesamtszene erlaubt.

Oftmals ist nicht nur die Betrachtung eines einzelnen Programmablaufes interessant; zusätzliche Informationen können insbesondere durch den Vergleich unterschiedlicher Durchläufe eines Programms mit geänderten Ausgangswerten gewonnen werden. Unterstützung erhält der Nutzer an dieser Stelle durch das Plug-In, das ermöglicht, Sequenzen von Snapshots zu speichern und zu einem späteren Zeitpunkt wiederzugeben. So können ohne großen Aufwand mehrere Läufe eines Programms oder Programmteils miteinander verglichen werden, indem die erzeugten Snapshot-Sequenzen hintereinander abgespielt werden. Diese Vergleichsmöglichkeit übersteigt die Funktionalität der meisten textuellen Debugger und bedeutet für das Plug-In somit einen echten Mehrwert. Außerdem können (beispielsweise im Rahmen eines Vortrags) Snapshots aus Sequenzen entfernt und neue Sequenzen aus Bestehenden zusammengesetzt werden. Auf diese Weise ist auch der Export einer Sequenz mehrerer alternativer Programmabläufe in einen 2D-Film denkbar.

18.4 Beispiele aus der Praxis

Der mögliche Nutzen des Plug-Ins wurde schon während der Entwicklung an kurzen, praxisorientierten Beispielen getestet, um Funktionalität und Usability zu prüfen und eine Zwischenpräsentation vorzubereiten. Einige Erfahrungen mit diesen Tests dienen im Folgenden als Beispiel für die Einsatzbereiche unseres Plug-Ins in der Praxis.

18.4.1 Auto

Zu Demonstrationszwecken unserer Software wurde ein kleines Testprogramm geschrieben, in welchem ein Auto objektorientiert modelliert wird. Dieses Beispiel wurde gewählt, da die dazu nötigen Objektbeziehungen intuitiv verständlich sein sollten. Konkret instanziiert wurden zwei Objektstrukturen, die in den beiden folgenden Abbildungen jeweils in der linken bzw. rechten Bildhälfte zu sehen sind. Modelliert wurde ein Auto jeweils durch ein Objekt vom Typ `Auto` (blau), das auf je ein `Fahrwerk` (grün), einen `Motor` (lila), einen `Kofferraum` (türkis) und ein Array vom Typ Sitz (grau) verweist. Dieses Array wiederum beinhaltete vier Objekte des gleichen Typs, die gemeinsam auf ein Objekt `Stoff` (hellblau) verweisen. Das `Fahrwerk` verweist auf vier Objekte des Typs `Rad` (rot). Der `Motor` verweist auf ein Array vom Typ Zylinder (gelb) mit zwölf Objekten dieses Typs. In Abbildung 18.1 fällt dabei sofort auf, dass die Teilstrukturen links und unten im Bild nicht zusammenhängend sind, da das Fahrwerk nicht mit dem Auto verbunden ist.

Ein zweiter Snapshot wurde einen Programmaufruf später gemacht. Nun war das Fahrwerk korrekt „eingehängt“. Beide Teilstrukturen sind isomorph, also bezüglich Zusammengehörig-

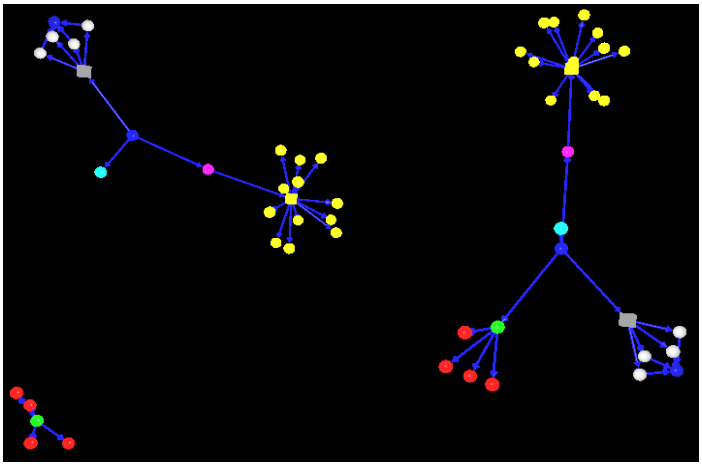


Abbildung 18.1.: Das Auto-Beispiel: Die linke und die untere Teilstruktur sind nicht zusammenhängend

keit von Strukturelementen (Objekte) identisch. Die Visualisierung durch unser Plug-In ist in Abbildung 18.2 dargestellt.

Während das Testprogramm entworfen wurde, zeigte sich der praktische Nutzen des Plug-Ins: ein Auto war „programmiert“ worden - fehlerfrei, wie wir meinten - *3D Debug* lieferte jedoch die in Abbildung 18.3 dargestellte Visualisierung.

Zuerst konnten wir uns die „frei in der Luft hängenden“ Räder nicht erklären. Es bedurfte mehrfachen Lesens der entsprechenden Stelle im Quelltext, bis uns der Fehler klar wurde. Wir hatten ein Array von Rädern erzeugt, es jedoch nicht dem Fahrwerk hinzugefügt. Dieser fehlende Befehl fiel jedoch nicht auf, da das Fahrwerk ansonsten korrekt initialisiert wurde. Entgegen anderer Autoteil-Klassen wird das Fahrwerk nämlich im Konstruktor der Fahrwerk-Klasse erzeugt. Ein sichtbarer Programmfehler wäre daher erst dann aufgetreten, wenn die seperat erzeugten Räder noch hätten hinzugefügt werden sollen oder wenn Veränderungen dieser Objekte nicht die gewünschten Auswirkungen gehabt hätten. Das Programm lief also zunächst korrekt, und man wäre gar nicht auf die Idee gekommen, einen textuellen Debugger zu benutzen. Doch selbst wenn einer benutzt worden wäre, hätte das überflüssige Array nicht auffallen müssen. Mit unserem Plug-In dagegen fiel die unerwartete Teilstruktur sofort und „intuitiv“ auf.

In beiden Beispielfällen lässt sich die Anwendung des Plug-Ins dem Situationsmodell zuordnen. Die Beobachtungen wurden nicht beim zeilenweisen Betrachten einer kurzen Quelltextpassage getroffen, sondern betraf strukturelle Aspekte eines klassenübergreifenden Programmteils. Dennoch handelte es sich nicht um einen grundsätzlichen Designfehler, sondern lediglich um einen einzelnen falschen oder fehlenden Befehl. Der erste Fall lässt sich allerdings auch dem Top-Down-Modell zuordnen, sofern man nicht einen konkreten Programmierfehler, sondern tatsächlich einen Designfehler damit aufgespürt hätte.

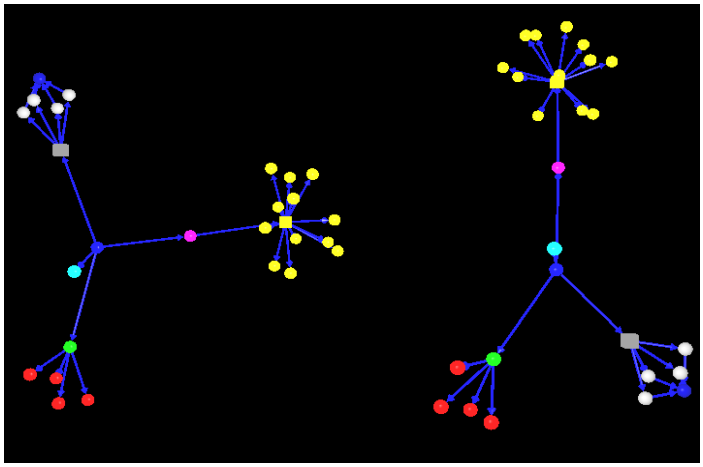


Abbildung 18.2.: Das Auto-Beispiel: Teilstrukturen sind zusammenhängend

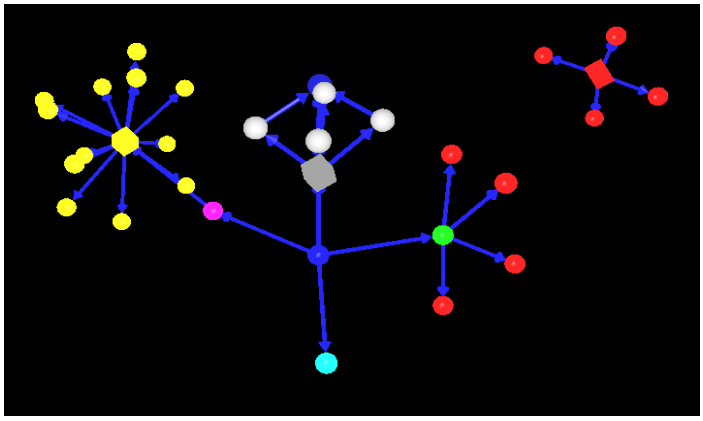


Abbildung 18.3.: Entwicklung des Auto-Beispiels: Wo kommen die freihängenden Räder samt Array her?

18.4.2 Erkennen von Strukturen

Im Rahmen eines Workshops zwischen den beiden PG-Semestern wurden verschiedene Fallbeispiele besprochen, um herauszufinden, in welchen Fällen von Verständnisfragen an ein Programm das Plug-In Hilfestellung leisten kann. Anhand eines Experiments innerhalb der Gruppe konnte festgestellt werden, dass strukturelle Eigenschaften eines Programms in der grafischen Darstellung deutlich besser zu erfassen sind als in einer textuellen Ausgabe. Konkret sollten zwei Testpersonen jeweils fünf Fragen zu einer Debuggingausgabe eines Testprogramms beantworten, bei denen nach Instanziierungen von Objekten gefragt wurde. Es sollten sowohl Fragen zur Zahl von instanziierten Objekten als auch Fragen zu Erzeugungsbeziehungen beantwortet werden. Einer Testperson wurde dazu die textuelle Ausgabe des Eclipse-Debuggers (Abbildung 18.4) vorgelegt und der anderen eine grafische Ausgabe des Plug-Ins (Abbildung 18.5).

```
args= String[0]  (id=11)
test= Test1  (id=13)
obj= Object[4]  (id=14)
    [0]= testObjecte  (id=16)
        inhalt= AnderesObject  (id=21)
    [1]= testObjecte  (id=18)
        inhalt= null
    [2]= testObjecte  (id=19)
        inhalt= null
    [3]= testObjecte  (id=20)
        inhalt= AnderesObject  (id=23)
```

Abbildung 18.4.: Beispielausgabe (textuell) zur Erkennung von Strukturen

Zwei Feststellungen konnten getroffen werden:

- Die Testperson, die die grafische Ausgabe nutzen konnte, war mit der Beantwortung der Fragen schneller fertig.
- Zwei Fragen zu Erzeugungsbeziehungen zwischen Objekte konnten nur von der Testperson mit der grafischen Ausgabe beantwortet werden, da die textuelle Ausgabe die benötigten Informationen nicht zur Verfügung stellen konnte.

Auch wenn dieses experimentelle Ergebnis keineswegs als sicherer Beweis für die Vorzüge des grafischen Debuggings zu sehen ist und durch weitere Experimente mit einem größeren Kreis von Testpersonen untermauert werden muss, so lässt sich doch die These ableiten, dass die grafische Ausgabe des Plug-Ins im Bereich der strukturellen Details eines Programms tatsächlich einen Mehrwert gegenüber textuellen Ausgaben liefert, da sie schneller erfasst werden kann und bei geeigneter Darstellung mehr Informationen enthält. Auch diese Anwendung ist weitgehend dem Situationsmodell oder dem Top-Down-Modell zuzuordnen.

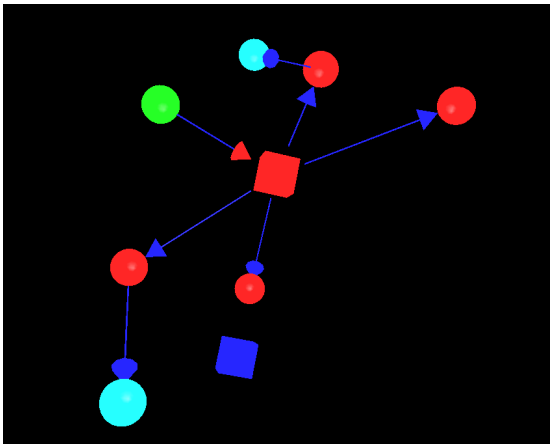


Abbildung 18.5.: Beispielausgabe (grafisch) zur Erkennung von Strukturen

18.4.3 Algorithmus

Als ein praktisches Beispiel soll nun eine falsch implementierte doppelt-verkettete Liste dienen. Einige Methoden sind im folgenden Code-Abschnitt dargestellt.

```
1  public class List {
2
3      Element head;
4      Element tail;
5      Element current;
6
7      public List(Object firstContent) {
8          current=new Element();
9          current.content=firstContent;
10         head=current;
11         tail=current;
12     }
13
14     public Object get() {
15         return current.content;
16     }
17
18     public void add(Object content) {
19         Element element=new Element();
20         element.content=content;
21         tail.next=element;
22         element.last=tail;
23         tail=element;
24         current=tail;
25     }
26
```

```
27      public boolean moveToNext() {
28          if (current.next==null) {
29              return false;
30          } else {
31              current=current.next;
32              return true;
33          }
34      }
35
36      // ...
37
38  }
```

Die Liste enthält mindestens ein Element. Außerdem hält sie Referenzen auf das erste und das letzte Element. Entsprechende Zugriffsmethoden existieren, sind jedoch im Listing nicht enthalten. Der eigentliche Inhalt des Elements wird durch die `Element`-Klasse gekapselt.

