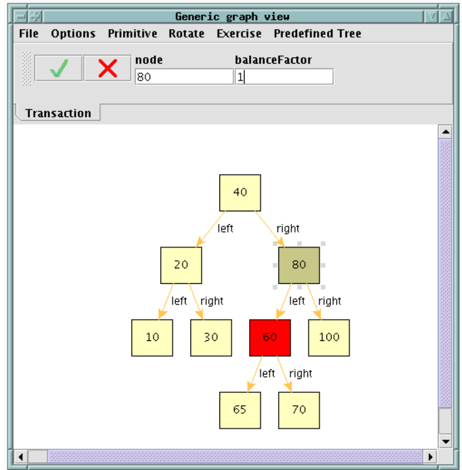


Peter Aschenbrenner

Generierung interaktiver Lerneinheiten aus visuellen Spezifikationen

Generierung interaktiver Lerneinheiten aus visuellen Spezifikationen

Dissertation

Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
der Fakultät für Informatik
der Universität der Bundeswehr München

von

Diplom-Informatiker
Peter Aschenbrenner

1. Gutachter: Prof. Dr. Andy Schürr
2. Gutachter: Prof. Dr. Mark Minas

Datum des Kolloquiums: 10. Februar 2006

Danksagung

Ich danke Herrn Prof. Dr. Andy Schürr dafür, dass er mich ursprünglich für das MuSofT-Projekt – und damit auch für diese Arbeit – ins Boot geholt hat.

Meinen beiden Gutachtern Herrn Prof. Dr. Andy Schürr und Herrn Prof. Dr. Mark Minas danke ich für ihre wertvollen Anregungen.

Frau Hwa Feron danke ich für die Implementierung eines Teiles der ersten VIDEA-Version und für die Teilnahme an der beobachtenden Evaluation der beiden VIDEA-Instanzen.

Meinen Eltern und Freunden danke ich für ihr Vertrauen und ihre moralische Unterstützung.

Außerdem danke ich denjenigen Kollegen, die die Evaluation von VIDEA unterstützt haben: Dr. Lothar Schmitz als dem Verantwortlichen der Veranstaltung und Dr. Jan Scheffczyk, der die Übungen der Kontrollgruppe leitete.

Last, but noch least, danke ich den an den beiden Evaluationsterminen teilnehmenden Studierenden für ihre Mitarbeit – auch beim Ausfüllen der verschiedenen Fragebögen.
Die Rücklaufquote lag bei unglaublichen 100%!

Inhaltsverzeichnis

1. Einführung **1**

2. Didaktischer Ansatz **9**
2.1. Erkenntnisse aus der Lernpsychologie 9
2.2. Flüssige Animationen 11
2.3. Experimente mit Visualisierungen im Allgemeinen 12
2.4. Studien über die Wirksamkeit von Algorithmenanimationen 14
2.5. Ergebnisse einer aktuellen Metastudie 21
2.6. Zusammenfassung 22

3. Anforderungen **25**
3.1. Grundlegendes Szenario 26
3.2. Anforderungen 28
3.3. Beispielszenarien 30
3.3.1. Vorführen im Rahmen einer Präsenzveranstaltung 30
3.3.2. Passives Betrachten vorgegebener Abläufe 33
3.3.3. Interaktives Erforschen und Aufgaben-Lösen 34
3.3.4. Visuelles Debugging eigener Programme der Studenten 35
3.4. Kategorisierung der Anforderungen 36

4. Related Work **41**
4.1. Vorstellung der genutzten Paradigmen 41
4.2. Verdichtung der Anforderungsliste 43
4.3. Vorstellung der Spezifikationsansätze 45
4.3.1. Textuelle Formen der Spezifikation 45
4.3.2. Visuelle Formen der Spezifikation 45
4.4. Vorstellung der untersuchten Animationsansätze 50
4.4.1. Nicht-spezialisierte Animationssysteme für Algorithmen 50
4.4.2. Elektronische Bücher für die Lehre von Algorithmen und Datenstrukturen 52
4.4.3. Algorithmenanimation durch Code-Interpretation 53
4.4.4. API-basierte Graphenbibliotheken 57
4.4.5. Erzeugung von Animationen durch direkte Manipulation 57
4.4.6. Deklarative Systeme für die Animation von Algorithmen 58
4.4.7. Skriptbasierte Algorithmenanimation 60
4.4.8. Visuelle, regelbasierte Entwicklungsumgebung 60
4.4.9. Icon-basierte Simulationssprache 60
4.4.10. Entwicklungsumgebungen für graphische Oberflächen oder Benutzerschnittstellen 60
4.4.11. Sonstige Ansätze für die Erstellung von Animationen 61
4.5. Bewertung gemäß der genannten Anforderungen 62
4.6. Fazit 73

5. Überblick über VIDEA **75**
5.1. Vier beispielhafte Algorithmen und Datenstrukturen . . . 75
5.1.1. AVL-Bäume . . . 76
5.1.2. Doppelt verkettete Liste . . . 78
5.1.3. Der Dijkstra-Algorithmus zur kürzesten Wegesuche . . . 80
5.1.4. Der Kruskal-Algorithmus zur Berechnung des minimalen Spannbaums in einem gerichteten Graphen . . . 81
5.2. Überblick über das VIDEA-Konzept . . . 83
5.2.1. Einsatzgebiete und konkrete Szenarien . . . 84
5.2.2. Abgrenzung von klassischer Lernsoftware . . . 87
5.2.3. Fahrpläne . . . 89
5.2.4. Lernschritte in VIDEA . . . 90
5.2.5. Erstellen einer Spezifikation . . . 95
5.2.6. Verfeinern einer Spezifikation . . . 99
5.2.7. Constraints . . . 100
5.2.8. Aufbrechen von Schleifen . . . 100
5.2.9. Generierung und Konfiguration . . . 102
5.2.10. Bedienung und Oberfläche . . . 104
5.2.11. Zusammenspiel . . . 104

6. Spezifikation **107**
6.1. Einschränkungen und Abgrenzung . . . 107
6.2. Operationen auf Graphen . . . 109
6.3. Implementierung grundlegender Operationen in PROGRES . . . 110
6.4. Spezifikationen für den AVL-Baum . . . 114
6.5. Spezifikationen für die doppelt verkettete, sortierte Ringliste . . . 122
6.6. Spezifikationen für den Dijkstra-Algorithmus . . . 133
6.7. Spezifikationen für den Kruskal-Algorithmus . . . 141

7. Nutzersicht auf vier Typen von Instanzen **149**
7.1. Vorführen im Rahmen einer Präsenzveranstaltung oder Übung . . . 149
7.2. Passive Nutzung von VIDEA . . . 153
7.3. Interaktives Erforschen und Aufgaben-Lösen . . . 156
7.3.1. Eine VIDEA-Instanz für AVL-Bäume . . . 156
7.3.2. Eine VIDEA-Instanz für die kürzeste Wegesuche nach Dijkstra . . . 160
7.3.3. Eine VIDEA-Instanz für Kruskal . . . 161
7.3.4. Eine VIDEA-Instanz für die doppelt verkettete Liste . . . 164
7.4. Ferngesteuerte Animation eigener Programme der Lernenden . . . 164

8. Erzeugung und Konfiguration von VIDEA-Instanzen **173**
8.1. Design einer VIDEA-Instanz . . . 174
8.1.1. Die Unterscheidung verletzbarer und unverletzbarer Eigenschaften . . . 177
8.1.2. Erstellung von Fahrplänen und Lernpfaden . . . 179
8.1.3. Detaillierungsgrade bei der Algorithmenanimation . . . 183

8.1.4. Auswahl der Konstrukte für Hilfsdatenstrukturen 184
8.1.5. Tests 184
8.1.6. Vorbereiten der ferngesteuerten Algorithmenvisualisierung 186
8.2. Generierung und grundsätzlicher Aufbau einer VIDEA-Instanz 189
8.3. Statische Konfiguration 193
8.3.1. Einbindung vordefinierter Skripte 193
8.3.2. Konfiguration der Schnittstelle für Nutzeraktionen 193
8.3.3. Konfiguration des Standardverhaltens einer VIDEA-Instanz 197
8.3.4. Layoutalgorithmen in VIDEA 200
8.4. Bedienung und dynamische Konfiguration 207
8.4.1. Interaktive Nutzung der VIDEA-Manager 208

9. Evaluation 213
9.1. Begriffsklärung und Ablauf 213
9.2. Nicht-inhaltliche Evaluation 216
9.2.1. Der Motivationsfragebogen 216
9.2.2. Ergebnisse der AVL-Übung 217
9.2.3. Ergebnisse der Dijkstra-Übung 219
9.3. Inhaltliche Evaluation der Übung „AVL-Bäume“ 221
9.3.1. Die Übung der VIDEA-Gruppe 221
9.3.2. Die Übung der Kontrollgruppe 223
9.3.3. Der Evaluationsfragebogen 223
9.3.4. Die Auswertung des inhaltlichen Fragebogens 223
9.4. Inhaltliche Evaluation der Übung „Dijkstras kürzeste Wegesuche“ 226
9.4.1. Die Übung der VIDEA-Gruppe 230
9.4.2. Die Übung der Kontrollgruppe 231
9.4.3. Der Evaluationsfragebogen 234
9.4.4. Die Auswertung des inhaltlichen Fragebogens 234
9.5. Zusammenfassung 234

10. Abschließende Bemerkungen 237
10.1. Zusammenfassung 237
10.2. Bewertung von VIDEA 243
10.2.1. Bewertung von VIDEA anhand des eigenen Bewertungsschemas 243
10.2.2. Bewertung von VIDEA anhand von *Best Practices* 248
10.3. Ausblick 253

Abbildungsverzeichnis

1. Erstellungsprozess einer neuen Instanz der Visualisierungsumgebung 2
2. Spezifikation der Linksrotation auf einem AVL-Baum als Graphtransformation 3
3. Eine mit VIDEA generierte Lerneinheit vor und nach Ausführung der Linksrotation auf dem markierten Knoten eines unbalancierten AVL-Baums 5
4. Eine Übersicht über diese Arbeit 6
5. Operation auf einer Datenstruktur; es ist schwer auf Anhieb zu sehen, was sich geändert hat, wenn die Bilder nacheinander dargestellt werden 12
6. Erstellungsprozess einer neuen Instanz der Visualisierungsumgebung 27
7. Übliches Layout zweier Datenstrukturen. Oben: Baum. Unten: Liste. 31
8. Beispiel einer algebraischen Spezifikation in CASL 46
9. Oben: Beispiel einer logikbasierte Spezifikation in Z. Unten: Eine Liste als UML-Klassendiagramm. 47
10. Das Fährmannproblem in PROGRES 48
11. Ein Klassendiagramm in Fujaba 49
12. Eine Ersetzungsregel in DiaGen 50
13. Ein animierter Beweis des Satzes von Pythagoras in Zeus; die Texte sind nicht lesbar und für die Illustration nicht entscheidend 51
14. Die Abbildung zeigt die Animation von verschiedenen Sortieralgorithmen in JCAT in einer Ansicht 53
15. Beispielapplet der SALABIM-Bibliothek: Blöcke stapeln 54
16. Beispielsitzung mit DDD 55
17. Beispielsitzung in Jeliot 56
18. Zwei yFiles-Layoutalgorithmen beispielhaft gezeigt. Links: CircularLayout. Rechts: TreeLayout. 58
19. Türme von Hanoi in Leonardo 59
20. Ein interaktives Szenario in einem mit Stagecast entwickelten Computerspiel . 61
21. Vergleichstabelle der Spezifikationsansätze; mit 'UML' sind hauptsächlich UML-Klassendiagramme gemeint (siehe Text). 63
22. Vergleichstabelle der Werkzeuge 64
23. Links: Ein AVL-Baum in einer üblichen Darstellung. Mitte: Maximal entarteter AVL-Baum. Rechts: Maximal entarteter Binärbaum. 76
24. Links: Unbalancierter Baum. Rechts: Nach Linksrotation um die Wurzel gültiger AVL-Baum 77
25. Die Linksrotation auf dem AVL-Baum in einer schematischen Darstellung . . . 78
26. Eine einfache, verkettete Liste mit Listenkopf 78
27. Eine doppelt verkettete Ringliste mit Listenkopf 79
28. Beispiel-Graph für die Anwendung des Dijkstra-Algorithmus zur kürzesten Wegesuche 81
29. Der Kruskal-Algorithmus in prozeduraler Notation, entnommen aus [Ein05a] . 82
30. Vier Stadien des Kruskal-Algorithmus an einem Beispiel-Graphen gezeigt; die Richtung der Kanten spielt für Kruskal keine Rolle. 83
31. Die Generierungs-Schritte für eine neue VIDEA-Instanz 84

32. Konstruktor eines einfachen Objekts 90
33. Konstruktor eines kompletten Baums 91
34. Erzeugen einer *left*-Beziehung zwischen zwei Objekten 92
35. Das Entfernen der kleinsten noch nicht betrachteten Kante aus der Warteschlange in Kruskal 94
36. Oben: AVL-Knotentyp als UML-Klassendiagramm. Unten: Die Linksrotation im AVL-Baum, dargestellt als Schema von Paaren von Objektdiagrammen . . . 95
37. AVL-Knotentyp in PROGRES 96
38. Die Linksrotation im AVL-Baum, dargestellt als PROGRES-Produktion 97
39. Klassische, lehrbuchartige Darstellung der Linksrotation in einem AVL-Baum . 98
40. Die Hauptschleife des Dijkstra-Algorithmus als Pseudocode 101
41. Haupt-„Schleife“ des Dijkstra-Algorithmus 102
42. AVL-Instanz. Oben: Leere Instanz. Unten: Verständnistest für die Berechnung des Balancefaktors 103
43. Nutzung von PROGRES, UPGRADE2, yFiles in VIDEA 105
44. 2-3-Baum. Oben: Gängige Darstellung. Unten: Standard-Darstellung in VIDEA. 108
45. Umhängen oder Löschen und Neu-Erzeugen einer Kante? 109
46. Erzeugen eines Knotens in PROGRES 110
47. Oben: Ändern eines Knotenattributs. Unten: Erzeugen einer einfachen Kante. . 111
48. Links: Die zu den Spezifikationen in den Abb. 46, 47 und 49 gehörigen Typdeklarationen in PROGRES. Rechts: Ein Beispiel-Graph mit den Objekten der Spezifikation auf der linken Seite. 112
49. Erzeugen einer attributierten Kante in PROGRES in Form eines Knotens vom Knotentyp *ComplexEdge* 113
50. Beispiel für eine einfache Kontrollstruktur: *handleNode* 115
51. Spezifikationen für den AVL-Baum. Oben: Nochmal die Spezifikation des *AVL-Node*-Knoten-Typs selbst. Unten: Hilfsdeklarationen 116
52. Ein unbalancierter AVL-Baum mit allen in Abb. 51 oben definierten Attributen 117
53. Links-Rechts-Doppelrotation des AVL-Baums 118
54. Finden der Wurzel eines AVL-Baums 119
55. Einfügen in einen AVL-Baum 121
56. Test des Verständnisses bei der Angabe des Balancefaktors eines Knotens . . . 122
57. Spezifikation der Knoten- und Kantentypen für die doppelt verkettete Liste . . 123
58. Oben: Die Erzeugung des Listenkopfes für eine Liste. Unten: Setzen einer *next*-Kante. 124
59. Spezifikation von *InMenu_Exercise_addFirst* für die doppelt verkettete Liste . 125
60. Zwei Spezifikationen zum Löschen aller Listenelemente in einer Liste 126
61. Spezifikation von *prepareInsert* für die doppelt verkettete Liste 127
62. Spezifikation von *InMenu_Exercise_insertStep* für die doppelt verkettete Liste . 128
63. Spezifikation von *findPlaceForInsertion* für die doppelt verkettete Liste 129
64. Spezifikation von *setPreviousFromNewElem* für die doppelt verkettete Liste . . 130
65. Beispielablauf beim Einfügen in eine Liste 131
66. Beispielablauf beim Einfügen in eine Liste 132

67. Der Dijkstra-Algorithmus in prozeduraler Form mit den gängigen Farben weiß, grau und schwarz, entnommen aus [Ein05a] 134
68. Typ-Deklarationen für den Dijkstra-Algorithmus auf Graphen 135
69. Erster Schritt des Dijkstra-Algorithmus (Finden und Initialisieren des Wurzel-Knotens) 136
70. Oben: Testen des Verständnisses bzgl. des nächsten Schrittes des Dijkstra-Algorithmus. Unten: Haupt-„Schleife“ des Dijkstra-Algorithmus 137
71. Finden der aktuellen V-Knoten gemäß Dijkstra 138
72. Umfärben aller V-Knoten gemäß Dijkstra 139
73. Neu-Berechnen der geschätzten Entfernung aller V-Knoten bei Dijkstra 140
74. Typ-Deklarationen für den Kruskal-Algorithmus 141
75. Oben: Eine Produktion zum Erzeugen eines Beispiel-Graphen. Unten: Der von der Produktion kruskal_predefinedGraph1 erzeugte Beispiel-Graph. 142
76. Zwei Verständnistests: Oben: welche Kante wird von Kruskal als Nächstes betrachtet? Unten: Wird die nächste betrachtete Kante zum Spannbaum hinzugenommen? 144
77. Oben: Eine Nutzeraktion für das Ausführen genau eines Schrittes des Kruskal-Algorithmus. Unten: Das Entfernen der kleinsten noch nicht betrachteten Kante aus der Warteschlange. 145
78. Hilfsproduktion zum Verschmelzen zweier Spannbäume mittels einer neu hinzugenommenen Kante 146
79. AVL-Instanz: Ein vorher definierter Ablauf, hier Einfügen des Knotens '120' . 151
80. Skript für den flüssigen Aufbau eines AVL-Baums 152
81. DLL-Instanz: Eine Reparatur-bedürftige Liste wurde aufgebaut 153
82. Beispielsitzung mit Simulation des Dijkstra-Algorithmus 154
83. AVL-Instanz: Verletzte Sortierungseigenschaft zweier Knoten 157
84. AVL-Instanz: Verständnistest bei der Berechnung des Balance-Faktors 158
85. Oben: Nach Wahl der falschen Rotationsoperation. Unten: Der Benutzer hat die korrekte Rotationsoperation gefunden 159
86. Dijkstra-Instanz: Test, um das Verständnis des Algorithmus zu überprüfen . . . 160
87. Nutzung eines booleschen Tests in der Kruskal-Instanz. Oben: *isNextConsideredEdgeTaken* wird aufgerufen. Unten: Der Test wurde durch Anklicken der 'checkbox' korrekt mit „Ja“ beantwortet. 162
88. Der Nutzer fügt ein neues Element '5' in eine doppelt verkettete Liste ein. Oben: Zwei Knoten wurden markiert, um eine neue 'previous'-Kante von Element '6' nach Element '5' zu ziehen und gleichzeitig die alte 'previous'-Kante zu entfernen. Unten: Die Kante wurde erzeugt, die Knoten '5' und '6' sind noch markiert. 163
89. Ausschnitt eines möglichen, eigenen Java-Programms der Lernenden, das auf der Klasse `Node` basiert 165
90. Ausschnitt aus der Klasse `Node`. 166
91. Oben: Aus einer Debugging-Sitzung in Together wurde bereits ein unbalancierter AVL-Baum in VIDEA über die CORBA-Schnittstelle aufgebaut. Unten: Der Zustand nach der Hälfte von rotateLeft; man sieht, dass vorübergehend die Baum-Eigenschaft verloren geht 168

92. Oben: Eine falsche Lösung, die durch Umhängen den Baum komplett zerstört. Unten: Eine korrekte Lösung: Der Baum wurde linksrotiert und ist deshalb wieder ein gültiger AVL-Baum. 169
93. Klassendiagramm des AVL-Baums in Together 170
94. Schritte bei der Erstellung und Nutzung einer VIDEA-Instanz 173
95. Einbettung des VIDEA-Rahmenwerks in das Werkzeug-Umfeld, Wiederholung von Abb. 43 175
96. Detaillierter Ablauf beim Design einer VIDEA-Instanz; die Kanten bedeuten den sich ergebenden Informationsfluss, beschreiben also auch die zeitliche Abfolge der Beschäftigung des Dozenten mit den einzelnen Punkten. 176
97. Strukturell nicht erlaubt: Links: Ins Leere gehende Kanten. Rechts: Drei Nachfolger im Binärbaum. 178
98. Drei Lernschritte der im Text beschriebenen, einfachen VIDEA-Listen-Instanz 182
99. Wiederholung von *recolorVs* 185
100. Oben: Wiederholung von *checkBalance*. Unten: Wiederholung von *textNextU*. . 186
101. Aufrufhierarchie-Ebenen 187
102. VIDEAs CORBA-Architektur 189
103. Die Generierung einer VIDEA-Instanz, genauer als in Abb. 31 190
104. Die PROGRES-Entwicklungsumgebung, hier aus dem VIDEA-Generierungsskript gestartet 191
105. Die Verzeichnisstruktur eines Nutzers, der VIDEA-Instanzen generiert 192
106. Eine beispielhafte Ausprägung der Konfigurationsdatei `prototype.xml` für die VIDEA AVL-Instanz 195
107. Die Menüleiste unserer Dozenten AVL-Instanz 196
108. Ein Ausschnitt aus `musoft_config.xml` für die Dijkstra-Instanz 198
109. Unbalancierter AVL-Baum, links mit sichtbaren Pseudo-Knoten, rechts ohne . 202
110. Ein Ausschnitt aus doDllLayout() 203
111. Beispiel einer Liste, die trotz Unterbrechungen eindeutig zu layouten ist 205
112. Wiederholung von Abb. 28 aus Kapitel 5 207
113. Die Oberfläche zweier Manager. Links: Der Node Coloring Manager der AVL-Instanz. Rechts: Der Edge Coloring Manager der Dijkstra-Instanz. 209
114. Der Node Attribute Display Manager 210
115. Der verwendete Motivationsfragebogen 215
116. Ausschnitte aus dem Übungsblatt für die AVL-Baum-Übung 222
117. AVL-Übung der Kontrollgruppe 224
118. Inhaltlicher AVL-Evaluationsbogen 225
119. Inhaltlicher AVL-Evaluationsbogen, Fortsetzung 226
120. Dijkstra VIDEA-Uebung 227
121. Dijkstra VIDEA-Uebung, Fortsetzung 228
122. Dijkstra-Übung der Kontrollgruppe 229
123. Dijkstra-Übung der Kontrollgruppe, Fortsetzung 231
124. Inhaltlicher Dijkstra-Evaluationsbogen 232

Tabellenverzeichnis

1. Ergebnisse des Motivationsfragebogens für die AVL-Übung 217
2. Ergebnisse des Motivationsfragebogens für die Dijkstra-Übung 219
3. Ergebnisse der Auswertung der Kontroll-Fragebögen zum inhaltlichen Verständnis beider Gruppen bezüglich AVL-Bäumen. Die mit ’>’ gekennzeichneten Zahlen bedeuten, dass es externe Faktoren gab, durch die die Zahlen möglicherweise verringert wurden (siehe Beschreibung im Text). 230
4. Ergebnisse der Auswertung der Kontroll-Fragebögen zum inhaltlichen Verständnis beider Gruppen bezüglich Dijkstra-Algorithmus. Die mit ’>’ gekennzeichneten Zahlen bedeuten, dass es externe Faktore n gab, durch die die Zahlen möglicherweise verringert wurden (siehe Beschreibung im Text). 233
5. Spezifikations-Anforderungen 243
6. Anforderungen an die logische Verarbeitung 244
7. Anforderungen an die Oberfläche 245
8. Anforderungen an die Konfigurierbarkeit 246
9. Sonstige Anforderungen 246

Abkürzungsverzeichnis

Abb.	Abbildung
bzw.	beziehungsweise
ca.	circa
etc.	et cetera
evtl.	eventuell
ggf.	gegebenenfalls
MuSofT	**Mu**ltimedia in der **Sof**tware**T**echnik
o. Ä.	oder Ähnliches
PROGRES	**PRO**grammierte **GR**aph-**E**rsetzungs-**S**ysteme
s. u.	siehe unten
u. a.	unter anderem
usw.	und so weiter
v. a.	vor allem
vgl.	vergleiche
VIDEA	**V**isual **I**nteractive **D**ata Structure **E**nvironment for **A**nimations
z. B.	zum Beispiel

1. Einführung

In dieser Arbeit wird ein Rahmenwerk zur automatisierten Erstellung einer definierten Klasse interaktiver Animationswerkzeuge zur Lehre von Algorithmen und Datenstrukturen vorgestellt.

Gerade im Bereich der Datenstruktur- und Algorithmenanimation gibt es mittlerweile eine fast unüberschaubare Anzahl fertiger Animationen (z. B. [CCA03, Ani03, Sor03, Bub03]) mit teils sehr ansprechender Präsentation und manchmal auch sehr stimmiger und klarer Darstellung der wesentlichen Lerninhalte. Außerdem existieren bereits seit einiger Zeit Werkzeuge zum Animationsbau (wie [RSF00, SK93] etc.), so dass man sich fragen mag, warum schon wieder ein Rahmenwerk für diesen Zweck vorgestellt werden soll.

Eine in diesem Zusammenhang seltsam erscheinende Tatsache ist, dass die Mehrzahl von Dozenten von Algorithmen und Datenstrukturen in Umfragen angeben, dass sie einerseits der Nutzung von Animationen in der Lehre einen großen Lerneffekt zuschreiben, aber andererseits selten bis nie Animationen in der Lehre einsetzen (siehe z. B. [NRA+02]).

Der Grund wird deutlicher, wenn man die Situation genauer betrachtet. Dann zeigen gerade viele der schönsten und überzeugendsten Animationen eine entscheidende Schwäche: Sie sind nämlich fest auf eine Datenstruktur oder einen Algorithmus festgelegt und oft zusätzlich weder skalierbar noch erweiterbar oder änderbar. Hinter anderen steckt ein Konzept, das eine eingeschränkte Erweiterbarkeit ermöglicht, aber im Gegenzug Einschränkungen bei der Klarheit der Visualisierung oder bei der Interaktivität macht, so dass man sich im letzteren Fall passiv einen Film ansieht. Dabei haben gerade Untersuchungen der Wirkungen von Algorithmenanimationen auf den Lerneffekt immer wieder die Interaktivität und die aktive Einbindung der Lernenden als besonders wichtig für den Lernerfolg herausgestellt (siehe etwa die Metastudie [Naj95]).

Aus diesem Grund wurden in den letzten Jahren auch Werkzeuge und Rahmenwerke entwickelt, die Wert legen auf die Umsetzung bisher als wesentlich erkannter Eigenschaften multimedialer Lernumgebungen (z. B. [Röß02, Fal02, HNH02]). Diese sind dann aber oft so allgemein, dass Datenstrukturen nicht auf natürliche Art modellierbar sind, haben konzeptionelle Einschränkungen, z. B. eine feste maximale Anzahl ausgehender Kanten aus jedem Knoten, oder vernachlässigen Aspekte, die sich in bisherigen Studien und auch in der Lernpsychologie als wesentlich herausgestellt haben, wie Interaktivität oder flüssige Animationen. Oft wird der Fokus hauptsächlich auf Algorithmenanimation gelegt, wodurch die Lernchancen des freien Erforschens der Operationen auf einer Datenstruktur nicht ausreichend genutzt werden.

Besonders die interaktiven Fähigkeiten multimedialer Lernsoftware werden immer wieder im Zusammenhang mit einem nachvollziehbar höheren Lerneffekt genannt. Hierbei lassen sich verschiedene Grade von Interaktivität unterscheiden: Eine einfache Interaktivität liegt bereits bei Systemen vor, die es dem Benutzer erlauben, vordefinierte Abläufe vorübergehend anzuhalten. Mehr Interaktivität kommt ins Spiel, wenn Instanzen einer Datenstruktur manuell aufgebaut werden können, auf denen dann z. B. ein vordefinierter Algorithmus abläuft. Ein noch größeres Maß an Interaktivität bieten Systeme, die die Beantwortung vorbereiteter oder zufälliger Fragen während der Nutzung der jeweiligen Lerneinheit verlangen und intelligent auf die Antwort reagieren. Einen noch höheren Grad an Interaktion mit dem Lernenden bieten Systeme, die es erlauben, auf Instanzen einer zu lernenden Datenstruktur mit einem Satz zur Verfügung gestellter Operationen in beliebiger Reihenfolge zu arbeiten, wobei des System dem Lernenden geeignete Rückmeldungen gibt, wenn er eine wichtige Eigenschaft der jeweiligen Datenstruktur

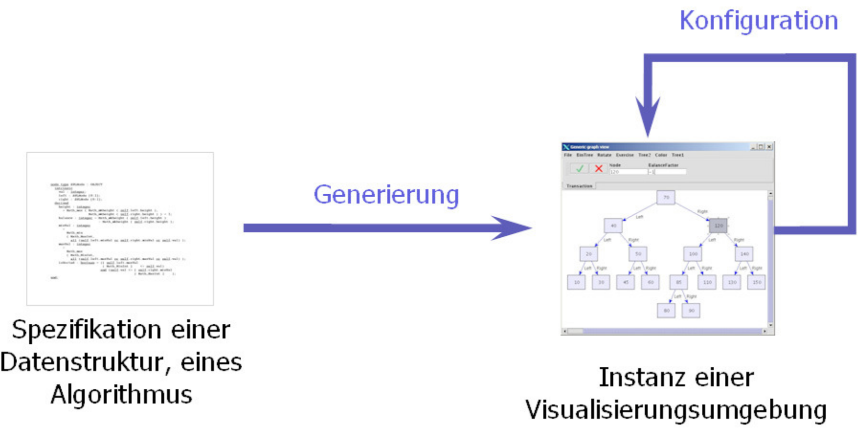


Abbildung 1: Erstellungsprozess einer neuen Instanz der Visualisierungsumgebung

verletzt oder die eigentlich vorgesehenen Schritte eines zu lernenden Algorithmus nicht korrekt nachvollzieht. Am meisten Aktivität vom Lernenden wird gefordert, wenn er seine eigenen Programme vom System visualisieren lässt oder selbst Animationen erstellt.

Diese willkürliche Auswahl interaktiver Möglichkeiten eines multimedialen Lernsystems soll zeigen, dass Interaktivität ein komplexes Thema ist. Wie wichtig das Thema für den Lerneffekt ist, haben wir bereits betont. In [NRA+02] gehen die Autoren so weit, zu sagen: 'We argue that such technology, no matter how well it is designed, is of little educational value unless it engages learners in an active learning activity'.

Deshalb ist es nicht verwunderlich, dass einige der existierenden Ansätze versuchen, die an die Oberfläche eines multimedialen Lehrwerkzeugs zu stellenden Anforderungen mit einer möglichst hohen Interaktivität zu koppeln – leider oft zu dem Preis, einen erheblichen Aufwand für die Erzeugung neuer Lerneinheiten für andere oder abgewandelte Datenstrukturen und Algorithmen betreiben zu müssen.

Wir propagieren deshalb ein Grundszenario, das sowohl die einfache und schnelle Erzeugung von Lerneinheiten für die Lehre von Algorithmen und Datenstrukturen gewährleistet als auch die wesentlichen didaktischen Eigenschaften multimedialer Lehrwerkzeuge ermöglicht. Die schnelle Erzeugung und einfache Anpassung soll dabei durch die Generierung einer Lerneinheit aus einer Spezifikation der zu lehrenden Datenstruktur und des zu lehrenden Algorithmus erreicht werden. Abb. 1 zeigt die zugrundeliegende Idee: Aus der Spezifikation wird eine ablauffähige Lerneinheit generiert, die durch Konfiguration angepasst werden kann. Dabei erfolgt eine prinzipielle Trennung des Lerninhaltes, der spezifiziert wird, von den Details der Animation und Visualisierung, die konfiguriert werden. Auf diese Weise kann gewährleistet werden, dass bei einer kleinen Änderung oder Erweiterung der zu unterrichtenden Datenstruktur eine kleine Änderung der Spezifikation ausreicht, um eine angepasste Lerneinheit zu erzeugen.

Bei genauerer Analyse ergeben sich für eine Visualisierungsumgebung, die die wichtigsten im Hinblick auf einen maximalen Lerneffekt erforderlichen Eigenschaften aufweist und in dem

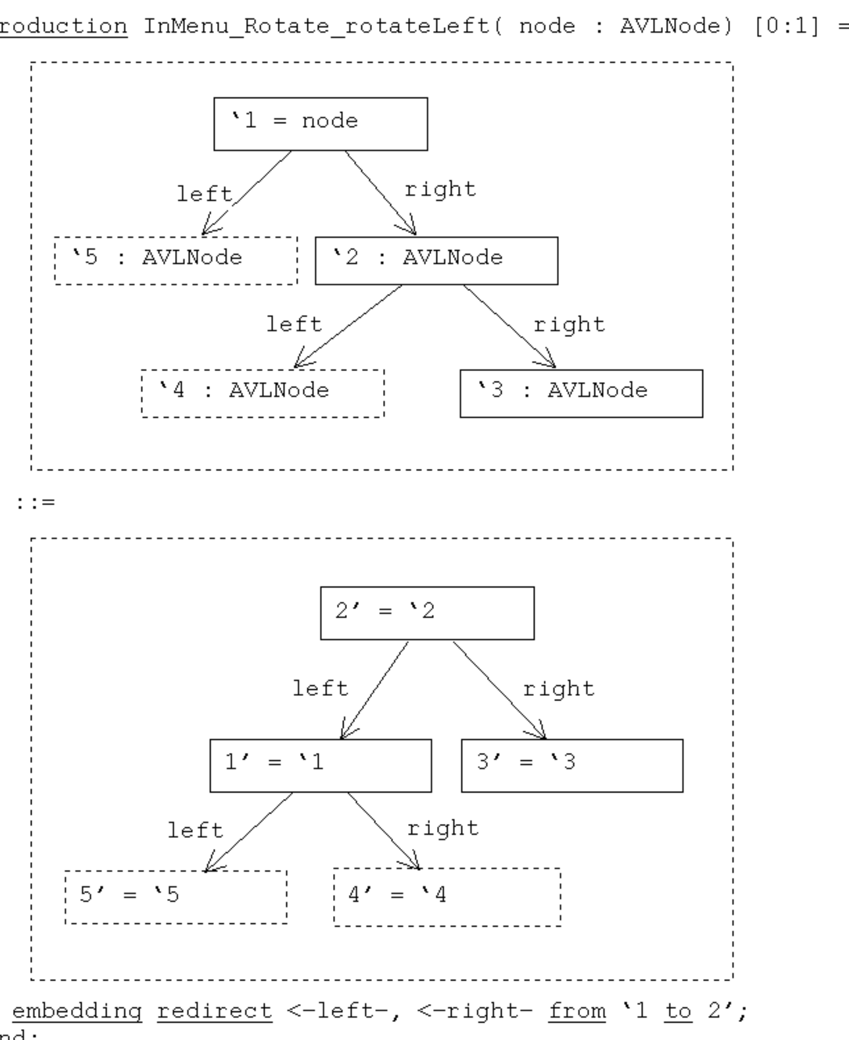


Abbildung 2: Spezifikation der Linksrotation auf einem AVL-Baum als Graphtransformation

propagierten Grundszenario einsetzbar ist, eine Reihe weiterer technischer Anforderungen. Wir haben bestehende Paradigmen und Werkzeuge, die für die Spezifikation und Visualisierung von Datenstrukturen und Algorithmen geeignet sind, auf ihre Fähigkeit hin überprüft, diese Anforderungen zu erfüllen. Dabei hat es sich gezeigt, dass kein Ansatz unmittelbar dazu in der Lage war und nur wenige Ansätze die wichtigsten Punkte erfüllen konnten.

Deshalb stellen wir im Rahmen dieser Arbeit das neue Konzept VIDEA (**V**isual **I**nteractive **D**ata Structure **E**nvironment for **A**nimations) vor, das sich einer Kombination derjenigen untersuchten Ansätze bedient, die am vielversprechendsten sind. Fehlende Eigenschaften werden hinzugefügt.

Algorithmen und Datenstrukturen werden in VIDEA visuell spezifiziert. Dafür werden UML-Klassendiagramme, UML-Objektdiagramme und Graphtransformationen eingesetzt (für UML siehe [UML03], für Graphtransformationen [MM98]). Wie im Grundszenario gefordert, wird aus der Spezifikation in einem automatisierten Verfahren Code generiert. Dieser Code wird zusammen mit generischem Code, der für VIDEA entwickelt wurde, zur Laufzeit der Lernumgebung für die Visualisierung der Datenstruktur und die Interaktion mit dem Lernenden genutzt. Die technische Umsetzung basiert auf der Zusammenarbeit einer Graphtransformationsmaschine mit einer mächtigen Java-basierten Bibliothek für die Visualisierung und Animation von Graphen.

In VIDEA wird bewusst darauf verzichtet, mögliche Abläufe des Lernprozesses vorab im Werkzeug zu spezifizieren. Stattdessen ergeben sich so genannte Fahrpläne, die eine Menge vorgesehener konkreter Nutzungsszenarien darstellen, nach einer vorgeschlagenen Vorgehensweise aus dem vom Dozenten jeweils vorgesehenen didaktischen Konzept. Diese werkzeugunabhängigen Fahrpläne wirken sich auf die Erstellung der Spezifikation aus, da neben der eigentlichen Datenstruktur und ihren Schnittstellenoperationen zusätzliche Operationen zur Unterstützung der vorgesehenen Abläufe gebraucht werden und somit spezifiziert werden müssen. Ein Teil eines solchen Fahrplans kann sich in bestimmten Nutzungsszenarien auch in der Aufgabenstellung einer Übungsaufgabe, die mit VIDEA unterstützt wird, widerspiegeln.

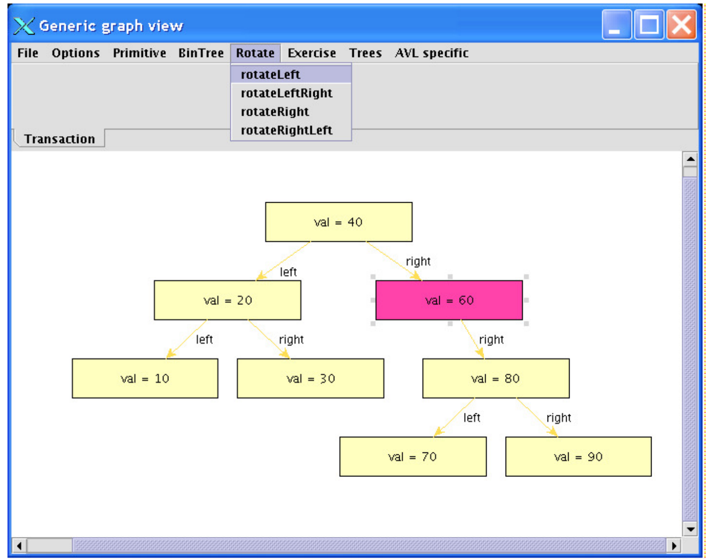
Auf der Ebene der in VIDEA verwendeten Werkzeuge wird eine *Datenstruktur* als Graph dargestellt. *Operationen* auf Datenstrukturen werden als Graphtransformationen modelliert. *Invarianten* der Datenstruktur können ebenso in der verwendeten Graphtransformationssprache spezifiziert werden. Für die Umsetzung des didaktischen Konzepts (zum Beispiel Zwischenfragen des Systems an die Lernenden) werden weitere Operationen spezifiziert. *Algorithmen* schließlich werden durch eine geeignete Kombination der spezifizierten Operationen implementiert, wobei es u. a. möglich ist, diese imperativ in der Programmiersprache des verwendeten Graphersetzungssystems anzugeben oder durch die Nutzung externer Skripte oder durch eine Kombination aus beiden Möglichkeiten.

Um praktische Beispiele angeben zu können, nutzen wir exemplarisch vier bekannte und wichtige Algorithmen und Datenstrukturen, die in vielen Datenstruktur-Lehrbüchern beschrieben werden (wir benutzen die Definitionen aus [Ein05a]):

- AVL-Bäume,
- die doppelt verkettete (Ring-)Liste,
- Dijkstras Algorithmus zur kürzesten Wegesuche auf Graphen und
- Kruskals Algorithmus zur Berechnung des minimalen Spannbaums eines Graphen.

Abb. 2 zeigt die Spezifikation einer Linksrotation auf einem AVL-Baum als Graphtransformation. Die Ähnlichkeit zu herkömmlichen Darstellungen in Lehrbüchern durch Trennung der Zustände vor und nach der Operation ist beabsichtigt, da auf diese Art die Spezifikation für den Dozenten einfacher und natürlicher wird als bei einer rein textuellen Schreibweise. Zusätzlich können solche Spezifikationen als Dokumentation der Lerneinheit dienen und z. B. dem Lernenden bei der Bearbeitung einer Aufgabe zur Verfügung gestellt werden.

Abb. 3 zeigt zwei Bildschirmausschnitte einer generierten Lerneinheit für AVL-Bäume. In dem Zustand, der oben dargestellt wird, ist ein teilweise unbalancierter AVL-Baum zu sehen.



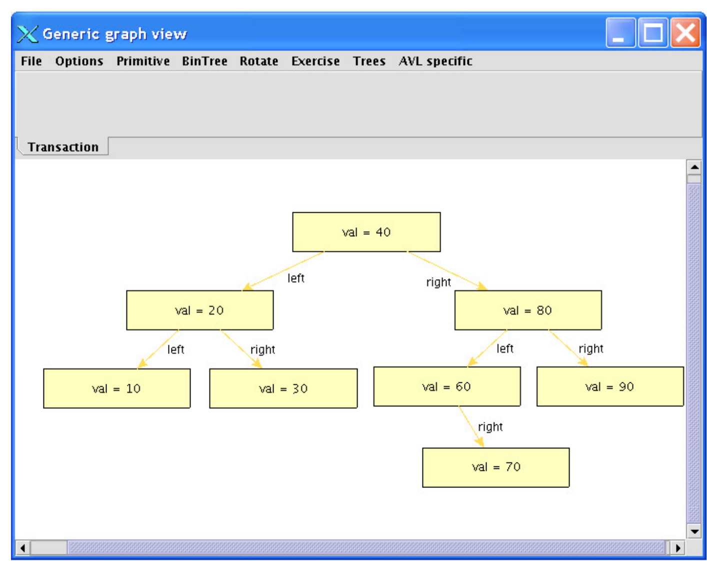


Abbildung 3: Eine mit VIDEA generierte Lerneinheit vor und nach Ausführung der Linksrotation auf dem markierten Knoten eines unbalancierten AVL-Baums

Die Wurzel des unbalancierten Teilbaums ist farblich mit der Fehlerfarbe rot gekennzeichnet und wurde zusätzlich mit der Maus markiert, um als Parameter der Operation *rotateLeft* zu dienen, deren Spezifikation in Abb. 2 gezeigt wurde. In Abb. 3 unten ist der Zustand nach Ausführung von *rotateLeft* zu sehen: Der Baum ist wieder ein gültiger AVL-Baum und wird somit wieder in der vorkonfigurierten Standardfarbe gezeichnet. Der Übergang zwischen den beiden Zuständen oben und unten geschieht in der realen Animation auf eine kontinuierliche Art und Weise, nämlich mithilfe eines Morphing-Vorgangs.

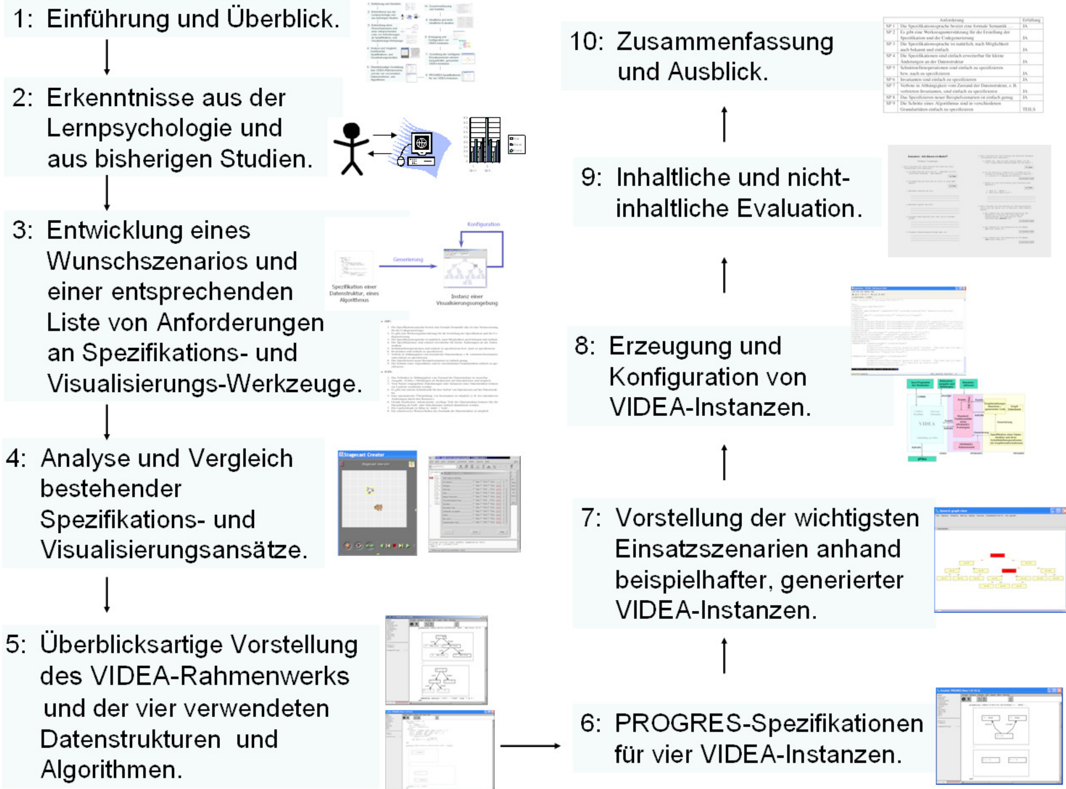


Abbildung 4: Eine Übersicht über diese Arbeit

Die in VIDEA generierten Lerneinheiten, die im Rahmen dieser Arbeit auch *VIDEA-Instanzen* heißen, können in verschiedenen Szenarien eingesetzt werden, von denen vier besonders wichtige im Laufe dieser Arbeit genauer beschrieben und mithilfe beispielhafter VIDEA-Instanzen erläutert werden:

- Vorführen im Rahmen einer Präsenzveranstaltung.
- Passives Betrachten vorgegebener Abläufe.
- Interaktives Erforschen und Aufgaben-Lösen.
- Ferngesteuerte Animation eigener Programme der Lernenden.

Obwohl VIDEA in einer Reihe weiterer Szenarien genutzt werden kann, die ebenfalls in dieser Arbeit vorgestellt werden, ermöglichen bereits diese vier die Umsetzung der wesentlichen Elemente multimedialer Lernsoftware. Gleichzeitig werden die üblichen Lehrformen Präsenzveranstaltung, Übungen und Selbststudium unterstützt, woran zu sehen ist, dass durch den hier vorgestellten Ansatz die herkömmliche Lehre nicht ersetzt, sondern ergänzt werden soll. Das VIDEA-Konzept bewährte sich im bisherigen praktischen Einsatz sowohl bezüglich der geforderten einfachen Anpassbarkeit von Lerneinheiten für abgewandelte Datenstrukturen als auch im Einsatz mit Studierenden. So konnte in empirischen Untersuchungen demonstriert werden, dass

eine Übung mit VIDEA einen größeren Lernfortschritt erbrachte als eine übliche Papier-und-Tafel-Übung. Außerdem zeigte sich in einer beobachtenden Evaluation und als Resultat einer Befragung der Studierenden ein hoher Motivationsgewinn, der sich nach bisherigen Erkenntnissen stets unterstützend auf einen langfristigen Lernerfolg auswirkt.

Abb. 4 zeigt die Gliederung dieser Arbeit. Als Grundlage für VIDEA werden zunächst in Kapitel 2 Erkenntnisse aus der Lernpsychologie und Ergebnisse abgeschlossener Studien über die Wirksamkeit multimedialer Lehransätze betrachtet. Aus diesen Erkenntnissen wird in Kapitel 3 ein aus unserer Sicht optimales Grundszenario für eine Visualisierungsumgebung für die multimediale Unterstützung der Lehre von Algorithmen und Datenstrukturen entwickelt. Dieses Grundszenario wird verfeinert in eine Reihe konkreter Anforderungen, die in Kategorien gruppiert und priorisiert werden und in ihrer Gesamtheit als neues Bewertungsschema für Visualisierungsansätze für die Lehre von Algorithmen und Datenstrukturen dienen können.

In Kapitel 4 werden bestehende Ansätze der Spezifikation und Visualisierung auf ihre Brauchbarkeit für die Implementierung eines möglichst umfassenden Konzeptes, das die wesentlichen Punkte des in Kapitel 3 entwickelten Bewertungsschemas umsetzt, untersucht. Da keiner der evaluierten Ansätze für sich auch nur die meisten Anforderungen erfüllen kann, legen wir bei der Evaluation dieser bestehenden Ansätze eine kondensierte Teilliste der am höchsten priorisierten Punkte des Bewertungsschemas zugrunde. Die aussichtsreichsten Kandidaten, die aus dieser Untersuchung hervorgehen, werden in dem neuen Ansatz verwendet. Kapitel 5 führt die vier in dieser Arbeit hauptsächlich zur Illustration verwendetenden Datenstrukturen und Algorithmen ein und stellt den neuen Ansatz überblicksartig vor; dabei werden auch die wichtigsten Teilkonzepte von VIDEA beschrieben und in das Gesamtkonzept eingeordnet.

In Kapitel 6 wird zunächst das graphbasierte Konzept der Spezifikation von Datenstrukturen eingeführt. Dann werden vier beispielhafte Instanzen für die vier in Kapitel 5 eingeführten Datenstrukturen visuell, nämlich zuerst in UML, dann mit Graphtransformationen, modelliert, wie das auch ein Dozent, der den hier vorgestellten Ansatz verwendet, für eine neue Datenstruktur tun würde. Der beispielhafte Einsatz von VIDEA in den vier wesentlichen Szenarien wird zusammen mit zugehörigen Bildschirmausschnitten in Kapitel 7 in aufsteigender Komplexität bezüglich der Möglichkeiten der Einflussnahme vorgestellt. Die Bedienung von VIDEA aus der Sicht des Dozenten wird in Kapitel 8 erläutert. Dazu gehören Themen wie das Design, die Generierung und die Konfiguration einer VIDEA-Instanz. Eine durchgeführte Evaluation, die Effekte des Einsatzes von VIDEA auf Lernerfolg und Motivation der Studierenden untersucht, wird in Kapitel 9 beschrieben.

Kapitel 10 schließlich fasst diese Arbeit zusammen, prüft den hier vorgestellten Ansatz auf die Umsetzung der geforderten Eigenschaften und gibt einen Ausblick auf mögliche Weiterentwicklungen.

„Alles Lernen ist keinen Heller wert, wenn Mut und Freude dabei verloren gehen.“

Johann Heinrich Pestalozzi

2. Didaktischer Ansatz

In diesem Kapitel werden Erkenntnisse aus der Lernpsychologie und aus bisherigen Studien über die Wirksamkeit von Visualisierungen, insbesondere von der Visualisierung von Algorithmen und Datenstrukturen, zusammengefasst.

Dabei wird es sich zeigen, dass es trotz der Schwierigkeit in diesem Bereich, allgemeingültige Aussagen zu machen, einige Erfolgsfaktoren gibt, die immer wieder genannt werden (wie z. B. verschiedene Formen der Interaktivität und flüssige, kontinuierliche Animationen im Gegensatz zu 'Sprüngen' im Animationsablauf). Dabei werden wir aus der Menge des verfügbaren Materials einen möglichst repräsentativen Querschnitt herausgreifen, indem wir nicht nur Ergebnisse der Evaluation der Wirksamkeit bestimmter Animations-Systeme präsentieren, sondern auch einen Blick auf Untersuchungen zu verwandten Fragestellungen werfen. Beispiele sind die Untersuchung der Wirksamkeit einer E-Learning-Umgebung für Kinder, die Art, wie Lernende selbst Animationen entwerfen würden, oder Beobachtungen, wie Lernende die angebotenen Animations-Systeme nutzen.

Schließlich werden wir versuchen, Gründe zu erklären, warum die Wirksamkeit von Visualisierungen zwar – mit durchaus positiven Ergebnissen – intensiv erforscht wird, aber entsprechende Systeme noch sehr wenig in der Lehre eingesetzt werden. Der Hauptgrund dafür ist, dass Systeme, die wirklich die wichtigsten als lernwirksam erkannten Eigenschaften in sich vereinen, im Moment fast nur manuell für jede Ausprägung einer Datenstruktur oder eines Algorithmus zu entwerfen sind.

Deswegen werden wir uns anschließend in Kapitel 3 genauer überlegen, welche konkreten technischen Anforderungen für die Umsetzung der geforderten Eigenschaften notwendig sind.

2.1. Erkenntnisse aus der Lernpsychologie

Lernen geschieht über die Sinneskanäle. Multimedia stellt Informationen in mehreren Repräsentationen gleichzeitig zur Verfügung. Eine einfache Vermutung ist also, dass der Lerneffekt sich durch multimediale Lehrmaterialien automatisch verbessert, weil mehrere Sinneskanäle angesprochen werden. Das ist auch untersucht worden (siehe z. B. [Bal90]).

Hierbei wird aber einerseits übersehen, dass ein überfrachtetes oder schlecht synchronisiertes Informationsangebot den Lerneffekt auch schmälern kann. Mit „schlecht synchronisiert“ wird hier auf die Synchronisation der Modalitäten angespielt, also z. B. der visuell und auditiv angebotenen Informationen. Andererseits wird bei obiger Vermutung übersehen, dass für das Lernen und Verstehen aus kognitionspsychologischer Sicht weniger die angesprochenen Sinneskanäle, sondern die internen Codierungen und Verarbeitungsprozesse des Lernenden ausschlaggebend sind; für eine genauere Diskussion siehe [Wei95]. Man spricht hier auch vom Unterschied zwischen *Codierung* (der internen Repräsentation) und *Modalität* (dem angesprochenen Sinneskanal).

Ein Beispiel ist ein Lernender, der visuell angebotene Informationen – z. B. in einem Lehrbuch – verarbeitet, indem er den Lernstoff intern auditiv wiederholt, sich also in Gedanken nochmal vorsagt. Bezüglich des Sprachgebrauchs folgen wir hier der gängigen Unterscheidung zwischen erinnerten oder vorgestellten (= internen) und mit den Sinnen gegenwärtig wahrnehmbaren (= externen) Bildern, Geräuschen, Körperempfindungen, Gerüchen oder Geschmacksempfindungen.

Ein anderes Beispiel für den Unterschied zwischen Codierung und Modalität ist ein Lernender, der mithilfe eines Hörbuchs lernt und dabei wesentliche Inhalte bildlich visualisiert, bis sich innerlich ein Gesamtbild formt, das sich im Augenblick des Verstehens an einen bestimmten (inneren) Platz bewegt und dessen Stimmigkeit durch ein bestimmtes, angenehmes Körpergefühl ausgedrückt wird. So „weiß" der Lernende, wann er verstanden hat. In diesem Fall wurde also eine extern-auditiv dargebotene Information intern-visuell (mit extern-kinästhetischer Komponente) repräsentiert.

Die erwähnten internen Prozesse und Repräsentationen des Lernenden sind also zum Teil hochgradig individuell und sogar weiter unterteilbar in Submodalitäten [BG01, BM00]. Submodalitäten der visuellen Modalität sind z. B. die dreidimensionale Position des Bildes, die Größe und der Abstand vom Sehenden. Somit sind diese Prozesse prinzipiell auch unabhängig von der konkreten Repräsentation des Lernmaterials.

Allerdings lassen sich die intern ablaufenden kognitiven Prozesse durch die äußerliche Repräsentation des Lernstoffes bis zu einem gewissen Grad lenken: Wenn etwa einem Lernenden, der sonst textuell angebotene Begriffe intern auditiv repräsentiert, zum richtigen Zeitpunkt zusätzlich ein Bild angeboten wird, kann dieses intern als zusätzliche visuelle Repräsentation dienen. Ein anderes Beispiel sind bewegte Bilder: Diese lassen sich intern nur visuell repräsentieren. [Wei95] empfiehlt in diesem Zusammenhang u. a.:

- Die Information sollte möglichst ohne Überlastung gut synchronisiert auf mehrere Sinnesmodalitäten verteilt werden, was für den Einsatz multimedialer Techniken spricht; mit synchronisiert ist hier gemeint, dass z. B. die auditive Erklärung *zum gleichen Zeitpunkt* angeboten wird, in dem die visuelle Information gezeigt wird, oder genauer gesagt, wahrgenommen wird.
- Dem Lernenden sollte durch Interaktivität eine Verankerung der Lerninhalte ermöglicht werden; von einem gewissen Standpunkt aus ist Interaktivität, also die Einbindung eines „Tuns" ebenso eine Synchronisation, nämlich diejenige der sowieso angebotenen visuellen und/oder auditiven mit der externen kinästhetischen Modalität „tun" – inklusive dafür notwendiger Körperbewegungen und Muskelaktivitäten wie beim Klick auf eine Maustaste, Veränderungen des Atemmusters oder Ähnliches.

Ein Ansatz, der speziell die gleichzeitige Codierung der dargebotenen Information in Bild und Erklärung (auditiv oder textuell) erforscht hat, ist die Theorie der Doppelcodierung von Paivio [Pai86]. Hier wird davon ausgegangen, dass im Falle tatsächlich multimodal angebotener Informationen, also z. B. Graphik und Erklärungen oder Graphik und erläuternder Text, diese vor allem dann den Lernprozess unterstützen, wenn sie zeitlich (und im Falle von Text auch räumlich) möglichst nahe angeboten werden. Auf diese Art bestehen laut Paivio die besten Chancen, sinnvolle interne Repräsentationen zu bilden.

Ein für den Lernerfolg immer wieder als entscheidend genannter Aspekt ist der Motivationsgewinn durch multimediale Techniken. Hier muss allerdings klar unterschieden werden zwischen einem diffusen Fesseln der Aufmerksamkeit durch viel sensorischen Input wie beim rein passiven Fernsehen oder Computerspielen, und der klaren Lenkung einer gerichteten Aufmerksamkeit auf das Objekt des Lernens.

Ein Beispiel für eine Situation der letzteren Art ist die Verwendung von flüssigen Animationen. Unter „flüssig“ verstehen wir hier Animationen, die einen bewegten Film darstellen, im Gegensatz zu nacheinander und ohne Übergänge dargestellten Endzuständen eines Vorgangs. Flüssige Abläufe können vom Lernenden intern nur visuell repräsentiert werden und ermöglichen es, durch die nachvollziehbare Bewegung die Aufmerksamkeit des Lernenden zu lenken. Dieser Effekt wird dadurch unterstützt, dass keine abrupten Kontextwechsel verarbeitet werden müssen.

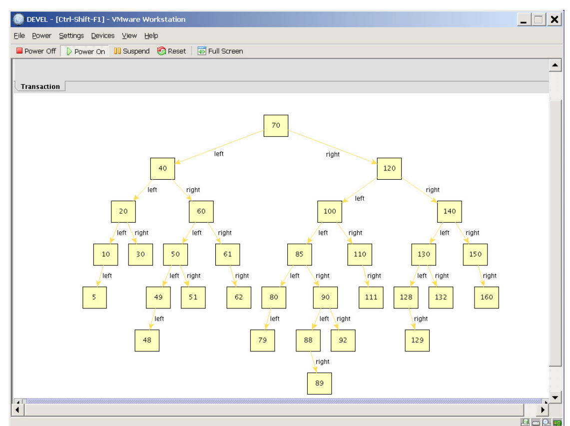
2.2. Flüssige Animationen

Dass flüssige Animationen wie im letzten Abschnitt erwähnt die Aufmerksamkeit des Lernenden lenken können, klingt möglicherweise etwas abstrakt und soll deswegen noch genauer beschrieben werden.

Eine interessante Frage bei der Visualisierung von Datenstrukturen ist, wie beim Übergang zwischen zwei komplexen Zuständen einer Datenstruktur die wesentlichen Details so kenntlich gemacht werden sollen, dass der Lernende sie erfassen kann und so erst die Chance hat, sinnvolle interne Repräsentationen zu bilden. Das ist zum Beispiel dann schwer vorstellbar, wenn sich in einer komplexen Struktur nur wenige Details ändern. Ein Beispiel im Bereich der Datenstrukturen sind die so genannten Rotationsoperationen auf den in der Einleitung erwähnten AVL-Bäumen, die man in Lehrbüchern oft als Abbildungen zweier Zustände (vor und nach der Operation) darstellt. Nun werden in Lehrbüchern natürlich kleine, bewusst übersichtliche Beispiele genutzt und die fraglichen Veränderungen zum Beispiel farblich gekennzeichnet. Trotzdem muss der Lernende mehrmals zwischen den beiden Bildern hin- und herschauen, um im besten Fall eine korrekte interne Repräsentation des Übergangs zwischen den beiden Zuständen zu bilden. Abb. 5 zeigt ein solches Beispiel in einer Visualisierungsumgebung ohne Kennzeichnung der gerade wesentlichen Stellen. Der Leser kann selbst beurteilen, ob es einfach ist, sofort die Stellen zu identifizieren, an denen sich Knoten bewegt haben, und die Stellen, wohin sich diese bewegt haben. Man möge hierbei aber auch berücksichtigen, dass es leichter ist, für nebeneinanderstehende Bilder Unterschiede zu finden als in einer Visualisierungsumgebung, die diese Bilder *nacheinander* produzieren würde.

Im Falle der Visualisierung einer Rotationsoperation auf einem AVL-Baum vernünftiger Größe haben die Lernenden unserer Meinung nach nur dann eine Chance, der Operation sofort folgen zu können, wenn flüssige Animation eingesetzt wird. Dann gibt es nur *ein* Bild des gesamten Baumes, das im Ganzen unverändert bleibt, in dem sich aber die wenigen sich ändernden Bestandteile in einer flüssigen, kontinuierlichen Bewegung an ihren neuen Platz bewegen.

Ähnliche Überlegungen wurden auch bei Fragestellungen zu reiner Algorithmenanimation durchgeführt (siehe z. B. [BH98, Nap]), also in komplexeren Abläufen als reinen Schnittstellenoperationen auf einer Datenstruktur.



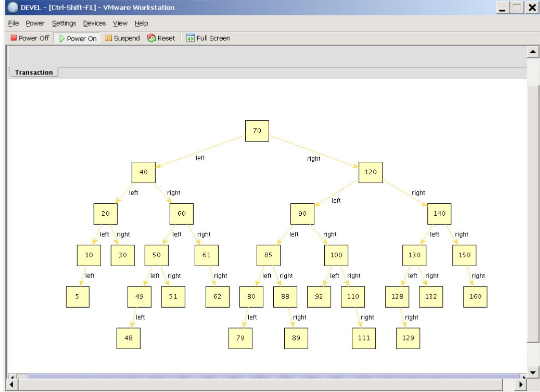


Abbildung 5: Operation auf einer Datenstruktur; es ist schwer auf Anhieb zu sehen, was sich geändert hat, wenn die Bilder nacheinander dargestellt werden

Wir fügen also zu den bisher genannten grundsätzlichen Forderungen der Synchronisation und der Interaktivität die Fähigkeit der kontinuierlichen Animation hinzu.

Nachdem bis hierher grundlegende Erkenntnisse aus der Erforschung des Lernens im Zusammenhang mit multimedialen Materialien vorgestellt wurden, werden wir in den nächsten Abschnitten die Ergebnisse einiger Studien über die Wirksamkeit existierender Visualisierungen zusammenfassen. Dabei ist es nicht unser Ziel, bestehende Ansätze zur Visualisierung aufzuzählen, sondern einen Überblick über durch Studien belegte oder vermutete lernwirksame *Eigenschaften* multimedialer Lernumgebungen zu geben. Deswegen werden wir bei den folgenden Beschreibungen den Fokus auf die beobachteten oder gemessenen Ergebnisse und die Schlussfolgerungen der Autoren in Bezug auf lernwirksame Eigenschaften multimedialer Lernumgebungen legen. Wir werden sehen, dass es durchaus gemischte Ergebnisse gibt und dass sich trotzdem wiederholt genannte Erfolgsfaktoren identifizieren lassen.

2.3. Experimente mit Visualisierungen im Allgemeinen

Sorting out Sorting: Baecker 1981

Ein frühes Beispiel der Verwendung animierter Abläufe in der Lehre von Algorithmen ist der Lehrfilm 'Sorting out Sorting', der einen unmittelbaren Vergleich von neun verschiedenen Sortieralgorithmen zeigte (beschrieben in [Bae98]). Auf diese Weise konnten die Zuschauer nicht nur durch unmittelbaren Vergleich den Unterschied in der Komplexität und damit auch in der Geschwindigkeit der Algorithmen sehen, sondern zusätzlich eine visuelle Repräsentation der verschiedenen verwendeten Konzepte bilden. Auch wenn in diesem Falle keine veröffentlichte formale Evaluation stattfand, wurde der Einsatz dieses Videos als sehr bereichernd für die Lehre empfunden und wird im Zusammenhang von Algorithmenanimationen immer wieder erwähnt. Außerdem erwähnt Baecker in [Bae98], dass es sich in Interviews mit Studierenden und

einem unveröffentlichten Experiment zeigte, dass die Substanz der Algorithmen gut aufgenommen worden war. Dass dieses Konzept immer noch funktioniert, zeigt sich auch darin, dass trotz fehlendem Marketing immer noch Kopien des Films verkauft werden, insgesamt schon über 600.

Die Autoren betonen u. a. die Wichtigkeit von guten Erklärungen, Timing und flexibler Kontrolle über die Bewegungsabläufe gegenüber der relativen Unwichtigkeit einer schönen und bunten Darstellung.

Mayer 1991, 1992:
Ein Beispiel, das nicht aus dem Bereich der Algorithmen und Datenstrukturen stammt, aber in einer durchgeführten Evaluation den Nutzen animierter Darstellung zeigte, sind mehrere Studien von Mayer und Anderson in den Jahren 1991 und 1992 [MA91, MA92]. Hier wurden Gruppen von Kindern und Erwachsenen die Funktionsweise einer Fahrradpumpe bzw. einer hydraulischen Bremse mit Animationen vorgeführt. Einer Vergleichsgruppe wurde nur eine theoretische Erklärung gegeben. Bei einer dritten und vierten Gruppe schließlich wurden Erklärungen und Animationen kombiniert, einmal nacheinander und einmal gleichzeitig, so dass im letzteren Fall die Erklärung im gleichen Moment abgegeben wurde, in dem die Visualisierung auf dem Bildschirm stattfand. In allen Fällen schnitten diejenigen Gruppen am besten ab, die Erklärung und Animation gleichzeitig präsentiert bekamen, die anderen Gruppen waren ungefähr vergleichbar. Als Erklärungsversuch wird die oben erwähnte 'dual coding'-Theorie von Paivio herangezogen.

Rieber 1996:
Rieber führte im Jahr 1996 zwei sich nur in der Zeitdauer unterscheidende Experimente durch (siehe [Rie96]), die drei Arten von Feedback in einer Simulation miteinander verglichen: Graphisches (animiertes) Feedback, textuelles Feedback und graphisches plus textuelles Feedback. Insgesamt 89 Universitätsangestellte wurden zufällig in drei Gruppen eingeteilt, in denen sie drei computerbasierte Simulationen nutzten, die das Verhältnis zwischen Beschleunigung und Geschwindigkeit modellierten. In allen drei Simulationen konnten sie eine 'up' und eine 'down'-Schaltfläche nutzen, um einen Ball zu kontrollieren. Die Graphik-Gruppe sah den Ball auf dem Bildschirm, die textuell arbeitende Gruppe bekam permanentes textuelles Feedback, wo sich der Ball gerade befand und die kombinierte Gruppe bekam beide Arten von Feedback.

Die Teilnehmer bekamen während der Simulation wie in einem Computerspiel Punkte, basierend auf der Genauigkeit, mit der sie den Ball an einer geforderten Stelle umdrehen ließen. Nach den Simulations-Sitzungen wurde ein Multiple-Choice-Test durchgeführt, der Aufschluss über explizit gewonnenes Wissen geben sollte.

Die Resultate zeigten keinen signifikanten Unterschied zwischen den Ergebnissen der drei Gruppen im Multiple-Choice-Test, aber einen klaren Unterschied in den erreichten Punkten: Die graphische und die gemischte Gruppe schnitten wesentlich besser ab als die textuell arbeitende Gruppe.

Die Forscher deuteten die Ergebnisse so, dass das animierte Feedback das Lernen auf einem taktilen und visuellen Level gefördert hatte, aber die Lernenden nicht darin unterstützte, das Wissen in eine verbale, explizite Form zu bringen.

Bei diesem sehr einfachen Beispiel der Steuerung eines Balls stellt sich die Frage der Über-

tragbarkeit der Ergebnisse auf multimediale Lehre; da mögen berechtigte Zweifel bestehen. Trotz allem hat dieser Versuch gezeigt, dass die immerhin auch permanent vorliegenden textuellen Feedback-Informationen nicht so gut genutzt werden konnten wie die visuelle Repräsentation des Balls.

Traynor 2003:
Repräsentativ aus einer Menge von Studien und Erkenntnissen zum Thema E-Learning in einem nicht-technischen Gebiet sei noch eine neuere Studie erwähnt [Tra03], die bewusst mit Probanden von ziemlich unterschiedlichem Hintergrund arbeitete. Es handelte sich dabei um Schüler aus vier verschiedenen amerikanischen Ausbildungsprogrammen: Sonderschüler, nicht-Englisch-Sprechende, wenig-Englisch-Sprechende und reguläre Schüler. Diese vier Gruppen absolvierten Lerneinheiten des für Kinder entwickelten E-Learning-Systems Cornerstone [Cor05]. Dieses System bietet die Möglichkeit, Sprache und einfache mathematische Konzepte in verschiedenen Schwierigkeitsgraden zu lernen. In einer formalen Evaluation mit Vortests und Nachtests machten alle vier Gruppen signifikante Fortschritte. Außerdem gab es einen statistisch relevanten Unterschied im Fortschritt zwischen den Sonderschülern und den regulären Schülern; letztere machten besonders große Fortschritte im Verhältnis zu ihrem Wissen vor dem Test.
Der Autor macht drei Hauptmerkmale von Cornerstone für diese Resultate verantwortlich:

- Es werden praktische Aktivitäten ermöglicht, die herausfordernd sind und die Neugierde der Kinder wecken, so dass Interaktivität intensiv genutzt wurde.
- Cornerstone bietet den Lernstoff teilweise in einem Fantasie-Kontext an, so dass Berührungsängste minimiert wurden.
- Den Lernenden wird ein großes Maß an Kontrolle über ihr Lernen (das Tempo, Wiederholbarkeit von Abläufen etc.) gegeben, ganz im Gegensatz zu einem starren Ablaufszenario.

Obwohl dieses Experiment einen Lernfortschritt aller Gruppen nachweisen konnte und zeigen konnte, dass die Gruppe der regulären Schüler im Verhältnis zu ihrem Vorwissen am meisten profitierte, fehlt doch der Nachweis, dass der Einsatz von Cornerstone einen Lernvorteil gegenüber klassischer Lehre bot. Dafür wäre der Einsatz jeweils einer Vergleichsgruppe von Schülern pro Cornerstone-Gruppe sinnvoll gewesen. Wir werden im Rahmen dieser Arbeit in Kapitel 9 ebenso eine Evaluation des hier vorgestellten Ansatzes durchführen und dort versuchen, einen Vorteil gegenüber Vergleichsgruppen nachzuweisen.

2.4. Studien über die Wirksamkeit von Algorithmenanimationen

Stasko et al. 1993:
Eines der ersten Algorithmenanimations-Systeme, für das eine formale Evaluation stattfand, ist das von Stasko entwickelte System XTANGO [Sta92]. XTANGO basiert wie schon frühere Systeme (z. B. BALSA, [Bro88] oder TANGO, [Sta90]) auf so genannten 'interesting events', d. h. der Programmierer des Algorithmus entscheidet sich für interessante Stellen und fügt an diesen Stellen Aufrufe von Funktionen der Animationsbibliothek hinzu. Die Festlegung der interessanten Stellen erfolgt manuell und dient dazu, eine Animation unwesentlicher Details möglichst zu vermeiden. Zum Beispiel wird bei einem Sortieralgorithmus die Vertauschung zweier Feldelemente oft als interessant angesehen, aber nicht unbedingt jede Zuweisung an eine Hilfsvariable.

Das entstehende Programm wird normal übersetzt und gestartet. Neu in XTANGO, im Vergleich zu früheren Systemen, ist die Möglichkeit, flüssige Animationen durch die explizite Angabe von Pfaden, auf denen sich die graphischen Primitive (z. B. Kreise, Ellipsen, Rechtecke) bewegen sollen, zu erzeugen.

Die von Stasko, Badre und Lewis im Jahr 1993 durchgeführte Studie nutze eine mit XTANGO erstellte Animation eines Algorithmus auf der so genannten *pairing heap*-Datenstruktur [FSST86]. Genauer gesagt wurden Operationen auf einer Prioritätswarteschlange mithilfe der 'pairing heaps' implementiert und diese als baumartige Strukturen dargestellt. Die Hypothese vor der Studie war, dass die Animation den Studierenden helfen würde, den Algorithmus zu lernen; insbesondere wurde vermutet, dass sie mehr dem prozeduralen als dem deklarativen Verständnis hilfreich wäre.

Es wurden zwei Gruppen zu je 10 freiwilligen Studenten miteinander durch einen 45-minütigen Test verglichen. Die erste Gruppe durfte sich zuvor in eine Beschreibung der Datenstruktur einlesen, die zweite Gruppe hatte insgesamt die gleiche Zeit zur Verfügung, um sich einzulesen und anhand von Animationen Experimente zu machen.

Die Ergebnisse waren ein nicht-signifikantes besseres Abschneiden der Animationsgruppe. Diese Gruppe beendete den Test auch im Durchschnitt etwas schneller. Die Ergebnisse beider Gruppen waren insgesamt eher enttäuschend. Allerdings gab es interessantes Feedback der Studierenden nach dem Test: Einige hätten sich zum Zeitpunkt der Animation eine Erklärung gewünscht, was eigentlich gerade geschah. Anderen war es während der Animation klar, was geschah, sie konnten es aber später nicht mehr abrufen. Als besonders positiv wurden die Einfärbungseffekte und die flüssigen Animationen bewertet und außerdem die Möglichkeiten, selbst ausprobieren zu können und Einfluss in Form der Geschwindigkeitskontrolle (eine rudimentäre Form von Interaktivität) nehmen zu können. Als negativ wurde u. a. das Fehlen der Möglichkeit, schrittweise den Algorithmus durchlaufen zu können (es gab nur die Möglichkeit 'pause' / 'unpause' zu wählen) und Schritte rückwärts machen zu können ('undo' / 'redo') bewertet. Insgesamt war das Interesse der Animationsgruppe geweckt, um sich weiter mit der Datenstruktur zu beschäftigen. Die Schlussfolgerungen der Autoren sind folgende Empfehlungen:

- Es sollte verständliche zusätzliche Beschreibungen von Seiten des Lehrers oder in textueller Form geben.
- Spezifische Lernziele sollten vor Entwicklung einer speziellen Animation feststehen.
- Benutzertests sollten möglichst früh in die Entwicklungsphase eines neuen Systems eingebaut werden, weil sie wertvolles Feedback liefern können.
- Zurück- und Vorwärtsspulen, also ein höherer Grad an Interaktivität, sollte möglich sein.

Lawrence 1994:
Im Jahr 1994 testeten Lawrence, Badre und Stasko die Effekte der Algorithmenanimation von Kruskals Algorithmus zum Finden des minimalen Spannbaums eines Graphen im Vergleich zu einer Folien-Präsentation desselben Themas [LBS94]. Verwendet wurde wieder XTANGO und zusätzlich POLKA [SK93], eine C++-Animationsbibliothek, die die Funktionsmerkmale von XTANGO implementiert.

Untersucht werden sollte diesmal primär der Nutzen von Animationen in einer typischen Übungssituation, insbesondere der Unterschied verschiedener Lehransätze, die in der Menge

der Kontrollmöglichkeiten der Studierenden und im Maß ihrer aktiven Einbeziehung im Lernprozess variierten. Dadurch sollten verschiedene Lerntypen angesprochen werden, im einfachsten Fall unterscheidbar in Studierende, die am besten durch eigenes Experimentieren lernen, und Studierende, denen eher die Entwicklung eigener Theorien im Lernprozess hilft.

62 Studierende wurden auf drei Arten unterrichtet: Die erste Art war nur eine Vorlesung über das Thema, die zweite Art bestand in Vorlesung und einem passiven Praktikum, die dritte Art in Vorlesung und einem aktiven Praktikum. Im aktiven Praktikum bekamen die Teilnehmer eine Beschreibung, wie sie einen Graphen erzeugen konnten, bevor sie die Arbeitsweise des Algorithmus auf diesem betrachteten. Im passiven Praktikum bekamen die teilnehmenden Studenten nur die Information, wie sie vorbereitete Graphen öffnen und die Arbeitsweise des Algorithmus auf dem jeweils geöffneten Graphen beobachten konnten. Da die Vorlesung auf zwei Arten gehalten wurde, nämlich mit vorbereiteten Folien oder mit POLKA-Animationen, gab es insgesamt sechs unterscheidbare Gruppen. Danach bearbeiteten alle Studierenden einen Multiple-Choice-Online-Verständnistest, Freiwillige zusätzlich einen Papier-Verständnistest.