Zusätzlich wurde eine remove-Methode implementiert. Diese enthält zwei Fehlerstellen: Einige Zeiger auf das zu löschende Element werden nicht entfernt. Das folgende Listing zeigt den fehlerhaften Code.

```
1     public boolean remove() {
2         if ((current==head)&&(current==tail)) {
3             return false;
4         } else if (current==head) {
5             current=current.next;
6             head=current;
7         } else if (current==tail){
8             current=current.last;
9             current.next=null;
10            tail=current;
11        } else {
12            current.last.next=current.next;
13            current=current.next;
14        }
15        return true;
16    }
```

Treten bei der Verwendung der Liste Fehler auf, so muss die Suche nach diesem nicht unmittelbar zur remove-Methode führen. Ein Blick in einen textuellen Debugger kann unter Umständen nur wenig hilfreich sein. Abbildung 18.6 verdeutlicht dies. Abbildung 18.7 zeigt die 3Debug-Ausgabe für denselben Haltepunkt eines Testprogramms. Eine gestörte Struktur ist offensichtlich - zum Vergleich zeigt Abbildung 18.8 eine korrekt verkettete Liste. In derartigen Fällen ermöglicht unser Plug-In also eine wesentlich schnellere Erfassung des Problems. Auch im Fall, dass die Liste korrekt gewesen wäre, hätte man dies schnell feststellen können. Die Zeitersparnis ist also in beiden Fällen deutlich spürbar.

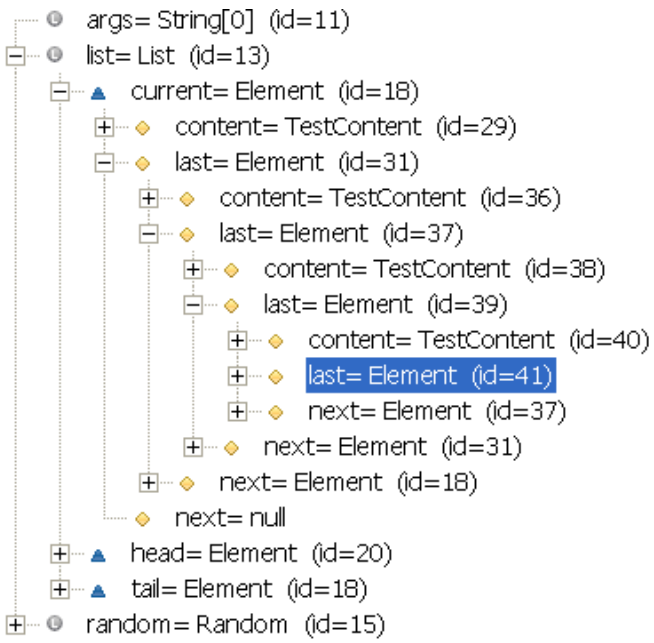


Abbildung 18.6.: Die Liste im textuellen Debugger

Die korrigierte Version des Code-Fragments ist im folgenden Listing enthalten.

```
1     public boolean remove() {
2         if ((current==head)&&(current==tail)) {
3             return false;
4         } else if (current==head) {
5             current=current.next;
6             current.last=null;
7             head=current;
8         } else if (current==tail){
9             current=current.last;
10            current.next=null;
11            tail=current;
12        } else {
13            current.last.next=current.next;
14            current.next.last=current.last;
15            current=current.next;
16        }
17        return true;
18    }
```

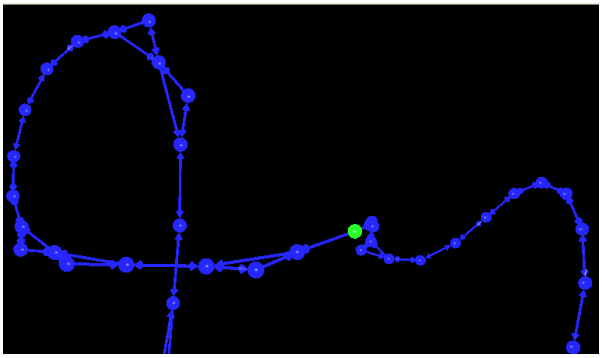


Abbildung 18.7.: Fehlerhafte Verkettung

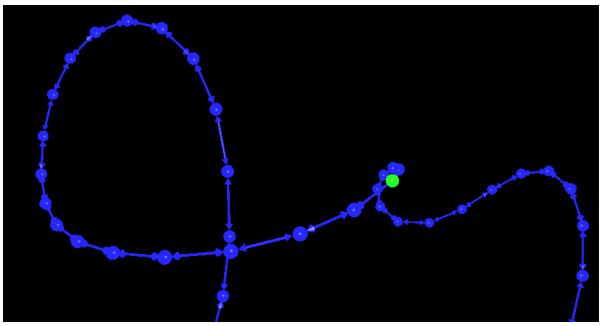


Abbildung 18.8.: Korrekte Verkettung

KAPITEL 19

Grenzen des Plug-Ins

Boris Brodski, Boris Düdder

Dieser Abschnitt behandelt die Grenzen des realisierten Plug-Ins. Diese Grenzen sind die Folge von technischen als auch physiologischen Beschränkungen.

19.1 Kriterien

Es existieren verschiedene Kriterien, die uns Grenzen für die Informationsbeschaffung und für die Darstellung und somit für die Verwendbarkeit im Alltag gesetzt haben. Zu diesen Kriterien zählen insbesondere die menschliche Auffassungsgabe, die Systemressourcen, die technischen Grenzen der Informationskollektoren der virtuellen Maschine. Innerhalb des Projektes versuchten wir die Grenzen der jeweiligen Verfahren (Informationsbeschaffungsalgorithmen, sowie Anordungsalgorithmen) zu untersuchen und nach Möglichkeit auszuloten bzw. mit alternativen Verfahren zu überwinden.

19.2 Darstellung

Die Darstellung ist ein zentraler Punkt unseres Plug-Ins. Dabei stellte sich die Informationsdarstellung von großen Datenmengen als kompliziert und sehr ressourcenhungrig heraus. Wir verwenden einen dreidimensionalen Raum für die Darstellung. Es hat sich herausgestellt, dass bei über 100 Objekten die Übersichtlichkeit drastisch sinkt. Dies hängt aber auch sehr stark von den verwendeten Anordnungsalgorithmen ab.

Für die Anordnung von geometrischen Primitiven existieren nur vier Freiheitsgrade. Dies sind die kontinuierlichen drei Raumkoordinaten und die drei diskreten Farbwerte (ein Farbwert, z.B. RGB-Wert als Festkommazahl). Ein zusätzliches Problem besteht darin, dass man die Klassen anhand ihrer farbigen Repräsentation unterscheidet, Objekte derselben Klassen aber die gleiche Farbe und Form und somit eine identische graphische Repräsentation besitzen

und aus diesem Grund schwierig zu unterscheiden sind. Schwierig ist auch eine konsistente reichhaltige Farbgebung bei vielen verschiedenen Klassen, wobei diese noch unterscheidbar sein sollen.

Zusätzlich ist es problematisch, sich in dem virtuellen Universum zu orientieren. Da es möglich ist, sich beliebig im dreidemensionalen Universum zu bewegen, stößt man auf das Problem, früh gefundene Objekte und Relationen zwischen Objekten in aufeinander folgende Snaphshots wiederzufinden. Als eine mögliche Lösung wurde ein optionales Hintergrundbild eingebaut. Allerdings hat es sich als problematisch erwiesen, ein optimales Bild für den Hintergrund zu finden. Der Hintergrund sollte nicht zu bunt sein, trotzdem sollte es möglich sein, die verschiedenen Stellen des Hintergrunds zu erkennen und zu unterscheiden, um sich daran orientieren zu können.

19.3 Ressourcen

19.3.1 CPU

Das Plug-In benötigt viel CPU-Zeit. Zum Einen für die Sammlung sowie Filterung der Debug-Informationen, die von den angeschlossenen Kollektoren geliefert werden, zum Anderen für die graphische Darstellung. Durch Verwendung eines 3D-Beschleunigers (Grafikkarten mit GPU) kann man die CPU bei der Darstellung der graphischen Objekte teilweise entlasten. Dies bringt vor allem Vorteile für die Animation. Speziell der JDI-Kollektor benötigt zusätzliche CPU-Zeit für die permanente Überwachung des Debuggees.

19.3.2 Speicher

Das Plug-In benötigt viel Hauptspeicher aufgrund der drei-dimensionalen Szenen, die viele graphische Primitive respektive Objekte beinhalten. Dies liegt in unserer Verwendung von Java3D begründet.

Zusätzlich ist es notwendig den maximalen Heapspace von Eclipse zu modifizieren, um eine bessere Performanz zu erreichen.

19.3.3 Speicherung

Um ein möglichst universelles Datenaustauschformat zu unterstützen, verwenden wir zum Export und zur permanten Speicherung von Snapshot-Informationen XML. Dies ist ein Standard zur Erstellung von maschinen- und menschenlesbaren Dokumenten in einer Baumstruktur. Der Nachteil an diesem Format ist, dass es nicht unbedingt platzsparend ist.

Um diesem Missstand Abhilfe zu schaffen, unterstützt 3Debug zusätzlich ein proprietäres binäres Speicherformat.

19.3.4 Suboptimale Ausnutzung der CPU

Ein weiteres Problem ist die suboptimale Ausnutzung der CPU durch einige Kollektoren sowie der Java Virtual Machine (JavaVM). Ein Grund für diese Performanzprobleme sind durch Synchronisationsprobleme und die Verwendung von Netzwerkprotokollen zur Kommunikation mit dem Debuggee zu erklären.

19.3.5 Performanzverlust durch Netzwerkprotokoll (JDWP)

Die meisten verwendeten Informationskollektoren verwenden das *Java Debug Wire Protocol* (JDWP) als grundlegendes Kommunikationsprotokoll. Da bei jeder Kommunikation der komplette Protokollstapel durchlaufen wird, ist die transferierte Datenmenge um einiges größer als die transferierte Informationsmenge. Zusätzlich wird der Kommunikationsprozess auch durch CPU-Nutzung sowie die Sicherheitsüberprüfungen gebremst.

19.4 Informationskollektoren

Jeder Informationskollektor hat besondere Stärken und Schwächen. Der Übersicht halber sind diese hier als Eigenschaftstabelle der Kollektoren (s. Tabelle 19.1) dargestellt. Dabei repräsentiert ein ($\surd$) das Vorhandensein dieses Features. Ein ($+$) besagt, dass diese Eigenschaft vorhanden ist und akzeptabele Charakteristiken (Laufzeit, Speicherbedarf) hat. Ein ($-$) besagt, dass die Eigenschaft zwar vorhanden ist, sie aber nicht akzeptabele Charakteristiken besitzt.

19.4.1 Recursive Snapshot Collector

Der Recursive Snapshot Collector verwendet eine Schnittstelle von Eclipse zum Debuggen. Die in der *Choose Classes View* ausgewählten Objekte werden gesammelt, indem man über Aufrufstapel aller Threads iteriert und die Referenzen auflöst. Die gesammelten Objekte werden rekursiv auf weitere Referenzen untersucht, wobei nur Objekte von ausgewählten Klassen betrachtet und Schleifen in den Referenzen eliminiert werden. Die Auswahl von vielen Klassen bedingt eine hohe Auslastung der CPU. Eine kleine Auswahl von Klassen stellt gegebenenfalls nicht die Gesamtheit aller Objekte der ausgewählten Klassen dar. Daher ist eine geschickte Klassenauswahl nötig, um alle erwünschten Objekte zu sammeln.