Die Erwartungen waren, dass die Gruppen, die die Animationen in der Vorlesung sahen, besser abschneiden würden als die Folien-Gruppen. Außerdem wurde vermutet, dass die Praktikumsgruppen zusätzliches Verständnis gewonnen hätten.
Die Auswertung ergab folgende Ergebnisse:

- Ob bei der Präsentation des Algorithmus im Rahmen des Vortrags nun Animationen oder Folien verwendet worden waren, hatte weder auf die reinen Vortragsgruppen, noch auf die Laborgruppen einen signifikanten Effekt. Bei den Textaufgaben schnitt die Animationsgruppe im Schnitt sogar schlechter ab als die Foliengruppe. Die Autoren und Autorinnen werfen ein, dass dies auch an dem speziellen Algorithmus gelegen haben kann und bei anderen Algorithmen eventuell mehr von Animationen profitiert werden könnte.
- Die allgemeine Hypothese, dass die Laborgruppen alle besser abschneiden würden, hat sich so nicht bestätigt. Die passiven Laborgruppen hatte lediglich einen nicht-signifikanten Vorteil gegenüber den reinen Vortraggruppen. Erst bei den aktiven Laborgruppen konnte ein signifikanter Unterschied festgestellt werden. Animationen scheinen also beim Nacharbeiten des Gelernten nur dann einen signifikanten Lerneffekt zu haben, wenn sie interaktiv gestaltet werden.
- Die aktiven Laborgruppen waren in beiden Tests besser als die reinen Vortragsgruppen. Der Unterschied war jedoch nur bei den konzeptionellen Textaufgaben signifikant, bei den Multiple-Choice-Test-Fragen zu einzelnen Schritten des Algorithmus lediglich nicht-signifikant. Animationen scheinen also im Rahmen einer Nacharbeitung einen positiven Effekt auf das konzeptionelle Verständnis eines Algorithmus zu haben, weniger jedoch auf das Verständnis des Ablaufs an sich.

Najjar 1995:
In einem Papier aus dem Jahr 1995 [Naj95] stellt Najjar nochmal die Wesentlichkeit der Interaktivität bei Algorithmenanimationen heraus, indem er über 75 Studien aus den Jahren 1986 bis 1990 erwähnt, die den Mehrwert von Interaktivität nachgewiesen haben. Er definiert dabei Interaktivität nach Fowler als '*mutual action between the student, the learning system, and the learning material*'.

Byrne et al. 1996:
Im August 1996 führten Byrne, Catrambone und Stasko eine Studie mit XTANGO und POLKA durch [BCS96]. Diesmal wurden Animationen zusammen mit Anweisungen an die Testpersonen verwendet, die Vorhersagen über den weiteren Ablauf des Algorithmus verlangten.

Im ersten Experiment wurde ein Algorithmus zur Tiefensuche auf einem Baum untersucht, dessen Knoten mit den Buchstaben ’a’ bis ’z’ gekennzeichnet waren. Die Animation war mit POLKA erzeugt worden. Die 88 teilnehmenden Personen wurden zunächst auf zwei Gruppen aufgeteilt: Eine Gruppe, die Vorhersagen während des Trainings machen musste, und eine Gruppe, die keine Vorhersagen machen musste. Jede dieser Gruppen wurde aufgeteilt in eine Gruppe, die mit Animationen arbeitete, und eine Gruppe, die mit Papier-Materialien arbeitete. Zunächst wurden die Vorhersagen miteinander verglichen, wobei sich ein statistisch verlässliches besseres Abschneiden der Animationsgruppen zeigte. Danach machten alle 88 Teilnehmer einen Test anhand anderer Beispiel-Bäume. Die Resultate waren nicht-signifikant bessere Ergebnisse sowohl der Vorhersage-Gruppe als auch der Animationsgruppe, wobei keine Wechselbeziehung zwischen Vorhersage und Animation nachgewiesen werden konnte.

Im zweiten Experiment wurden zwei Operationen auf der weiter oben erwähnten Datenstruktur ’binomial heap’ mithilfe des Visualisierungssystems XTANGO unterrichtet. Es handelte sich dabei um das Einfügen neuer Elemente und das Finden des minimalen Elements. Die 62 Teilnehmer füllten einen Fragebogen zur Bewertung ihres Hintergrund-Wissens aus, sahen ein 11-minütiges Video einer Vorlesung über binomial heaps und bekamen dann einen 12-seitigen Beschreibungstext über diese Datenstruktur. Die Aufteilung war analog zu derjenigen im ersten Experiment. Die Algorithmen-Gruppen hatten wieder weniger Zeit zum Lesen zur Verfügung, dafür die Differenzzeit für die Animation. Danach füllten alle Beteiligten wieder einen Abschlusstest aus. Diesmal lagen die Ergebnisse aller Gruppen wesentlich näher zusammen, so dass sich überhaupt keine statistisch relevanten Aussagen machen ließen. Auffallend war nur, dass es zwei relativ gut unterscheidbare Gruppen von Probanden unter den Vorhersagenden zu geben schien: Die einen sagten die Ergebnisse gut voraus und schnitten im darauffolgenden Test gut ab, die anderen waren in beidem schlecht. Ein Versuch, dieses Ergebnis zu erklären, ist die Vermutung der Autoren, einige Studierende hätten bei diesem komplexeren Algorithmus als im ersten Experiment sowohl das Vorlesungs-Video als auch das Papier-Material nicht verstanden und so auch weder Nutzen aus Vorhersagen noch aus der Animation ziehen können.

Alles in allem ziehen die Autoren die Schlussfolgerung, dass Algorithmenanimationen sorgfältig auf das jeweilige Klientel abgestimmt werden müssen. Auch solle möglichst verhindert werden, dass evtl. schon vorhandenes Wissen, für das bereits innere Repräsentationen bestehen können, in Konflikt gerät mit den Animationen der Werkzeug-Umgebung. Ein weiterer Vorschlag für spätere Studien ist die Zeitkomponente. Die Autoren mutmaßen, dass mögliche positive Effekte der Animationen durch den straff organisierten Zeitrahmen ausgelöscht worden sein könnten. Die Empfehlung ist, in einer späteren Studie mehr Zeit für „Herumspielen“ und Ausprobieren zu geben und erst danach einen weiteren Verständnistest durchzuführen.

Kehoe 1996:
Kehoe et al. führten 1996 eine weitere Evaluation des POLKA-Systems durch mithilfe einiger Animationen zu den Operationen *union* und *extract minimum* der ’pairing heap’-Datenstruktur [KS96]. Dabei durften die Studierenden die Lehrmaterialien (statische Materialien und die Ani-

mationen) nach Belieben nutzen, um einige Fragen zu den 'pairing heaps' schriftlich zu beantworten. Der Fokus lag hier nicht auf den richtigen Antworten der Studierenden, sondern auf der Untersuchung ihres Lernverhaltens. Es wurde festgestellt, dass die verfügbaren Algorithmenanimationen von den Studenten als hauptsächliches Lernmittel benutzt wurden, aber auf sehr unterschiedliche Art. Zwar sprangen die meisten Studierenden immer wieder zwischen Animationen und textuellen Beschreibungen hin und her, um die angebotenen Konzepte zu verstehen. Unterschiede gab es jedoch in der Art der Nutzung: Eine Studierende nutzte z. B. die Animationen nur für diejenigen Stellen, die ihr noch unklar waren, also sehr selektiv, während eine andere systematisch alle angebotenen Animationen durchging, nachdem ihr klargeworden war, dass ihr Verständnis nicht zur Beantwortung der Fragen ausreichte. Diese sehr individuelle Art dieser Benutzung erfordert nach Meinung der Autoren viel mehr Flexibilität von Seiten der Animationen, als dies bei den gängigen Animationen der Fall ist. Außerdem wurde betont, dass Studenten die Animationen mochten und sie interessant fanden.

Douglas 1996:
Douglas et al. widmeten sich im Jahr 1996 in einer Studie zunächst der Frage, wie Menschen natürlicherweise Algorithmen konzeptionalisieren würden und bis zu welchem Grad das mit herkömmlicher Algorithmen-Visualisierungssoftware übereinstimmt [DHM96]. Deshalb wurden im ersten Teil der Studie Zweiergruppen von Informatik-Studenten dazu angehalten, mit üblichen Kunsterziehungs-Materialien wie farbigen Papiersorten, Scheren und Stiften in verschiedenen Farben einen einfachen Sortieralgorithmus ('Bubble Sort') zu visualisieren. Im zweiten Teil kreierten sie eine Bubblesort-Visualisierung mit dem Algorithmen-Visualisierungswerkzeug LENS [MS94], das es erlaubt, aus C-Programmen Animationen zu erzeugen und zu visualisieren.

Der Hauptunterschied zwischen LENS und früherer Evaluationssoftware wie den schon vorgestellten Systemen besteht darin, dass Nutzer Animationen interaktiv aus graphischen Objekten und Transformations-Prozessen erzeugen können. Zusätzlich kann C-Quellcode in die Arbeitsumgebung geladen werden, so dass ein Vergleich von Quellcode und der Animation möglich ist.

Es zeigte sich, dass die Art, in der die Probanden den Algorithmus ohne Software darstellten, sich unter den Gruppen sehr stark unterschied und keine einzige dieser manuell erzeugten Visualisierungen innerhalb der Möglichkeiten von LENS lag. Trotz der Unterschiedlichkeit der manuell erzeugten Visualisierungen gab es gemeinsame Muster. Zunächst einigten sich die Teilnehmer innerhalb ihrer Zweiergruppe darauf, wie der Algorithmus arbeitete. Danach überlegten sie, wie sie den Algorithmus einem Anfänger erklären würden. Unter anderem wurden ausgiebig Farben genutzt und teils durch Legenden erklärt. Schließlich zeigten alle Gruppen die Funktionsweise des Algorithmus an einer beispielhaften Datenmenge. In einer nachträglichen Befragung zeigten sich mehrere Punkte, die ins LENS nicht möglich waren, aber gewünscht worden wären:

- Farbschemata für die Darstellung hervorzuhebender Daten
- Das Verstecken unwichtiger Details des Algorithmus im Pseudocode
- Die Möglichkeit, die Datenstruktur so anzuzeigen wie gewohnt – zum Beispiel zusammenhängende Rechtecke im Falle eines Feld-Typen

Stasko 1997:
1997 führte Stasko Tests mit einem neuen System durch, in dem Studierende selbst Animationen bauten (SAMBA; es wurden u. a. Animationen für Quicksort und Prims Spannbaum-Algorithmus auf Graphen erstellt, [Sta97]). Laut Stasko würden die beobachtete Motivation und Begeisterung alleine den Einsatz der Animations-Umgebung für diesen Zweck rechtfertigen. Die Studierenden revidierten während ihrer Aktivitäten mehrfach ihre Meinung von der Funktionsweise eines Algorithmus – offensichtlich bis sie stimmte. Die Klausur-Ergebnisse in dem geübten Bereich waren überwältigend gut. Allerdings wird eingeräumt, dass es unklar ist, ob es an der Beschäftigung mit dem Thema an sich lag, oder wirklich an den Animationen.

Da insbesondere der enorme Zeitaufwand für den Bau der Animationen eine große Rolle für den Lerneffekt spielen dürfte, wäre ein Vergleich sehr interessant, in dem eine Gruppe, die Animationen baut, mit einer Gruppe verglichen wird, die die gleiche Zeit für das Studium der Algorithmen aufwendet.

Stasko 1998:
Im Jahr 1998 fasst Stasko im Standardwerk 'Software Visualization' [SDBP98] einige seiner schon genannten Untersuchungen und weitere Erkenntnisse zusammen, indem er drei Hypothesen für den Einsatz von Algorithmenanimationen formuliert:

1. Selbst in Fällen, in denen kein nachweisbarer Lerneffekt vorliegt, gibt es einen pädagogischen Wert (u. a. einen Motivationsgewinn).
2. Nur passive Animationen zu beobachten führt zu geringen Lerneffekten.
3. Es kommt darauf an, wie Animationen eingesetzt werden: Beim aktiven Einsatz, wie beim Entwickeln eigener Animationen der Studenten oder bei anderen Formen der Interaktion wie der Formulierung von Vorhersagen gibt es große positive Effekte.

Hansen 1998:
Im Jahr 1998 führte Hansen eine Studie durch, die einen Ansatz evaluierte, der möglichst viele der ihm bis dahin bekannten lernförderlichen Eigenschaften multimedialer Lehre in einem Rahmenwerk zusammenfasste [HSNH98]. Es handelt sich dabei um das System HalVis [HNH02]. HalVis bezeichnet sich selbst als hypermediales Algorithmen-Visualisierungssystem, das nur zu einem Teil aus Animationen besteht. Um die Animationen herum gruppieren sich erläuternde Texte, statische Diagramme, auditive Erklärungen etc., die alle – angepasst für das jeweilige Thema – über Links miteinander verbunden sind, so dass die Lernenden diesen Pfaden folgen können. Weiterhin werden die Animationen nicht als ganzer Ablauf, sondern in kleinen Portionen dargestellt, um wesentliche Details sichtbar zu machen. Schließlich werden Interaktionen in Form von nicht-rückmeldenden, so genannten Tickler-Fragen, benutzerdefinierbaren Eingabemengen für den Algorithmus und Multiple-Choice-Fragen, die während des Erforschens erscheinen können, angeboten. Die Studierenden werden in diesem Prozess nicht unmittelbar mit den üblichen Algorithmenanimationen konfrontiert, sondern über die zunächst stattfindende Animation von Analogien schrittweise an das Thema herangeführt.

Zum Zeitpunkt der Vorlage des Berichtes waren vier Sortieralgorithmen und ein Graph-Algorithmus in HalVis implementiert. Hansen führte eine ausführliche Evaluation durch, die auch Vergleiche zu früheren Studien beinhaltete und die im Folgenden kurz beschrieben wird:

In den ersten zwei Experimenten wurde der Lerneffekt bei der Nutzung von HalVis mit dem eines Textbuches verglichen. Die Ergebnisse zeigten, dass sowohl Neulinge (in Experiment 1) als auch Fortgeschrittene (in Experiment 2) von den interaktiven hypermedialen Materialien weit mehr profitierten als vom Lernen aus dem Textbuch.

Im dritten Experiment wurde HalVis mit einer Kombination aus Textbuch und Problemlösen von Seiten der Studierenden verglichen. Hier ergaben sich vergleichbare Ergebnisse ohne signifikanten Unterschied zwischen den beiden Gruppen, mit leichter Favorisierung von HalVis.

In einem vierten Experiment wurden der Effekt vom Hören einer Vorlesung und dem nachträglichen Einsatz von HalVis mit der umgekehrten Kombination (erst HalVis, dann Vorlesung) verglichen. Da drei Verständnistests, nämlich vor dem Experiment, zwischen den beiden Lerneinheiten und nach dem Experiment, durchgeführt wurden, konnte auch ein Vergleich der Wirksamkeit der Vorlesung mit dem der hypermedialen Lernumgebung durchgeführt werden; dieser zeigte einen deutlichen Vorteil zugunsten von HalVis. Interessanterweise stellte sich kein signifikanter Unterschied zwischen den beiden erwähnten Reihenfolgen des Einsatzes von HalVis und der Vorlesung heraus: Die Studierenden, die die Animationen *nach* der Vorlesung nutzten, holten die andere Gruppe wieder ein. Trotzdem weisen die Ergebnisse nicht darauf hin, dass die Vorlesung nutzlos war, da die Ergebnisse über den gesamten Zeitraum, also mit Vorlesung und HalVis, einen größeren Lernfortschritt zeigten als bei der Nutzung durch HalVis alleine; sie trug nur weniger zu den Ergebnissen bei wie die Nutzung der Animationen, obwohl für die Präsenzveranstaltung ein äußerst renommierter Dozent gewonnen werden konnte, der auch bei Studenten-Umfragen immer sehr gut abschnitt.

Schließlich wurde in einem fünften Experiment eine Gruppe, die mit dem HalVis-System arbeitete, verglichen mit einer Gruppe, die eine ebenso vorbereitete TANGO-Animation zum gleichen Thema nutzte. TANGO entspricht im Wesentlichen dem oben vorgestellten XTANGO, allerdings ohne X-Oberfläche. Hier zeigte sich ein signifikanter Mehrwert von HalVis, wobei die TANGO-Ergebnisse vergleichbar waren mit denjenigen früherer Studien von TANGO/XTANGO [LBS94].

Der Autor nennt einige Vermutungen darüber, welche Eigenschaften von HalVis zu diesen positiven Ergebnissen beigetragen haben könnten. Vereinfacht ausgedrückt handelt es sich um zwei Hauptpunkte:

- Die schrittweise Einführung in das Thema und den Algorithmus durch die Nutzung von Analogien und danach immer weiterführende Teil-Animationen bis hin zum Quellcode und
- das erhebliche Maß an Interaktivität durch die einfachen und komplexeren Zwischenfragen, die Möglichkeit, im Kontext passenden Links zu folgen, nach Belieben kleine 'animation chunks' zu wiederholen und so weiter.

Kehoe et al. 1999:
Kehoe et al. untersuchten 1999 in einem Hausaufgabenszenario, auf welche Arten Studenten ein gegebenes Animationssystem nutzten, um einen neuen Algorithmus zu verstehen (veröffentlicht in [KST01]). Das Ziel dabei war, Informationen darüber zu gewinnen, wie bestehende Algorithmenanimationen in erfolgreiche Lernstrategien passen. Genutzt wurde das schon erwähnte Werkzeug POLKA, unterrichtet wurde wieder der 'binomial heap'.

Die Studie zeigte eine komplexe Nutzung des gegebenen Materials mit dem Effekt schwindender Berührungsängste dem Algorithmus gegenüber, was wiederum zu Interaktionen und einem positiven Lerneffekt führte. In der Evaluation, die gegen eine Vergleichsgruppe ohne Animationen durchgeführt wurde, schnitt die Animationsgruppe bei den meisten Fragen ähnlich gut ab, aber deutlich besser bei einigen prozeduralen Fragen zum geübten Algorithmus. Die Schlussfolgerungen der Autoren waren:

- Der pädagogische Wert von Algorithmenanimationen wird deutlicher in offenen, interaktiven Lernsituationen als in geschlossenen, zeitlich starren Szenarien mit der Vorgabe, zu einem bestimmten Zeitpunkt einen vorgegebenen Lerninhalt zu lernen.
- Selbst im schlechtesten Falle, wenn eine Animation nicht zum grundsätzlichen Verständnis eines Algorithmus beiträgt, verbessert sie den pädagogischen Ansatz durch Abbauen der Hemmschwelle und erhöht die Motivation.
- Besonders klar war die Verbesserung des Lerneffekts bei prozeduralen Fragen zum Algorithmus.

In den bisher in diesem Kapitel genannten Studien haben wir Erkenntnisse aus der Lernpsychologie, aus allgemeinen Studien zur Wirksamkeit von Visualisierungen und aus Studien über die Wirksamkeit von Animationen im Algorithmen-Umfeld vorgestellt. Dabei lag der Schwerpunkt nicht darin, eine auch nur annähernd vollständige Übersicht über Visualisierungsumgebungen und -werkzeuge zu geben, sondern Eigenschaften derjenigen Systeme zu zeigen, für die Studien über die Wirksamkeit vorlagen. Dabei wurde es klar, dass es Eigenschaften von Visualisierungssoftware gibt, die den Lerneffekt wirksam fördern können. Es scheint also einiges dafür zu sprechen, Animationen in der Lehre von Algorithmen und Datenstrukturen einzusetzen.

Allerdings gibt es auch noch eine andere Art von Untersuchungen, nämlich diejenigen, die erforschen, in welchem Maße Visualisierungen in der Lehre wirklich eingesetzt werden. Einige Erkenntnisse und Aussagen einer aktuelle Metastudie, die sich auch mit diesem Thema auseinandersetzt, werden wir im nächsten Abschnitt vorstellen.

2.5. Ergebnisse einer aktuellen Metastudie

Im Jahr 2002 hat sich die 'Working Group on Improving the Educational Impact of Algorithm Visualization' gebildet, die noch im Jahr 2005 aktiv ist. In ihrer Metastudie aus dem Jahr 2002 [NRA+02] wird die These aufgestellt, dass graphische Lernsoftware, egal wie gut sie entworfen ist, von geringem didaktischen Wert ist, wenn sie es nicht schafft, den Lernenden aktiv in das Geschehen miteinzubeziehen. Diese Aussage ist nach den bisher vorgestellten Erkenntnissen über die Wichtigkeit von Interaktivität nicht verwunderlich.

In dem erwähnten Bericht wird als größtes Problem derzeit die Diskrepanz zwischen der hohen Meinung vieler Lehrender im Bereich der Algorithmen und Datenstrukturen über die vermutete Wirksamkeit von Algorithmenanimationen und dem geringen Einsatz von Animationen in der Lehre angesprochen. Als Gründe werden zwei Haupthindernisse des Einsatzes von Algorithmenanimationen in der Lehre beschrieben:

- Aus der Perspektive der Lernenden hat Visualisierungs-Technologie nicht unbedingt einen didaktischen Wert.

- Und: Aus Sicht der Dozenten bedeutet der Einsatz von Visualisierungs-Technologie viel zu viel zusätzlichen Aufwand.

Als mögliche Lösung dieses Dilemmas wird vorgeschlagen, einer breiteren Öffentlichkeit als bisher bekanntzumachen, dass in bisherigen Studien Visualisierungswerkzeuge, die den Lernenden in aktive Handlungsweisen verwickeln, zu klar besseren Ergebnissen führten als rein passive Lehrmethoden – multimedial oder nicht. Dann bestünde sowohl die Hoffnung, dass die Skepsis der Studierenden nachlässt, als auch, dass Dozenten Mittel und Wege finden, Visualisierungs-Technologie einzusetzen; sei es, dass sie existierende Werkzeuge einsetzen, oder, dass sie Eigenentwicklungen in diesem Bereich starten.

Die offene Frage ist, wie besonders der Mehraufwand für Dozenten in den Griff zu bekommen ist. Wir werden im nächsten Abschnitt sehen, dass schon die Zusammenfassung der bisher genannten wesentlichsten Eigenschaften multimedialer Lehrmaterialien für die Lehre von Algorithmen genügt, einen ersten Eindruck zu vermitteln, dass das gar nicht so einfach sein wird.

2.6. Zusammenfassung

Die Essenz aus den bisherigen Abschnitten dieses Kapitels kann man wie folgt zusammenfassen: Durch den bloßen Einsatz multimedialer Materialien kann eine Verbesserung des Lerneffekts nicht garantiert werden.

Es gibt allerdings einige wesentliche Eigenschaften einer multimedialen Lernumgebung für die Unterstützung der Lehre von Algorithmen und Datenstrukturen, die nicht nur in der Art der Implementierung der Lernumgebung zu suchen sind, sondern auch in der Art des Einsatzes, der natürlich von der Implementierung ermöglicht werden muss. Besonders wesentlich ist es, dass folgende Möglichkeiten angeboten werden:

- Das Lernziel sollte von Anfang an klargemacht werden.
- Die Anfänger-Stufe sollte immer mit angeboten werden, da Anfänger und Kinder wegen ihrer noch nicht vorhandenen internen Repräsentationen am meisten profitieren; siehe auch Traynor 2003.
- Erforschbare Lernszenarien mit selbst bestimmbarem Tempo sollten vorhanden sein.
- Soviel Interaktivität wie möglich sollte angeboten werden (Schritte des Algorithmus abrufen und zurückspulen lassen, Operationen aufrufen lassen, Vorhersagen verlangen, kurze Zwischenfragen stellen).
- Unmittelbares Feedback auf Vorhersagen und Antworten sollte möglich sein.
- Nach Möglichkeit sollten die Lernenden selbst Animationen bauen können.
- Im Falle mehrerer genutzter Modalitäten (wie visuell und textuell oder visuell und auditiv) sollte eine saubere Synchronisation zwischen ihnen (z. B. durch gleichzeitiges Angebot) gewährleistet werden.
- Es sollte in irgendeiner Form eine Erklärung (auditiv, als beiliegende textuelle Beschreibung oder als Erklärung innerhalb der Animation) dessen angeboten werden, was gerade geschieht.

- Farbschemata für die Darstellung besonderer (Fehler-) Situationen sollten angeboten werden, da diese dann geeignet – nämlich als Sonderfall oder Fehlerfall – intern repräsentiert werden können (siehe auch das intuitive Verständnis der Versuchspersonen bei Douglas 1996).
- Im Falle animierter Schnittstellenoperationen sollten flüssige Abläufe gezeigt werden.

Andererseits werden anscheinend multimediale Techniken in der Lehre wenig oder gar nicht eingesetzt. Ein Haupthindernis dabei ist mit großer Wahrscheinlichkeit der Mehraufwand für den Dozenten.

Nun kann man sich die Frage stellen, worin dieser Mehraufwand besteht. Obige Liste vermittelt bereits einen guten Eindruck, dass es alles andere als trivial ist, Algorithmenanimationen und sicher auch Datenstruktur-Animationen, auf denen in den meisten der oben vorgestellten Studien nicht der Fokus lag, mit den angegebenen Eigenschaften zu erstellen; insbesondere gilt dies für Animationen, *die genau zu der vom jeweiligen Dozenten konzipierten Lehrveranstaltung passen*. Selbst, wenn wir bereit sind, auf einige der aufgezählten Eigenschaften zu verzichten, bleiben immer noch große Anforderungen übrig.

Als Beispiel können wir das System HalVis betrachten, das in der oben erwähnten Studie sehr gute Ergebnisse aufweisen kann, allerdings zu dem Preis eines sehr hohen, für jede Datenstruktur neuen Aufwands für die Erstellung geeigneter Analogien, Animationen für diese Analogien, so genannte 'animation chunks' verschiedenen Inhalts und verschiedener Detaillierung für den Algorithmus, vorbereiteter Stücke an Quelltext, interaktiven Fragen (teils Multiple-Choice) zum Lernstoff und, nicht zu vergessen, der im Kontext sinnvollen Querverbindungen all dieser Materialien.

Deswegen werden wir uns in Kapitel 3 die Frage stellen, ob es möglich ist, eine Lernumgebung zu entwerfen, die einerseits wesentliche didaktische Erfordernisse erfüllen kann und andererseits den Dozenten von der Notwendigkeit erhebt, jedesmal aufwändige, neue Animationen manuell zu entwickeln, die ja nicht nur auf die jeweilige Datenstruktur, sondern teils auch hochgradig auf die eigene Veranstaltung angepasst sein sollen. Dazu werden wir zunächst ein Szenario vorschlagen, das uns eine gute Ausgangsbasis liefert, um zuerst klarere technische Anforderungen als jetzt formulieren zu können, und dann in Kapitel 4 bestehende Ansätze und Werkzeuge im Hinblick auf ihre Fähigkeit, unsere Anforderungen zu erfüllen, unter die Lupe zu nehmen.

3. Anforderungen

In Kapitel 2 wurden Erkenntnisse aus der Lernpsychologie und aus existierenden Studien über die Wirksamkeit von Animationen vorgestellt und dabei wesentliche bisher bekannte Erfolgskriterien für multimediale Lehrwerkzeuge identifiziert.

Obwohl Price et al. bereits im Jahr 1998 die Zahl existierender Werkzeuge zum Bau von Animationen und bestehender Rahmenwerke für die Erstellung von Animationen auf über 150 schätzten (siehe[PBS98]), gibt es – wie in Kapitel 2 dargestellt – Anzeichen dafür, dass diese in der Lehre kaum eingesetzt werden; als Haupthindernis wird der Aufwand auf der Seite des Dozenten angegeben. Unsere persönlichen Erfahrungen decken sich mit diesen Vermutungen.

Worin besteht nun dieser Aufwand? Wir haben am Ende von Kapitel 2 als Erklärung angegeben, dass Dozenten, die Wert darauf legen, dass die eingesetzten Animationen einerseits den Lerneffekt wirksam fördern können und andererseits zur eigenen Veranstaltung passen, in den meisten Fällen darauf angewiesen sind, eigene Animationen fast von Null an neu zu entwickeln.

Der Grund dafür ist, dass sowohl die Kombination aus Eigenschaften, die besonders lernwirksam sind, als auch ihre Realisierung äußerst komplex sind. Zum Beispiel ist eine der wichtigsten zu fordernden Eigenschaften eine hohe Interaktivität der Animationen, so dass etwa in einem von den Lernenden interaktiv erforschbaren Szenario Zwischenfragen an den Lernenden gestellt werden können und dieser geeignet darauf reagieren muss; eine sinnvolle weitere sind Farbschemata zur unmittelbaren Markierung fehlerhafter oder besonderer Zustände.

Dem erfahrenen Programmierer wird es klar sein, dass selbst bei Vorliegen einer ausgefeilten Animation schon minimale Änderungen in der Definition der Datenstruktur oder der Schnittstellenoperationen oder des zu lehrenden Algorithmus für eine bestimmte Datenstruktur genügen, um große Änderungen in der Implementierung der Animation notwendig zu machen. Der Grund ist der möglicherweise sehr komplexe innere Zustand der Datenstruktur, der je nach Datenstruktur u. a. laufend auf die Erfüllung der Invarianten überprüft werden muss, mit korrektem Layout visualisiert werden muss und für verschiedene Arten der Interaktion genutzt werden muss – und zwar inklusive der passenden interaktiven und programmatischen Schnittstellen.

Ein hohes Maß an Interaktivität mit austauschbaren Beispielen stellt eben viel höhere Anforderungen an eine Animationsrealisierung als ein festes Beispiel mit festem Ablauf und wenigen vorab festgelegten Interaktionsmöglichkeiten.

Das Problem ist nun, dass es nicht nur viele verschiedene Algorithmen auf vielen verschiedenen Datenstrukturen gibt, sondern dass sich selbst gängige Datenstrukturen je nach Implementierung oder Unterrichtsschwerpunkt des Dozenten tatsächlich oft in kleinen Details unterscheiden, die eine eigentlich für diese Datenstruktur vorhandene Animation ggf. unbrauchbar machen können. Hier sprechen wir aus eigener Erfahrung, da uns die Situation wohlbekannt ist, dass eine Animation durch eine geänderte Definition einer Datenstruktur unbrauchbar wird.

Die Frage ist: Ist es möglich, ein System für die Animation von Algorithmen und Datenstrukturen auf eine solche Art zu entwerfen, dass bei einer in diesem System vorliegenden Animation einer Datenstruktur bei relativ kleinen Änderungen an dieser Datenstruktur auch schnell wieder eine Animation für die geänderte Definition der Datenstruktur erstellt werden kann? Das ist sicherlich nicht einfach, wenn die Animationen in einer gängigen Programmiersprache wie Java, C oder auch Visual Basic erstellt sind.

In den folgenden Abschnitten werden wir uns der Frage widmen, welche technischen Ei-

genschaften eine Visualisierungsumgebung haben muss, um aus einer Spezifikation einer Datenstruktur in einer gegebenen Spezifikationssprache in einem zumindest teilweise automatisierten Verfahren Animationen mit den wesentlichsten geforderten Eigenschaften aus Kapitel 2 zu erstellen. Dazu werden wir im nächsten Abschnitt zunächst einen Ansatz vorschlagen, der mit einer Spezifikationssprache für Datenstrukturen arbeitet, und ein grobes Szenario für den Einsatz dieses Ansatzes beschreiben.

Aus diesem groben Szenario werden dann durch Verfeinerung eine Reihe von Anforderungen an eine Visualisierungsumgebung erarbeitet, die genutzt werden können, um existierende Werkzeuge und Paradigmen auf Brauchbarkeit für unseren Ansatz zu untersuchen.

3.1. Grundlegendes Szenario

Als Lösung des im letzten Abschnitt beschriebenen Dilemmas schlagen wir ein Rahmenwerk vor, mit dem man imstande ist, Visualisierungen und Animationen von Algorithmen und Datenstrukturen mit den wesentlichen in Kapitel 2 genannten Eigenschaften in Abhängigkeit irgendeiner Form von Spezifikation von Datenstrukturen und Algorithmen zu erstellen. Auf diese Weise genügt es zur Erzeugung einer ersten Visualisierung für eine neue Datenstruktur, eine neue Spezifikation zu erstellen. Es handelt es sich also um eine Art 'rapid prototyping'. Wenn dann ein Prototyp existiert, können weitere Anpassungen an Darstellung, Layout und Meldungen vorgenommen werden. Beim Anpassen einer Animation für eine etwas geänderte Datenstruktur muss dann nur die Spezifikation angepasst werden.
Genauer gesagt, setzen wir folgendes grundlegende Szenario voraus:

- Der Dozent will eine bestimmte Datenstruktur und evtl. einen darauf aufsetzenden Algorithmus lehren.
- Er spezifiziert die Datenstruktur und / oder den Algorithmus in einer geeigneten Spezifikationssprache.
- In wenigen (im Rahmenwerk festgelegten) automatisierten Schritten erhält er daraus eine Instanz einer Visualisierungsumgebung, die interaktives Arbeiten und Experimentieren ermöglicht. Im Falle einer interaktiven Übung wird der Dozent in irgendeiner Form zusätzlich darin unterstützt, einen Ablauf bzw. ein Übungsblatt, an dem sich der Lernende orientieren wird, vorzugeben.
- Er führt Konfigurationsschritte aus, die das Verhalten und besonders die Oberfläche der generierten Laufzeitumgebung an den jeweiligen Anwendungszweck anpassen.
- Der Dozent oder ein Studierender startet die Instanz und führt nach einem impliziten oder expliziten Fahrplan eine mehr oder weniger interaktive Sitzung an einer Algorithmenanimation durch.
- Während dieser Sitzung können interaktiv weitere Konfigurationsschritte ausgeführt werden, die vom Dozenten vorher so vorgesehen worden sind.

Abb. 6 stellt das eben beschriebene grundlegende Szenario graphisch dar. Es können also mindestens zwei Arbeitsumgebungen unterschieden werden:

- Die Umgebung, in der die Datenstruktur spezifiziert oder an neue Zwecke angepasst werden kann, soll im Folgenden *Entwicklungsumgebung* heißen. Die Entwicklungsumgebung wird üblicherweise vom Dozenten bedient.

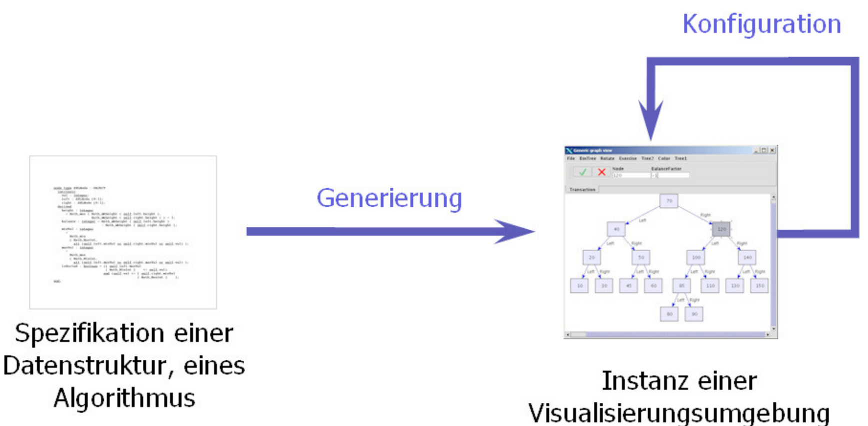


Abbildung 6: Erstellungsprozess einer neuen Instanz der Visualisierungsumgebung

- Die Umgebung, in der eine fertig generierte Instanz der Visualisierungsumgebung eingesetzt wird, soll im Folgenden *Laufzeitumgebung* heißen. Diese Laufzeitumgebung kann für Test- oder Vorführzwecke vom Dozenten oder – für die eigentlich im Mittelpunkt stehenden Übungszwecke – vom Studierenden genutzt werden.

Damit vereinfacht unserer Grundkonzept die bekannte Einteilung von Price et al. [PBS98] für reine Algorithmenanimations-Systeme, die von vier Rollen ausgeht: Dort entwirft der *Programmierer* den Algorithmus, der *Entwickler* stellt das Animationssystem zur Verfügung, der *Visualisierer* spezifiziert die Animation und der *Benutzer* nutzt die Animation (die Begriffe wurden ins Deutsche übertragen).

In unserem geforderten Konzept wird bereits ein System vorausgesetzt, das Animationen generieren kann, also ist die Arbeit des *Entwicklers* bei bestehendem System bereits getan. Der Algorithmus und / oder die Datenstruktur werden als existierend vorausgesetzt, wodurch auch die Rolle des Programmierers vor der Nutzung der geforderten Visualisierungsumgebung ausgeübt wurde, und vom Dozenten spezifiziert. Daraus soll dann nicht nur eine Animation erzeugt werden, sondern ein hochgradig interaktives System, das u. a. der selbständigen Erforschung von Lernräumen durch die Studierenden dienen kann. Somit teilen sich das System und der Dozent die Rolle des *Visualisierers*. Falls der Dozent auch die entstehende Laufzeitumgebung für Vorführ-, Vorbereitungs- oder Testzwecke benutzt, übt er zusätzlich die Rolle des *Benutzers* aus. Der Lernende übt in unserem Ansatz nur die Rolle des *Benutzers* aus.

Da es unseres Wissens kein System zur Erstellung von Animationen gibt, das alle bisher genannten Erfordernisse erfüllt, geht es nun darum, aus den genannten Grundanforderungen konkrete Anforderungen an Werkzeuge und Werkzeug-Paradigmen zu formulieren.

Es gibt eine große Menge an möglichen Eigenschaften eines Visualisierungssystems für Algorithmen und Datenstrukturen, von denen einige formuliert wurden (siehe z. B. [Glo98], [Röß02]). In [Röß02] werden 69 Eigenschaften für Algorithmenanimations-Systeme unterschieden. Diese sind für unsere Zwecke aber nicht unmittelbar zu gebrauchen, weil sie einerseits auf

die reine Algorithmenanimation abzielen, das heißt, im Vordergrund steht die Frage:

> *Welche Eigenschaften können für ein Algorithmenanimations-System sinnvoll sein, das für einen gegebenen Algorithmus eine Animation erstellt?*

Für uns ist das aber nur ein Teil der Fragestellung. Stattdessen lautet unsere Frage:

> *Welche Eigenschaften können für ein System sinnvoll sein, das aus der Spezifikation einer Datenstruktur und ggf. eines Algorithmus eine an die Datenstruktur angepasste, hochgradig interaktive Visualisierungsumgebung erzeugen kann?*

Der Unterschied besteht zum Einen in der zusätzlichen Betonung des Arbeitens auf der Datenstruktur selbst; ob mit einfachen Operationen oder im Rahmen eines oder mehrerer Algorithmen, ist dabei zunächst nicht entscheidend. Die erforderlichen Spezifikationen der Datenstruktur sollen leicht anpassbar sein und werden sich deshalb von der Spezifikation eines Algorithmus in programmatischer Form deutlich unterscheiden.

Zum Anderen fordern wir ein System, das wegen seines starken interaktiven Potentials am ehesten mit Simulationssystemen vergleichbar ist, wie sie bisher z. B. in der Medizin [Wei98], in der Physik [xyz05] oder im Automobilbau eingesetzt wurden [Sim05]. Allerdings werden ähnliche Systeme inzwischen auch vermehrt in der Lehre der Informatik eingesetzt. Ein Beispiel im Compilerbau ist das Simulationssystem Ganimal [Ker03].

Ein weiterer Grund für die nur bedingte Anwendbarkeit bestehender Vorschläge für Anforderungen an Animationssysteme für unser Grundszenario liegt in deren Aufbau als Klassifikationsschema, in das sich bestehende Systeme einordnen lassen. Dadurch werden z. B. Anforderungen genannt, die sich gegenseitig widersprechen oder sehr nebensächlich sind.

Im Gegensatz dazu brauchen wir eine Menge von Anforderungen als Zielvorgabe. Dazu ist es zunächst notwendig, zu priorisieren. Wir werden Eigenschaften vorstellen, die sich aus den in Kapitel 2 identifizierten Erfordernissen ergeben und danach weitere, die in einem Brainstormingprozess als wünschenswert zusammengetragen wurden. Dabei werden wir das größte Gewicht auf die Einfachheit und Übersichtlichkeit der Spezifikation der Datenstruktur (ohne unnötigen Ballast) und möglichst große Interaktivität der Laufzeitumgebung bis hin zu völlig frei erforschbaren Szenarien legen. In einem zweiten Schritt werden wir dann anhand einer Evaluation identifizieren, welcher Anteil der wünschenswerten Eigenschaften mit existierenden Werkzeugen und Paradigmen realistisch umsetzbar ist, und zunächst nur mit dieser letzteren Teilmenge weiterarbeiten.

3.2. Anforderungen

In diesem Abschnitt wird eine Liste von Anforderungen erstellt, die ein System erfüllen sollte, das in unserem geforderten Grundszenario einsetzbar ist.

Nach den Erkenntnissen aus Kapitel 2 – insbesondere über die Wichtigkeit der Aktivierung des Lernenden und der Reduzierung des Aufwands der Dozenten bei der Spezifikation – sind mindestens folgende Eigenschaften eines Visualisierungssystems für Algorithmen und Datenstrukturen wünschenswert:

- Datenstrukturen und Algorithmen sollen in einer Sprache spezifizierbar sein, die es auch ungeübten Dozenten ermöglicht, ohne allzu großen Mehraufwand Nutzen aus diesem Konzept zu ziehen. Es soll sich also bei der Spezifikationssprache entweder um eine weiträumig bekannte Sprache oder ein für die Spezifikation von Datenstrukturen bekanntes oder „natürliches“ Konzept handeln.

- Falls eine Spezifikation für eine Datenstruktur vorliegt, soll eine Spezifikation einer ähnlichen bzw. etwas komplexeren Datenstruktur durch Änderungen der ursprünglichen Spezifikation an möglichst wenigen Stellen möglich sein.

- Die Erstellung der Spezifikation erfolgt werkzeugunterstützt.

- Die im Werkzeug erstellte Spezifikation lässt sich in einem Format abspeichern oder weiterverarbeiten, aus dem sich eine lauffähige Instanz einer Visualisierungsumgebung generieren lässt.

  Damit eine Generierung von Code möglich ist, der der Spezifikation genau entspricht, sollte die gewählte Form der Spezifikation eine formal definierte Semantik haben.

- Die generierte Visualisierungsumgebung kann nicht nur Algorithmen ablaufen lassen, sondern ermöglicht, wie in Kapitel 2 beschrieben, erkundbare Lernräume. Diese Lernräume können in der Laufzeitumgebung in einer Art interaktivem Datenstruktur-Browser erforscht werden, der flüssige Animationen darstellen kann und diese mit verschiedenen Arten der Interaktion koppelt. Dabei muss die Möglichkeit bestehen, den Algorithmus schrittweise ablaufen zu lassen. Wir brauchen dafür eine Abspiel-Schnittstelle, wie sie zum Beispiel von Werkzeugen zum Abspielen von Filmen bekannt ist mit mindestens einer Schaltfläche zum Starten eines Animationsfilms und zum Weiterschalten der Schritte, und natürlich die logische Funktionalität, die die entsprechenden Funktionen zur Laufzeit ausführt.

- Außerdem soll der Lernende aktiv in den Visualisierungsablauf einbezogen werden. Dafür sind Ein- und Ausgabemöglichkeiten und Unterstützung seitens der Verarbeitung notwendig. Ebenso soll die Möglichkeit bestehen, durch den völlig freien, interaktiven Aufruf von Operationen beliebige Instanzen der Datenstruktur aufzubauen und jederzeit Schnittstellenoperationen aufzurufen. Dafür muss es eine graphische Aufruf-Schnittstelle zur Benutzung durch den Lernenden und eine logische Schnittstelle zwischen der graphischen Aufruf-Schnittstelle und der eigentlichen Verarbeitung geben.

- Da hierfür intelligent auf richtige Aktionen wie auch auf Fehler reagiert werden muss, muss es möglich sein, in irgendeiner Weise das Verhalten in Abhängigkeit vom Zustand zu konfigurieren.

Die generierten interaktiven Visualisierungen und Animationen sind nicht als alleinige Lehrveranstaltung gedacht, sondern sollen zur Unterstützung einer bestehenden Veranstaltung dienen. Also soll mit unserem Ansatz weder eine Vorlesung noch eine übliche „Tafel-Übung“ ersetzt werden; schließlich legen es einige der in Kapitel 2 beschriebenen Studien nahe, dass eine Kombination aus verschiedenen Lehrformen am wirkungsvollsten ist. Eine scheinbare Ausnahme ist

nach den ersten in Kapitel 2 beschriebenen Ergebnissen das hypermediale System HalVis. Allerdings gilt es hier zu bedenken, dass der Dozent im Rahmen der Erzeugung der in HalVis angebotenen Materialien den üblichen Vorlesungsstoff zum großen Teil als hypermediale Komponenten in HalVis codieren muss und der Studierende sich durch diesen Stoff auch durcharbeiten muss, so dass die Vorlesung eigentlich mit ins Visualisierungssystem verschoben wird.

Da es den Studenten freigestellt sein soll, ob sie z. B. auf Windows oder Linux arbeiten, ist eine möglichst große Plattformunabhängigkeit wünschenswert. Außerdem müssen der Funktionsumfang und die Bedienung der Laufzeitumgebung so einfach und naheliegend sein, dass der durchschnittliche Student der Ingenieurswissenschaften oder der Informatik nach einer minimalen Einarbeitungszeit gut damit zurechtkommt. Ähnliches gilt in etwas abgeschwächter Form für die Entwicklungsumgebung.

Weitere technische Anforderungen können leichter formuliert werden, wenn die Szenarien, in denen unser Rahmenwerk eingesetzt werden soll, detaillierter herausgearbeitet sind.

Deswegen werden wir im nächsten Abschnitt vier Beispielszenarien skizzieren, die in einer von uns geforderten Visualisierungsumgebung mindestens möglich sein sollen. Diese sollen keine Einschränkung, sondern eine Minimal-Forderung darstellen und sind dafür gedacht, uns zu helfen, eine ganze Reihe weiterer Anforderungen an ein Rahmenwerk für die Implementierung einer solchen Visualisierungsumgebung anzugeben.

3.3. Beispielszenarien

Bei den im Folgenden beschriebenen Szenarien handelt es sich um beispielhaft gewählte Möglichkeiten, wie Lernende mit einer Visualisierungsumgebung für Algorithmen und Datenstrukturen agieren können. Diese Beispiele ließen sich in ein Kontinuum möglicher Szenarien – rangierend von extrem passiver Nutzung bis extrem aktiver Nutzung – einordnen, in dem noch viele weitere Beispiel-Szenarien Platz fänden. Wir werden in Kapitel 5 näher darauf eingehen und auch einige weitere mögliche Szenarien aufzählen. Trotz allem halten wir die getroffene Auswahl für ausreichend, um alle wesentlichen Elemente von den in der Literatur erwähnten Lernszenarien zu ermöglichen (siehe Kapitel 2), und doch für klein genug, um übersichtlich zu bleiben.

Die Reihenfolge der beschriebenen Szenarien bewegt sich von sehr passiver Nutzung zu sehr aktiver Nutzung, wobei jeweils die Nutzung durch die Studierenden gemeint ist.

3.3.1. Vorführen im Rahmen einer Präsenzveranstaltung

Das erste Anwendungsszenario, das wir bezüglich seiner Anforderungen durchspielen wollen, ist das Vorführen im Rahmen einer Präsenzveranstaltung. Es handelt sich hier um eine – vom Standpunkt des Lernenden aus gesehen – rein passive Präsentationsunterstützung in der Präsenzlehre, zum Beispiel um die Schritte beim Aufbau einer Datenstruktur visuell darzustellen.

Hier ist es hilfreich, wenn der Dozent – wie bereits im letzten Abschnitt gefordert – eine Abspiel-Funktionalität zur Verfügung hat, im Rahmen derer er Abläufe schrittweise, aber in einer kontinuierlichen Animation darstellen kann. Da diese Abläufe üblicherweise beim Vorführen einer Präsenzveranstaltung komplexere Abläufe zeigen sollen, braucht der Dozent eine Möglichkeit, Abläufe aus einfacheren Einzelaktionen vorbereiten und ablaufen lassen zu können, zum

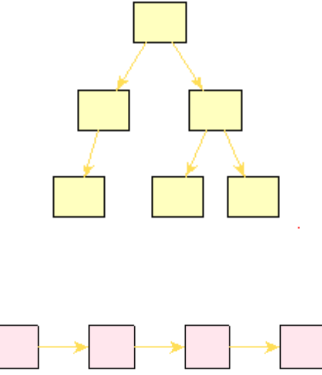


Abbildung 7: Übliches Layout zweier Datenstrukturen. Oben: Baum. Unten: Liste.

Beispiel durch Skripte. Diese Skripte müssen also konfiguriert werden können, eine Schnittstelle zur Ausführungslogik besitzen (für den Anstoß eines Skriptes und für die Interpretation der Befehle der Skriptsprache) und eine Schnittstelle für den interaktiven Aufruf eines Skripts. Diese Animationsfilme sollten an beliebiger Stelle angehalten werden können, um synchrone Erklärungen zu geben.

Ebenso sollte es einfach möglich sein, verschiedene Beispielszenarien im Sinne von Instanzen einer Datenstruktur statisch zu spezifizieren, ohne sie durch Skripte aufbauen lassen zu müssen.

Gerade im Falle von Skripten für vordefinierte Abläufe ist es wichtig, die Ablaufgeschwindigkeit an die Bedürfnisse der Lernenden anpassen zu können. Wir fordern also eine einstellbare Geschwindigkeit, die konfiguriert werden können muss und vom Laufzeitsystem umzusetzen ist.

Da auch komplexere Algorithmen dargestellt werden sollen, ist es notwendig, die Schritte eines Algorithmus vorher in einer sinnvollen Art spezifizieren zu können. Sinnvoll soll heißen, dass die Anzeige weder zu grob noch zu feingranular ist. Das heißt, an wichtigen Stellen des Algorithmus sollen alle Schritte bis ins Detail sichtbar gemacht werden können, während dies an anderen Stellen zu einer Überfrachtung und unnötigen Ausdehnung der Animation führen würde. Es sollen also in irgendeiner Form verschiedene Detaillierungsstufen von Schritten spezifizierbar sein.

Die meisten Datenstrukturen haben eine „übliche“ Darstellung, so werden etwa Bäume auf eine typische, eben baumartige Weise dargestellt (siehe Abb. 7 oben) und Listen auf eine andere (siehe Abb. 7 unten). Obwohl beide Datenstrukturen Graphen sind, sollen sie natürlich nicht einfach unspezifisch „als Graph“ dargestellt werden, sondern in ihrer üblichen Darstellung. Wir brauchen also die Möglichkeit, Datenstrukturen mithilfe möglichst stabiler Layoutalgorithmen in ihrem korrekten Layout darzustellen. „Stabil“ meint in diesem Zusammenhang, dass sich das Layout beim Ändern eines Details einer Datenstruktur, z. B. dem Hinzufügen einer einfachen Komponente, möglichst auch nur in diesem Detail ändert. Wenn das nicht der Fall ist, greift unser Konzept des Führens des psychologischen Fokus trotz flüssiger Animationen nicht mehr, da flüssig animierte, überflüssige Bewegungen an Stellen, wo sich gar nichts ändert, eher verwirren. Es sollten zunächst grundlegende fertige Layoutalgorithmen im System existieren, die das

Layout automatisch berechnen können. Zusätzlich fordern wir, ein Layout für eine Datenstruktur festlegen zu können, ohne diese Layout-Information in die Spezifikation der Datenstruktur selbst aufzunehmen.

Der Hauptgrund ist die möglichst weitgehende Vereinfachung der Spezifikation einer Datenstruktur; denn es gibt viele Fälle von Datenstrukturen, deren Layout ähnlich oder gleich, deren Spezifikation aber unterschiedlich ist. Beispiele sind die verschiedenen Arten von Bäumen (Allgemeine Bäume, Binärbäume, Suchbäume, AVL-Bäume, Rot-Schwarz-Bäume etc.) oder Algorithmen auf allgemeinen Graphen, bei denen es weniger darauf ankommt, wie der Graph dargestellt wird, sondern dass jeder Knoten im Zuge der Veränderungen des Algorithmus an seinem Platz bleibt; das ist ein Extrembeispiel von stabilem Layout.

In solchen Fällen bietet es sich durch die logische Trennung von Spezifikation und Darstellung an, den Layoutalgorithmus auch von der Spezifikation der Datenstruktur und des Algorithmus zu trennen; er wird also nach der Generierung der Instanz der Visualisierungsumgebung durch Konfiguration eingestellt. Das erleichtert weiter die Arbeit des Dozenten bei der Erzeugung interaktiver Lernszenarien für neue Datenstrukturen, wenn bereits für ähnliche Datenstrukturen Lernszenarien generiert worden sind.

Da es nie möglich ist, jedes denkbare Layout schon vorbereitet im System darstellen zu können, muss es zusätzlich einfach genug sein, eigene Layout-Algorithmen ins System einzubauen. Diese Notwendigkeit ist ein weiterer Grund, die logische Spezifikation von der Layout-Spezifikation zu trennen.

Beim Animieren einer Datenstruktur oder eines Algorithmus gibt es üblicherweise neben der strukturellen Anzeige der Elemente und ihrer Verbindungen zueinander (bei Graphen wären das zum Beispiel die Knoten und Kanten) auch die Beschriftung der genannten Komponenten der Animation. Um alle relevanten Informationen visualisieren zu können, ist es oft notwendig, den Zustand der Datenstruktur als eine Menge von Eigenschaften ihrer Elemente darstellen zu können. Wir fordern also die Möglichkeit, komplexe Zustands-Informationen detailliert anzeigen zu können und die entsprechenden Einstellungen konfigurieren zu können.

Zusätzlich sollten Fehlersituationen oder besondere Werte o.Ä. durch visuelle Markierung kenntlich gemacht werden können, zum Beispiel durch besondere Farben oder im Falle von Linien auch durch besondere Arten der Strichelung für fehlerhafte bzw. besondere Teile der Datenstruktur. Wir fordern also die Möglichkeit, Farbschemata und ggf. Stilschemata konfigurieren und darstellen zu können. Das funktioniert natürlich nur, wenn die Verarbeitungslogik die dafür notwendigen Daten besitzt und nutzt.

Für ein und dieselbe Datenstruktur gibt es bei Ausführung bestimmter Schritte oder Operationen oder bei Ausführung eines zu lehrenden Algorithmus oft mehrere hervorhebenswerte Stellen der Datenstruktur, etwa die gerade bearbeitete, diejenige an der die Invariante temporär verletzt ist oder eine aus einem anderen Grund interessante. Gerade beim Vorführen in der Präsenzveranstaltung bietet es sich an, diese Stellen nicht alle gleichzeitig, zum Beispiel durch unterschiedliche Farben, hervorzuheben, sondern in mehreren Durchläufen einzeln vorzuführen. Deswegen reicht es nicht aus, Farbschemata in Abhängigkeit vom Zustand der Datenstruktur zur Verfügung zu haben, sondern diese müssen auch einfach und unkompliziert zu wechseln sein.

3.3.2. Passives Betrachten vorgegebener Abläufe

Beim passiven Betrachten vorgegebener Abläufe sitzt der Studierende selbst vor der Oberfläche der Laufzeitumgebung der vorbereiteten Visualisierungsinstanz und hat die Möglichkeit, in seinem eigenen Rhythmus Abläufe eines Algorithmus oder Abfolgen von Operationen auf einer Datenstruktur nachvollziehen.

Im Unterschied zum Vorführen in der Präsenzveranstaltung ist es hier besonders wichtig, auch die Möglichkeit zur Verfügung zu stellen, die zu lernenden Abläufe auf verschiedenen Datenmengen oder verschiedenen Instanzen einer komplexeren Datenstruktur beobachten zu können. Um zu vermeiden, dass der Dozent hier prinzipiell mehrere Sätze von Daten von Hand vorbereiten muss, fordern wir die Möglichkeit, zufällige Datenmengen und zufällige Instanzen komplexer Datenstrukturen generieren zu können. Das ist kein Widerspruch zu vorgegebenen Abläufen, wenn sichergestellt wird, dass die möglichen Abläufe einem klarem Ablaufschema entsprechen, das auf jeder der generierbaren Datenmengen sinnvoll ist. Eine Möglichkeit ist hier der Ablauf eines festen Algorithmus, der auf jeder Datenstruktur des gegebenen Typs funktioniert. Eine andere Möglichkeit bei einer unflexibleren Form von Ablauf ist die Generierung nur dazu passender Instanzen der Datenstruktur bzw. passender Datenmengen.

Zusätzlich zu zufälligen Datenmengen sollte es möglich sein, benutzerdefinierte Datenmengen eingeben zu können bzw. benutzerdefinierte Strukturen aufbauen zu können, die dann natürlich von der Laufzeitlogik verarbeitet werden können sollten. Sonst könnte der Fall entstehen, dass der Lernende ein für ihn besonders interessantes Beispiel auf herkömmliche Art, zum Beispiel mit Stift und Papier, nachstellen muss, nachdem er auf dem Rechner einige andere Beispiele beobachtet hat. Die Gefahr wäre dann wohl, dass dieses Nachstellen, was in jedem Fall dem Lerneffekt zuträglich wäre, aus Frustration unterbleibt.

Außerdem ist es, wie wir in Kapitel 2 gesehen haben, für den Lerneffekt beim Selbststudium besonders hilfreich, wenn der Lernende Abläufe in seinem eigenen Rhythmus vor- aber auch zurückspulen kann. Dafür muss eine Aufruf-Schnittstelle bestehen. Im System wird für das vor- und zurückspulen die Fähigkeit des 'undo' / 'redo' notwendig sein.

Noch wichtiger als beim Vorführen durch einen Dozenten ist bei der individuellen Nutzung eines Studierenden die Behandlung von Fehlersituationen, die ja auch in vordefinierten Abläufen durchaus gewollt sein können. Die Nutzung von Farben zur Markierung besonderer oder fehlerhafter Details wurde im letzten Abschnitt erwähnt.

Während beim Vorführen eines Ablaufs der Dozent notfalls auf besondere oder fehlerhafte Situationen hinweisen kann, ist es bei der individuellen Nutzung durch Lernende wichtig, dass zustandsabhängig unmissverständliche Meldungen ausgegeben werden können, was nur möglich ist, wenn die Logik der Laufzeitumgebung die notwendigen Informationen dafür hat und kontextspezifisch nutzt. Da diese individuell von der jeweiligen Datenstruktur und auftretenden Situationen abhängen, müssen sie – möglichst entkoppelt von der Spezifikation der Datenstruktur und ihrer Operationen selbst – in irgendeiner Form spezifizierbar sein.

Ein wichtiges Beispiel für Fehlersituationen im Zusammenhang mit Datenstrukturen sind verletzte Invarianten. Zunächst einmal muss der Dozent in der Lage sein, Invarianten überhaupt zu spezifizieren. Und schließlich bleibt die Entscheidung zu fällen und zu spezifizieren, ob die Verletzung bestimmter Invarianten der Datenstruktur temporär möglich sein soll. Das ist besonders bei explorativen Lernszenarien (siehe weiter unten) eine wichtige Möglichkeit, da die

Studierenden auf diese Weise bestimmte Fehler machen können und durch die entstehenden Rückmeldungen des Systems ein zusätzlicher Lerneffekt zu erwarten ist – im Gegensatz zu einem System, das jede Verletzung von Invarianten von vorneherein verbietet. Aber auch schon beim Ansehen vordefinierter Abläufe spielt die Definition der Invarianten eine Rolle, wenn z. B. bewusst Fehlersituationen dargestellt werden sollen, wo das System die Verletzung einer Invariante nicht erlaubt und somit mit einer lernförderlichen Fehlermeldung quittiert. Im Falle temporär verletzbarer Invarianten muss konfiguriert werden können, wie eine solche Verletzung als Fehlersituation – z. B. farblich – dargestellt werden soll.

3.3.3. Interaktives Erforschen und Aufgaben-Lösen

Beim interaktiven Erforschen und Aufgaben-Lösen sollen die Studierenden nicht nur einfache Formen der Interaktivität zur Verfügung haben wie Vor- und Zurückspulen vorgegebener Abläufe, sondern sich in einem vorbereiteten Lernraum frei bewegen können.

Das kann zum Beispiel so aussehen, dass eine Datenstruktur mit vorgegebenen Konstruktoren frei aufgebaut werden kann und Schnittstellenoperationen der Datenstruktur in nicht determinierter Reihenfolge vom Studierenden getestet werden können, wobei idealerweise Situationen entstehen, in denen durch Feedback ein Lerneffekt stattfindet.

Es müssen also zusätzlich zum bisher Geforderten Operationen auf der Datenstruktur spezifiziert werden können, in der Laufzeitumgebung – zum Beispiel über Menüpunkte – aufgerufen werden können und sich in der Laufzeitumgebung auswirken. Als Feedback einer solchen Aktion kommt je nach Entscheidung des Dozenten eine Fehlermeldung in Frage, die ausdrückt, dass diese Aktion in dieser Situation nicht erlaubt ist, oder das Durchführen der Operation mit entsprechender Darstellung des Zustandes der Datenstruktur. Dabei kann durch geeignete Farbgebung oder Musterung, wie es bereits in Abschnitt 3.3.2 beschrieben wurde, auf einen besonderen Zustand hingewiesen werden, oder es kann eine „gutartige“ Meldung im Sinne einer Ermunterung ausgegeben werden. Im Falle einer guten Meldung muss diese analog zu Fehlermeldungen konfiguriert, von der Programmierlogik zum geeigneten Zeitpunkt und im korrekten Kontext aufgerufen und angezeigt werden.

Wie im Abschnitt über das Vorführen beschrieben, setzen wir stabile, automatisch arbeitende Layoutalgorithmen voraus. Da wir aber die positiven Auswirkungen der Interaktivität und des Explorierens so hoch einschätzen, soll es auch möglich sein, ein Layout manuell zu ändern. Das kann bei einer graphbasierten Datenstruktur zum Beispiel so aussehen, dass die Datenstruktur vom System zunächst in einem schönen Standard-Layout dargestellt wird, der Lernende aber beliebig Knoten verschieben darf. Somit sollen sowohl die Oberfläche als auch die Konfigurationseinstellungen die manuelle Anpassung des Layouts unterstützen.

Dieses Erforschen kann entweder völlig frei angeboten werden oder anhand eines Fahrplans, der die Studierenden durch eine Reihe zu lösender Aufgaben führt, aber dabei genügend Spielraum für eigene Experimente lässt. Das heißt, wir fordern eine Art von Fahrplan, der konzeptionell fähig ist, den Lernenden durch ein offenes Lernszenario zu führen. Dieser Fahrplan muss in irgendeiner Art für den Lernenden sichtbar und durch die Programmierlogik abgedeckt sein. Außerdem muss er unabhängig von der Spezifikation der Datenstruktur angepasst werden können. Auf diese Weise ist ein Mittelweg möglich zwischen zu wenig Freiheit mit einer dementsprechenden Behinderung des Lerneffekts und zu viel Freiheit mit der Gefahr, dass sich

die Studierenden in einem zu kleinen Teil des Lernraumes verlieren.

Im Rahmen eines solchen Fahrplans soll es z. B. auch möglich sein, dem Lernenden Fragen zum aktuellen Zustand der Datenstruktur oder zu weiteren Schritten des Algorithmus zu stellen und geeignet auf die Antwort zu reagieren.

Es muss also neben der Möglichkeit, solche Fragen z. B. über Meldungen zu stellen, auch die Funktionalität geben, Antworten auf solche Fragen entgegenzunehmen und mit weiteren Meldungen darauf zu reagieren. Da die Informationen hierzu im System vorhanden sein müssen (zum Beispiel in Form der oben genannten Eigenschaften der Teile der Datenstruktur), muss es die Möglichkeit geben, Informationen bewusst zu verstecken, die vielleicht beim Vorführen in der Präsenzveranstaltung gezeigt werden würden. Dieses Verstecken muss konfigurierbar sein und von der Oberfläche umgesetzt werden. Diese Fähigkeit dient zusätzlich der Übersichtlichkeit, da beim Vorführen üblicherweise kleinere Beispiele einer Datenstruktur gewählt werden, als sie beim interaktiven Lernen möglich und sinnvoll sind.

Weitere wichtige Punkte wie die Nutzung flüssiger Animationen und die Möglichkeit, Fehlerzustände durch Farbschemata darzustellen und Nachrichten vom System zu erhalten, wurden bereits erwähnt.

Möglicherweise ist es sinnvoll, neue Formen der Interaktion zu ermöglichen, die in herkömmlichen Benutzeroberflächen nicht standardmäßig angeboten werden. Als Beispiel nennen wir das Rotieren eines Knotens um einen anderen mit der Maus, um die bei AVL-Bäumen vorhandenen Rotationsoperationen anschaulich auslösen zu können. Das könnte z. B. geschehen, indem beim Klicken auf einen dafür vorgesehenen Knoten, um den also rotiert werden darf, solange die Maustaste gedrückt bleibt, ein halbkreisförmiger Transitionspfad graphisch dargestellt wird, auf dem sich der zu rotierende zweite Knoten bewegen lässt. Eine solche Interaktion müsste von der Oberfläche angeboten werden, konfiguriert und logisch verarbeitet werden können.

3.3.4. Visuelles Debugging eigener Programme der Studenten

In diesem Szenario sollen die Studenten eigene Programme in einer üblichen Programmiersprache (Java, C++, Ada, . . .) entwerfen, in denen sie das grundsätzliche Verhalten einer Datenstruktur oder eines Algorithmus implementieren. Neben den eigentlichen Ausgaben ihres Programms sollen sie dann den Aufbau und die Manipulationen ihres Programms an der Datenstruktur in einem zusätzlichen Browser-Fenster visualisiert sehen. Diese Visualisierung soll zusammen mit dem Ablauf des Programms schrittweise geschehen können. Streng genommen handelt es sich also nicht ganz um ein visuelles Debugging, da z. B. der Wert von Programmvariablen nicht angezeigt werden soll. Ebenso können nur diejenigen Attribute der Datenstruktur gesehen werden, die gemäß der Spezifikation dafür vorgesehen sind, andere werden ja wie beschrieben bewusst versteckt. Genauer sollte man es vielleicht ferngesteuerte, synchronisierte Animation und Datenstrukturvisualisierung nennen. Andererseits geht es im visuellen Debugging um die

- Reproduktion
- Lokalisierung
- und Behebung

von Fehlern. Der Programmierer stellt dabei typischerweise eine Hypothese darüber auf, an welcher Stelle in seinem Programm was fehlschlägt. Diese Absicht kann durch zu viel und zu

detailliert angebotene Information auch erschwert werden. Deshalb sollte dafür gesorgt werden, dass nur relativ grobe Schritte visualisiert werden, und die Verletzung wichtiger Invarianten gleich graphisch (z. B. farblich) markiert wird. Da deshalb der Begriff „visuelles Debugging“ die Intention des hier beschriebenen Szenarios gut trifft, werden wir ihn an manchen Stellen dieser Arbeit synonym zu der Bezeichnung „ferngesteuerte Animation“ verwenden.

Dazu wird es notwendig sein, eine Schnittstelle für den Aufruf von Operationen auf der Datenstruktur anzubieten, die von einem externen Programm der Studierenden aufgerufen werden kann, so dass dieses von der Programmlogik des Laufzeitsystems genauso verarbeitet wird wie ein interaktiver Aufruf der jeweiligen Operation. Diese Schnittstelle muss also auch von mehreren Programmiersprachen aus verfügbar sein.

Da die Visualisierung des Programm-Verhaltens auch die Visualisierung von Fehlern an bestimmten Stellen des Programms mit einschließt, ist es notwendig, parallel zur Visualisierung des Programms auch den ausgeführten Programmtext sehen zu können. Dieser soll also angezeigt werden und synchronisiert sein mit dem Ablauf der Visualisierung. Dabei ist es notwendig, dass sich mögliche Programmierfehler und mögliche Inkonsistenzen der Animation exakt entsprechen.

Ein Nebengewinn durch die Forderung dieser Fernsteuerung ist die oft geforderte Eigenschaft von Algorithmenanimations-Systemen, den Bau eigener Animationen zu unterstützen, hier sogar in mehreren potentiellen Programmiersprachen.

3.4. Kategorisierung der Anforderungen

Der Übersichtlichkeit halber werden wir nun die in diesem Kapitel gesammelten Anforderungen in Kategorien unterteilen. Als Kategorien wählen wir

- **(SP):** Anforderungen an die Spezifikationssprache und die Mächtigkeit der Entwicklungsumgebung, diese weiter zu verarbeiten.
- **(LO):** Anforderungen an die logische Verarbeitung von Datenstrukturen in der Laufzeitumgebung.
- **(O):** Anforderungen an die Oberfläche der Laufzeitumgebung inklusive interaktiven und Layout-Fähigkeiten.
- **(K):** Konfigurationsmöglichkeiten der Laufzeitumgebung.
- **(SO):** Sonstige Anforderungen, zum Beispiel Anforderungen an Schnittstellen zum Betriebssystem, einfache Installation, freie Verfügbarkeit.

Die Unterscheidung zwischen **(SP)** und **(K)** rührt daher, dass wir, wie bereits bei der Beschreibung des Grundszenarios klargestellt, strikt unterscheiden zwischen der Spezifikation der Datenstruktur und des Algorithmus selbst auf der einen Seite und der generierten Instanz der Visualisierungsumgebung auf der anderen Seite (siehe Abb. 6). Diejenigen Einstellungen, die nach der Generierung der Instanz der Lernumgebung getätigt werden können, nennen wir Konfiguration, sogar wenn es in Ausnahmefällen darum geht, zusätzlichen Code zu schreiben, zum Beispiel beim Einbinden eines neuen Layout-Algorithmus. Die Gründe für diese Trennung bestehen aus einer Vereinfachung und höheren Wiederverwendbarkeit der Spezifikationen, womit insbesondere dem genannten Problem des Mehrfachaufwands des Dozenten beim Erstellen von

Lerneinheiten für ähnliche oder verwandte Datenstrukturen begegnet werden soll.

Hier besteht eine Ähnlichkeit zum bekannten 'model-view-controller'-Muster (MVC), in dem ja auch ein Datenmodell (*model*) getrennt wird von der Programmierlogik einer Applikation (*controller*) und der Anzeige der Daten (*view*), die ohne Änderung des Modells variabel bzw. auswechselbar ist. In dieser Betrachtungsweise entspricht die Spezifikation der Datenstruktur dem *model*, der operative Code einer Instanz der Visualisierungsumgebung dem *controller* und die entstehende Visualisierung dem *view*. Die Konfiguration einer bestehenden Instanz wirkt sich über die *controller*-Funktionalität auch wieder auf das Verhalten (*controller*) und die Anzeige (*view*) aus.

Zusätzlich zu der systematischen Analyse der Szenarien im letzten Abschnitt fand ein Brainstorming-Prozess im Hinblick auf weitere sinnvolle Anforderungen an eine Visualisierungsumgebung, die unser Grundszenario ermöglichen kann, statt. Auf der Grundlage dieser beiden Quellen ergibt sich folgende Liste von Anforderungen, die nach den vorgestellten Kategorien klassifiziert ist:

- **(SP)**:

  1. Die Spezifikationssprache besitzt eine formale Semantik (das ist eine Voraussetzung für die Codegenerierung).
  2. Es gibt eine Werkzeugunterstützung für die Erstellung der Spezifikation und die Codegenerierung.
  3. Die Spezifikationssprache ist natürlich, nach Möglichkeit auch bekannt und einfach.
  4. Die Spezifikationen sind einfach erweiterbar für kleine Änderungen an der Datenstruktur.
  5. Schnittstellenoperationen sind einfach zu spezifizieren bzw. nachzuspezifizieren.
  6. Invarianten sind einfach zu spezifizieren.
  7. Verbote in Abhängigkeit vom Zustand der Datenstruktur, z. B. verletzten Invarianten, sind einfach zu spezifizieren.
  8. Das Spezifizieren neuer Beispielszenarien ist einfach.
  9. Die Schritte eines Algorithmus sind in verschiedenen Granularitäten einfach zu spezifizieren.

- **(LO)**:

  1. Das Verhalten in Abhängigkeit vom Zustand der Datenstruktur ist steuerbar.
  2. Ausgabe- (Fehler-) Meldungen als Reaktionen auf Interaktionen sind möglich.
  3. Vom Nutzer eingegebene Datenmengen oder Instanzen einer Datenstruktur können zur Laufzeit verarbeitet werden.
  4. Es gibt eine interne Schnittstelle für den Aufruf von Operationen auf der Datenstruktur.
  5. Eine automatische Überprüfung von Invarianten ist möglich (z. B. bei interaktiven Änderungen durch den Benutzer).
  6. Gerade bearbeitete, inkonsistente, wichtige Teile der Datenstruktur können (für die Darstellung als Farb- oder Stilschemata) einfach identifiziert werden.
  7. Die Laufzeitlogik ist fähig zu 'undo' / 'redo'.

8. Ein schrittweises Weiterschalten der Zustände der Datenstruktur ist möglich.
9. Es gibt eine Schnittstelle für eine beispielspezifische Taktung (nächste Schritte der Animation) von außen.
10. Es gibt einen logischen Mechanismus, um Vorhersagen in einem bestimmten Kontext verlangen und testen zu können.
11. Eine parallele Anzeige der zum aktiven Teil der Datenstruktur oder des Algorithmus passenden Programmteile ist synchronisierbar mit der Visualisierung.
12. Zufällige Datenmengen, zufällige Instanzen einer Datenstruktur und zufällige Szenarien können zur Laufzeit generiert werden.
13. Das Laden und Speichern verschiedener Beispielszenarien ist möglich.
14. Die Animation von Algorithmen kann zwischen verschiedenen Detaillierungsstufen wechseln.
15. Eine Kenntnis über die Struktur von Datenstrukturen ist codierbar, etwa die Struktur von Graphen oder Bäumen.
16. Es gibt eine Schnittstelle für den Start von Skripten.
17. Es gibt eine Schnittstelle für den Aufruf von Befehlen in einer Skriptsprache.
18. Verschiedene Geschwindigkeiten sind für die Animation möglich.
19. Ein Fahrplan kann von der Laufzeitumgebung überwacht werden (ggf. mit Aufruf von Meldungen).
20. Es gibt einen Mechanismus für das Wechseln von Farbschemata zur Laufzeit.
21. Es sind Strukturen für die Verarbeitung neuer Formen der Interaktion vorhanden.
22. Ein Protokoll aufgerufener Operationen kann angezeigt werden.

- **(O)**:

  1. Die entstehenden Animationen sind flüssig (kontinuierlich).
  2. Es gibt eine Schnittstelle für die Eingabe von Vorhersagen der Studierenden.
  3. Es gibt eine Schnittstelle für Fragen / Rückmeldungen an Studierende.
  4. Es gibt eine Möglichkeit, Schnittstellenoperationen aufzurufen.
  5. Explorative Lernszenarien mit genügend Wahlmöglichkeiten des Benutzers (zum Beispiel Aufruf auch „falscher“ Schnittstellenoperationen) sind möglich.
  6. Das interaktive Erzeugen eigener Beispiele und eigener Datenmengen ist einfach.
  7. Vordefinierte oder zur Laufzeit definierte Farb- und Stilschemata können dargestellt werden.
  8. Es gibt einen Graphbrowser mit Layout (Darstellung erzeugter Strukturen).
  9. Graphartige Datenstrukturen wie Listen, Bäume, Graphen können dargestellt werden.
  10. Das Einbinden von Layoutalgorithmen mit automatischer Berechnung eines stabilen Layouts ist möglich.
  11. Eine Abspiel-Schnittstelle ist vorhanden oder zumindest einfach zu implementieren.
  12. Animationsfilme können an beliebiger Stelle angehalten werden.
  13. Mit dem Player können Abläufe vorwärts *und* rückwärts abgespielt werden.
  14. Die Animation statischer Diagramme (Farben ändern etc.) ist möglich.
  15. Es ist möglich, die Anzeige komplexer Zustände zur Laufzeit zu konfigurieren, ein Beispiel hierfür ist das Verstecken bestimmter vorhandener Informationen.