	Snapshotcollector	Snapshotcollector (optimiert)	NetDumper	JDI Collector
Snapshot Erzeugungsgeschwindigkeit	+	−	+	+
Ausführungsgeschwindigkeit	+	+	+	−
Findet alle Objekte		√	√	√
Arrays	√	√	√	
Kompatibilität	+	+	−	+
Creator-Createe-Relation				√
Kein externer Puffer	√	√		√
Große Programme	+	−	+	−

Tabelle 19.1.: Eigenschaftstabelle der Kollektoren

19.4.2 Recursive Snapshot Collector (optimiert)

Eine Optimierung bezüglich der Anfälligkeit des vorhergehenden Kollektors gegenüber der Auswahl der Klassen stellt der Recursive Snapshot Collector (optimiert) dar. Dieser sammelt alle Objekte der ausgewählten Klassen, wobei man über alle bestehenden Referenzen iterieren und daher eine hohe Verwendung der CPU-Zeit in Anspruch nehmen muss.

19.4.3 NetDumper Collector

Beim NetDumper handelt es sich um ein inoffizielles Tool von SUN. Dieses existiert für verschiedene Plattformen und verwendet zur Übertragung der Debuginformationen ein propietäres Protokoll. Da es sich nicht um eine standardisierte Technologie handelt, besitzt der NetDumper Collector ein begrenztes Einsatzgebiet. Somit ist auch die zukünftige Wartung dieses Produktes durch SUN und eine Unterstützung von anderen JavaVM's nicht zwingend gewährleistet, da es sich nicht um eine Referenztechnologie handelt. Die Debug-Informationen werden bei Bedarf von der JavaVM in einer speziellen Datei abgelegt und von dem NetDumper Collector importiert. Diese Schnittstelle liefert alle angefordeten Objekten zu einem bestimmten Zeitpunkt.

Bei der Verwendung dieser Schnittstelle kommt es zu einer suboptimalen Nutzung der CPU, weil durch eine nicht ausreichende Synchronisierung unerwünschte Wartezeiten entstehen.

19.4.4 JDI Collector

Der JDI Collector basiert auf einem Bestandteil der *Java Debug Architecture* (JDA), dem sogenannten *Java Debug Interface* (JDI). Dieses Interface existiert als Referenzimplementierung von SUN und wird auch von anderen entsprechenden Engines (IBM Blackdown, etc.) zur Verfügung gestellt. Diese Technologie verwendet ein verbindungsorientiertes Kommunikationsprotokoll, JDWP, um mit dem Remote-Debuggee zu interagieren.

Vor dem Start des Debuggees werden Methodenbreakpoints gesetzt, die es erlauben, Objektinstanziierungen zu überwachen. Ein negativer Seiteneffekt dieses Vorgehens ist das permanente Anhalten und Fortsetzen des Debuggees, wenn die ausgewählten Objekte aktiv sind. Trotz der geschickten Konfiguration des Methodenbreakpointfilters ist es nicht gelungen die Ausführungsgeschwindigkeit auf ein akzeptables Niveau zu bringen. Ein Vorteil dieses Kollektors ist die hohe Geschwindigkeit, mit der der Snapshot geliefert wird. Ein Nachteil ist, dass diese Schnittstelle keine Informationen über Arrays liefern kann. Der JDI Collector kann als einziger Kollektor Informationen bezüglich der Creator-Createe-Relation erfassen.

19.5 Allgemeine Verwendbarkeit

Die Verwendbarkeit des Plug-Ins ist sehr stark von der Wahl des Kollektors, der Komplexität sowie der Größe des zu debuggenden Programms und auch der Performanz der graphischen Komponenten abhängig. Zuallererst ist es notwendig, dass der Nutzer über die Art der Informationen, gemäß der Featuretabelle, die zu beobachteten Klassen und die gewünschte graphische Repräsentation reflektiert. Eine unbedachte Wahl dieser Variablen führt zu einem äußerst zeitintensiven oder sogar zu einem zeitverschwenderischen Vorgang. Die frühe Festlegung auf möglicherweise relevante Informationen zu Beginn des Debuggingprozesses ist problematisch, da oft erst im Verlauf des Debugging klar wird, welche Informationen überhaupt interessant sind. Eine zu enge Begrenzung der Informationen führt gegebenfalls dazu, dass der Prozess mit einer erweiterten Begrenzung wiederholt werden muss. Falls die Begrenzung zu grob war, muss der Prozess wiederholt werden, da eine unübersichtliche Informationsflut zurückgeliefert wird.

3Debug ist in der Lage, sowohl *Exception*-Klassen als auch ihre Erstellung (throwing) anzuzeigen. Jedoch ist es nicht in der Lage zu lokalisieren, welche Klasse die Exception geworfen hat.

Sowohl der *NetDumper Collector* als auch der *Recursive Snapshot Collector* sind nicht in der Lage, die für viele Aufgabenstellungen relevanten Creator-Createe-Relationen zu erfassen. Der *JDI Collector* kann Felder sowohl komplexer als auch primitiver Datentypen nicht erfassen. Da Felder für viele Algorithmen notwendig sind, ist der Einsatz von 3Debug mit diesem Kollektor nur bedingt sinnvoll.

KAPITEL 20

Bedeutung der Anordnungsalgorithmen

Carina Klar, Michael Striewe

Bei der praktischen Erprobung des Plug-Ins zeigte sich sehr schnell, dass der Nutzen einer dreidimensionalen Darstellung von Laufzeitinformationen im Kontext des Debugging sehr stark vom verwendeten Anordnungsalgorithmus abhängt. Für die Darstellung eines einzelnen Snapshots stand zunächst eine sinnvolle Anordnung der Objekte im Raum im Vordergrund. Schon bei wenigen Objekten zeigte sich, dass die Kriterien, wann eine Anordung sinnvoll ist, sehr unterschiedlich sein können. Beispielsweise ist zur Beobachtung der schrittweisen Konstruktion einer Objektstruktur eine Anordnung wichtig, in der neue oder veränderte Relationen leicht beobachtet werden können. Zur Feststellung zentraler Klassen in unbekanntem Code ist dagegen eine Gruppierung der Objekte nach Klassenzugehörigkeit hilfreich. Zur Identifizierung von Objekten mit vielen Relationen ist eine Anordnung sinnvoll, die diese Objekte zentriert und abhängige Objekte möglichst symmetrisch um sie herum anordnet. Aufgrund der Vielfalt an Zielen und weil ein Anordnungsalgorithmus alleine nicht alle dieser Ziele abdecken kann, haben wir uns dafür entschieden mehrere Anordnungsalgorithmen zu implementieren und darüberhinaus eine Schnittstelle zu entwerfen, die es ermöglicht weitere Anordnungsalgorithmen zu implementieren und zu nutzen. Die von uns implementierten Anordnungsalgorithmen decken dabei unterschiedliche Schwerpunkte ab und sind vom Nutzer nach seinen Wünschen parametrisierbar. Das bedeutet die Anordnungsalgorithmen sind so gestaltet, dass der Nutzer über einige Parameter den Anordnungsalgorithmus auf seine spezielle Situation einstellen und somit den Fokus des Anordnungsalgorithmus innerhalb seines Schwerpunktgebietes variieren kann.

Als grundsätzliche Forderung lässt sich festhalten, dass die 3D-Objekte im Verhältnis zur Zahl der Objekte nicht zu weit auseinander, aber auch nicht zu dicht beieinander liegen sollen. Im ersten Fall verliert man zu schnell den Überblick über Gesamtzusammenhänge bzw. muss zu weit scrollen, um Relationen zwischen weit entfernten Objekten zu verfolgen. Im zweiten Fall werden einzelne Objekte dagegen zu leicht durch andere Objekte oder Anhäufungen von Pfeilen verdeckt. Zusätzlich zur Optimierung der Abstände zwischen Objekten sollten die Positionen so berechnet werden, dass Pfeile möglichst einfach zu verfolgen sind und keine unnötigen Kreuzungen entstehen. Ein allgemeiner Ansatz für Anordnungsalgorithmen ist es daher, Objekte mit vielen Beziehungen zueinander nahe beieinander anzuordnen, um Pfeillängen zu minimieren und einen groben Überblick über Objektzusammenhänge zu bieten.

Für die Darstellung von Animationen zur expliziten Nachverfolgung des Laufzeitverhaltens spielt es auch eine wichtige Rolle, dass identische Objekte in aufeinander folgenden Snapshots möglichst an derselben Position angezeigt werden, denn die Übersichtlichkeit wird durch unvorhersehbare Positionswechsel von unveränderten Objekten beim animierten Übergang zwischen zwei Snapshots negativ beeinflußt oder geht ganz verloren. Bereits bei der statischen Darstellung ist diese Positionstreue von Bedeutung, da der Nutzer die Möglichkeit für vergleichende Programmabläufe erhalten soll und somit identische Programmsituationen im Idealfall deterministisch zur selben Darstellung führen sollten.

Neben diesen Anforderungen war weiterhin zu beachten, dass die Anordnungen auch bei vielen Objekten und Relationen effizient zu berechnen sein sollten, da zu hohe Laufzeiten die Benutzbarkeit stark einschränken würden.

20.1 Bewertung des LinLog-Algorithmus

Der LinLog-Algorithmus (siehe Abschnitt 15.2.2) ist ein parametrisierbarer Clustering-Algorithmus. Er lässt sich vor allem dann gewinnbringend einsetzen, wenn eine Übersicht über die Zugehörigkeit von Objekten zu Klassen geschafft werden soll. Durch die verschiedenen Parameter können beispielsweise Objekte gleicher Klassen oder Objekte mit vielen Relationen zu Partitionen zusammengezogen werden. Durch diese Häufung von Objekten sind quantitative Abschätzungen zur Zahl der Instanziierungen einer Klasse sehr einfach und es lassen sich rein visuell Anhäufungen von Relationen innerhalb einer Partition oder auch zwischen zwei unterschiedlichen Partitionen durch eine entsprechende Anhäufung von Pfeilen feststellen. Dadurch können besonders einfach wichtige Programmteile identifiziert werden, weil diese in der Regel durch eine große Anzahl anderer Objekte referenziert werden. „Hilfsklassen“, gekennzeichnet durch wenige Instanziierungen oder Relationen, werden an den Rand der Anordnung gedrängt. Durch eine solche Randlage wird deutlich symbolisiert, dass es sich um eine Hilfsklasse handelt. Somit wird beispielsweise die Einarbeitung in unbekannten Code unterstützt. Das Beispiel in Abbildung 20.1 zeigt die Häufung der Instanziierung von drei unterschiedlichen Klassen. Im oberen Bereich der Abbildung ist eine Ansammlung von roten Objekten, im unteren Bereich ist eine Ansammlung von hellgrauen Objekten und in der Mitte, leicht auf der linken Seite, ist sehr deutlich eine Ansammlung von gelben Objekte zu erkennen. Die weniger häufig vertretenen türkisen Objekte treten dagegen in den Hintergrund, wenngleich sie durch die Rotation zufälligerweise in der Mitte des Bildschrims liegen.

Bei einer großen Zahl von Klassen (über 100 Klassen) im Programm reicht diese Art der Partitionierung allerdings nicht mehr aus, um einen groben Überblick über eine Programmstruktur zu erlangen. Eine Partitionierung nach Paketen, die entweder Klassenzugehörigkeiten nicht berücksichtigt, oder die Durchführung einer zweistufigen Partitionierung, bei dem zunächst Objekte nach Klassen und dann Objektgruppen nach Packeten gruppiert werden, sind aber als Erweiterungen der vorhandenen LinLog-Implementierung denkbar und würden für mehr Übersicht sorgen.

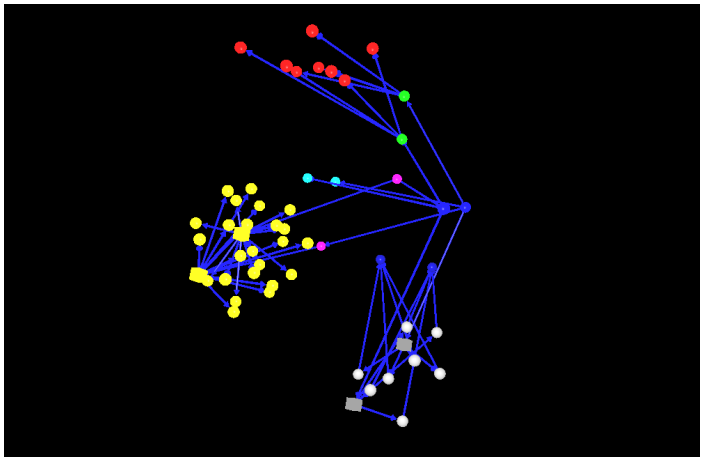


Abbildung 20.1.: Partitionierung durch den LinLog-Algorithmus

Das Gruppieren nach Relationen, das eine optische Trennung von unabhängigen Programmteilen ermöglichen würde, ist zwar vom Prinzip her möglich, allerdings erweist sich in der Praxis die Wahl geeigneter Parameter als sehr schwierig. In diesem Fall ist der Force-Directed-Layout-Algorithmus (Abschnitt 20.2) besser geeignet.

Die im Plug-In eingesetzte Implementierung setzt die konzeptionellen Anforderungen auch für große Objektmengen effizient um. Da der Algorithmus mit einer festen Positionszuordnung startet, ist eine stets deterministische Anordnung gewährleistet, sofern die Datenstruktur die instanziierten Objekte für identische Programmsituationen in derselben Reihenfolge an den Algorithmus übergibt. Dies ist gewährleistet, solange der Algorithmus zur Informationsbeschaffung nicht gewechselt wird und interne Vorgänge der virtuellen Maschine von Java nicht zu einer veränderten Bereitstellung der Laufzeitinformationen führen.
Die allgemeinen Anforderungen zu den Abständen zwischen den Objekten werden dagegen nicht vollständig erfüllt, was sich auch negativ auf die Positionstreue während Animationen auswirkt. Bereits minimale Änderungen an den Parametern sowie in der Zahl der Objekte oder Relationen haben große Auswirkungen auf die Abstände zwischen den 3D-Objekten. Daher ist es sehr schwer, für einen Debug-Lauf über mehrere Breakpoints, in dessen Verlauf sich die Zahl der Objekte verändert, einen optimalen Parametersatz zu finden. In einigen Fällen neigt der Algorithmus dazu, die Abstände in einzelnen Snapshots zu groß oder zu klein zu wählen, was zu einem „Pulsieren“ von Objekthaufen im Raum beim animierten Übergang zwischen Snapshots führt. Eine sinnvolle Beobachtung des Laufzeitverhaltens ist damit nicht mehr möglich.

Als Variante des LinLog-Algorithmus wurde der Fruchterman-Reingold-Algorithmus implementiert. Die beiden Algorithmen unterscheiden sich nur durch den Exponenten zur Energieberechnung. Bei geeigneter, nicht notwendigerweise für beide Algorithmen identischer Wahl der übrigen Parameter ergeben sich damit in der Praxis allerdings kaum nennenswer-

te Auswirkungen auf das Ergebnis der Anordnung, wie die Abbildungen 20.1 und 20.2 zeigen. Abbildung 20.1 wurde mit folgenden Parametern erstellt: 100 Iterationen, Gravitation 0.0 sowie Anziehungskräfte 0.25, 0.15, 0.25 und 0.1. Abbildung 20.2 wurde mit folgenden Parametern erstellt: 100 Iterationen, Gravitation 0.0 sowie Anziehungskräfte 0.2, 0.08, 0.2, 0.08. Die in Noack (2003a) beschriebene Beobachtung, dass LinLog etwas besser clustert als Fruchterman-Reingold, lässt sich damit allerdings bestätigen. Dies spricht jedoch nicht gegen die Verwendung des letzteren, denn in Abbildung 20.2 ist sehr viel einfacher zu erkennen, dass sich die Instanzen der gelben Klasse auf zwei Gruppen (in diesem Fall Arrays) aufteilen, während dies in Abbildung 20.1 aufgrund des stärkeren Clusterings nicht sofort erkennbar ist. Eine geeignete Wahl der Parameter für den LinLog-Algorithmus ermöglicht also sowohl quantitative Analysen zur Zahl der Instanziierungen als auch einfache Analysen der Programmstruktur.

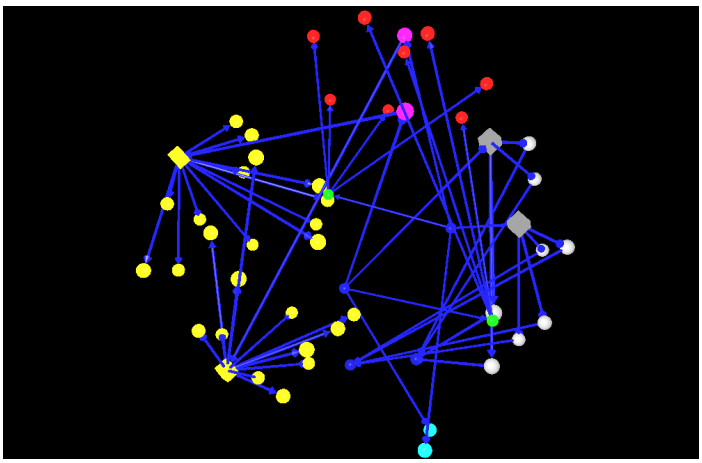


Abbildung 20.2.: Partitionierung durch den Fruchterman-Reingold-Algorithmus

20.2 Bewertung des Force-Directed-Layout

Das Force-Directed-Layout (siehe Abschnitt 15.2.3) ist ein parametrisierbarer Anordnungsalgorithmus. Er basiert auf einem Kräftemodell mit Anziehungskräften entlang der Relationen und einer allgemeinen abstoßenden Kraft zwischen allen Objekten. Dadurch entstehen molekülartige Objektgruppen, die im Idealfall ggf. deutlich voneinander getrennt sind und die eine sehr gute Übersicht über strukturelle Zusammenhänge ermöglichen. Die Konstruktion von Objektstrukturen zur Laufzeit lässt sich damit sehr gut beobachten. Dies ermöglicht sowohl die Einarbeitung in unbekannten Code als auch die schnelle Feststellung von Fehlinstanziierungen und fehlenden oder falschen Relationen. Die Kräfte können für verschiedene Arten von Relationen getrennt definiert werden und erlauben somit eine Anpassung der Darstellung an die Ziele der Beobachtung. In Abbildung 20.3 ist deutlich zu erkennen, wie sich zwei nahezu symmetrische grafische Strukturen für zwei unabhängige Instanziierungen einer komplexen

Objektgruppe ergeben. Das Fehlen von erwarteten Relationen würden in einer solchen Anordnung sofort zu erkennen sein, da sich dadurch insbesondere die Kräfte ändern würden und somit keine Symmetrie zustande kommt (siehe auch Beispiel in Abschnitt 18.4.1).

Eine Gruppierung der Objekte nach Klassen ist dagegen mit diesem Ansatz grundsätzlich nicht möglich, da keine Anziehungskräfte zwischen Objekten der gleichen Klasse aber ohne Relationen definiert werden können.

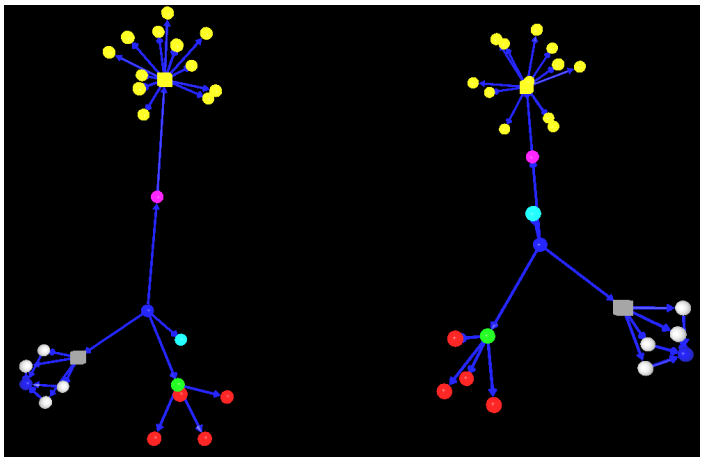


Abbildung 20.3.: Force-Directed-Layout nach 1000 Iterationen

Die Anforderungen an eine sinnvolle Berechnung von Objektabständen und Pfeillängen setzt der Algorithmus sehr gut um und führt auch für mittelgroße Objektmengen (100 bis 200 Objekte) zu übersichtlichen Darstellungen. Als nicht-deterministischer Algorithmus kann das Force-Directed-Layout allerdings nicht garantieren, dass zwei Durchläufe mit derselben Datenbasis zum selben Anordnungsergebnis führen. Daher sind auch die beiden Objektstrukturen in Abbildung 20.3 nicht völlig identisch. Darunter leidet die Übersichtlichkeit der Darstellung. Zudem führt eine ungünstige Wahl der Iterationsanzahl zu schlechten Ergebnissen, wie die Abbildungen 20.3 und 20.4 deutlich zeigen. Beide Abbildungen wurden mit identischen Paramtern und lediglich mit einer unterschiedlichen Zahl von Iterationen erzeugt. Ein brauchbares Ergebnis der Anordnung stellt sich erst nach einer recht großen Zahl von Iterationen ein, ist dann aber keineswegs stabil, sondern kann sich mit weiteren Iterationen noch ändern. Insbesondere erfolgt die deutliche Trennung von Objektgruppen erst nach vielen Iterationen. Verändert sich die Zahl der Objekte von einem Snapshot zum nächsten, so sind die Strukturen ebenfalls nicht stabil und Objekte behalten nicht zwangsläufig ihre Position. Im Extremfall könnten die beiden Objektgruppen in Abbildung 20.3 vollständig ihre Position tauschen. Dies kann in Animationen zum Verlust von Übersicht führen und macht Beobachtungen zum Laufzeitverhalten entsprechend schwierig.

Besonders gut unterstützt der Algorithmus die zentrale Anordnung von Objekten mit vielen Referenzen. In Abbildung 20.3 ist in der oberen Bildhälfte sowohl links als auch rechts

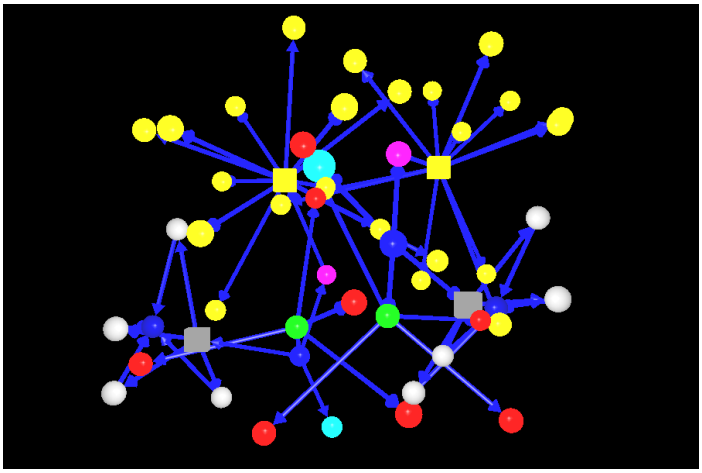


Abbildung 20.4.: Force-Directed-Layout nach 100 Iterationen

deutlich zu erkennen, wie sich die Elemente eines Arrays gleichmäßig um den Array-Würfel verteilen. In der unteren Hälfte ist eine ähnlich strukturierte Anordnung für die Beziehungen zwischen den weißen und dem blauen Objekt sowie dem weißen Array ebenfalls sowohl links als auch rechts sichtbar. Eine solche Struktur lässt sich in einer zweidimensionalen grafischen oder in einer textuellen Debugging-Ausgabe nicht in dieser unmittelbaren Eindeutigkeit darstellen. Bei Ketten von sich gegenseitig referenzierenden Objekten entstehen nach einer geeignet großen Zahl von Iterationen intuitiv erfassbare lineare Strukturen und im Fall von zyklischen Referenzen über mehrere Objekte sogar kreisförmige Anordnungen. Derartige Eigenschaften einer Datenstruktur sind damit deutlich einfacher zu erfassen als in einem weniger strukturierten Objekt-Cluster oder in der textuellen Auflistung von Elementen.

Da die rechnerische Komplexität der Implementierung sehr groß ist, ergeben sich quadratische Laufzeiten in der Größe der Zahl der Objekte. Das führt bei großen Snapshots zu deutlich höheren Laufzeiten als beim LinLog-Algorithmus (Abschnitt 20.1).

KAPITEL 21

Bedeutung der Farb- und Formgebung der Objekte

Michael Striewe, Daniel Vogtland

Im Kapitel 4 wurden verschiedene dreidimensionale Visualisierungskonzepte vorgestellt. Letztendlich entschieden wir uns für eine graphbasierte Visualisierung. Dafür sprachen zwei Gründe:

- Ein graphbasierter Ansatz erfüllt unsere Anforderungen:
  - Die Momentaufnahme eines Systems kann als gerichteter Graph, bestehend aus Objekt-Knoten und Relations-Kanten, aufgefasst werden.
  - Layout-Algorithmen (vgl. Kapitel 20), die eine intuitiv verständliche Anordnung der Visualisierungselemente als Ziel haben, können leicht auf einem Graphen agieren.
- Ein graphbasierter Ansatz ist z.B. aufgrund einer Vielzahl vorhandener Bibiotheken technisch einfach zu realisieren.

Dabei gehen wir über eine WALRUS-ähnliche Darstellung hinaus, indem wir unterschiedliche Formen für Knoten benutzen. Außerdem spielt Farbe innerhalb unserer Visualisierung eine Rolle.