16. Eine parallele Anzeige der zum aktiven Teil der Datenstruktur passenden Programmteile (beim Debugging) ist – synchronisiert mit dem Animationsablauf – möglich.
17. Die dargestellten Teile einer Datenstruktur können mit HTML-Links (verschiedenen in Abhängigkeit vom internen Zustand) versehen werden, die z. B. auf erklärende Texte verweisen.
18. Es kann mehrere Fenster mit verschiedenen Sichten geben.
19. Es gibt eine Schnittstelle für den Aufruf von Skripten.
20. Die Bedienung der Laufzeitumgebung ist intuitiv.
21. Das automatisch berechnete Layout kann manuell geändert werden.
22. Ein vorhandener Fahrplan für eine interaktive Übung in einem explorativen Szenario kann dargestellt werden.
23. Eine parallele Anzeige von zum aktiven Teil der Datenstruktur passendem Pseudocode ist – synchronisiert mit dem Animationsablauf – möglich.
24. Mischformen aus graphartigen Datenstrukturen können dargestellt werden.
25. Nicht-graphartige Datenstrukturen wie Felder oder Matrizen können dargestellt werden.
26. Eine asynchrone Anzeige erläuternder HTML-Seiten oder Filme ist als Hilfesystem möglich.
27. Der Benutzer kann zoomen.
28. Die Laufzeitumgebung kann als Applet auf einer Website eingesetzt werden.
29. Sound ist – synchronisiert mit dem Ablauf der Animation – möglich.
30. Es sind neue Formen der Interaktion vorgesehen (z. B. Rotieren eines Graphknotens um einen anderen Graphknoten mit der Maus).

- **(K)**:

1. Farbschemata sind in Abhängigkeit vom Zustand, von Invarianten, von aktuellen Geschehnissen usw. einfach konfigurierbar.
2. Meldungen sind in Abhängigkeit vom Zustand konfigurierbar.
3. Die Anzeige komplexer Zustände der Datenstruktur ist in verschiedenen Details konfigurierbar. Dazu gehört auch das Verstecken bestimmter Informationen!
4. Layoutalgorithmen sind unabhängig von Datenstruktur und Algorithmus konfigurierbar.
5. Das Konfigurieren häufig genutzter Layoutalgorithmen wird dadurch unterstützt, dass wichtige Layoutalgorithmen bereits im System vorhanden sind.
6. Animationen sind über eine Datenstruktur-spezifische Bibliothek vorhandener Basisoperationen in einer Skriptsprache erstellbar.
7. Die in der Skriptsprache erstellten Animationen sind einfach auszuwechseln.
8. Verschiedene Geschwindigkeiten sind für die Animation konfigurierbar.
9. Stilschemata (z. B. für Kanten) sind konfigurierbar.
10. Die Umstellung zwischen erzwungenem, automatischen Layout und der Möglichkeit der manuellen Veränderung des Layouts ist konfigurierbar.
11. Es kann ein Fahrplan für interaktive Übungen konfiguriert werden.
12. Neue Formen der Interaktion sind konfigurierbar.

- **(SO)**:

    1. Es gibt eine Schnittstelle für den externen Aufruf von Operationen auf der Datenstruktur, die von verschiedenen Programmiersprachen aus nutzbar ist.
    2. Das System ist stabil.
    3. Zumindest die Laufzeitumgebung ist möglichst plattformunabhängig.
    4. Es kann auf einen Netzwerkanschluss verzichtet, also rein lokal gearbeitet werden.
    5. Die Komponenten des Systems sind frei verfügbar.
    6. Das Werkzeug ist unabhängig von „evtl. bald nicht mehr verfügbaren Bibliotheken".
    7. Das System ist einfach zu installieren.
    8. Man kann den Zustand der Animation speichern.

Die Punkte jeder Kategorie dieser Liste sind der Wichtigkeit nach absteigend sortiert. Dabei ist die relative Wichtigkeit eines Punktes natürlich ein subjektiver Wert. In unserem Fall flossen die bis zu diesem Punkt der Arbeit beschriebenen Erkenntnisse bisheriger Studien und Forschungen über die Wichtigkeit der jeweiligen Eigenschaft genauso ein wie eigene Erfahrungen und die Wichtigkeit im Hinblick auf die Umsetzbarkeit einer Visualisierungsumgebung, die alle beschriebenen Szenarien unterstützen kann.

Damit steht eine Liste von Anforderunen an ein Rahmenwerk zur Erstellung von interaktiven Lerneinheiten zur Lehre von Algorithmen und Datenstrukturen zur Verfügung, die sowohl didaktische als auch für die Anpassbarkeit und Wiederverwendbarkeit der entstehenden Lerneinheiten wichtige Grunderfordernisse berücksichtigt.

In diesem Kapitel wurde ein Einsatzszenario zusammen mit einer ausführlichen Liste konkreter Anforderungen an eine lernwirksame Visualisierungsumgebung entwickelt. Wir werden im nächsten Kapitel eine Untersuchung der Nutzbarkeit einer Reihe von Werkzeugen und Paradigmen für unseren Ansatz beschreiben und dabei sehen, dass der Anforderungskatalog in seiner jetzigen Form von keinem der untersuchten Ansätze zufriedenstellend erfüllt werden kann.

Deshalb werden wir in Kapitel 4 zunächst eine wesentliche Teilmenge von Anforderungen aus unserer Liste identifizieren, die unserer Meinung nach unbedingt erforderlich sind und weiterhin unser neues Grundkonzept ermöglichen. Aufbauend auf dieser kondensierten Liste werden wir den eigentlichen Werkzeug-Vergleich durchführen.

In Kapitel 10 werden wir nochmal auf die komplette Anforderungsliste zurückkommen und sie als Bewertungsschema für den in dieser Arbeit vorgestellten Ansatz nutzen.

4. Related Work

In diesem Kapitel geht es darum, vorhandene Paradigmen und Werkzeuge in den Bereichen Datenstruktur-Spezifikation, Algorithmen-Spezifikation und verschiedenen Konzepten der Visualisierung auf ihre Eignung für unser im letzten Kapitel vorgestelltes Grundszenario hin zu überprüfen. Dabei werden die Werkzeuge zunächst alle kurz vorgestellt und in einem zweiten Schritt ihre jeweilige Eignung im Hinblick auf eine verdichtete, wesentliche, im Folgenden beschriebene Teilmenge der im letzten Kapitel hergeleiteten Liste von Anforderungen beurteilt.

Bei der Auswahl der Paradigmen der untersuchten Ansätze und Werkzeuge wurde versucht, eine möglichst große Bandbreite durch den Einsatz dreier Quellen zu erzielen. Dabei handelt es sich um

1. diejenigen Paradigmen, die in der Literatur über Algorithmen-Visualisierungssysteme als generative Ansätze bekannt sind (siehe z. B. [SDBP98]),
2. alle auf der Visualisierungs-Website [Sof04] aufgeführten, technisch in Frage kommenden Systeme und
3. bekannte Paradigmen und Werkzeuge für Spezifikation und Visualisierung, wobei dieser Begriff naturgemäß unscharf ist.

Innerhalb jeder im Folgenden beschriebenen Kategorie wurden ein bis vier bekannte Vertreter ausgewählt. Im Falle mehrerer Werkzeuge wurde versucht, auch innerhalb der jeweiligen Kategorie auf eine gewisse Bandbreite in Konzept und Umsetzung zu achten.

Wie in Kapitel 2 erwähnt, wurde bereits im Jahr 1998 allein die Zahl der bekannten Algorithmenanimations-Systeme auf über 150 geschätzt [PBS98]. Es musste also eine Vorauswahl getroffen werden; die im Folgenden nicht erwähnten Werkzeuge und Paradigmen kommen entweder unserer Meinung nach konzeptionell für unser Grundszenario überhaupt nicht in Frage oder sind durch die beschriebenen Paradigmen und Werkzeuge ausreichend repräsentiert. Abgesehen davon ist an dieser Stelle ein Anspruch auf Vollständigkeit weder unser Ziel noch überhaupt möglich.

4.1. Vorstellung der genutzten Paradigmen

Wir betrachten die im Folgenden beschriebenen Werkzeuge von zwei verschiedenen Blickwinkeln aus, nämlich

- in Bezug auf ihre Brauchbarkeit für die Spezifikation einer Datenstruktur und eines Algorithmus und
- in Bezug auf ihre Brauchbarkeit für Animation bzw. Visualisierung.

Ungeachtet der Einordnung in die im Folgenden beschriebenen Kategorien ist uns bewusst, dass einige Werkzeuge beide Teilaufgaben abdecken. Beispiele hierfür sind die weiter unten (in Abschnitt 4.4) vorgestellten deklarativen Algorithmenanimations-Systeme Pavane und Leonardo. Trotzdem werden wir sehen, dass es kein Werkzeug gibt, das alleine in der Lage ist, auf befriedigende Weise unser Grundszenario zu unterstützen, und dass es sogar schwierig ist, eine aussichtsreiche Kombination von Werkzeugen dafür zu finden.

Bei den Spezifikationsansätzen werden wir zwischen Text-basierten und visuellen Spezifikationsarten unterscheiden und jeweils einige bekannte Beispiele betrachten.

Bei den Visualisierungsansätzen gibt es kein so naheliegendes Klassifikationsschema, obwohl für Teilbereiche Ansätze vorhanden sind; so unterscheidet [SDBP98] folgende Arten von Algorithmenanimations-Systemen:

- Systeme mit 'interesting events', in denen also zur Vermeidung einer zu großen Anzahl von Einzelschritten bei der Animation Annotationen in den Programmtext oder in die Animationsdeklaration eingefügt werden, die besonders interessante Stellen markieren.
- Systeme mit deklarativer Spezifikation, in denen zum Beispiel durch prädikatenlogische Ausdrücke Bedingungen angegeben werden, die die Animation zu einem bestimmten Zeitpunkt erfüllen muss.
- Systeme, die direkte Manipulation nutzen, in denen also in einem gewissen Kontext Objekte vom Benutzer in einem Fenster manipuliert werden können, was sich unmittelbar auf das spätere Verhalten des Systems auswirkt.

[Röß02] fügt diesen Möglichkeiten noch einige weitere hinzu, von denen nur die zusätzliche Unterscheidung in API-basierte Systeme, skriptbasierte Systeme und Systeme mit Code-Interpretation für uns interessant ist.

Es gibt auch andere Klassifizierungsversuche, wie etwa die in [PH01] vorgeschlagenen abstrakten Typen I, II, III etc. gemäß dem Abstraktionsgrad der Spezifikation. Da wir auch Werkzeuge und Paradigmen betrachten wollen, die nicht unmittelbar etwas mit herkömmlicher Algorithmenanimation zu tun haben, und die Einteilung von [PH01] extrem unanschaulich ist, nutzen wir die Einteilung von [SDBP98] und [Röß02]. Zusätzlich fügen wir eigene Kategorien, die darin nicht enthalten sind, hinzu, und ordnen Systeme mit 'interesting events' der Einfachheit halber in die API-basierten Systeme ein. Der letzte Punkt ist deswegen sinnvoll, weil die im Programm bzw. in der Spezifikation annotierten 'interesting events' gleichgesetzt werden können mit dem Aufruf externer Funktionen einer vordefinierten Schnittstelle, nämlich derjenigen der Funktionsschnittstelle des zugrundeliegenden Animationssystems.

Aus diesen Überlegungen ergeben sich folgende zu untersuchende Kategorien für die Visualisierung im Algorithmen-Umfeld:

- Nicht-spezialisierte Animationssysteme für Algorithmen.
- Elektronische Bücher (EBooks) für die Lehre von Algorithmen und Datenstrukturen.
- Algorithmenanimation durch Code-Interpretation.
- API-basierte Graphenbibliotheken.
- Erzeugung von Animationen durch direkte Manipulation.
- Deklarative Systeme für die Animation von Algorithmen.
- Skriptbasierte Algorithmenanimation.
- Visuelle, regelbasierte Entwicklungsumgebungen.
- Icon-basierte Simulationssprachen.
- Entwicklungsumgebungen für graphische Oberflächen oder Benutzerschnittstellen.
- Sonstige generative Ansätze für die Erstellung von Animationen.

Diese Kategorien überlappen sich teilweise und wären somit ungeeignet als Systematik zur sauberen Erfassung aller existierenden Ansätze der Algorithmenanimation. Für unsere Zwecke können wir diese Einteilung allerdings trotzdem nutzen, denn es geht an dieser Stelle ja primär darum, eine möglichst umfassende Abdeckung im Bereich der Algorithmenanimation zu erzielen. Mit dieser groben Einteilung können wir dann pro Kategorie einige Ansätze vorstellen, die im ungünstigsten Fall auch in einer anderen Kategorie vorgestellt hätten werden können. Aus diesen Gründen verzichten wir auf die Einführung einer neuen, sicherlich schwerer lesbaren und für unsere Zwecke überflüssigen Systematik.

Im Rahmen unserer Evaluation wurde es schnell klar, dass kein Werkzeug auch nur annähernd alle im letzten Kapitel als wünschenswert für unseren Ansatz genannten Anforderungen erfüllen kann. Wir werden deshalb bei der folgenden Beschreibung nur eine Untermenge der Anforderungen betrachten, und zwar die nach unserem Ermessen wirklich entscheidenden. Da die Listen der Anforderungen in Kapitel 3 innerhalb der Kategorien absteigend nach Wichtigkeit sortiert waren, handelt es sich um die jeweils obersten Anforderungen jeder Kategorie, die kategorienübergreifend zu griffigen Bezeichnungen zusammengefasst werden. Bei der folgenden Evaluation wird es sich zeigen, dass es eine Kombination zweier Werkzeuge gibt, die einen vielversprechenden Eindruck bezogen auf diese verdichtete Liste macht und die konsequenterweise für die Implementierung unseres Ansatzes genutzt wird.

4.2. Verdichtung der Anforderungsliste

Wir legen bei den folgenden Betrachtungen eine Verdichtung der in Kapitel 3 erarbeiteten Liste zugrunde. Ziel von Kapitel 3 war es, einerseits einen möglichst umfassenden und detaillierten Überblick über wesentliche Anforderungen an eine Visualisierungsumgebung für Algorithmen und Datenstrukturen zu geben, und andererseits dem Leser ein detailliertes Verständnis auch scheinbar nebensächlicher Unterscheidungen in diesem Umfeld nahe zu bringen. Aus diesen Gründen entstand eine Liste von Anforderungen, die in ihrer Detailliertheit auch in zukünftigen Forschungen und Entwicklungen wiederverwendbar ist.

Für die Zwecke dieses Kapitels wollen wir wie angekündigt eine etwas griffigere Form dieser Anforderungsliste nutzen, damit der folgende Vergleich der verschiedenen Paradigmen nicht zu unübersichtlich wird und wir überhaupt eine Chance zu haben, geeignete Kandidaten zu finden.

Zusätzlich zum eben Gesagten fassen wir im Vertrauen darauf, dass der Leser bei unserer Entwicklung der kompletten Liste in Kapitel 3 die notwendigen Details verinnerlicht hat, in der verdichteten Liste verwandte Punkte zusammen, und führen aus den gleichen Gründen zum Teil griffigere Bezeichnungen ein; ein Beispiel hierfür ist der neue Punkt „Interaktivität möglich", was die Punkte „Es gibt eine Schnittstelle für Fragen / Rückmeldungen an Studierende" (O3), „Das Verhalten in Abhängigkeit vom Zustand der Datenstruktur ist steuerbar" (LO1) und „Ausgabe- (Fehler-) Meldungen als Reaktionen auf Interaktionen sind möglich" (LO2) aus Kapitel 3 umfasst.

Außerdem gibt es gewisse natürliche Abhängigkeiten zwischen verschiedenen Anforderungen. So impliziert die Möglichkeit, Schnittstellenoperationen von der Oberfläche des Werkzeugs aus aufrufen zu können (O4), dass diese zuvor spezifiziert werden konnten, allerdings nicht unbedingt, dass diese einfach zu spezifizieren sind (SP5). Deshalb haben wir letzteren Punkt unter „Die Spezifikation einer Datenstruktur ist natürlich, erweiterbar, minimal" mit aufgenommen,

und den ersteren unter „explorative Lernszenarien für Datenstrukturen sind möglich“.

Beim konzeptionellen Punkt K3 zeigte es sich, dass er für die Überprüfung konkreter Werkzeuge besser aufgeteilt wird in die Konfigurierbarkeit der angezeigten Informationen und die Konfigurierbarkeit des Versteckens bestimmter Teile; diese beiden Aspekte stecken in den folgenden Punkten „Funktionalität und einzelne Attribute von Elementen einer Datenstruktur können versteckt werden“ und „Angezeigte Attribute sind einfach zu konfigurieren“ (s. u.).

In der Praxis zeigt es sich, dass Anforderungen selten eindeutig erfüllt oder eindeutig nicht erfüllt sind, sondern eher entweder auf natürliche Weise in einem bestehenden Konzept umgesetzt werden können, oder ihre Implementierung mit erheblichem Mehraufwand oder einer strukturellen Änderung des jeweiligen Werkzeugs verbunden wäre. Deshalb wird auch bei der Zusammenfassung der Beschreibungen der Werkzeuge in der Tabelle am Ende dieses Kapitels die *Tendenz* angegeben, ob die Mehrzahl der zugehörigen Unteranforderungen bereits erfüllt sind oder leicht zusätzlich implementierbar sind oder ob beides nicht einfach möglich ist.

Die sich ergebende, kondensierte Liste wesentlicher Anforderungen besteht aus folgenden 18 Punkten:

1. Es gibt eine formale Semantik der Spezifikation (SP1).
2. Es gibt eine Werkzeugunterstützung für die Codegenerierungi (SP2).
3. Die Spezifikation einer Datenstruktur ist natürlich, erweiterbar, minimal (SP3, SP4, SP5, SP6, LO15).
4. Eine automatische Prüfung von Invarianten ist möglich (LO5).
5. Zufällige Szenarien können generiert werden (LO12).
6. Flüssige Animationen sind möglich (O1).
7. Explorative Lernszenarien für Datenstrukturen sind möglich (inklusive Schnittstelle für Aufrufe und Ereignisse und eigenen Eingabedatenmengen) (LO3, LO4, O4, O5).
8. Exploratives Lernen von Algorithmen (inklusive Vorhersage-Möglichkeit und konfigurierbaren Meldungen) ist möglich (O2, K2, LO10).
9. Interaktivität ist möglich (O3, LO1, LO2).
10. Eine Plattformunabhängigkeit zumindest der Laufzeitumgebung ist gegeben (SO3).
11. Es gibt einen Player mit ’undo’ und ’redo’ (O11, O12, O13, LO7, LO8, LO9).
12. Layoutalgorithmen sind möglich, vorhanden und erweiterbar (O8, O9, O10, K5).
13. Das Layout ist unabhängig von Datenstruktur und Algorithmus konfigurierbar (K4).
14. Die Laufzeit-Lizenzen sind kostenlos (SO5).
15. Funktionalität und einzelne Attribute von Elementen einer Datenstruktur können versteckt werden (O15, K3).
16. Ein entfernter Aufruf von Operationen ist von gängigen Programmiersprachen aus möglich (SO1, LO11).
17. Angezeigte Attribute sind einfach zu konfigurieren (K3).
18. Farbschemata als Reaktion auf die Änderung des Zustandes der Datenstruktur sind einfach zu konfigurieren (K1).

In den folgenden Abschnitten werden zunächst die Spezifikationsansätze und danach die Visualisierungsansätze kurz vorgestellt, bevor eine Bewertung gemäß dieser Liste erfolgt.

4.3. Vorstellung der Spezifikationsansätze

In diesem Abschnitt werden wir uns zunächst auf zwei Beispiele für textuelle Spezifikation konzentrieren, bevor wir einige Vertreter für zwei Arten visueller Spezifikationstechniken betrachten.

4.3.1. Textuelle Formen der Spezifikation

Unter den gebräuchlichen textuellen Formen der Spezifikation wählen wir die logikbasierte und die algebraische für eine nähere Betrachtung.

Die **algebraische Spezifikation** [Kla83] nutzt, wie der Name schon sagt, Algebren, um Spezifikationen zu formulieren. Es werden also repräsentativ für die zugrundeliegenden Trägermengen eines Datentyps (Typ-)Symbole und für die Funktionen auf diesen Trägermengen Funktionssymbole eingeführt. Durch Signaturen wird das Abbildungsverhalten der Funktionen, also die Reihenfolge, Anzahl und Typinformation der Parameter, und der Typ des Rückgabewertes angegeben. Die möglicherweise komplexen Beziehungen zwischen den Trägermengen können so auf Gleichungen zwischen typbehafteten Termen, die sich aus den erwähnten Symbolen zusammensetzen, zurückgeführt werden. Auf diese Weise findet eine operationelle Formulierung der im Zusammenhang mit dem Datentyp definierten Funktionen statt.

Die **logikbasierte Spezifikation** [Tar77] nutzt im Gegensatz zur gerade beschriebenen algebraischen Spezifikation meistens keine Typen und Signaturen, sondern arbeitet unmittelbar auf mathematischen (Träger-)Mengen, wie zum Beispiel den ganzen Zahlen. Hier werden für die Angabe von Spezifikationen auch komplexere Beziehungen als die Gleichheit oder Ungleichheit genutzt, wie zum Beispiel einfache Relationen wie '<' oder '>', oder auch eigens definierte komplexere Relationen, die durch Variablen benannt werden können. Im Falle der prädikatenlogischen Spezifikation schließt das auch die Nutzung von Quantoren, wie z. B. dem Existenz- und dem Allquantor für Aussagen über (möglicherweise unendliche) Mengen, ein.

Für beide genannten Spezifikationsarten existiert auch eine gewisse Werkzeugunterstützung, die es erlaubt, z. B. abstrakte Datentypen zu spezifizieren. Für die algebraische Spezifikation wurde etwa die Sprache CASL eingeführt [BM04], ein Beispiel für eine logikbasierte Spezifikationssprache ist Z [Jac97].

Um einen Eindruck von der Lesbarkeit dieser textuellen Arten der Spezifikation zu vermitteln, zeigt Abb. 8 eine algebraische Spezifikation einer einfachen Datenstruktur, einer sortierten Liste, mit CASL (entnommen aus [Kah05]). Abb. 9 oben zeigt eine einfache logikbasierte Spezifikation mit Z für einen endlichen Automaten (entnommen aus [Jac97]).

Im nächsten Abschnitt beschreiben wir visuelle Formen der Spezifikation, die in den letzten Jahren zum Teil weit mehr Popularität erlangt haben als die bisher beschriebenen textuellen.

4.3.2. Visuelle Formen der Spezifikation

Im Rahmen eines Systems, das mithilfe graphischer Darstellung einen Mehrwert in der Lehre schaffen will, bietet es sich natürlich an, diesen bereits an der Dozentenschnittstelle anzubieten. Da visuelle Spezifikationstechniken nicht nur in realen Projekten, sondern auch in der Lehre von

```
spec ELEM =
  sort Elem;
end

spec PARTIAL_ORDER =
  sort Elem;
  pred __ < __ : Elem × Elem;
  vars x, y, z : Elem
  •  x < x
  •  x < y ∧ y < x ⇒ x = y
  •  x < y ∧ y < z ⇒ x < z
end

spec LIST
  [ELEM]
=
  free type List[Elem] ::= nil | cons(first :?Elem; rest :?List[Elem]);
  op __ + +__ : List[Elem] × List[Elem] → List[Elem],
               assoc, unit nil;
  vars e    : Elem;
       l, l′ : List[Elem]
  axiom cons(e, l) + +l′ = cons(e, l + +l′);
  op reverse : List[Elem] → List[Elem];
  axioms reverse(nil) = nil;
         reverse(cons(e, l)) = reverse(l) + +cons(e, nil);
end

spec LIST_WITH_ORDER
  [PARTIAL_ORDER]
=
  LIST [sort Elem;]
then
  ops insert        : Elem × List[Elem] → List[Elem];
      order[__ < __] : List[Elem] → List[Elem];
  vars x, y : Elem;
       l    : List[Elem]
  axioms x < y ⇒
         insert(x, cons(y, l)) = cons(x, insert(y, l));
         order[__ < __](nil) = nil;
         order[__ < __](cons(x, l)) =
         insert(x, order[__ < __](l));
         insert(x, nil) = cons(x, nil);
         ¬ x < y ⇒
         insert(x, cons(y, l)) = cons(y, insert(x, l));
end
```

Abbildung 8: Beispiel einer algebraischen Spezifikation in CASL

```
STATE ::= patients | fields | setup | ready | beam_on
EVENT ::= select_patient | select_field | enter | start | stop | ok | intlk
FSM == (STATE × EVENT) ⇸ STATE

| no_change, transitions, control: FSM
|______________________
| control = no_change ⊕ transitions
| no_change = { s: STATE; e: EVENT • (s, e) ↦ s }
| transitions = { (patients, enter) ↦ fields,
|     (fields, select_patient) ↦ patients, (fields, enter) ↦ setup,
|     (setup, select_patient) ↦ patients, (setup, select_field) ↦ fields, (setup, ok) ↦ ready,
|     (ready, select_patient) ↦ patients, (ready, select_field) ↦ fields, (ready, start) ↦ beam_on, (ready, intlk) ↦ setup,
|     (beam_on, stop) ↦ ready, (beam_on, intlk) ↦ setup }
```

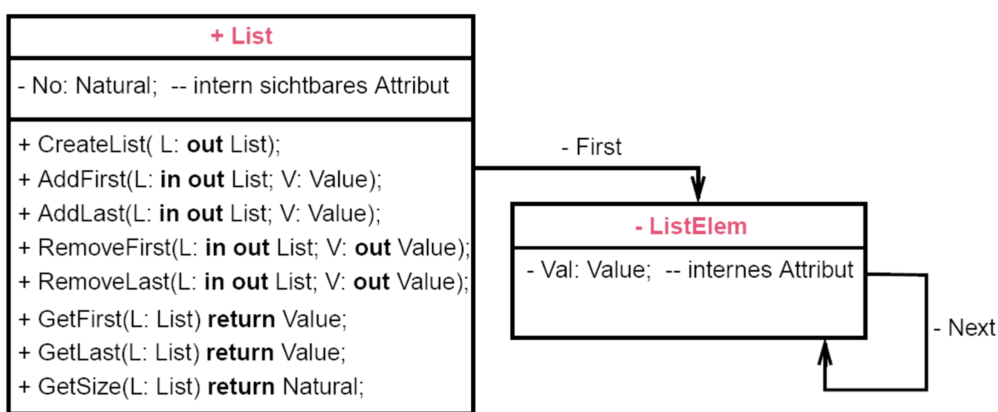


Abbildung 9: Oben: Beispiel einer logikbasierte Spezifikation in Z. Unten: Eine Liste als UML-Klassendiagramm.

Datenstrukturen häufig zum Einsatz kommen, wird unsere Forderung nach Bekanntheit oder Natürlichkeit der Spezifikationssprache von vorneherein besser erfüllt als mit anderen Techniken. Das gilt in besonderem Maße für die nächste beschriebene Spezifikationstechnik:

Die **UML** [UML03] ist eine wohlbekannte visuelle Modellierungssprache, die sich u. a. in verschiedenen Phasen der Softwareentwicklung immer mehr durchsetzt. Außerdem wird sie bereits in Lehrveranstaltungen dazu genutzt, Datenstrukturen zu spezifizieren. Für diesen Zweck sind insbesondere UML-Klassendiagramme geeignet. Abb. 9 unten zeigt ein Beispiel der als UML-Klassendiagramm spezifizierten Datenstruktur Liste (entnommen aus [Ein05a]).

Eine zweite Möglichkeit der visuellen Spezifikation, die **Graphtransformationen** oder -ersetzungen [MM98], werden seit Ende der 60'er Jahre untersucht und wurden zunächst eingesetzt

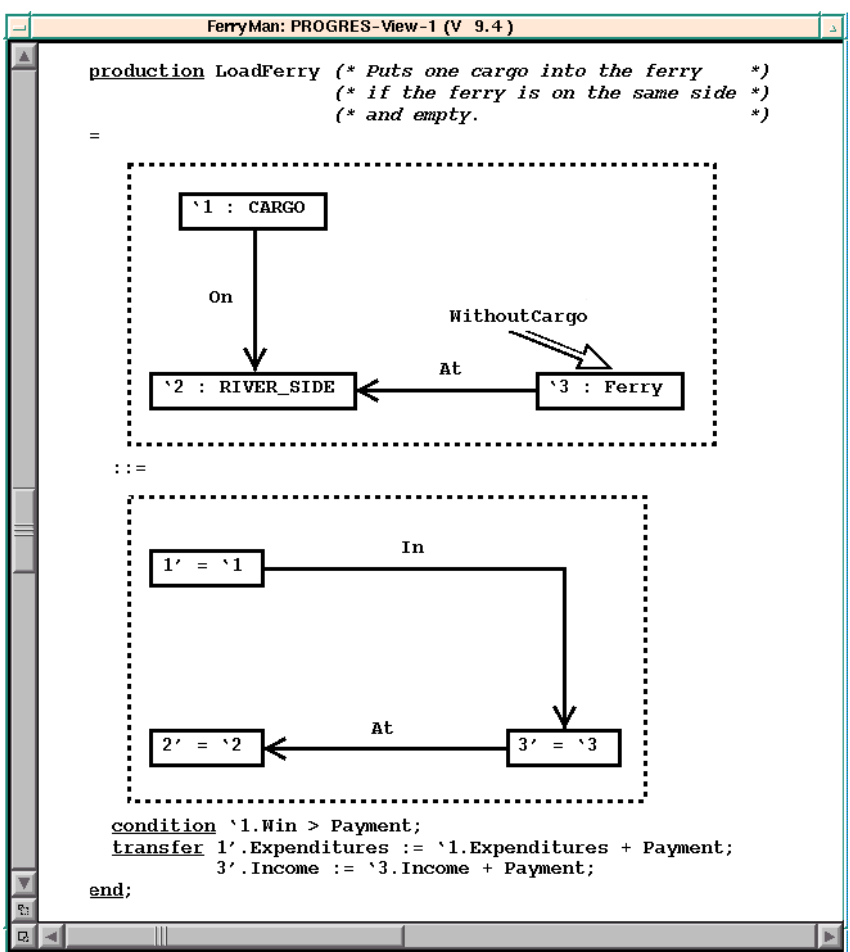


Abbildung 10: Das Fährmannproblem in PROGRES

für Themen wie die Bildanalyse (siehe z. B. [PR69, Sha70, Cha70]). Die Grundidee besteht darin, Änderungen an Graphen durch Operationen, eben die erwähnten Graphtransformationen, zu modellieren. Dabei können Bedingungen angegeben werden, unter denen bestimmte Operationen anzuwenden sind. Diese Operationen können dann zur Laufzeit einer regelbasierten Maschine auf einem Ausgangsgraphen Schritt für Schritt ausgeführt werden. Im Folgenden stellen wir beispielhaft drei Graphtransformationssysteme vor:

PROGRES [Sch94] ist ein Graphtransformations- und Visualisierungssystem inklusive der Möglichkeit der programmierten Graphersetzung. Letzteres bedeutet, dass man mit PROGRES Programme wie in einer höheren Programmiersprache schreiben kann, insbesondere auch unter Verwendung von Kontrollstrukturen, so dass Graphersetzungsschritte innerhalb einer Anwendung gesteuert und überwacht werden können. PROGRES wurde an der RWTH Aachen entwickelt. Abb. 10 zeigt die Modellierung des Fährmannproblems [ZS92] in PROGRES.

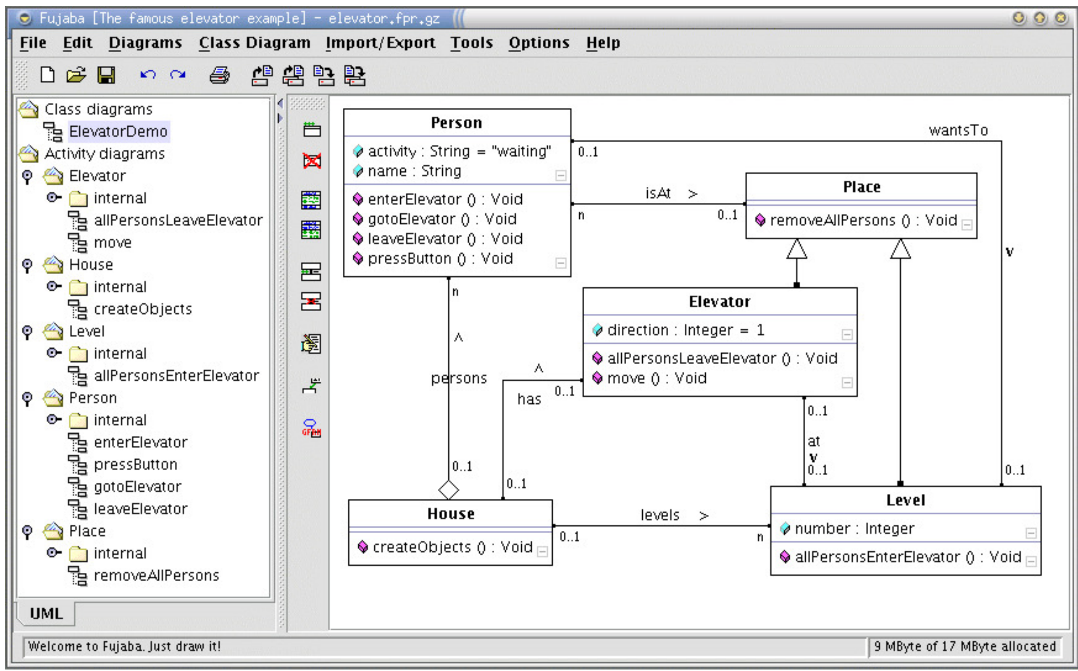


Abbildung 11: Ein Klassendiagramm in Fujaba

Fujaba [NNZ00] ist eine Programmierumgebung, die den grafischen Entwurf von UML-Klassenhierarchien, die Programmierung mit Kontrollflussgraphen und die Erstellung von Graphersetzungsregeln kombiniert. Fujaba wurde an der Universität Paderborn entwickelt. Abb. 11 zeigt eine Fujaba-Sitzung, in der mithilfe eines UML-Klassendiagramms ein System aus Haus, Aufzug, Etagen und Personen spezifiziert wird.

DiaGen [Min01] ist ein Rahmenwerk und Werkzeug zur Generierung mächtiger Diagrammeditoren aus Spezifikationen. Diese Spezifikationen bestehen zum Teil aus Hypergraphgrammatiken und ermöglichen ein fehlertolerantes Parsen und Nutzen von Diagrammen, die möglicherweise nur partiell korrekt sind. Die Möglichkeit der Codegenerierung ist dank der formalen Semantik der Spezifikationssprache gegeben. DiaGen ist komplett in Java geschrieben, wurde an der Universität Erlangen entwickelt und wird an der Universität der Bundeswehr München weiterentwickelt. Abb. 12 zeigt eine Beispiel-Sitzung mit DiaGen.

Nach dieser kurzen Vorstellung einiger Spezifikationsansätze und Werkzeuge werden wir im nächsten Abschnitt weitere Ansätze nennen, die für die Visualisierung geeignet sein könnten. Dabei werden auch verschiedene Paradigmen berücksichtigt, die neben unseren Anforderungen an die Visualisierung vielleicht auch noch passende Spezifikationsansätze beitragen können.

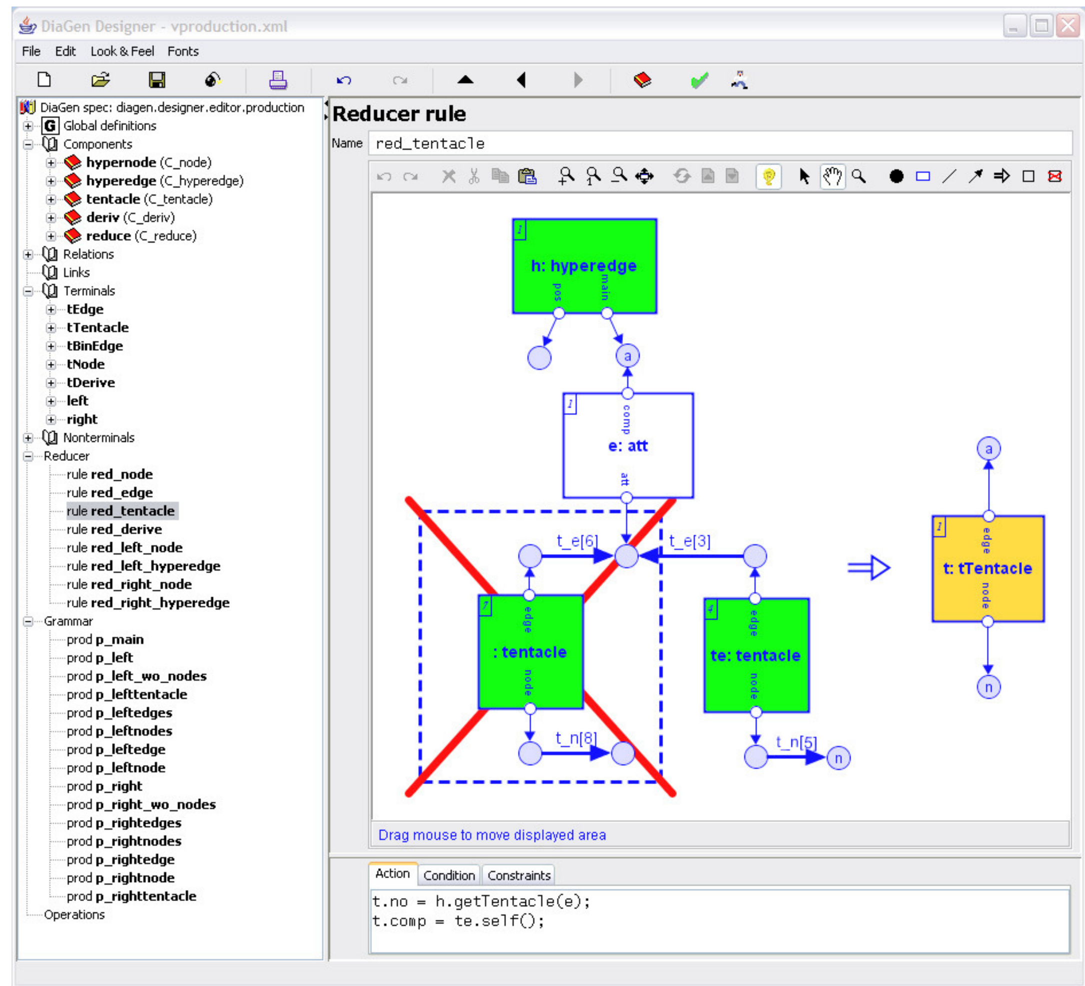


Abbildung 12: Eine Ersetzungsregel in DiaGen

4.4. Vorstellung der untersuchten Animationsansätze

In diesem Abschnitt werden Werkzeug-Repräsentanten der in Abschnitt 4.1 genannten Visualisierungs- und Animationsparadigmen aufgeführt. Wir beginnen mit nicht-spezialisierten Animationssystemen.

4.4.1. Nicht-spezialisierte Animationssysteme für Algorithmen

Die in diesem Abschnitt beschriebenen Systeme können zur Animation von Algorithmen und Datenstrukturen genutzt werden. Weil sie Grundprimitive zur Animation anbieten und durch eine lose Schnittstelle von einem „aufrufenden“ Client auch dazu genutzt werden können, von Datenstrukturen unabhängige Visualisierungen vorzunehmen, fassen wir sie unter der ersten Kategorie zusammen.

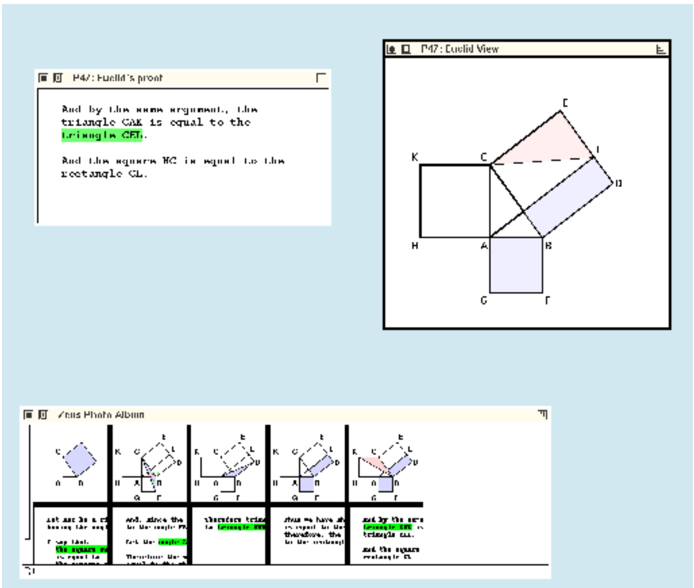


Abbildung 13: Ein animierter Beweis des Satzes von Pythagoras in Zeus; die Texte sind nicht lesbar und für die Illustration nicht entscheidend

Zeus [Bro91] ist ein System, das ebenso wie die älteren in Kapitel 2 erwähnten Systeme Balsa, Tango und Polka auf dem Konzept der 'interesting events' aufsetzt. Im Falle von Zeus heißt das, dass ein Programm die interessanten Ausgabe-Ereignisse in eine Datei schreibt, die von Zeus interpretiert wird. Zeus kann mehrere Sichten darstellen. Abb. 13 zeigt eine Sitzung mit Zeus, in der der Nutzer eine Animation des Beweises des Satzes von Pythagoras sieht.

Animal [RSF00] ist ein Werkzeug zur Erzeugung und Nutzung von Animationen aus wenigen Grundbausteinen. Das Erstellen neuer Animationen kann interaktiv, mit einer eigenen Skriptsprache oder über eine API geschehen. Animal wurde an der Universität Siegen entwickelt und ist in Java geschrieben. Die Stärke von Animal liegt in der Möglichkeit, mit einem relativ einfachen Editor oder einer Skriptsprache schnell kontinuierliche Animationen entwickeln zu können. Diese können dann interaktiv oder über eine API manipuliert und abgespielt werden. Dabei gibt es sehr hilfreiche graphische Effekte wie Ein- und Ausblendeeffekte, Farbänderungseffekte und Rotationseffekte für die unterstützten graphischen Primitive.

Einige Zeit nach seiner Erstellung wurde Animal in ein Rahmenwerk, genannt **ANIMALFARM** [Röß02], eingebettet, das einen Rahmen auch für komplexere Algorithmenanimations-Systeme mit einem wesentlich höheren Abstraktionsgrad als Animal schafft.

Ein anderes System, das aus Grundprimitiven Animationen erzeugen kann, ist das 1998 vorgestellte System **GAWAIN** [HD98]. GAWAIN kann als Applikation oder als Applet betrieben werden und wird über Ereignisse gesteuert. In GAWAIN 4.0 gibt es zwei unterstützte Arten der Kommunikation mit dem Klienten-Programm. Die eine Art ('animation') läuft über die Standard-Ein- und -Ausgabe des fernsteuernden Programms, d. h. das eigene Programm sendet seine

Ereignis-Aufrufe über die Standardausgabe an GAWAIN und erhält über die Standardeingabe die Eingaben von GAWAIN. Auf diese Weise können Fernsteuerungen von jeder Programmiersprache aus genutzt werden, allerdings zu dem Preis, dass sich der Programmierer Gedanken über die Erzeugung, Löschung und Änderung graphischer Primitive machen muss. Diese umfassen in GAWAIN dreidimensionale Punkte, Kreise, endliche und sogar unendliche Linien und Polygone. Neben der geometrischen Information, wozu in GAWAIN z. B. Koordinaten von Punkten oder der Radius eines Kreises zählen, werden auch nicht-geometrische Informationen der Primitive verwaltet, wozu z. B. die Größe eines Objektes oder seine Farbe gehören. Für die Programmiersprache Java gibt es zusätzlich eine explizite Schnittstelle für die Fernsteuerung.

Die andere mögliche Art der Kommunikation mit GAWAIN ('immediate') zeigt die Ergebnisse eines zu animierenden Algorithmus an, sobald Änderungen bestimmter Daten mit der Maus an den graphischen Objekten vollzogen werden.

Es existiert eine Player-Schnittstelle mit flüssigen Animationen, die dank des Vorhandenseins der kompletten Historie den Ablauf der Animation unabhängig von der Laufzeit des Programms nachvollziehen kann. Die Ausgabe kann in GAWAIN auf mehrere Sichten verteilt werden. Zufällige und geordnete Eingabedaten können über Eingabe-Generatoren erzeugt und getestet werden.

4.4.2. Elektronische Bücher für die Lehre von Algorithmen und Datenstrukturen

Ein *elektronisches Buch* (engl. *EBook*) in seiner allgemeinen Form ist – wie der Name schon sagt – einfach eine elektronische Version eines Buches. In unserem speziellen Fall der Untersuchung von Algorithmenanimations-Systemen fassen wir in diesem Unterabschnitt Werkzeuge zusammen, die es ermöglichen, ein elektronisches Buch für die Lehre von Algorithmen und Datenstrukturen zu erstellen oder zu simulieren. Ein solches Buch kann der Lernende als Applikation auf seinem Rechner starten und darin Informationen zu Algorithmen und Datenstrukturen nachlesen, zum Beispiel über Hypertext-Links navigieren und bei Bedarf Animationen starten, die Datenstruktur-bezogene Themen visualisieren.

Ein Beispiel für diese Kategorie haben wir schon in Kapitel 2 in **HalVis** kennengelernt. Wir nehmen HalVis in die Liste zu untersuchender Ansätze auf, ohne es hier nochmal vorzustellen.

Das Animationssystem **JCAT** [BR97], das in Java geschrieben wurde, stellt eine Entwicklungsumgebung für Algorithmenanimationen im Web dar. JCAT bietet weitaus mehr Möglichkeiten als ad hoc programmierte Java-Applets. Es können Algorithmenanimationen kreiert und online demonstriert werden. Es gibt ein gewisses Maß an Interaktivität, die der Lehrende limitieren oder erweitern kann. Die Koppelung von Logik und Anzeige geschieht wie z. B. auch bei Zeus über 'interesting events'. Abb. 14 zeigt die Animation verschiedener Sortieralgorithmen in JCAT.

AlgoViz [UF04] bietet das Prinzip eines Baukastens: Durch Nutzung der 'Java Bean'-Technologie ist es möglich, auf verschiedenen Ebenen wiederverwendbare Elemente für die Visualisierung von Algorithmen und Datenstrukturen zu schreiben. Diese können dann zu größeren Einheiten zusammengefügt werden. Beispiele für die Funktionalität dieser 'Java Beans' rangieren von einer einfachen Schaltfläche über einen Datei-Dialog bis zu Teilen von Algorithmen. Die Bausteine können in einer visuellen Entwicklungsumgebung zusammengefügt werden.

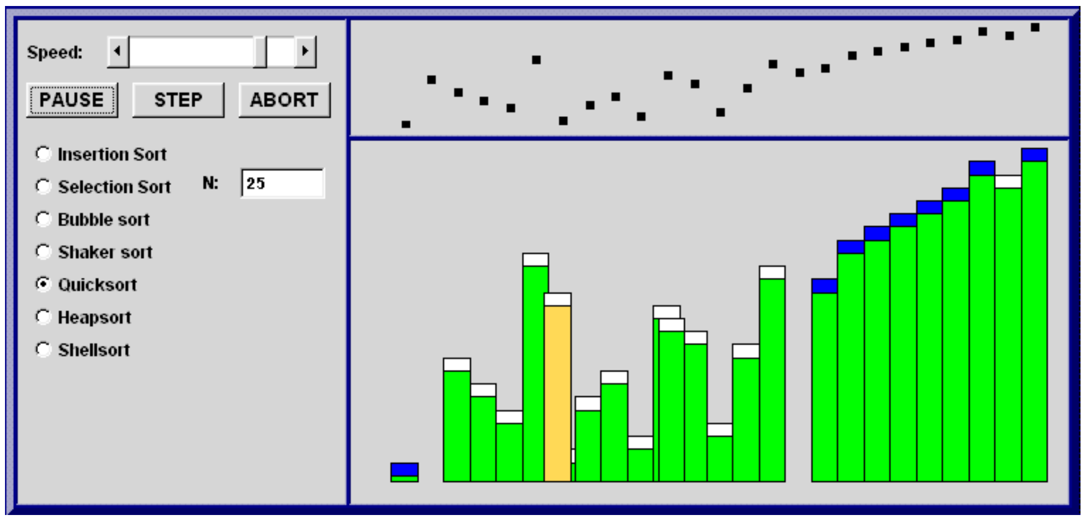


Abbildung 14: Die Abbildung zeigt die Animation von verschiedenen Sortieralgorithmen in JCAT in einer Ansicht

SALABIM [SAL04] ist eine Java-Bibliothek für die Animation von Algorithmen, die innerhalb des zugehörigen SALA-Konzeptes [Fal02] für die hypermediale Lehre von Algorithmen in einem Webbrowser eingesetzt werden kann. Wie wir schon bei der Vorstellung von HalVis in Kapitel 2 gesehen haben, kann diese Art der Einbettung von Algorithmenanimationen in komplexe Erklärungen, Querverbindungen zwischen diesen etc. ziemlich viel Aufwand für den Dozenten bedeuten. Deswegen wurde in SALA Wert darauf gelegt, Muster zur Verfügung zu stellen, die den Vorbereitungsaufwand des Dozenten bei der Erstellung neuer Übungsaufgaben minimieren sollen. So sind zum Beispiel für jede interaktive Übungsaufgabe drei fest implementierte Arten von Interaktivität vorgesehen ('simulation', 'practice' und 'animation'), die durch ein zugrunde liegendes, in SALA definiertes Modell und den zugehörigen, nur einmal zu schreibenden Code ermöglicht werden. Insgesamt wurde an viele der Anforderungen gedacht, die auch wir in unserem Grundszenario als wesentlich erachtet haben; so gibt es neben verschiedenen Formen der Interaktivität auch eine Player-Funktionalität mit 'undo' und flüssigen Animationen (wenn dies vom Dozenten geeignet vorbereitet wurde). Abb. 15 zeigt ein einfaches, in SALABIM mitgeliefertes Applet, in dem anhand des korrekten Stapelns von Blöcken das Konzept der Interaktivität veranschaulicht wird.

4.4.3. Algorithmenanimation durch Code-Interpretation

Nach den im letzten Abschnitt genannten Werkzeugen, die viele Möglichkeiten in der Animation, aber einen niedrigen Abstraktionsgrad der Spezifikation besitzen, kommen wir jetzt zu einigen Beispielen von Quellcode-Visualisierern, also Werkzeugen, die weitgehend automatisch aus dem Programmcode einer gängigen Programmiersprache eine Visualisierung erzeugen können.

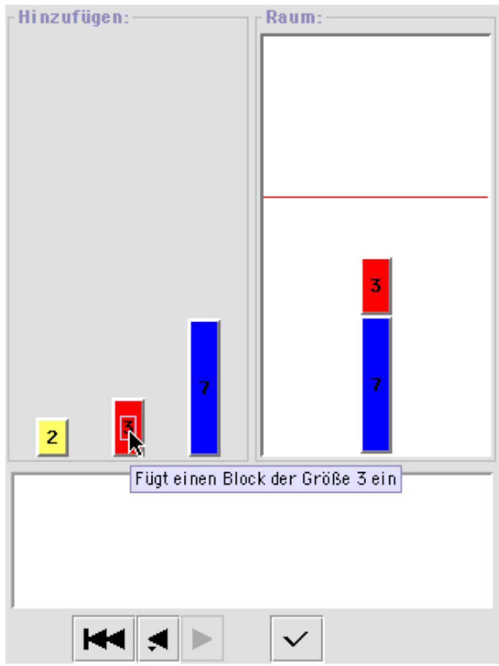


Abbildung 15: Beispielapplet der SALABIM-Bibliothek: Blöcke stapeln

DDD [DDD03] ist ein graphisches Frontend für verschiedene Unix-Debugger und geeignet zum Visualisieren von Datenstrukturen, die auch aus Zeigerstrukturen bestehen dürfen. DDD wurde an der Universität Passau entwickelt und wird über GNU vermarktet. Nachdem Haltepunkte in den Code, der schrittweise ausgetestet werden soll, eingefügt worden sind, können mit DDD sehr schnell Animationsinformationen aus den in einem Programm verwendeten Datenstrukturen extrahiert und dargestellt werden. Allerdings gilt dies nur für Unix-Prozesse. Elemente in einer Datenstruktur werden durch Graphknoten repräsentiert, Zeiger zwischen den Elementen entsprechend als Kanten zwischen den Graphknoten. Durch eine Alias-Erkennung können sogar zirkuläre Zeigerstrukturen identifiziert werden.

Die Spezifikation der Datenstruktur besteht in dem Prozess-Quellcode, der noch dazu angereichert werden muss durch Haltepunkte oder spezielle 'continue statements'. Diese dienen dazu, eine dynamische Animation zu kreieren. Durch die Möglichkeit, ganze Sitzungen abzuspeichern, ist es möglich, Animationen vorzubereiten. Außerdem besteht die Möglichkeit des 'undo' und 'redo', da sich DDD die dargestellten Programmzustände merkt. Abb. 16 zeigt eine Beispielsitzung mit DDD.

Jeliot [MMSBA04] ist ein System, das Java-Quellcode mit wenigen Änderungen, oder in einfachen Fällen auch direkt animieren kann. Die Animationen zeigen alle Zustandsänderungen, nachdem der Benutzer die vom Präprozessor identifizierten graphischen Objekte interaktiv platziert hat. Abb. 17 zeigt die Animation eines einfachen Java-Programms, in dem die veränderten Abhängigkeiten und Attribute graphischer Primitive animiert dargestellt werden.

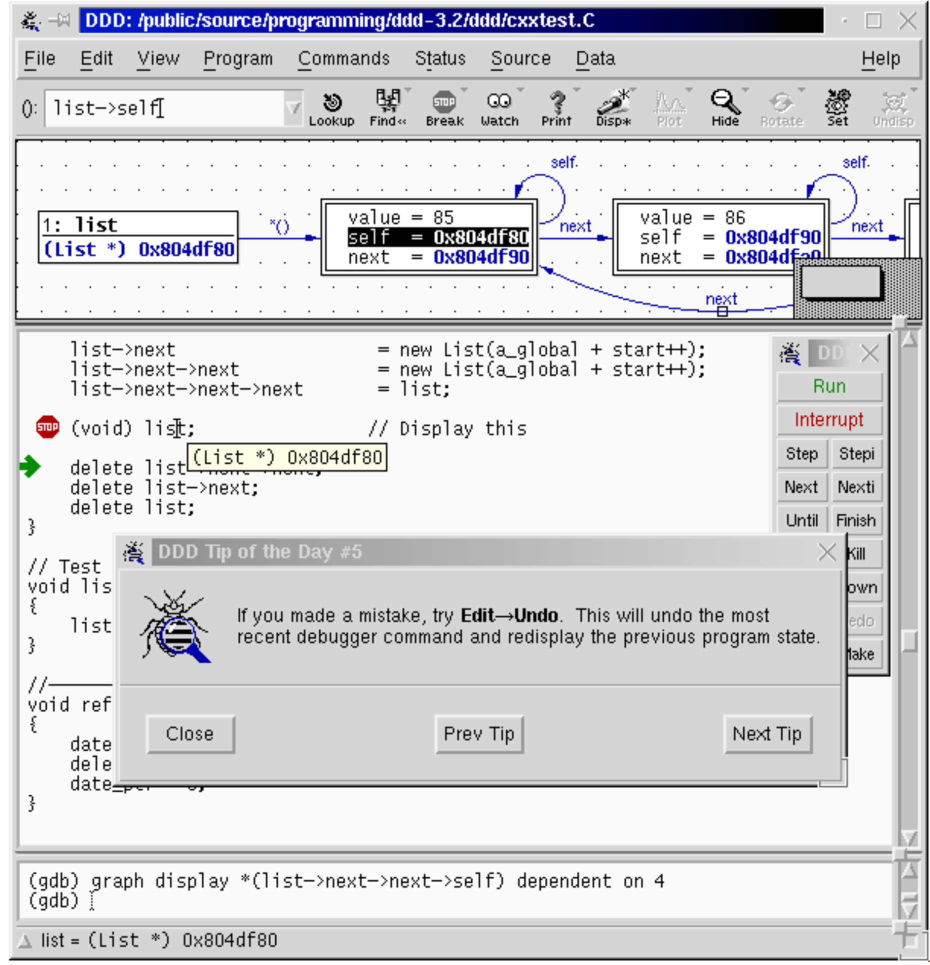


Abbildung 16: Beispielsitzung mit DDD

Ein weiterer interessanter Ansatz, der immer noch der Code-Interpretation zugeordnet werden kann, aber auch Elemente der weiter unten genannten Visualisierungswerkzeuge für eine deklarative Art der Algorithmen-Spezifikation enthält, ist **Javiva** [TZ01]. Javiva ermöglicht die Visualisierung und Validierung von Studentenprogrammen, indem Vorbedingungen, Nachbedingungen und Funktionen zur Laufzeit aus speziellen Kommentaren im Programmtext extrahiert werden. Das erfordert natürlich, dass die Lernenden vorher diese Deklarationen in den Java-Programmtext eingefügt haben. Im Falle von Javiva sind die Deklarationen logische Formeln, die auch angereichert werden können durch so genannte Abstraktions-Funktionen. Letztere definieren in üblicher Java-Syntax Hilfsfunktionen, die in den Formeln für die Vor- und Nachbedingungen genutzt werden können.

Javiva generiert aus diesen Bedingungen geeignete Aktionen für den Fall einer verletzten Bedingung. Dann wird der Benutzer mit einer Fehlermeldung informiert. Es ist auch möglich, Methoden mithilfe einer API-Schnittstelle aufzurufen. Eine rudimentäre Art der Visualisierung,

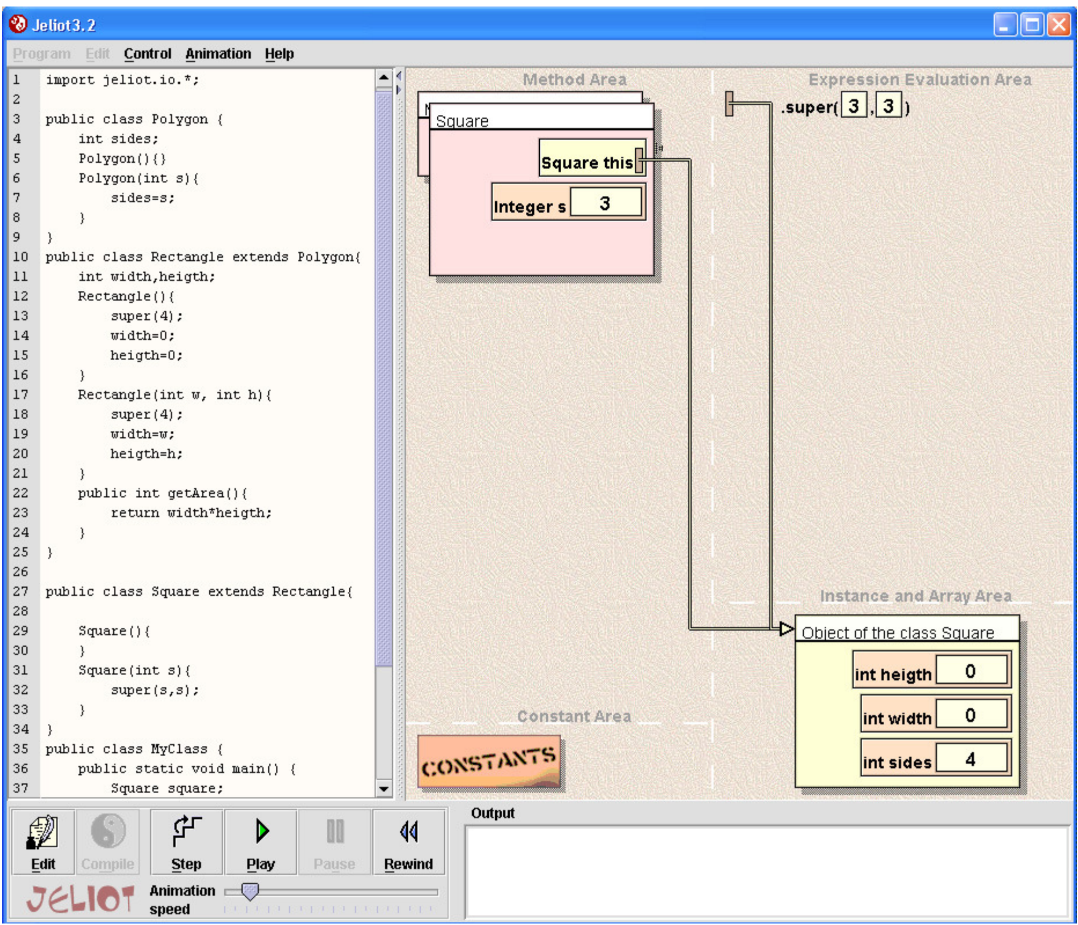


Abbildung 17: Beispielsitzung in Jeliot

die für bestimmte Datenstrukturen ausreicht, ist die Möglichkeit, zusätzlich zu einer Abstraktions-Funktion Visualisierungs-Funktionen anzugeben, die dann zur Laufzeit von Javiva für jedes Objekt dieser Klasse aufgerufen werden.

Bezüglich Interaktivität bietet Javiva die Möglichkeit, den Ablauf des Codes zu verfolgen und Verletzungen der spezifizierten Bedingungen durch Meldungen anzuzeigen.

Aus einer Reihe weiterer ähnlicher Systeme wie Kami [Ter01], WinHipe [NBPFnVI00] und **ZStep95** [LF97] wird noch letzteres in unsere Liste zu untersuchender Werkzeuge aufgenommen. ZStep95 bietet animierte Code-Interpretation mit ausgefeilten Möglichkeiten der Anpassung der Anzeige. Beispiele hierfür sind 'step over' und 'back up'-Operationen und die Anpassbarkeit der Ablaufgeschwindigkeit. Das System wurde für die Code-Interpretation von LISP-Programmen entwickelt. Ähnlich wie bei anderen Werkzeugen zur Code-Interpretation wird eine statische Algorithmenanimation angeboten.

4.4.4. API-basierte Graphenbibliotheken

In diesem Abschnitt werden einige Graphenbibliotheken der Liste zu untersuchender Werkzeuge hinzugefügt, die neben anderen Zwecken auch für die Animation von Algorithmen und graphbasierten Datenstrukturen genutzt werden können. Je nachdem, von welchem anderen Teil-Werkzeug diese Graphenbibliotheken genutzt werden, kann die Brauchbarkeit für unsere Zwecke variieren: Zum Beispiel kann jede Graphenbibliothek genutzt werden, um hochinteraktive Anwendungen zu entwickeln, oder um irgendwelche Inhalte passiv zu visualisieren. Deshalb können wir bei den beschriebenen Werkzeugen in Bezug auf Interaktivität oder Themen wie Spezifizierung oder Player-Schnittstelle im nächsten Abschnitt nur dann eine Aussage machen, wenn im jeweiligen Werkzeug bereits ein Gegenüber in Form eines Nutzers der angebotenen API angeboten wird.

AGD [AGD03] ist eine Bibliothek von Algorithmen zur Visualisierung von Graphen und wurde von mehreren deutschsprachigen Universitäten entwickelt. In AGD sind viele graphische Grundanforderungen, die für die Darstellung von graphähnlichen Datenstrukturen notwendig sind, bereits in C++ zur Verfügung gestellt. Zusätzliche gibt es schon fertige, mächtige Pakete wie einen 'orthogonal tree' und einen 'general graph'. Das Paket ist frei verfügbar, wobei dies für das zugrundeliegende LEDA nur unter bestimmten Umständen gilt. Es stehen verschiedene Layoutalgorithmen zur Verfügung.

yFiles [WEK03] vereinigt die Vorteile einer Graphbibliothek wie AGD mit flüssigen Animationen, einer großen Anzahl von bereits bestehenden Layoutalgorithmen in Java und der einfachen Möglichkeit, das Layout auch manuell einfach anzupassen. Wegen der Implementierung in Java ist yFiles plattformunabhängig. Die Laufzeitlizenzen für yFiles sind für die akademische Lehre frei. Abb. 18 zeigt zwei der vielen in yFiles enthaltene Layoutalgorithmen in Aktion: Das CircularLayout und das TreeLayout.

VisualGraph [LNR03] ist eine Bibliothek von Graphen-Kommandos, die Animalscript-Code – die Skriptsprache von Animal – erzeugen können und damit eine große Bandbreite von graphischen Primitiven zur Verfügung haben. Die erzeugten Funktionen können als API im eigenen Java-Programm aufgerufen werden. VisualGraph sieht die Möglichkeit vor, den entstehenden Animal-Code in JHAVE [NEN00] einzubinden. JHAVE ist eine Java-basierte Plattform für Animations-Engines, die Möglichkeiten bietet, einfache 'instructional interaction objects' in den Strom der visuellen Ausgabe-Informationen einzuflechten. Auf diese Art ist es möglich, Zwischenfragen an die Studierenden zu stellen und deren Antworten zu überprüfen. VisualGraph ist auch fähig, zufällige Graphen-Szenarien zu generieren.

Nach diesem Ausflug in die Welt einiger Graphenbibliotheken gehen wir im kommenden Abschnitt kurz auf eine interaktive Art der Erzeugung von Animationen ein.

4.4.5. Erzeugung von Animationen durch direkte Manipulation

Das **DANCE**-System [Sta91] ist ein Beispiel für ein Algorithmenanimations-System, in dem der Nutzer ein so genanntes 'programming by demonstration' durchführt. Im Falle von DANCE heißt das, dass der Benutzer mithilfe eines graphischen Editors ein Animationsszenario demonstriert, woraufhin das System aus dieser Beschreibung ein Programm extrapoliert. DANCE ist

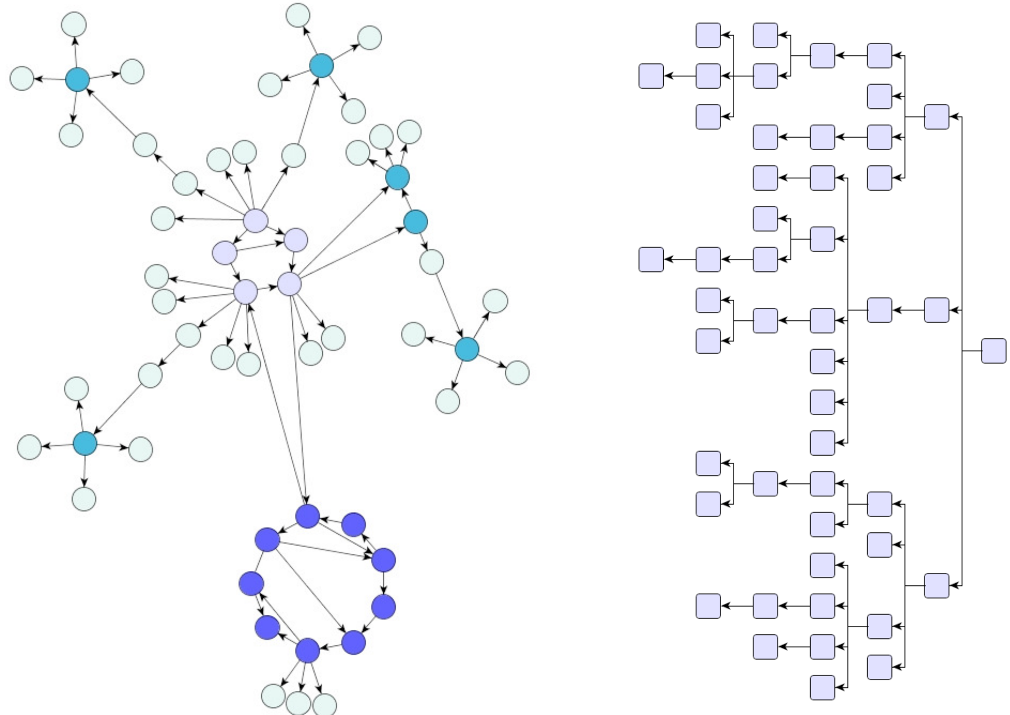


Abbildung 18: Zwei yFiles-Layoutalgorithmen beispielhaft gezeigt. Links: CircularLayout. Rechts: TreeLayout.

ein System, das auf TANGO aufsetzt (siehe Kapitel 2). Genauer gesagt werden Textzeilen erzeugt, in denen die Animation als Eingabe für TANGO spezifiziert ist. Der Vorteil von DANCE ist seine Fähigkeit des 'rapid prototyping', wodurch die Autoren hoffen, die Aktivität der Lernenden und damit auch die Häufigkeit der Nutzung zu steigern.

Auch das bereits in Kapitel 2 erwähnte System **LENS** [MS94] passt in die Kategorie der direkten Manipulation und wird in die Liste zu untersuchender Werkzeuge aufgenommen. LENS bietet wie in Kap. 2 beschrieben einen hohen Grad an Interaktivität vom Standpunkt eines Studierenden aus, der selbst eine Animation erstellt.

4.4.6. Deklarative Systeme für die Animation von Algorithmen

Beim deklarativen Ansatz werden Beziehungen zwischen dem Programmzustand und einer graphischen Repräsentation hergestellt. Ein besonders einfaches, aber typisches Beispiel hierfür ist die Zuordnung „interessanter“ Teile der Datenstruktur zu graphischen Primitiven, also etwa eines Variablenwertes zu der Höhe eines Rechtecks. Diese Deklarationen oder auch Prädikate werden üblicherweise in speziellen Kommentaren im ursprünglichen Quellcode abgelegt, so dass der Code im Wesentlichen unverändert bleibt und von einem Standard-Compiler übersetzt werden kann. Ein zusätzlicher Compiler oder Interpreter, der in das deklarative System einge-

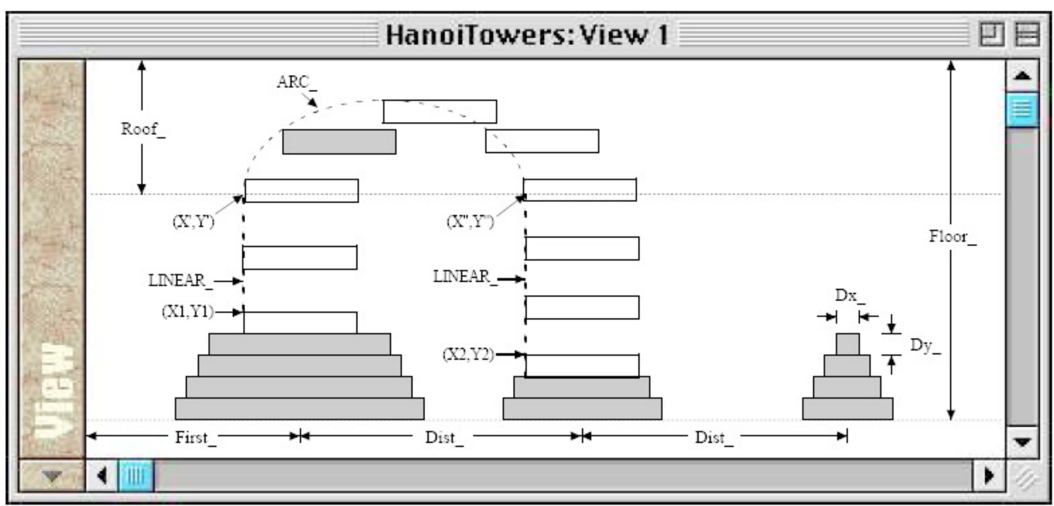


Abbildung 19: Türme von Hanoi in Leonardo

bunden ist, extrahiert die spezifizierten Prädikate aus dem Code und nutzt sie für die Abbildung des aktuellen Zustands in geeignete Aktionen, die dann regelmäßig ausgeführt werden.

In **Pavane** [RCWP] werden Prädikate als Annotationen in den Programmtext eingefügt. Aus diesen Einfügungen werden Animationen generiert. Änderungen im Programmzustand, die ein definiertes Prädikat betreffen, führen zu einer Aktualisierung der graphischen Anzeige. Dies wird durch ein ständiges Überwachen von Variablen und anderen kritischen Ressourcen erreicht. Pavane konnte zunächst nur PROLOG-Code verarbeiten, später auch C.

Pavane wurde bisher u. a. für die Visualisierung nebenläufiger Algorithmen in der Programmiersprache Swarm und für Computerarchitektur-Simulationen genutzt. In Pavane wird auch Layoutfunktionalität deklarativ spezifiziert.

Das etwas neuere System **Leonardo** [CDFP00] ist ein weiteres Beispiel eines deklarativen Algorithmenanimations-Systems. Hier wird eine logische Programmiersprache ALPHA benutzt, um Algorithmen und in ihnen genutzte Datenstrukturen auf Visualisierungen abzubilden. Durch zusätzliche Deklaration von Bewegungskurven können kontinuierliche Animationen entlang dieser Kurven dargestellt werden. Zusätzlich ist es möglich, durch spezielle Prädikate die Visualisierung an- und abzuschalten.

Durch mögliche Eingabe von Datenwerten von Hand und eine Kontrollleiste mit der Möglichkeit, vorwärts und rückwärts abzuspulen, wird eine gewisse Interaktivität erreicht. Abb. 19 zeigt eine Sitzung mit Leonardo, in der die Türme von Hanoi visualisiert werden.

PROVIDE [Moh88] ist ein System, in dem durch interaktive Computergraphiken die Programmausführung von Unix-Prozessen illustriert werden kann. Der Benutzer spezifiziert die Variablen, die sichtbar sein sollen, woraufhin die Visualisierung weitgehend automatisch erfolgt. Die Besonderheit in PROVIDE ist, dass der Benutzer die Bindung an bestimmte Variablen zur Laufzeit ändern kann. Das Haupt-Einsatzgebiet von PROVIDE ist das Debugging.

Forms/3 [BAD+01] verwendet einen deklarativen Ansatz der Spezifikation von Datenabhängigkeiten. Programme bestehen hier aus Programmblättern (Formularen, Tabellen) ähnlich denjenigen bekannter Tabellenkalkulationsprogramme. Die Zellen der Formulare enthalten Daten oder Formeln zur Berechnung von Daten. Die Graphik folgt dem Spreadsheet-Paradigma. Es gibt Zellen auf dem Formular und Formeln für diese Zellen. Bewegung kann durch Constraints zwischen den Zellen unter Angabe von Zeitangaben auf Objekten mit Koordinaten, Intensität usw. spezifiziert werden. Wegen der permanenten Anpassung und Auswertung nennen die Autoren ihren Ansatz 'responsive' statt 'declarative'. Eine automatische Überprüfung von Constraints ist in das System eingebaut. Die graphische Schnittstelle erlaubt es, die Animation vorwärts- und rückwärts laufen zu lassen. Insgesamt ist eine einfache Interaktivität möglich,

4.4.7. Skriptbasierte Algorithmenanimation

Skriptsprachen wie **JAWAA** [AFJ+03] können schnell schöne Animationen generieren. Generiert werden von JAWAA Dateien mit der Namenserweiterung '.anim', die dann im Browser visualisiert werden. Vom Konzept her hat JAWAA große Ähnlichkeit zu dem in Kapitel 2 erwähnten JSamba. JAWAA zielt besonders darauf ab, Primitive für die gängigen Datenstrukturen Felder, Stapel, Warteschlange, Graphen und Bäume anzubieten. Im Gegensatz zu JSamba ist die Anzahl der Knoten von Polygonen nicht auf eine feste Anzahl eingeschränkt, was es möglich macht, beliebige Graphen aufzubauen. Durch die Skriptbasierung kann eine Fernsteuerung von beliebigen Programmiersprachen aus erreicht werden.

4.4.8. Visuelle, regelbasierte Entwicklungsumgebung

Stagecast [Sta03] ist eine visuelle Entwicklungsumgebung für regelbasierte Spiele, die speziell für die Nutzung durch Kinder konzipiert wurde (entwickelt und vertrieben durch die Firma Stagecast). Stagecast unterstützt den Benutzer darin, graphisch ansprechende Szenarien mit kontinuierlicher Animation schnell zu entwickeln. Abb. 20 zeigt ein mit Stagecast entwickeltes Computerspiel.

4.4.9. Icon-basierte Simulationssprache

Simul8 [Sim03] ist eine 'icon'-basierte Simulationssprache, mit der Produktionsabläufe nachgebildet, getestet und optimiert werden können (entwickelt und vertrieben von der Firma Simul8). Simul8 bietet die kontinuierliche Animation und verschiedene Möglichkeiten der zustandsabhängigen Visualisierung an.

4.4.10. Entwicklungsumgebungen für graphische Oberflächen oder Benutzerschnittstellen

Amulet [Amu03] ist eine Entwicklungsumgebung für die Erstellung graphischer, interaktiver Benutzerschnittstellen (entwickelt an der Carnegie Mellon University). Neben der einfachen Realisierung von Benutzeroberflächen mit direkter Manipulation werden Animationen unterstützt.