21.1 Bedeutung der Formen

Die Autoren von Irani u. a. (2001) wiesen nach, dass Menschen visuelle Objekte hauptsächlich anhand von formbezogenen Merkmalen klassifizieren. Diesen Umstand sollte eine Visualisierungstechnik berücksichtigen.

Eine entsprechende Darstellungsform für statische Systemansichten wird in Irani und Ware (2003) vorgestellt. Allerdings haben wir es mit (dynamischen) Laufzeitinformationen zu tun.

Die Aufgabe besteht also darin, verschiedene Objekt-Typen zu definieren und entsprechende Darstellungsformen festzulegen.

Klassenzugeörigkeit durch jeweilige Formen zu visualisieren erweist sich als schwierig. Zum Einen müssten dynamisch neue Formen entwickelt werden, da die Anzahl verwendeter Klassen beliebig ist. Diese Formen müssten auch noch gut unterscheidbar sein. Zum Anderen besteht das Problem, dass mit steigender Anzahl der dargestellten Objekte die Objektgröße immer weiter schrumpft und die Formen irgendwann nicht mehr zu unterscheiden sind.

Also wählten wir eine andere Klassifikation der Objekte. Die Objekte sollten nach Funktionalitäten (bezüglich der Java Basissprache) unterschieden werden. Unser Plug-In unterscheidet drei Typen von Objekten:

Array: Arrays von primitiven Datentypen und Objekten. In jeder unterstützenden Programmiersprache dient ein Array als Container. In Java werden Arrays auch als (spezielle) Objekte behandelt. Ein Array wird durch einen *Würfel* dargestellt.

Exception: Von der Klasse `Exception` abgeleitete Objekte. Eine Exception wird von unterstützenden Programmiersprachen zur Fehlerbehandlung innerhalb einer Anwendung verwendet. Eine Exception wird durch eine *Pyramide* dargestellt.

gewöhnliches Objekt: Jedes Objekt, das nicht in eine der beiden vorherigen Klassen fällt. `String`-Objekte werden als primitive Datentypen behandelt und deshalb nicht als Objekt angezeigt. Ein gewöhnliches Objekt wird durch eine *Kugel* dargestellt.

Relationen werden als gerichtete Pfeile dargestellt. Mehrere Relationen zwischen denselben Objekten werden zu einem einzigen Pfeil zusammengefasst. Zwischenzeitlich hatten wir auch mit unterschiedlichen Pfeilspitzen experimentiert, der technische Mehraufwand wurde jedoch nicht durch eine intuitivere Darstellung gerechtfertigt. Auch variable Pfeildurchmesser (abhängig von der Anzahl der Relationen auf die sich der Pfeil bezieht) waren keine Bereicherung und verursachten eher Probleme (zu dünne Pfeile).

Diese Beschränkung auf wenige Objekt-Typen und die einfache Darstellungsform von Pfeilen sind ausreichend für eine Verbesserung der Verständlichkeit einer Darstellung von Laufzeitinformation. Ein zusätzlicher Vorteil dieser Einschränkungen besteht darin, dass Java3D in Form der `Primitive`-Implementierungen direkte Unterstützung bietet.

21.2 Bedeutung der Farben

Wenn auch Farbe nicht primär für die Klassifikation von visuellen Objekten herangezogen wird, so stellt sie doch ein wichtiges sekundäres Attribut dar (vgl. Irani u. a. (2001); Irani und Ware (2003)).

Auch diesen Effekt wollten wir für unser Plug-In nutzen. Die meisten Objekte eines Snapshots, also die Momentaufnahme des zu untersuchenden Systems, werden vom Typ „gewöhnliches Objekt“ sein. Damit haben sie alle die gleiche Form. Allerdings können diese Objekte nach Klassenzugehörigkeit aufgeteilt werden.

Jeder Klasse wird eine Farbe zugeordnet. Jedes Objekt erhält wiederum die Farbe der Klasse, deren Instanz es ist. Eine Ausnahme bilden hierbei Array-Objekte für nicht-primitive Datentypen. Sie erhalten die Farbe der Klasse, deren Instanzen sie als Elemente beinhalten. Auf diese Weise ist eine intuitive visuelle Nähe zwischen Array und Array-Element gegeben, ohne die unterschiedliche Funktionalität zu verbergen, die durch die unterschiedliche Form dargestellt wird.

Die Farben werden durch einen entsprechenden Algorithmus dynamisch erzeugt, wobei durch das menschliche Auge als möglichst hoch wahrgenommene Farbunterschiede angestrebt werden. Innerhalb eines *3Debug*-Laufs wird diese Information nach Erzeugung eines Snapshots und Berechnung seiner Visualisierungs-Information nicht verworfen. Dies ist wichtig, damit Objekte der gleichen Klasse auch in unterschiedlichen Snapshots die gleiche Farbe erhalten.

Die Benutzung des sekundäres Attributs Farbe zur Bestimmung des Objekttyps bietet noch einen anderen Vorteil. Will man sich einen Eindruck der Gesamtstruktur eines Systems machen, so bietet neben den durch Verbindungen zwischen Objekten und durch Layout-Algorithmen berechneten Koordinaten oftmals auch die Klassenzugehörigkeit ein entscheidenes Merkmal zur Analyse. Selbst wenn die einzelnen Objekte sehr klein werden, ist eine Farbunterscheidung (und somit die Klassenbestimmung) noch möglich.

Die Pfeile werden in einheitlicher Farbe gezeichnet. Blau entspricht dabei einer Vorliebe des für die Implementierung Verantwortlichen – es könnte auch eine ähnlich neutrale Farbe wie grau oder braun sein. Allerdings unterscheiden wir zwei Arten von Pfeilspitzen:

- *Rot*: Diese Pfeilspitze beschreibt eine *created*-Relation. Das Objekt, von dem der Pfeil ausgeht, hat das Objekt, auf das die rote Spitze zeigt, erzeugt. Sollte auch eine anders-farbige Pfeilspitze in Frage kommen, z.B. da auch eine Enthaltensein-Relation vorliegt, so wird nur die rote angezeigt, da die *created*-Relation stärker bindet.

- *Blau*: Diese Pfeilspitze beschreibt eine *has-a*-Relation. Das Objekt, auf das der Pfeil zeigt, ist dem Objekt, von dem der Pfeil ausgeht, untergeordnet (z.B. member). Sollte gleichzeitig auch eine *created*-Relation vorliegen, so wird nur die rote Pfeilspitze angezeigt.

Auf diese Weise können verschiedene Typen von Relationen dargestellt werden, ohne unterschiedliche Formen für Pfeile zu verwenden.

21.3 Ausblick

Als Erweiterung für die bisher eingesetzten Möglichkeiten sind verschiedene weitere Ansätze denkbar. Es wäre möglich, die Aussagekraft eines Clusterings von Objekten durch die Einblendung von halbtransparenten Kugeln um große Cluster herum in der entsprechenden Klassenfarbe zu erhöhen. Dies würde auch beim weiten Herauszoomen Übersichtlichkeit garantieren, die bisher durch das Verschwinden der Kugel im Bildhintergrund verloren geht. Auf ähnliche Weise könnte man Gruppen von Relationen zwischen solchen Clustern durch größere Pfeile zusammenfassen. Im Zusammenwirken beider Ideen wird somit von den konkreten Objektbeziehungen abstrahiert, und die allgemeinen Beziehungen zwischen Klassen werden stärker beleuchtet. So ist ein unmittelbarer Vergleich zwischen der konkreten Implementierung und einem Programmkonzept (beispielsweise in Form von Klassendiagrammen) möglich.

Zudem wäre es im Sinne des Bedienkomforts wünschenswert, wenn der Nutzer die Möglichkeit erhalten würde, Einfluss auf die Zuordnung von Farben zu Klassen oder Paketen zu nehmen. Dann wäre es zum Beispiel möglich, einzelnen Programmteilen einen Farbraum zuzuordnen und somit eine noch stärkere visuelle Gliederung der Snapshots zu erzielen. In welchen konkreten Anwendungsfällen des Debugging dies hilfreich ist, ist zu prüfen.

Von besonderem Nutzen bei der Analyse des Kontrollflusses innerhalb eines Programms wäre eine farbliche Hervorhebung der zum Zeitpunkt des Snapshots aktiven Objekte. Dies setzt voraus, dass die Techniken zur Informationsbeschaffung die dazu benötigten Informationen liefern können. Dann würde das Plug-In auch in den Situationen an Wert gewinnen, in denen der Programmablauf keine neuen Objekte erzeugt, sondern nur auf den vorhandenen Objekten arbeitet.
In Anlehnung an die aus textuellen Debuggern bekannte farbliche Hervorhebung von geänderten Variablenwerten ist auch eine farbliche Hervorhebung der zugehörigen Objekte im Snapshot denkbar. Auf diese Weise ließe sich auch der Datenfluss zumindest partiell in einer Folge von Snapshots nachvollziehen.

KAPITEL 22

Erfahrungen mit Java3D

Daniel Vogtland, Henning Zeller

Wir haben uns zu Anfang unseres Projektes dazu entschlossen, dreidimensionale Visualisierungen durch Java3D zu realisieren. Im Folgenden wollen wir diese Entscheidung kurz rückblickend bewerten. Da bereits in Kapitel 8 eine Einführung in die Konzepte von Java3D erfolgte, werden wir hier auf diese nicht mehr detailliert eingehen. Grafische Transformationen wurden in Kapitel 7 erläutert.

22.1 Vorteile von Java3D

Java3D wird kostenlos von SUN zur Verfügung gestellt und bietet eine auf OpenGL oder DirectX aufsetzende Java API zur dreidimensionalen grafischen Ausgabe. Erweiterte Audiomöglichkeiten sind ebenfalls Bestandteil von Java3D, dies war jedoch für unser Projekt nicht relevant.

Java3D ist vollständig objektorientiert. Dies macht das Zusammenspiel mit anderem Java-Code problemlos möglich und ersparte uns die Einarbeitung in eine spezielle Grafikprogrammiersprache und den Wechsel zwischen verschiedenen Programmierkonzepten.

Vorgefertigte Primitive wie Kugeln oder Zylinder stehen zur Verfügung, es können jedoch auch eigene geometrische Figuren definiert werden. Oberflächeneigenschaften können festgelegt und gängige Typen von Lichtquellen verwendet werden. Diese Möglichkeiten reichten für die angestrebte Darstellung von einfarbigen, einfachen geometrischen Objekten sowie zur Komposition von Pfeilen aus.

Transformationen werden über `TransformGroup`-Objekte realisiert, welche mit homogenen Koordinaten arbeiten. Grafische Objekte werden in diese „eingehängt“, und die TransformGroups können hierarchisch geschachtelt werden. Dies macht es extrem einfach, komplexe Transformationen durchzuführen, da man sich dabei jeweils nur auf lokale Probleme beschränken kann. Praktisch alle häufig benötigten grafischen Berechnungen (Matrixmanipulation anhand von Komponenten, Schnitttestberechnungen, Projektionen) sind ebenfalls schon Bestandteil von Java3D und müssen nicht mehr manuell programmiert werden.

Implementierungen der abstrakten `Behavior`-Klasse sind für die Ereignisbehandlung im dreidimensionalen Raum verantwortlich. Einige nützliche Realisierungen für Tastatur- und Mausinteraktion sind bereits vorgefertigt und reichen für unsere Zwecke aus. Lediglich in den Situationen, in denen die Standard-Implementierung zu Performanzproblemen auf Windows-Rechnern führte, mussten eigene Implementierungen erstellt werden.

22.2 Nachteile von Java3D

Java3D ist leider nicht immer zuverlässig: Bei der Funktion des Ausblendens von nicht relevanten Objekten wurde trotz richtiger Werte in den Java3D-Funktionsaufrufen nicht alles korrekt ausgeblendet (dieser Fehler war zudem auch noch nicht-deterministisch). Das Picking eines grafischen Objekts lieferte manchmal ebenfalls trotz korrekter Funktionsaufrufe falsche Ergebnisse. Auch die Überschneidung zweier grafischer Objekte führt mitunter zu Clipping-fehlern mit unschönen visuellen Effekten, was auch als Nachteil angesehen werden muss.

Java3D bietet zwar Plattform-Unabhängigkeit, allerdings sind die verschiedenen Implementierungen der Native-Bibliotheken von unterschiedlicher Güte. So visualisiert die DirectX-Variante unter Windows mitunter Oberflächen besser als die Windows OpenGL-Variante. Das `KeyNavigatorBehavior` führt unter Windows zu einer massiven CPU-Auslastung. Ein weiteres Problem unter Windows besteht in dem manchmal auftretenden „Verschwinden“ des Inhalts der 3D-Sicht, so dass stattdessen nur ein „weißer Screen“ zu sehen ist (z.B. bei Änderung der Größe des umgebenden `Frame` Objekts). Dies lässt sich durch eine Aktualisierung der Szene oder Größenänderung des umgebenden `Frame` Objekts wieder beseitigen, doch so ein Verhalten lässt die Anwendung unprofessionell erscheinen.

Das Hauptproblem von Java3D besteht jedoch auf allen Plattformen in der Performanz. Selman macht in Selman (2002) vor allem den Garbage-Collector für die Performanzprobleme verantwortlich. Da Java3D viele Objekte erzeugt, verzögert sein Aufruf während einer kritischen Rendering-Phase die Ausführung dieser bedeutsam. Dies dürfte vor allem ein Grund für unsere Ressourcenprobleme in Verbindung mit flüssigen Animationen sein.

Java3D bietet zwar viele Funktionen, jedoch vermisst man mitunter die ein oder andere Möglichkeit: Für manche Berechnungen, wie z.B. für den Mittelpunkt eines Objekts oder den Schnittpunkt zweier Linien, wären vordefinierte Funktionen wünschenswert. Außerdem waren die von Java3D mittels `Interpolator` zur Verfügung gestellten Animationsmöglichkeiten nicht nutzbar, da sie nur periodische oder lineare Animation über Java3D Attribute ermöglichen.

Die Dokumentation der Klassen ist zufriedenstellend, sucht man jedoch tiefergehende Informationen, die über die API-Dokumentation und die SUN-Tutorials hinausgehen, so ist die Suche im Internet oft aussichtslos. Dies liegt wahrscheinlich an der geringen Verbreitung von Java3D. Eine Alternative bieten oft nur teure Lehrbücher.

22.3 Verbesserungsmöglichkeiten und Alternativen

Die Dokumentation der Java3D API beschreibt in `Canvas3D` die Methoden und ihre Aufrufreihenfolge bezüglich der Java3D-Rendering-Schleife. Diese Methoden sind überschreibbar und bieten so vielleicht eine Möglichkeit, entsprechend den speziellen Ansprüchen der Anwendung zu optimieren. Es war uns jedoch aus zeitlichen Gründen nicht möglich zu untersuchen, ob so eine Verbesserung der Performanz realisierbar ist.

Als wirkliche Alternative stellt sich die direkte Verwendung von OpenGL (z.B. über die JOGL API oder GL4Java API) dar. Java3D selbst setzt in der entsprechenden Variante auf OpenGL auf.

OpenGL ist schnell, denn die angebotenen Operationen werden meist von der Grafik-Hardware unterstützt (und die CPU dem entsprechend entlastet). OpenGL wird außerdem wegen der breiteren Anwendung schneller weiter entwickelt, was auch ein Argument für die Verwendung sein kann.

Als die Idee entstand, wegen aufkommender Probleme lieber OpenGL statt Java3D zu verwenden, war es jedoch aufgrund des zu weit fortgeschrittenen Entwicklungszeitpunktes nicht mehr möglich, die Visualisierung auf OpenGL umzustellen. Ein gravierender Nachteil von OpenGL besteht nämlich darin, dass keine objektorientierte Programmierschnittstelle angeboten wird. Zwar kann wahrscheinlich alles, was mit Java3D realisiert wird, auch mit OpenGL bewerkstelligt werden, aber es muss selbst implementiert werden. So müsste beispielsweise eine Kapselung der Funktionalität in eine hierarchische Klassenstruktur „von Hand“ geschehen. Es lohnt sich aber, diesen Mehraufwand in Anbetracht der Vorteile von OpenGL als Möglichkeit zur Realisierung von dreidimensionalen Visualisierungen zu erwägen.

KAPITEL 23

Erfahrungen mit eXtreme Programming

Jonas Mathis, Antonio Pedicillo, Michael Striewe

Die Anwendung von eXtreme Programming (XP) als Vorgehensmodell wurde der Projektgruppe von den Betreuern vorgeschlagen und den Teilnehmern durch ein eintägiges Training während der Seminarphase vorgestellt. Keines der Gruppenmitglieder verfügte vor diesem Zeitpunkt über praktische Erfahrungen im Einsatz mit XP. Im Folgenden soll kurz auf die Umsetzung des Vorgehensmodells zurückgeblickt werden.

23.1 Umsetzung der Techniken

Wenn man von eXtreme Programming spricht, wird dies meist mit dem Schlagwort „Pair Programming“ verbunden. Doch die Philosophie von XP besteht nicht allein aus dieser Technik, sondern außerdem noch aus 11 weiteren Techniken, die bei XP zum Einsatz kommen (siehe Abschnitt 12.5). Wenn man rückblickend auf die Entwicklung des Plug-Ins schaut, haben wir nicht alle Techniken von XP vollständig umgesetzt, was allerdings auch weder zwingend vorgeschrieben noch im Rahmen der Projektgruppe sinnvoll umsetzbar gewesen wäre.

Die *Systemmetapher* wurde bei dem Workshop auf der Seminarfahrt festgelegt (siehe Abschnitt 13). Auf diese Metapher haben wir die vier aufeinander folgenden Releases aufgebaut. Sowohl bei der Planung als auch bei der Abnahme der Releases wurde immer wieder Bezug auf die Metapher genommen. Dies hat sich als sehr nützlich erwiesen. Viele Zielsetzungen der einzelnen Releases orientierten sich an der Systemmetapher, indem sie Anforderungen formulierten, die entweder Teile der Metapher direkt realisierten, oder die nötigen Grundlagen schafften.

Das *Planungsspiel* wurde durch den Einsatz der Kundenrolle in den ersten zwei Releases genutzt. Es wurden von jeweils zwei Kunden Userstories erstellt, aus denen dann die Tasks von Entwicklern formuliert und abgeschätzt wurden. Bei dem dritten und vierten Release kam das Planungsspiel nicht mehr voll zum Einsatz, da im dritten Release unter anderem Aufräumarbeiten des zweiten Releases anstanden. Desweiteren sollte nach zusätzlichen Möglichkeiten zur Beschaffung von Debug-Informationen gesucht werden. Dieses Verständnis des dritten

Release ließ nicht viel Spielraum für die Ausarbeitung von Userstories und die Rolle des Kunden. Somit entschieden wir uns, die Kundenrolle im dritten Release nicht einzubringen und das Planungsspiel auf die Zeitabschätzung und Priorisierung der Tasks zu beschränken. Im vierten Release wurde die Rolle des Kunden wieder eingeführt, aber auch hier wurde das Planungsspiel nur teilweise angewendet. Es wurden keine Userstories im eigentlichen Sinne erstellt, sondern die Aufgaben wurden in vier Kategorien eingeteilt. Diese hatten keine Priorisierung, sondern wurden erst durch die Tasks festgelegt.
Die Zeitabschätzung der Tasks zu Beginn unserer ersten beiden Releases waren meist ungenau. Dies lag teilweise an der Unerfahrenheit in den Gebieten Debugging, Eclipse und Java3D, aber auch daran, dass manche Tasks mehrere Teilprobleme besaßen, die im Vorfeld nicht aus dem gegebenen Kontext erkannt wurden, und somit für einen eigenständigen Task zu mächtig waren. Im späteren Verlauf der Projektgruppe funktionierte das Abschätzen der Tasks erheblich besser, da man Erfahrung und Wissen zu den jeweiligen Gebieten aus den vorherigen Releases gewonnen hatte.

In den ersten beiden Releases haben wir die Technik *Kunden vor Ort* genutzt, die allerdings nicht im klassischen Sinne umgesetzt wurde. Laut Vorgaben von XP sollte der Kunde vor Ort den Zweck erfüllen, während der Arbeit am Projekt immer ein Auge auf den aktuellen Stand der Entwicklung zu werfen, um diese zeitnah beeinflussen zu können, sowie den Entwicklern für Rückfragen zur Verfügung zu stehen. Für letzteren Zweck wurden die Kunden bei uns sehr intensiv genutzt. Da die betroffenen Personen aber nicht nur Kunden, sondern zeitgleich auch immer noch Entwickler waren, haben sie zudem stets aktiv am Programmierprozess teilgenommen und besaßen somit auch internes technisches Wissen. Dies erleichterte zum Einen die Kommunikation, ließ aber zum Anderen die deutliche Abgrenzung der Rollen verschwinden, da die Betroffenen teilweise in der Lage waren, gefundene Fehlentwicklungen selber zu beheben.
Im dritten und vierten Release spielte diese Technik keine Rolle mehr in der Projektarbeit, da im dritten Release die Rolle der Kunden nicht vergeben war und sich im vierten Release alle Projektteilnehmer unmittelbar an der Realisierung der Systemmetapher als Ziel der Projektgruppe orientieren konnten. Eventuelle Rückfragen wurden dann in der Gruppe besprochen. Zudem besaßen nun alle Gruppenmitglieder so viel internes Wissen, dass eine deutliche Abgrenzung der Rollen kaum möglich gewesen wäre.

Das *einfache Design* wurde durch den Verzicht auf eine umfangreiche Entwurfsphase gewährleistet. Stattdessen wurde der Entwurf neuer Programmteile bei Bedarf als Task in die Planung eines Releases mit aufgenommen. Dies führte wie zu erwarten war dazu, dass größere Teile des Projekts zu späteren Zeitpunkten überarbeitet werden mussten. Der Mehraufwand beeinflusste die Arbeit der Projektgruppe jedoch nicht negativ. Lediglich beim Erstellen der Dokumentationen der ersten beiden Releases sorgte die nötige Überarbeitung der grafischen Darstellung der Systemarchitektur für einen erheblichen zusätzlichen Zeitbedarf.

Mit zwei Releases pro Semester, für die jeweils eine Entwicklungszeit von etwa sechs Wochen zur Verfügung standen, wurde die Technik der *kurzen Releasezyklen* im Grundsatz umgesetzt. Mit Blick auf die Unerfahrenheit der Projektgruppe mit XP und die Tatsache, dass die Gruppe nicht Vollzeit am Projekt arbeiten konnte, wären kürzere Zyklen nicht sinnvoll gewesen. Die

Aufteilung des Projekts auf mehrere Releases hat uns geholfen, das Endziel schrittweise zu erreichen und dazu beigetragen, dass jedes Mitglied der Projektgruppe einen Überblick über den Entwicklungsstand des Plug-Ins behalten konnte. Außerdem konnte auf diese Weise schnell auf das Feedback der Kunden eingegangen werden.

Die *fortlaufende Integration* wurde durch die Verwaltung des Projekts über das Versions-Kontroll-System CVS gewährleistet. Es stand somit immer allen Teilnehmern eine lauffähige Versions des Produktes zur Verfügung. In der Regel wurden von allen Mitgliedern Projektfortschritte einmal oder sogar mehrmals täglich eingepflegt und selbst bei größeren Programmteilen wie zum Beispiel den neuen Infobeschaffungsalgorithmen aus dem dritten Release erfolgt die Integration einer ersten lauffähigen Version wenige Tage nach dem Beginn der Entwicklung.

Das *Test First*-Prinzip wurde nur in der Anfangsphase der Entwicklung des Plug-Ins ansatzweise genutzt. Viele Mitglieder der Projektgruppe schätzten den Aufwand dafür, vor der Implementierung erst Tests zu schreiben, als zu hoch im Vergleich zum Nutzen ein. Das Fehlen der Tests machte sich selbst in Fällen von umfangreichen Umstrukturierungen im Code oder der Einbindung neuer Programmteile nicht negativ bemerkbar. Da durch die fortlaufende Integration stets ein lauffähiges Produkt als unmittelbare Prüfinstanz existierte, wurden praktisch keine Programmteile unabhängig vom Gesamtprojekt entwickelt, so dass unabhängige Tests damit ebenfalls entbehrlich wurden.

Ein gezieltes *Refactoring* kam im Verlauf des Projekts nur sehr selten zum Einsatz. Die einzig nennenswerte größere Umstrukturierung des Codes unter Beibehaltung der Funktionalität fand im Rahmen des „Code-Reviews“ zwischen dem zweiten und dritten Release statt. Dabei wurde die Effizienz der bis dahin erfolgten Implementierungen überprüft und beispielsweise durch Umstellung von Schleifenkonstrukten oder geeigneteren Datenstrukturen verbessert. In allen anderen Fällen, in denen zu anderen Zeitpunkten Klassen weitgehend oder vollständig überarbeitet wurden, ging dies mit der Hinzunahme von Funktionalität einher. Beispielsweise erhielten die Anordnungsalgorithmen bei einer grundlegenden Bearbeitung ein gemeinsames Interface, welches der Parameterisierbarkeit durch den Nutzer dienen sollte.