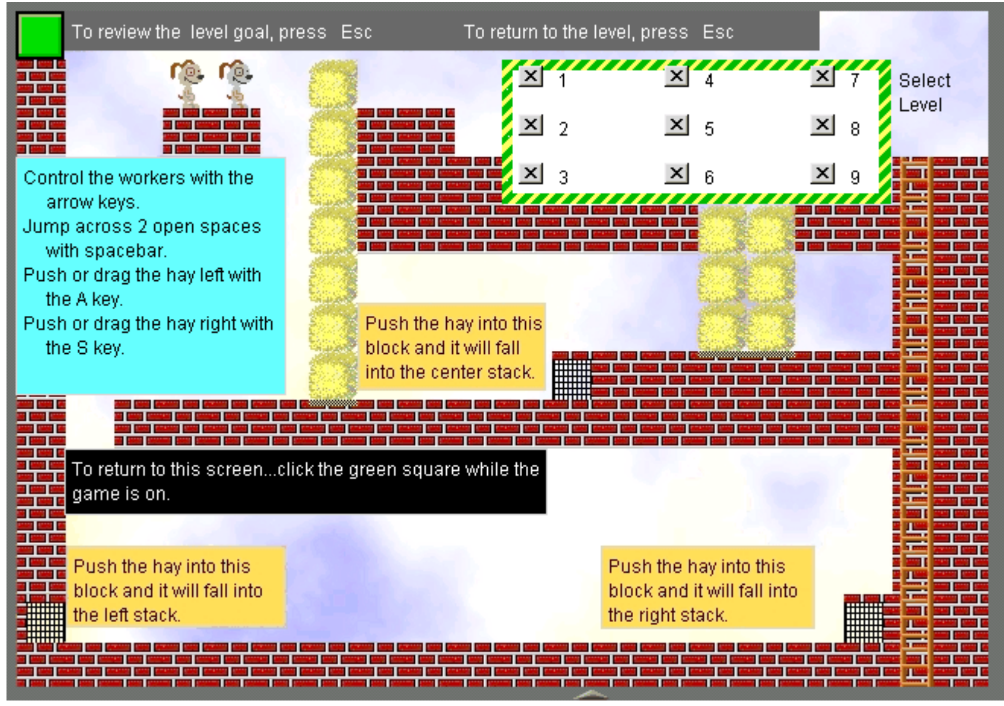


Abbildung 20: Ein interaktives Szenario in einem mit Stagecast entwickelten Computerspiel

Tilcon [Til03] ist ein Graphikeditor zur Entwicklung grafischer Benutzeroberflächen für Simulationen und Messwerte (entwickelt und vertrieben von der Firma Tilcon). Für die Anzeige von Animationen bietet Tilcon, das auf C++ basiert, so genannte 'animation areas' an. Das Werkzeug bietet auch spezielle Unterstützung für Animationen mit kontinuierlichen Bewegungen an. Anwendungen können über Ereignisse angebunden werden.

4.4.11. Sonstige Ansätze für die Erstellung von Animationen

Ein Ansatz, der die Generierung von spezifizierbaren Prozessen untersucht, ist **Ganimal** [Gan05]. Ganimal ist für die Simulation und Lehre verschiedener Phasen des Compilerbaus entwickelt worden. Ein Beispiel des Einsatzes ist die Simulation endlicher Automaten.

Ganimal wurde auch schon prototypisch für die Visualisierung des Heapsort-Algorithmus eingesetzt. Ganimal nutzt die Spezifikationssprache Ganifa, die eine Erweiterung von Java um verschiedene Möglichkeiten der Annotation – u. a. um 'interesting events' – bietet. Nach der Implementierung des Algorithmus in Ganila wird dieser nach Java übersetzt.

Die Stärken dieses Ansatzes bei der – bisher allerdings nur ansatzweise erprobten – Anwendung für die Algorithmenanimation liegen in der prinzipiellen Interaktivität der erzeugten Szenarien und der Offenheit des Ansatzes, der hochgradig konfigurierbar ist. Zum Beispiel können auch Invarianten und komplexe Aufruf-Funktionen implementiert werden.

Ebenfalls schön sind die Möglichkeiten, ein vorausschauendes Layout zu implementieren und gleichzeitig zum Ablauf der Animation zugehörigen Code darzustellen.

Das generische Programmvisualisierungssystem **Alma** [PH03] ermöglicht die Visualisierung von Algorithmen in einer nicht festgelegten Eingabesprache. Stattdessen kann jede denkbare Eingabesprache mit einer abstrakten Grammatik spezifiziert werden, für die jeweils zusätzlich ein Frontend zu schreiben ist. Zur Laufzeit eines Programms baut das Frontend gemäß der Analyse dieses Programms einen so genannten DAST – einen abstrakten Syntaxbaum – auf, der den Programmzustand repräsentiert. Dieser DAST wird dann mithilfe einer regelbasierten Umgebung ausgewertet und entsprechend den Auswertungsregeln visualisiert. Der Vorteil dieses Ansatzes ist die Allgemeinheit der möglichen Eingabe-Sprache; so betonen die Autoren das Wegfallen der Notwendigkeit vieler anderer Ansätze, Annotationen in Programme einer vorgegebenen Programmiersprache einzufügen.

Es wird an zusätzlicher Werkzeugunterstützung und der Komplettierung des Alma-Rahmenwerks gearbeitet.

4.5. Bewertung gemäß der genannten Anforderungen

Die in den letzten Abschnitten eingeführten Werkzeuge und Paradigmen wurden einer Evaluation im Hinblick auf ihre Brauchbarkeit zur Implementierung des in dieser Arbeit vorgeschlagenen Grundszenarios unterzogen. Dabei wurde die verdichtete Anforderungsliste vom Anfang dieses Kapitels zugrunde gelegt. Die Ergebnisse sind tabellarisch zusammengefasst: In Abb. 21 werden die reinen Spezifikationsansätze beurteilt, in Abb. 22 die Ansätze und Werkzeuge, die auch Visualisierungen erstellen können.

Bezüglich der Schreibweise in Abb. 21 und 22 sei gesagt, dass wir fünf Einträge unterscheiden:

- Ein Haken bedeutet, dass die geforderte Eigenschaft implementiert ist. Beispiel: Flüssige Animationen sind implementiert.

- Ein eingeklammerter Haken bedeutet, dass die geforderte Eigenschaft konzeptionell sauber und leicht nachzuimplementieren ist, oder dass im Falle mehrerer Teileigenschaften nicht alle hundertprozentig implementiert sind. Beispiel: Das Verhalten ist in Abhängigkeit vom Zustand steuerbar, und es können Meldungen als Reaktionen auf Benutzeraktionen dargestellt werden, aber keine eigenen, ermunternden Meldungen konfiguriert werden.

- Ein eingeklammertes Kreuz bedeutet, dass die Eigenschaft zwar zu implementieren wäre, aber entweder mit großem Aufwand oder konzeptionell nicht im Sinne unseres Grundszenarios. Beispiel: Diejenigen Java-basierten Werkzeuge, die schon eine Schnittstelle für Layoutalgorithmen anbieten, können natürlich bezüglich Layoutalgorithmen konfiguriert werden. Eine Konfiguration in unserem Sinn liegt aber nur vor, wenn es dafür nicht notwendig ist, den Quellcode zu ändern, sondern ein im System vorhandener Layoutalgorithmus zur Laufzeit extern, zum Beispiel in einer XML-Datei, konfiguriert werden kann. Letztere Funktionalität könnte wohl in den meisten Fällen nachträglich eingebaut werden, das wäre aber ein erheblicher Aufwand.

	Formale Semantik der Spezifikation	Werkzeugunterstützung für Codegenerierung	Natürliche, erweiterbare, minimale Spezifikation einer Datenstruktur	Automatische Prüfung einfach zu spezifizierender Invarianten	Zufällige Szenarien	Flüssige Animationen	Explorative Lernszenarien für Datenstrukturen möglich	Exploratives Lernen von Algorithmen	Interaktivität möglich	Plattform-Unabhängigkeit der Laufzeitumgebung	Player mit Undo / Redo	Layoutalgorithmen möglich, vorhanden, erweiterbar	Kostenlose Laufzeit-Lizenzen	Funktionalität und Attribute können versteckt werden	Das Layout ist unabhängig von Datenstruktur und Algorithmus spezifizierbar	Entfernter Aufruf von Operationen, auch von gängigen Programmiersprachen aus möglich	Angezeigte Attribute einfach zu konfigurieren	Farbschemata als Reaktion auf Änderungen des inneren Zustands der Datenstruktur einfach zu konfigurieren
CASL	✓	✗	(✗)	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Z	✓	✗	✗	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
UML	✗	(✓)	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Abbildung 21: Vergleichstabelle der Spezifikationsansätze; mit 'UML' sind hauptsächlich UML-Klassendiagramme gemeint (siehe Text).

- Ein Kreuz bedeutet, dass die Eigenschaft nur mit höchstem Aufwand zu implementieren wäre oder nicht unterstützt wird. Beispiel: Ein System hat keine flüssigen Animationen vorgesehen.
- Ein waagrechter Strich bedeutet, dass die Eigenschaft in diesem Fall nicht anwendbar ist. Beispiel: Bei einer reinen Spezifikationssprache wie UML ist es bedeutungslos, zu fragen, ob sie flüssige Animationen bereitstellt.

Die Felder ohne Einträge in Abb. 22 rühren entweder daher, dass wir bisher keine Informationen über die betreffende Eigenschaft gewonnen haben, oder, dass es in Anbetracht der schon in der jeweiligen Zeile vorhandenen Kreuze nicht mehr lohnenswert war, zusätzliche Informationen anzugeben, da der Ansatz nicht mehr als interessant für unsere Zwecke eingestuft werden kann.

In der folgenden Aufzählung werden alle bisher vorgestellten Werkzeuge und Paradigmen kritisch auf die Nicht-Erfüllung von Anforderungen aus unserer kondensierten Liste untersucht.

- Textuelle Formen der Spezifikation:

  Die für die algebraische und logikbasierte Spezifikation entstandenen Notationen sind nicht unbedingt für den ungeübten Dozenten 'natürlich' zu nennen. Die leichte Erweiterbarkeit ist bei der algebraischen Spezifikation in stärkerem Maße gegeben, da die bestehenden Funktionen leichter genutzt werden können, um neue oder erweiterte Beziehungen zwischen den existierenden Typen aufzubauen, als das in den auf bestimmte Mengen bezogenen Formeln der logikbasierten Spezifikation der Fall ist.

  Alles in allem gibt es für die genannten Spezifikationssprachen eine eingeschränkte Werkzeugunterstützung (für Z siehe [ZWe05], für CASL [Mos]), die z. B. Hilfe bei der Eingabe von Spezifikationen für einfache Datenstrukturen und Algorithmen bietet. Durch die vorliegende formale Semantik kann bis zu einem gewissen Grad auch eine Unterstützung

	Formale Semantik der Spezifikation	Werkzeugunterstützung für Codegenerierung	Natürliche, erweiterbare, minimale Spez. einer Datenstruktur	Automatische Prüfung einfach zu spezifizierender Inv.	Zufällige Szenarien	Flüssige Animationen	Explorative Lernszenarien für Datenstrukturen möglich	Exploratives Lernen von Algorithmen	Interaktivität möglich	Plattform-Unabhängigkeit der Laufzeitumgebung	Player mit Undo / Redo	Layoutalgorithmen möglich, vorhanden, erweiterbar	Kostenlose Laufzeit-Lizenzen	Funktionalität und Attribute können versteckt werden	Das Layout ist unabhängig von Datenstruktur und Algorithmus spezifizierbar	Entfernter Aufruf von Operationen, auch von gängigen Programmiersprachen aus möglich	Angezeigte Attribute einfach zu konfigurieren	Farbschemata als Reaktion auf Änderungen des inneren Zustands der Datenstruktur einfach zu konfigurieren
PROGRES /JViews	✓	✓	(✓)	✓	✓	✗	✓	✗	✓	(✗)	✗	✓	✗	(✓)	✗	(✗)	✓	(✗)
Fujaba	✓	✓	(✓)	✗		✗			✓	✓	✓	(✓)	✓					
DiaGen	✓	✓	(✓)	(✗)	(✗)	✓	(✓)	✗	(✓)	✓	(✓)	(✗)	✓	(✓)	(✓)	(✗)	(✓)	(✗)
Zeus					✓		(✗)	(✗)	(✗)									
Animal	✓	(✓)	✗	✗		✓	(✓)	(✓)	(✓)	✓	✓	✗	✓		✗	✓		
GAWAIN		✗	✗		✓	✓	(✓)	(✓)	(✓)	✓	✓		✗			✓		
Halvis		✗	✗	(✗)		✓	✓	✓	✓		✓				✗	(✗)		
JCAT			✗		(✗)		✗	(✓)	(✓)	✓				✓				
AlgoViz		✗	✗	✗	(✗)	✓	(✓)	✓	✓	✓	(✗)	(✓)	✓	(✓)		✗		
SALA(BIM)	(✓)	✗	✗	(✓)		✓	(✓)	(✓)	✓	✓	✓		✓		✗	✗		
DDD		(✓)	✗	✗		(✗)		(✗)	(✗)	✗	(✓)	(✗)	✓			(✗)		
JEliot		(✓)	✗		✗					✓		✗		✗	✗	✗		
Javiva	(✓)	(✓)	✗					(✗)	(✓)	✓		(✓)				✗		(✗)
ZStep95		(✓)	✗			✓	✗	(✗)	(✓)		✓					✗		
AGD			✗	✗		✗			✓	✓	✓	✓	(✓)			(✗)		
yFiles			✗	✗		✓			✓	✓	(✓)	✓	✓			(✗)		
VisualGraph					✓		(✗)	(✓)	(✗)	✓		✗						
DANCE	✓	✓	✓	✗					(✗)		✓							
LENS	✓	✓	(✓)			✓	(✓)	✓	(✓)		✓	✗		✗	✗	✗		✗
Pavane	✓	(✓)	✗	(✓)			(✗)	✓	(✗)			✗			✗			(✓)
Leonardo			✗		✗	✓	✗	✓	(✓)	✗	✓	(✗)			✗			
PROVIDE			✗				✗		(✓)					(✓)				
Forms/3		✓	(✗)	✓			(✗)		(✓)		✓	(✗)		✗	✗	✗		
JawAA									(✗)	✓		(✗)		✗	✗	(✓)		
Stagecast		✓	✗			✓	✗	(✗)	✓	✓	✓	✗	(✓)			✗		
Simul8		✓				✓			(✓)	✗	(✓)	✗	✗					
Amulet		✓	✗			✓	(✓)	(✗)	✓	✓	(✓)	✗	✓		✗	✗		
Tilcon		✓				✓	(✓)	(✗)	✓	✗	(✓)	✓	✗		✗	✗		
Ganimal		✓	✗	(✓)		✓	✗	✓	✓	✓	✓	(✓)		✗	✓	(✗)	✗	✗
Alma	✓	(✓)	(✗)	✓				(✓)	(✗)			(✗)		✗	✗	(✗)		

Abbildung 22: Vergleichstabelle der Werkzeuge

beim Nachweis konkreter Eigenschaften eines in diesen Sprachen spezifizierten Systems gegeben werden. Allerdings kann man – abgesehen von weniger bekannten, ausführbaren Untermengen – nicht von einer Codegenerierung sprechen, die aber notwendig wäre, um ein von uns gefordertes Rahmenwerk auf diesen Konzepten aufsetzen zu können. Die genannten Gründe erklären wohl zum Teil, warum es nicht leicht ist, aktuelle Projekte zu finden, die mit algebraischen oder logikbasierten Spezifikationsmethoden erstellt worden sind.

- UML:

  Die Modellierung von Datenstrukturen mit Diagrammen der UML, also insbesondere Klassendiagrammen und Objektdiagrammen, ist wegen der verbreiteten Nutzung dieser Modellierungssprache sowohl den Dozenten, die Algorithmen oder Datenstrukturen unterrichten, als auch den Studierenden, die die entsprechenden Kurse besuchen, vertraut. Das heißt, es kann davon ausgegangen werden, dass für die Dozenten bei der Nutzung von UML kein Einarbeitungsaufwand nötig ist. Zusätzlich könnten bei einer Verwendung von UML für die Spezifikation die gleichen Diagramme in den Übungsaufgaben verwendet, von den Studierenden erstellt oder einfach als Illustrationen vorgegeben werden. Damit ist unsere grundlegende Forderung der „natürlichen und einfachen Spezifikation“ hier eindeutig gegeben.

  Ein großer Nachteil der genannten Diagrammarten der UML für unsere Zwecke ist ihre fehlende formal definierte Semantik. Auf diese Weise ist es nicht möglich, ohne Hinzunahme weiterer Konstrukte eine automatische Generierung von Code zu gewährleisten, zumal UML-Klassen- und Objektdiagramme besonders viele Freiheitsgrade bei der Interpretation zulassen. Wir sprechen also nicht von den ebenfalls zur UML gehörenden Statecharts, die zwar geeignet sind, exaktes Programmverhalten zu modellieren, aber eben nicht so geeignet zur Modellierung von Datenstrukturen.

- Graphtransformationssysteme:

  Durch das zugrunde liegende inhärent graphische Modell und die formal definierte Semantik können Graphtransformationen gut als graphische Spezifikationssprache für Datenstrukturen genutzt werden. Außerdem sind sie für die werkzeugunterstützte Codegenerierung geeignet. Dafür sind sie aber nicht so bekannt wie die UML und erfordern damit einen gewissen Einarbeitungsaufwand von Seiten des Dozenten, der unserer Meinung nach aber geringer ist als bei den beschriebenen textuellen Formen der Spezifikation. Letztere Aussage gilt verstärkt dann, wenn es gelingt, die Spezifikation mit Graphtransformationen an eine solche mit UML anzulehnen, worauf wir bei der Vorstellung unseres gewählten Ansatzes in Kapitel 5 genauer eingehen werden.

  - PROGRES überzeugt durch die Möglichkeit, sofort komplexe Datenstrukturen (als Graphen) definieren und mit einer visuellen Sprache manipulieren zu können. Ebenso existieren bereits Schnittstellen für den Aufruf von Operationen und die Rückgabe von Fehlermeldungen. Eine Schwäche ist die fehlende Möglichkeit, kontinuierliche Animationen darzustellen; allerdings existiert auch die Möglichkeit, neue Ausgaberoutinen, z. B. in Java, anzubinden. Wegen der darunterliegenden Graphdatenbank

'GRAS', die auf Unix und Linux läuft, bisher aber nicht auf Windows, ist die geforderte Plattformunabhängigkeit der Laufzeitumgebung nur durch eine Netzwerkinstallation oder die Nutzung einer virtuellen Maschine zu erreichen. PROGRES kann kostenlos eingesetzt werden, das ab Version 10 miteingesetzte JViews [JVi03] ist aber lizenzpflichtig.

  - Fujaba bietet schöne Möglichkeiten, Datenstrukturen inklusive Graphen und Bäume (ohne sortierte Kinder) zu spezifizieren, zu manipulieren und darzustellen, inklusive aller oben genannten Vorteile von Graphtransformationswerkzeugen. Durch die komplette Implementierung in Java ist auch die Forderung nach Plattformunabhängigkeit erfüllt. Es fehlen die Möglichkeiten für 'undo' / 'redo' und die Überprüfung von Invarianten. Laut mündlicher Erfahrungsberichte mangelte es der aktuellen Version zum Zeitpunkt unserer ersten Evaluation noch an Stabilität.
  - Wegen der konsequenten Nutzung der Programmiersprache Java bietet DiaGen die geforderte Plattformunabhängigkeit. Flüssige Animationen sind ebenso vorgesehen wie die Möglichkeit des 'undo' und 'redo'. Die Einbindung von Layoutalgorithmen ist über eine vorgesehene Schnittstelle möglich. Die einfache Spezifizierbarkeit und Nutzung von Invarianten ist im Moment nicht vorhanden, genauso wie die Erstellung zufälliger Szenarien. Der – auch externe – Aufruf von Schnittstellenoperationen ist möglich; für Studentenprogramme, die nicht in Java geschrieben werden sollen, müsste aber eine neue, externe Schnittstelle erstellt werden.

- Nicht-spezialisierte Animationssysteme für Algorithmen:

  - Zeus bietet nur eine eingeschränkte Interaktivität an. Die Erstellung neuer Animationen bedeutet einen erheblichen Aufwand. Hilfreich ist die Fähigkeit von Zeus, zufällige Daten generieren zu können, die als Eingabedaten verwertbar sind. Explorative Lernszenarien für Datenstrukturen sind wegen der eingeschränkten Interaktivität schwierig mit Zeus zu erstellen. Außerdem kann man gar nicht von einer Spezifikation in der Art unserer Erfordernisse sprechen, da die Ausgangsbasis für die Animationen mit Zeus eine Liste von 'interesting events' ist.
  - Da bei Animal das zum Bau von Listen vorgesehene graphische Grundprimitiv „Listenelement“ ein Kästchen mit bis zu 9 Pfeilen ist, könnten die meisten praktisch relevanten Graphen und Bäume zwar dargestellt werden, aber eigentlich ohne irgendein Wissen über die Datenstruktur von Seiten des Tools. Das Problem, bezogen auf unser Grundszenario, ist, dass sich auch die Interaktivität von Animal auf die Änderungen von Animationen auf primitiven graphischen Objekten bezieht, also ohne Wissen über Datenstrukturen oder Schnittstellenoperationen oder Layoutalgorithmen.

    Das zusätzlich angebotene Rahmenwerk ANIMAL-FARM bietet uns auch keinen geeigneten Rahmen, da unsere geforderten erforschbaren, hochinteraktiven Datenstruktur-Szenarien damit nicht beschrieben werden. Außerdem existiert bisher keine Implementierung einer Animationsumgebung mit den für unsere Zwecke wesentlichen in ANIMAL-FARM vorgesehenen Möglichkeiten.

- Ähnlich wie bei Animal ist in GAWAIN einer der Schwachpunkte in Bezug auf unsere Forderungen das fehlende Wissen um komplexe Datenstrukturen, was sich u. a. zusätzlich dahingehend auswirkt, dass keine Layoutalgorithmen für diese Datenstrukturen angeboten werden.

- Elektronische Bücher für die Lehre von Algorithmen und Datenstrukturen:

  Die grundsätzliche Problematik der EBooks, nämlich der geringe Abstraktionsgrad der Spezifikation und die für unsere Zwecke zu starke Kopplung von Spezifikation und Implementierung, wird uns bei den meisten Systemen dieses Abschnitts wieder begegnen.

  - Daraus folgte bei HalVis ein viel zu großer Aufwand für die Anpassung von existierenden Visualisierungen an neue, möglicherweise nur leicht veränderte Algorithmen oder Datenstrukturen.
  - Trotz eines gewissen Maßes an Interaktivität ist es in JCAT nicht möglich, die Eingabedaten für eine bestehende Animation zu ändern. Von einer natürlichen Spezifikationssprache kann man nicht sprechen. Zur Erzielung schöner Effekte in JCAT ist viel Code nötig.
  - Durch das Baukastenprinzip von AlgoViz kann zwar eine hohe Wiederverwendung von existierendem Code erreicht werden, aber bei kleinen Änderungen in der Definition einer Datenstruktur oder eines Algorithmus muss der Quellcode eines oder mehrerer betroffener 'Java Beans' geändert werden. Von einer abstrakten Spezifikationsmöglichkeit kann also gar nicht besprochen werden; damit ist unsere Idee der Code-Generierung mit AlgoViz nicht umsetzbar.
  - Das Rahmenwerk SALA gibt zwar viele Möglichkeiten der Animation vor, bietet bei der Erstellung der Spezifikation der Datenstruktur und des Algorithmus aber keine Werkzeugunterstützung; d. h. der Dozent muss eine Reihe von Java-Klassen implementieren, bevor erste Resultate sichtbar werden und getestet werden können. Auch die Konfigurationseinstellungen wie Fehlermeldungen in bestimmten Szenarien müssen in den Java-Quellcode aufgenommen werden, so dass bei jeder minimalen Änderung das ganze System neu übersetzt werden muss. Immerhin schafft es die mitgelieferte Bibliothek SALABIM, den Dozenten weitgehend vom Aufwand der Mehrfach-Implementierung zu befreien, und stellt außerdem einen einfachen Scheduler zur Verfügung, der das vom Dozentenprogramm angestoßene 'event handling' in die eigentliche Visualisierung umsetzt.

    Da SALA möglichst wenige Einschränkungen bei der Ausdrucksfähigkeit machen will, muss aber einiges an Layout- und sogar Swing-Komponenten-Informationen in den vom Dozenten für jede Aufgabe zu erstellenden Code einfließen. Die von uns geforderte Entkoppelung von Spezifikation und Layout und auch die mögliche einfache Art der Spezifikation ist damit nicht mehr gegeben. Auch die Erweiterbarkeit mit neuen Layoutalgorithmen, die zu flüssigen Animationen führen, ist sehr fraglich. Für unser Grundszenario problematisch ist die völlige Fokussierung auf Java, was dazu führt, dass unser Szenario des visuellen Debuggings zwar wahrscheinlich

durch eine Erweiterung von SALA für Java-Programme zu implementieren wäre, was aber z. B. für C++, Ada oder C Programme nicht unmittelbar möglich ist.

Insgesamt handelt es sich um ein System, das sein Ziel, nämlich die Erstellung interaktiver Animationen mit allen Möglichkeiten von Java, also auch mit so gut wie allen von uns erwähnten Oberflächen-orientierten Anforderungen an eine lernwirksame Visualisierungsumgebung, voll erreicht, allerdings zu dem Preis, Java-Code für graphische Objekte schreiben zu müssen.

Obwohl der Abstraktionsgrad der Spezifikation niedrig ist, konnte in SALA eine Minimierung des Dozentenaufwands für die Erstellung neuer Animationen erreicht werden. Wünschenswert wäre hier eine Werkzeugunterstützung, die zumindest Teile der jetzt von Hand zu schreibenden Java-Klassen generieren kann.

- Algorithmenanimation durch Code-Interpretation:

  - Ein Problem bei der Nutzung von DDD – bezogen auf unser Grundszenario – ist, dass sich der Lernende mit den Feinheiten von DDD und dem 'low level debugging' vertraut machen muss.

    Die Animationen von DDD sind nicht kontinuierlich. Da immer schrittweise durch den Code gegangen wird, ist es nicht möglich, direkt Schnittstellenoperationen aufzurufen oder gar die Visualisierung über eine externe Schnittstelle von eigenen Programmen der Studenten aus fernzusteuern. Es fehlen also einige grundsätzliche, für uns notwendige Funktionen in DDD.

  - Jeliot ist auf primitive Datentypen und Felder beschränkt. Ähnlich wie bei DDD ist der Abstraktionsgrad für unsere Zwecke zu gering.

  - Durch die Nutzung der Visualisierungs-Funktionen in Javiva kann eine einfache Art von Layoutalgorithmus simuliert werden, der allerdings eng mit dem Code verwoben ist, und damit in keiner Weise unsere Forderung nach der Unabhängigkeit von Layout und Spezifikation der Datenstruktur erfüllt. Überhaupt sind in diesem Ansatz die Spezifikation von Datenstruktur, Algorithmus und Layout, wie es bei Systemen der Code-Interpretation bis zu einem gewissen Grad zwangsläufig der Fall ist, untrennbar ineinander verflochten. Diese Tatsache macht eine natürliche und einfach erweiterbare Spezifikation neuer Beispiele nicht im geforderten Maß möglich.

    Die Möglichkeit, ergänzend zu Fehlermeldungen auch ermunternde Meldungen auszugeben, ist gemäß Beschreibung nicht vorgesehen.

    Dieser Ansatz bietet für einen gewissen Anwendungszweck einen hohen Grad an Interaktivität durch die Anforderung an die Studierenden, neben dem Bau des eigentlichen Java-Programms gleich noch Vor- und Nachbedingungen anzugeben, deren Erfüllung automatisiert überprüft wird. Durch den extrem speziellen Anwendungsfall bietet sich hier allerdings keine Möglichkeit, Javiva als Werkzeugkomponente für den Bau eines von uns geforderten vielfältigen Simulationssystems für die Lehre von Datenstrukturen zu nutzen, da z. B. keine Unterstützung für Vorhersagen der Lernenden angeboten wird.

  - Die von ZStep95 angebotene statische Algorithmenanimation erreicht nicht den Grad an Interaktivität, der heute unter Algorithmenanimation verstanden wird. Da wir aber noch höhere Anforderungen an die interaktiven Fähigkeiten des entstehenden Systems stellen, kommt auch ZStep95 für uns nicht in Frage.

  Ein weiteres Manko all derjenigen Systeme für Code-Interpretation, die auf eine bestimmte Programmiersprache festgelegt sind, ist natürlich die fehlende Möglichkeit ihrer Nutzung in der Lehre der Datenstrukturen und Algorithmen einer anderen Programmiersprache bezüglich unserer Forderung nach einer Fernsteuerungs-Möglichkeit von Studentenprogrammen aus.

- API-basierte Graphenbibliotheken:

  Bei den Werkzeugen dieser Kategorie hängt die Brauchbarkeit zu einem großen Teil von der Nutzung durch andere Werkzeuge ab. Die Werkzeuge an sich können also nur schwierig in Bezug auf Themen wie Spezifizierbarkeit, Interaktivität etc. beurteilt werden.

  Im direkten Vergleich fiel auf, dass bezüglich Animation AGD keine flüssige Animation darstellen konnte. Bei yFiles fielen keine zusätzlichen Einschränkungen zu den in der Kategorie genannten generellen auf.

  VisualGraph bietet bereits inhärent eine gewisse Interaktivität an und ist dabei auf die Möglichkeiten von Animal angewiesen, die wie beschrieben zwar mächtiger als die einer reinen Graphen-Bibliothek sind, aber eben doch zu beschränkt für unsere Zwecke.

- Erzeugung von Animationen durch direkte Manipulation:

  - DANCE bietet eine natürliche Spezifikationssprache mit eingebauter Codegenerierung. Durch den Ansatz der Extrapolation wurde in DANCE der Fokus auf die schnelle Erzeugung von Animationen erstellt, was hier auf Kosten der Mächtigkeit der Spezifikationssprache geht. Einfach zu spezifizierende Invarianten oder ein generierter Code, der einfach bezüglich eigener Fehlermeldungen konfiguriert werden kann, sind damit nur schwer möglich.
  - Bezogen auf unsere Anforderungen mangelt es LENS an Farbschemata für die Darstellung von besonderen und Fehler-Zuständen. Auch das Verstecken unwichtiger Details des Algorithmus ist nicht möglich – das heißt, es werden immer alle verfügbaren Attribute von Objekten angezeigt. Layout-Informationen müssen ins LENS explizit als Formeln zwischen den Koordinaten der beteiligten graphischen Primitive spezifiziert werden und sind damit zusammen mit den anderen Teilen der Spezifikation vermischt. LENS bietet einen hohen Grad an Interaktivität vom Standpunkt eines Studierenden aus, der selbst eine Animation baut. Allerdings steht bei LENS – ähnlich wie bei DANCE – der Prozess der *Erstellung* der Spezifikation im Vordergrund, und weniger die für uns entscheidenden Fähigkeiten der Spezifikationssprache und der Visualisierungsumgebung.

- Deklarative Systeme für die Animation von Algorithmen:

  Da es beim deklarativen Ansatz – je nach verwendeter Syntax und der Größe der entstehenden textuellen Deklarationen – nicht unbedingt selbstverständlich für Dozenten von Algorithmen und Datenstrukturen ist, die textuell geschriebenen Prädikate sofort lesen und geeignet einsetzen zu können, ist die Bedingung der natürlichen oder einfachen Spezifikationssprache nur eingeschränkt gegeben.

  - In Pavane ist durch die Notwendigkeit, sowohl die Spezifikation der Datenstruktur und des Algorithmus als auch Layoutalgorithmen deklarativ zu spezifizieren, weder die Möglichkeit der natürlichen, einfachen Spezifizierbarkeit, noch die erforderliche Trennung der Spezifikation von Datenstruktur und Layout gegeben. Hinzu kommt, dass durch die Beschränkung auf PROLOG und C heute interessante Programmiersprachen wie Java oder C++ nicht sinnvoll durch Animationen unterstützt werden können.

    Trotz der vorhandenen einfachen Interaktivität dürften die geforderten interaktiv-erforschbaren Datenstruktur-Szenarien nicht ohne sehr große Änderungen im System machbar sein.

  Ein generelles Problem des deklarativen Ansatzes im Zusammenhang mit Animationen ist die datengetriebene statt ereignisgetriebene Arbeitsweise. Denn durch die einmal festgelegte Relation zwischen Daten und Visualisierung wird jede Änderung einer einmal als interessant eingestuften Eigenschaft der Datenstruktur sofort visualisiert, was den Fokus des Lernenden auf sich zieht, ob es sich um eine wesentliche Änderung handelt oder nicht. Und jede einmal als unwichtig eingestufte Eigenschaft wird nicht visualisiert, auch wenn es sich ausnahmsweise doch um ein wesentliches Ereignis handelt. Zum Beispiel können manche Änderungen eines Variablenwertes wichtig sein, während die meisten anderen unwichtig sind.

  - Leonardo versucht dieses Problem zu umgehen, indem ein virtueller 'event manager' entscheidet, welche Ereignisse wichtig sind und welche nicht, und mit einer virtuellen CPU kommuniziert, die im Bedarfsfall die notwendigen Prädikate wieder auswertet. Explorative Datenstruktur-Szenarien sind in Leonardo wohl nicht zu implementieren. Im Moment ist Leonardo nur auf MacOS verfügbar, was es leider für unsere Zwecke unbrauchbar macht.
  - Trotz seines interessanten Ansatzes ist gerade der Automatismus in der Visualisierung in PROVIDE ein Problem für unser Grundszenario, da Informationen nachträglich nicht zu verstecken sind, ganz abgesehen von der fehlenden Möglichkeit, explorative Datenstruktur-Szenarien verarbeiten zu können.
  - Die in Forms/3 erreichbare einfache Interaktivität wird unseren Anforderungen bisher bei weitem nicht gerecht. Forms/3 wurde beispielhaft für die Animation eines einfachen Algorithmus (Selectionsort) eingesetzt. Da das System für andere und allgemeinere Zwecke als für Algorithmenanimation entwickelt wurde, kann es bisher nur mühsam zur Animation eines Algorithmus gebracht werden. Dazu kommt, dass

die Spezifikation eines einfachen Algorithmus hier mehr Abstraktion erfordert, als dem Nutzer einer Visualisierungsumgebung, der mit wenig Aufwand eine Animation erstellen will, zuzumuten ist.

Im Moment prüfen die Autoren die Mächtigkeit dieses Ansatzes, der ja gar nicht für die Animation von Datenstrukturen oder Algorithmen entwickelt wurde. Unseres Wissens wurden noch keine auf Bäumen basierten Algorithmen implementiert. Layoutalgorithmen sind mit diesem Ansatz schwierig zu implementieren, ähnlich wie interaktives Aufrufen von Operationen. Auch die Möglichkeit, vorhandene Informationen zu verstecken, müsste erst noch entworfen und realisiert werden.

- Skriptbasierte Algorithmenanimation:

  Abgesehen von der fast zwangsläufigen Vermischung von Datenstruktur-Spezifikation und Layout, gibt es in JAWAA im Moment keine Layoutalgorithmen für Graphen und Bäume. Damit ist es für unseren Ansatz unbrauchbar.

- Visuelle, regelbasierte Entwicklungsumgebung:

  Stagecast ist nicht fähig, Datenstrukturen wie Bäume oder Graphen in einem vertretbaren Aufwand darzustellen, da es auf die unabhängige Bewegung einzelner Graphikobjekte ausgelegt ist. Schnittstellen für einen Aufruf von Operationen auf der Datenstruktur oder eine API fehlen ebenfalls und können nicht ergänzt werden, da dies kein open-source Produkt ist.

- Icon-basierte Simulationssprache:

  Simul8 ist im Moment nur auf Windows verfügbar und kostenpflichtig, wenn auch vergleichsweise günstig. Da keine gängigen Bibliotheken zugrunde liegen, ist eine eigene Erweiterung der Funktionalität nicht möglich.

- Entwicklungsumgebungen für graphische Oberflächen oder Benutzerschnittstellen:

  - In Amulet müssten viele Interaktionen erst programmiert werden, wie z. B. ein 'undo'-Mechanismus in den 'common objects' oder ein Anhalten von Animationsfilmen über ein vorher zu definierendes Ereignis. Animationen entstehen über die Veränderung von Eigenschaften der Objekte. Leider findet eine Unterstützung von 'trees and graphs in the future' statt.
  - Tilcon ist nicht frei verfügbar und nicht auf Linux erhältlich. Der entfernte Aufruf von Operationen von anderen Programmiersprachen aus dürfte laut Beschreibung nicht nur, wie in den meisten hier betrachteten Werkzeugen, zusätzlich, sondern wahrscheinlich gar nicht implementierbar sein.

- Sonstige Ansätze für die Erstellung von Animationen:

  - Ganimal wurde nicht für die Animation von Datenstrukturen entworfen und enthält keine Layoutalgorithmen; ebenso sind explorative Szenarien auf Datenstrukturen nicht auf natürliche und einfache Art spezifizierbar, wohingegen interaktive

Szenarien, die sich auf Abläufe konzentrieren, also z. B. Algorithmen auf einfachen Datenstrukturen, relativ schnell entwickelt werden können.

Ebenso unbefriedigend wie das Fehlen hochgradiger Interaktivität sind die fehlenden Möglichkeiten der Fernsteuerung von Programmen, die nicht in Java geschrieben sind, über eine frei konfigurierbare Menge von Schnittstellenoperationen. Es besteht die Möglichkeit, geschriebene Funktionen zur Laufzeit zu verstecken; ob das aber auch für Attribute gilt, ist aus den Veröffentlichungen, soweit sie uns vorliegen, nicht zu ersehen. Ebenso ist die Möglichkeit, Farbschemata an Zustände der Datenstruktur zu koppeln, nicht einfach zu konfigurieren.

Alles in allem lässt sich zu Ganimal sagen, dass hier der Fokus eindeutig auf der Erzeugung von Abläufen liegt und sich das Werkzeug deshalb nicht unmittelbar für die Erstellung interaktiver Datenstruktur-Szenarien eignet. Wegen der offenen Schnittstellen und der durchgängigen Nutzung von Java wäre eine Implementierung der fehlenden Möglichkeiten zwar in irgendeiner Form möglich, aber eben individuell auf das jeweilige Beispiel zugeschnitten und – ohne Entwicklung eines zusätzlichen Konzeptes – nicht inhärent vom Ansatz unterstützt.

– In Alma ist es, abgesehen von der Notwendigkeit, zunächst eine geeignete Sprache für die Spezifikation von Algorithmen und Datenstrukturen aufzusetzen, im Moment noch notwendig, Teile der Werkzeugunterstützung manuell zu implementieren. Deshalb erscheint uns der Einsatz von Alma trotz der verlockend hohen Abstraktion der Eingabe-Sprache für unsere Zwecke als nicht zielführend. Unklar ist es zur Zeit auch, wie und ob der Grad an Interaktivität zu erreichen ist, den wir in der Anforderungsliste fordern. Generelle Vorbehalte gibt es bei den Oberflächen-bezogenen Anforderungen: Layoutalgorithmen, Farbschemata und Meldungen zum richtigen Zeitpunkt müssten – im Falle einer entsprechend offenen Schnittstelle – von außen zusätzlich hinzugefügt werden. Auch Aufrufe über eine z. B. von C++ aus zugängliche API gehen über die momentanen Fähigkeiten von Alma hinaus.

Wie man sieht, erfüllt keiner der beschriebenen Ansätze und Werkzeuge alleine unsere sowieso schon gekürzte Anforderungsliste. Bei näherer Betrachtung gibt es allerdings eine vielversprechende Kombination zweier Kandidaten: Eine kombinierte Nutzung von PROGRES und yFiles kann die genannten Anforderungen zu einem hohen Grad erfüllen:

Die Stärken von PROGRES liegen im Bereich der Spezifikation, der Codegenerierung und der angebotenen Schnittstellen. Außerdem kann durch Nutzung der Graphdatenbank GRAS nicht nur ein beliebiges 'undo' und 'redo' erreicht werden, sondern es können z. B. zufällige Szenarien und Eingabe-Datenmengen erzeugt werden.

Die Stärken von yFiles liegen im Bereich der Visualisierung. Flüssige Animationen, mächtige Layoutalgorithmen und Layoutmorpher und die Plattformunabhängigkeit durch die reine Basierung auf Java ermöglichen anspruchsvolle Animationen. Im Gegensatz zu anderen mächtigen Visualisierungsbibliotheken, wie dem bei PROGRES mitgelieferten JViews, sind die Laufzeit-Lizenzen von yFiles im Rahmen der universitären Lehre frei.

Fasst man die Kombination von PROGRES und yFiles ins Auge, gibt es noch zwei Punkte zu bedenken:

Erstens erfüllt PROGRES unsere ganz zentrale Forderung nach einer natürlichen, erweiterbaren und einfachen Spezifikationssprache in einem hohen Maße, ist aber kein so bekanntes Konzept wie die UML. UML als abstrakter Ansatz ohne formale Semantik kann unsere Werkzeug-spezifischen Forderungen nicht erfüllen, wird aber weiträumig in der Lehre eingesetzt und ist bei Dozenten wie bei Studierenden wohlbekannt. Wir werden in Kapitel 5 zeigen, dass es eine sehr naheliegende Art gibt, UML-Klassen- und -Objektdiagramme und die in PROGRES möglichen Graphtransformationen auf eine Art zu kombinieren, die die Vorteile beider Ansätze in sich vereint.

Zweitens gibt es einige wenige Punkte, die weder von PROGRES noch von yFiles auf zufriedenstellende Art erfüllt werden, besonders der Punkt des entfernten Aufrufs von Operationen von verschiedenen Programmiersprachen aus fällt da ins Auge. Hier zeigt es sich, dass beide Werkzeuge in Kombination relativ leicht erweitert werden können, da PROGRES eine Java-Schnittstelle für Aufrufe und Ereignisse anbietet und yFiles sowieso komplett Java-basiert ist.

4.6. Fazit

Bei einer ausführlichen Evaluation einer Reihe von Paradigmen und Werkzeugen für die Spezifikation und Visualisierung von Algorithmen und Datenstrukturen zeigte es sich, dass kein Ansatz alleine unsere wichtigsten Forderungen erfüllen kann. Deshalb haben wir uns dafür entschieden, zur Realisierung unseres beschriebenen Grundszenarios UML, PROGRES und yFiles auf neuartige Art miteinander zu kombinieren. In Kapitel 5 werden zunächst vier Datenstrukturen und Algorithmen vorgestellt, auf die wir im Rest dieser Arbeit immer wieder als konkrete Beispiele zurückkommen werden, bevor unser Ansatz überblicksartig beschrieben und eingeführt wird.

5. Überblick über VIDEA

Dieses Kapitel gibt einen Überblick über das in dieser Arbeit vorgestellte Konzept VIDEA. VIDEA steht für ***V**isual **I**nteractive **D**ata Structure **E**nvironment for **A**nimations* und umfasst einerseits ein Konzept der Spezifikation, Generierung und Visualisierung interaktiver Lernumgebungen für Algorithmen und Datenstrukturen, das aus den Erkenntnissen und Forderungen der bisherigen Kapitel entstanden ist und das im folgenden Teil dieser Arbeit detaillierter als bisher beschrieben wird. Andererseits steht der Name VIDEA auch für die im VIDEA-Konzept genutzte und im Rahmen dieser Arbeit vorgestellte Java-basierte Visualisierungsumgebung, die wiederum PROGRES und yFiles einbindet.

Um im Folgenden besser mit konkreten Beispielen arbeiten zu können, stellen wir zu Beginn dieses Kapitels zunächst vier typische Datenstrukturen und Algorithmen vor, die bekanntermaßen in der Lehre genutzt werden. Im zweiten Teil dieses Kapitels wird dann – teils unter Nutzung dieser Algorithmen und Datenstrukturen – ein Überblick über VIDEA gegeben.

5.1. Vier beispielhafte Algorithmen und Datenstrukturen

In den folgenden vier Unterabschnitten werden vier Beispiele von in der Lehre häufig genutzten Algorithmen und Datenstrukturen vorgestellt, nämlich

- AVL-Bäume,
- die doppelt verkettete (Ring-)Liste,
- der Dijkstra-Algorithmus zur kürzesten Wegesuche
- und der Kruskal-Algorithmus zum Finden minimaler Spannbäume.

Neben der Bekanntheit spielte bei der Auswahl dieser Algorithmen und Datenstrukturen primär die Erreichung einer möglichst großen Bandbreite an Anforderungen an die Visualisierungsumgebung eine Rolle. Diese Anforderungen sind aus den folgenden Gründen erheblich:

- Es liegen unterschiedlich komplexe Datenstrukturen vor. Zum Beispiel sind die AVL-Bäume um einiges komplexer als die Listen.
- Es liegen unterschiedlich komplexe Algorithmen vor: Das Einfügen in eine sortierte Liste ist algorithmisch wesentlich einfacher als etwa der Dijkstra-Algorithmus.
- Es liegen Operationen in unterschiedlicher Komplexität vor. Zum Beispiel ist ein Schritt des Dijkstra komplexer als eine Rotationsoperation des AVL-Baums, während das Einhängen in eine Liste einfacher ist.
- Es sind verschiedene Layoutalgorithmen für Graphen, Ringe, sortierte Bäume etc. gefordert.
- Es werden flüssige und statische Animationen gebraucht: Wir animieren unsere Beispiel-Instanzen für Liste und AVL-Baum flüssig, wohingegen es bei den Graphen-Algorithmen, wie wir zeigen werden, gerade darauf ankommt, dass das Layout statisch bleibt.
- Es kommen attributierte Kanten und getypte Kanten vor. So nutzen Dijkstra und Kruskal mit Zahlen markierte Kanten, während es z. B. beim AVL-Baum nur 'left' und 'right' gibt.

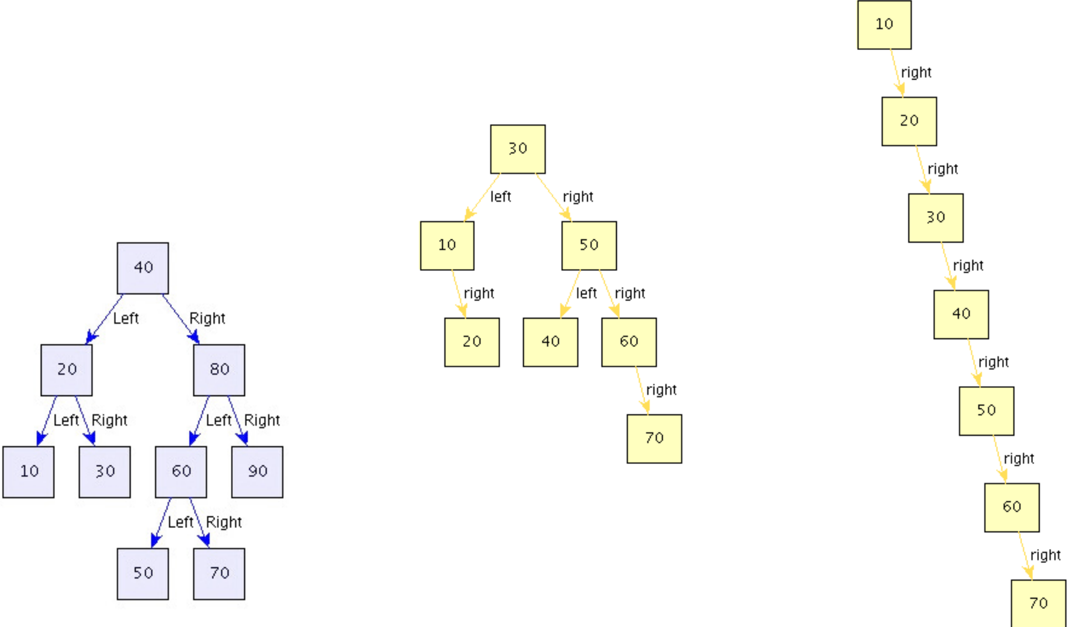


Abbildung 23: Links: Ein AVL-Baum in einer üblichen Darstellung. Mitte: Maximal entarteter AVL-Baum. Rechts: Maximal entarteter Binärbaum.

5.1.1. AVL-Bäume

Als erstes und am häufigsten in dieser Arbeit genutztes Beispiel für eine genügend interessante Datenstruktur stellen wir den schon erwähnten AVL-Baum vor. AVL-Bäume sind spezielle binäre Suchbäume, die dafür verwendet werden, Mengen von Daten-Elementen zu speichern. Auf der Menge dieser Elemente – genauer gesagt auf einem ausgezeichneten Attribut dieser Elemente – muss eine Totalordnung existieren, die eine Sortierung der Elemente ermöglicht. Abb. 23 links zeigt exemplarisch einen AVL-Baum aus Elementen mit nur einem ganzzahligen Attribut, das in der Mitte jedes Knotens dargestellt wird.

Die Besonderheit von AVL-Bäumen im Unterschied zu normalen Suchbäumen liegt darin, dass für AVL-Bäume permanent eine Mindest-Balancierung garantiert wird. Genauer gesagt, dürfen sich die Höhen je zweier Unterbäume eines beliebigen Knotens nicht um mehr als eins unterscheiden. Der Grund hierfür ist, dass durch diese vernachlässigbare Unsymmetrie des Baums zu jedem Zeitpunkt eine Einfüge-Operation in den Baum ebenso wie die Suche nach einem bestimmten Element im Schnitt wesentlich schneller möglich ist als bei einem gewöhnlichen Suchbaum, nämlich mit logarithmischem Aufwand verglichen mit linearem Aufwand beim entarteten binären Suchbaum. Abb. 23 zeigt in der Mitte einen maximal entarteten AVL-Baum, rechts einen maximal entarteten Suchbaum ohne die AVL-Baum-Eigenschaft, der die gleichen Elemente enthält.

Erkauft wird dieser Vorteil der AVL-Bäume durch den Verwaltungsaufwand, der notwendig ist, um den Baum balanciert zu halten. Technisch wird diese Balancierung bewerkstelligt, indem nach der jeweiligen potentiell balancegefährdenden Operation (in Frage kommen hier

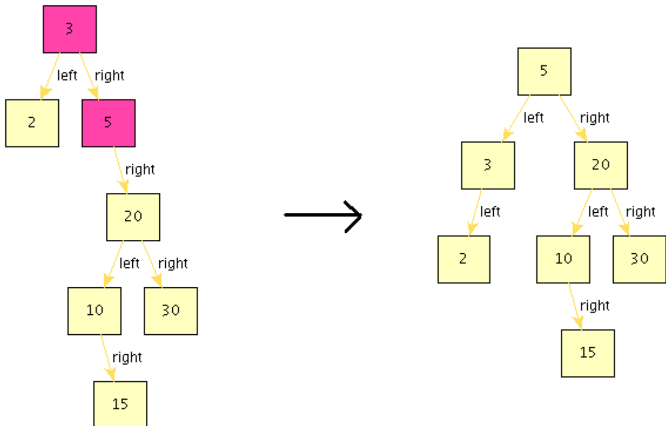


Abbildung 24: Links: Unbalancierter Baum. Rechts: Nach Linksrotation um die Wurzel gültiger AVL-Baum

nur das Einfügen eines neuen Elements oder das Löschen eines vorhandenen) eine so genannte *Rotation* ausgeführt wird. Rotationen in AVL-Bäumen erhalten die Sortierungseigenschaft des Binärbaums und korrigieren lediglich die Balance des Baums um eine Einheit in die gewünschte Richtung, also nach links oder rechts. Somit gibt es Links- und Rechtsrotation; eine zusätzliche Unterscheidung wird gemacht bezüglich einfachen und Doppelrotationen, worauf wir hier jedoch nicht näher eingehen wollen, da uns die Linksrotation exemplarisch völlig ausreichen wird. Eine detailliertere Beschreibung von AVL-Bäumen findet der interessierte Leser in fast jedem Datenstruktur-Lehrbuch und in vielen Lehrveranstaltungen, erwähnt seien hier [Bud01, Ein05a].

Jede dieser Rotationsoperationen benötigt in imperativen Programmiersprachen nur eine konstante Zahl von Zeiger-Zuweisungen. Nach dem Einfügen eines neuen Elements in den Baum ist höchstens eine Rotation notwendig, während das Löschen eines Elementes sich evtl. durch den Baum weiterpropagiert und so höchstens zu logarithmischem Aufwand relativ zur Menge der gespeicherten Elemente im Baum führen kann. Abb. 24 zeigt ein Beispiel für eine Linksrotation, die einen vorher unbalancierten Baum wieder zu einem AVL-Baum macht.

In Abb. 25 ist das abstrakte Schema der Linksrotation dargestellt, das zeigt, was mit den einzelnen Knoten im allgemeinen Fall passiert. Im Bild sind nur die beteiligten Knoten gezeigt, das heißt, es handelt sich hier um eine Stelle im gesamten Baum, die nicht notwendigerweise die Wurzel sein muss. Alle eingehenden und ausgehenden Kanten, die nicht eingezeichnet sind, bleiben dann im Falle einer Linksrotation unverändert, ausgenommen einer evtl. eingehenden Kante in die Wurzel des links gezeichneten Teilbaums: Diese würde so umgehängt werden, dass sie in die neue Wurzel des rechts gezeichneten Teilbaums eingeht. Da sich auch zwischen den beteiligten Knoten nicht alle Kanten ändern, wurde mit drei Pfeilen gezeigt, wohin sich die drei eingezeichneten Gruppen von Knoten bewegen. Damit die Linksrotation überhaupt erlaubt ist, werden zumindest die zwei oberen Knoten in den beiden umschließenden Ellipsen als existent vorausgesetzt. In manchen Beschreibungen kommt sogar noch die Existenz des auf der linken

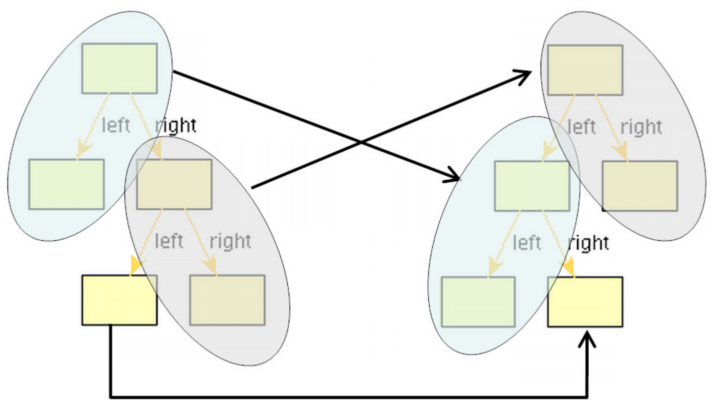


Abbildung 25: Die Linksrotation auf dem AVL-Baum in einer schematischen Darstellung

Seite des Bildes sich ganz rechts unten befindliche Knotens als Voraussetzung für eine Linksrotation hinzu. Wir werden bei den weiter unten vorgestellten visuellen Spezifikationen beide Varianten nutzen.

Die Höhe eines AVL-Baums definieren wir als die maximale Anzahl der besuchten Knoten auf dem Weg von der Wurzel zu einem Blatt minus eins. Beim leeren Baum sind das also -1, beim Baum in Abb. 24 rechts sind es drei. Der Balancefaktor eines beliebigen Knotens sei die Höhe des linken Teilbaums minus der Höhe des rechten Teilbaums; das ergibt bei der Wurzel des Baumes aus Abb. 24 rechts den Balancefaktor -1. Damit kann ein AVL-Baum also auch beschrieben werden als ein binärer Suchbaum, dessen Knoten alle einen Balancefaktor von -1, 0 oder 1 haben.

5.1.2. Doppelt verkettete Liste

Eine grundlegende Datenstruktur, die ebenso wie der AVL-Baum häufig in Datenstruktur-Büchern eingeführt wird (siehe z. B. [Wir83]) oder zumindest zur Erklärung anderer Konstrukte genutzt wird, ist die Liste.

Eine einfache Liste besteht aus Elementen, die jeweils durch *next*-Zeiger miteinander verbunden sind und dadurch in eine gerichtete Nachfolgebeziehung gebracht werden. Abb. 26 zeigt eine solche einfach verkettete Liste, hier gleich mit Listenkopf, also einem ausgezeichneten Ele-

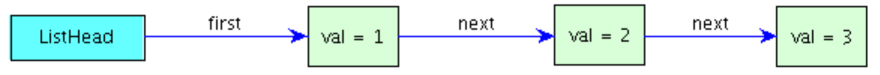


Abbildung 26: Eine einfache, verkettete Liste mit Listenkopf

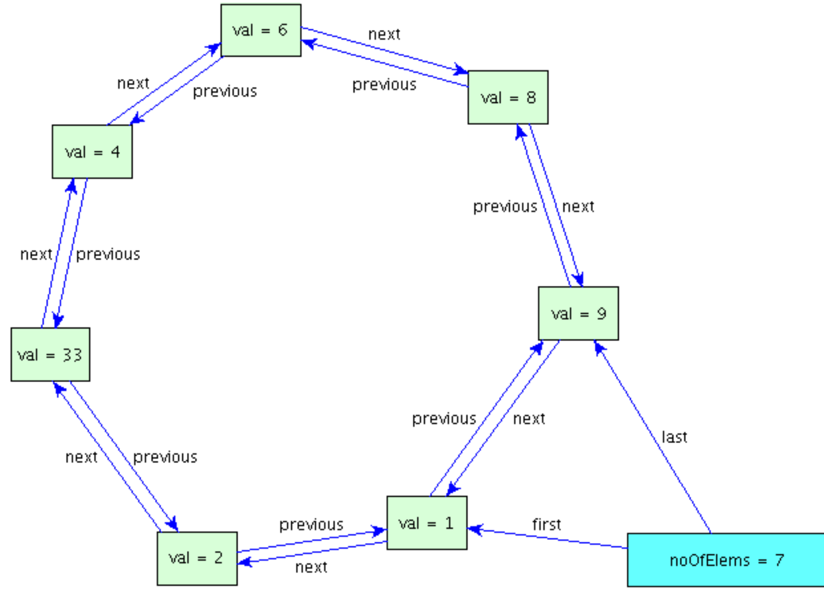


Abbildung 27: Eine doppelt verkettete Ringliste mit Listenkopf

ment, das durch eine *first*-Kante einen direkten Zugang zum eigentlichen ersten Element der Liste ermöglicht. Wir nutzen im Folgenden eine etwas komplexere Form der Liste, nämlich mit einer zusätzlichen Rückwärtsverkettung durch *previous*-Zeiger und mit einem optionalen *last*-Zeiger vom Listenkopf auf das letzte Listenelement. Im Falle keines vorhandenen Listenelements gibt es natürlich weder ein erstes noch ein letztes und somit auch keine *first*- oder *last*-Kante. Dann handelt es sich um die leere Liste, die durch den alleinstehenden Listenkopf repräsentiert wird. Zusätzlich wollen wir die Ringform erlauben, was bedeutet, dass vom letzten Listenelement wieder ein *next*-Zeiger auf das erste Listenelement zeigen kann, genau wie vom ersten Listenelement ein *previous*-Zeiger wieder auf das letzte Element zeigen kann. Man kann also zwei Typen von Listen-Objekten unterscheiden: Den Listenkopf und die Listenelemente.

Um die Datenstruktur noch ein bisschen interessanter zu machen, werden wir in der zugehörigen VIDEA-Instanz eine Variante der beschriebenen Liste nutzen, deren Elemente zusätzlich zum bisher Gesagten ein ganzzahliges *val*-Attribut besitzen, nach dem die Listenelemente vom ersten zum letzten aufsteigend sortiert sein sollen. Abb. 27 zeigt ein Beispiel für eine solche Liste, wobei der Wert 33 hier nicht korrekt einsortiert ist.

5.1.3. Der Dijkstra-Algorithmus zur kürzesten Wegesuche

Der Dijkstra-Algorithmus (siehe z. B. [Ein05b, Güt92]) ist ein bekannter Algorithmus zum Finden kürzester Wege in einem Graphen. Ein anderer bekannter Vertreter der so genannten 'shortest path'-Algorithmen ist der Floyd-Warshall-Algorithmus (siehe ebenso [Ein05c, Güt92]). Im direkten Vergleich ist Dijkstra dabei meistens schneller, findet aber die Wege nur von einem gegebenen Startknoten aus und setzt nicht-negative Kanten-Gewichte voraus.

Die Knoten des Graphen sind als durch irgendeine Markierung unterscheidbar vorausgesetzt, wohingegen die Kanten mit Kantengewichten beschriftet sind. Ein kürzester Weg zu einem gegebenen Knoten des Graphen ist dann eine Folge von besuchten Kanten auf dem Weg vom Startknoten zu diesem Knoten, dessen summierte Kantengewichte eine zumindest nicht größere Weglänge ergeben als alle anderen möglichen Wege zwischen dem Startknoten und dem betrachteten Knoten. Abb. 28 zeigt einen möglichen Beispiel-Graphen. Der Startknoten dieses Graphen sei derjenige mit *ID*=1. Der Dijkstra-Algorithmus arbeitet folgendermaßen:

- Vom gegebenen Startknoten S aus werden kürzeste Wege zu allen anderen Knoten berechnet und nebenbei der kürzeste Pfad von S zu jedem Knoten.
- Die Menge der Knoten des Graphen zerfällt zu jedem Zeitpunkt in 3 Teilmengen, die durch 3 Farben dargestellt werden:

  1. Die Knoten, die noch nicht betrachtet wurden, sind weiß. Da dies am Anfang die gesamte Menge ist, werden alle Knoten weiß initialisiert.
  2. Die Knoten, für die es schon eine Schätzung gibt, werden grau eingefärbt.
  3. Die Knoten, zu denen bereits der von S aus kürzeste Weg gefunden wurde, werden schwarz eingefärbt.

- Alle grauen Knoten sind zu jedem Zeitpunkt in einer Prioritäts-Warteschlange gespeichert, immer aufsteigend sortiert nach der bisher bekannten minimalen Distanz vom Startknoten S.
- Zu Beginn werden alle Knoten mit quasi-unendlicher Distanz initialisiert und nur die Distanz des Startknotens wird auf 0 gesetzt; seine Farbe wird als grau vorbelegt. Schließlich wird S in die Prioritäts-Warteschlange gestellt.
- In jedem Schritt des Algorithmus passiert Folgendes:
  Der graue Knoten mit der kürzesten bekannten Distanz, im Folgenden U genannt, wird aus der Warteschlange entnommen und schwarz eingefärbt. Gleichzeitig werden alle Ziele V auslaufender Kanten von U besucht mit den folgenden Aktionen:

  – Ist V weiß, wird er grau eingefärbt und in die Prioritäts-Warteschlange übernommen.
  – War oder wurde V nun grau, wird geprüft, ob seine bisher bekannte Distanz durch die neue Kante verbessert werden kann; wenn ja, wird die Distanz angepasst und U als Vorläuferknoten von V eingetragen.

- Das passiert solange, bis entweder alle Knoten bearbeitet (also schwarz) sind, so dass die Prioritäts-Warteschlange leer ist, oder bis ein bestimmter Knoten, zu dem die kürzeste Distanz gesucht war, bearbeitet ist.

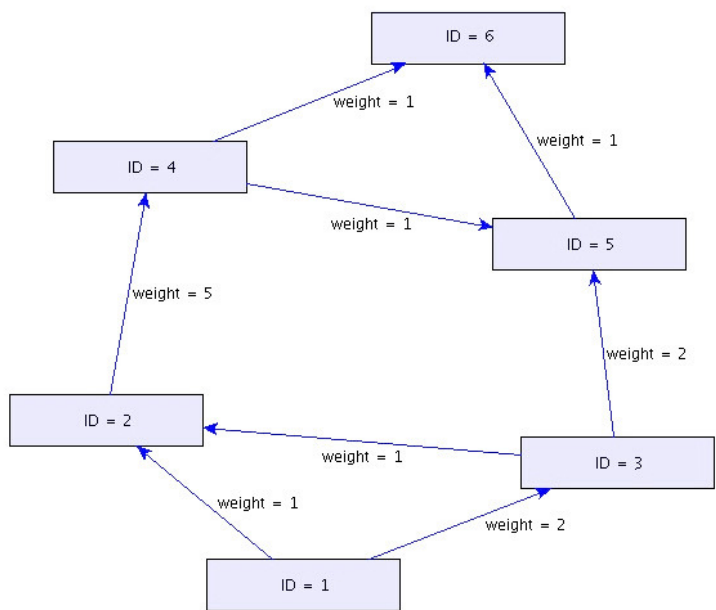


Abbildung 28: Beispiel-Graph für die Anwendung des Dijkstra-Algorithmus zur kürzesten Wegesuche

Auf Abb. 28 bezogen heißt das, dass im ersten Schritt Knoten 1 in die Prioritätswarteschlange gelegt wird, mit vorläufiger Distanz 0 markiert wird und grau eingefärbt wird. Im zweiten Schritt wird Knoten 1 als neues 'U' aus der Prioritätswarteschlange entnommen und schwarz eingefärbt. Alle unmittelbaren Nachfolgerknoten, also in unserem Beispiel die Knoten 2 und 3, werden grau eingefärbt und in die Prioritätswarteschlange gestellt. Dann wird ihre neue geschätzte Entfernung vom Startknoten berechnet. Da beide Knoten mit unendlicher Distanz initialisiert wurden, ergibt sich für den Knoten 2 die Verbesserung der geschätzten Distanz auf 1 und für den Knoten 3 die neue bessere geschätzte Distanz 2. Im nächsten Schritt wird Knoten 2 als derjenige mit der kleinsten geschätzten Distanz aus der Warteschlange entnommen und so weiter. Zu diesem Zeitpunkt ist die kürzeste Distanz vom Startknoten zu den Knoten 1 und 2 bekannt, genau wie die kürzesten Pfade.

5.1.4. Der Kruskal-Algorithmus zur Berechnung des minimalen Spannbaums in einem gerichteten Graphen

Der Kruskal-Algorithmus zur Berechnung minimaler Spannbäume [Ein05d, Güt92] ist neben Dijkstra ein zweites Beispiel eines Graph-Algorithmus und soll die Auswahl an Beispielen in diesem Kapitel abrunden. Ein möglicher praktischer Einsatz ist etwa das Finden einer minimal zu asphaltierenden, flächendeckenden Menge von Straßen in einem gegebenen Straßennetz. Die

```
procedure Kruskal (G: in out Graph) is
    -- Bestimmt im gewichteten Graphen G einen spannenden Baum, in dem alle Knoten
    -- über Wege minimaler Länge mit einem Wurzelknoten verbunden sind.
    EQ: EdgePriorityQueue := CreateWith(G.E);  -- alle Kanten sortiert nach Gewichten in Queue
                                                -- Sortierung erfolgt einmal, ändert sich nie mehr
    NoOfSubtrees := NoOfNodes(G);
    u, v: Node;  e: Edge;
begin
    while not IsEmpty(EQ) and NoOfSubtrees > 1 loop
        -- die Menge der zu betrachtenden Kanten ist noch nicht leer und
        -- es sind noch nicht alle Knoten in einem Unterbaum gelandet
        Dequeue(EQ, e);
        u := e.source; v := e.target;
        if FindSubtreeRoot(u) /= FindSubtreeRoot(v) then
            -- die billigste noch nicht betrachtete Kante verbindet zwei getrennte Teilbäume
            MakeTreeEdge(e);
            MergeSubtrees(u, v);
            NoOfSubtrees := NoOfSubtrees - 1;
        end if;
    end loop;
end Kruskal;
```

Abbildung 29: Der Kruskal-Algorithmus in prozeduraler Notation, entnommen aus [Ein05a]

Optimalität wird durch eine minimale Summe von Kantengewichten der Kanten des Spannbaums erreicht.

Zum Finden eines Minimums der Summe der Kantengewichte bedient sich Kruskal einer Partitionierung der Knotenmenge des Graphen, die in jedem Abarbeitungsstand einer Menge gegebener Teil-Spannbäume entspricht, eben den Partitionen des Graphen. Begonnen wird mit lauter einelementigen Partitionen, die aus je einem Knoten bestehen. Schritt für Schritt werden dann durch Hinzufügen von Kanten mit minimalen Gewichten zu einem der Teilspannbäume bestimmte Partitionen miteinander verschmolzen, bis am Schluss nur noch *eine* Partition existiert und die Menge der bisher gefundenen Kanten den Kanten des gesuchten Spannbaums entspricht. Es wird also in üblichen Programmiersprachen eine Datenstruktur benötigt, die die Partitionierung der Knoten ermöglicht und zumindest eine Operation zum Verschmelzen zweier Partitionen zur Verfügung stellt. Abb. 29 zeigt den Algorithmus in Ada-Schreibweise.

In Abb. 30 sind vier Stadien des Kruskal-Algorithmus, angewandt auf einen Beispiel-Graphen, gezeigt. Dabei ist die Reihenfolge der Bilder wie ein Text zu lesen, also von links oben nach rechts unten. Links oben sind alle Kanten gestrichelt gezeichnet, um anzudeuten, dass die Kanten noch nicht untersucht wurden. Rechts oben wurden drei Kanten untersucht und durch die dunkle Färbung als Teile des minimalen Spannbaums identifiziert. Links unten ist eine weitere Kante untersucht worden, aber nicht zum Spannbaum hinzugenommen worden, was durch die durchgezogene, aber helle Zeichnung ausgedrückt ist (es handelt sich um die Kante mit Gewicht 48). Rechts unten schließlich wurden alle Kanten untersucht, der Algorithmus ist terminiert, wobei als gefundener Spannbaum der aus den dunklen Kanten zusammengesetzte zurückgeliefert

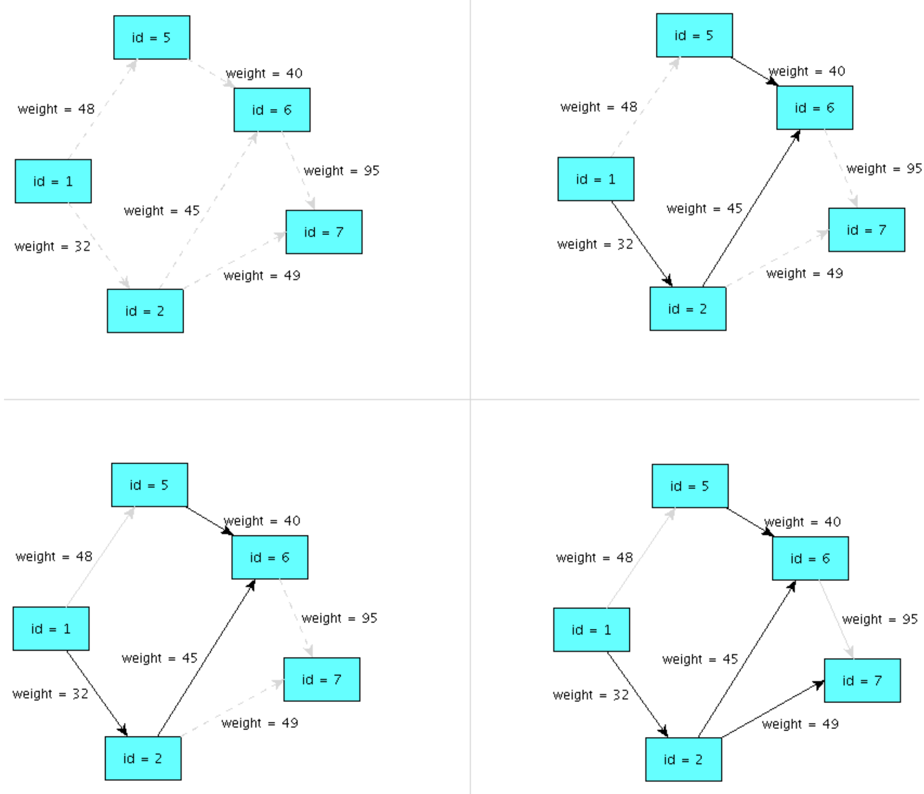


Abbildung 30: Vier Stadien des Kruskal-Algorithmus an einem Beispiel-Graphen gezeigt; die Richtung der Kanten spielt für Kruskal keine Rolle.

wird. Kruskal arbeitet auf ungerichteten Graphen, so dass in Abb. 30 die Richtung der Kanten einfach zu ignorieren ist.

Nachdem nun die vier Datenstruktur-Beispiele dieser Arbeit vorgestellt worden sind, werden wir im Rest dieses Kapitels unser Konzept der Erzeugung von Instanzen einer hochinteraktiven Visualisierungsumgebung aus visuellen Spezifikationen beschreiben.

5.2. Überblick über das VIDEA-Konzept

Im Rest dieses Kapitels wird ein Überblick über den in dieser Arbeit vorgestellten Ansatz gegeben (siehe auch [AS03, Asc03]). Dazu werden wir zunächst in Abschnitt 5.2.1 beschreiben, welche Einsatzgebiete wir für unseren Ansatz in der Lehre von Algorithmen und Datenstrukturen sehen. Zugrunde gelegt wird das in Kapitel 3 beschriebene grundsätzliche Szenario, das eine Phase der Spezifikation voraussetzt, nach der durch einen Generierungsvorgang eine Instanz einer hochinteraktiven Visualisierungsumgebung erzeugt wird. Zum besseren Verständnis der Einordnung von VIDEA wird in Abschnitt 5.2.2 der VIDEA-Ansatz von anderen Formen des E-Learning abgegrenzt. Im Zusammenhang damit werden in den Abschnitten 5.2.3 und 5.2.4

Fahrpläne und Lernschritte in VIDEA erläutert. In Abschnitt 5.2.5 stellen wir unser Konzept der visuellen Spezifikation von Datenstrukturen anhand eines kleinen Beispiels vor. Im darauf folgenden Abschnitt 5.2.6 werden einige von VIDEA angebotene informelle Hilfsmittel zur Verfeinerung einer vorliegenden Spezifikation im Hinblick auf die Erreichung des gewünschten didaktischen Ziels des Dozenten genannt. Einen groben Überblick über die Generierung einer VIDEA-Instanz aus einer vorhandenen Spezifikation gibt Abschnitt 5.2.9. Die generierte Instanz kann schließlich in den in Abschnitt 5.2.1 beschriebenen Szenarien eingesetzt werden. Ein Überblick über die Bedienung und Oberfläche einer solchen VIDEA-Instanz wird in Abschnitt 5.2.10 gegeben, während Abschnitt 5.2.11 das Zusammenspiel zwischen VIDEA und den genutzten Werkzeugen (hauptsächlich PROGRES und yFiles) beschreibt.

5.2.1. Einsatzgebiete und konkrete Szenarien

Wie bereits an der Vielfalt der in Kapitel 3 vorgestellten Anforderungen gesehen, ist eine große Bandbreite an Einsatzgebieten für eine konfigurierbare Visualisierungsumgebung, die in der Lehre von Algorithmen und Datenstrukturen eingesetzt werden soll, denkbar. Andererseits haben wir bei der Beschreibung früherer Studien in Kapitel 2 und bei der Vorstellung verschiedener Lehrwerkzeuge in Kapitel 4 gesehen, dass Ansätze, die möglichst viele Einsatzgebiete unterstützen wollen, oft Defizite bei der einfachen Spezifizierbarkeit, Generierbarkeit oder Konfigurierbarkeit aufweisen. Deswegen ist es eher ein Vorteil, dass wir durch die Wahl unseres Grundszenarios und der zu verwendenden Werkzeuge bereits eine gewisse Einschränkung getroffen haben:

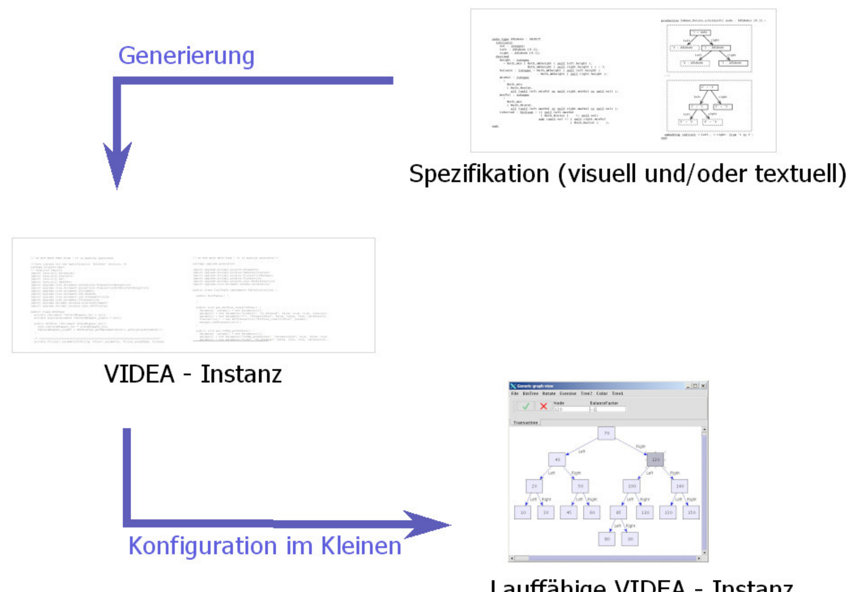


Abbildung 31: Die Generierungs-Schritte für eine neue VIDEA-Instanz

- Wir beschränken uns in VIDEA auf die Lehre graphbasierter Datenstrukturen. Das hat den Vorteil, dass wir uns sowohl bei der Visualisierung als auch bei der Spezifikation auf Graphen spezialisieren können. Allerdings umfassen die graphbasierten Datenstrukturen abgesehen vom Sortieren auf Feldern die am häufigsten in der Lehre eingesetzten Beispiele; unter anderem sind das Listen, Bäume oder allgemeine Graphen. Felder und komplexe Verbunde werden im Moment von den vorliegenden Layoutmechanismen nicht unterstützt.
- VIDEA verzichtet auf die werkzeugunterstützte Einbindung der Visualisierungsumgebung in komplexe Szenarien wie hochgradig verknüpfte hypermediale Ansätze. Dadurch erreichen wir unsere geforderte einfache Spezifizierbarkeit in einem durchgängigen Konzept.

Wegen der genannten Einschränkungen ist es VIDEA möglich, einfache Spezifizierbarkeit mit erheblichen interaktiven Fähigkeiten zu koppeln und damit unproblematisch eine große Bandbreite an Einsatzgebieten zu unterstützen, die natürlich alle im Grundszenario enthalten sind, das nochmal in Abb. 31 gezeigt wird:

- Zunächst wird eine Spezifikation erstellt, die alle erforderlichen Informationen über die Datenstruktur, ggf. den Algorithmus und das verwendete didaktische Grundkonzept enthält.
- Aus dieser Spezifikation wird in einem Generierungsschritt eine VIDEA-Instanz erstellt.
- Die generierte VIDEA-Instanz kann nun bezüglich der Benutzer-Schnittstellen durch Anpassung von XML-Dateien konfiguriert werden.

Zum letzten Punkt zählen die Darstellung von Knoten und Kanten, Farbschemata für Visualisierung und Fehlermeldungen, situationsbedingte Meldungen und Fehlermeldungen, vordefinierte Skripte und die Auswahl eines der vordefinierten Layoutalgorithmen. Falls keiner der vorhandenen Layoutalgorithmen genutzt werden kann, kann ein eigener erstellt werden. Näheres hierzu folgt in Kapitel 8.

Wir führen einige Beispiele von Einsatzgebieten in der Reihenfolge ansteigender Interaktivität auf, in die sich anschließend die vier in Kapitel 3 als besonders wünschenswert vorgestellten Szenarien einordnen lassen. Dabei wird der Begriff „Fahrplan“ vorweggenommen, der hier einfach als „Arbeitsanweisung“ verstanden werden kann und genauer in Abschnitt 5.2.3 diskutiert wird:

1. Der Einsatz vorbereiteter, fest spezifizierter Animationsabläufe oder Filme in der Lehrveranstaltung.
2. Der Einsatz vom Dozenten vorbereiteter, fest spezifizierter Animationsabläufe durch die Lernenden.
3. Das wiederholte Vorführen eines fest spezifizierten Algorithmus oder fest spezifizierter Operationen auf einer gegebenen Datenstruktur auf verschiedenen, zum Beispiel vorbereiteten oder zufällig generierten Szenarien oder Daten durch den Dozenten.
4. Der Einsatz vorbereiteter, eingeschränkt explorativer Szenarien in der Lehrveranstaltung, vorgeführt vom Dozenten oder vom Tutor in der Übung im Sinne von „Schaun wir doch mal, was passiert, wenn ...“. Hier kann konkretes Feedback gleich miteinbezogen werden.

5. Der Einsatz vorbereiteter, eingeschränkt explorativer Szenarien, diesmal genutzt von den Lernenden selbst.
6. Gemeinsame, aber vom Tutor moderierte Durchführung interaktiver Übungen mit Fahrplan.
7. Individuelle Durchführung interaktiver Übungen mit Fahrplan in Gruppen von Lernenden oder einzeln mit der Möglichkeit, einen Tutor zu fragen.
8. Aktive Nutzung ohne Fahrplan, also Selbstlernszenario mit gegebenem Lernziel, aber eben ohne Fahrplan.
9. Visuelles Debugging bzw. ferngesteuerte Animation eigener Programme.
10. Spezifikation, Generierung und Test eigener Instanzen der Visualisierungsumgebung von Seiten der Lernenden.

Obige Liste entstammt eigenen Überlegungen auf der Grundlage der in Kapitel 2 und 3 vorgestellten Erkenntnisse.

Die vier wesentlichen Szenarien aus Kapitel 3 lassen sich wie folgt einordnen:

- Vorführen im Rahmen einer Präsenzveranstaltung (obige Punkte 1/4)
- Passives Betrachten vorgegebener Abläufe (obiger Punkt 2)
- Interaktives Erforschen und Aufgaben-Lösen (obige Punkte 7/8)
- Ferngesteuerte Animation eigener Programme der Lernenden (obiger Punkt 9)

Diese Einsatzgebiete decken fast das ganze Spektrum von passiv bis hochgradig interaktiv ab und beinhalten gleichzeitig alle wesentlichen Elemente obiger Aufstellung 1-9. Somit können die meisten der obigen Szenarien durch Nicht-Nutzen bestimmter Fähigkeiten von VIDEA aus einer mächtigeren Variante abgeleitet werden. Ein Beispiel ist der Einsatz eingeschränkt explorativer Szenarien von Seiten des Dozenten (4.), was als ein begrenztes Szenario des Einsatzes eingeschränkt explorativer Szenarien von Seiten der Studenten (5.) gesehen werden kann: Der Unterschied besteht eben im fehlenden Zugriff der Lernenden auf bestimmte Operationen und Attribute und darin, dass der Dozent evtl. nicht alle verfügbaren interaktiven Möglichkeiten nutzt, die aber durchaus schon vorhanden sein dürfen. Mit anderen Worten: Der Dozent hat im Allgemeinen Zugriff auf alle vorhandenen Operationen und Manager einer VIDEA-Instanz. *Manager* heißen in VIDEA diejenigen Klassen, die das Anzeigeverhalten regeln und eine interaktive Schnittstelle anbieten, Details folgen in Kapitel 8. Bei der Verwendung durch Lernende wird üblicherweise ein Teil der interaktiven Schnittstellen der Manager nicht angeboten, um auf diese Weise die Nutzung zu steuern.

Die Szenarien 4 und 5 heißen eingeschränkt interaktiv, weil dort durch ein geringeres Angebot an interaktiv aufrufbaren Operationen, Selbsttests usw. eine klarere Führung der Lernenden gegeben ist als z. B. in den Szenarien 6 und 7. Dort wird stattdessen ein externer Fahrplan genutzt.

Eine Besonderheit stellt obiges Szenario 10 dar, in dem die Lernenden die Rolle des Dozenten übernehmen. In diesem Fall benötigen die Lernenden neben der Einführung in den eigentlichen Lernstoff, also die zu lehrende Datenstruktur oder den zu lehrenden Algorithmus, zusätzlich eine Einführung in die Spezifikation von Datenstrukturen. Das beinhaltet Informationen über die PROGRES-Sprache und vom Dozenten vorgegebene Rahmenbedingungen, wie

z. B. eine bereits in der PROGRES-Sprache formulierte Teilmenge von Constraints der Datenstruktur. Diese Lernenden können dann die Spezifikation vollenden und eine VIDEA-Instanz daraus erzeugen und konfigurieren. Diese VIDEA-Instanz kann dann in einem zweiten Schritt von den gleichen Lernenden verwendet werden, um ihr Verständnis der jeweiligen Datenstruktur in einem der Szenarien 1 bis 9 zu vertiefen. Oder die Lernenden werden als Tutoren für andere Lernende eingesetzt, die die entstandene VIDEA-Instanz in einem der Szenarien 1 bis 9 nutzen. In jedem Fall ist nach den in Kapitel 2 vorgestellten Erkenntnissen über die Wirksamkeit multimedialer Lehre für die „Lernenden der zweiten Stufe", also denjenigen, die die VIDEA-Instanz erstellt haben, ein deutlicher zusätzlicher Lerngewinn zu erwarten.

Das Szenario 10 ist durchaus sinnvoll, wird aber im Rahmen dieser Arbeit nicht näher beschrieben. Stattdessen gehen wir bei der Beschreibung ohne Beschränkung der Allgemeinheit davon aus, dass der Dozent die Spezifikation, Generierung und Konfiguration einer VIDEA-Instanz vornimmt, und der Lernende der Nutzer der Instanz ist. Sollte der Lernende die Rolle des Dozenten übernehmen, kann die Beschreibung der entsprechenden Abschnitte dieses Kapitels und insbesondere diejenige in Kapitel 8 als Erläuterung genutzt werden.

Nach diesem Überblick über mögliche Einsatz-Arten von VIDEA wird im nächsten Abschnitt eine Abgrenzung von anderen Arten der werkzeugunterstützen elektronischen Lehre vollzogen.

5.2.2. Abgrenzung von klassischer Lernsoftware

Nachdem im letzten Abschnitt die typischen Einsatzgebiete für VIDEA beschrieben wurden, mag sich die Frage stellen, wie eine Einordnung in das große Feld existierender E-Learning-Software und E-Learning-Konzepte möglich ist.

Die Antwort hängt von der Sichtweise auf VIDEA ab. Wenn unter VIDEA nur die Werkzeug-Unterstützung verstanden wird, die es ermöglicht, aus visuellen Spezifikationen VIDEA-Instanzen zu erstellen, diese zu konfigurieren und ablaufen zu lassen, ist eine Einordnung in existierende Konzepte schwierig. Das liegt hauptsächlich daran, dass das Einsatzgebiet des Werkzeugs an sich im Verhältnis zum riesigen Feld multimedialer Lehre eher eng ist. Außerdem lassen sich die erzeugten VIDEA-Instanzen wegen der starken Interaktivität schwer mit herkömmlicher Lernsoftware vergleichen, sondern – wie schon an anderer Stelle erwähnt – eher mit Simulationsumgebungen. Somit fällt auf dieser Ebene auch die Einordnung in Lern-Standards wie LOM [LOM05] und SCORM [SCO05] schwer, die beide aus den gleichen Gründen nicht recht für VIDEA passen wollen.

Wenn andererseits unter VIDEA das VIDEA-Konzept verstanden wird, fällt schon bei der Beschreibung der Szenarien im vorigen Abschnitt auf, dass weiterhin die klassischen Lernsituationen Präsenzveranstaltung, Übungen, Selbststudium etc. im VIDEA-Konzept vorgesehen sind. VIDEA will somit nicht herkömmliche Lehre ersetzen, sondern ergänzen. So gesehen soll VIDEA auch keine Konkurrenz zu dem sein, was oft unter Lernsoftware verstanden wird, also zum Beispiel werkzeugunterstützte Verwaltung von Lehrveranstaltungen und Praktika, das Angebot von Web-Schnittstellen für die Registratur, Verwaltung und Annahme gelöster Aufgaben von Lernenden, Hilfe bei der Erstellung von Multiple-Choice-Tests oder beim Installieren von Chat-Räumen. All diese vorhandenen Technologien, die eher organisatorischer Art sind, können bei Bedarf problemlos zur Ergänzung des VIDEA-Ansatzes genutzt werden.

Eine weitere Möglichkeit ist, Lernsoftware einfach nach dem Wortsinn zu verstehen, also als Software, die zum Lernen bestimmt ist, und eine der wenigen verfügbaren Definitionen [wik05] hinzuzunehmen. Dann kann man Lernsoftware insgesamt beschreiben als „Software, die zum Lernen bestimmt ist und die der Lernende auf seinem Computer gespeichert hat“. In einem solch allgemeinen Sinn fällt natürlich auch VIDEA unter den Begriff Lernsoftware.

Eine weitere mögliche Frage der Einordnung in bisherige Konzepte kann die Frage nach Lernpfaden sein. Da hier der Sprachgebrauch uneinheitlich ist, könnten Missverständnisse entstehen. Um diese zu vermeiden, verwenden wir folgende Definitionen:

Definition: Ein *Lernpfad* ist eine Folge von Lernschritten. Ein *Lernschritt* ist die Kombination einer atomaren Aktion des Lernenden mit der sich daraus ergebenden Beschäftigung des Lernenden mit dem hierauf multimedial angebotenen Lernstoff bis zur nächsten atomaren Aktion.

Ein Beispiel für einen Lernschritt ist also das Aufrufen einer Operation aus dem Menü zusammen mit dem darauffolgenden Verfolgen der sich ergebenden Änderungen der Datenstruktur. Ein anderes ist die Beantwortung einer interaktiv gestellten Frage zusammen mit der Reflektion der Reaktion des Systems, z. B. einer ermunternden Meldung oder einer Fehlermeldung. Ein Grenzfall ist der Aufruf eines vordefinierten Skriptes, das eine rein passiv zu betrachtende Abfolge von Animationen auslöst, also einen Animationsfilm: Hier ist es eine Frage des Blickwinkels, ob man im Falle existierender Pausen zwischen verschiedenen Teilsequenzen der Animation den Aufruf des ganzen Films zusammen mit dessen Betrachtung als *einen* Lernschritt ansieht oder als *eine Folge von Lernschritten*, deren Aktionen jeweils aus der Beschäftigung „Warten auf den nächsten Schritt“ des Nutzers bestehen. Schließlich kann man nicht davon ausgehen, dass jeder Lernende einem längeren passiven Film interessiert folgt. Wir erwähnen diesen Fall so ausführlich, da der Dozent sich bei der Erstellung einer Spezifikation für eine VIDEA-Instanz über die Lernschritte, die er ermöglichen will, klar werden muss. Eine Klärung könnte hier die Nutzung einfacher interaktiver Trennaktionen sein, so dass der Lernende etwa eine Schaltfläche „weiter“ nutzen kann, um die Teilsequenzen abzuspielen. Dieses Beispiel ist laut der in Kapitel 2 vorgestellten Erkenntnisse aus der Lernpsychologie durchaus aus praktischen Gründen sinnvoll.

Die Frage ist nun, welche Unterstützung VIDEA bzgl. Lernpfaden bietet. Eine VIDEA-Instanz stellt zunächst einmal zur Gewährleistung der hochgradig interaktiven Möglichkeiten im Allgemeinen absichtlich keinen festen, vom Werkzeug erzwungenen Lernpfad zur Verfügung, dem der Nutzer folgen *muss* (außer der Dozent konfiguriert in Ausnahmefällen eine VIDEA-Instanz für einen besonderen Anwendungsfall so). Wenn zusätzlich zu den bisherigen Definitionen ein *Lernszenario* als die Menge der möglichen Lernpfade verstanden wird, die in einer gegebenen Ausgangssituation, also einer spezifischen darstellten Instanz einer Datenstruktur, möglich sind, dann sind die Lernszenarien in VIDEA schon für kleine Datenstrukturen sehr komplex. Man mag sich die Frage stellen, ob es nicht sinnvoll oder sogar notwendig ist, zumindest in manchen Szenarien gewisse Einschränkungen zu treffen, und Lernpfade sozusagen abzuschneiden. Tatsächlich sieht VIDEA hier zwei Möglichkeiten vor, die in den nächsten beiden Abschnitten beschrieben werden:

- Externe Fahrpläne und
- implizit in die Spezifikation eingebaute Lernschritte.

5.2.3. Fahrpläne

Ein *Fahrplan* im Sinne von VIDEA entspricht einer Vorgabe, wie eine vorliegende Instanz zu nutzen ist, um zu einem vom Dozenten vorgegebenen Lernziel zu gelangen.

Prinzipiell gäbe es zwei Möglichkeiten, Fahrpläne für die Benutzung einer VIDEA-Instanz zu implementieren: Werkzeugunterstützt oder nicht werkzeugunterstützt. Obwohl es in VIDEA durchaus möglich ist, in der visuellen Spezifikation auch Abhängigkeiten zwischen Operationen zu spezifizieren, z. B. um eine bestimmte Reihenfolge zu erzwingen, ist dies in den meisten Fällen nicht sinnvoll, außer natürlich für die Implementierung eines festen Algorithmus. Denn durch ein solches Vorgehen würde einerseits die von uns angestrebte maximale Einfachheit und Wiederverwendbarkeit der Spezifikation in Frage gestellt, und andererseits würden Informationen über die Lernumgebung mit den Informationen über die Datenstruktur vermischt werden, was keinesfalls dem VIDEA-Konzept entspricht (siehe Punkte 13, 17 und 18 in Kapitel 4 auf Seite 44).

Deshalb sind im VIDEA-Konzept *externe Fahrpläne* vorgesehen. Da unser Credo dasjenige einer hochinteraktiven Simulationsumgebung ist, deren Konfiguration nicht mehr Aufwand bedeuten soll als unbedingt notwendig, werden die zu einem bestimmten Zeitpunkt auszuführenden Aktionen nicht rein programmatisch, sondern durch eine Kombination mehrerer Bausteine festgelegt:

- Der Hauptbestandteil eines VIDEA-Fahrplans ist die Formulierung einer Übungsaufgabe, die auf die Variante 3 (*interaktives Erforschen und Aufgaben-Lösen*) einer VIDEA-Instanz zugeschnitten ist.
- Die Nutzung dieses Fahrplans wird nach der VIDEA-Philosophie nicht überwacht, sondern durch Freischaltung nur der hierfür notwendigen Aktionen unterstützt und entweder durch die Anwesenheit eines Tutors oder die Abgabe von Ergebnissen kanalisiert.

Ein solcher Fahrplan, für den wir allgemeine Hinweise in Kapitel 8 geben werden und in Kapitel 9 zwei konkrete Beispiele vorstellen werden, ist vom Moment der Spezifikation einer neuen VIDEA-Instanz an Teil des didaktischen Konzeptes. Da bereits durch die Nutzung von VIDEA unser didaktisches Konzept im Hintergrund steht und automatisch in Wirkung tritt, kann sich der Dozent auf die Formulierung der typischen Lernschritte einer bestimmten VIDEA-Instanz konzentrieren und passend für diese ein grundlegendes Verwendungsszenario entwerfen. Das Verwendungsszenario für eine VIDEA-AVL-Instanz könnte z. B. wie folgt aussehen:

1. AVL-Baum mit kontinuierlicher Animation aufbauen.
2. Dabei alle Rotationsoperationen zeigen.
3. Verständnis der Definition der Höhe eines Teilbaums testen.
4. Richtige Rotationen in bestimmten Situationen finden lassen.
5. Knoten an richtiger Stelle einfügen lassen.

Nachdem das grobe Szenario feststeht, können Spezifikation und Fahrplan parallel entwickelt werden. In den Kapiteln 7 und 9 werden für ein etwas komplexeres Szenario als das oben skizzierte mögliche Fahrpläne in dem entsprechenden Umfeld der visuellen Spezifikation vorgestellt.

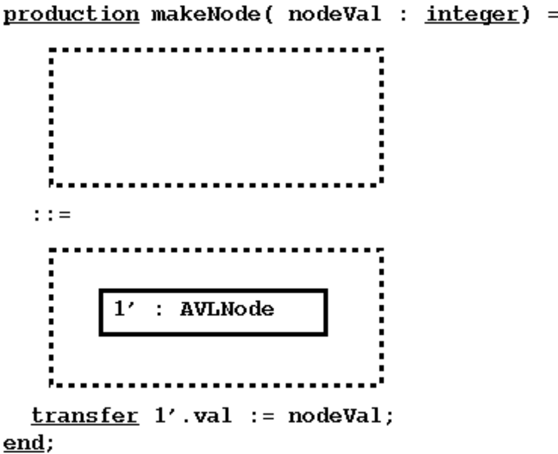


Abbildung 32: Konstruktor eines einfachen Objekts

Die weitere Konkretisierung eines Fahrplans führt zu Lernpfaden und -schritten, auf die wir ausführlicher als bisher im nächsten Abschnitt eingehen werden.

5.2.4. Lernschritte in VIDEA

Lernschritte dienen in VIDEA zur Strukturierung einer Spezifikation und sind damit eine Hilfe für den Dozenten, Fahrpläne zu konkretisieren.

Um zu zeigen, dass implizit in der Spezifikation bereits Lernschritte vorhanden sind, betrachten wir zuerst, wie überhaupt Bearbeitungsschritte in VIDEA vorgegeben werden. Es wurde bereits gesagt, dass wir in VIDEA nicht die explizite Definition von Lernschritten ermöglichen, da die Spezifikation der Datenstruktur

- so einfach wie möglich sein soll und
- strikt getrennt sein soll von organisatorischen Informationen aller Art.

Beispiele für den letzten Punkt sind Meldungen und Layoutalgorithmen. Obwohl also keine explizite Definition von Lernschritten in VIDEA vorgesehen oder erwünscht ist, werden bei der Definition von Operationen implizit manche Abläufe erlaubt und andere verboten, deren elementare Einheiten man als äußerst grundlegende Lernschritte bezeichnen könnte. Die Schritte, die ein Lernender beim Nutzen einer VIDEA-Instanz durchlaufen kann, sind nämlich vorbestimmt durch drei Vorgaben des Dozenten, und zwar:

1. Durch die Möglichkeiten, eine Datenstruktur aufzubauen, indem Grundbausteine erzeugt werden,
2. Durch die möglichen Transformationen einer Datenstruktur mithilfe der Anwendung im jeweiligen Kontext erlaubter Operationen. Hierzu gehört auch die Überprüfung der Erfüllung möglicher Bedingungen, die an die Ausführung einer Operation geknüpft sind.
3. Durch einen ggf. vorgegebenen Fahrplan.

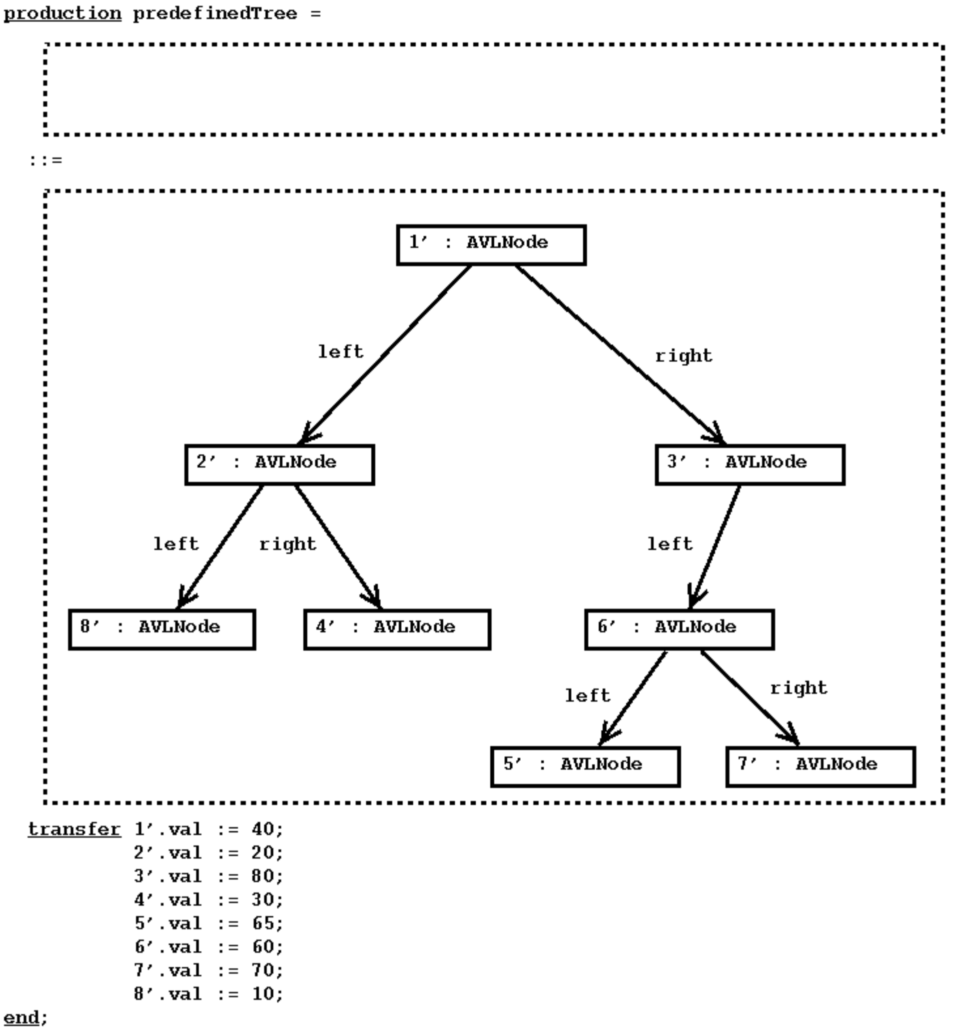


Abbildung 33: Konstruktor eines kompletten Baums

Punkt 3 ist aus den genannten Gründen im VIDEA-Konzept enthalten, allerdings üblicherweise als externes Dokument und nicht vom Werkzeug unterstützt.

Zu Punkt 1 lässt sich sagen, dass der Dozent Operationen definieren kann und sollte, die neue Objekte erzeugen. Beispiele hierfür sind Operationen für die Erzeugung von Grundbausteinen der Datenstruktur, wie die Erzeugung von Objekten oder Beziehungen zwischen ihnen (in Graphen-Sprechweise also Knoten oder Kanten) oder auch die direkte Erzeugung vordefinierter fester Instanzen der jeweiligen zu unterrichtenden Datenstruktur. Abb. 32 zeigt die visuelle Spezifikation der Erzeugung eines AVLNode-Objektes in PROGRES, während Abb. 33 die visuelle Spezifikation eines fertigen AVL-Baums zeigt. Abb. 34 zeigt die Erzeugung einer *left*-Beziehung zwischen zwei schon vorhandenen AVLNode-Objekten. Diese Operationen zum Erzeugen von

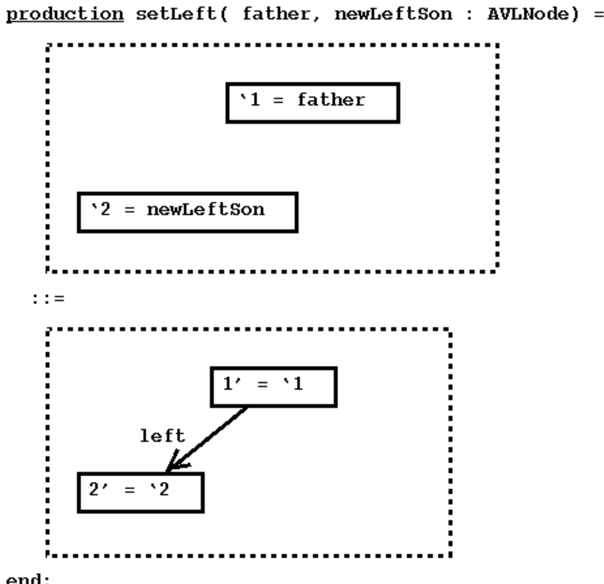


Abbildung 34: Erzeugen einer *left*-Beziehung zwischen zwei Objekten

Objekten und grundlegenden Beziehungen zwischen ihnen kann man auch Erzeugendensystem nennen, da durch diese Basis-Operationen jede Instanz der jeweiligen Datenstruktur, in unserem Beispiel also (zusammen mit einer analogen *setRight*-Operation) jeder AVL-Baum, aufgebaut werden kann. Wenn der Aufbau der Datenstruktur durch mehrere grundlegende Basisoperationen erfolgen soll, ist die Verwendung eines Skriptes eine Möglichkeit, den detaillierten Prozess der Erzeugung vor den Lernenden zu verbergen. Allerdings ist der Pfad zum Aufbau einer speziellen Instanz einer Datenstruktur nicht unbedingt eindeutig, wie man schon an dem AVL-Baum in Abb. 33 sieht; denn dieser Baum könnte auch mit den Operationen aus Abb. 32 und Abb. 34 und einer zusätzlichen *setRight*-Operation erstellt werden. So gesehen gibt der Dozent durch die zum Aufbau einer Datenstruktur zur Verfügung gestellten Operationen bereits mögliche Lernpfade vor. Es ist für den Dozenten wichtig, sich über die möglichen Lernpfade im Klaren zu sein, weil sie, ergänzend zu den Fahrplänen, das hauptsächliche Mittel in VIDEA sind, den Lernenden trotz der erwünschten hohen Interaktivität einer VIDEA-Instanz vorbereiteten Lernerfahrungen auszusetzen.

Wenn dann eine Instanz der jeweiligen Datenstruktur, z. B. ein AVL-Baum, vorliegt, können weitere verändernde Operationen auf die Datenstruktur angewandt werden. Die Trennlinie zwischen den erzeugenden Operationen (obiger Punkt 1) und den unter Punkt 2 zusammengefassten „verändernden Operationen“ ist zwar in gewisser Weise willkürlich; trotzdem ist eine Unterscheidung unserer Meinung nach gerechtfertigt, weil die unter Punkt 2 zusammengefassten Operationen tendenziell eher Elemente verändern als erzeugen. Ein typisches Beispiel hierfür ist die bereits vorgestellte Linksrotation auf dem AVL-Baum, im Zuge derer zwar auch Kanten erzeugt werden, aber dafür andere wegfallen, so dass man vom üblichen didaktischen Blickwinkel aus eher von einer Umordnung spricht.

Damit ist unsere Einteilung vergleichbar mit der in [LG86] vorgeschlagenen Einteilung von Operationen in *constructors*, die dem Aufbau einer Datenstruktur oder eines Objektes dienen, und *modifiers*, die ein existierendes Datenobjekt verändern. Die dritte in [LG86] vorgeschlagene Klasse von Operationen, die *observers*, liefern Eigenschaften eines existierenden Objektes zurück. Da wir alle relevanten Eigenschaften anzeigen, brauchen wir *observers* an dieser Stelle nicht, können aber die weiter unten beschriebenen Verständnistests grob in diese Kategorie einordnen.

Die Operationen, die wir unter Punkt 2 zusammenfassen, erhöhen in besonderem Maße die zur Verfügung stehenden Lernpfade, da im Normalfall auch zyklische Abläufe, die also nach einigen Operationen wieder zur ursprünglichen Instanz einer Datenstruktur führen, angeboten werden. Extreme Beispiele hierfür sind die Nutzung von Rechtsrotation und folgender Linksrotation auf einem AVL-Baum oder – noch offensichtlicher – ein 'undo' nach einer beliebigen Operation, das in VIDEA prinzipiell immer zur Verfügung steht, ohne explizit definiert werden zu müssen. Andererseits sind die unter Punkt 2 zusammengefassten verändernden Operationen üblicherweise gebunden an gewisse Vorbedingungen, unter denen sie erlaubt sind. Auf diese Weise findet eine Einschränkung der zur Verfügung stehenden Lernpfade in einer bestimmten Situation statt, die bei der Spezifikation von Operationen üblicherweise darin besteht, dass sie nur in bestimmten Kontexten einsetzbar sind. Ein Beispiel hierfür ist wiederum die Linksrotation, die nur dann erlaubt ist, wenn bestimmte Muster im Baum vorliegen. Wird die Operation in einer unerlaubten Situation aufgerufen, dann soll eine Fehlermeldung ausgegeben werden, deren Wirkung zwar die weitere Nutzung der VIDEA-Instanz durch den Lernenden wegen des erhofften Lerneffekts hoffentlich verändert, aber zunächst die Instanz der Datenstruktur unverändert lässt. Bei der Implementierung von Teilschritten eines Algorithmus kann es komplexere, weitergehende Einschränkungen geben, wie diejenige in Abb. 35. Dort wird ein Teil eines Schrittes des Kruskal-Algorithmus davon abhängig gemacht, dass es keine Kante `'1` in der Warteschlange gibt, deren Gewicht kleiner ist als die im Moment zu betrachtende Kante `'2` (in der oberen `valid`-Klausel), dass `'2` ebenfalls in der Warteschlange ist (in der unteren `valid`-Klausel) und dass `'2` zwei Teilspannbäume mit der gleichen *subTreeID* miteinander verbindet (in der `condition`-Klausel). Nur dann wird `'2` zur weiteren Bearbeitung aus der Warteschlange entnommen (in der `transfer`-Klausel). Es ist an dieser Stelle nicht notwendig, dieses Beispiel im Detail nachzuvollziehen, denn es dient nur als exemplarische Darstellung für eine etwas komplexere Vorbedingung einer Operation.

Eine verwandte Möglichkeit zu den beschriebenen Vorbedingungen für bestimmte Operationen ist die Definition von Invarianten. Invarianten können analog zu den bisher gezeigten Beispielen in PROGRES formuliert werden, wobei es wieder die Entscheidung des Dozenten ist, ob die Verletzung einer Invariante unmöglich ist oder zu einer Zustandsveränderung der Datenstruktur führt, die einen Fehler anzeigt. In beiden Fällen kann von einer Einschränkung oder Kanalisierung des jeweiligen Lernpfades gesprochen werden.

Syntaktisch unterscheidet sich die Formulierung von Invarianten nicht von den bisher gezeigten Vorbedingungen. Die bisher vorgestellten Beispiele der visuellen Spezifikation in PROGRES sind so formulierbar wie hier dargestellt, können teilweise aber eleganter implementiert werden. Da es uns hier um den Überblick geht, verschieben wir Details hierzu und weitere Erklärungen auf Kapitel 8.

Mit den bisher vorgestellten Mechanismen in der Spezifikation der Datenstruktur, die ei-

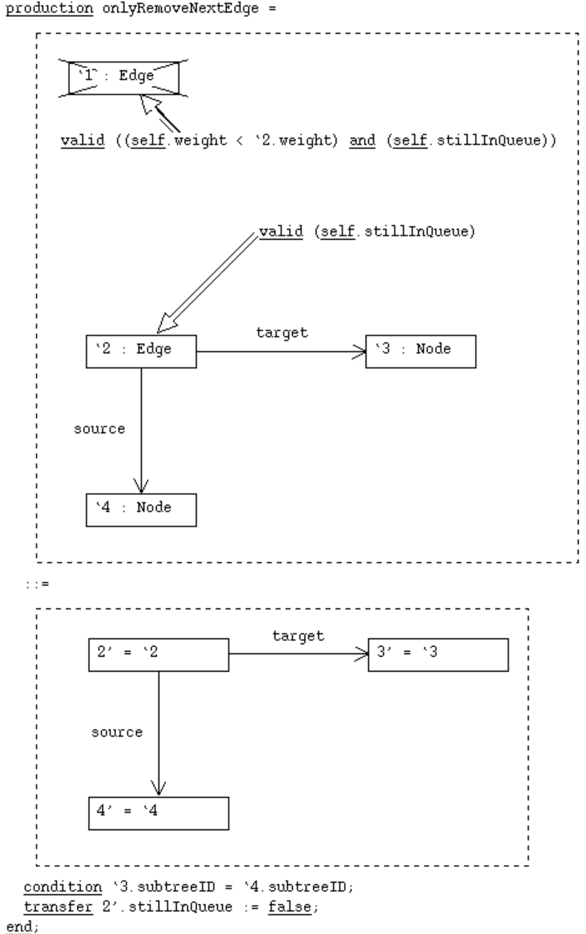
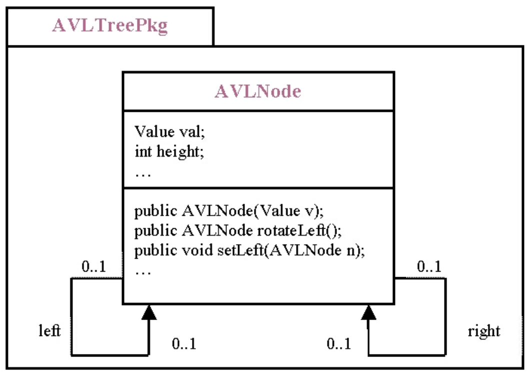


Abbildung 35: Das Entfernen der kleinsten noch nicht betrachteten Kante aus der Warteschlange in Kruskal

nerseits eine Vielzahl an impliziten Lernpfaden ermöglichen, andererseits die Möglichkeiten an manchen Stellen bewusst einschränken, bleiben den Lernenden typischerweise in jeder Situation immer noch eine größere Anzahl an Entscheidungsmöglichkeiten, als dies in klassischer Lernsoftware der Fall ist. Wie bereits mehrfach betont, ist das ja auch einer der Punkte in VIDEA, auf die von Anfang an bewusst Wert gelegt wurde. Allerdings ist es in manchen Lernszenarien, zum Beispiel dem *interaktiven Erforschen und Aufgaben-Lösen* wünschenswert, den Lernenden besser zu führen. Dafür werden die im letzten Abschnitt eingeführten und unter Punkt 3 genannten Fahrpläne genutzt.

Im nächsten Abschnitt werden wir mit unserem Überblick über VIDEA fortfahren und eine Kurzeinführung in die Erstellung einer Spezifikation geben.



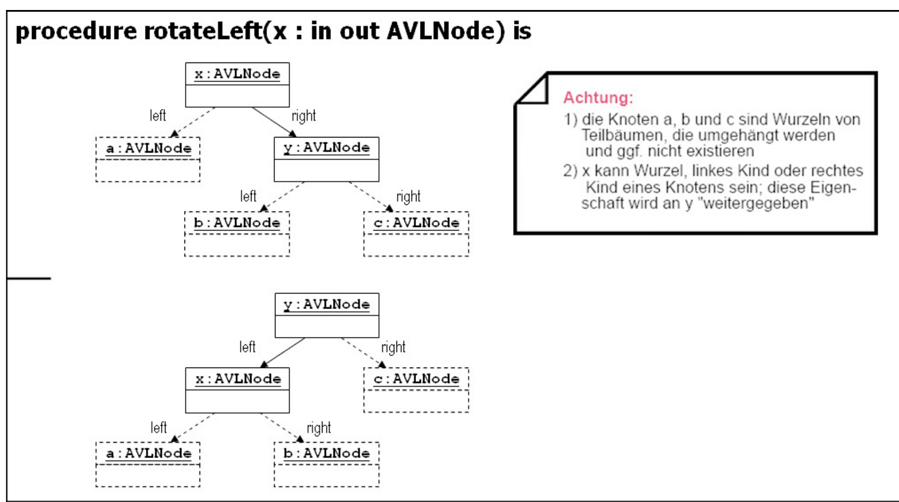


Abbildung 36: Oben: AVL-Knotentyp als UML-Klassendiagramm. Unten: Die Linksrotation im AVL-Baum, dargestellt als Schema von Paaren von Objektdiagrammen

5.2.5. Erstellen einer Spezifikation

Da wir uns in VIDEA für die Spezifikation von Datenstrukturen der UML und damit einer objektorientierten Vorgehensweise bedienen, besteht der erste Schritt bei der Erstellung einer Spezifikation in der Aufteilung der Datenstruktur in Klassen und der Darstellung als UML-Klassendiagramm. Bei den AVL-Bäumen ist eine einfache Möglichkeit die Nutzung nur einer Klasse `AVLNode` mit einem Attribut *val* für den Wert des jeweiligen Knotens und einigen organisatorischen Attributen wie *height* für die Speicherung der Höhe desjenigen Teilbaums, dessen Wurzel der jeweilige Knoten ist. Als Methoden der Klasse `AVLNode` kommen Konstruktionsoperationen wie *createNode*, *setLeft* und natürlich die Rotationsoperationen in Frage. Abb. 36 oben zeigt ein beispielhaftes UML-Klassendiagramm, wobei die Verbindungen zum linken und rechten Kind als Assoziationen *left* und *right* modelliert sind.

Dann werden wichtige Operationen als UML-Objektdiagramme spezifiziert. Wir zeigen am

```
node type AVLNode : OBJECT
  intrinsic
    val : integer;
    left : AVLNode [0:1];
    right : AVLNode [0:1];
  derived
    height : integer
      = Math_max ( Math_mkDef ( self.left.height ),
                   Math_mkDef ( self.right.height ) ) + 1;
    balance : integer = Math_mkDef ( self.left.height )
                      - Math_mkDef ( self.right.height );
    minVal : integer
      =
        Math_min
        ( Math_maxInt,
          all (self.left.minVal or self.right.minVal or self.val) );
    maxVal : integer
      =
        Math_max
        ( Math_minInt,
          all (self.left.maxVal or self.right.maxVal or self.val) );
    isSorted : boolean = ([ self.left.maxVal
                          | Math_minInt ]    <= self.val)
                        and (self.val <= [ self.right.minVal
                                         | Math_maxInt ]    );
    noTree : boolean
      = card ( self.<-left- or self.<-right- ) > 1;
end;
```

Abbildung 37: AVL-Knotentyp in PROGRES

Beispiel der Linksrotation in Abb 36 unten, wie in VIDEA das UML-Objektdiagramm genutzt werden kann, um die Operationen auf graphische Art zu spezifizieren, die sowohl für Lernende verständlich ist als auch als Übergang zur weiter unten beschriebenen PROGRES-Schreibweise dient (Abb. 38). Genauer gesagt, stellt Abb 36 unten ein Schema von Paaren von Objektdiagrammen dar. Es fällt auch auf, dass Abb. 36 unten ziemlich ähnlich zu derjenigen Schreibweise ist, die in Lehrbüchern immer schon für die Darstellung von Schnittstellenoperationen auf einer Datenstruktur üblich ist, nämlich eine getrennte graphische Darstellung der Zustände vor und nach der Operation.

In der Abbildung werden zwei Arten von Darstellungen für obligatorische und optionale Knoten genutzt: Durchgezogene Kästen für Objekte, die existieren müssen, und gestrichelte Kästen für Objekte, die existieren können. Eine ähnliche Darstellung war in der gängigeren Schreibweise aus Abb. 24 zu sehen.

Als zweiter Schritt folgt die händische, aber recht intuitive Übertragung der UML-Klassendiagramme und der UML-Objektdiagramme nach PROGRES. Da es uns an dieser Stelle darum geht, einen Überblick über VIDEA zu geben, verschieben wir eine ausführlichere Beschreibung der PROGRES-Sprache auf Kapitel 6, und stellen hier nur kurz und beispielhaft die Erstellung von Teilen einer Spezifikation für die AVL-Bäume vor. Obwohl PROGRES auch eine graphische Schema-Darstellung bietet, zeigen wir für AVLNode gleich die ausdrucksstärkere textuelle Darstellung der aus dem Klassendiagramm nach PROGRES übertragenen Informationen plus einer Reihe zusätzlicher Definitionen. Die textuelle Darstellung ist hier angemessen, da die Informationen des Klassendiagramms in Abb. 36 oben im Wesentlichen auch nur textuelle Beschriftungen in einem Kasten sind. Die Klasse `AVLNode` wird zu einem Knotentyp *AVLNode*,

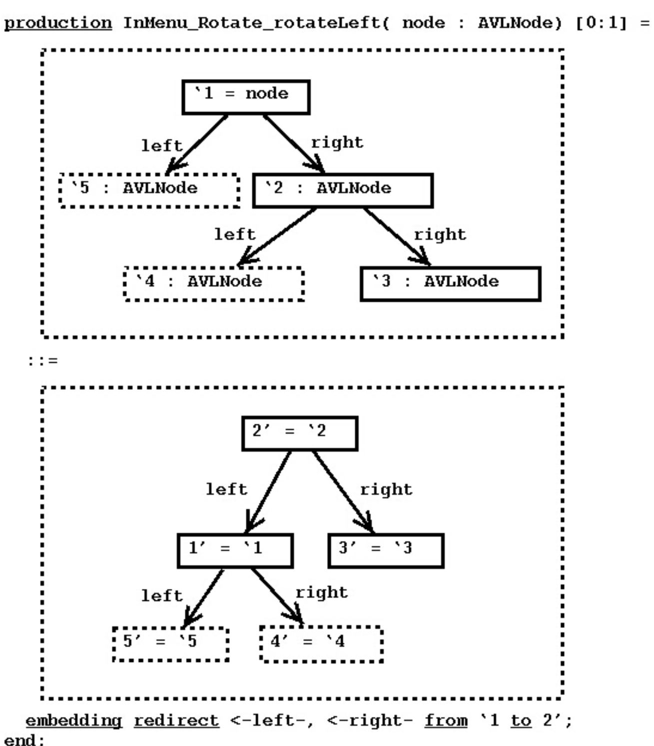


Abbildung 38: Die Linksrotation im AVL-Baum, dargestellt als PROGRES-Produktion

wie in Abb. 37 zu sehen ist.

Als erstes fällt auf, dass die Schreibweise strukturell ähnlich ist wie im Klassendiagramm; der Knotentyp wird als Klasse gesehen und darunter stehen die Attribute, wobei wir diese in zwei Gruppen aufteilen: *Intrinsic* (also eigenständig) heißen diejenigen Attribute, die unabhängig von Werten anderer Attribute sind und deren Werte durch Zuweisung gesetzt werden. Als `derived` (also zur Laufzeit berechnet) werden Attribute bezeichnet, die durch Gleichungen über andere Attribute definiert sind. In unserem Beispiel ist z. B. der Wert *val* des Knotens etwas, das fest zum Knoten gehört, während die Höhe *height* und der Balancefaktor *balance* aus der Struktur des Baumes zur Laufzeit hervorgehen.

An dieser Stelle gehen wir nicht näher auf die genaue Bedeutung aller verwendeten Konstrukte ein. Es sei nur erwähnt, dass *Math_mkDef* eine Hilfsfunktion ist, die an anderer Stelle der PROGRES-Spezifikation definiert wird und aus einem undefinierten Wert eine 0 macht, während *minVal* und *maxVal* Hilfsattribute für das boolesche Attribut *isSorted* sind. Näheres dazu im nächsten Kapitel über die in der AVL-Spezifikation verwendeten PROGRES-Konstrukte.

Die in Abb. 36 oben gezeichneten Assoziationen *left* und *right* wurden in der PROGRES-Spezifikation als knotenwertige optionale Attribute modelliert, was im Prinzip gleichwertig zu einer Kante von *AVLNode* zu *AVLNode* ist.

Das verwendete PROGRES-Schlüsselwort `self` ist eine Referenz auf den spezifizierten

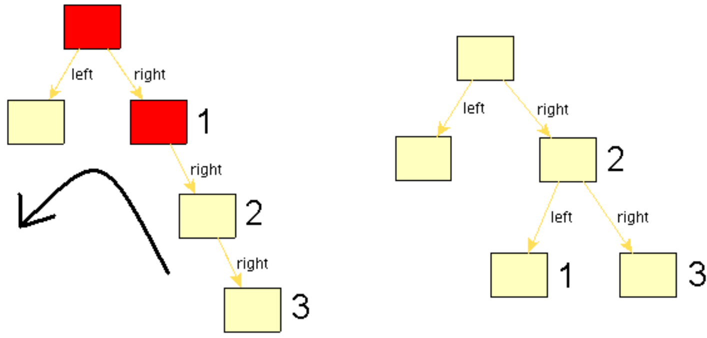


Abbildung 39: Klassische, lehrbuchartige Darstellung der Linksrotation in einem AVL-Baum

Knoten vom Typ *AVLNode* selbst.

Das von einem graphischen Gesichtspunkt aus interessantere Schema von Paaren von Objektdiagrammen zur Beschreibung der Linksrotation wird übertragen in die visuelle Spezifikation einer PROGRES-Produktion, deren Code in Abb. 38 zu sehen ist. Man beachte die Ähnlichkeit der Abbildungen in Abb. 36 unten und 38.

In den beiden durch `::=` getrennten gestrichelten Rechtecken steht der Zustand der zu bearbeitenden Stelle des Datenstruktur-Graphen vor und nach der Operation.

Zunächst ist der knotenwertige Parameter *node* zu erwähnen, der den Knoten spezifiziert, um den die Rotation erfolgen soll. Eine Alternative wäre, hier einen Knotenschlüssel (z. B. vom Typ `integer`) zu übergeben. Der Vorteil des *AVLNode*-Typs selbst ist die Möglichkeit, zur Laufzeit der VIDEA-Instanz einen Knoten mit der Maus zu markieren und dann direkt *rotateLeft* aufzurufen; der zugehörige Parameter wird dann automatisch von VIDEA gefüllt.

Der Name *InMenu_Rotate_rotateLeft* entspringt einer Namenskonvention der VIDEA-Instanz, durch die die Operation *rotateLeft* im Menü *Rotate* einsortiert und dort auch angezeigt wird. Die Kardinalität [0:1] bedeutet, dass die Linksrotation entweder fehlschlägt oder ein deterministisch definiertes Resultat liefert.

Im Rumpf fällt auf, dass der Knoten `1 als äquivalent zum bereits erwähnten Parameter *node* angegebenen ist; das heißt, das PROGRES-Laufzeitsystem sucht eine Passstelle im aktuellen Baum, in dem der übergebene Knoten, dessen rechtes Kind und wiederum dessen rechtes Kind existieren. Diesen drei Knoten werden dann respektive die Nummern `1, `2 und `3 zugewiesen. Das (wie sein gestricheltes Kästchen andeutet) optionale linke Kind des Knotens `1 bekommt im Falle der Existenz die Nummer `5 zugeordnet und das optionale linke Kind des Knotens `2 die Nummer `4. Die *embedding*-Klausel in Abb. 38 entspricht der in Pseudonotation gegebenen Zusatzbedingung 2 in Abb. 36.

Wer Abb. 36 unten und 38 genau vergleicht, wird feststellen, dass die als existierend vorausgesetzten Knoten, also die Vorbedingung der Operation, unterschiedlich gewählt sind. Das wäre bei zwei Diagrammen derselben VIDEA-Instanz ein Übertragungsfehler. Wir stellen beide Optionen dar, um auf diesem Weg zu zeigen, dass der Dozent in den Details einer Spezifikation mehr sinnvolle Wahlmöglichkeiten hat, als vielleicht auf den ersten Blick zu vermuten wäre.

5.2.6. Verfeinern einer Spezifikation

Nach der Erstellung der UML-Diagramme und der entsprechenden PROGRES-Konstrukte für die wesentlichsten Objekte und Operationen einer Datenstruktur bedarf es weiterer Spezifikationen im Hinblick auf das Einsatzszenario der VIDEA-Instanz. Denn ein Teil des didaktischen Konzepts wird in die Spezifikation der Datenstruktur codiert.

Das widerspricht in keiner Weise unserer Trennung von Spezifikation der Datenstruktur und Konfiguration anderer Informationen wie Layoutalgorithmen, Farbschemata oder Meldungen. Denn das didaktische Konzept gehört in VIDEA zur Datenstruktur von vorneherein dazu, was auch der Wiederverwendbarkeit von Spezifikationen entgegenkommt; schließlich will man bei einer leichten Änderung einer Datenstruktur das didaktische Grundkonzept nicht jedes Mal neu entwerfen. VIDEA bietet u. a. folgende Bausteine und Vorgehensweisen für den Entwurf eines Verwendungsszenarios an:

- Formulierung von Tests für die später den Lernenden zu stellenden Fragen zum Algorithmus.
- Das Üben verschiedener Detaillierungsstufen eines Algorithmus durch die Zerlegung eines Algorithmus in Schritte bzw. Teilschritte.
- Aufbrechen von Schleifen in Teilschritte.
- Üben des Einsatzes der Operationen einer Datenstruktur durch deren Definition als PROGRES-Produktionen oder -Transaktionen.
- Erstellung einer Hierarchie von Operationen für die spätere interaktive Simulation oder die Programmierung eines eigenen Studentenprogramms, das ferngesteuert animiert werden soll.
- Aufteilung von Aktionen in Menüs, um eine logische Zuordnung zu erleichtern, was einerseits den Lernenden zugute kommt, und andererseits das spätere Freischalten oder Verbergen von Aktionen für spezielle Anwendungsfälle erleichtert.
- Definition zusätzlicher, zunächst nicht in der Datenstruktur vorkommender Attribute für die Implementierung von Invarianten und deren Nutzung in Tests.
- Beispiele für die Erstellung von Übungsblättern als Teil der Fahrpläne einer VIDEA-Instanz.
- Definition harter und weicher Constraints.

Diese Grundbausteine und Vorgehensweisen sind in VIDEA nicht streng formalisiert, sondern zum Teil informell beschrieben (siehe Kapitel 8) und zum Teil bei der Beschreibung der vier beispielhaften VIDEA-Instanzen für die vier beschriebenen Datenstrukturen und Algorithmen in Kapitel 7 bzw. 9 exemplarisch dargestellt. Damit ähneln sie im Vergleich zur Erstellung von Programmen in üblichen Programmiersprachen weniger der Syntax eines Verwendungsszenarios, sondern eher bewährten Mustern, deren Nutzung dem Dozenten nahegelegt, aber nicht von ihm gefordert wird.

Obwohl es hier nur um den Überblick geht, sollen zwei in VIDEA zentrale Punkte der obigen Liste, deren Verständnis für den Nachvollzug der folgenden Kapitel hilfreich ist, in den folgenden zwei Abschnitten kurz angerissen werden: Die Definition von Constraints und das Aufbrechen von Schleifen.

5.2.7. Constraints

Ein wichtiger Punkt bei der Erstellung von Lerneinheiten ist natürlich die Tatsache, dass es den Lernenden erlaubt sein soll, bestimmte Fehler zu machen, um überhaupt experimentieren und lernen zu können. Wir verweisen hier auf den didaktischen Begriff des „Explorierens“ [MRR95], der bereits in Kapitel 2 eingeführt wurde. So wollen wir zum Beispiel beim balancierten Suchbaum die Möglichkeit haben, zeitweise auch unbalancierte Bäume zu verarbeiten und darzustellen, um dann eben durch Fehlermeldungen oder eine neue Farbgebung einen Lerneffekt zu bewirken und dem Lernenden eine überprüfbare „Reparatur“ des Baums zu ermöglichen. Wir können also grob unterscheiden:

- Eine Menge von immer gültigen, „harten“ Eigenschaften der Datenstruktur, die zu jedem Zeitpunkt gelten müssen und die der Lernende nicht umgehen darf. Ein Beispiel hierfür ist die Binärbaum-Eigenschaft beim AVL-Baum; das heißt, es dürfen zum Beispiel nie mehr als zwei Kanten den gleichen Knoten als Ausgangspunkt haben.

- Eine Menge von temporär verletzbaren, „weichen“ Eigenschaften, die durch farbliche Markierung der jeweiligen Knoten oder Kanten und durch (Fehler-)Meldungen behandelt werden. So kann etwa die temporär verletzbare Balancierung eines AVL-(Teil-)Baums durch ein Knoten-Attribut *balance* (das wie in Abb. 37 definiert sein kann) seiner Wurzel modelliert werden und dann farblich hinterlegt dargestellt werden oder durch Anzeige des *balance*-Attributs die Aufmerksamkeit der Lernenden auf die fehlerhafte Stelle gelenkt werden.

  Ein weiteres Beispiel für eine weiche Eigenschaft ist die temporär falsche Sortierung eines AVL- bzw. Binär-Baums.

5.2.8. Aufbrechen von Schleifen

Das Aufbrechen von Schleifen in Teilschritte eines Algorithmus ist eine algorithmische Transformation, die notwendig ist, um sinnvolle Lernschritte zu ermöglichen. Hintergrund ist einerseits, dass es für eine Lernumgebung für Algorithmen und Datenstrukturen nicht ausreicht, Algorithmen als Ganzes ausführen zu können. Es ist also nicht zielführend, einen Algorithmus einfach in die PROGRES-Sprache zu übersetzen und dem Benutzer z. B. als Transformation zur Verfügung zu stellen, weil dann für den Lernenden nur der Zustand der Datenstruktur vor und nach Ausführung des gesamten Algorithmus sichtbar wäre. Somit wäre der Lernende überfordert – abgesehen davon, dass keine Lerneffekte über einzelne wesentliche Schritte des Algorithmus möglich wären.

Andererseits würde es nicht ausreichen, wenn Schritte automatisiert würden, indem z. B. das PROGRES-Laufzeitsystem am Ende jedes Schleifendurchlaufs automatisch stoppt – eine Erweiterung, die relativ einfach in PROGRES zu implementieren wäre. Denn dann wäre der Benutzer bei geschachtelten Schleifen gezwungen, sich schrittweise in den feinen Schritten der innersten Schleife durch den gesamten Algorithmus zu bewegen. Allerdings sind Erweiterungen denkbar, die kurz in Kapitel 10 diskutiert werden.

Dieses Problem ist natürlich nicht neu, denn die in Kapitel 2 beschriebenen Ansätze, die sich der 'interesting events' bedienen, sind ja zunächst entwickelt worden, um dieses Dilemma der

While es gibt noch graue Knoten **loop**

- nimm den grauen Knoten mit der kleinsten Distanz (mithilfe der Prioritätswarteschlange)
- nenne ihn U
- färbe ihn schwarz,
- setze seine einlaufende Kante, die Kandidat für den kürzesten Weg ist, als sicher auf dem kürzesten Weg
- **For all** von U auslaufenden Kanten E **loop**
  - nenne den Zielknoten V
  - wenn V weiß ist, setze ihn grau (und füge ihn in die Prioritätswarteschlange ein)
  - wenn V nun grau ist und sich seine berechnete Distanz über die neue Kante E verbessern lässt, setze seine Distanz neu und nimm E als neuen Kandidaten für den kürzesten Weg zu V

  end loop;

end loop;

Abbildung 40: Die Hauptschleife des Dijkstra-Algorithmus als Pseudocode

Algorithmenanimation zu lösen.

Die Lösung im VIDEA-Konzept ist, bei der Spezifikation eines Algorithmus in PROGRES gleich von Anfang an Schleifen durch Schritte zu simulieren, indem Schrittweiten verschiedener Größe definiert werden.

Diese Lösung hat zwei große Vorteile:

1. Das oben geschilderte Problem mit zu wenig oder zu viel dargestellten Details eines Algorithmus entfällt vollständig, da der Lernende zu jedem Zeitpunkt die Schrittweite wählen kann. Ein Fahrplan kann hier sinnvolle Vorgaben machen, so dass wichtige Stellen der Algorithmenausführung auch wirklich bearbeitet werden.
2. Die Interaktivität der VIDEA-Instanz bleibt auch während der Simulation eines Algorithmus durch den Lernenden voll erhalten. Denn auch im Inneren einer Schleife ist es dem Lernenden möglich, den vorbereiteten Ablauf des Dozenten zu verlassen und den nächsten (Teil-)Schritt des Algorithmus frei zu simulieren.

Als konkretes Beispiel sei hier der Dijkstra-Algorithmus genannt, dessen Hauptschleife nochmal in Abb. 40 als Pseudocode dargestellt ist. Nach den bisher dargelegten Überlegungen ist es nicht sinnvoll, den Algorithmus in eine PROGRES-Transaktion zu übersetzen. Stattdessen werden beide Schleifen in Schritte zerlegt, wobei die Schritte der inneren Schleife dann zu Teilschritten eines Schrittes der äußeren Schleife werden. Für den Dijkstra-Algorithmus sieht das dann so aus wie in Abb. 41 gezeigt.

Wir sehen hier eine Transaktion, die im Zuge des Ablaufs des Dijkstra-Algorithmus immer wieder vom Benutzer der VIDEA-Instanz aufgerufen werden kann und dabei – bedingt durch die Struktur – jeweils den gerade anfallenden Schleifendurchlauf der äußeren Schleife von Abb. 40

```
transaction InMenu_Dijkstra_nextStep =
    makeUToNormalNode
  & dequeueMinimal
  & setUBlack
  & choose
      makeUsPredToUsFixedPred
    else
      skip
    end
  & findCurrentVs
  & recolorVs
  & redistanceVs
  & makeVsToNormalNodes
end;
```

Abbildung 41: Haupt-„Schleife“ des Dijkstra-Algorithmus

vollzieht. Dazu bedient sich die Transaktion *nextStep* mehrerer Hilfsproduktionen, die jeweils Teilaufgaben des Algorithmus erledigen. Hier sei nur soviel gesagt, dass die innere `For all`-Schleife aus Abb. 40 durch die Aktionen *findCurrentVs*, *recolorVs* und *redistanceVs* in Abb. 41 repräsentiert werden.

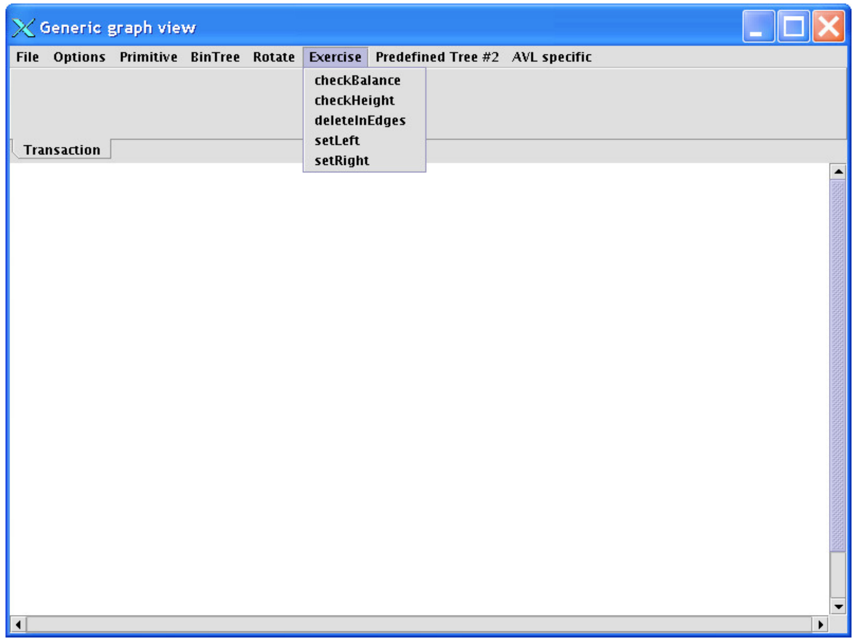
Eine genauere Beschreibung dieser Bausteine und der Implementierung des Dijkstra-Algorithmus in PROGRES verschieben wir auf Kapitel 8 und gehen nun als nächstes auf die Generierung einer VIDEA-Instanz ein.

5.2.9. Generierung und Konfiguration

Die in unserem Grundszenario geforderte Trennung von Spezifikation und Konfiguration bedeutet für den Dozenten, dass er nach erfolgter Spezifikation in PROGRES unmittelbar eine VIDEA-Instanz generieren kann, die lauffähig ist. Diese eigentliche Generierung erfordert in VIDEA lediglich die Nutzung eines vorliegenden Skriptes, das alle erforderlichen Dateien in einem neuen Verzeichnis erstellt. Dieses Verzeichnis ist das Verzeichnis der entsprechenden VIDEA-Instanz und kann beliebig an andere Stellen im Verzeichnisbaum verschoben werden.

Nach der Generierung empfiehlt es sich für den sinnvollen Einsatz, einige Konfigurationsschritte vorzunehmen. Eine Möglichkeit hierfür ist die Anpassung zweier generierter XML-Dateien, ohne dass nochmal etwas generiert oder kompiliert werden muss. Zu den konfigurierbaren Eigenschaften der Visualisierung gehören

- Schemata für die angezeigten Attribute des jeweiligen Knotentyps,
- Farbschemata für Knoten und Kanten,
- Stilschemata für Kanten,
- (Fehler-)Meldungen in Abhängigkeit von der Transaktion und drei vordefinierten Standardproblemcodes,
- zu nutzende Layoutalgorithmen,
- zu nutzende Skripte,
- Einstellungen für das Verbergen oder Anbieten von Operationen und
- die Aufteilung letzterer auf Menüs.



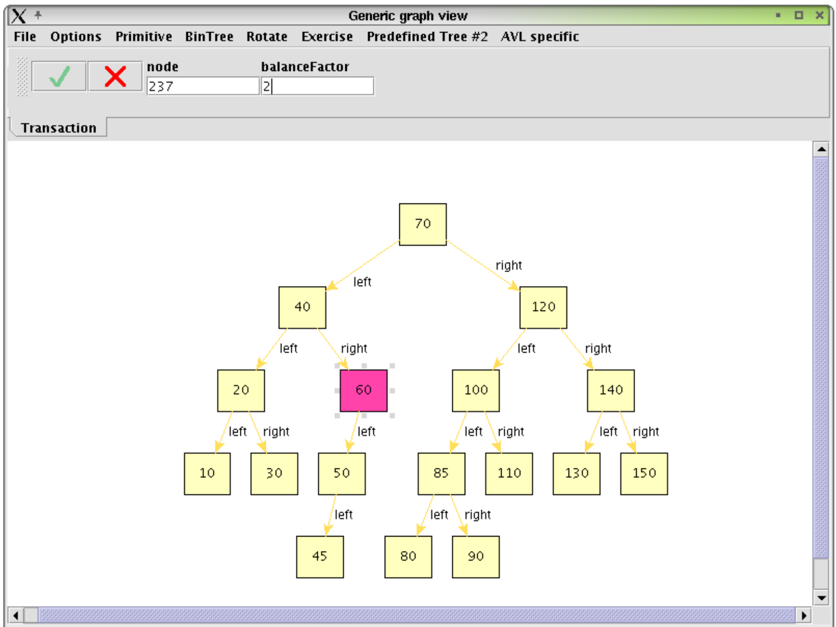


Abbildung 42: AVL-Instanz. Oben: Leere Instanz. Unten: Verständnistest für die Berechnung des Balancefaktors

Ein Teil der erwähnten Einstellungen kann zusätzlich zur Laufzeit in den VIDEA-Managern (siehe Kapitel 8) konfiguriert werden. Lediglich die Nutzung neuer Layoutalgorithmen erfordert den Einbau des entsprechenden Java-Codes an einer genau definierten Stelle des Layoutmanagers, wobei eine Reihe fertiger yFiles-Layoutmanager als Bausteine genutzt werden können.

Da hiermit die einfache Generierung und Konfiguration in VIDEA für einen Überblick ausreichend beschrieben sind, gehen wir weiter zur Beschreibung der Bedienung und Oberfläche einer VIDEA-Instanz.

5.2.10. Bedienung und Oberfläche

Nach dem Start einer VIDEA-Instanz sieht der Benutzer das leere Graphbrowser-Fenster (Abb. 42 oben). Das Haupt-Bedienelement des Lernenden ist das Menü. Standardmäßig enthält das Menü

- einen Menüpunkt `File->Close` zum Beenden der VIDEA-Instanz,
- mindestens einen Eintrag für den Aufruf von Skripten, deren Name vom Dozenten konfiguriert wurde,
- mögliche statische Verweise auf VIDEA-Funktionalität, zum Beispiel für den Aufruf der interaktiven Oberfläche eines Farbmanagers oder für die Implementierung eines neuen Farbschemas und
- beliebig viele Menüpunkte, die auf in PROGRES spezifizierte Operationen verweisen.

Die Verteilung aller Menüpunkte auf Untermenüs erfolgt über eine XML-Konfigurationsdatei, wobei es möglich ist, die Verteilung der PROGRES-Operationen über Präfix-Analyse der Operationsnamen automatisch vornehmen zu lassen.

Neben der Auswahl von Menüpunkten hat der Benutzer je nach Konfiguration die Möglichkeit, Knoten zu bewegen oder Knoten bzw. Kanten für die weitere Verwendung zu markieren. Es ist auch möglich, die Erzeugung von Knoten durch einen Mausklick auf den Hintergrund manuell anzustoßen, wenn eine PROGRES-Produktion zum Erzeugen eines Knotens existiert und für diesen Zweck konfiguriert wurde.

Abb. 42 unten zeigt die Oberfläche der AVL-Instanz bei der Nutzung eines Verständnistests; in diesem Fall soll der Lernende eingeben, wie groß der Balancefaktor des im Baum gerade markierten Knotens ist. Nach dem Drücken der Schaltfläche mit dem Häkchen erhält er dann eine positive oder negative Reaktion – je nach der Korrektheit der Eingabe.

5.2.11. Zusammenspiel

Abb. 43 zeigt eine generierte VIDEA-Instanz im Umfeld von PROGRES und yFiles. Dadurch wird das Zusammenspiel zwischen VIDEA, PROGRES, UPGRADE2 und yFiles deutlich. Das bisher nicht erwähnte UPGRADE2 [UPG03] ist ein Java-basiertes Rahmenwerk, das u. a. eine Schnittstelle anbietet, um PROGRES über Java-Aufrufe und Java-Ereignisse ansprechen zu können. In Kapitel 8 werden wir genauer auf UPGRADE2 eingehen. Für die Zwecke dieses Kapitels kann UPGRADE2 vereinfacht als eine Java-API von PROGRES gesehen werden.

Innerhalb des großen Kastens in Abb. 43 sind diejenigen Komponenten rechts dargestellt, die mit PROGRES realisiert sind, während die zu UPGRADE2 bzw. VIDEA gehörigen Teile sich links befinden. Wir setzen voraus:

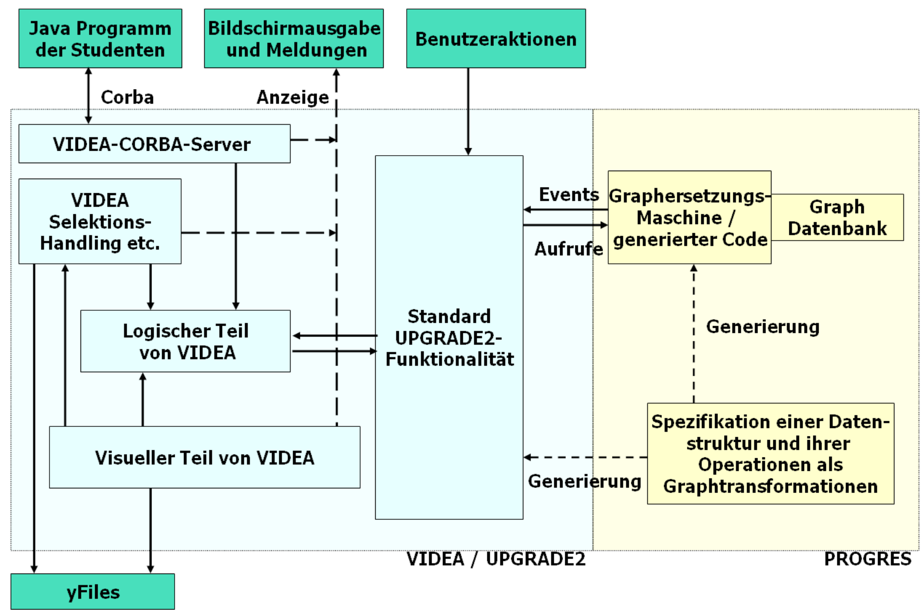


Abbildung 43: Nutzung von PROGRES, UPGRADE2, yFiles in VIDEA

- Eine Datenstruktur wie etwa der AVL-Baum wurde als PROGRES-Knotentyp spezifiziert.
- Wichtige Schnittstellenoperationen (Einfügen eines neuen Elementes, Linksrotation, ...) wurden als Graphtransformationen spezifiziert.
- Für Übungsaufgaben benötigte Hilfsprozeduren auf dem AVL-Baum (wie z. B. ein Test *isBalanced*) wurden ebenso als Graphtransformationen spezifiziert.

Aus dieser Spezifikation wird der Code für die VIDEA-Instanz generiert, der sowohl die Basis für die Standardfunktionalität für UPGRADE2 darstellt, als auch die notwendige Basis für die PROGRES-Laufzeitumgebung. Der generierte Code operiert auf einer speziellen Graph-Datenbank, die – neben wesentlichen Eigenschaften üblicher Datenbanksysteme – die Möglichkeit anbietet, durch Ereignisse von Änderungen im Graphzustand zu berichten. Außerdem wird hier die inkrementelle Auswertung abgeleiteter Attribute durchgeführt und durch die Verwendung von Vorwärts- und Rückwärtsdeltas die geforderte 'undo' / 'redo'-Funktionalität für beliebig lange Sequenzen Graph-verändernder Operationen angeboten. Insgesamt führt die Laufzeitumgebung (in der Abbildung bestehend aus „Graphersetzungs-Maschine“ und „Graphdatenbank“) die Graphtransformationen aus, nimmt Aufrufe von VIDEA über UPGRADE2 entgegen und generiert Ereignisse für UPGRADE2 und VIDEA.

Die restlichen Komponenten seien anhand eines Beispielablaufs charakterisiert: Wenn der Lernende etwa im Rahmen einer Übung zum manuellen Aufbau eines AVL-Baumes einen gerade neu erzeugten Knoten mithilfe einer Produktion *addLeft* als linkes Kind eines schon im Baum befindlichen Knotens einfügen will, kann er zuerst den zukünftigen Elternknoten mit der Maus markieren, wobei der Selektionsmanager (im Selektions-Handling in Abb. 43) über ei-

ne yFiles-Funktionalität den markierten Knoten identifiziert, mithilfe des logischen Teils von VIDEA in eine PROGRES- / UPGRADE2-Knotennummer umrechnet und für die weitere Verwendung in einer entsprechenden UPGRADE2-Datenstruktur vormerkt. Das gleiche passiert nun mit dem neuen linken Kind. Unter der Bedingung, dass die vorher erwähnte PROGRES-Produktion *addLeft* zwei Knoten als Parameter erwartet, und zwar zuerst den Elternknoten und dann das neue linke Kind, kann der Lernende nun im Menü *addLeft* aufrufen. In diesem Fall verpackt die Basisfunktionalität des generierten UPGRADE2 Prototypen diesen Aufruf in einen Aufruf der entsprechenden PROGRES-Produktion und übergibt die zugehörigen Parameter. Die Graphersetzungsmaschine führt die Graphtransformation auf ihrer Datenbank aus und liefert dementsprechende Ereignisse zurück, etwa im Erfolgsfall ein `AddEdge`-Ereignis für die neu hinzukommende Kante. Der Layoutmanager, der im visuellen Teil von VIDEA enthalten ist, erfährt davon mithilfe der Basisfunktionalität des Prototypen und führt mit Ausnutzung von yFiles-Funktionalität einen flüssigen Animationsschritt aus, in dem das neue linke Kind an seinen ihm nun zustehenden Platz im Graphen links unter seinem Elternknoten bewegt wird. Im Fehlerfall wäre eine Fehlermeldung ausgegeben worden und der Graph hätte sich nicht verändert.

Die bisher nicht beschriebene Komponente „Java-Programm der Studenten“ bezieht sich auf die erwähnte Möglichkeit in VIDEA, eine ferngesteuerte Animation eigener Programme der Lernenden durchzuführen.

Eine ausführlichere Beschreibung der in diesem Kapitel überblicksartig dargestellten Funktionsweise und Bedienung von VIDEA geben wir im Rest dieser Arbeit – inklusive einer genaueren Beschreibung des Konzeptes der Spezifikation von Algorithmen und Datenstrukturen. Zunächst werden wir uns dazu in Kapitel 6 weitere Beispiele der Spezifikation von Algorithmen und Datenstrukturen in VIDEA ansehen und die dahinterliegenden Konzepte genauer als bisher beleuchten.

6. Spezifikation

Nach dem Überblick über VIDEA in Kapitel 5 gehen wir jetzt etwas genauer auf die Grundlagen der Spezifikation von Algorithmen und Datenstrukturen mithilfe von Graphen und Graphtransformationen ein. Dazu wird einerseits unser Ansatz der Spezifikation von Datenstrukturen – inklusive konkreter Beispiele der graphbasierten Spezifikation – genauer als bisher beschrieben. Andererseits wird gezeigt, bei welchen Datenstrukturen diese graphbasierte Spezifikation weniger sinnvoll ist.

Nach der Erläuterung der Einschränkungen unseres Ansatzes zur Spezifikation von Algorithmen und Datenstrukturen erklären wir die Grundzüge einer PROGRES-Spezifikation zunächst allgemein und dann anhand der vier bereits eingeführten Beispiele AVL-Bäume, doppelt verkettete Liste, Dijkstra-Algorithmus und Kruskal-Algorithmus.

6.1. Einschränkungen und Abgrenzung

Wie bereits im letzten Kapitel beschrieben, betrachten wir in dieser Arbeit Datenstrukturen, die sich als Graphen darstellen lassen. Im Prinzip ist das keine Einschränkung, da die meisten Datenstrukturen sich aus einfachen Datentypen, Verbunden und Zeigern zusammensetzen, die sich jeweils als Knoten bzw. im letzten Fall als Kanten darstellen lassen und somit einen Graphen bilden. Allerdings gibt es drei Fälle, in denen eine andere Art der Visualisierung sinnvoll sein könnte:

1. **Die Visualisierung feldbasierter Datenstrukturen.**
   Gerade die Visualisierung dieser Art von Algorithmen hat in Form der Sortieralgorithmen einen prominenten Vertreter. Natürlich wäre in VIDEA eine Darstellung als verkettete Liste von Elementen möglich, auf die ein direkter Zugriff besteht. Es bestünde dann aber die Gefahr, dass durch die Darstellung der zusätzlichen Kanten und anderen Elemente eine Verwirrung oder Ablenkung vom eigentlichen Lerninhalt auftritt. Deshalb und weil für typische Algorithmen auf Feldern genügend gut implementierte Beispiele existieren, die zudem ausreichend parametrisierbar sind, nehmen wir Datenstrukturen, die Felder nutzen, aus. Anders verhält es sich mit auf Feldern arbeitenden, aber graphartig gut visualisierbaren Algorithmen. Ein Beispiel für letzteren Fall ist Heapsort mit einem als Baum dargestellten Heap, den wir in unserem Ansatz ebenso visualisieren können.

2. **Spezielle Darstellungsweisen.**
   Manche Datenstrukturen werden von bestimmten Dozenten auf eine Art dargestellt, die von der üblichen Art, Graphen zu zeichnen, abweicht.