Die Technik des *Pair Programming* konnte in unserem Projekt gewinnbringend eingesetzt werden. Durch die gemeinsame Programmierung eines Programmteils war durch die Anwendung des „Vier-Augen-Prinzips“ die Fehlerquote sowohl in der Syntax als auch in der Semantik sehr gering. Die in der Gruppe festgelegten Programmierstandards wurden gut eingehalten, da sich die beiden zusammen arbeitenden Entwickler beispielsweise regelmäßig zum Einbringen von Code-Kommentaren ermahnten.
Allerdings erwies sich die Technik der Paararbeit nicht immer als förderlich im Bezug auf die Arbeitsgeschwindigkeit. Ohne Einfluss auf die benötigte Zeit blieb sie in den Fällen, in denen beide Entwickler sowohl mit der gestellten Aufgabe als auch mit den einzusetzenden Techniken und Programmkonstrukten vertraut waren. Dies war beispielsweise bei der Implementierung der grundlegenen Datenstruktur im ersten Release der Fall, die praktisch keine späteren Korrekturen aufgrund fehlerhafter Implementierung benötigte.
In Fällen, in denen einer der beiden Entwickler parallel zur Programmierung erst in die gestellte Aufgabe oder die zum Einsatz kommenden speziellen Techniken und Konstrukte von Java

einarbeiten musste, führten die daraus resultierenden Nachfragen und Unterbrechungen des Programmierflusses zu Zeitverlust. Dieser Effekt war zu erwarten und wurde dadurch kompensiert, dass das so eingearbeitete Gruppenmitglied später umso produktiver arbeiten konnte. In Fällen, in denen lediglich kleinere Korrekturen an vorhandenem Code vorgenommen oder Fehler gesucht werden mussten, erwies sich die Paararbeit in Verbindung mit den Arbeitsgewohnheiten einiger Gruppenmitglieder als hinderlich. Solche Arbeiten erfordern häufig den schnellen Wechsel zwischen verschiedenen Programmteilen sowie den wiederholten Ablauf von Programmsituationen. Der zuschauende Entwickler hatte dabei in vielen Fällen Schwierigkeiten, seinem Partner zu folgen. Zudem entwickelten häufig beide unterschiedliche Lösungsansätze, die nicht zeitgleich getestet werden konnten. In solchen Situationen war die getrennte Arbeit an zwei Rechnern in der Regel zeiteffizienter.
Grundsätzlich verlief der Wechsel der Mitglieder zwischen Teams nicht immer wie in XP vorgesehen in sehr kurzen Zeitabschnitten, da sich bei uns in Bezug auf einige Programmteile Entwickler mit Domainwissen oder speziellem technischen Wissen herauskristallisiert haben. Somit waren diese Spezialisten in der Regel ständig Mitglied des gleichen Teams und auch der erste Ansprechpartner bei Problemen bezüglich ihres Spezialthemas. Ferner setzte die zeitliche Verfügbarkeit der Gruppenmitglieder den Wechselmöglichkeiten gewisse Grenzen. Paarwechsel fanden daher nicht nach einem festen zeitlichen Schema, sondern an anderen markanten Stellen, z.B. nach Beendigung einer Task, mit Beginn eines Arbeitstages oder nach der Mittagspause statt.

Die *gemeinsame Verantwortlichkeit* konnte in der Projektgruppe weitgehend umgesetzt werden. Wenn vor der Änderung von Programmteilen zunächst Rücksprache mit den bisher daran beteiligten Entwicklern genommen wurde, so lag dies meist an fehlendem Detailwissen über den Code und seltener an einem „Besitzanspruch“ des betroffenen Entwicklers auf „seinen“ Code.

Die Teilnehmer der Projektgruppe einigten sich zu Beginn des Projektes auf *Programmierstandards*, die im Laufe der Arbeit meistens eingehalten wurden. Auf Fehler wiesen sich die Entwickler gegenseitig hin und korrigierten diese. Die Vorteile eines einheitlichen und übersichtlichen Programmierstils waren während des gesamten Projekts deutlich zu erkennen.

Die Technik der *40-Stunden-Woche* wurde nur vom Grundgedanken her umgesetzt, da der Zeitbedarf für eine Projektgruppe als Lehrveranstaltung wesentlich niedriger anzusetzen ist. Der Grundsatz, dass die Gruppe die Arbeit ohne regelmäßige Überstunden erledigen sollte, konnte gut befolgt werden. Auch unmittelbar vor der Fertigstellung der Releases kam es nicht zu Zeitdruck, der sich negativ auf Konzentration oder Motivation der Gruppe ausgewirkt hätte.

23.2 Umsetzung der Rollen

Wie bereits in Abschnitt 12.7.1 angedeutet, mussten für den Einsatz in der Projektgruppe einige Rollen angepasst werden, da in der Lehrveranstaltung Teilaspekte eines realen Projekts

fehlten. Daher mussten einige wichtige Rollen als „Teilzeitaufgabe“ von Mitgliedern der Projektgruppe wahrgenommen werden. Andere Rollen blieben ganz unberücksichtigt.

Vollkommen problemlos und unverändert konnte die Rolle des *Programmierers* in der Projektgruppe umgesetzt werden. Die Mitglieder der Gruppe füllten diese Rolle ständig aus, sofern sie nicht einen Teil ihrer Zeit mit der Verkörperung anderer Rollen verbrachten.

Die hohe Bedeutung, die den *Kunden* im XP-Prozess zukommt, wurde auch in der Umsetzung dieser Rolle in der Projektgruppe deutlich. Außer in Release drei wurden in jedem Release zwei Mitglieder der Gruppe ausgewählt, die die Rolle des Kunden übernahmen. Sie entwarfen in den ersten beiden Releases die User-Stories und begleiteten die Entwicklung des Projekts soweit wie möglich aus dem nötigen externen Blickwinkel. Zudem waren sie für die Aufstellung der Akzeptanztests zum Ende des Releases verantwortlich. Damit wurde von ihnen auch die Rolle der *Tester* verkörpert, die somit nicht explizit an weitere Gruppenmitglieder vergeben wurde.

In den ersten beiden Releases wurde zudem viel Wert auf die Rolle des *Trackers* gelegt. Diese Rolle wurde nacheinander von jedem Gruppenmitglied für jeweils eine Woche ausgefüllt. Der Tracker sammelte für die wöchentlich stattfindende Teamsitzung Daten über den Projektfortschritt und berichtete dort den Betreuern und dem Team. Da die Rolle von jedem Gruppenmitglied einmal ausgefüllt wurde, erhielt jedes Mitglied so einmal die Gelegenheit, sich einen umfassenden und detailierten Einblick in alle Teile der Arbeit zu verschaffen. Trotz dieses Nutzens wurde diese Rolle in den beiden folgenden Releases nicht weiter besetzt. Die Gruppe tauschte sich auch ohne Datensammlung regelmäßig über den Stand des Projektes aus und zudem gab der Protokollführer der Teamsitzungen meist einen zusammenfassenden Bericht über den Stand der Entwicklung.

Die Rolle des *Big Boss* wurde von den Betreuern der Projektgruppe übernommen. Sie gaben der Gruppe steuernde Impulse und griffen lenkend in die Planung der Releases mit ein, um zu hoch oder zu niedrig angesetzte Ziele zu verhindern.

Weitere Rollen wurden nicht explizit besetzt, sondern bei Bedarf von Mitgliedern der Projektgruppe übernommen. Beispielsweise unterrichteten einzelne Mitglieder die Gruppe in Kurzvorträgen, überwiegend während der Anfangsphase, über spezielle Techniken und übernahmen somit kurzzeitig die Rolle des *Consultant*. Die Mitglieder, die in der Seminarphase die Einführung in XP vorbereitet hatten, standen auch während des Projekts bei Bedarf als *XP-Coach* zur Verfügung. Im Allgemeinen machte es sich aber nicht negativ bemerkbar, dass Rollen unbesetzt blieben und somit Teile des XP-Prozesses nur teilweise umgesetzt werden konnten.

23.3 Fazit

Unsere Erfahrungen mit XP führen zu einer differenzierten Bewertung dieses Vorgehensmodells. Für sehr nützlich halten wir es, zu Beginn der Arbeit eine Systemmetapher zu formulie-

ren. Auch für Projekte, die ansonsten nicht XP nutzen, ist es sicher hilfreich, stets ein anschauliches Ziel vor Augen zu haben. Ebenfalls für empfehlenswert halten wir die Unterteilung des Projekts in mehrere kleinere Releases sowie die weitere Aufteilung dieser Releases durch das Planungsspiel. Das Formulieren von Teilaufgaben passender Größe erfordert allerdings bereits ein tiefgehendes Problemverständnis und eine gute Selbsteinschätzung der eigenen Leistungsfähigkeit, welche ungeübten Gruppen zu Beginn fehlen. Dies erschwert die Einarbeitung in XP. Insbesondere führt es zu Konflikten mit dem Prinzip der 40-Stunden-Woche, wenn die geschätzten Bearbeitungszeiten regelmäßig überschritten werden. Es ist dann sehr stark von der Einstellung der Gruppenmitglieder abhängig, ob man an der Abarbeitung der zugeteilten Aufgaben festhält oder die festgelegte Arbeitszeit nicht überschreitet. Die zweite Variante entspräche dem XP-Modell, erscheint aber nicht jedem Gruppenmitglied als befriedigend.

Als deutlich störend und weitgehend überflüssig empfand die Gruppe den Test-First-Ansatz von XP. Die Erfahrung zeigte, dass durch die fortlaufende Integration die vollständige und korrekte Implementierung aller nötigen Schnittstellen bereits gesichert ist und daher auf separate aufwändige Tests verzichtet werden kann. Auch die besondere Hervorhebung von Refactoring als XP-Technik stellte sich als unnütz heraus, denn Code wurde immer nur dann umgestellt, wenn dies für den Einbau neuer Funktionalität nötig wurde. Sonstige Codeänderungen ergaben sich entweder nebenbei „von alleine“ oder es war kein Bedarf dafür vorhanden.

Der Einsatz von Pair Programming hat sich unter Berücksichtigung der oben genannten Einschränkungen als empfehlenswerte Technik erwiesen. Vor allem wurde die Codequalität dadurch positiv beeinflusst. Ob Pair Programming eingesetzt werden sollte, ist allerdings von der gestellten Aufgabe abhängig, da nicht jedes Problem der Bearbeitung durch zwei Programmierer bedarf. In einigen Fällen kann daher sehr viel zeiteffizienter gearbeitet werden, wenn die Paare aufgelöst werden.

Die zu XP gehörende gemeinsame Verantwortung aller Gruppenmitglieder für den gesamten Code ist zwar grundsätzlich gut umzusetzen und hilfreich, scheitert aber immer dann, wenn aufgrund von mangelndem Fachwissen über Programmfunktion oder Implementierung nicht jeder Programmierer jeden Codeabschnitt bearbeiten kann. Allerdings hat es sich in der Praxis als völlig ausreichend erwiesen, wenn zumindest ein großer Teil der Gruppe die nötigen Kenntnisse hatte. In einem Projekt mit vielfältigen Anforderungen wird es sich vermutlich nicht vermeiden lassen, dass jeder Programmierer gewisse Vorlieben entwickelt und daher für bestimmte Programmteile mehr Verantwortung übernimmt.

Zusammenfassend lässt sich feststellen, dass einige Techniken von XP sehr nützlich sind, andere dagegen eher störend. Aus unseren Erfahrungen heraus empfehlen wir, das XP-Modell dementsprechend frei umzusetzen und nur die Techniken zu nutzen, die allen oder den meisten Gruppenmitgliedern als sinnvoll und praktisch umsetzbar erscheinen.

KAPITEL 24

Erfahrungen mit Eclipse

Andrey Lysenko, Antonio Pedicillo

Eclipse ist eine Entwicklungsumgebung, die sich vor allem durch Erweiterbarkeit aufgrund ihrer Plug-In-Struktur von anderen Entwicklungsplattformen unterscheidet. Dank dieser Fähigkeit wird ein hoher Grad an Flexibilität und Konfigurierbarkeit gewährleistet. Da unser Projekt auf keinen Fall einen alternativen Debugger darstellen, sondern viel mehr bestimmte Zusatzfunktionalitäten zum herkömmlichen Debugger anbieten soll, ist unbedingt eine erweiterungsfähige Debuggingplattform wie Eclipse für unser Werkzeug nötig. Aus all diesen Gründen haben wir uns für Eclipse entschieden.

24.1 Vorteile von Eclipse

Die Eclipse-Plug-In-Struktur lässt sich durch Plug-Ins bzw. Projekte beliebiger Größe mithilfe von Extension Points schnell und problemlos erweitern. Der Vorteil dabei besteht darin, dass das Projekt in Eclipse integriert ist und deswegen alle von Eclipse angebotenen Features nutzen kann, die den Programmieraufwand erheblich verringern. Der Overhead, der durch die Entwicklung eines Werkzeugs als Eclipse-Plug-In entsteht, ist gemessen an den dadurch gewonnenen Vorteilen relativ gering gewesen.
Eine weitere nützliche Funktion, mit der wir gute Erfahrungen sammeln konnten, ist unter Eclipse die Startkonfigurtion, die nicht nur alle herkömmlichen Einstellungen wie z.B. JVM-Parameter beinhaltet, sondern auch für einen Entwickler die Möglichkeit bietet, beliebig viele weitere Parameter für einen Benutzer einzubauen. Der Benutzer kann dann die Konfiguration erstellen und für alle späteren Ausführungen speichern oder auch nach Bedarf verändern. Wir nutzen dies beispielsweise dazu, dem Benutzer die Möglichkeit zu geben, die zu verwendeten Anordnungsalgorithmen und die dazugehörigen Parameter einzustellen.