   Ein Beispiel hierfür sind B-Bäume, in deren Darstellung oft auf eine waagrechte Anordnung der Attribute wert gelegt wird. Ähnliches gilt auch manchmal für die Darstellung von Listen. Außerdem werden manchmal für die Blätter andere graphische Primitive als für innere Knoten verwendet. Abb. 44 oben zeigt ein Beispiel dieser Zeichenweise für einen 2-3-Baum. In unserem Ansatz würde der gleiche Baum mit den Standard-Visualisierungsmechanismen wie in Abb. 44 unten gezeichnet werden. Hier werden beispielhaft ein paar innere Attribute angezeigt, die natürlich auch ausgeblendet werden könnten. Sie

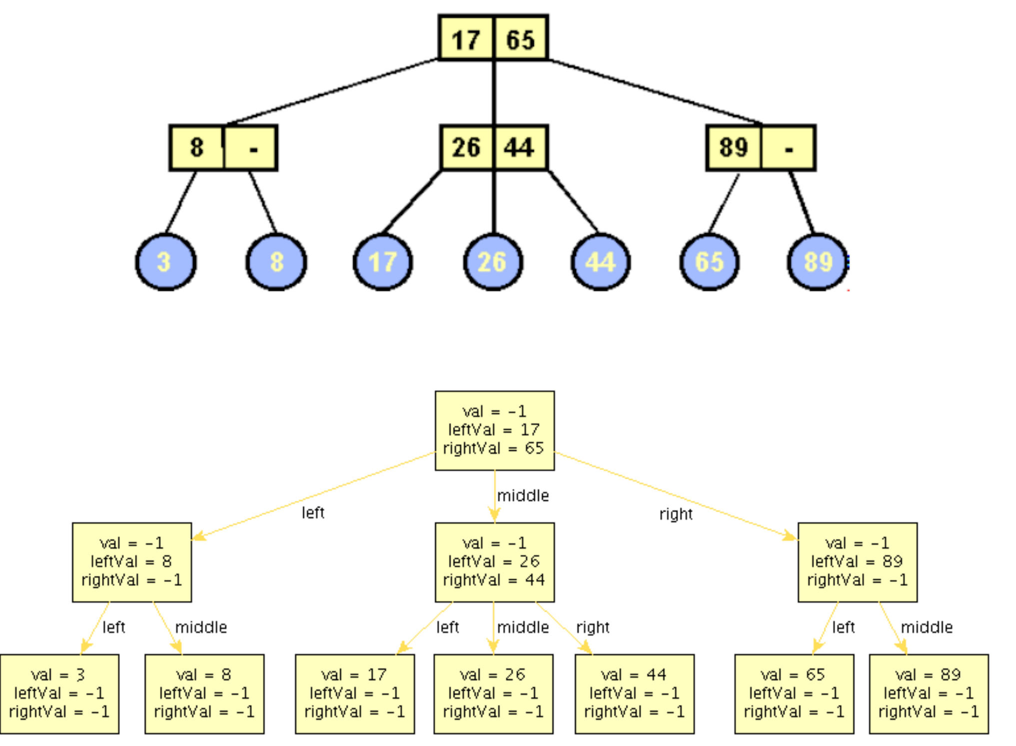


Abbildung 44: 2-3-Baum. Oben: Gängige Darstellung. Unten: Standard-Darstellung in VIDEA.

werden aber untereinander dargestellt. Die Eigenschaft, ob es sich um ein Blatt oder einen inneren Knoten handelt, wird hier ebenfalls durch ein Attribut festgelegt. Natürlich ist es möglich, Blätter z. B. in einer anderen Farbe darzustellen.

Um aber die exakte Darstellung wie in Abb. 44 oben zu erreichen, wäre es nötig und sinnvoll, die Layoutmöglichkeiten des Rahmenwerks zu erweitern.

3. **Die Darstellung spezieller Diagrammarten.**
Bezüglich der graphischen Elemente, die angezeigt werden können, bedeutet unsere Einschränkung auf Graphen, dass wir Rechtecke, Beschriftungen und Pfeile in verschiedenen Farben bzw. Stilen darstellen, aber z. B. keine Balkendiagramme, Tortendiagramme, dreidimensionalen Effekte etc. Ein größeres Rahmenwerk, das diese Sichten mit einschließt, wäre somit eine sinnvolle Erweiterung oder Weiterentwicklung von VIDEA.

Es sei noch einmal betont, dass alle drei ausgeschlossenen Arten von Einschränkungen auf keiner konzeptionellen Restriktion der Spezifikationssprache beruhen, sondern auf dem bestehenden Konzept der Visualisierung von Datenstrukturen als Standardgraphen. Denn jede Datenstruktur, die mit UML-Klassen- und Objektdiagrammen spezifiziert werden kann, kann auch mit Graphen und Graphtransformationen in PROGRES spezifiziert werden.

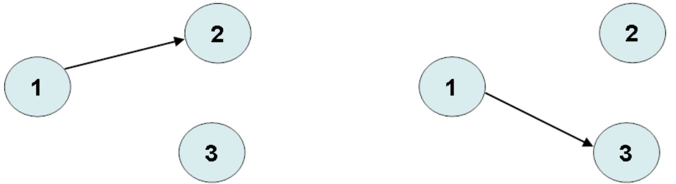


Abbildung 45: Umhängen oder Löschen und Neu-Erzeugen einer Kante?

6.2. Operationen auf Graphen

In VIDEA setzen wir auf eine Kombination der Stärken von

- visuellen Spezifikationen mit Graphersetzungen (PROGRES),
- den Möglichkeiten von Java und Swing (wie Erweiterbarkeit und Plattformunabhängigkeit) als Plattform für die Visualisierungsumgebung und
- yFiles mit den in Kapitel 4 genannten Fähigkeiten und Vorteilen als effiziente Nutzung der Java-Plattform für die Visualisierung und flüssige Animation.

Beispiele von Operationen auf Datenstrukturen, die visualisiert werden, sind Rotationen auf einem AVL-Baum, das farbliche Markieren eines kürzesten Weges in einem gerichteten Graphen, das Erstellen und Einhängen neuer Listenelemente in einer doppelt verketteten Liste oder das Markieren mehrerer Knoten in einem Spannbaum. Diese groben Operationen lassen sich auffassen als Folgen einfacher Aktionen auf einer Datenstruktur wie

- das Umhängen eines Zeigers
- das Hinzukommen eines Objektes
- die Änderung eines Objekt-Attributs.

Alle diese Möglichkeiten können natürlich wiederum zurückgeführt werden auf folgende Basisoperationen auf dem zur Darstellung verwendeten Graphen:

- Das Erzeugen oder Löschen eines Knotens,
- das Ändern eines Knoten-Attributs,
- das Erzeugen oder Löschen einer Kante,
- das Ändern eines Kanten-Attributs.

Ein grundlegendes Beispiel der Rückführung einer einfachen Operation auf eine Basisoperation ist das Umhängen eines Zeigers, wie es bereits bei den Rotationsoperationen auf dem AVL-Baum gezeigt wurde. Das Umhängen eines einzigen Zeigers ist in Abb. 45 dargestellt. Dieses Umhängen kann prinzipiell reduziert werden zu einem Löschen der alten und Erzeugen einer neuen Kante.

Um das Arbeiten mit Graphersetzungen und insbesondere mit PROGRES einzuführen, werden wir im folgenden Abschnitt beispielhafte, aber typische Implementierungen der beschriebenen Basisoperationen zeigen, bevor wir komplexere Spezifikationen betrachten.

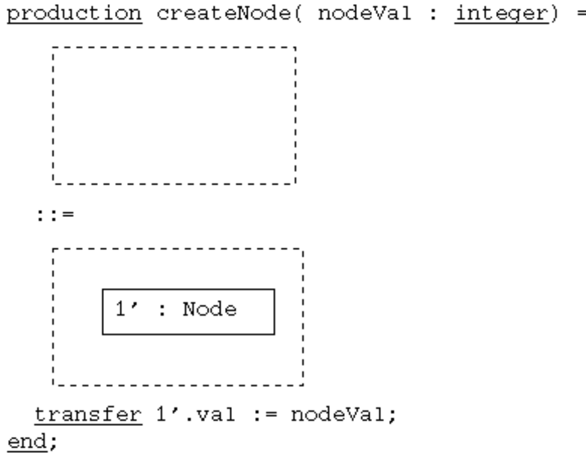


Abbildung 46: Erzeugen eines Knotens in PROGRES

6.3. Implementierung grundlegender Operationen in PROGRES

Für die Beschreibung von Operationen auf Graphen liefern Graphersetzungen die formale Beschreibung dessen, was in einem definierten Teilschritt bei der Erzeugung oder Manipulation einer Datenstruktur passiert. Dabei können wir unterscheiden zwischen Basisoperationen, wie dem Erzeugen eines Knotens oder einer Kante, dem Ändern eines Attributwertes etc., und komplexeren Operationen, die eher typisch für die jeweilige Datenstruktur sind und mit denen man auch Schnittstellenoperationen oder komplexe Teilschritte eines Algorithmus modellieren und spezifizieren kann. Für solche Spezifikationen haben wir in Kapitel 5 erste Beispiele gegeben und werden weitere in diesem Kapitel zeigen. Zunächst werden einige Beispiele für einfache, generische Operationen in PROGRES dargestellt.

Wir werden also an dieser Stelle etwas genauer auf die PROGRES-Sprache eingehen, soweit es im Rahmen dieser Arbeit sinnvoll und notwendig ist. Für eine detailliertere Beschreibung der PROGRES-Sprache sei auf [Sch96] verwiesen, die Definition einer formalen Semantik der PROGRES-Sprache in einer älteren Version findet der interessierte Leser in [Sch91]. Für die Nutzung der in dieser Arbeit vorgestellten Visualisierungsumgebung reicht es jedoch völlig aus, einige vorgestellte PROGRES-Konstrukte auf Benutzer-Niveau anwenden zu können.

Eine der einfachsten Graphtransformationen ist die Erzeugung eines Knotens, die in Abb. 46 in Form einer PROGRES-Produktion *createNode* gezeigt wird. Hier wird ein Knoten eines gegebenen Typs *Node* mit einem im Parameter *nodeVal* übergebenen Wert erzeugt. Dabei wird vorausgesetzt, dass der Knotentyp *Node* ein ganzzahliges Attribut *val* besitzt.

Die informelle Interpretation ist die Beschreibung des Zustands der Datenstruktur vor und nach der Produktion durch die vor und nach dem Zeichen `::=` angegebenen Graphschemata. Vereinfacht ausgedrückt passiert im Laufe der Bearbeitung einer Produktion durch das PROGRES-Laufzeitsystem Folgendes (wir nennen ab jetzt das obere gestrichelte Rechteck einer Produktion auch *linke Seite*, das untere gestrichelte Rechteck entsprechend *rechte Seite*):

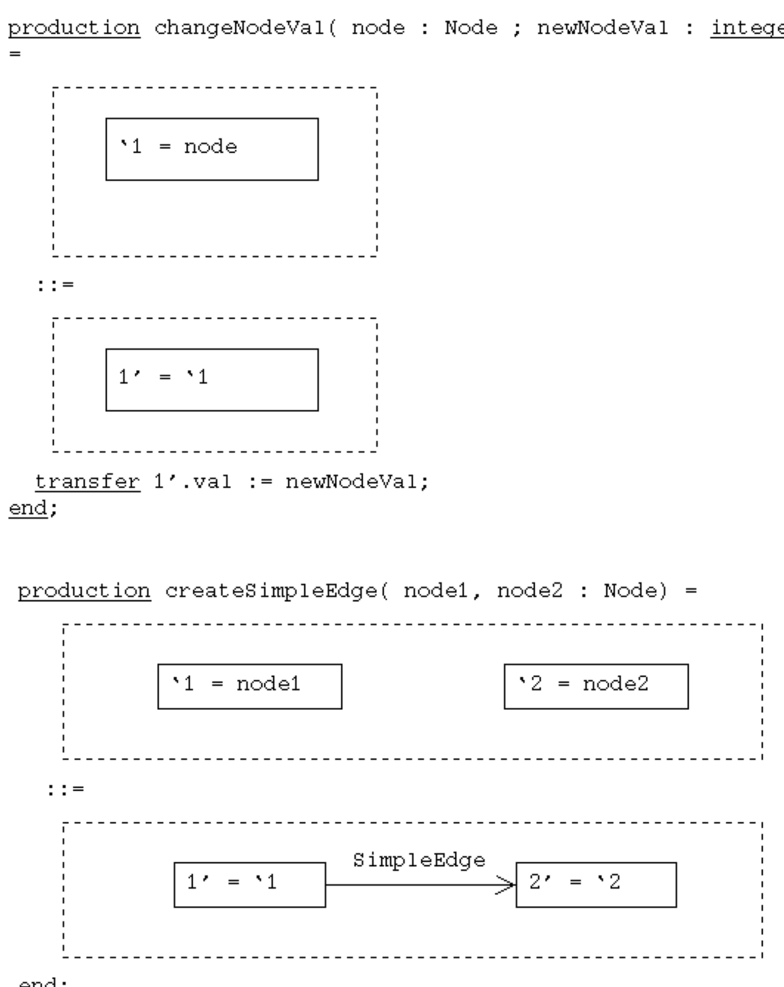


Abbildung 47: Oben: Ändern eines Knotenattributs. Unten: Erzeugen einer einfachen Kante.

1. Finde ein Vorkommen der linken Seite.
2. Entferne diejenigen Elemente der linken Seite, die kein Äquivalent auf der rechten Seite haben.
3. Erzeuge die Elemente der rechten Seite, die kein Äquivalent auf der linken Seite haben.
4. Berechne die notwendigen Attributwerte.

In Falle der Produktion *createNode* heißt das, dass die Produktion auf jede Datenstruktur anwendbar ist, da für die leere Menge trivialerweise immer ein Vorkommen gefunden wird. Es wird nichts aus den bestehenden Strukturen gelöscht, und es wird ein zusätzlicher Knoten neu erzeugt. Dieser Knoten besitzt ein *val*-Attribut, dessen Wert auf den Wert des übergebenen Parameters *nodeVal* gesetzt wird.

```
section Type_Declarations

  declares

    node class OBJECT
      intrinsic
        val : integer;
    end;

    node type Node : OBJECT end;

    edge type SimpleEdge : Node -> Node;

    node type ComplexEdge : OBJECT
      intrinsic
        source : Node;
        target : Node;
    end;

end;
```

Abbildung 48: Links: Die zu den Spezifikationen in den Abb. 46, 47 und 49 gehörigen Typdeklarationen in PROGRES. Rechts: Ein Beispiel-Graph mit den Objekten der Spezifikation auf der linken Seite.

Wie in früheren Beispielen von PROGRES-Programmstücken sieht man, dass die Knotenmarkierungen durch natürliche Zahlen erfolgen, denen ein ‘ vor- oder ein ´ nachgestellt ist. Ein Apostroph ´ wie in Abb. 46 markiert dabei ein Objekt 1´ *nach* der Operation und kann – wie in späteren Beispielen, z. B. in Abb. 47 zu sehen – mit einem Objekt *vor* der Operation, z. B. markiert mit ‘1, identifiziert werden.

Es kann nicht a priori davon ausgegangen werden, dass die Zuordnungen der mit diesen Knotenmarkierungen versehenen Platzhalter der Nummerierung entspricht, obwohl dies oft so gewählt wird, also bezeichnen z. B. die Markierungen ‘1 und 1´ nicht zwingend das gleiche Objekt im Graphen.

Die optionale `transfer`-Klausel beschreibt die abschließende Zuweisung des übergebenen aktuellen Parameterwertes von *nodeVal* an das *val*-Attribut des neuen Knotens 1´.

Abb. 47 oben zeigt eine etwas kompliziertere Produktion *changeNodeVal*, in der einem existierenden Knotenattribut ein neuer Wert zugewiesen wird. Die zwei Parameter der Produktion sind der Knoten *node* selbst und sein neuer Wert *newNodeVal*. Wie man sieht, muss zur Anwendbarkeit von *changeNodeVal* bereits ein Knoten ‘1 existieren, der gleich dem übergebenen Parameter *node* ist. Neu ist die Verwendung eines Parameters vom Typ *Node*, also eines knotenwertigen Parameters. Genauso gut hätten wir natürlich einen Schlüssel von einem einfacheren Typ verwenden können (hier z. B. den Wert des Knotens vor der Operation), wenn dieser Schlüssel eindeutig ist oder ein nichtdeterministisches Verhalten an dieser Stelle nicht stört.

Wie wir später noch sehen werden, eröffnet die Verwendung des Typs *Node* die Möglichkeit, den Parameter zur Laufzeit von VIDEA füllen zu lassen. Genauer gesagt kann der Benutzer beim Arbeiten mit einer VIDEA-Instanz einen Knoten selektieren und dann etwa *changeNode-*

```
production createComplexEdge( node1, node2 : Node ;
                              complexEdgeVal : integer)
```

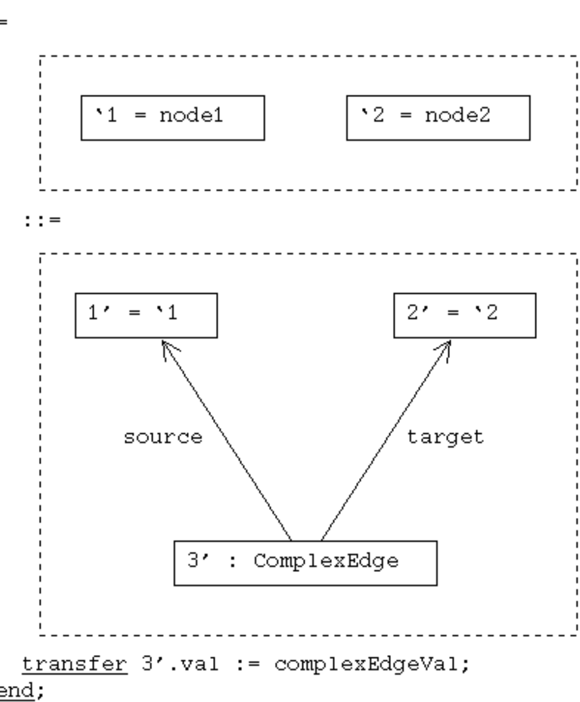


Abbildung 49: Erzeugen einer attributierten Kante in PROGRES in Form eines Knotens vom Knotentyp *ComplexEdge*

Val direkt als Operation auf dem markierten Knotenobjekt (bzw. dessen Pendant in der Graphdatenbank) aufrufen. Der ganzzahlige Wert *newNodeVal* muss dann natürlich noch zusätzlich spezifiziert werden. In der `transfer`-Klausel wird – ebenso wie schon in *createNode* – der Wert des Knoten nach Ausführung der Operation neu gesetzt.

Die Erzeugung einer Kante mithilfe einer PROGRES-Produktion ist in Abb. 47 unten zu sehen. Hier wurden als Parameter zwei Knoten *node1* und *node2* gewählt, zwischen denen die Kante erzeugt werden soll. Es wird verlangt, dass vor Ausführung der Operation zwei Knoten im Graphen existieren, die den beiden Parametern entsprechen (siehe oberer Teil des Rumpfes). Danach wird eine Kante vom Typ *SimpleEdge*, zu sehen im unteren Teil des Rumpfes von Abb. 47 unten, eingefügt. Die bisher und auch noch im Folgenden benutzten PROGRES-Typdeklarationen für *Node, SimpleEdge* und den anschließend verwendeten Typ *ComplexEdge* zeigt Abb. 48 links. Wie man sieht, gibt es eine generische Knotenklasse (in PROGRES *node class* genannt) *OBJECT*, die ein eigenständiges (in PROGRES `intrinsic`) Attribut *val* hat und dieses an die Knotentypen *Node* und *ComplexEdge* vererbt. „Eigenständig" meint in diesem Zusammenhang, dass das Attribut ab dem Zeitpunkt der Erzeugung des Knotens existiert und sein Wert durch Wertzuweisungen geändert oder durch Nutzung in anderen Konstrukten ausgelesen werden kann.

Ein einfacher Kantentyp wird in Abb. 48 links in Form des schon erwähnten Kantentypen *SimpleEdge* einfach als Kante zwischen zwei Knoten vom Typ *Node* definiert. Das Schlüsselwort `section` in Abb. 48 links leitet einen Abschnitt in der PROGRES-Spezifikation ein, ein einfaches Mittel ohne eigene Semantik, um große Spezifikationen zu strukturieren.

Der ungewöhnlich erscheinende Typ *ComplexEdge* dient dazu, mithilfe eines Knotentyps eine Kante mit Attributierung zu simulieren. Hintergrund ist, dass PROGRES a priori nur Attribute für Knoten, nicht aber für Kanten zulässt; es sei hier nur der Vollständigkeit halber erwähnt, dass UPGRADE2 mithilfe eines sog. Edge-Node-Edge-Filters solche Objekte direkt als Kanten darstellen kann. Diese Fähigkeit von UPGRADE2 wird in VIDEA genutzt, so dass wir für den Knotentyp *ComplexEdge* zwei zusätzliche einfache Kantentypen *source* und *target* einführen, die auf den Quell- und Zielknoten der emulierten Kante verweisen. Ein beispielhafter Graph, der die Objekttypen aus Abb. 48 links im Zusammenspiel zeigt, ist in Abb. 48 rechts zu sehen.

In Abb. 49 sehen wir, wie mithilfe einer Produktion *createComplexEdge* ein Objekt vom Typ *ComplexEdge* zwischen zwei gegebenen Knoten erzeugt wird.

Wie man in den Beispielen bisher schon sah, werden in PROGRES Teile der Spezifikation textuell, andere visuell angegeben. Analog zu üblichen textuellen Programmiersprachen wie Pascal, C oder Java bietet PROGRES auch die Möglichkeit, Kontrollstrukturen anzugeben (siehe Abb. 50). In diesen können dann wiederum Transaktionen, Produktionen oder einfache Statements, wie z. B. Zuweisungen, aufgerufen werden.

Eine Transaktion in PROGRES ist eine Programmeinheit, die die Nutzung mehrerer Produktionen beinhalten kann, und die die ACID-Eigenschaften von Datenbanktransaktionen gewährleistet: Atomarität (*atomicy*), Konsistenz (*consistency*), Isolation (*isolation*) und Dauerhaftigkeit (*duration*). Anders als bei Datenbanktransaktionen kann das Backtracking der PROGRES-Graphdatenbank *GRAS* erfolgreich beendete Transaktionen wieder zurücksetzen.

Abb. 50 zeigt die Nutzung der Kontrollstruktur `choose ...else ...end` für die Implementierung einer einfachen Fallunterscheidung. In diesem Beispiel wird bei Übergabe eines negativen Parameters *nodeVal* an die Transaktion *handleNode* der Knoten mit dem Absolutwert von *nodeVal* gelöscht, im Falle eines positiven Parameters *nodeVal* wird ein Knoten dieses Wertes neu erzeugt.

Die bisher beschriebenen PROGRES-Sprachmittel reichen aus, um einen groben Eindruck der Sprache PROGRES zu vermitteln, soweit er als Grundlage für das Verständnis der kommenden Abschnitte notwendig sein wird. Deshalb gehen wir nun dazu über, Spezifikationen unserer vier gewählten Beispiele für Algorithmen und Datenstrukturen in PROGRES zu zeigen.

6.4. Spezifikationen für den AVL-Baum

Zwei grundlegende Teile der AVL-Baum-Spezifikation wurden bereits bei der Einführung in die Methodik in Kapitel 5 vorgestellt, nämlich die Spezifikation des verwendeten Knotentyps *AVLNode* und der Linksrotation. Die Definition des Knotentyps *AVLNode* ist noch einmal in Abb. 51 oben zu sehen.

Das Attribut *val* ist in jedem Knoten vom Typ *AVLNode* enthalten und wird somit als obligatorisches eigenständiges Attribut definiert, d. h. jeder Knoten dieses Knotentyps hat zu jedem Zeitpunkt seiner Existenz einen definierten Attributwert für das Attribut *val*. Die beiden möglichen Kinder *left* und *right* sind optional, wie man an der Kardinalität *[0:1]* sehen kann. Das

```
transaction handleNode( nodeVal : integer) =
  choose
    when (nodeVal < 0)
    then
      deleteNode ( - nodeVal )
  else
    createNode ( nodeVal )
  end
end;
```

Abbildung 50: Beispiel für eine einfache Kontrollstruktur: *handleNode*

bedeutet, dass ein Knoten ein linkes bzw. rechtes Kind haben kann, aber nicht muss. Die Definition als eigenständige Attribute vom Typ *AVLNode* selbst entspricht der impliziten Definition eines einfachen Kantentyps *AVLNode -> AVLNode*.

Die restlichen Attribute sind abgeleitet (in PROGRES: `derived`) von bereits vorhandenen Informationen wie anderen Knoten und Kanten und deren Attributen.

Das Attribut *height* berechnet für jeden Knoten die Höhe desjenigen Teilbaums des Gesamtbaumes, dessen Wurzel der jeweilige Knoten ist. Definiert ist der Wert des *height*-Attributs rekursiv, nämlich als das Maximum der Höhen des linken und des rechten Unterbaums des betrachteten Knotens plus eins. Da der aktuelle Knoten in PROGRES durch das Schlüsselwort `self` referenziert werden kann, ist z. B. `self.left.height` die Höhe des linken Unterbaums oder, anders ausgedrückt, die Höhe des linken Kindknotens, deren Berechnung ja wieder analog zur Berechnung der Höhe des aktuellen Knotens abläuft. Da es möglich ist, dass eines der beiden Kinder oder sogar beide nicht existieren, wurde die Hilfsfunktion *Math_mkHeight* eingeführt, die für eine Zahl i die Zahl selbst und für eine undefinierte Zahl (also in unserem Fall einem undefinierten Kindknoten) den Wert -1 zurückliefert. Auf diese Weise wird auch der Abbruchfall der rekursiven Berechnungsvorschrift für die Höhe eines Binärbaums so implementiert wie in Kapitel 5 für Binär- und AVL-Bäume eingeführt.

Falls ein Dozent eine etwas andere Definition eines AVL-Baumes benutzt, z. B. eine Höhe von 0 für den leeren Baum, so genügt es, die PROGRES-Spezifikation an dieser einen Stelle zu ändern und daraus eine angepasste VIDEA-Instanz zu generieren.

Für die Definition der beschriebenen Funktion *Math_mkHeight* (siehe Abb. 51 unten) verwenden wir den Operator `[ a | b ]`, der a zurückgibt, falls a definiert ist, sonst b.

Die Definitionen der ebenfalls in Abb. 51 unten gezeigten Funktionen *Math_min* und *Math_max* nutzen eine erweiterte Variante des `[]`-Operators, bei der für eine boolesche Bedingung b und zwei Werte $i1$ und $i2$ bei der Nutzung als `[b :: i1 | i2]` den Wert $i1$ zurückliefert, falls $b = true$, und $i2$ sonst.

Auch beim Attribut *balance* kann einfach folgende Definition hingeschrieben werden: Der Balancefaktor ist definiert als die Höhe des linken minus der Höhe des rechten Unterbaums.

Das nächste wichtige Attribut *isSorted* soll anzeigen, ob der Teilbaum, der den aktuellen Knoten als Wurzel enthält, gemäß der Sortierungseigenschaft für Binärbäume korrekt sortiert

```
node type AVLNode : OBJECT
  intrinsic
    val : integer;
    left : AVLNode [0:1];
    right : AVLNode [0:1];
  derived
    height : integer
      = Math_max ( Math_mkHeight ( self.left.height ),
                   Math_mkHeight ( self.right.height ) ) + 1;
    balance : integer = Math_mkHeight ( self.left.height )
                        - Math_mkHeight ( self.right.height );
    minVal : integer
      =
        Math_min
        ( Math_maxInt,
          all (self.left.minVal or self.right.minVal or self.val) );
    maxVal : integer
      =
        Math_max
        ( Math_minInt,
          all (self.left.maxVal or self.right.maxVal or self.val) );
    isSorted : boolean = ([ self.left.maxVal
                          | Math_minInt ]    <= self.val)
                         and (self.val <= [ self.right.minVal
                                          | Math_maxInt ]    );
    noTree : boolean
      = card ( self.<-left- or self.<-right- ) > 1;
end;
```

```
function Math_min : ( i1, i2 : integer) -> integer =
  [ i1 < i2 :: i1
  | i2 ]
end;

function Math_max : ( i1, i2 : integer) -> integer =
  [ i1 < i2 :: i2
  | i1 ]
end;

path child : AVLNode [0:1] -> AVLNode [0:n] =
      -left->
  or  -right->
end;

path parent : AVLNode [0:1] -> AVLNode [0:n] =
  [ <-left-
  | <-right- ]
end;

function Math_mkHeight : ( i : integer [0:1]) -> integer =
  [ i
  | - 1 ]
end;
```

Abbildung 51: Spezifikationen für den AVL-Baum. Oben: Nochmal die Spezifikation des *AVL-Node*-Knoten-Typs selbst. Unten: Hilfsdeklarationen

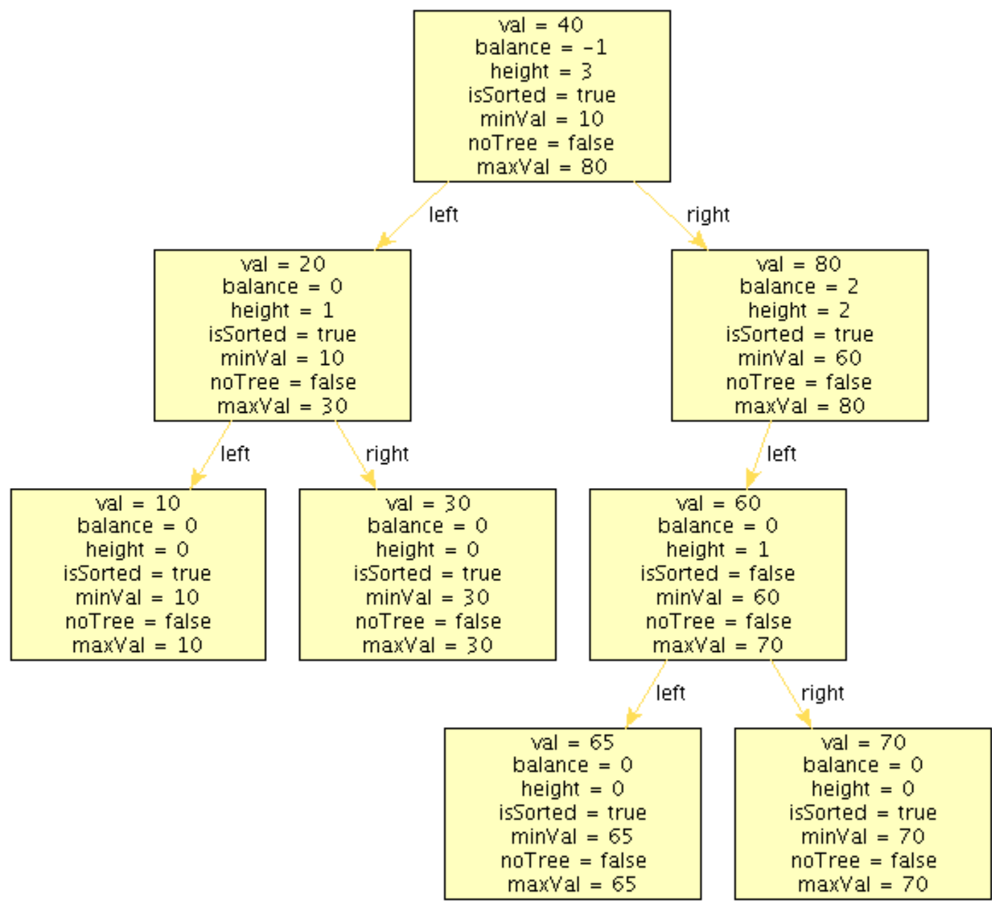


Abbildung 52: Ein unbalancierter AVL-Baum mit allen in Abb. 51 oben definierten Attributen

ist. Dieses Attribut wird eingeführt, weil die Sortierung in einem Baum in dieser VIDEA-Instanz einerseits temporär verletzbar sein soll, andererseits eine mögliche Verletzung in irgendeiner Form markiert werden soll. Zusätzlich wollen wir die Möglichkeit anbieten, Lernenden konkrete Fragen nach der Sortierung stellen zu können. Ein Knoten heißt in unserer Definition *sortiert*, wenn der größte Wert im linken Unterbaum kleiner oder gleich dem Wert der Wurzel und der Wert der Wurzel kleiner oder gleich dem kleinsten Wert im rechten Unterbaum ist. Genauso ist das Attribut *isSorted* in Abb. 51 oben definiert. Dabei muss einerseits wieder berücksichtigt werden, dass bis zu zwei Unterbäume undefiniert, also nicht existent, sein können. Andererseits sieht man in Abb. 51 oben, dass wir zwei Hilfsattribute *minVal* und *maxVal* einführen, die für den aktuellen Knoten den minimalen bzw. maximalen Wert in demjenigen Teilbaum zurückliefern, der durch den aktuellen Knoten als Wurzel aufgespannt wird. Abb. 52 zeigt zur Erleichterung des Verständnisses für einen (absichtlich unbalancierten und unsortierten) Beispiel-Baum alle

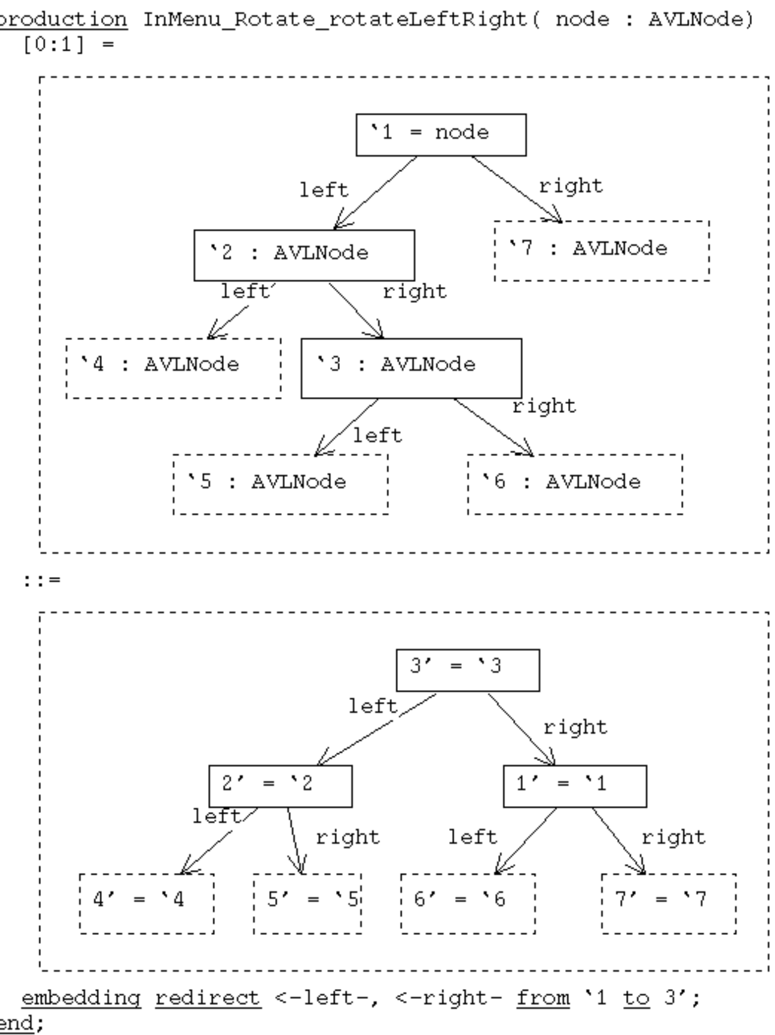


Abbildung 53: Links-Rechts-Doppelrotation des AVL-Baums

für *AVLTree* eingeführten Attribute, auch die normalerweise nicht angezeigten Attribute *minVal* und *maxVal*.

Man beachte, dass sich bei unserer Definition eventuelle Verletzungen der Sortierungs- oder Balancierungseigenschaft nicht durch den gesamten Baum weiterpropagieren, sondern lokale Aussagen über die verletzten Eigenschaften sind. Auf diese Weise können leichter Fragen nach den entsprechenden Stellen im Baum an die Lernenden gestellt und von diesen beantwortet werden. Zum Beispiel gibt es im Baum in Abb. 52 durch die falsche Einsortierung von Knoten $val = 65$ nur *eine* Stelle, an der die Sortierungseigenschaft verletzt ist, und zwar bei seinem Elternknoten $val = 60$. Hätte der falsch einsortierte Knoten z. B. den Wert 85 gehabt, dann wäre zusätzlich die Sortierungseigenschaft des Knotens $val = 80$ verletzt gewesen und so weiter.

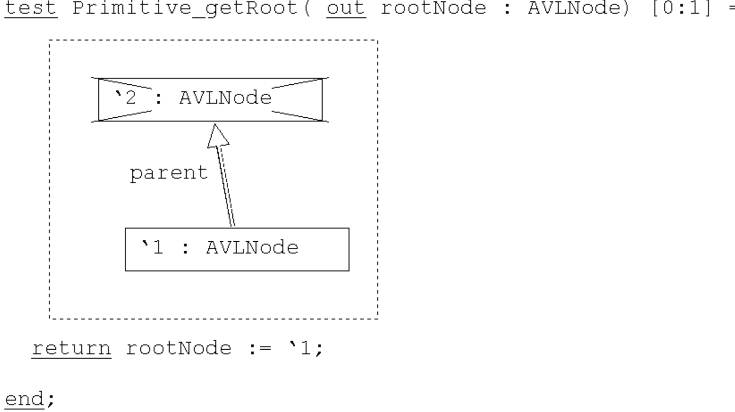


Abbildung 54: Finden der Wurzel eines AVL-Baums

Für die Spezifikation von *minVal* und *maxVal* verwenden wir den Mengenkonstruktor `all`, der die Vereinigung aus den mit `or` verknüpften Werten bildet. Im Falle von *minVal* wird also die höchstens dreielementige Menge aus *self.left.minVal*, *self.right.minVal* und *self.val* gebildet. Schließlich wird mithilfe von *Math_min* und *Math_maxInt* das minimale Element dieser Menge als Wert von *minVal* festgelegt. Analoges gilt für *maxVal*.

Das letzte Attribut *noTree* schließlich gibt an, ob überhaupt ein Baum vorliegt. Dieses Attribut wurde primär für die Nutzung dieser VIDEA-Instanz als Simulation eines visuellen Debuggings eigener Programme der Lernenden eingeführt, da es beim Programmieren mit Zeigern schnell geschehen kann, den Baum völlig zu zerstören. Für die Definition von *noTree* wird mithilfe des beschriebenen `or`-Operators eine Menge von einlaufenden Kanten gebildet. Es handelt sich dabei um die in den aktuellen Knoten `self` einlaufenden *left*-Kanten `self.<-left-` und die einlaufenden *right*-Kanten `self.<-right-`. Die Baum-Eigenschaft ist u. a. dann verletzt, wenn die Mächtigkeit *card* dieser Menge von Kanten größer als eins ist. Bei der Angabe der einlaufenden Kanten handelt es sich um einfache Formen so genannter Pfadausdrücke, die auch als eigenes Konstrukt verwendet werden können wie in in Abb. 51 unten als Definition zweier Pfad-Namen *parent* und *child*. Im Laufe der weiteren Beschreibung werden wir Beispiele für deren Anwendung sehen.

Als Beispiel für die Spezifizierung einer Schnittstellenoperation haben wir in Kapitel 5 die Linksrotation gezeigt. Nun sei ein weiteres Beispiel vorgestellt, anhand dessen wir nochmal den Ablauf der vier Schritte bei Ausführung einer PROGRES-Produktion nachvollziehen: Die Links-Rechts-Doppelrotation.

In *InMenu_Rotate_rotateLeftRight* (Abb. 53) wird zuerst ein Vorkommen der obligatorischen Knoten `1, `2 und `3 gesucht, wobei `1 gleich dem übergebenen Parameter *node* ist, und sowohl die *left*-Kante zwischen `1 und `2, als auch die *right*-Kante zwischen `2 und `3 existieren muss. Die restlichen Elemente sind optional und werden deswegen nur im Fall des Vorhandenseins als Bestandteile des Vorkommens weiterverarbeitet.

Dann werden die vorhandenen der vier nur auf der linken Seite vorkommenden Kanten (`1 -> `2, `2 -> `3, `3 -> `5 und `3 -> `6) gelöscht.

Danach werden die vier nur auf der rechten Seite vorkommenden Kanten (3´ -> 2´, 3´ -> 1´, 2´ -> 5´ und 1´ -> 6´) erzeugt, soweit die jeweiligen Knoten vorliegen. Außerdem wird entsprechend der Angabe der `embedding`-Klausel eine evtl. in `1 einlaufende Kante nach 3´ umgeleitet.

Zuletzt werden die abgeleiteten Attribute *height*, *balance*, *minVal*, *maxVal*, *isSorted* und *noTree*, wo das notwendig ist, neu berechnet.

Abb. 54 zeigt eine beispielhafte Verwendung des weiter oben definierten Pfadausdrucks *parent*, der einer *left*- oder *right*-Kante in der Gegenrichtung folgt (also zum Elternknoten hin). *Primitive_getRoot* liefert denjenigen Knoten zurück, der keinen Elternknoten hat, also die Wurzel. Außer der Verwendung eines Pfadausdrucks zeigt Abb.54 auch zum ersten Mal die Verwendung eines negativen Knotens (`2), dessen Nicht-Existenz im aktuellen Graphen die Voraussetzung für das Finden der Passstelle der Produktion ist.

Für *Primitive_getRoot* wurde das Konstrukt des PROGRES-Tests gewählt. Ein Test in PROGRES kann als vereinfachte Produktion gesehen werden, da zwar eine passende Stelle im Graphen gesucht wird, aber keine Veränderungen im Graphen vorgenommen werden.

Die Funktion *InMenu_BinTree_insert* mit der Hilfsfunktion *BinTree_insertInTree* in Abb. 55 zeigt wieder ein Beispiel für ein Programmstück, das in rein textueller Schreibweise angegeben wurde. Neu ist hier die Verwendung der Rekursion, die analog zu bekannten textuellen Sprachen eingesetzt wird. Das Einfügen eines Elements in einen Suchbaum wird in *InMenu_BinTree_insert* bewerkstelligt, indem zuerst die Wurzel des Gesamtbaums gesucht und dann eine Hilfsfunktion aufgerufen wird, die von dort aus die passende Stelle zum Einfügen des neuen Elements findet und das Einfügen durchführt. Falls dieser Ablauf fehlschlägt, wird im `else`-Zweig von *InMenu_BinTree_insert* ein Knoten mit dem gewünschten Wert erzeugt.

BinTree_insertInTree prüft, ob der übergebene Knoten bereits den einzufügenden Wert hat. Wenn ja, wird nichts getan; wir fügen also keine doppelten Werte im Suchbaum ein. Im anderen Fall wird entsprechend der Suchbaum-Eigenschaft je nachdem rekursiv links oder rechts weitergesucht und – wenn ein Blatt erreicht wird – ein neuer Knoten mit dem einzufügenden Wert erzeugt. Dieser wird schließlich mit einer *left*- oder *right*-Kante an seinen neuen Elternknoten angehängt.

Ein anschauliches Beispiel für die Verwendung eines PROGRES-Tests für die Überprüfung des Verständnisses von Lerninhalten zeigt Abb. 56.

Hier werden als Parameter ein Knoten und ein `integer`-Wert übergeben. Der Test *InMenu_Exercise_checkBalance* überprüft nur, ob der übergebene `integer`-Wert dem Balancefaktor des übergebenen Knotens entspricht. Im positiven Fall ist der Test erfolgreich, sonst schlägt er fehl. Benutzt wird dieses Konstrukt zur Laufzeit der VIDEA-Instanz zum Beispiel in der Art, dass der Lernende einen Knoten im Baum anklickt und nach dem Balancewert gefragt wird; zumindest erscheint ihm der Aufruf des hier beschriebenen Tests so. Je nach Eingabe sieht er eine bestätigende Meldung oder eine Fehlermeldung mit angezeigter Definition des Balancefaktors eines Knotens in einem AVL-Baum. Beispiele für die Außensicht von VIDEA, die auch die gerade beschriebenen Abläufe umfassen, werden ausführlich in Kapitel 7 beschrieben.

```
transaction InMenu_BinTree_insert( nodeVal : integer) [1:1]
=
  use t : AVLNode
  do
    choose
        Primitive_getRoot ( out t )
      & BinTree_insertInTree ( t, nodeVal )
    else
      InMenu_Primitive_makeNode ( nodeVal, out t )
    end
  end
end;

transaction BinTree_insertInTree( nodeVal : AVLNode ; v : integer)
  [1:1] =
  use tl, tr : AVLNode
  do
    choose
      when (v = nodeVal.val)
      then
        skip
    else
      when (v < nodeVal.val)
      then
        choose
          BinTree_insertInTree ( nodeVal.left, v )
        else
            InMenu_Primitive_makeNode ( v, out tl )
          & InMenu_Exercise_setLeft ( nodeVal, tl )
        end
    else
      choose
        BinTree_insertInTree ( nodeVal.right, v )
      else
          InMenu_Primitive_makeNode ( v, out tr )
        & InMenu_Exercise_setRight ( nodeVal, tr )
      end
    end
  end
end;
```

Abbildung 55: Einfügen in einen AVL-Baum

Die zugehörige, vollständige PROGRES-Spezifikation kann – zusammen mit allen folgenden Spezifikationen – von [PRO05] bezogen werden.

An dieser Stelle gehen wir nun über zur Beschreibung der Spezifikation einer VIDEA-Instanz für die doppelt verkettete Liste.

```
test InMenu_Exercise_checkBalance( node : AVLNode ; balanceFactor : integer)
=

    '1 = node

  condition '1.balance = balanceFactor;

end;
```

Abbildung 56: Test des Verständnisses bei der Angabe des Balancefaktors eines Knotens

6.5. Spezifikationen für die doppelt verkettete, sortierte Ringliste

Ähnlich wie bei Spezifikation der AVL-Bäume im letzten Abschnitt beginnen wir wieder mit der Spezifikation der Knoten- und Kantentypen. In Abb. 57 ist die Verwendung der im letzten Kapitel vorgeschlagenen zwei Knotentypen gezeigt: Der Listenkopf *ListHead* kommt nur einmal in einer Liste vor, was wir durch die Implementierung von *createListHead* (s.u.) sicherstellen. Das eigentliche Listenelement *ListElem* enthält einen Wert *val* und besitzt einen Zeiger *previous* zum vorhergehenden Listenelement und einen Zeiger *next* zum folgenden Listenelement.

Der Listenkopf verfügt neben der Anzahl der existierenden Listenelemente in seiner Liste in *noOfElems* auch über bis zu vier Zeiger: *First* zeigt auf das erste Element der Liste, *last* auf das letzte; bei einer einelementigen Liste zeigen beide auf das gleiche Element. Die Zeiger *current* und *new* sind zwei im letzten Kapitel nicht erwähnte Zusätze, die dazu dienen, den im Folgenden implementierten Einfüge-Algorithmus visuell ansprechend und übersichtlich darstellen zu können. Zusätzlich haben beide Knotentypen auch berechnete Elemente:

ListHead führt für interne Zwecke die Gesamtzahl der Knoten in *noOfNodes* mit, die in diesem einfachen Fall einer einzigen Liste mit nur einem Listenkopf einfach die Zahl der Elemente plus eins für den Listenkopf ist. Dieses Attribut wird in dieser Arbeit nicht weiter verwendet, kann aber z. B. genutzt werden, um Verständnistests für die Lernenden zu kreieren.

Das Listenelement hat folgende zur Laufzeit berechnete boolesche Attribute:

- *isCurrent* beschreibt, ob auf diesen Knoten eine *current*-Kante trifft,
- *isFirst* bzw. *isLast* geben an, ob der Knoten eine eingehende *first*- bzw. *last*-Kante hat,
- *isLocallyUnsorted* zeigt, ob die lokale Sortierung durch einen größeren Wert des unmittelbaren Vorgänger-Elementes oder einen kleineren Wert des unmittelbaren Nachfolger-Elementes verletzt ist und
- *dllBroken* zeigt, ob die vollständige doppelte Verkettung (durch *previous*- und *next*-Kanten) irgendwo unterbrochen ist.

Zur Definition von *isLocallyUnsorted* lässt sich sagen, dass wir einen Knoten als „unsortiert“ markieren, wenn er relativ zu seinem Vorgängerknoten *oder* seinem Nachfolgerknoten – jeweils

```
node class OBJECT end;

node type ListHead : OBJECT
  intrinsic
    noOfElems : integer := 0;
    new : ListElem [0:1];
    first : ListElem [0:1];
    current : ListElem [0:1];
    last : ListElem [0:1];
  derived
    noOfNodes = self.noOfElems + 1;
end;

node type ListElem : OBJECT
  intrinsic
    val : integer := - 1;
    next : ListElem [0:1];
    previous : ListElem [0:1];
  derived
    isCurrent : boolean = card ( self.<-current- ) > 0;
    isFirst : boolean = card ( self.<-first- ) > 0;
    isLast : boolean = card ( self.<-last- ) > 0;
    isLocallyUnsorted : boolean
      = (([ self.previous.val
          | minInt ]          > self.val) and not self.isFirst)
        or ((self.val > [ self.next.val
                        | maxInt ]     ) and not self.isLast)  ;
    dllBroken : boolean = not (self.previous.next = self)
                          or not (self.next.previous = self);
end;
```

Abbildung 57: Spezifikation der Knoten- und Kantentypen für die doppelt verkettete Liste

natürlich nur, falls existent – falsch einsortiert ist. Auf diese Weise werden bei einer auf *isLocallyUnsorted* basierenden farblichen Markierung einer unsortierten Liste immer mindestens zwei aufeinanderfolgende Knoten markiert. So wird erreicht, dass derjenige Knoten, der im größeren Kontext der Liste vom Lernenden als „fehlerhaft einsortiert“ erkannt wird, auf jeden Fall auch hervorgehoben ist. Da die doppelt verkettete Liste im Falle einer echten Ringliste zwischen dem letzten und dem ersten Element notwendigerweise die Sortierung verletzen muss, werden diese beiden Knoten bei der Definition von *isLocallyUnsorted* gesondert behandelt.

Die Definitionen von *isCurrent*, *isFirst* und *isLast* nutzen in unserer Spezifikation den gleichen strukturellen Aufbau: `card (self<-edge-)` repräsentiert die Kardinalität der Menge der in den aktuellen Knoten einlaufenden *edge*-Kanten, wobei *edge* hier für *first*, *current* oder *last* stehen kann. Beispielsweise wird *isCurrent* somit auf *true* gesetzt, wenn es mindestens eine in den aktuellen Knoten einlaufende *current*-Kante gibt. Alternativ könnte hier ein in den neuesten PROGRES-Versionen vorhandenes Konstrukt verwendet werden: `def(self.current)` hat den selben Effekt.

Das erste Beispiel einer Operation für die doppelt verkettete Liste ist die bereits erwähnte Produktion *InMenu_Primitive_createListHead*, die einen Listenkopf erzeugt und dabei sicher-

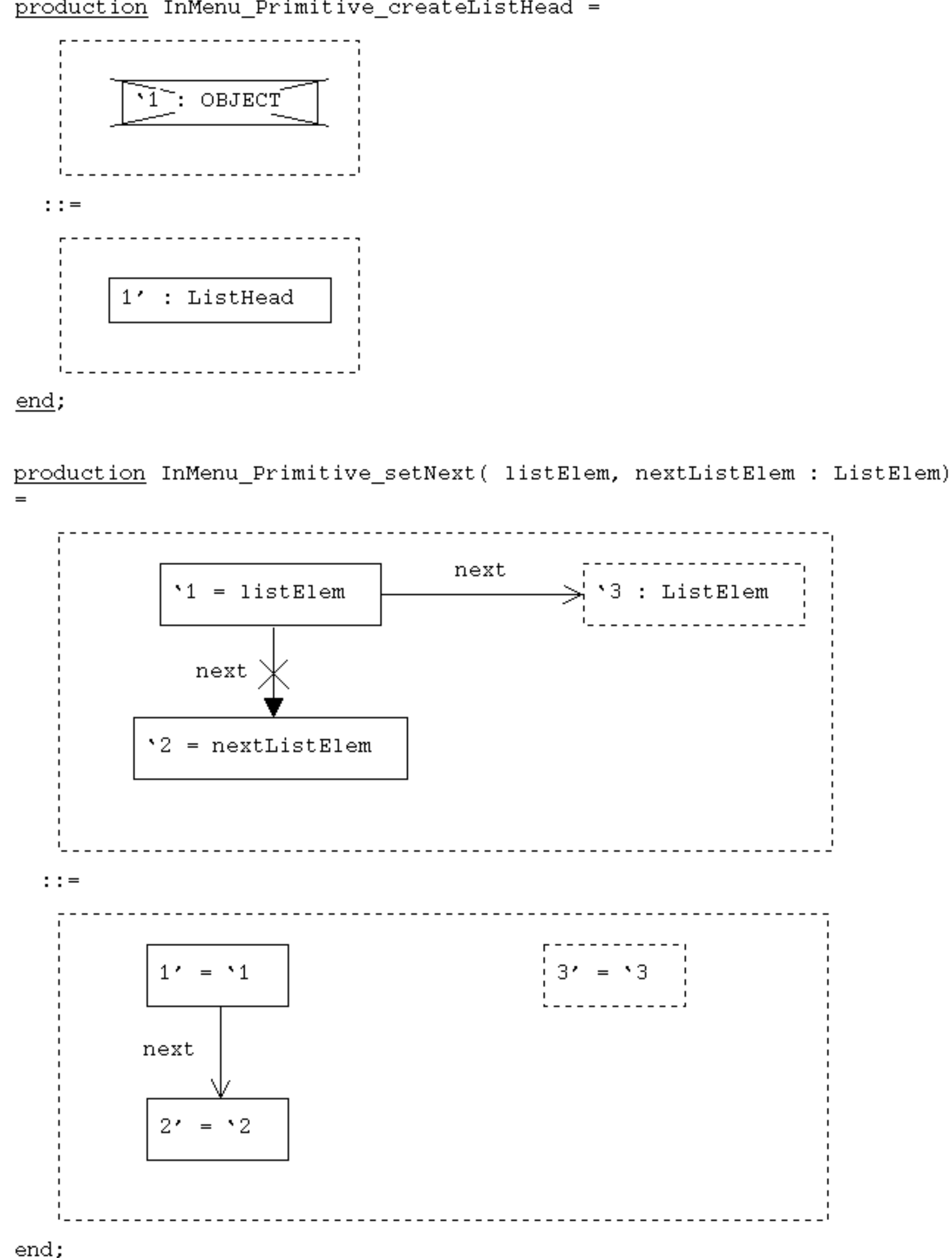


Abbildung 58: Oben: Die Erzeugung des Listenkopfes für eine Liste. Unten: Setzen einer *next*-Kante.

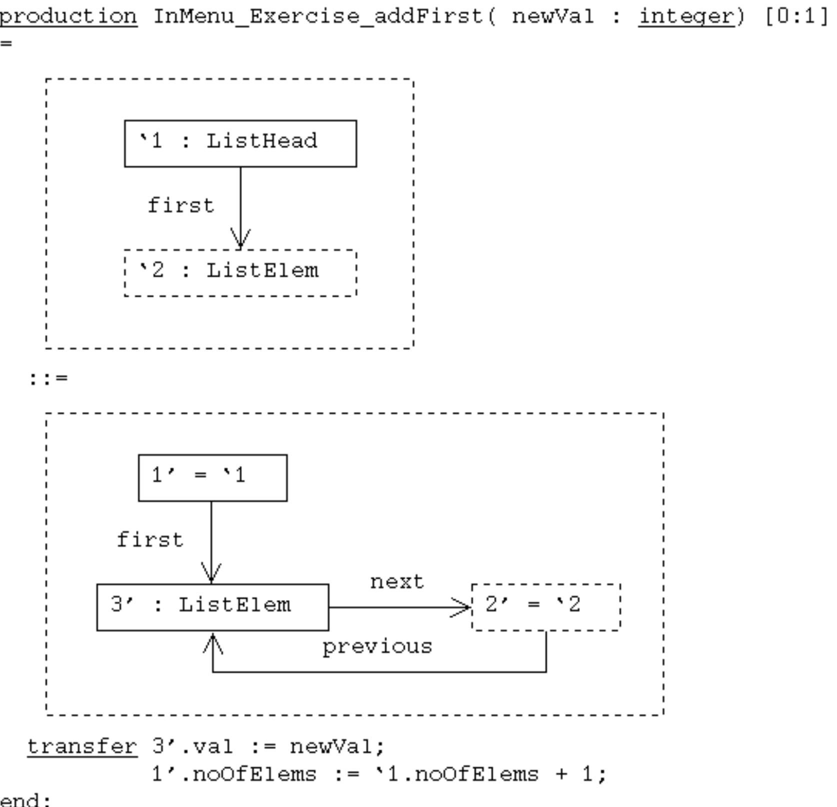


Abbildung 59: Spezifikation von *InMenu_Exercise_addFirst* für die doppelt verkettete Liste

stellt, dass im vorliegenden Graphen nicht bereits ein anderer existiert. In Abb. 58 oben sehen wir, dass die linke Regelseite einen negativen Knoten `1 vom Typ *OBJECT* enthält, was bedeutet, dass vor Anwendung der Produktion weder ein Knoten vom Typ *ListHead* noch ein Knoten vom Typ *ListElem* existieren darf, die ja beide von der Klasse *OBJECT* abgeleitet sind. Unter dieser Bedingung wird ein neuer Listenkopf mit leerer Liste erzeugt, was auch dadurch ausgedrückt ist, dass das Attribut *noOfElems* des Listenkopfes, wie in Abb. 57 zu sehen war, standardmäßig auf den Wert 0 gesetzt wird.

Ein nächstes Beispiel einer einfachen Operation ist die Produktion *InMenu_Primitive_setNext*, die im Menü *Primitive* zu finden ist und eine *next*-Kante zwischen zwei übergebenen Listenelementen einfügt. Wie in Abb. 58 unten zu sehen ist, wird einerseits überprüft, dass zwischen den beteiligten Elementen `1 und `2 noch keine *next*-Kante existiert, und andererseits eine evtl. bereits vorhandene *next*-Kante zwischen dem Ausgangs-Knoten und einem möglichen dritten Element gelöscht.

Zum Einfügen eines neuen Listenelements an erster Stelle einer existierenden Liste dient die PROGRES-Produktion *InMenu_Exercise_addFirst* im Menü *Exercise*. In Abb. 59 sehen wir, dass in der Beispiel-Spezifikation durch die Optionalität des *first*-Elements `2 offengelassen

Abbildung 60: Zwei Spezifikationen zum Löschen aller Listenelemente in einer Liste

wird, ob sich schon Elemente in der Liste befinden. Nur der Listenkopf `1 muss existieren. In jedem Fall wird das neue Element mit dem Wert *newVal* als neues *first*-Element 3´ eingetragen und das möglicherweise vorhandene bisherige erste Element der Liste `2 (vollständig verkettet mit *previous*- und *next*-Kante) auf Platz zwei verschoben. Gleichzeitig wird die Anzahl der vorhandenen Listenelemente im Listenkopf hochgezählt. In Abb. 59 wurde die korrekte Verwaltung des *last*-Zeigers noch ausgeklammert, da die Produktion sowohl in einem vorbereiteten Skript, als auch interaktiv bewusst für eine fehlende oder fehlerhafte *last*-Kante sorgen soll, um einen weiteren Lerneffekt anzustoßen.

Der Stern * in der Deklaration der Produktion *deleteAnyListElem* in Abb. 60 links führt zur nebenläufigen Abarbeitung aller möglichen Vorkommen des oberen gestrichelten Kastens (in diesem Fall also von Paaren des Listenkopfes `1 und eines beliebigen Listenelementes `2), wobei jeweils der Knoten `2 gelöscht wird. Insgesamt löscht also *deleteAnyListElem* alle Listenelemente in der gegebenen Liste. Allerdings ist diese Implementierung nicht optimal, weil die Wertänderung des Attributs *noOfElems* des Listenkopfes für jede Löschung eines Listenelementes durchgeführt wird. Deswegen ist die Lösung in Abb. 60 rechts in diesem Fall besser: Hier wird die Menge aller Listenelemente als Mengenknoten `2 dargestellt (erkennbar durch die Stapel-Darstellung) und in einem Schritt gelöscht. Das funktioniert auch im Falle nicht-verbundener Teillisten von Listenelementen im Graphen, die der Lernende trotz unseres Verbots, mehrere Listenköpfe zu erstellen, natürlich konstruieren könnte.

Eine Alternative wäre die Nutzung von Pfadausdrücken und das Löschen des ersten Elements gefolgt von allen mit *next* verketteten Elementen. Allerdings können wir nicht davon ausgehen, dass die Liste zum Zeitpunkt des Löschens sauber aufgebaut wurde; vielleicht fehlen

```
production prepareInsert( valueToBeInserted : integer) [0:1]
=
```

```
`5 : ListElem    last    `1 : ListHead    new
first    current    `4 : ListElem
`2 : ListElem    `3 : ListElem
```

```
::=
```

```
5' = `5    last    1' = `1    new    4' : ListElem
current    first
2' = `2
```

```
  transfer 4'.val := valueToBeInserted;
end;
```

Abbildung 61: Spezifikation von *prepareInsert* für die doppelt verkettete Liste

etwa mittlere *next*-Kanten oder die *first*-Kante. Auf diese Weise würde kein sicheres Löschen implementiert werden. Als zusätzliche Illustration eines Löschalgorithmus auf einer Liste mit Überprüfung für die Lernenden, ob ihre Liste korrekt aufgebaut war, könnte eine solche alternative Lösung allerdings interessant sein.

Nun wollen wir einen einfachen Algorithmus für das Einfügen eines neuen Elementes in die Liste vorstellen, der davon ausgeht, dass sich schon mindestens ein Element in der Liste befindet, und wie folgt arbeitet:

1. Bereite das Einfügen vor durch Erzeugen des einzufügenden Elements mit dem gegebenen Wert, einer Markierung dieses Elements als neu und einer Markierung des bisherigen ersten Listenelements als gegenwärtig betrachtete Stelle zum Einfügen.
2. Füge dieses neue Element an der richtigen Stelle voll-verzeigert ein (mit *previous*- und *next*-Kanten).
   - Finde dazu den korrekten Platz für das Einfügen des neuen Listenelements gemäß der geforderten aufsteigenden Sortierung, also im Normalfall zwischen zwei bereits in der Liste befindlichen Elementen.

```
transaction InMenu_Exercise_insertStep =
  choose
    findPlaceForInsertion
  else
    setNextFromNewElem
  else
    setPreviousToNewElem
  else
    setNextToNewElem
  else
    setPreviousFromNewElem
  else
    removeNewEdge
  else
    removeCurrentEdge
  end
end;
```

Abbildung 62: Spezifikation von *InMenu_Exercise_insertStep* für die doppelt verkettete Liste

- Setze eine *next*-Kante vom neuen Element zum neuen Nachfolger.
- Setze eine *previous*-Kante vom neuen Nachfolger zum neuen Element.
- Erzeuge eine *next*-Kante vom neuen Vorgänger zum neuen Element.
- Erzeuge eine *previous*-Kante vom neuen Element zum neuen Vorgänger.
- Lösche die in Schritt 1 erzeugten temporären Hilfsstrukturen.

Die Aufgabe der Produktion *prepareInsert* ist es, Schritt 1 des gerade dargestellten Algorithmus zu implementieren, also die Umstände zu schaffen, die der Einfüge-Algorithmus braucht, um beginnen zu können. Dazu wird (siehe Abb. 61) eine Situation vorausgesetzt,

- in der *first*- und *last*- Kante bereits auf Listenelemente gesetzt sind,
- in der es keine *current*-Kante gibt und
- in der es kein mit *new* ausgezeichnetes Element gibt.

Die letzten beiden Punkte sollen verhindern, dass *prepareInsert* ein zweites Mal aufgerufen wird, bevor der letzte Einfüge-Prozess abgeschlossen ist.

Die ausgeführten Aktionen sind dann das Erstellen eines neuen Elementes 4´ mit dem einzufügenden Wert *valueToBeInserted* (siehe die `transfer`-Klausel), das Markieren dieses Elements mit der *new*-Kante und das Setzen der *current*-Kante auf das erste Element der Liste.

Schritt 2 des Algorithmus wird mithilfe einer Transaktion *insertStep* durchgeführt (Abb. 62), die das im letzten Kapitel grob beschriebene Aufbrechen einer Schleife in Schritte verwendet, indem für die verschiedenen auftretenden Situationen Hilfsfunktionen verwendet werden. Von diesen Hilfsfunktionen sind *findPlaceForInsertion* und *setPreviousFromNewElem* besonders interessant, so dass wir im Folgenden diese beiden näher vorstellen wollen und für die restlichen

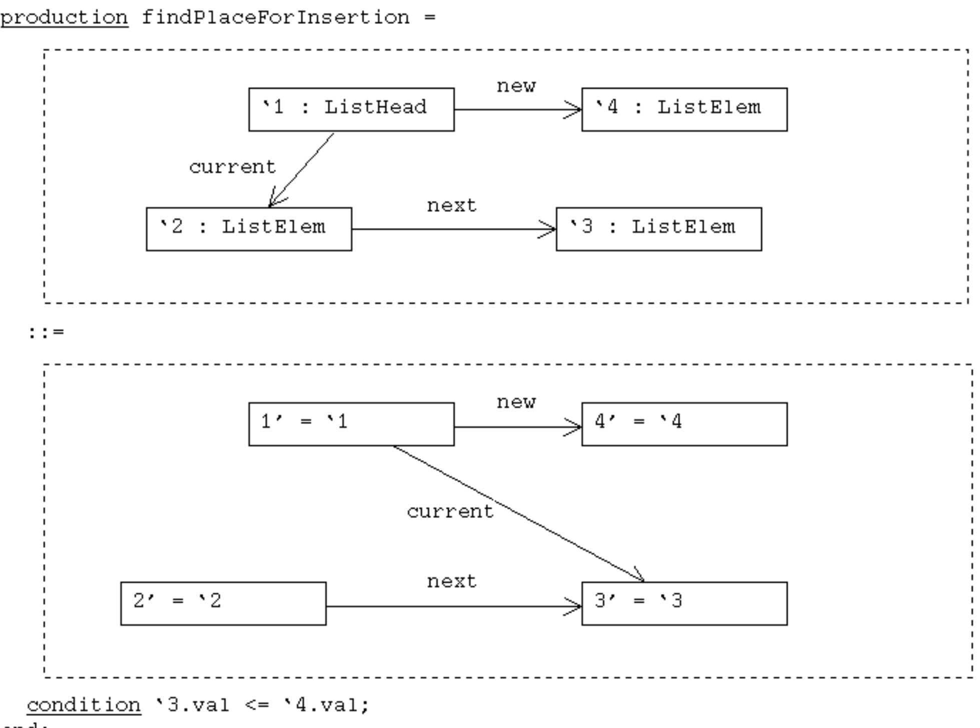


Abbildung 63: Spezifikation von *findPlaceForInsertion* für die doppelt verkettete Liste

genutzten Programmeinheiten nur informell die grobe Funktionsweise beschreiben. Die Schleife des Algorithmus steckt im Fall des Einfügens in die doppelt verkettete Liste im Durchlaufen der Elemente bis zur richtigen Position. Wieder ist bei der Transformation in einen Schritt das beherrschende Konstrukt `choose ... else ... end`. Dabei führt `choose findPlaceForInsertion else ... end` einfach die Produktion *findPlaceForInsertion* aus, falls diese erfolgreich endet. Die Programmeinheit *findPlaceForInsertion* wiederum ist so implementiert, dass sie nicht mehr erfolgreich endet, falls der korrekte Platz für das Einfügen des neuen Elements schon gefunden wurde. In diesem Fall wird also im *else*-Zweig weitergemacht, was bedeutet, dass als Nächstes *setNextFromNewElem* ausgeführt wird. Wie der Name schon sagt, wird hier die *next*-Kante *vom neuen Element* zu seinem Nachfolger gesetzt. Falls das bereits im letzten Aufruf von *InMenu_Exercise_insertStep* erledigt wurde, schlägt wiederum *setNextFromNewElem* fehl und so weiter. Auf diese Weise wird, für den Nutzer von *setNextFromNewElem* transparent, immer der zum jeweiligen Zeitpunkt richtige Schritt des Algorithmus ausgeführt. Mit anderen Worten kann ein Benutzer, der interaktiv an dieser Lerneinheit arbeitet, durch beliebig häufiges Klicken auf die Aktion *InMenu_Exercise_insertStep* nacheinander die vorgesehenen Schritte des Algorithmus ausführen.

Die in Abb. 63 dargestellte Produktion *findPlaceForInsertion* setzt unter folgenden Bedingungen die *current*-Kante um eine Stelle weiter und nähert sich somit der korrekten Einfügestelle

```
production setPreviousFromNewElem =
```

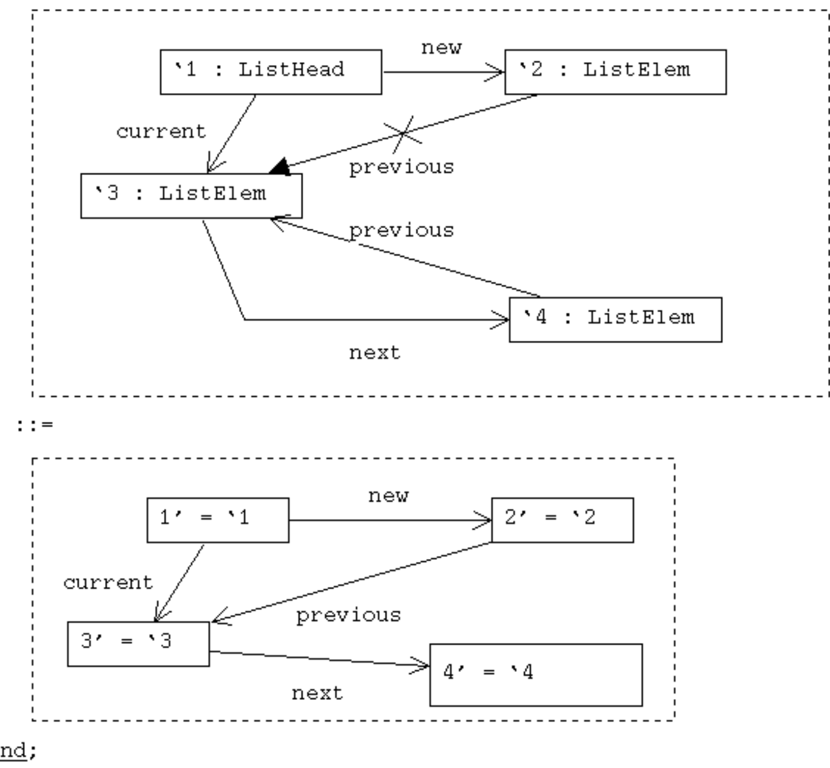


Abbildung 64: Spezifikation von *setPreviousFromNewElem* für die doppelt verkettete Liste

für das neue Element `4 immer mehr:

- Es gibt bereits ein *new*-Element `4.
- Es gibt ein *current*-Element `2, das mindestens eine ausgehende *next*-Kante auf ein Element `3 hat.
- Der Wert von `4, also der einzufügende Wert in die Liste, ist größer oder gleich dem Wert von `3 (siehe die `condition`-Klausel).

Man beachte, dass *findPlaceForInsertion* nur für Fälle korrekt spezifiziert ist, in denen die Liste vor dem Einfügen aufsteigend sortiert ist und keiner der beiden Randfälle vorliegt, in denen das neue Element eigentlich vor dem ersten oder nach dem letzten bisherigen Element eingefügt werden müsste. Die Spezifikation wurde in der gezeigten Form entwickelt, weil in der entsprechenden Übungsaufgabe die Studierenden herausfinden sollen, in welchen Fällen der Algorithmus fehlschlägt. Sonst müssten entweder die von *InMenu_Exercise_insertStep* genutzten Produktionen (wie eben *findPlaceForInsertion*) geeignet erweitert werden oder weitere Programmeinheiten für die Randfälle hinzugenommen und in *InMenu_Exercise_insertStep* aufgerufen werden.

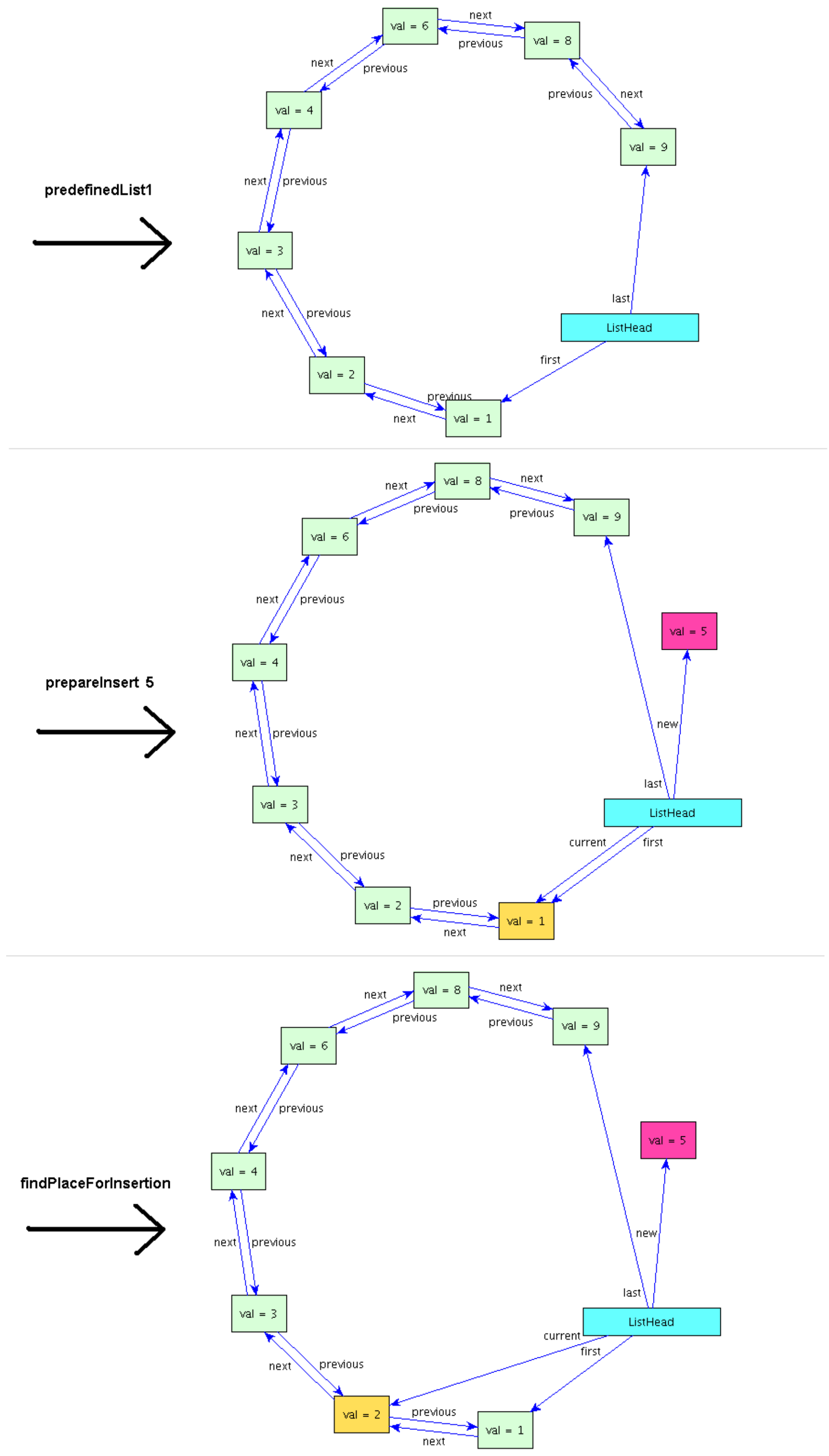


Abbildung 65: Beispielablauf beim Einfügen in eine Liste

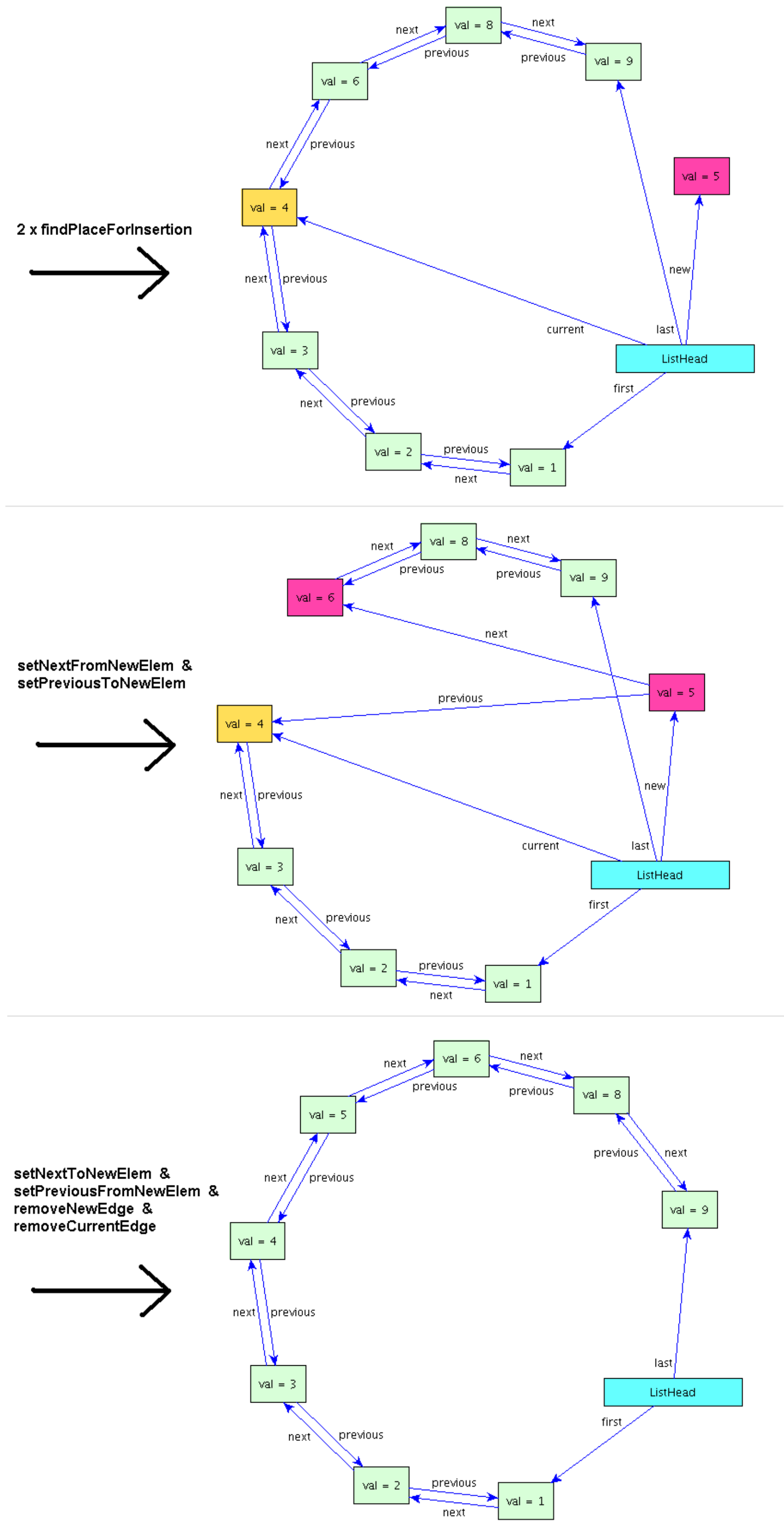


Abbildung 66: Beispielablauf beim Einfügen in eine Liste

Die weiteren Schritte des vorgestellten Algorithmus fügen, sobald die Einfügestelle gefunden wurde, also sobald *findPlaceForInsertion* fehlschlägt, das neue Element durch eine komplette Erstellung der erforderlichen Zeiger zu und von den beiden neuen Nachbarn in die Liste ein. Abb. 64 zeigt dazu beispielhaft die Modellierung der Produktion *setPreviousFromNewElem*, die einen *previous*-Zeiger von dem neuen Element `2 auf den neuen Vorgänger `3 erstellt; dabei ist durch die *current*-Kante der Knoten `3 eindeutig bestimmt. Wie man in Abb. 64 ebenfalls sieht, wird noch geprüft, dass nicht bereits ein *previous*-Zeiger an der richtigen Stelle existiert, und die bisher bestehende *previous*-Kante vom bisherigen Vorgänger `4 zum Knoten `3 gelöscht.