Es lag nahe, unser Plug-In für Eclipse auch unter Verwendung von Eclipse als Programmierumgebung zu erstellen. Wir konnten dabei auch viele positive Erfahrungen als Anwender von Eclipse sammeln. Angefangen bei den nützlichen kleinen Hilfestellungen wie z.B. automatisierte Fehlerkorrektur im Quellcode oder Anzeige von Javadoc-Informationen direkt während

der Codevervollständigung bis hin zu großen und aufwändigen Eclipse-Plug-Ins wie GANYMEDE LOG4J oder JAR CLASS FINDER haben uns viele Funktionen von Eclipse unsere Arbeit erheblich erleichtert. Darüber hinaus beinhaltet Eclipse auch einen CVS-Client, der es für unsere 12 PG-Teilnehmer ermöglicht hat, den Code unabhängig vom Betriebssystem mithilfe des sicheren SSH-Protokolls ständig aktuell zu halten und zu verwalten.

24.2 Nachteile von Eclipse

Als Nachteil von Eclipse kann ein relativ hoher Speicherverbrauch erwähnt werden. Die Eigenschaft ist dadurch bedingt, dass Eclipse versucht, Geschwindigkeitsnachteile von Java durch ständiges Puffern verschiedenster Informationen auszugleichen. Wechselt man bei der Arbeit zwischen Eclipse und anderen Anwendungen, so stellt man bei jeder Rückkehr in Eclipse oft erhebliche Verzögerungen fest. Das Phänomen ist dadurch zu erklären, dass Eclipse dann immer interne Daten wieder in den Hauptspeicher verlagert. Manche Plug-Ins benötigen zudem eine zusätzliche Starteinstellung, damit Eclipse einen bestimmten Speicherbereich reserviert. Da wir auch meistens zwei Eclipseinstanzen gleichzeitig laufen hatten, war der Speicherkonsum besonders hoch.
Unsere im Laufe der Zeit gesammelten Erfahrungen bei der Benutzung von JAVA3D unter Eclipse haben etliche Schwierigkeiten aufgedeckt, weil unser Projekt sowohl mit WINDOWS als auch mit LINUX kompatibel sein soll. Dafür liefert SUN zwei verschiedene Pakete für WINDOWS und für LINUX. Eclipse bietet für solche Probleme, die wegen der Programmentwicklung und -benutzung unter mehreren Betriebssystemen entstehen, keine Lösung. Deshalb wurde unser Projekt in zwei zueinander in Abhängigkeit stehende Plug-Ins unterteilt. Ein Plug-In beinhaltete den gesamten Quellcode und das andere die entsprechende Ressourcen für das benutzte Betriebssystem. Durch Nutzung des zum vom Lehrstuhl 10 entwickelten Tools VISE3D gehörenden Ressourcen-Plug-Ins konnten wir diesen Mehraufwand für einige Nutzer beseitigen. Allerdings blieb insgesamt die Aufteilung in zwei Projekte bestehen zwecks der einfacheren Verwaltung und der klaren logischen Trennung zwischen unserem Quellcode und der zahlreichen in unser Werkzeug eingeflossenen Java-Bibliotheken, die von anderen Entwicklern erarbeitet wurden, wie z.B. das Java Media Framework von SUN.
Ein weiterer Nachteil ist die fehlende Dokumentation bei komplexeren Anwendungen mit Eclipse. So hatten sich am Anfang bei der Kommunikation der Views innerhalb der Perspektive Fehler eingeschlichen. Wurde versucht, eine View mit neuen Informationen zu aktualisieren, kam es zu der Fehlermeldung „Widget disposed“. Dieser Fehler beruhte darauf, dass bei Eclipse nur ein *Display* aktiv sein darf, weil *Display* die Schnittstelle zum Betriebssystem ist und für die Abarbeitung der Ereignis-Schleife sorgt. Deshalb mussten wir alle Vorgänge in Eclipse asynchron laufen lassen, da nicht alle Views gleichzeitig auf das *Display* zugreifen können.

24.3 Fazit

Eclipse hat sich als eine hervorragende Entwicklungsumgebung erwiesen, deren zwar großer aber immer noch erweiterbarer Leistungsumfang sich von dem anderer Entwicklungplattformen sehr stark unterscheidet. Eclipse hat seinen Ruf als eine ausgezeichnete und zuverlässige Hilfe in vielen Fällen bestätigt. Der Einsatz von Eclipse hat sich trotz des hohen Speicherverbrauchs und der manchmal auftretenden Verzögerungen gelohnt.

TEIL 5

Anhang

ANHANG A

Handbuch, Quelltexte und Lizenz

Daniel Maliga

Das Projekt 3DEBUG wird als Open-Source-Projekt unter der *Common Public License* (CPL) auf der Internet-Plattform SOURCEFORGE veröffentlicht, um interessierten Entwicklern sowohl die Nutzung des Plug-Ins als auch dessen Weiterentwicklung zu ermöglichen. Auf der dortigen Projektseite (`http://www.sourceforge.net/projects/threedebug`) ist daher u.a. folgendes zu finden:

- Das Plug-In selbst zum Herunterladen und Installieren;
- die Quelltexte des Plug-Ins;
- ein englischsprachiges Handbuch. Dieses enthält auch den vollen Text der Lizenz.

Literaturverzeichnis

[Beck 2000] BECK, Kent: *eXtreme Programming explained*. Addison Wesley, Reading, 2000

[Biederman 1987] BIEDERMAN, Irving: Recognition-by-Components: A Theory of Human Image Understanding. In: *Psychological Review* 94 (1987), Nr. 2, S. 115–147

[Bouvier 2001] BOUVIER: *Getting started with the Java3D API*. SUN Microsystems, 2001

[CAIDA 2004] CAIDA: *Homepage des Walrus Projekts*. 2004. – http://www.caida.org/tools/visualization/walrus/ (zuletzt gesichtet am 1.8.2005)

[Casey und Exton 2003] CASEY, Ken ; EXTON, Chris: A Java 3D Implementation of a Geon Based Visualisation Tool for UML. In: *Proceedings of the 2nd International Conference on Principles and Practise of Programming in Java*, 2003, S. 63–65

[Dwyer 2001] DWYER, Tim: Three Dimensional UML Using Force Directed Layout. In: EADES, Peter (Hrsg.) ; PATTISON, Tim (Hrsg.): *Conferences in Research and Practice in Information Technology* Bd. 9, 2001

[Eclipse JDT] *Eclipse JDT Help*. – http://www.eclipse.org/documentation/html/plugins/org.eclipse.jdt.doc.isv/doc/reference/api/ (zuletzt gesichtet am 1.8.2005)

[Foote 2004] FOOTE, Bill: *HAT – The Java Heap Analysis Tool*. 2004. – https://hat.dev.java.net/doc/README.html (zuletzt gesichtet am 1.8.2005)

[Furnas 1986] FURNAS, G. W.: Generalized fisheye views. In: *Proceedings on the conference on Human Factors in Computing Systems*. Boston, March 1986, S. 16–23

[Gamma u. a. 1997] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E.: *Design Patterns - Elements Of Reusable Object-Oriented Software*. Addison-Wesley, New York, 1997

[Gaylard und Zeller 2004] GAYLARD, Andrew ; ZELLER, Andreas: *Data Displaying Debugger*. 2004. – http://www.gnu.org/software/ddd/ (zuletzt gesichtet am 1.8.2005)

[Heisenbugs & Bohrbugs 2003] *Heisenbugs and Bohrbugs: Why are they different*. 2003. – http://www.cs.rutgers.edu/~rmartin/teaching/spring03/cs553/papers01/06.pdf (zuletzt gesichtet am 1.8.2005)

[Herman u. a. 2000] HERMAN, Ivan ; MELAN, Guy ; MARSHALL, M. S.: Graph Visualization and Navigation in Information Visualization: A Survey. In: *IEEE Transactions on Visualization and Computer Graphics* 6 (2000), January-March, Nr. 1, S. 24–43

[HPROF 2001] The HPROF Profiler. In: *iPlanet Web Server, Enterprise Edition Programmer's Guide to Servlets* (2001). – http://docs.sun.com/source/816-5689-10/xprof.htm (zuletzt gesichtet am 1.8.2005)

[Irani und Ware 2003] IRANI, Pourang ; WARE, Colin: Diagramming Information Structures Using 3D Perceptual Primitives. In: *ACM Transactions on Computer-Human Interactions* 10 (2003), March, Nr. 1, S. 1–19

[Irani u. a. 2001] IRANI, Pourang ; WARE, Colin ; TINGLEY, Maureen: Using Perceptual Syntax to Enhance Semantic Content in Diagrams. In: *IEEE Computer Graphics and Applications* 21 (2001), September, Nr. 5

[JPDA] *Java Platform Debugging Architecture API*. – http://java.sun.com/products/jpda/ (zuletzt gesichtet am 1.8.2005)

[Lewerentz und Noack 2003] LEWERENTZ, Claus ; NOACK, Andreas: CrocoCosmos - 3D Visualization of Large Object-Oriented Programs. In: JNGER, Michael (Hrsg.) ; MUTZEL, Petra (Hrsg.): *Graph Drawing Software* Springer-Verlag (Veranst.), 2003, S. 279–297

[Lippert u. a. 2002] LIPPERT, Martin ; ROOCK, Stefan ; WOLF, Henning: *Software entwickeln mit eXtreme Programming*. dPunkt-Verlag, Heidelberg, 2002

[von Mayrhauser und Vans 1997] MAYRHAUSER, Anneliese von ; VANS, A. M.: Program understanding behavior during debugging of large scaled software. In: *In Proceedings of the 7th Workshop on Empirical Studies of Programmers* (1997), S. 157–179

[Munzner 2000] MUNZNER, Tamara: *Interactive Visualization of large Graphs and Networks*, Stanford University, Dissertation, June 2000

[NetDumper 2004] *Dump Generator Program*. 2004. – https://hat.dev.java.net/misc/net_dumper/README.html (zuletzt gesichtet am 1.8.2005)

[Noack 2003a] NOACK, Andreas: An Energy Model for Visual Graph Clustering. In: LIOTTA, Guiseppe (Hrsg.): *Graph Drawing* Bd. 2912. Springer, September 2003, S. 425–436

[Noack 2003b] NOACK, Andreas: Energy Models for Drawing Clustered Small-World Graphs. In: *Computer Science Report* 7 (2003)

[Noack 2004] NOACK, Andreas: *Homepage des CrocoCosmos Projekts*. 2004. – http://www-sst.informatik.tu-cottbus.de/CrocoCosmos/ (zuletzt gesichtet am 1.8.2005)

[Radfelder und Gogolla 2000] RADFELDER, Oliver ; GOGOLLA, Martin: On Better Understanding UML Diagrams through Interactive Three-Dimensional Visualization and Animation. In: *Proceedings of the working conference on Advanced visual interfaces*, ACM Press, 2000, S. 292–295

[Rosenberg 1996] ROSENBERG, Jonathan B.: *How Debuggers Work*. 1996

[Selman 2002] SELMAN: *Java3D Programming*. Manning, 2002

[Steimann u. a. 2002] STEIMANN, F. ; THADEN, U. ; SIBERSKI, W. ; NEJDL, W.: Animiertes UML als Medium für die Didaktik der objektorientierten Programmierung. In: *Modellierung 2002* (2002). – GI Lecture Notes in Informatics

[Szurszewski 2003] SZURSZEWSKI, Joe: *We Have Lift-off: The Launching Framework*. Januar 2003. – http://eclipse.org/articles/Article-Launch-Framework/launch.html (zuletzt gesichtet am 1.8.2005)

[Telles und Hsieh 2001] TELLES, Matt ; HSIEH, Yuan: *Science of Debugging*. Coriolis, 2001

[Wikipedia] *Debugging*. – http://de.wikipedia.org/wiki/Debugging (zuletzt gesichtet am 1.8.2005)

[Wright und Freeman-Benson 2004] WRIGHT, Darin ; FREEMAN-BENSON, Bjorn: *How to write an Eclipse debugger*. August 2004. – http://eclipse.org/articles/Article-Debugger/how-to.html (zuletzt gesichtet am 1.8.2005)