Abb. 65 und 66 zeigen einige Schritte eines Beispielablaufs mit dem gerade beschriebenen Algorithmus. Dabei wurde zuerst eine nicht geschlossene (also noch inkorrekte) Ringliste erzeugt, dann *prepareInsert 5* aufgerufen, und dann immer weiter mit *InMenu_Exercise_insertStep* vorgegangen, bis das neue Element 5 schließlich an seinem Platz in der Liste war und die Aufräumarbeiten vollendet waren. Zu beachten ist, dass mit Ausnahme von *predefinedList1* und *prepareInsert* die in Abb. 65 und 66 angegebenen PROGRES-Programmeinheiten nicht vom Nutzer der Lerneinheit direkt aufgerufen werden (wie auch jeweils aus dem Namen ersichtlich ist), sondern durch den wiederholten Aufruf von *InMenu_Exercise_insertStep*.

Die hier vorgestellten PROGRES-Produktionen und -Transaktionen sind ein repräsentativer Querschnitt aus der zugehörigen Spezifikation. Viele Programmteile daraus, auch die nicht explizit vorgestellten, sind strukturell ähnlich.

Im nächsten Abschnitt beschreiben wir wesentliche Teile der Spezifikation der Dijkstra-VIDEA-Instanz.

6.6. Spezifikationen für den Dijkstra-Algorithmus

Der Dijkstra-Algorithmus bietet uns relativ zu den bisher betrachteten Spezifikationen zwei Neuerungen, die es uns ermöglichen, zusätzliche Konstrukte zu zeigen, nämlich

- einen allgemeinen Graphen als Datenstruktur
- und attributierte Kanten.

Der erste Punkt schränkt uns in Bezug auf die existierenden Zeiger zwischen den Knoten in keiner Weise mehr ein, zwischen je zwei Knoten sind beliebig viele Kanten in jeder Richtung erlaubt (allerdings werden wir uns in Lehrbeispielen auf übersichtliche Beispiele beschränken, aber die Lernenden dürfen bei ihrem explorativen Erkunden beliebig komplexe Beispiele konstruieren). Der zweite Punkt erfordert statt einer bestimmten Art fester, vordefinierter Kantentypen ohne eigenen internen Zustand, die wir als knotenwertiges Attribut in die Typ-Deklaration schreiben konnten, wie z. B. bei der Deklaration der *left*- und *right*-Kante, jetzt nur *einen* Kantentyp, der aber im Gegensatz zum AVL-Baum ein Attribut besitzt. Nun kennt PROGRES an sich keine attributierten Kanten, so dass wir diese mit einem speziellen Knotentyp simulieren, von dem aus eine *source*- und *target*- Beziehung zu den Endpunkten der simulierten Kante besteht.

Hintergrund ist die bereits in Kapitel 5 erwähnte Nutzung des so genannten Edge-Node-Edge-Filters. VIDEA macht von dieser Möglichkeit auf eine Art und Weise Gebrauch, die völlig transparent für den Benutzer ist. Dieser sieht somit am Bildschirm eine attributierte Kante und

```
procedure Dijkstra(G: in out Graph; s:  in out Vertex) is
     Q: PriorityNodeQueue;              -- Knoten-Queue sortiert nach aktueller Entfernung zu s
                                        -- Unterschiede zu BFS sind in rot markiert
begin
     for each u ∈ G.V loop                   -- alle Knoten initialisieren
          u.Colour := white; u.Distance :=    ;  u.pred := null;
     end loop;
     s.Colour := grey; s.Distance := 0;  -- Startknoten initialisieren
     Q := CreateQueue(s);                -- Startknoten wird in leere Warteschlange aufgenommen
     while not IsEmpty(Q) loop
          DequeueMinimal(Q, u);          -- Knoten mit minimaler Entfernung zu s wird entnommen
          for each v ∈ u.TargetNodes loop -- alle Ziele auslaufender Kanten besuchen
               if v.Colour = white then
                    v.Colour := grey; Enqueue(Q, v);
               end if;
               if v.Colour = grey and (u.Distance+G.EdgeWeight(u,v) < v.Distance) then
                    v.Distance := u.Distance+G.EdgeWeight(u,v) ; v.pred := u;
               end if;
          end loop;
          u.Colour := black;
     end loop;
end Dijkstra;
```

Abbildung 67: Der Dijkstra-Algorithmus in prozeduraler Form mit den gängigen Farben weiß, grau und schwarz, entnommen aus [Ein05a]

kann mit dieser auf übliche Art arbeiten – sie z. B. als Parameter für PROGRES-Programmeinheiten nutzen, wie wir noch sehen werden.

In Abb. 67 ist der in Kapitel 5 beschriebene Dijkstra-Algorithmus zum besseren Verständnis des Folgenden in prozeduraler Form dargestellt.

Abb. 68 zeigt die Grundlage einer Implementierung für den Dijkstra-Algorithmus. Hier werden passend zu dem gerade Gesagten zwei Knoten-Klassen *NODE* für echte Knoten und *EDGE* für simulierte Kanten und zwei davon abgeleitete Knotentypen *Node* und *Edge* definiert. Es gibt diesmal keine berechneten Attribute, was damit zusammenhängt, dass die Zeigerstrukturen a priori keine bestimmte Struktur aufweisen müssen und sich somit auch keine strukturell benötigten Attributwerte zur Laufzeit kumulieren.

Ein Knoten vom Typ *Node* hat folgende Attribute:

- *id* ist eine eindeutige Knotenbezeichnung,
- *distance* drückt aus, wie groß die Entfernung vom Wurzelknoten des Graphen höchstens ist,
- *color* speichert die Farbe des Knotens laut Algorithmus; dabei gibt es weiß, grau und schwarz (alle drei als `integer` codiert),
- *isCurrentU* legt fest, ob zum aktuellen Zeitpunkt dieser Knoten der im Algorithmus ausgezeichnete Knoten *U* ist (siehe Abb. 67),

```
node class EDGE end;

node class NODE end;

node type Edge : EDGE
  intrinsic
    source : Node;
    target : Node;
    weight : integer := 0;
    couldBeOnTheWay : boolean := false;
    isOnTheWay : boolean := false;
end;

node type Node : NODE
  intrinsic
    id : integer;
    distance : integer := maxInt;
    color : integer := white;
    isCurrentU : boolean := false;
    isOneOfCurrentVs : boolean := false;
    isInQueue : boolean := false;
end;
```

Abbildung 68: Typ-Deklarationen für den Dijkstra-Algorithmus auf Graphen

- *isOneOfCurrentVs* zeigt, ob der Knoten zur aktuellen Menge der *V*s gehört und
- *isInQueue* enthält die Information, ob der Knoten in der vom Algorithmus verwendeten Warteschlange ist (die Ordnung der Warteschlange wird dabei durch weitere Konstrukte sichergestellt, die im Folgenden beschrieben werden).

Eine Kante vom Typ *Edge* hat folgende Attribute:

- *source* als Zeiger auf den Ursprung der Kante,
- *target* als Zeiger auf das Ziel der Kante,
- *weight* als das eigentliche Attribut der Kante, nämlich das im Algorithmus gebrauchte Gewicht,
- *couldBeOnTheWay* als Angabe, ob die Kante möglicherweise auf dem gesuchten kürzesten Weg zu ihrem Zielknoten liegt und
- *isOnTheWay* als Angabe, ob die Kante bereits als auf dem kürzesten Weg zu ihrem Zielknoten liegend identifiziert wurde.

Wie schon bei der Beschreibung des Algorithmus erwähnt, setzen wir voraus, dass im Graphen genau ein Knoten ohne einlaufende Kanten existiert; dieser Knoten dient dann als Quelle aller berechneten kürzesten Pfade und Strecken. Eine Alternative wäre es gewesen, den Startknoten explizit zu setzen, z. B. über ein entsprechendes Attribut. Wir wählen die implizite Definition des Startknotens einerseits aus Gründen der Einfachheit, andererseits auch mit dem Hintergedanken, dass sich beim Einsatz dieser VIDEA-Instanz im Szenario des interaktiven Erforschens und Aufgaben-Lösens die Schritte der Lernenden im Laufe der Übung dann besser vergleichen lassen.

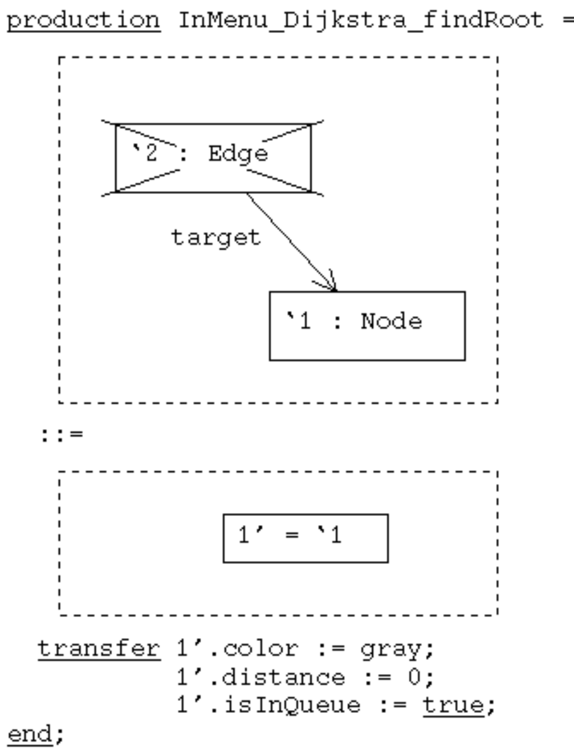


Abbildung 69: Erster Schritt des Dijkstra-Algorithmus (Finden und Initialisieren des Wurzel-Knotens)

Die Produktion *InMenu_Dijkstra_findRoot* in Abb. 69 findet einen solchen Knoten, indem sie nichtdeterministisch einen Knoten ohne einlaufende *target*-Kante sucht. Dieser bekommt dann entsprechend den Vorgaben des Dijkstra-Algorithmus (siehe Abb. 67) die Farbe grau und die Distanz 0 zugewiesen und wird als vorerst einziger Knoten in die Warteschlange gestellt.

Abb. 70 oben nutzt in dem PROGRES-Test *InMenu_Exercise_testNextU* eine Restriktion: Denn entsprechend dem Dijkstra-Algorithmus wird derjenige Knoten *U* vom Algorithmus als Nächstes betrachtet, und ist somit Ausgangspunkt der durch den Graphen laufenden „Welle", der in der Prioritäts-Warteschlange vorne steht, also derjenige mit der kleinsten bisher geschätzten Entfernung in der Warteschlange. Genau das steht in Abb. 70 oben: Es wird geprüft, ob der vom Lernenden übergebene Knoten *n* in der Warteschlange ist (das steht in der `condition`-Klausel) und ob es keinen Knoten gibt, dessen geschätzte Distanz kleiner als die des übergebenen Knotens *n* ist, und der gleichzeitig in der Warteschlange ist (das steht in der Restriktionsbedingung).

```
test InMenu_Exercise_testNextU( n : Node) =

  '1 : Node

  valid ((self.distance < '2.distance) and (self.isInQueue))

  '2 = n

  condition '2.isInQueue;

end;
```

```
transaction InMenu_Dijkstra_nextStep =
    makeUToNormalNode
  & dequeueMinimal
  & setUBlack
  & choose
      makeUsPredToUsFixedPred
    else
      skip
    end
  & findCurrentVs
  & recolorVs
  & redistanceVs
  & makeVsToNormalNodes
end;
```

Abbildung 70: Oben: Testen des Verständnisses bzgl. des nächsten Schrittes des Dijkstra-Algorithmus. Unten: Haupt-„Schleife“ des Dijkstra-Algorithmus

In Abb. 70 unten sehen wir eine Codierung des Haupt-Schrittes des Dijkstra-Algorithmus. Erwähnenswert ist hier wieder das Aufbrechen der im ursprünglichen Algorithmus (Abb. 67) vorhandenen Haupt-Schleife zu einem Haupt-Schritt, ähnlich wie wir es bereits beim Einfügen in die doppelt verkettete Liste gesehen haben.

Wir wollen hier nicht im Detail auf die Aufgabe jeder der einzelnen verwendeten Hilfsproduktionen eingehen, sondern nur anmerken, dass Abb. 70 unten eine fast wörtliche Umsetzung der Schritte innerhalb der Schleife des Dijkstra-Algorithmus ist mit noch zusätzlichen Schritten zur evtl. Bereinigung vorheriger Aufrufe von *InMenu_Dijkstra_nextStep* (wie z. B. *makeUTo-*

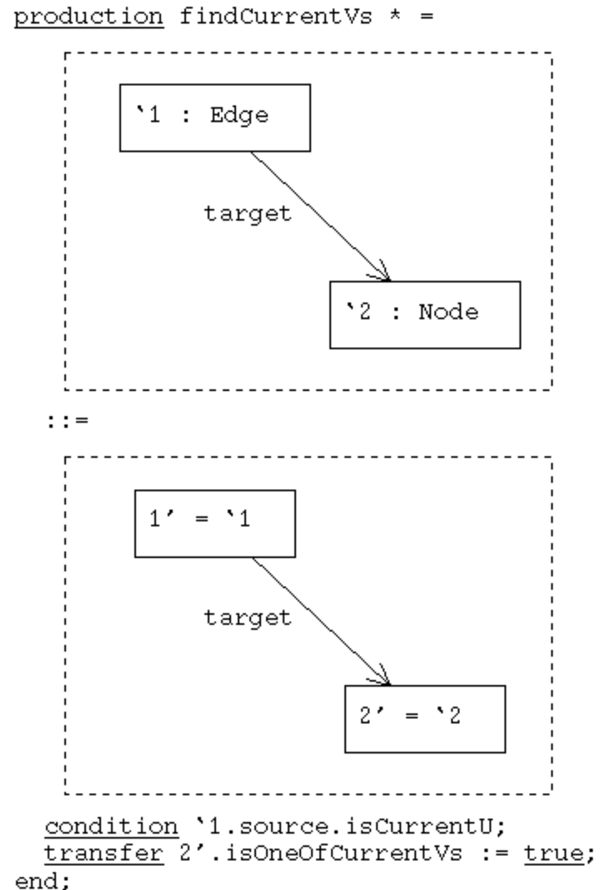


Abbildung 71: Finden der aktuellen V-Knoten gemäß Dijkstra

NormalNode). Die Namen der genutzten Programmeinheiten sollten bis auf *makeUsPredToUsFixedPred* selbsterklärend sein. Die zuletzt genannte Produktion fixiert eine weitere Kante als auf dem kürzesten Pfad zum aktuell betrachteten Knoten liegend.

In Abb. 71 sehen wir, wie die aktuellen *V*-Knoten, also diejenigen, die vom Knoten *U* aus durch Folgen *einer* Kante erreicht werden können, identifiziert wurden. Dazu betrachtet *findCurrentVs* wegen des Multiplizitäts-Operators * alle Kanten `1, die von dem momentanen *U* ausgehen (das steckt in der `condition`-Klausel) und setzt das Attribut *isOneOfCurrentVs* aller durch eine *target*-Verknüpfung erreichbaren Knoten auf `true` (in der `transfer`-Klausel).

Abb. 72 zeigt, wie durch abermalige Nutzung des *-Operators die fast wörtliche Übernahme der `if`-Bedingung in der inneren `For all`-Schleife aus dem Pseudocode in Abb. 67 erfolgen kann.

```
production recolorVs * =

   +-----------------------+
   |                       |
   |   `1 : Node           |
   |                       |
   +-----------------------+

  ::=

   +-----------------------+
   |                       |
   |    1' = `1            |
   |                       |
   +-----------------------+

  condition (`1.isOneOfCurrentVs) and (`1.color = white);
  transfer 1'.color := gray;
           1'.isInQueue := true;
end;
```

Abbildung 72: Umfärben aller V-Knoten gemäß Dijkstra

Der Hauptteil der zweiten `if`-Bedingung schließlich wird in Form der Produktion *redistanceVs* in Abb. 73 implementiert, wobei man sieht, dass wesentliche visuelle Teile der Anwendungsbedingung von *redistanceVs* im visuellen Teil spezifiziert sind, während andere, die die Abbildung unübersichtlicher als nötig machen würden, in der `condition`-Klausel verborgen sind. Neu ist hier die Verwendung einer optionalen Knotenmenge `` `4 `` (die Menge ist wieder durch die Stapel-Darstellung angedeutet, der optionale Charakter wieder durch die Strichelung), die hier für alle einlaufenden Kanten in den Knoten `` `1 `` (eines der *V*s) steht und den Zugriff darauf gewährt. Die notwendigen Änderungen in den Attributen werden dann im *transfer*-Teil durchgeführt.

Nach dieser Beschreibung großer Teile der Spezifikation des Dijkstra-Algorithmus wenden wir uns im nächsten Abschnitt unserer vierten und letzten Beispiel-Spezifikation zu.

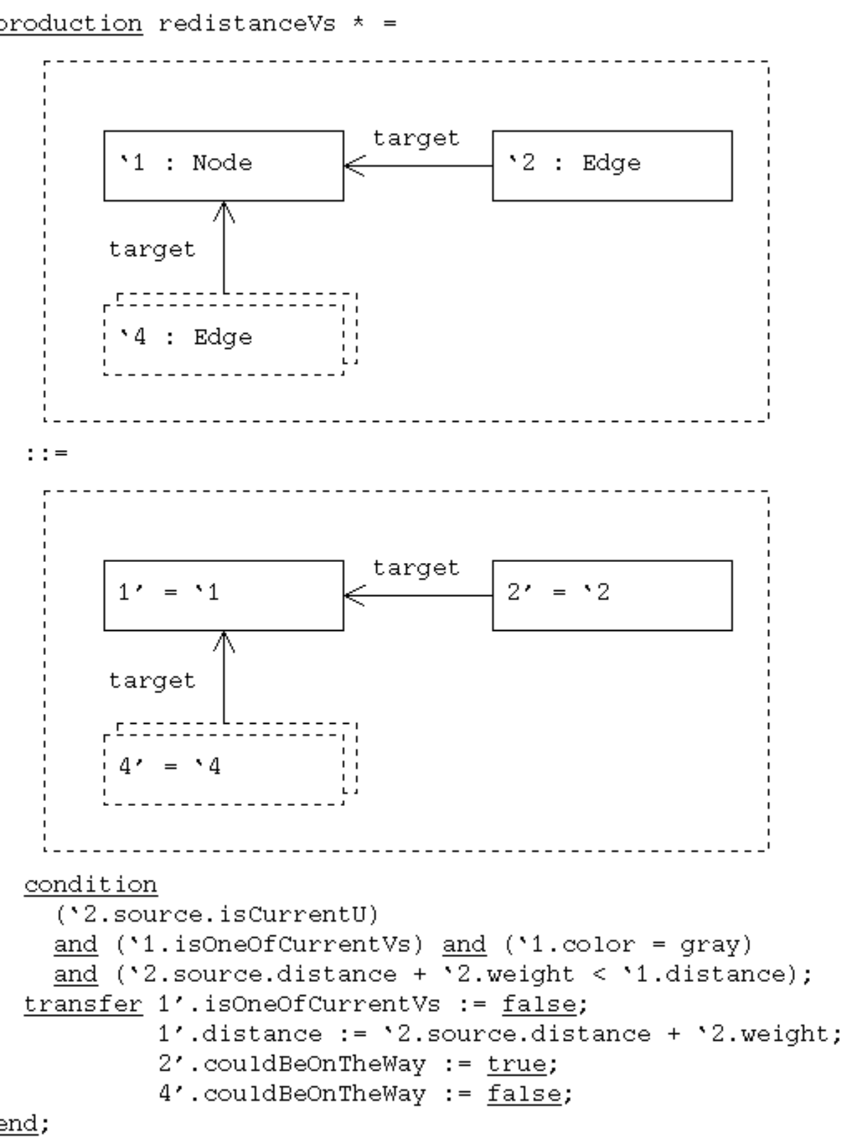


Abbildung 73: Neu-Berechnen der geschätzten Entfernung aller V-Knoten bei Dijkstra

```
node class NODE
  intrinsic
    id : integer;
    subtreeID : integer;
end;

node class EDGE
  intrinsic
    stillInQueue : boolean := true;
    belongsToSpanningTree := false;
    weight : integer;
    source : Node;
    target : Node;
end;

node type Node : NODE end;

node type Edge : EDGE end;
```

Abbildung 74: Typ-Deklarationen für den Kruskal-Algorithmus

6.7. Spezifikationen für den Kruskal-Algorithmus

In Abb. 74 sind die verwendeten Knoten- und Kantentypen für die Umsetzung des Kruskal-Algorithmus zu sehen. Der Knotentyp *Node* erbt von der Knotenklasse *NODE* einen eindeutigen Kennzeichner *id* zur Darstellung auf dem Bildschirm und ein Hilfsattribut *subtreeID*, das wir später für die Aufteilung in Partitionen nutzen werden.

Der attributierte Kantentyp *Edge*, der wie bei Dijkstra durch einen Knotentyp mit zusätzlichem *source*- und *target*-Attribut simuliert wird, enthält das Gewichts-Attribut *weight* und zwei Hilfsattribute *stillInQueue* und *belongsToSpanningTree*.

Da die Lernenden beim Einsatz dieser VIDEA-Instanz zu Beginn mit vordefinierten Graphen arbeiten (siehe Kapitel 7), ist ein Beispiel für eine vordefinierte Transaktion *predefinedGraph1* zum Aufbau eines Graphen in Abb. 75 oben angegeben. Es werden zuerst acht Knoten erzeugt, die dann durch Kanten verbunden werden. Man sieht am Namen *createEdge*, dass diese Hilfsproduktion (im Gegensatz zu *InMenu_Primitive_ createNode*) dem Nutzer nicht zum direkten Aufruf zur Verfügung steht. Der aufgebaute Graph ist in Abb. 75 unten zu sehen.

Wie in Kapitel 7 bei der Beschreibung der Nutzerschnittstelle der zugehörigen VIDEA-Instanz zu sehen sein wird, sind zu Beginn alle Kanten gestrichelt dargestellt, so lange sie noch nicht vom Algorithmus betrachtet wurden; jede abgearbeitete Kante wird durchgezogen dargestellt und je nach Ergebnis der Abarbeitung in einer von zwei Farben: Schwarz, wenn sie als zum gesuchten Spannbaum gehörig identifiziert wurden, und grau, wenn sie als nicht zum Spannbaum gehörig identifiziert wurden. Die PROGRES-Spezifikation hält diese Informationen in den erwähnten Attributen *stillInQueue* und *belongsToSpanningTree*.

```
transaction InMenu_Predefined_predefinedGraph1 =
  use n1, n2, n3, n4, n5, n6, n7, n8 : Node
  do
        InMenu_Primitive_createNode ( 1 )
      & InMenu_Primitive_createNode ( 2 )
      & InMenu_Primitive_createNode ( 3 )
      & InMenu_Primitive_createNode ( 4 )
      & InMenu_Primitive_createNode ( 5 )
      & InMenu_Primitive_createNode ( 6 )
      & InMenu_Primitive_createNode ( 7 )
      & InMenu_Primitive_createNode ( 8 )
      & createEdge ( 1, 2, 32 )
      & createEdge ( 1, 5, 48 )
      & createEdge ( 2, 6, 45 )
      & createEdge ( 2, 7, 49 )
      & createEdge ( 3, 4, 46 )
      & createEdge ( 3, 7, 18 )
      & createEdge ( 4, 8, 53 )
      & createEdge ( 5, 6, 40 )
      & createEdge ( 6, 7, 95 )
  end
end;
```

Abbildung 75: Oben: Eine Produktion zum Erzeugen eines Beispiel-Graphen. Unten: Der von der Produktion kruskal_predefinedGraph1 erzeugte Beispiel-Graph.

In Abb. 76 oben sehen wir ähnlich wie bei *InMenu_Exercise_checkBalance* bei den AVL-Bäumen und *InMenu_Exercise_testNextU* beim Dijkstra-Algorithmus wieder einen Selbsttest, den der Lernende nutzen kann, um unmittelbar das Verständnis klar definierter Eigenschaften einer Datenstruktur oder eines Algorithmus zu überprüfen. Während jedoch *InMenu_Exercise_checkBalance* ein internes Attribut abprüft, wird bei *InMenu_Exercise_testNextConsideredEdge* ähnlich wie bei *InMenu_Exercise_testNextU* die Berechnungsvorschrift unmittelbar im PROGRES-Test angegeben. Denn die nächste von Kruskal betrachtete Kante ist ja gerade diejenige in der Warteschlange, für die es keine ebenfalls in der Warteschlange befindliche Kante mit kleinerem Gewicht gibt – so ist es in der Restriktion in Abb. 76 oben formuliert. Man beachte, wie das Verhalten der Prioritäts-Warteschlange des ursprünglichen Kruskal-Algorithmus durch eine Kombination von Knotenattributen nachgebildet wird.

Abb. 76 unten zeigt den zu Abb. 76 oben passenden ergänzenden Test, der den Lernenden die Chance gibt, die Frage zu beantworten, ob die nächste betrachtete Kante zum Spannbaum hinzugefügt wird oder nicht. Das Lesen der Teile der Produktion entspricht wieder weitgehend der natürlich-sprachlichen Beschreibung dieses Schrittes des Algorithmus: „Wenn es noch eine Kante ‘2 gibt, die in der Warteschlange ist und es keine andere Kante mit kleinerem Gewicht in der Warteschlange gibt, wird ‘2 zum Spannbaum hinzugenommen, wenn ihr Quell- und Zielknoten (‘4 und ‘3) in unterschiedlichen Teilspannbäumen liegen.“

Der eigentliche Kern-Schritt des Algorithmus ist in Abb. 77 oben zu sehen. Wie man sieht, wird der bisherige Spannbaum ggf. mit der nächsten Kante verschmolzen oder diese nur aus der Menge der noch zu betrachtenden Kanten entfernt.

Die Implementierung der Produktion zum Verschmelzen zweier Spannbäume ist in Abb. 78 dargestellt: Falls keine Kante mehr existiert, die noch in der Warteschlange ist und deren Gewicht kleiner als das von Kante ‘2 wäre (ausgedrückt durch den oben durchgestrichenen *Edge*-Knoten ‘1), und ‘2 selbst noch in der Warteschlange ist (ausgedrückt durch den Pfadausdruck), werden der Quellknoten ‘4 und der Zielknoten ‘3 unserer Kante ‘2 auf die gleiche subtreeID gebracht und die Kante ‘2 in den Spannbaum aufgenommen. Die optionale Knotenmenge ‘5 entspricht der Knotenpartition, in der sich auch ‘3 befindet, und dient dazu, diese ganze Partition mit derjenigen von ‘4 zu verschmelzen.

Dementsprechend zeigt Abb. 77 unten das der zweiten Alternative des Algorithmus entsprechende Entfernen der dortigen Kante ‘2, das ganz ähnlich zu lesen ist.

Wir haben in diesem Kapitel gezeigt, wie durch visuelle Spezifikation in PROGRES verschiedene Algorithmen und Datenstrukturen insbesondere auch für den Dozenten anschaulicher als eine rein textuelle Spezifikation aufbereitet werden können, was die Wiederverwendbarkeit einer existierenden Datenstruktur-Spezifikation für Obermengen oder Varianten der jeweiligen Datenstruktur erleichtert. Dabei wurden die verwendeten Sprachmittel zwar relativ einfach gehalten, aber ein ausreichendes Spektrum an Möglichkeiten vorgeführt, um die im Rahmen von VIDEA betrachteten Datenstrukturen modellieren zu können. Da die Sprache PROGRES Schleifen und Rekursion enthält, ist sie gleich mächtig zu jeder anderen gängigen Programmiersprache. Die in VIDEA genutzte Teilsprache von PROGRES verzichtet lediglich auf die Verwendung von Schleifenkonstrukten und Rekursion in denjenigen Teilen einer Spezifikation, die vom Nutzer einer VIDEA-Instanz interaktiv und schrittweise zu nutzende Abläufe beschreiben. Wie diese

```
test InMenu_Exercise_testNextConsideredEdge( e : EDGE) =

    `1 : Edge

    valid ((self.weight < e.weight) and (self.stillInQueue))

  condition e.stillInQueue;

end;
```

```
test InMenu_Exercise_isNextConsideredEdgeTaken( b : boolean)
=

    `1 : Edge

    valid ((self.weight < `2.weight) and (self.stillInQueue))

    valid (self.stillInQueue)

    `2 : Edge      target      `3 : Node

    source

    `4 : Node

  condition b = not (`3.subtreeID = `4.subtreeID);

end;
```

Abbildung 76: Zwei Verständnistests: Oben: welche Kante wird von Kruskal als Nächstes betrachtet? Unten: Wird die nächste betrachtete Kante zum Spannbaum hinzugenommen?

```
transaction InMenu_Exercise_kruskalStep =
  choose
    mergeSpanningTreesWithNextEdge
  else
    onlyRemoveNextEdge
  end
end;
```

```
production onlyRemoveNextEdge =

  `1 : Edge
  valid ((self.weight < `2.weight) and (self.stillInQueue))

  valid (self.stillInQueue)
  `2 : Edge   --target-->  `3 : Node
  `2 : Edge   --source-->  `4 : Node

::=

  2' = `2   --target-->  3' = `3
  2' = `2   --source-->  4' = `4

  condition `3.subtreeID = `4.subtreeID;
  transfer 2'.stillInQueue := false;
end;
```

Abbildung 77: Oben: Eine Nutzeraktion für das Ausführen genau eines Schrittes des Kruskal-Algorithmus. Unten: Das Entfernen der kleinsten noch nicht betrachteten Kante aus der Warteschlange.

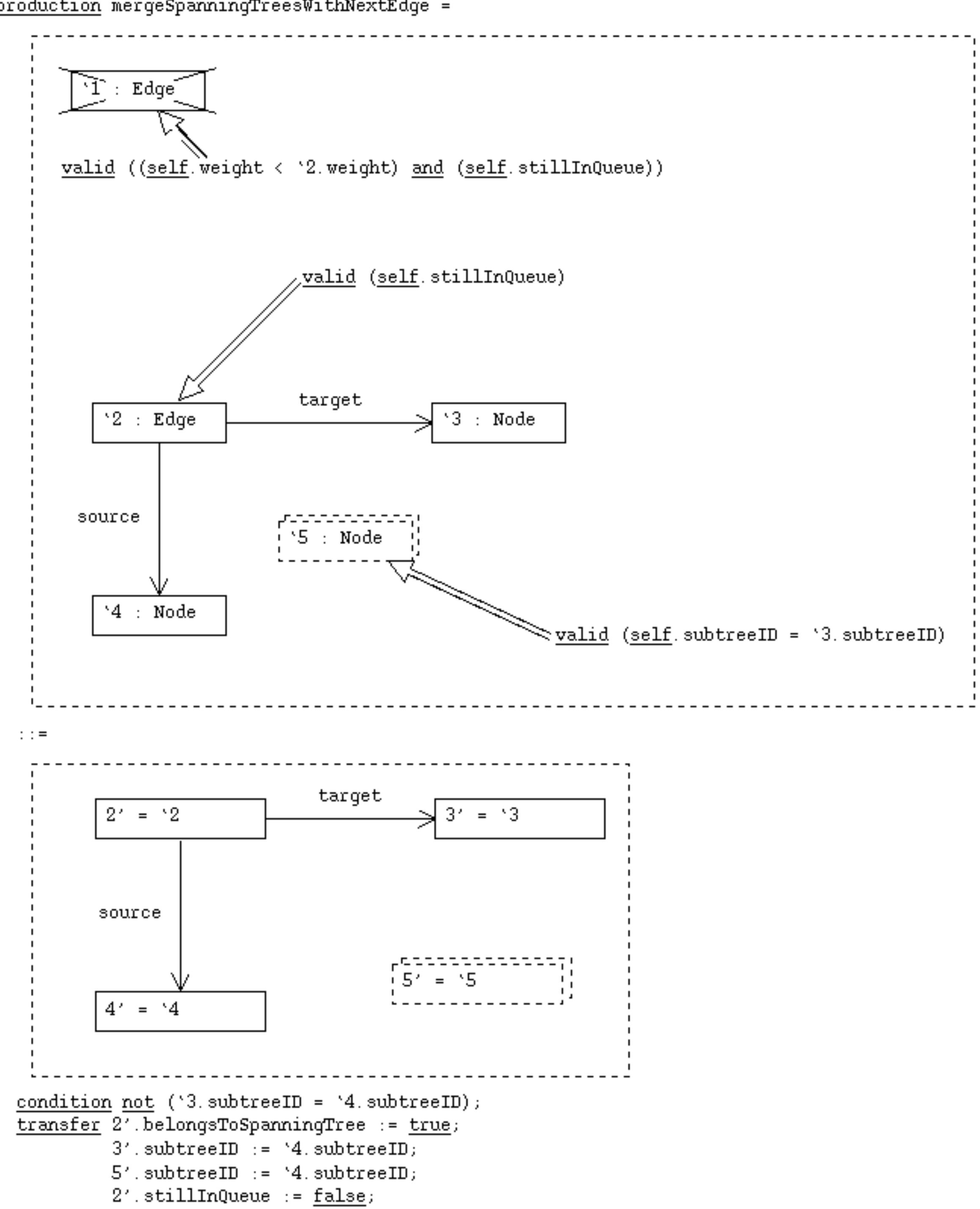


Abbildung 78: Hilfsproduktion zum Verschmelzen zweier Spannbäume mittels einer neu hinzugenommenen Kante

ohne Verlust an Mächtigkeit der Beschreibungssprache unter Nutzung neuer Hilfsattribute in eine Folge von Schritten umgewandelt werden können, wurde mehrfach in diesem Kapitel gezeigt.

Im folgenden Kapitel wird mithilfe vierer VIDEA-Instanzen, die auf den in diesem Kapitel vorgestellten Spezifikationen aufbauen, die Sicht des Benutzers beim Einsatz von VIDEA in den in Kapitel 5 vorgestellten vier wesentlichen Einsatzszenarien beschrieben.

7. Nutzersicht auf vier Typen von Instanzen

VIDEA wurde geschaffen, um die Lehre von Algorithmen und Datenstrukturen multimedial zu unterstützen. Eine zentrale Fähigkeit zur Erreichung dieses Ziels ist die Erstellung interaktiver, flüssiger Animationen. Im letzten Kapitel haben wir anhand mehrerer Beispiele gesehen, wie durch die Erstellung visueller (und textueller) Spezifikationen in PROGRES eine Datenstruktur, ein Algorithmus und Schnittstellenoperationen definiert werden können.

Der Anspruch dieser Arbeit war es von Anfang an, nicht nur ein Rahmenwerk zur automatisierten Erzeugung von Lerneinheiten aus visuellen Spezifikationen zur Verfügung zu stellen, sondern auch einige beispielhaft erstellte Instanzen anzubieten, die unmittelbar und produktiv in der Lehre von Algorithmen und Datenstrukturen eingesetzt werden können.

Diese vier Instanzen, deren PROGRES-Spezifikationen auszugsweise in Kapitel 6 vorgestellt wurden, heißen *AVL-Bäume*, *Dijkstra*, *Kruskal* und *doppelt verkettete Listen*.

Die Mehrzahl dieser VIDEA-Instanzen ist bereits in der Lehre eingesetzt worden. Für *AVL-Bäume* und *Dijkstra* wurde eine kleine Evaluation durchgeführt. Die Erfahrungen und bisherigen Ergebnisse werden in Kapitel 9 erörtert.

In diesem Kapitel werden wir einige Einsatzbeispiele für aus diesen Spezifikationen generierte VIDEA-Instanzen vorstellen. Als Anwendungs- und Lehrsituationen legen wir folgende in Kapitel 3 und 5 als besonders interessant und wichtig identifizierte Einsatzszenarien zugrunde:

- Vorführen im Rahmen einer Präsenzveranstaltung
- Passives Betrachten vorgegebener Abläufe
- Interaktives Erforschen und Aufgaben-Lösen
- Ferngesteuerte Animation eigener Programme der Lernenden

Wir werden sehen, dass der erste und zweite Punkt zwar noch wenige Anforderungen an die Interaktivität von VIDEA stellen, aber bereits die Fähigkeit zur Darstellung kontinuierlicher, flüssiger Animationen erfordern. Beim dritten Punkt kommt die geforderte Möglichkeit der Interaktivität hinzu. Der vierte Punkt schließlich stellt die größten technischen Anforderungen, nämlich die nach einem Mechanismus, der entfernte Aufrufe absetzen kann.

Wir werden in diesem Kapitel jedem der vier angesprochenen Punkte einen Abschnitt widmen und somit die für VIDEA vorgesehenen Lernszenarien beschreiben. Da die vier Einsatzszenarien bereits in Kapitel 3 konzeptionell eingeführt wurden und in Kapitel 5 in eine Skala denkbarer Lernszenarien eingeordnet wurden, legen wir hier den Schwerpunkt auf die Beschreibung der auftretenden Abläufe bei der Benutzung von VIDEA. Die Möglichkeiten der Konfiguration der VIDEA-Instanzen für die und in den beschriebenen Szenarien werden wir in Kapitel 8 behandeln.

7.1. Vorführen im Rahmen einer Präsenzveranstaltung oder Übung

Die einfachste Anwendungsmöglichkeit von VIDEA ist der Einsatz im Vorlesungsbetrieb. Wie schon in Kapitel 2 betont wurde, sind weder der Motivationsgewinn noch die Erleichterung der internen Lernvorgänge des Nutzers durch übersichtliche Visualisierungen zu unterschätzen.

Bereits vor der Entwicklung von VIDEA haben wir einige ermutigende Erfahrungen mit dem Vorführen vorbereiteter Abläufe bei der Arbeit mit einfachen Algorithmen und Datenstrukturen gesammelt. Genutzt wurden dabei einige existierende, frei verfügbare Visualisierungen, die zum Großteil fest implementiert waren (u. a. [CCA03, Ani03, Sor03, Bub03]).

In VIDEA wird das Vorführen vorbereiteter Inhalte eingeleitet durch den Aufbau einer Instanz einer Datenstruktur. Das geschieht üblicherweise durch die Nutzung vorbereiteter Produktionen oder durch Skripte. Ein Beispiel für die erste Variante ist die in Kapitel 6 beschriebene Produktion *predefinedGraph1* (Abb. 75 oben auf Seite 142). Auf diese Weise wird ein vorbereiteter Graph gezeichnet, an dem dann durch den Aufruf geeigneter Operationen die Schritte eines vordefinierten Algorithmus gezeigt werden können.

Auf die zweite Variante, die Nutzung eines Skripts, wird üblicherweise zurückgegriffen, um eine Instanz einer Datenstruktur unter Nutzung flüssiger Animationen aufzubauen. Beim Beispiel des AVL-Baums können die Lernenden zunächst den Aufbau eines Baums beobachten, in den nach und nach Knoten in einer definierten Reihenfolge eingefügt werden. Dabei kommt es natürlicherweise immer wieder zu Verletzungen der Balancierungseigenschaft, das heißt, dass rotiert werden muss. Die vordefinierte Reihenfolge der Einfüge-Operationen wird dann sinnvollerweise gerade so gewählt, dass nach und nach geeignete Ausgangssituationen für alle vier möglichen Rotationen im AVL-Baum entstehen. An diesen Stellen sind die entsprechenden Rotationen dann in den vordefinierten Ablauf eingebaut, so dass die Lernenden alle Rebalancierungs-Operationen zu sehen bekommen.

Gemäß unserem VIDEA-Konzept werden sowohl das Einfügen neuer Knoten als auch die Rotationsoperationen in weichen, animierten Bewegungen dargestellt. Abb. 79 zeigt einen Schnappschuss einer solchen Bewegung während des beschriebenen Einfüge-Ablaufs; hier wird gerade ein neuer Knoten mit Wert 120 in den Baum eingefügt und an seinen vorbestimmten Platz als neues linker Kindknoten des Knotens mit Wert 130 bewegt. Dabei wird der Knoten mit Wert 50 vorübergehend halb verdeckt. Diese weichen Bewegungen sollen, wie mehrfach beschrieben, stetige visuelle kognitive Prozesse im Lernenden ermöglichen.

Das genutzte Skript ist in Abb. 80 zu sehen. Wie man sieht, nutzt VIDEA einfach eine Textdatei, deren Zeilen die Namen der PROGRES-Programmeinheiten enthalten, gegebenenfalls mit Angabe der Parameter, die durch Leerzeichen voneinander und von den Produktionsnamen getrennt werden.

Für die optische Aufbereitung gibt es einige Möglichkeiten in VIDEA: Wichtige Messgrößen der Datenstruktur wie Balancefaktor, Höhe, Sortiertheit, Knotenwert eines AVL-Baum-Knotens können z. B. farblich markiert werden oder als Attribut im Knoten angezeigt werden. Abb. 79 zeigt zusätzlich zum beschriebenen Ablauf die farbliche Markierung einer fehlerhaften Situation; hier ist der Baum wegen des gerade stattfindenden Einfügens von Knoten 120 an der Wurzel temporär unbalanciert. Andererseits verdeutlichen einige vom Dozenten ausgewählte Attribute (hier: *val*, *balance* und *height*) den momentanen Zustand des Baumes.

Im Gegensatz zum hier beschriebenen *Vorführen*, bei dem es oft sinnvoll ist, alle wesentlichen Daten der Datenstruktur in Attributwerten anzuzeigen, empfehlen wir beim Selbststudium normalerweise eine Voreinstellung von VIDEA, in der möglichst wenig redundante Information gleichzeitig auf dem Bildschirm sichtbar ist. Alternativ können dann wesentliche Messgrößen zunächst durch Farbgebung gekennzeichnet werden. Erst bei einer größeren Menge von für den Lerneffekt wesentlichen Informationen sollte auch bei anderen Einsatzszenarien auf die Anzeige

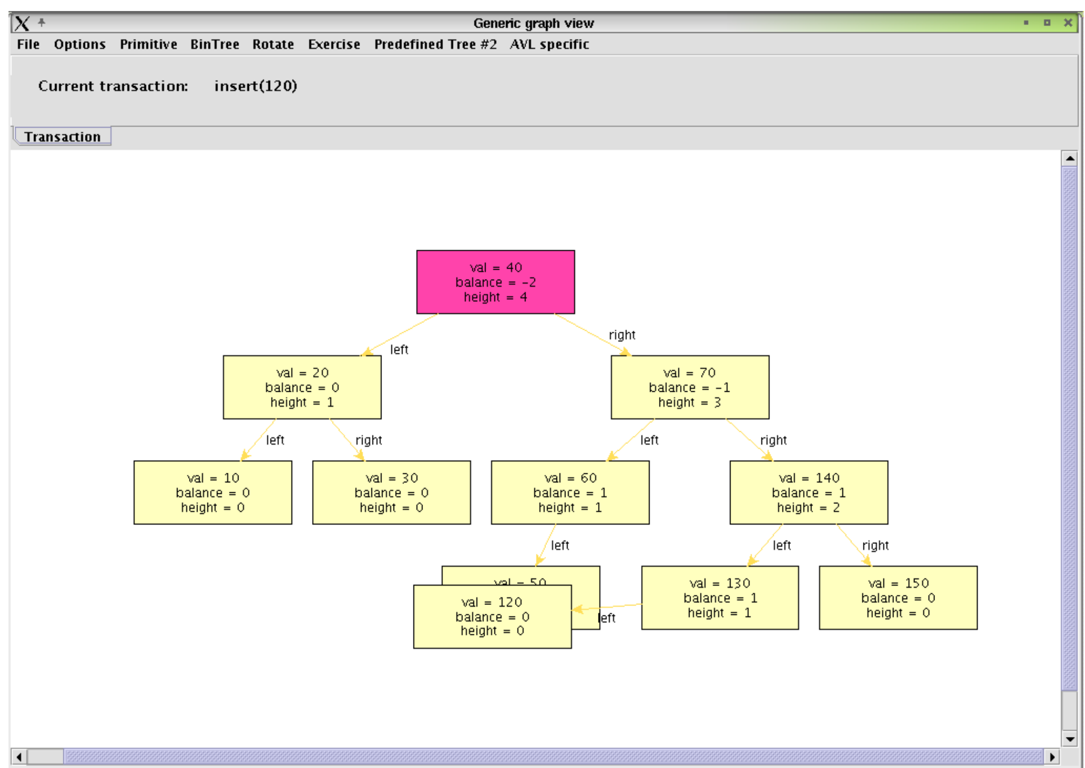


Abbildung 79: AVL-Instanz: Ein vorher definierter Ablauf, hier Einfügen des Knotens '120'

von Attributen ausgewichen werden. Sonst besteht beim Lernenden die Gefahr der Verwirrung durch zu viel gleichzeitig vorhandene Daten.

Beim *Vorführen* bestimmt der Dozent das Tempo der Erklärung und kann dieses nötigenfalls erheblich reduzieren, um explizit auf Attributwerte hinzuweisen, die das gerade zu Lernende oder Gelernte zusätzlich zu Layout- und Farb-Informationen numerisch oder textuell verdeutlichen. Außerdem ist es beim Vorführen natürlich jederzeit möglich, einen Ablauf – für alle anwesenden Lernenden synchron – zu wiederholen, wenn die Menge der angezeigten Information für einen Durchlauf zu groß war.

Nach den in Kapitel 2 beschriebenen Ergebnissen bisheriger Studien ist es für den Lerneffekt auch beim „bloßen“ Vorführen irgendwelcher Abläufe wichtig, nicht nur *einen* Sinneskanal zu versorgen, also etwa wie im Kino nur einen Stummfilm oder gar nur eine Folge von wenigen Bildern ablaufen zu lassen, sondern mehrere Sinneskanäle gleichzeitig anzusprechen. Eine Möglichkeit hierfür ist die zum Ablauf der Visualisierung synchrone verbale Kommentierung Stoff-relevanter Einzelheiten, wie z. B.:

> „und gleich werden wir sehen, wie der Baum um den Knoten 10 nach links rotiert wird ... *jetzt*. Beachten Sie, dass ...“.

Auf diese Weise werden durch die Kopplung von Sinneskanälen die in Kapitel 2 beschriebenen Verstärkungseffekte durch Doppelcodierung – in diesem Fall zwischen dem externen auditiven

```
InMenu_BinTree_insert 10
InMenu_BinTree_insert 20
InMenu_BinTree_insert 30
Rotate_rotateLeftByNodeVal 10
InMenu_BinTree_insert 40
InMenu_BinTree_insert 50
Rotate_rotateLeftByNodeVal 30
InMenu_BinTree_insert 60
Rotate_rotateLeftByNodeVal 20
InMenu_BinTree_insert 70
Rotate_rotateLeftByNodeVal 50
InMenu_BinTree_insert 150
InMenu_BinTree_insert 140
Rotate_rotateRightLeftByNodeVal 70
InMenu_BinTree_insert 130
Rotate_rotateRightLeftByNodeVal 60
InMenu_BinTree_insert 120
Rotate_rotateLeftByNodeVal 40
InMenu_BinTree_insert 110
Rotate_rotateRightByNodeVal 130
InMenu_BinTree_insert 100
Rotate_rotateRightByNodeVal 140
InMenu_BinTree_insert 90
Rotate_rotateRightByNodeVal 110
InMenu_BinTree_insert 80
InMenu_BinTree_insert 85
Rotate_rotateLeftRightByNodeVal 90
```

Abbildung 80: Skript für den flüssigen Aufbau eines AVL-Baums

und dem externen visuellen Sinneskanal – erst ermöglicht. In noch ausgeprägterer Form geschieht das beim weiter unten beschriebenen interaktiven Üben, dort mit einer Kopplung des externen visuellen Kanals mit dem externen kinästhetischen.

Ein weiteres Beispiel für den kontinuierlichen Aufbau einer Datenstruktur mit flüssiger Animation ist der Aufbau einer doppelt verketteten Liste. Hier werden nach und nach Listenelemente eingefügt und dabei einige in den Ablauf eingebaute Fehler gemacht. In Abb. 81 ist eine doppelt verkettete Liste mit Listenkopf zu sehen, die gerade aufgebaut wurde und einige Mängel aufweist. Zum Beispiel ist Knoten 33 in Bezug zur geforderten aufsteigenden Sortierung an der falschen Stelle einsortiert. Zwischen den Knoten 8 und 9 fehlt die *previous*-Kante. Außerdem gibt es keine *last*-Kante. All diese Probleme, die im Falle einer interaktiven Nutzung nach und nach vom Lernenden gelöst werden sollen, werden durch rote Färbung der jeweils betroffenen Knoten ausgedrückt, was im Schwarzweißdruck dunkler zu sehen ist. Knoten 33 und 4 sind also wegen der fehlerhaften Sortierung in Bezug zur unmittelbaren Nachbarschaft markiert, Knoten 8 und 9 wegen der fehlenden *previous*-Kante und Knoten 9 zusätzlich wegen der fehlenden *last*-Kante (vgl. die Spezifikation der doppelt verketteten Liste in Kapitel 6).

Außerdem ist hier natürlich keine Ringliste gegeben, was in diesem Beispielablauf allerdings auch nicht gefordert war. Nach dem Vorführen dieses Ablaufs kann der Dozent selbst entscheiden, ob er nur konzeptionell die Probleme erklärt, die durch die rote Färbung angedeutet werden,

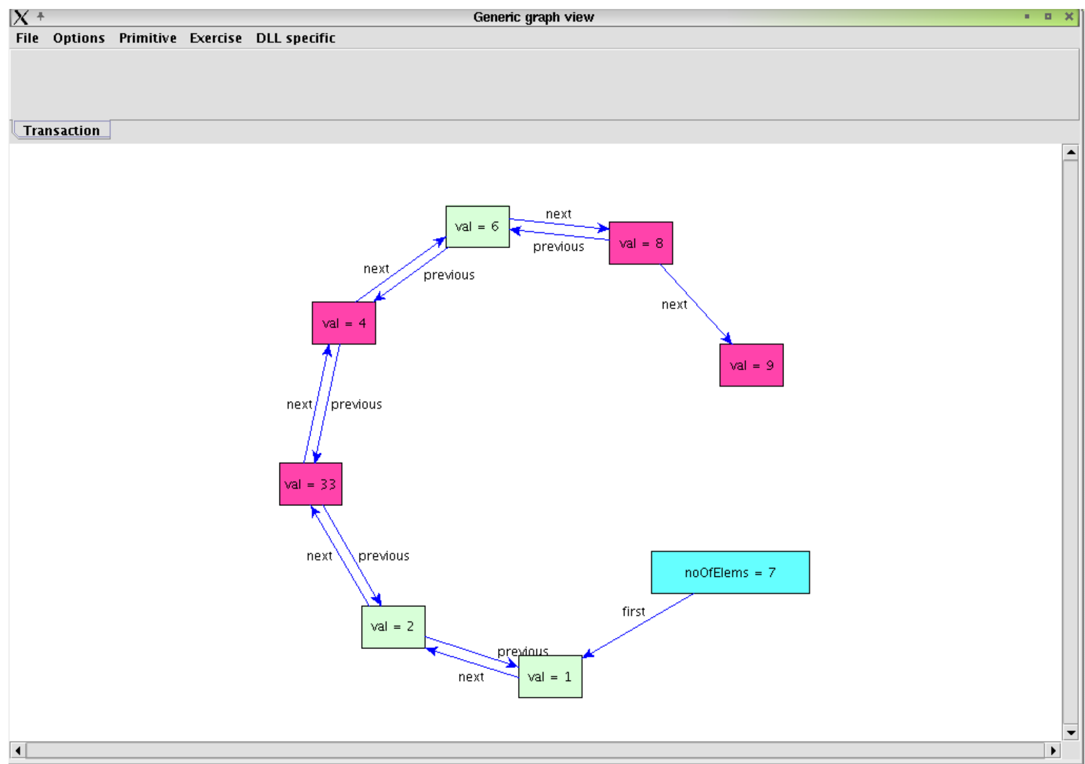


Abbildung 81: DLL-Instanz: Eine Reparatur-bedürftige Liste wurde aufgebaut

oder ob er anhand der laufenden VIDEA-Instanz die notwendigen Lösungsschritte vorführt. In dem gerade beschriebenen Beispiel könnte er etwa die fehlenden Kanten an den korrekten Stellen einfügen und den Knoten 33 entweder aus der Liste entnehmen und an der korrekten Stelle einfügen, oder seinen Wert in 3 ändern, je nachdem, welches Lernziel erreicht werden soll.

Im nächsten Abschnitt beschreiben wir ein Einsatzszenario mit etwas mehr Interaktivität als das reine Vorführen, in dem der Lernende zumindest die betrachteten Beispiele und die Reihenfolge der Nutzung dieser Beispiele selbst festlegt.

7.2. Passive Nutzung von VIDEA

Die zweite Einsatz-Möglichkeit von VIDEA, das *passive Betrachten vorgegebener Abläufe*, richtet sich an Lernende, die im Rahmen des Übungsbetriebs oder beim Selbststudium Teile von Visualisierungen ablaufen lassen wollen oder sollen, in denen es bis auf die grundsätzliche Steuerung des Ablaufs wenig Interaktion gibt.

Der Unterschied zum vorher beschriebenen *Vorführen* liegt hauptsächlich darin, dass nicht der Dozent die Oberfläche bedient, sondern der Lernende selbst. Das führt aber auch dazu, dass die VIDEA-Instanz auf eine andere Art aufbereitet werden muss, so dass z. B. die Abläufe selbsterklärend sind und die Fehlermeldungen deutlicher. Das soll aber nicht heißen, dass eine VIDEA-Instanz für die passive Nutzung einfach „besser“ aufbereitet und dokumentiert sein

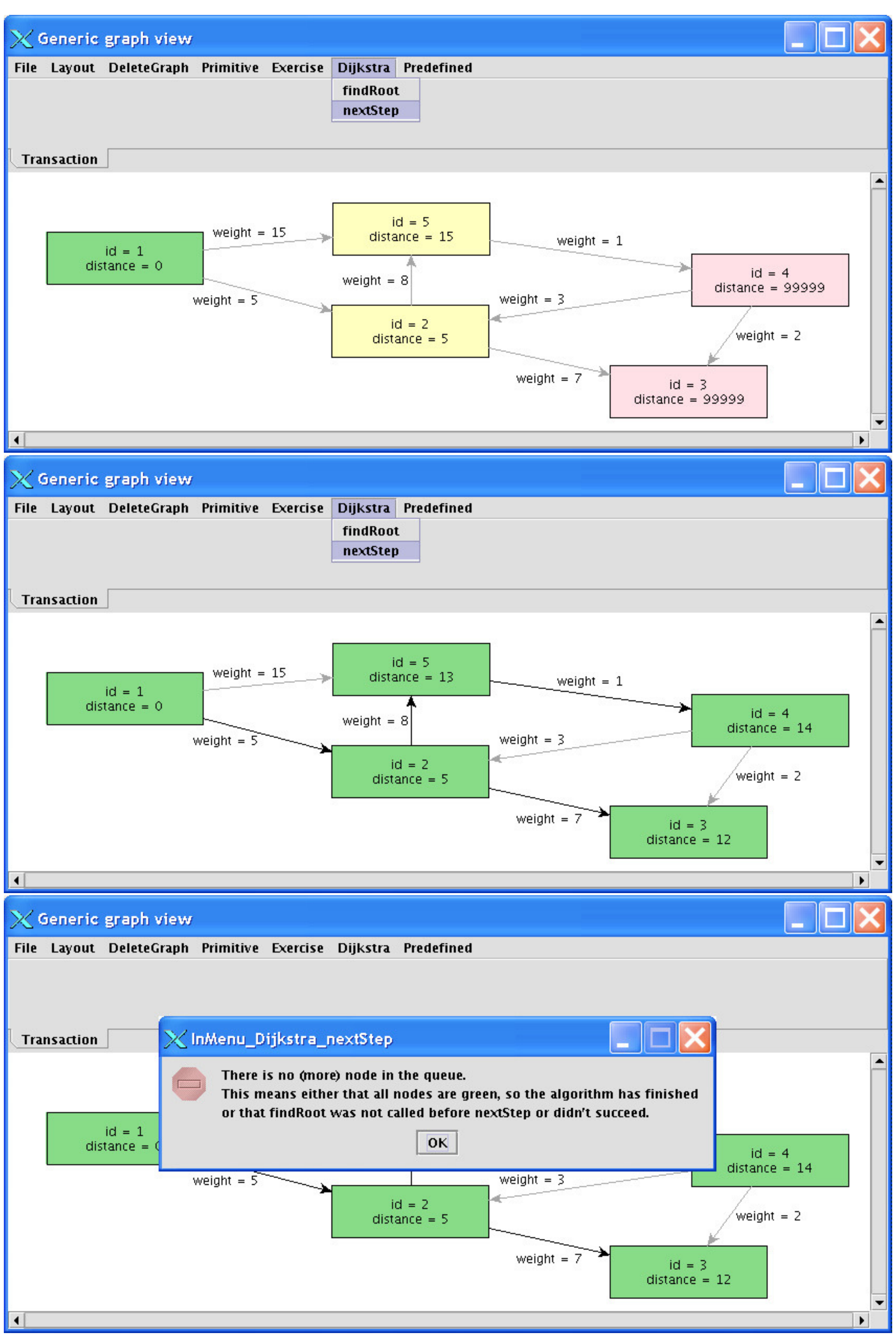


Abbildung 82: Beispielsitzung mit Simulation des Dijkstra-Algorithmus

muss als eine VIDEA-Instanz, die nur für das Vorführen konfiguriert wurde. Denn der Dozent baut ja das Wissen um seine mündlichen Erklärungen bereits in die VIDEA-Instanz ein, die zum Vorführen vorgesehen ist. Dadurch kann er an manchen Stellen die Übersichtlichkeit erhöhen, indem nur ein Minimum an Informationen angezeigt wird und zum Beispiel Meldungen möglichst kurz gehalten sind. Die fehlenden Informationen kann er mündlich ergänzen oder gemeinsam in der Vorlesung erarbeiten lassen. Im Kontrast dazu kann er an anderen Stellen viele Attribute anzeigen lassen und die zur Verfügung stehende Zeit nutzen, um nach und nach alle visualisierten Informationen zu erklären.

Beim hier beschriebenen passiven Betrachten gilt das Umgekehrte: Die Informationen sollten an allen Stellen ausreichend für das Verständnis sein, aber den Lernenden nicht überfordern. Bezogen auf die Konfigurationsmöglichkeiten der VIDEA-Instanz heißt das z. B., im Menü nur die notwendigen PROGRES-Programmeinheiten anzubieten, nur die notwendigen Knoten- und Kantenattribute anzuzeigen, Fehlermeldungen ausführlicher zu gestalten, Farbschemata unmittelbar verständlich zu wählen usw.

Es ist zu beachten, dass hier zwar von verschiedenen VIDEA-Instanzen die Rede ist, aber durchaus der gleiche generierte Code vorliegen kann. Technisch genügt es, das Verzeichnis der VIDEA-Instanz zu kopieren und die Konfigurationsdateien anzupassen, um diese Anpassungen persistent durchzuführen.

Ein weitere Möglichkeit, die bei der passiven Nutzung von VIDEA sinnvoll sein kann, ist das Stellen von Fragen in einigen vordefinierten Situationen. Die Antworten auf diese Fragen können etwa vom Lernenden auf einem Übungsblatt notiert und später in der Vorlesung oder im Übungsbetrieb diskutiert werden.

Abb. 82 zeigt die passive Nutzung der Dijkstra-Instanz, indem nach der Auswahl eines Beispielgraphen an diesem immer die gleichen Schritte nachvollzogen werden, nämlich das Finden der Wurzel und danach das Ausführen des nächsten Schrittes des Algorithmus, bis entweder der kürzeste Weg zu einem festen Knoten gefunden wurde und der Benutzer deshalb abbricht, oder bis für alle Knoten der kürzeste Weg gefunden wurde und der Algorithmus endet. Abb. 82 oben zeigt den Zustand des Graphen nach dem Finden der Wurzel (*findRoot*) und einmal *nextStep*. In der Mitte wurde fünf mal *nextStep* aufgerufen, was bewirkt, dass für alle Knoten die minimalen Abstände von der Wurzel und die zugehörigen kürzesten Wege bekannt sind (gekennzeichnet dadurch, dass alle Knoten grün sind und die zu kürzesten Pfaden gehörigen Kanten schwarz, im Schwarzweißdruck jeweils dunkler zu erkennen als die übrigen Knoten bzw. Kanten). Unten ist eine Fehlermeldung zu sehen, weil versucht wurde, nach Beendigung des Algorithmus nochmal *nextStep* aufzurufen.

Die normale Vorgehensweise bei der passiven Nutzung ist nun, den Graphen zu löschen, einen anderen auszuwählen und wieder den Algorithmus Schritt für Schritt zu verfolgen. Falls der Lernende nun doch Lust darauf bekommen sollte, aktiv zu werden, kann er vermuten oder raten, welcher Knoten als Nächstes grün werden wird. Falls es gewünscht wird, die Antwort auf diese Frage VIDEA-gestützt zu überprüfen, ohne den nächsten Schritt schon durchzuführen, ist dies möglich, falls der Dozent entsprechende Tests vorbereitet hat. Allerdings wird dadurch der Bereich der passiven Nutzung verlassen und derjenige des interaktiven Erforschens und Aufgaben-Lösens betreten, den wir im nächsten Abschnitt beschreiben.

7.3. Interaktives Erforschen und Aufgaben-Lösen

Das *interaktive Erforschen und Aufgaben-Lösen* ist eines der Hauptanwendungsgebiete der Instanzen unseres VIDEA-Rahmenwerks. Hier sitzen ein oder mehrere Lernende an einem Rechner, auf dem eine VIDEA-Instanz läuft, und lösen vordefinierte Aufgaben bzw. führen vordefinierte Schritte durch. Typischerweise findet das im Rahmen einer Übungsgruppe statt, die von einem Tutor geleitet wird. VIDEA wurde wiederholt in einem solchen Szenario eingesetzt; das jeweilige Umfeld und die Ergebnisse behandeln wir genauer in Kapitel 9. Dabei wird eine konkrete Aufgabenstellung besprochen, die z. B. in Form eines Übungsblattes vorliegt, und einen Rahmen für die Experimente der Lernenden vorgibt.

In den nächsten Abschnitten beschreiben wir die interaktive Nutzung von VIDEA in vier Instanzen, die aus den in Kapitel 6 vorgestellten Spezifikationen generiert und sinnvoll konfiguriert wurden.

7.3.1. Eine VIDEA-Instanz für AVL-Bäume

In unserer AVL-VIDEA-Instanz beobachten die Lernenden zunächst den Aufbau eines AVL-Baums inklusive aller Rotationsoperationen, in der Art, wie er in Abschnitt 7.1 beschrieben war. Der Start eines solchen Aufbauvorgangs wird, genau wie alle weiteren folgenden Aktionen, in VIDEA durch den Aufruf von Operationen im Menü – oder, falls vom Dozenten vorgesehen – über eine Toolbar aufgerufen.

Die Aktionen, die durch Benutzeraktionen angestoßen werden, lassen sich einteilen in Aktionen

- zum Beenden der VIDEA-Instanz,
- zum Start der evtl. angebotenen Farb-, Anzeige- oder Stilmanager (eine genauere Beschreibung dieser Manager folgt in Kapitel 8),
- zum Aufruf vordefinierter Skripte,
- zum Aufbau vordefinierter Datenstruktur-Instanzen, hier also von Bäumen,
- zum Erzeugen und Löschen von Knoten und Kanten,
- zum Ändern eigenständiger Attribute (z. B. *val*),
- zum Aufruf von Tests,
- zum Umschalten in ein neues, vom Dozenten vorbereitetes Farbschema,
- zum Aufruf von Schnittstellenoperationen der Datenstruktur (z. B. Rotationsoperationen)
- und zum Aufruf von Operationen, die Schritte des Algorithmus implementieren.

Je nach Definition der Operation muss diese mit oder ohne Parameter aufgerufen werden. Um hier eine sinnlose Aufzählung der einzelnen Operationen zu vermeiden, deren Spezifikation aus Kapitel 6 ja bereits bekannt ist, beschreiben wir im Folgenden die bei einer typischen Sitzung der jeweiligen VIDEA-Instanz genutzten Operationen in einem Lernzusammenhang. Wie mehrfach betont, wird die Reihenfolge der Schritte dabei nicht vom Werkzeug erzwungen, sondern als externer Fahrplan zur Verfügung gestellt.

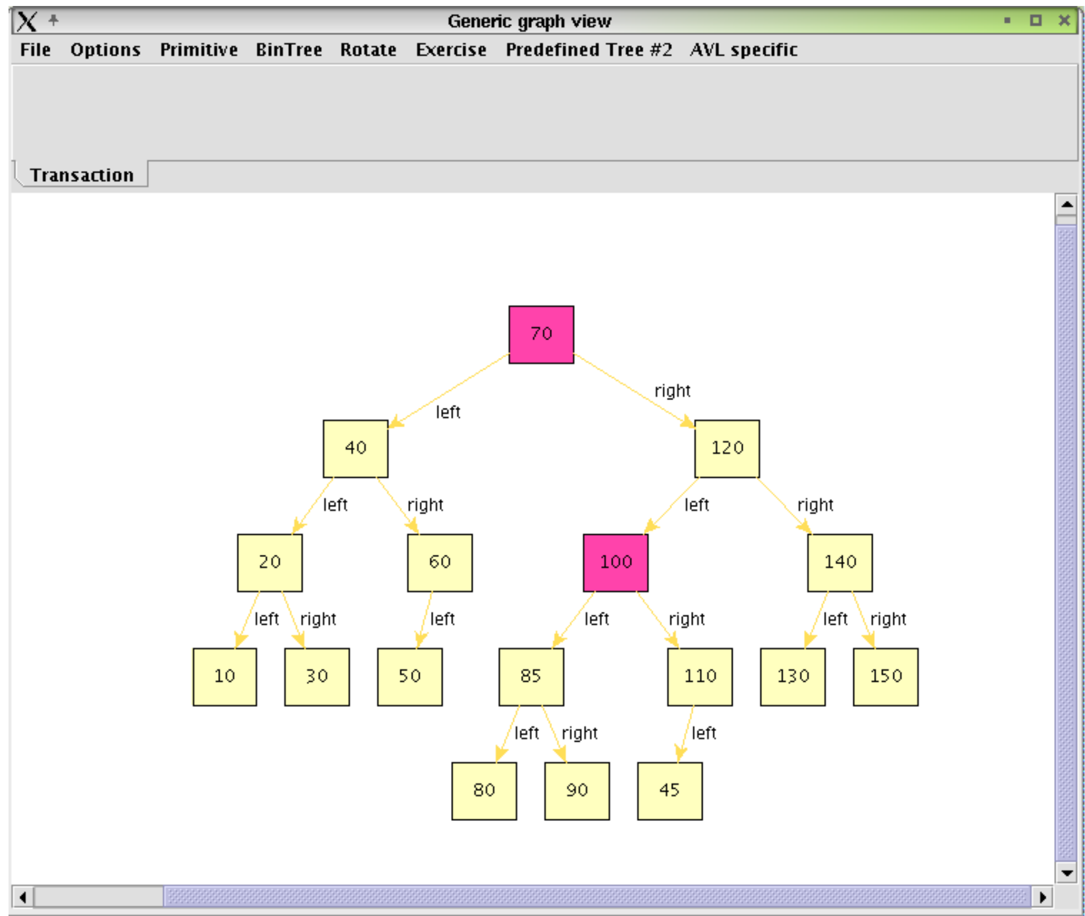


Abbildung 83: AVL-Instanz: Verletzte Sortierungseigenschaft zweier Knoten

Nachdem der AVL-Baum in flüssiger Animation korrekt aufgebaut ist, wird durch den Aufruf einer vordefinierten Operation ein Fehler in Form eines falsch einsortierten Knotens eingebaut, was dazu führt, dass das versteckte Sortierungsattribut *isSorted* mindestens eines Knotens im Baum den Wert *false* annimmt und somit mindestens für einen Knoten die Sortierungseigenschaft verletzt ist. Die erste Aufgabe der Lernenden ist es nun, herauszufinden, welche Knoten betroffen sind. Die Berechnungsformel dafür ist auf dem Übungsblatt so angegeben, wie in der PROGRES-Spezifikation verwendet. Nach Festhalten ihrer Vermutung auf dem Übungsblatt können sie diese überprüfen, indem sie die verletzten Eigenschaften eines AVL-Baums rot anzeigen lassen. Der resultierende Zustand ist in Abb. 83 dargestellt.

Eine Möglichkeit, den Baum zu reparieren, ist nun, den falsch einsortierten Knoten 45, der für die Teilbäume mit den Wurzeln 70 und 100 die Sortierungseigenschaft zerstört, von seinem Elternknoten zu entfernen. Dadurch wird der Baum wieder komplett in seiner normalen Farbe dargestellt, und Knoten 45 kann an geeigneter Stelle manuell eingefügt werden. Dafür stehen primitive Operationen wie *setLeft* und *setRight* zur Verfügung. Doch die Aufgabe ist in unserem

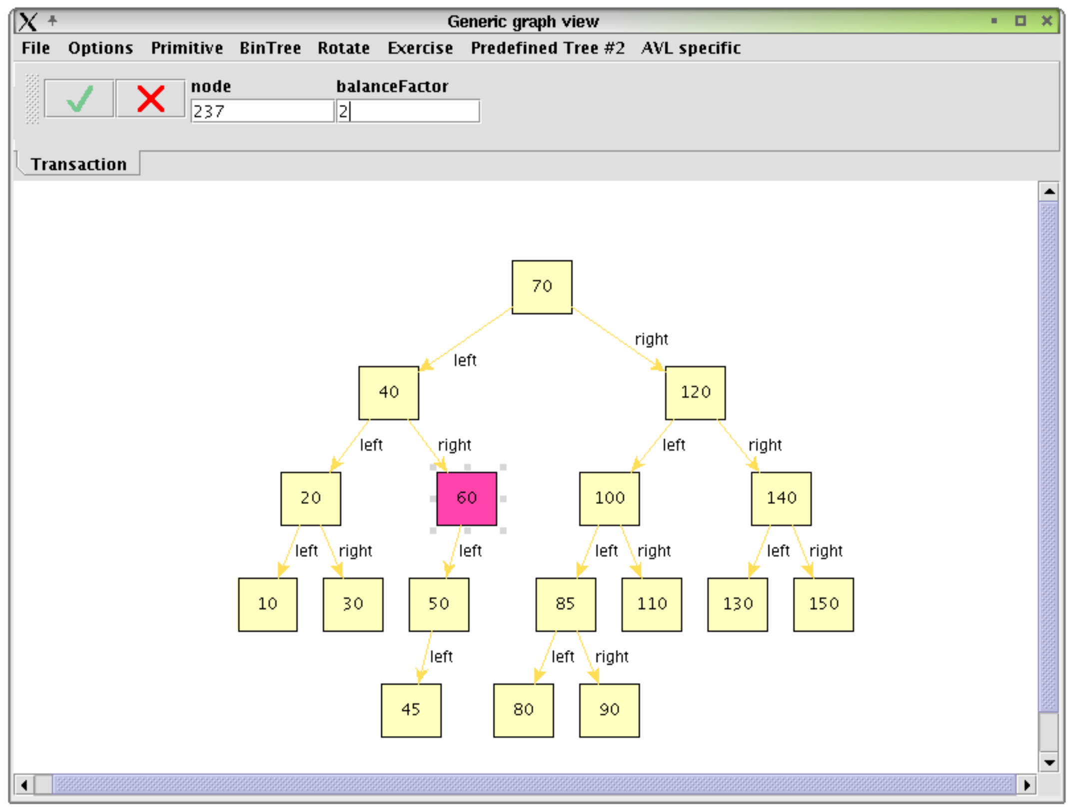


Abbildung 84: AVL-Instanz: Verständnistest bei der Berechnung des Balance-Faktors

vorausgesetzten Fahrplan so konzipiert, dass durch das korrekte Einfügen des Knotens 45 nun der Balance-Faktor einiger anderer Knoten gestört ist. Wiederum können die Lernenden selbst überlegen, an welchen Stellen im Baum dies der Fall ist. Sie können testen, ob sie den Balancierungsfaktor richtig berechnen können (Abb. 84, aufgerufen wurde hier übrigens der PROGRES-Test *checkBalance*, dessen Implementierung in Abb. 56 in Kapitel 6 zu sehen war). Wegen des knotenwertigen Parameters von *checkBalance* wird die interne Knotennummer 237 des Knotens mit Wert 60 angezeigt (Abb. 84). Natürlich wäre es auch möglich, den Knotenwert übergeben zu lassen. Dann würde statt 'node' über dem Parameterfeld der Bezeichner 'val' des Wert-Attributs stehen. Hier haben wir uns für die interne ID entschieden, da die Lernenden so gleich den Unterschied zwischen einem Objekt und dessen Attributen lernen.

Nun sollen die Lernenden durch eine geeignete Rotationsoperation auch die Balancierungseigenschaft wiederherstellen. In Abb. 85 oben wurde anstatt der eigentlich korrekten Rechtsrotation eine Linksrechtsrotation gewählt und eine dementsprechende Fehlermeldung angezeigt. Nachdem der Lernende das Feedback verstanden und hoffentlich in einen internen Lernschritt umgesetzt hat, führt er die Rechtsrotation aus und kommt damit in die Situation von Abb. 85 unten: Hier wird kein Knoten rot gemalt, denn wir haben einen gültigen AVL-Baum vor uns. Knoten 60 erscheint im Schwarzweißdruck nur deshalb etwas dunkler, weil er als ehemali-

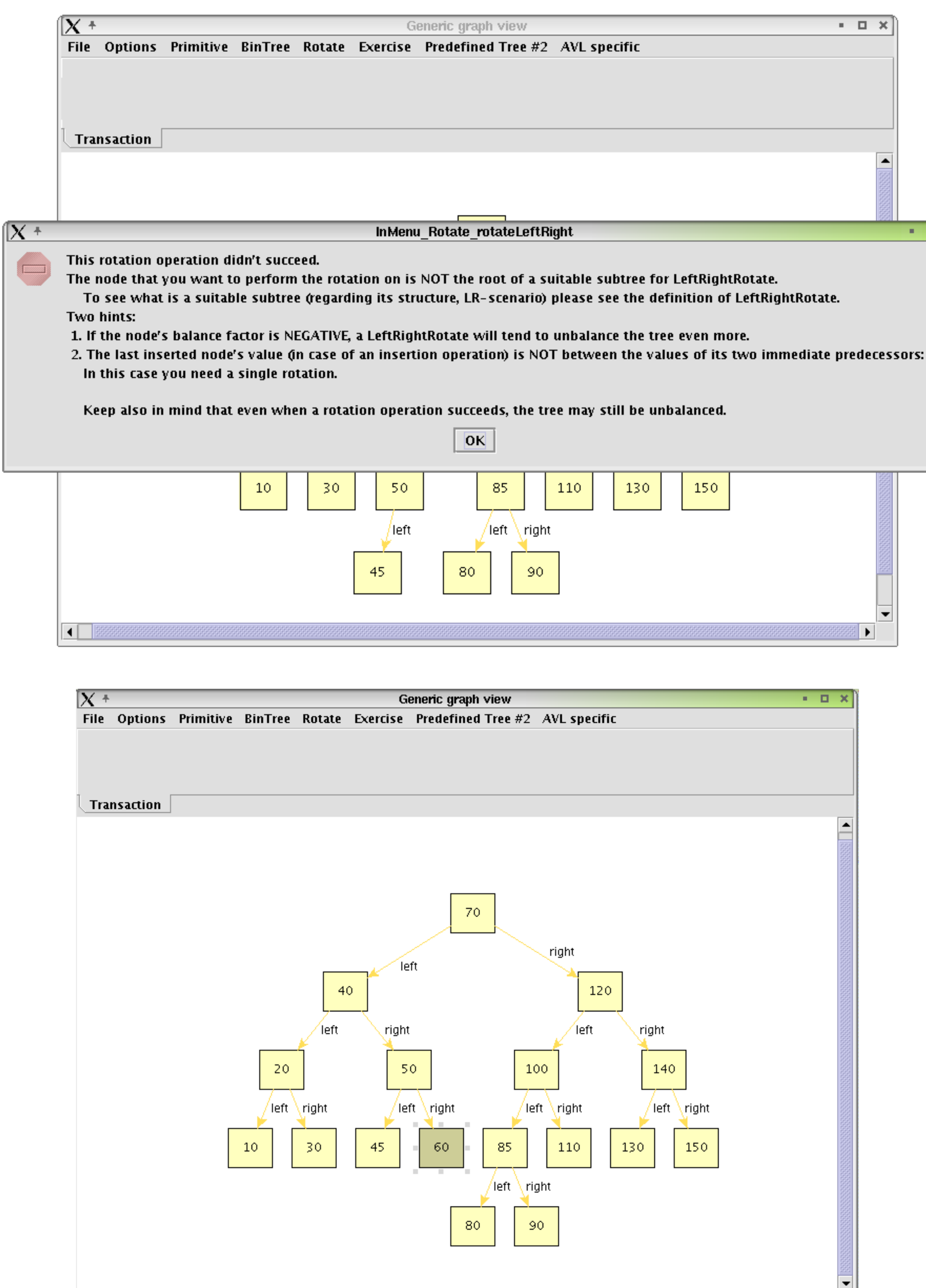


Abbildung 85: Oben: Nach Wahl der falschen Rotationsoperation. Unten: Der Benutzer hat die korrekte Rotationsoperation gefunden

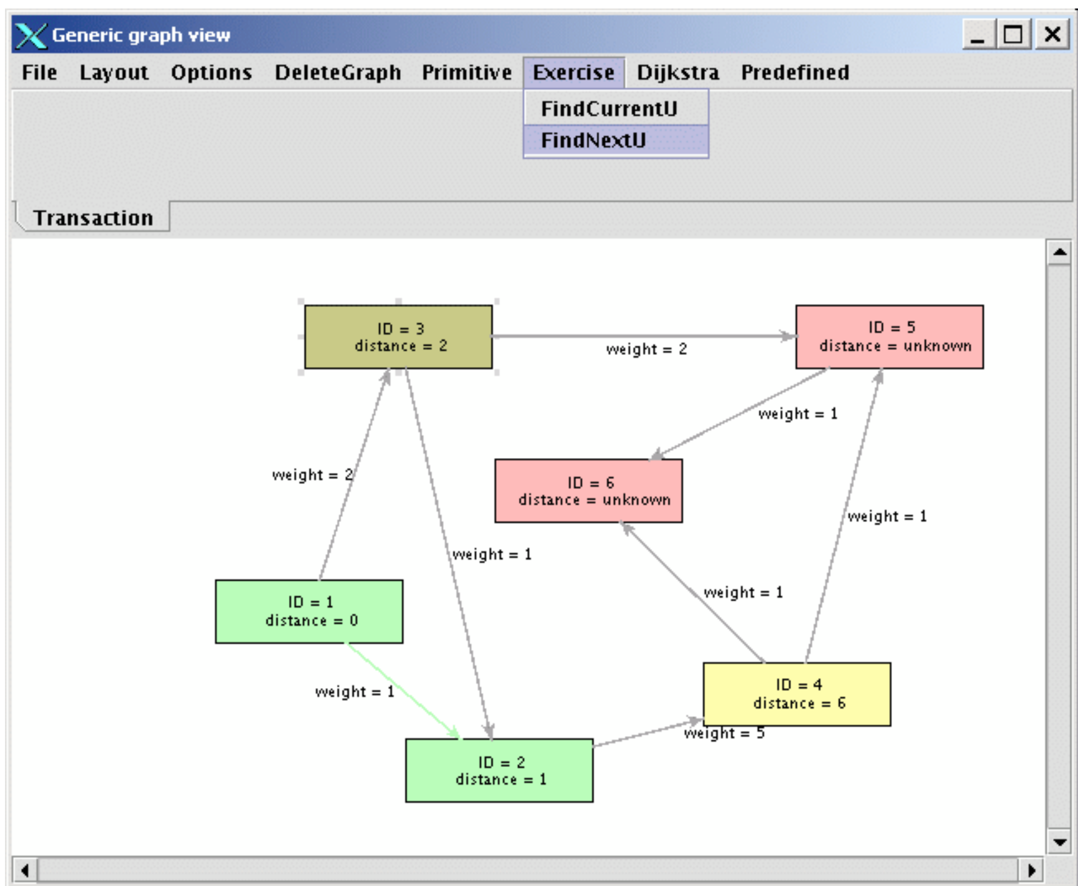


Abbildung 86: Dijkstra-Instanz: Test, um das Verständnis des Algorithmus zu überprüfen

ger Parameter des interaktiven Aufrufs der Rechtsrotation noch markiert ist.

Weitere Aufgaben sind die Bestimmung der Höhenwerte einiger Knoten und eigene abschließende Experimente mit dem Einfügen selbst gewählter Werte und – im Falle der dadurch entstehenden Unbalanciertheit – dem Finden der korrekten Balancierungs-Operationen.

7.3.2. Eine VIDEA-Instanz für die kürzeste Wegesuche nach Dijkstra

Als zweites Beispiel für *interaktives Erforschen und Aufgaben-Lösen* zeigen wir eine weitere VIDEA-Instanz, die implementiert wurde, um den Dijkstra-Algorithmus zur kürzesten Wegesuche auf gerichteten Graphen zu unterrichten. Wesentliche Teile der PROGRES-Spezifikation hatten wir bereits in Kapitel 6 vorgestellt. Genau wie für die AVL-Instanz werden wir auch für die Dijkstra-Instanz in Kapitel 9 ein Evaluationsszenario und die zugehörigen Ergebnisse präsentieren.

In dieser Lerneinheit kann zu Beginn ein Graph aus einer Liste von vier vordefinierten Graphen ausgewählt werden. Außerdem werden wieder Operationen für die schrittweise Ausfüh-

rung des Algorithmus zur Verfügung gestellt. Zusätzlich sind Tests vorhanden, mit denen der Lernende zeigen kann, ob er in der Lage ist, den weiteren Verlauf des Algorithmus selbständig vorauszusagen und wesentliche knotenwertige Variablen des Algorithmus als Knoten im aktuellen Graphen zu identifizieren. In Abb. 86 prüft der Benutzer gerade, ob der markierte Knoten (ID=3) der gemäß Dijkstra-Algorithmus als Nächstes betrachtete Knoten U ist (siehe auch in Kapitel 5, Seite 80 bei der Erklärung des Algorithmus).

Wir stellen die Knoten genau wie im Algorithmus in drei verschiedenen Farben dar, die wir allerdings aus Gründen der Prägnanz der Animation geändert haben. Statt der in der gängigen Form des Algorithmus vorgesehenen Farben weiß, grau und schwarz verwenden wir die Ampelfarben rot, gelb und grün. Somit sind zu Beginn alle Knoten rot gefärbt, beim Wandern in die Prioritäts-Warteschlange wird der jeweilige Knoten gelb, und wenn die genaue Distanz zu diesem Knoten bekannt wird, grün. In Abb. 86 erscheint der markierte Knoten mit (ID=3) nicht in seiner eigentlichen gelben Farbe, sondern durch die verminderte Transparenz etwas dunkler, wie auf graphischen Benutzeroberflächen üblich.

Natürlich muten wir den Lernenden in unserer Beispiel-Übung nicht zu, bezüglich der Farben umdenken zu müssen, sondern stellen ihnen auf dem zugehörigen Übungsblatt gleich den Algorithmus mit den neuen Farben vor.

Die Sitzung mit dieser VIDEA-Instanz endet, wenn der Lernende auf einer vorgegebenen Anzahl von Graphen in dieser Art den Algorithmus simuliert und erforscht hat. Ähnlich wie bei Dijkstra läuft die Arbeit mit der im folgenden Abschnitt beschriebenen Kruskal-Instanz ab.

7.3.3. Eine VIDEA-Instanz für Kruskal

In der VIDEA-Instanz für die Lehre des Kruskal-Algorithmus wird ähnlich wie bei Dijkstra zunächst vom Lernenden ein Graph ausgewählt und dann Schritt für Schritt der Algorithmus teils simuliert, teils vom Lernenden beschrieben oder vorhergesagt. Für die Vorhersage stehen die in Kapitel 6 beschriebenen Tests *testNextConsideredEdge* und *isNextConsideredEdgeTaken* zur Verfügung. *testNextConsideredEdge* ermöglicht dabei dem Lernenden, eine Kante zu identifizieren und zu markieren, die seiner Meinung nach die nächste von Kruskal betrachtete Kante ist. Danach gibt VIDEA entweder eine bestätigende oder eine verneinende Meldung aus, so dass der Lernende ggf. die nächste Chance erhält, die korrekte Kante zu bestimmen. Der Unterschied zum passiven Betrachten der Schritte des Algorithmus ist nicht nur, dass hier aktiv eine Eingabe gemacht werden muss, sondern auch, dass der Lernende bei einem fehlerhaften Verständnis des Algorithmus nicht sofort die richtige Lösung serviert bekommt. Dadurch erhöhen sich seine Chancen, eine realistische Selbsteinschätzung zu gewinnen, bevor er mit den Schritten des Algorithmus fortfährt.

Der zweite Test *isNextConsideredEdgeTaken* erwartet die Aussage, ob die nächste vom Algorithmus betrachtete Kante des Graphen zum Spannbaum hinzugenommen wird oder nicht. Wegen des booleschen Eingabeparameters wird hier eine bisher nicht gezeigte Benutzerschnittstelle angeboten: Der Lernende spezifiziert durch Ankreuzen einer 'checkbox' das von ihm vermutete Ergebnis des Tests. In Abb. 87 ruft der Lernende zunächst den Test *isNextConsideredEdgeTaken* auf und entscheidet sich für die Antwort „ja“, vermutet also, dass die nächste betrachtete Kante zum Spannbaum hinzugefügt wird. Da die Antwort korrekt ist, bekommt er eine bestätigende Rückmeldung von VIDEA (Abb. 87 unten).

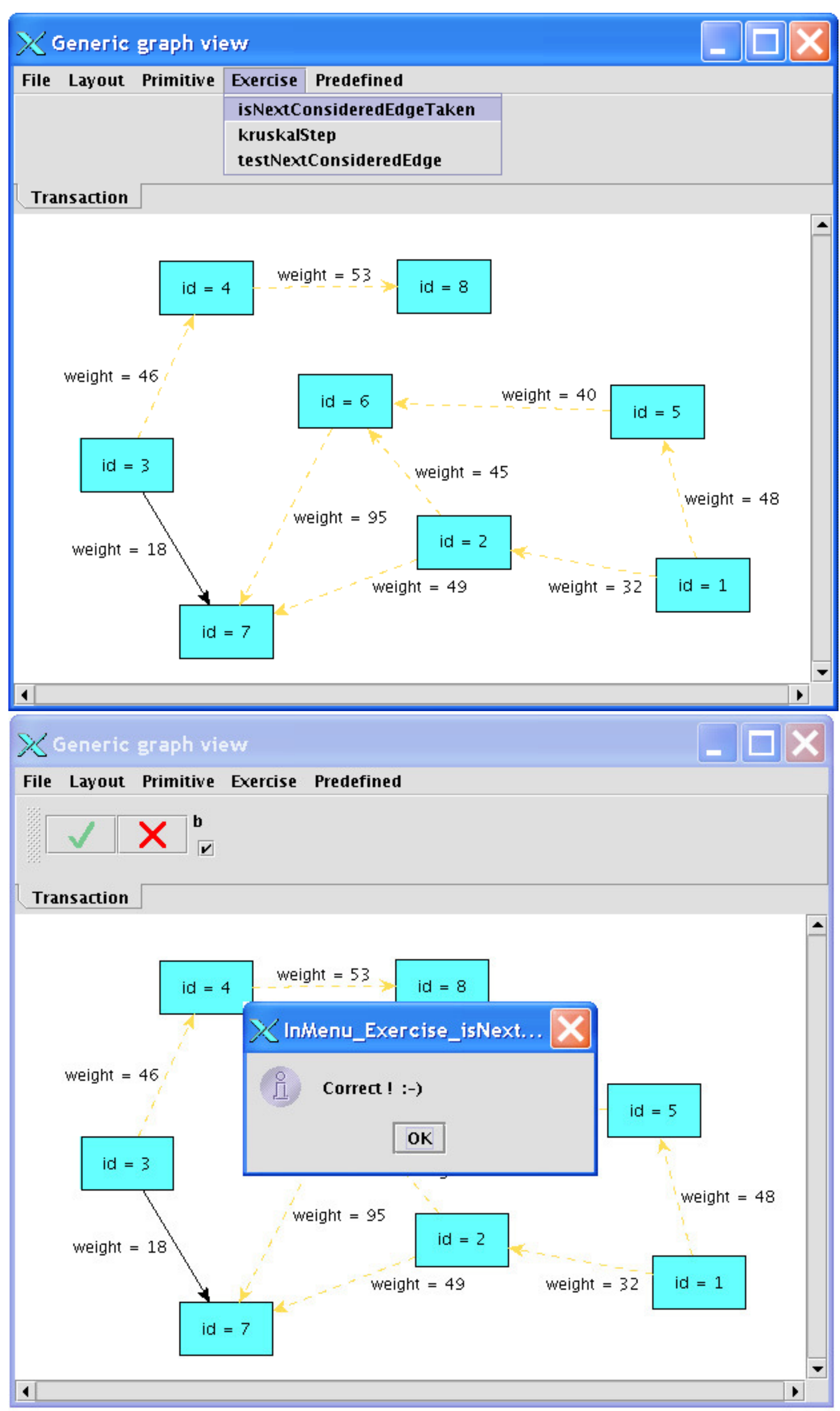


Abbildung 87: Nutzung eines booleschen Tests in der Kruskal-Instanz. Oben: *isNextConsideredEdgeTaken* wird aufgerufen. Unten: Der Test wurde durch Anklicken der 'checkbox' korrekt mit „Ja“ beantwortet.

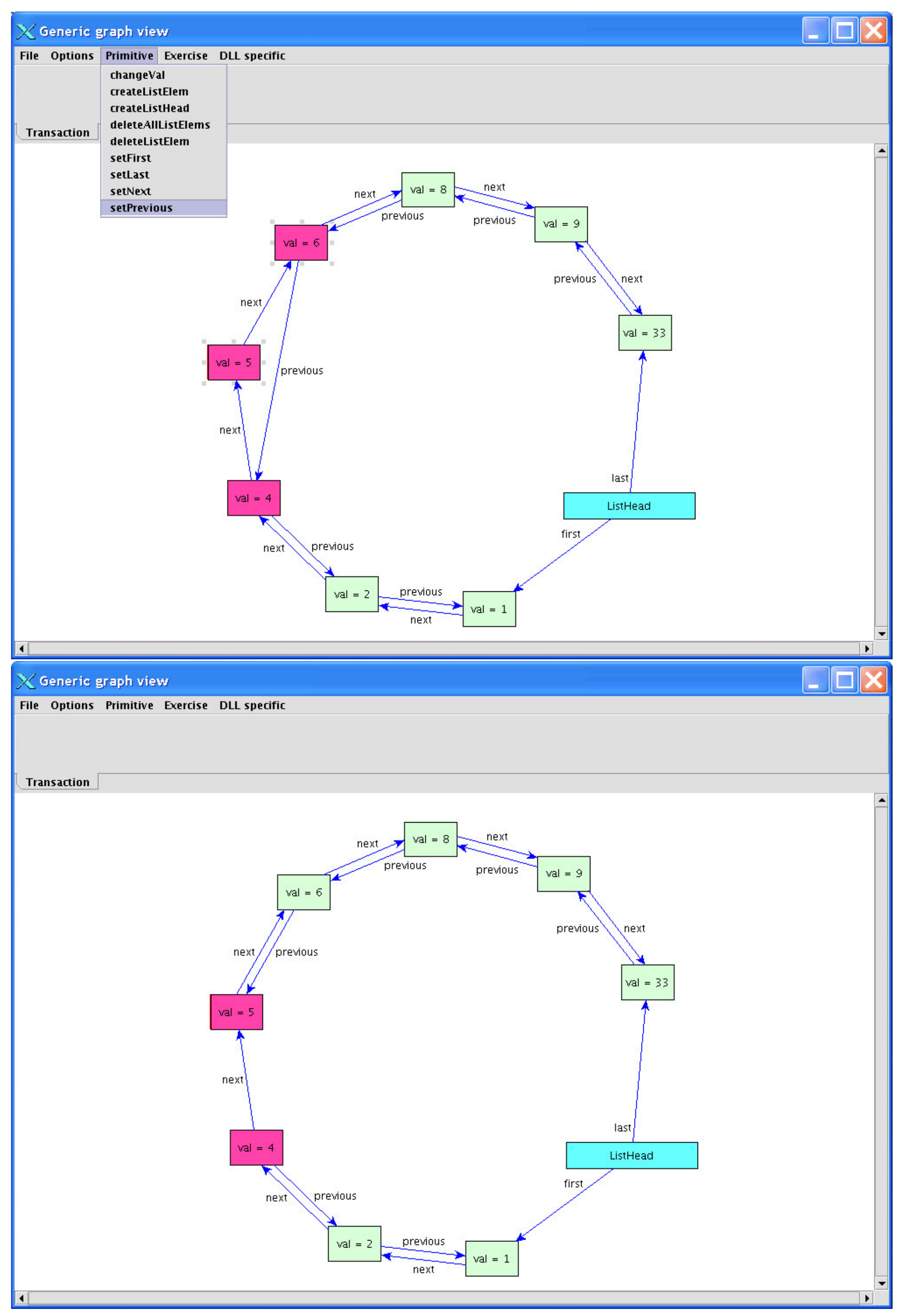


Abbildung 88: Der Nutzer fügt ein neues Element '5' in eine doppelt verkettete Liste ein. Oben: Zwei Knoten wurden markiert, um eine neue 'previous'-Kante von Element '6' nach Element '5' zu ziehen und gleichzeitig die alte 'previous'-Kante zu entfernen. Unten: Die Kante wurde erzeugt, die Knoten '5' und '6' sind noch markiert.

7.3.4. Eine VIDEA-Instanz für die doppelt verkettete Liste

Abb. 88 zeigt einen Teil des Ablaufs beim manuellen Einfügen eines neuen Listenelements '5' in eine zuvor aufgebaute Liste zwischen den existierenden Listenelementen '4' und '6'. Hier wurden neben dem neuen Listenelement auch schon einige neue Kanten erzeugt, es fehlt aber u. a. noch eine *previous*-Kante zwischen Knoten '6' und '5'. Gleichzeitig besteht noch eine jetzt fehlerhafte *previous*-Kante von '6' zu '4'. Beide Aufgaben werden durch den Aufruf von *setPrevious* erledigt. Den resultierenden Zustand zeigt Abb. 88 unten. Da immer noch eine *previous*-Kante zwischen '5' und '4' fehlt, werden die Knoten '4' und '5' weiterhin rot dargestellt. Die nächste Aufgabe der Lernenden ist es, die rote Färbung dieser beiden Knoten korrekt zu interpretieren und die *previous*-Kante an der richtigen Stelle einzufügen.

Nach diesem Querschnitt über beispielhafte Situationen beim interaktiven Erforschen und Aufgaben-Lösen zeigen wir im nächsten Abschnitt das Szenario, das den meisten Einsatz von Seiten der Lernenden erfordert, und deswegen die besten Lernresultate verspricht: Den ferngesteuerten Einsatz von VIDEA, der eine einfache Form des visuellen Debuggings eigener Programme ermöglicht.

7.4. Ferngesteuerte Animation eigener Programme der Lernenden

Eine weitere wichtige Fähigkeit von VIDEA im Rahmen unseres didaktischen Konzepts ist der erwähnte Einsatz zur ferngesteuerten Algorithmenanimation, mithilfe der etwas erreicht werden kann, was einem visuellen Debugging eigener Programme der Lernenden sehr nahe kommt.

Als Szenario kommt eine einfache Programmieraufgabe im Rahmen eines Praktikums oder einer Hausaufgabe im Übungsbetrieb in Frage, in der die Lernenden z. B. im Fall der Benutzung von Java eine Klasse `AVLNode` entwickeln sollen, die einen AVL-Baum implementiert. Wir setzen im Folgenden Java als zu lernende Programmiersprache voraus, möglich ist aber jede Sprache, die CORBA-Aufrufe absetzen kann (da wir intern CORBA verwenden, siehe Kapitel 8).

Gegeben ist eine lauffähige VIDEA-Instanz und eine vom Dozenten zur Verfügung gestellte abstrakte Klasse `Node`, die durch eine interne Schnittstelle bereits in der Lage ist, entfernte Aufrufe an die laufende VIDEA-Instanz abzusetzen. Details zur VIDEA-Architektur und zur Implementierung dieser entfernten Verbindung folgen in Kapitel 8.

Die Lösung der gestellten Aufgabe besteht in der Implementierung einer Klasse `AVLNode`, die `Node` ableitet. Genauer gesagt, sollen

- ein oder mehrere Konstruktoren der Klasse `AVLNode`,
- einfache Methoden wie *setLeft* und *setRight* und
- komplexere Methoden wie *rotateLeft* und *rotateRight*

von den Lernenden erstellt und getestet werden.

Üblicherweise ist es beim Testen der Lösung für ein solches Programmier-Beispiel sinnvoll, spezifische textuelle Debugging-Informationen zusätzlich zur eigentlichen Funktionalität des Programms ausgeben zu lassen, um beurteilen zu können, ob es korrekt arbeitet.

Zum Beispiel genügen bei der Implementierung einer Rotationsoperation für den AVL-Baum schon geringfügige Fehler bei den stattfindenden Zeiger-Zuweisungen, um den Baum

```
public class AVLNode extends Node  {

    public AVLNode(int val, AVLNode left, AVLNode right)  {
        super(val, left, right);
    }


    protected Node rotateLeft()  {
        // still missing: check a for !=null

        Node a = this.getRight();
        Node b = a.getLeft();

        this.setRight(b);
        this.setLeft(a);

        return a;

        // this.calculateHeight();
        // this.calculateBalance();
    }

    ....
    ....

    public static void main(String[] p)  {
        AVLNode a = new AVLNode(10,null,null);

        a.binTreeInsert(20);
        a.binTreeInsert(15);
        a.binTreeInsert(30);

        a = (AVLNode) a.rotateLeft();
    }
}
```

Abbildung 89: Ausschnitt eines möglichen, eigenen Java-Programms der Lernenden, das auf der Klasse `Node` basiert

völlig zu entstellen. Solche Fehler sind aber ohne Debugging schwer zu finden. Textuelles Debugging bietet bereits den Vorteil, dass schrittweise die Belegungen von Variablen etc. dargestellt werden und somit im Fehlerfall genau reproduziert werden kann, was, wann, warum schiefgeht. Trotzdem erfordert ein textuelles Debugging beim AVL-Baum entweder einige Erfahrung mit der zu lernenden Datenstruktur oder ein sehr gutes visuelles Vorstellungsvermögen. Wir gehen im Komfort deswegen noch einen Schritt weiter und bieten durch die Möglichkeit, ferngesteuert und schrittweise Animationen ausführen zu können, eine Art visuellen Debugger an.

Dazu geben wir eine Java-Klasse `MusoftClient` vor, die über CORBA eine Verbindung zum `MusoftServer` (ein Bestandteil des normalen VIDEA-Paketes) aufbaut und über diese Verbindung den Aufruf von PROGRES-Transaktionen veranlasst, die wiederum zu neuen Ani-

```
public abstract class Node {

    private int val;
    private Node left;
    private Node right;
    ...

    private static MusoftClient mc;

    protected Node(int val, Node left, Node right) {
        this.val = val;
        this.left = left;
        this.right = right;

        if(mc == null)
          {  mc = new MusoftClient();
             mc.init();
          }

        if(left == null && right == null)
          mc.showTransaction("InMenu_Primitive_makeNodeByNodeVal " + val);
    }

    protected Node getLeft() {
        return left;
      }

    protected void setLeft(Node newLeftChild) {
        this.left = newLeftChild;
        mc.setLeft(mkVal(this), mkVal(newLeftChild));
        // Re-calculate height
    }

    private int mkVal(Node a)  {
        if(a==null)
          return(0);
        else
          return(a.getVal());
    }

    protected void binTreeInsert(int nodeVal)  {
        ...
        ...
        mc.showTransaction("InMenu_BinTree_insert " + nodeVal);
    }

    ...
}
```

Abbildung 90: Ausschnitt aus der Klasse Node.

mations-Schritten führen.

Da es notwendig ist, dass der MusoftClient vom Programm der Lernenden aus aufgerufen wird, die Aufrufe aber möglichst von der eigentlichen Problemlösung entkoppelt sein sollen, stellen wir im AVL-Baum-Beispiel ebenso eine abstrakte Klasse Node zur Verfügung, die bereits den Konstruktor und einige Methoden wie *setLeft()* und *setRight()* verkapselt, und die dafür notwendigen Aufrufe an den MusoftClient absetzt. Die Methoden für die Rotation geben wir abstrakt vor, um ihre Implementierung mit einer definierten Signatur zu erzwingen.

Die Lernenden können sich also ganz auf die Java-Aufgabe konzentrieren und eine Klasse AVLNode entwickeln, die aussieht wie in Abb. 89, wo wir Ausschnitte einer lauffähigen Lösung dargestellt haben. Abb. 90 zeigt Teile der zur Verfügung gestellten Klasse Node.

Die verwendete Methode *binTreeInsert()* zum Einfügen in einen Binärbaum wurde bei uns bereits vorgegeben, kann aber natürlich auch Bestandteil der Aufgabe sein. In unserem Fall nutzt *binTreeInsert()* genau wie *setLeft()* und *setRight()* den MusoftClient *mc*, wie in *setLeft()* beispielhaft gezeigt (Abb. 90). Das setzt natürlich voraus, dass solche einfachen Methoden bereits im MusoftClient bekannt sind, was letztlich eine Design-Entscheidung des Dozenten ist, wie wir noch ausführlicher in Kapitel 8 darlegen werden.

Es sei an dieser Stelle betont, dass sich diese Design-Entscheidungen nicht auf VIDEA selbst auswirken und somit auch keine weitergehenden Aktionen des Dozenten erforderlich sind (also kein neues Übersetzen o.Ä.).

Abb. 91 oben zeigt ein Szenario, das bei der ferngesteuerten Animation des beschriebenen Programms AVLNode.java entstehen kann. Wir sehen zwei Fenster nebeneinander: Links wurde in Together ControlCenter AVLNode.java geöffnet und eine Debugging-Sitzung auf übliche Art mit einem Haltepunkt in der Prozedur für die Linksrotation (nach dem Aufbau eines rechtslastigen Baums in der *main()*-Methode) gestartet. Natürlich kann auch jede andere graphische Java-Entwicklungsumgebung verwendet werden, die ein schrittweises Debugging erlaubt. Wir haben Together eingesetzt, weil die Lernenden dann im Rahmen der Übungsaufgabe gleich ein UML-Klassendiagramm für AVLNode und Paare von Objektdiagrammen für die Rotationsoperationen zeichnen bzw. betrachten können.

Rechts ist das Fenster der VIDEA-AVL-Instanz zu sehen, die leer gestartet wurde und aufgrund der bisherigen CORBA-Aufrufe den Baum bis zum Punkt des Haltepunktes aufgebaut und als unbalanciert gekennzeichnet hat (durch die Rotfärbung des Knotens 10). Das Wissen um diese Verletzung der Balancierungsbedingung steckt zunächst nur in der VIDEA-Instanz. Das Programm der Lernenden arbeitet lediglich mithilfe der zur Verfügung gestellten Bausteine auf einer Datenstruktur und weiß nicht, dass durch die Implementierung dieser Bausteine (*Node()*, *setLeft()*, *binTreeInsert()* usw.) parallel in der VIDEA-Instanz auf einer äquivalenten Datenstruktur gearbeitet wird. Einen Debugging-Schritt weiter wurde der nächste Befehl *this.setRight(b)* ausgeführt und ein Zeiger umgehängt, was zu der Situation nach der „halben“ Rotation in Abb. 91 unten führt.

Bis hierher entspricht der gezeigte Ablauf einer korrekten Lösung der Aufgabenstellung, denn die temporäre Verletzung der Integritäts-Bedingung für AVL-Bäume (erkennbar am dunkleren Knoten 15 in Abb. 91 unten) ist nicht zu vermeiden. Damit der Lernende, der eine korrekte Lösung produziert hat, hier nicht in die Irre geführt wird, muss natürlich vorher eindeutig vermittelt worden sein, was genau die verwendeten Farbschemata bedeuten, und dass ein roter Knoten nicht unbedingt einen Fehler im Programm bedeutet, sondern eine temporäre Verletzung einer

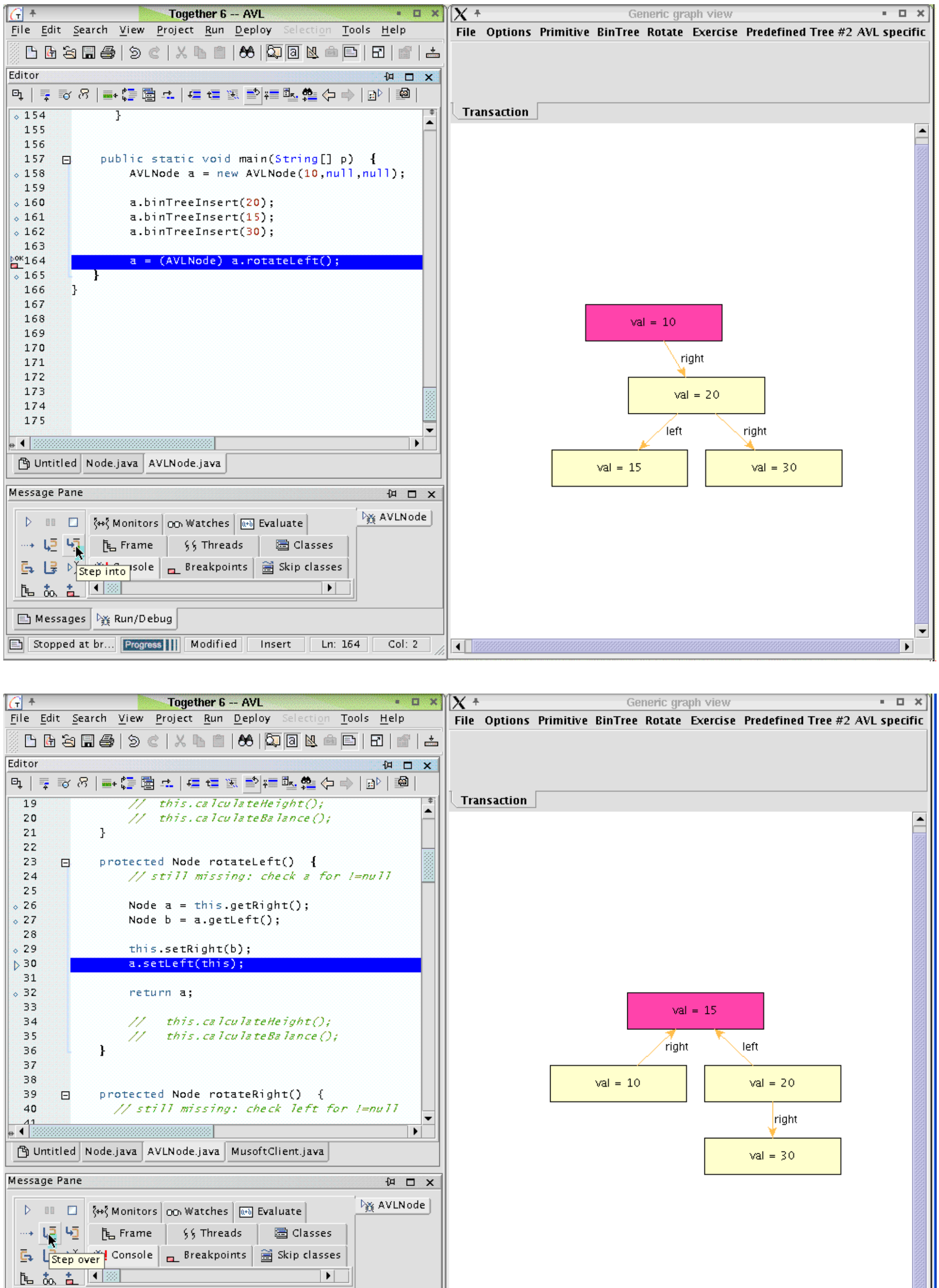


Abbildung 91: Oben: Aus einer Debugging-Sitzung in Together wurde bereits ein unbalancierter AVL-Baum in VIDEA über die CORBA-Schnittstelle aufgebaut. Unten: Der Zustand nach der Hälfte von rotateLeft; man sieht, dass vorübergehend die Baum-Eigenschaft verloren geht

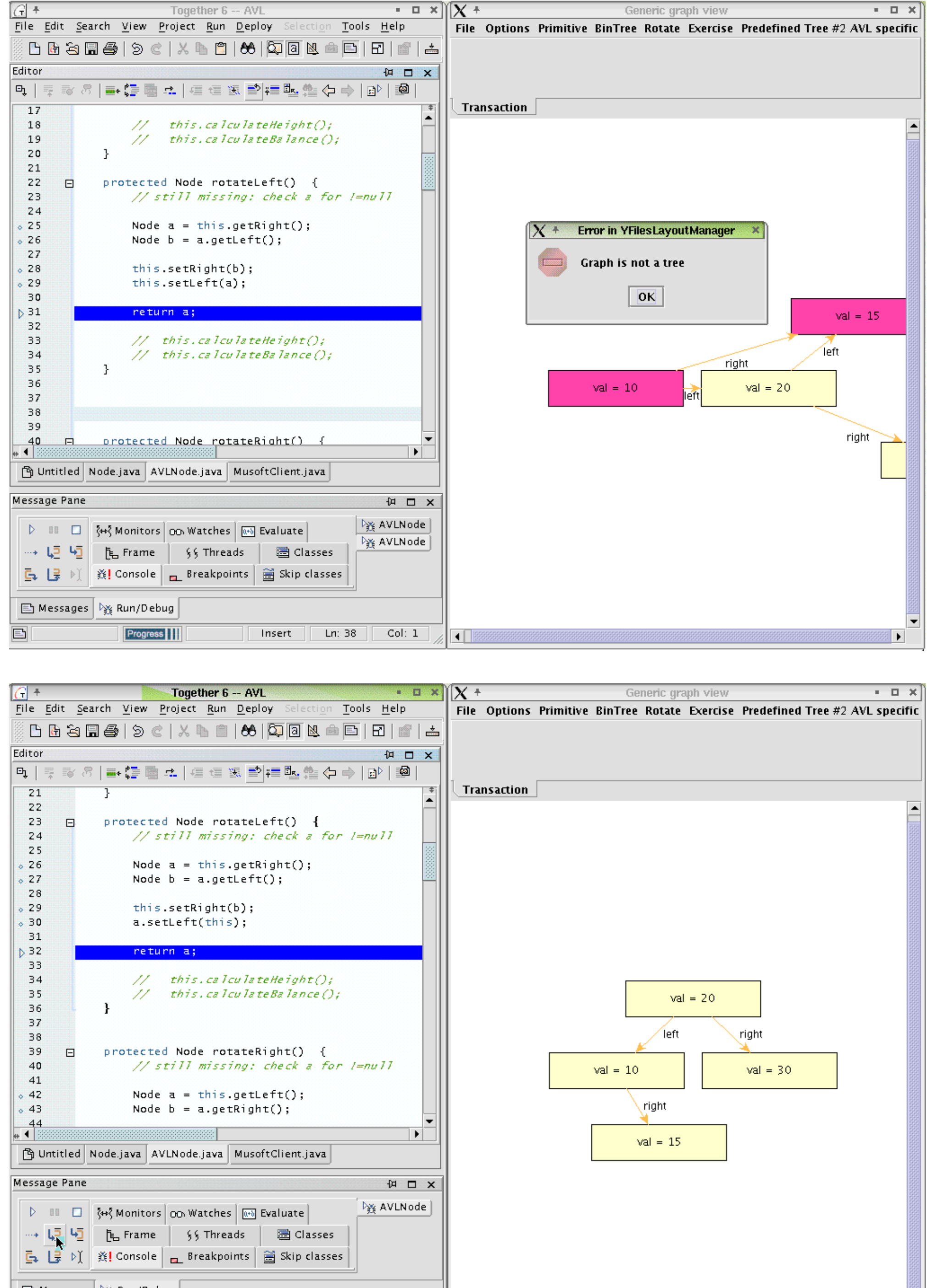


Abbildung 92: Oben: Eine falsche Lösung, die durch Umhängen den Baum komplett zerstört. Unten: Eine korrekte Lösung: Der Baum wurde linksrotiert und ist deshalb wieder ein gültiger AVL-Baum.

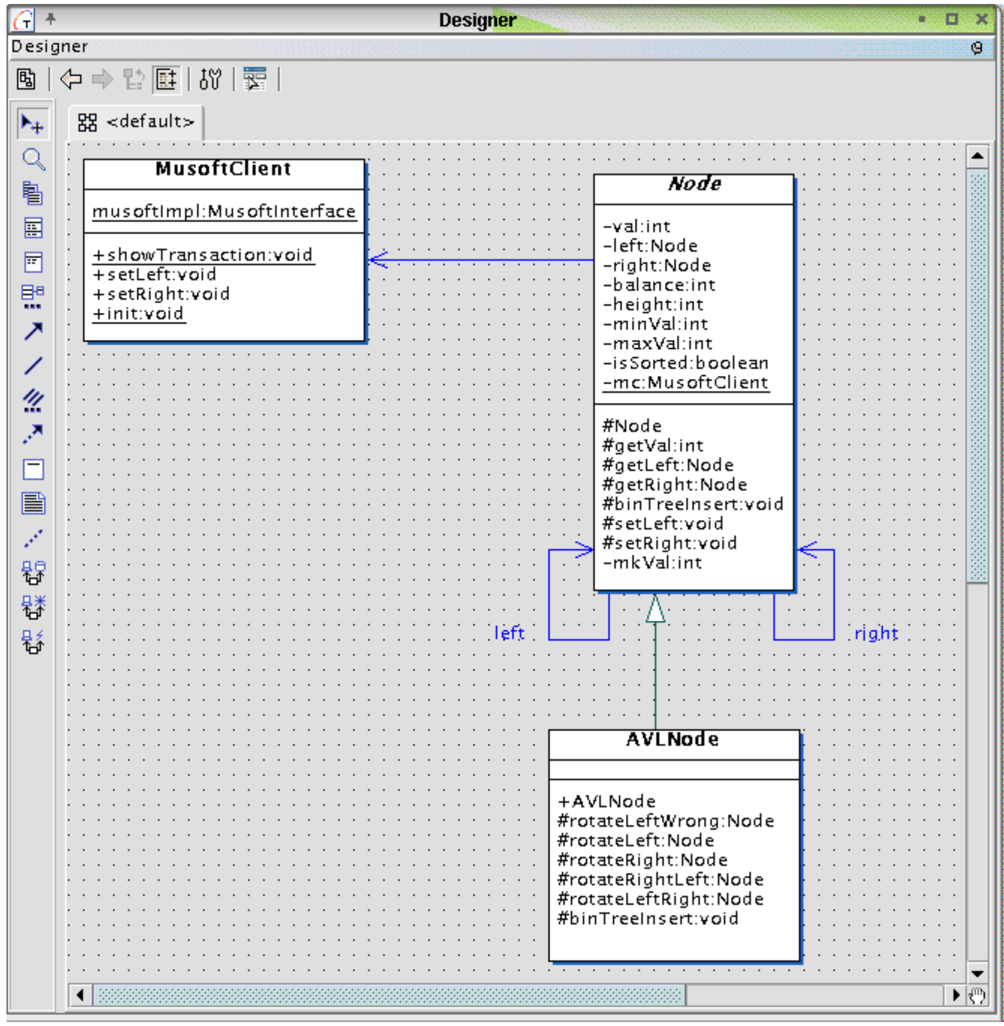


Abbildung 93: Klassendiagramm des AVL-Baums in Together

wesentlichen Eigenschaft der Datenstruktur. Hier wird also das Verständnis der Operationen auf AVL-Bäumen vertieft, indem vermittelt wird, dass es sich bei der AVL-Baum-Eigenschaft um keine strenge Invariante der Rotationsoperation handelt, die an jeder Stelle innerhalb der Methode gilt. Am Ende der Rotation sollte allerdings der Baum wiederhergestellt sein; falls es sich um das Ende einer übergeordneten Operation handelt, die wir hier nicht betrachten, die aber analog zu handhaben sind (z. B. das Einfügen oder Löschen in einem AVL-Baum), müssen alle AVL-Eigenschaften wiederhergestellt sein.

Aber nun zerstört in unserem Beispiel ein kleiner Fehler im Programm der Lernenden nicht nur die AVL-Baum-Eigenschaft, sondern macht es zusätzlich dem Layout-Algorithmus unmöglich, den Baum noch vernünftig zu zeichnen, wie in Abb. 92 oben zu sehen ist. Da aber die Knoten und ihre verbindenden Kanten trotz der Probleme des Layout-Algorithmus in diesem Fall für

den Lernenden deutlich zu erkennen sind, hat dieser die Chance, seinen Fehler zu identifizieren, auszubessern und in einer neuen Debugging-Sitzung das veränderte Verhalten zu überprüfen.

Zum Neustart einer ferngesteuerten Animation in VIDEA genügt es, über das Menü den Graphen zu löschen. Die Applikation muss also nicht jedesmal neu gestartet werden, weil VIDEA lokale Benutzung und Fernsteuerung problemlos verschränkt verarbeiten kann. Together kann in so einem Fall natürlich auch weiterlaufen.

Deswegen ist es also auch möglich, in VIDEA einen Baum oder ein beliebiges Szenario manuell über das Menü aufzubauen und dann das Java-Programm dagegen laufen zu lassen, wobei die entsprechenden Java-AVLNode-Objekte fehlen können. Eine sinnvolle Anwendung dieses Szenarios könnte der graphische Test eines Java-Prototypen sein, der grundlegende Operationen wie das Erzeugen von Objekten, *setLeft()* und *setRight()* noch gar nicht implementiert, sondern nur an VIDEA delegiert, und höhere Methoden wie die Rotationsoperationen auf die grundlegenden abstützt. Dadurch kann bereits graphisch getestet werden, bevor in einem zweiten Schritt der eigentliche, funktionale Java-Code für die restlichen Methoden erzeugt wird.

Das Ergebnis einer ferngesteuerten Animation der korrigierten Version des Programms der Lernenden zeigt Abb. 92 unten. Man sieht, dass – wiederum nach den Zwischenschritten aus Abb. 91) – die Rotationsoperation erfolgreich ausgeführt wurde und der rebalancierte Baum ein gültiger AVL-Baum ist (zu erkennen an der hellen Färbung aller Knoten).

Für Lernzwecke kann es sinnvoll sein, zusätzlich zum Erstellen und graphischen Testen von Java-Programmen auch die Erstellung von UML-Diagrammen zu fordern. Abb. 93 zeigt als Beispiel das Klassendiagramm von `AVLNode` basierend auf `Node` in Together. Analog kann man die Studierenden auch Diagramme von Operationen zeichnen lassen, ähnlich wie z. B. in Abb. 36 unten auf Seite 95 für die Linksrotation gezeigt.

Alles in allem bietet diese ferngesteuerte Algorithmen- und Datenstrukturanimation eigener Programme, die wir auch „visuelles Debugging“ nennen, eine Menge zusätzlicher Möglichkeiten für den Übungs- oder Heimeinsatz, die die mächtigen interaktiven Möglichkeiten von VIDEA sehr gut ergänzen.

Anhand mehrerer konkreter VIDEA-Instanzen haben wir in diesem Kapitel die vier Lernszenarien beschrieben, die wir für den Einsatz einer multimedialen Lernumgebung für die Unterstützung der Lehre von Algorithmen und Datenstrukturen für entscheidend halten. Im nächsten Kapitel werden wir zeigen, wie VIDEA-Instanzen erstellt werden. Neben einer ausführlicheren Beschreibung des Generierungsprozesses als bisher, werden wir insbesondere die statischen und dynamischen Konfigurationsmöglichkeiten einer VIDEA-Instanz erklären.

8. Erzeugung und Konfiguration von VIDEA-Instanzen

In Kapitel 5 wurde VIDEA – inklusive der Möglichkeiten und Schritte bei der Generierung und Konfiguration von VIDEA-Instanzen – nur überblicksartig beschrieben. Dabei wurden aus Platzgründen wesentliche Themen wie der Ablauf der Generierung, die Nutzung der Konfigurationsdateien, die interaktive Bedienung der VIDEA-Manager und die Erstellung eines eigenen Layoutalgorithmus teils nur gestreift, teils gar nicht behandelt.

In diesem Kapitel holen wir das Versäumte nach und zeigen detaillierter als bisher die Möglichkeiten eines Dozenten bei der Spezifikation, Generierung und Konfiguration von VIDEA-Instanzen und die Hilfestellungen, die VIDEA in der jeweiligen Phase anbietet.

Abb. 94 zeigt die Arbeitsschritte des Dozenten im Überblick. Die Pfeile stehen für die Reihenfolge der notwendigen Schritte bei der Erstellung einer VIDEA-Instanz. Zunächst erstellt der Dozent im Idealfall eine UML-Spezifikation der Datenstruktur und des Algorithmus und überträgt diese von Hand nach PROGRES. Wir haben in Kapitel 6 gezeigt, dass diese Übertragung ein kleiner Schritt ist. Alternativ kann auch gleich eine PROGRES-Spezifikation erstellt werden. Hierbei sind in jedem Fall außer der reinen Spezifikation der Datenstruktur noch weitere Designentscheidungen zu treffen, wie etwa die Anordnung der notwendigen Operationen in einer Hierarchie von PROGRES-Programmeinheiten und die Festlegung der Signaturen der sichtbaren Operationen, die sich wiederum auf die Benutzerschnittstellen der Operationen in VIDEA auswirken.

Abschnitt 8.1 gibt einen Überblick über notwendige und hilfreiche Designentscheidungen

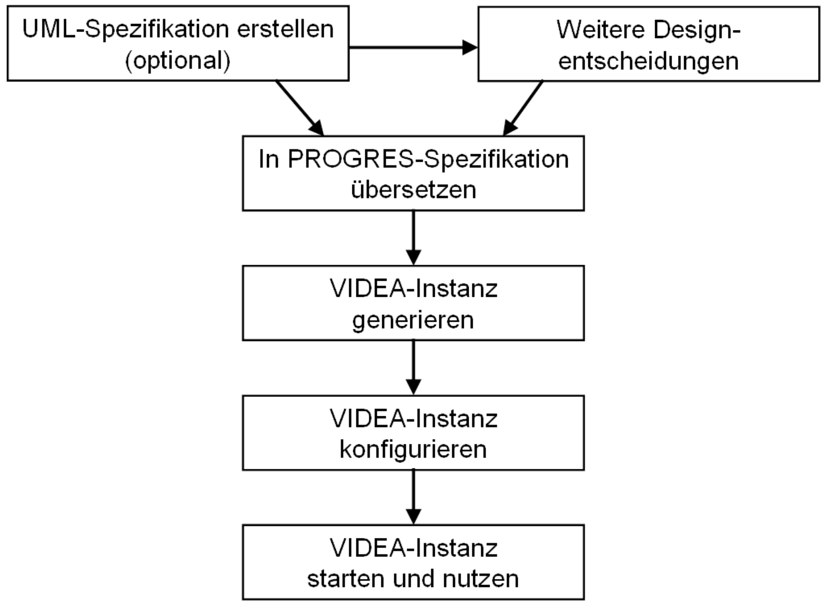


Abbildung 94: Schritte bei der Erstellung und Nutzung einer VIDEA-Instanz

des Dozenten in dieser Phase der Erstellung einer VIDEA-Instanz und zeigt Beispiele aus den in dieser Arbeit vorgestellten VIDEA-Instanzen.

Der nächste Schritt ist die Generierung einer VIDEA-Instanz. Dieser Schritt läuft in VIDEA automatisiert ab und stellt somit für den Dozenten keinen nennenswerten Aufwand dar. Der grobe Ablauf der Generierung wird in Abschnitt 8.2 beschrieben.

Die letzten beiden Schritte sind die Konfiguration und die Nutzung einer VIDEA-Instanz. Bei der Konfiguration spielen besonders die Erstellung von Skripten, die Nutzung der Konfigurationsdateien und gegebenenfalls die Erzeugung eines eigenen Layoutalgorithmus eine Rolle. Da die Bedienung einer VIDEA-Instanz keine erwähnenswerten Besonderheiten gegenüber gängiger Software auf graphischen Benutzeroberflächen bietet, ist auch im laufenden Betrieb die Konfiguration das eigentlich interessante Thema, nämlich die Konfiguration durch die interaktive Nutzung der VIDEA-Manager. *Manager* sind wie bereits erwähnt spezielle VIDEA-Klassen, die Ressourcen verwalten; bei den im Zusammenhang mit der Visualisierung gemeinten Managern geht es um diejenigen Klassen, die für die Verwaltung und Anzeige der Farbschemata für Knoten bzw. Kanten, für die Verwaltung und Darstellung der angezeigten Attribute von Knoten und Kanten und für den Stil von Kanten – gestrichelt oder durchgezogen gezeichnet – verantwortlich sind. In Abb. 95, wo VIDEA im Kontext der benutzten Komponenten und Werkzeuge dargestellt ist, befindet sich die Funktionalität dieser Manager aufgeteilt auf die beiden Funktionseinheiten „Logischer Teil von VIDEA“ und „Visueller Teil von VIDEA“. Die Themen der statischen bzw. dynamischen Konfiguration werden in den Abschnitten 8.3 und 8.4 beschrieben.

8.1. Design einer VIDEA-Instanz

Bezogen auf das in diesem Abschnitt beschriebene Design einer VIDEA-Instanz bewegen wir uns in Abb. 95 ausschließlich im Kasten „Spezifikation einer Datenstruktur und ihrer Operationen...“, was sich aber bei der Generierung sowohl auf den generierten Code für die Laufzeitumgebung der Graphersetzungsmaschine als auch auf den generierten Code für die Standard-Funktionalität der VIDEA-Instanz auswirkt.

Das Ziel bei der Erstellung einer VIDEA-Instanz ist

- eine neue oder abgewandelte *Datenstruktur* oder
- einen neuen oder abgewandelten *Algorithmus* auf einer Datenstruktur oder
- eine bekannte Datenstruktur oder einen bekannten Algorithmus *aus einer anderen Perspektive* zu unterrichten.

Im Falle der Entwicklung einer VIDEA-Instanz für eine abgewandelte Form einer Datenstruktur, für die bereits eine PROGRES-Spezifikation vorliegt, ist es möglich, auf der bestehenden VIDEA-Instanz aufzusetzen, wenn diese geeignet strukturiert wurde. Insbesondere spielt hier die in sinnvolle Ebenen aufgeteilte Hierarchisierung der PROGRES-Programmeinheiten eine Rolle, die in Abschnitt 8.1.6 genauer erläutert wird. Ein Beispiel ist die Erstellung einer VIDEA-Instanz für AVL-Bäume, wenn bereits eine VIDEA-Instanz für binärer Suchbäume vorliegt. Wenn sich das Design der existierenden Lerneinheit in Bedienung und Lehre bewährt hat,

- können viele Produktionen und Transaktionen, die sich auf gleichbleibende Eigenschaften der Datenstruktur beziehen, gleich bleiben; beim AVL-Baum z. B. *makeNode*, *insert* oder *setLeft*;

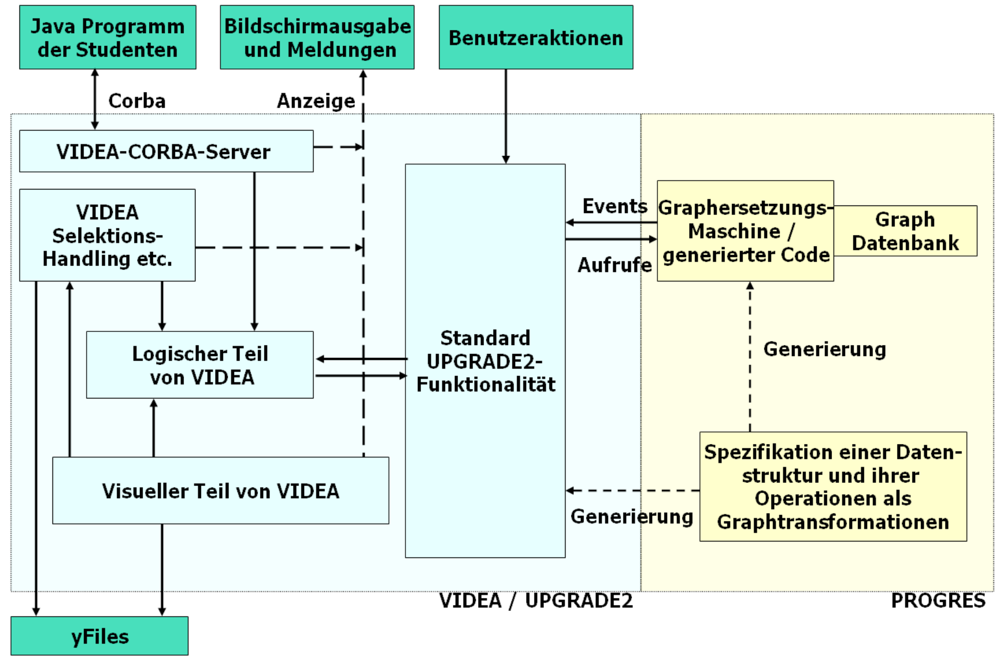


Abbildung 95: Einbettung des VIDEA-Rahmenwerks in das Werkzeug-Umfeld, Wiederholung von Abb. 43

- kann der im Fall der AVL-Bäume einzige Knotentyp *AVLNode* lediglich um einige Attribute für die zusätzlichen AVL-typischen Eigenschaften wie etwa *balance* erweitert werden; *val*, *isSorted* etc. können unverändert bleiben;
- können einige neue Produktionen und Transaktionen hinzugefügt werden, wie z. B. die Rotationsoperationen und zusätzliche Verständnistests für neue Eigenschaften der Datenstruktur – zum Beispiel *checkBalance* zur Überprüfung des Verständnisses des Balance-Faktors.

Soll dagegen eine VIDEA-Instanz von Anfang an neu erstellt werden, bietet sich eine gründlichere Bestandsaufnahme an mit Berücksichtigung der folgenden Fragen:

- Was genau soll unterrichtet werden?
- Worauf soll der Schwerpunkt des Lerneffekts der fertigen VIDEA-Instanz liegen?
- Wie soll die Darstellung am Bildschirm sein?
- Welche Schritte soll der Lernende ausführen müssen, welche soll er ausführen oder nicht ausführen dürfen?

Besonders die Antwort auf die letzte Frage kann dann individuell in die visuelle Spezifikation einfließen, wie wir im nächsten Abschnitt zeigen werden. Der Designvorgang einer VIDEA-Instanz ist graphisch in Abb. 96 aufgezeichnet. Wir gehen davon aus, dass der Dozent bereits

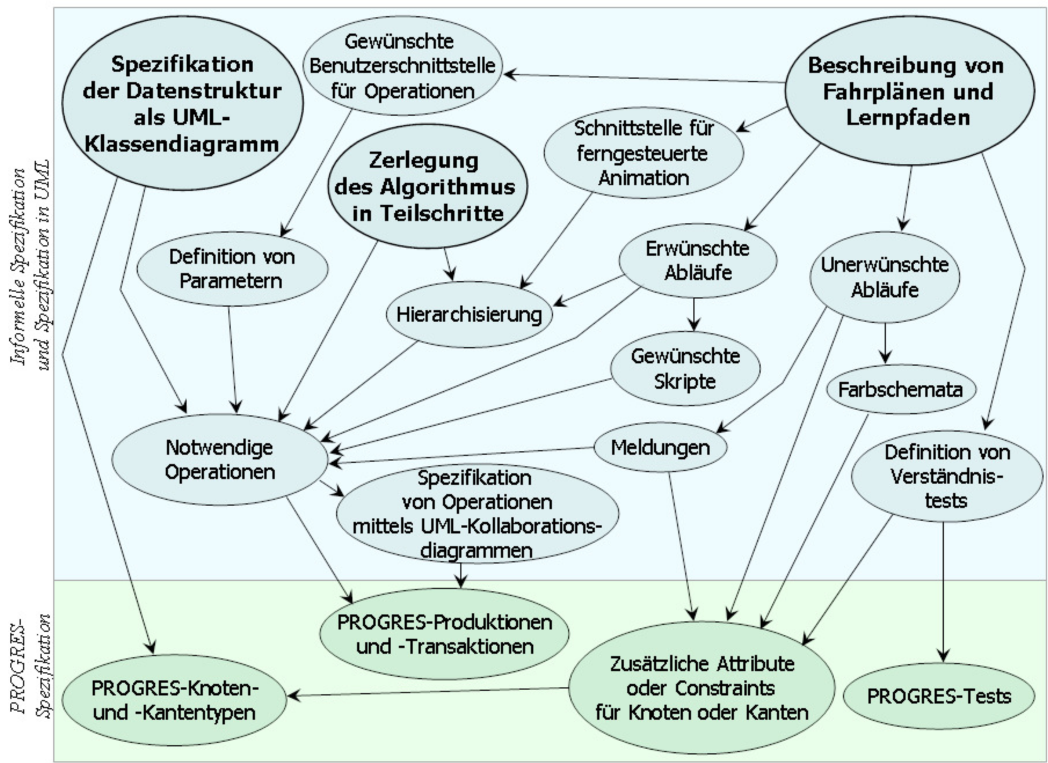


Abbildung 96: Detaillierter Ablauf beim Design einer VIDEA-Instanz; die Kanten bedeuten den sich ergebenden Informationsfluss, beschreiben also auch die zeitliche Abfolge der Beschäftigung des Dozenten mit den einzelnen Punkten.

UML-Diagramme der Datenstruktur, die Formulierung eines groben Handlungsablaufs für das zunächst angestrebte Einsatzszenario und ggf. eine Zerlegung des zu unterrichtenden Algorithmus in Teilschritte erstellt hat. Diese drei Quellen der weiteren Vorgehensweise sind in Abb. 96 als die drei dicker gezeichneten Vorgänge *Spezifikation der Datenstruktur als UML-Klassendiagramm*, *Beschreibung von Fahrplänen und Lernpfaden* und *Zerlegung des Algorithmus in Teilschritte* zu sehen. Von hier aus geht der Informationsfluss zu folgenden Arten von PROGRES-Konstrukten, die am unteren Bildrand aufgetragen sind:

- **PROGRES-Knoten- und Kantentypen** ergeben sich im Idealfall aus der Spezifikation der Datenstruktur als UML-Klassendiagramm. Im Beispiel der AVL-Bäume wurde z. B. aus der Klasse `AVLNode` mit Attributen *val*, *height* und so fort. der Knotentyp *AVLNode* mit Attributen *val*, *height* usw.

  Hinzu kommen Attribute, die sich aus der informellen Beschreibung von Fahrplänen und Lernpfaden ergeben, wie sie in Kapitel 5 eingeführt wurden. Zum Beispiel erfordert die gewünschte Möglichkeit, einen unsortierten Baum an den geeigneten Stellen rot einzufärben, die Definition eines Knotenattributs *isSorted* als Grundlage für den VIDEA-Farbmanager.

- **Eine Hierarchie (bzw. ein Abhängigkeits-DAG) von PROGRES-Produktionen und -Transaktionen** ergibt sich aus den für das gewünschte Lernszenario notwendigen Operationen. Diese wiederum ergeben sich aus den Schnittstellenoperationen der Datenstruktur (die idealerweise als Paare von UML-Objektdiagrammen spezifiziert werden), Operationen für das Ausführen eines zu unterrichtenden Algorithmus in Schritten verschiedenen Detaillierungsgrades und Hilfsoperationen für zu verwendende Skripte und ferngesteuerte Animationen.
- **PROGRES-Tests** ergeben sich aus den anzubietenden Verständnistests, die im Lernszenario vorgesehen sind. Zur Formulierung dieser Tests kann es wiederum sinnvoll sein, zusätzliche Knoten- oder Kantenattribute zu formulieren. Ein Beispiel in der AVL-Baum-Instanz ist der Test *checkBalance*, der testet, ob der Lernende den Balancefaktor eines Teilbaums korrekt berechnen kann. Zur Formulierung dieses Tests muss die entsprechende Information auf Knotenebene vorliegen – in diesem Fall als Knotenattribut *balance*.

In den folgenden Unterabschnitten werden wesentliche Themen des gerade beschriebenen Vorgangs genauer als bisher dargestellt.

8.1.1. Die Unterscheidung verletzbarer und unverletzbarer Eigenschaften

In Kapitel 5 hatten wir im Zusammenhang mit verletzbaren und unverletzbaren Eigenschaften einer Datenstruktur bereits die Unterscheidung von harten und weichen Constraints getroffen.

Harte Constraints sind demgemäß Eigenschaften, deren Verletzung den Lernenden nicht möglich sein soll, während die Verletzung weicher Constraints zu irgendeiner Form von Markierung führt, aber nicht das Weiterarbeiten mit der VIDEA-Instanz verhindert.

Die harten Constraints können noch einmal unterteilt werden in Eigenschaften, für deren Verletzung keine Operation bzw. keine Struktur zur Verfügung steht, und in solche, deren Verletzung durch das Fehlschlagen einer Operation sofort durch das System rückgängig gemacht und mit einer Fehlermeldung geahndet wird.

Die erste Art harter Constraints wird vom Dozenten beim Design der Spezifikation meistens implizit getroffen, da die Datenstruktur in PROGRES konstruktiv definiert wird. Abb. 97 zeigt zwei Beispiele dieser Art von nicht verletzbaren Bedingungen in der AVL-Baum-Instanz. Links wird eine ins Leere gehende Kante gezeigt. Eine solche Situation kann nicht entstehen, da die Kanten als knotenwertige Attribute spezifiziert sind und ein fehlender Zielknoten somit eine fehlende Kante bedingt.

Ein zweites Beispiel für die erste Art harter Constraints ist die Binärbaum-Eigenschaft beim AVL-Baum, die dadurch gewährleistet wird, dass es laut PROGRES-Spezifikation nur jeweils höchstens eine *left*- und eine *right*-Kante geben kann. Denn würde z. B. mehr als eine *left*-Kante erlaubt werden, wäre jeder Bezug auf das Knotenattribut *left* mehrdeutig. Somit kann ein Knoten bei der gewählten Struktur der Spezifikation nie mehr als zwei Nachfolger haben.

Um hier weiche Constraints zu nutzen, könnte dafür gesorgt werden, dass z. B. mehr als eine *left*-Kante aus einem Knoten ausgehen kann. Das kann z. B. erreicht werden, indem statt des Attributs *left* ein Kantentyp *left* in einer Form wie

```
edge type left : AVLNode -> AVLNode;
```

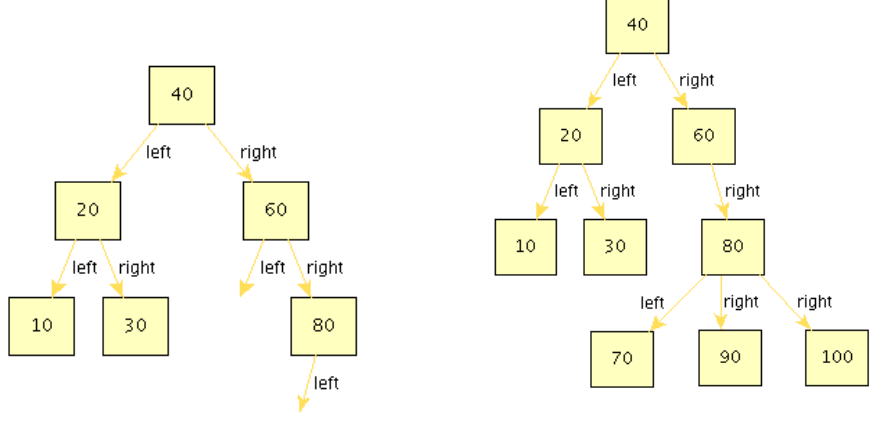


Abbildung 97: Strukturell nicht erlaubt: Links: Ins Leere gehende Kanten. Rechts: Drei Nachfolger im Binärbaum.

eingeführt wird. Analoges gilt für die *right*-Kante. Noch einen Schritt weiter führen würde die Manipulation, einen frei attributierbaren Kantentyp *Edge* einzuführen, der über einen neuen Knotentyp und die Nutzung des Edge-Node-Edge-Filters zu implementieren wäre, wie in Kapitel 6 bei der Beschreibung der Dijkstra- und der Kruskal-Instanz gezeigt wurde. Auf diese Art wären nicht nur beliebig viele *left*- und *right*-Kanten zwischen Knoten vom Typ *AVLNode* erlaubt, sondern beliebig viele beliebig bezeichnete Kanten.

Beide genannten Alternativen zu der hier vorgeschlagenen AVL-Instanz mit jeweils maximal einer ausgehenden *left*- und *right*-Kante pro Knoten würden die PROGRES-Spezifikation stark komplizieren, da nicht mehr mit den Attributwerten *left* und *right* gerechnet werden könnte, sondern zur Laufzeit bei jeder Operation passende Kanten im Graphen gesucht und Inkonsistenzen aufgedeckt werden müssten. Eine solche Vorgehensweise wäre also für die Implementierung einer Lernumgebung für AVL-Bäume aus unserer Sicht nicht zielführend.

Durch die Nutzung der hier sinnvollen harten Constraints der ersten Art für die Gewährleistung der Eigenschaft höchstens zweier Nachfolger eines Knotens wird allerdings nicht verhindert, dass die Baum-Eigenschaft an sich verletzt wird, denn es können mehrere Kanten in einen Knoten einlaufen. Um eine solche Situation zu verhindern, könnte man ein hartes Constraint der zweiten Art in die Operationen *setLeft* und *setRight* einführen, so dass sie fehlschlagen, wenn die Baumeigenschaft verletzt wird.

In unserer Implementierung der AVL-VIDEA-Instanz haben wir stattdessen wie gezeigt ein Attribut *noTree* eingeführt, das in der Art eines weichen Constraints verwendet wird, indem es „nur" zur farblichen Markierung der problematischen Stelle im Baum mit einer Fehlerfarbe führt. Außerdem hat hier der Layoutalgorithmus schnell ein Problem und wartet mit einer eigenen Fehlermeldung auf, die dem Nutzer zeigt, dass hier schnell etwas zu reparieren ist. Folgt er diesem Hinweis nicht, dann muss er auf ein Baum-Layout verzichten, so dass ein Wirrwarr von

Knoten entsteht. Die angesprochene Fehlermeldung gehört zum Layoutmanager und kann nicht vom Benutzer konfiguriert werden. Allerdings kann sie vom Dozenten ausgeschaltet werden, wenn er denkt, dass sie nicht hilfreich ist. Mit verletzter Baum-Eigenschaft kann nicht sinnvoll weitergearbeitet werden. Wird die fehlerhaft eingefügte Kante entfernt, lösen sich diese Probleme sofort auf, und der Baum wird wieder fehlerfrei dargestellt.

Weitere Beispiele weicher Constraints sind die mehrfach beschriebenen Farbschemata für die Sortierungs- und die Balancierungseigenschaft eines AVL-Baums. Der Ablauf ist hier immer der, dass zunächst die erwünschte Eigenschaft der Datenstruktur in Knoten- oder Kantenattributen codiert ist und im zweiten Schritt ein Farbschema unter Nutzung dieses Attributs oder dieser Attribute definiert wird. Bei der Erklärung der Logik von Farbschemata in VIDEA in Abschnitt 8.4 wird es zu sehen sein, wie mehrere Bedingungen zu einem Schema kombiniert werden können.

Nachdem nun die Grundlagen für das Verständnis der Erstellung von Fahrplänen und Lernpfaden in VIDEA geschaffen wurden, werden wir im kommenden Abschnitt beschreiben, wie der Dozent Fahrpläne erstellen und dabei gleichzeitig die Spezifikation der Datenstruktur verfeinern und konkretisieren kann. Dabei wird es sich zeigen, dass viele der auftretenden Fragestellungen ähnlich zu üblichen Problemen beim Software-Entwurf sind.

8.1.2. Erstellung von Fahrplänen und Lernpfaden

In Kapitel 5 wurden Fahrpläne als informelle Beschreibungen von erwünschten Bedienungsabläufen einer VIDEA-Instanz und Lernpfade als von der VIDEA-Instanz angebotene Folgen von Lernschritten, die aus Aktionen und Beschäftigung des Lernenden bestehen, eingeführt.

Wegen seiner hohen Interaktivität ist die Angabe von Fahrplänen besonders interessant für das *Interaktive Erforschen und Aufgaben-Lösen*. Wenn in einem solchen Fall ein bekanntes Thema der zugehörigen Lehrveranstaltung als eigenständiger Fahrplan einer VIDEA-Instanz mit konkretem Lernziel aufbereitet ist, kann man auch von einer *Übungsaufgabe* sprechen.

Beim Erstellen eines Fahrplans geht es zunächst darum, einen groben Handlungsablauf zu entwerfen. Soll zum Beispiel ein Fahrplan für eine VIDEA-Instanz mit dem Thema der aufsteigend sortierten, doppelt verketteten Ringliste erstellt werden, so überlegt sich der Dozent zunächst, welche wesentlichen Eigenschaften dieser Datenstruktur unterrichtet werden sollen. Das können zum Beispiel die aufsteigende Sortierung der Listenelemente, die vollständige und korrekte doppelte Verkettung, die Existenz von *first*- bzw. *last*-Kante auf das erste bzw. letzte Listenelement, die Geschlossenheit der Ringform etc. sein.
Im nächsten Schritt gibt der Dozent einen groben Handlungsablauf an, zum Beispiel:

1. Der Lernende beobachtet, wie eine beispielhafte Liste mithilfe von Standardkonstruktoren aufgebaut wird.
2. Es werden Fehler in die Liste eingebaut.
3. Die Fehler werden durch eine farbliche Markierung der fehlerhaften Stellen dargestellt.
4. Der Lernende repariert die Liste durch Behebung der einzelnen Fehler, wobei er jedes Mal geeignetes Feedback durch Fehlermeldungen oder neue Einfärbung bekommt.
5. Der Lernende fügt ein gegebenes, neues Element an der korrekten Stelle der Liste ein, so dass danach wieder alle Ansprüche an die Liste erfüllt sind.

6. Der Lernende führt schrittweise einen vorbereiteten Ablauf zum korrekten Einfügen eines neuen Elements aus und vergleicht diesen mit seinem vorher gewählten Ablauf.
7. Der Lernende überprüft sein Verständnis bestimmter Eigenschaften der Datenstruktur oder des Algorithmus, indem er vorbereitete Verständnistests startet und nach Möglichkeit mit korrekten Daten versorgt.

Dieser grobe Handlungsablauf ist die Basis für einen Fahrplan und wird im Folgenden verfeinert. Er kann bereits als Grundlage für benötigte Lernschritte der VIDEA-Instanz, für Details der Spezifikation der Datenstruktur und für eine Übungsaufgabe dienen, mithilfe der die entstehende VIDEA-Instanz im *Interaktiven Erforschen und Aufgaben-Lösen* genutzt wird.

Dazu muss der Handlungsablauf zunächst noch verfeinert werden. Zum Beispiel müssen für Schritt 1 Standardkonstruktoren festgelegt werden. Eine Möglichkeit, die wir für die VIDEA-Listen-Instanz gewählt haben, sind die Operationen *createListHead* zum Erzeugen des Listenkopfes und *addFirst* zum Einfügen eines neuen Listenelements am Anfang der Liste. Je nachdem, ob *addFirst* die korrekte Zeigerstruktur mit *first*-, *last*-, *previous*- und *next*-Kanten bewahrt, müssen zusätzliche Operationen in den Ablauf mit eingebunden werden oder nicht.

Im Schritt 2 ist die Frage, welche Fehler eingebaut werden sollen. Wir haben uns dafür entschieden, einige Kanten wegzulassen und einen Knoten falsch einzusortieren (siehe Kapitel 7).

Die in Schritt 3 des oben dargestellten Handlungsablaufs erwähnte Fehlermarkierung muss nun durch ein VIDEA-Farbschema ausgedrückt werden. Zum Beispiel soll ein Knoten in der Farbe Rot dargestellt werden, wenn er das Listenelement mit dem kleinsten Wert repräsentiert und keine *first*-Kante in ihn einmündet. Es empfiehlt sich, die Beschreibung des komplett ausformulierten Farbschemas später der Übungsaufgabe beizulegen, damit die Lernenden genau wissen, was die fehlerhafte Markierung bedeutet. Eine Alternative ist die Vorbereitung zusätzlicher Fehlertexte, die auf die Bedeutung hinweisen. Das kann dann sinnvoll sein, wenn es nicht zu viele Operationen gibt, die die fragliche Fehlersituation verursachen können.

Zum Reparieren der Liste in Schritt 4 und zum Einfügen eines neuen Elementes in Schritt 5 sind entsprechende Operationen notwendig, die es im Fall der Liste zumindest erlauben, neue Listenelemente und Kanten zu erzeugen und Kanten zu löschen. Hierbei ist zu untersuchen, welche besonderen Bedingungen auftreten können. Eventuell zeigt es sich, dass die ursprüngliche Menge unerwünschter und fehlerhafter Bedingungen nicht vollständig war und ergänzt werden muss. Werden fehlerhafte Aktionen ausgeführt, muss – wie immer in VIDEA – unterschieden werden zwischen strukturell nicht erlaubten Aktionen, Aktionen, deren Ausführung vom System mit einer Fehlermeldung verweigert wird, und Aktionen, die zu einer veränderten Einfärbung führen. Wenn z. B. die Operation *setNext* genutzt wird, um eine *next*-Kante zwischen zwei Knoten *n1* und *n2* zu erzeugen, ist es durch die Spezifikation von *setNext* gewährleistet, dass eine möglicherweise schon vorhandene von *n1* ausgehende *next*-Kante durch die neue *next*-Kante ersetzt wird; es können also keine mehrfachen ausgehenden *next*-Kanten entstehen. Soll aber zusätzlich verhindert werden, dass mehr als eine eingehende *next*-Kante in *n2* entsteht, muss dafür gesorgt werden, dass *setNext* in diesem Falle fehlschlägt. Zusätzlich muss dafür gesorgt werden, dass der Benutzer durch eine entsprechende Fehlermeldung auf den Fehler aufmerksam gemacht wird. Im Falle mehrerer eingehender *next*-Kanten in einen Knoten ist es aber – ähnlich wie bei einer unvollständigen Verkettung durch *next*- und *previous*-Kanten – sinnvoller, die Verletzung der Bedingung durch Fehlerfarben auszudrücken, da solche Konstellationen selbst

beim korrekten Reparieren einer Liste vorübergehend auftreten können. Obwohl diese Vorgehensweise sinnvoll ist, handelt es sich nicht um die einzig mögliche Lösung, sondern um eine willkürliche Entscheidung.

Um in Schritt 6 einen vordefinierten Ablauf interaktiv nutzen zu können, muss dieser in Form eines einfachen Algorithmus spezifiziert werden. Wir haben in Kapitel 6 einen solchen Ablauf zum Einfügen eines neuen Listenelements in eine aufsteigend sortierte Ringliste als Paar von Operationen *prepareInsert* und *insertStep* realisiert. Da auch bei Ausführung dieses vordefinierten Ablaufs temporär unerwünschte Situationen wie z. B. fehlende *previous*- oder *next*-Kanten auftreten können, ist es besonders wichtig, dass die hier genutzten Operationen, die zur Verletzung dieser Eigenschaften führen, nicht zurückgewiesen werden, sondern dass lediglich eine temporäre Umfärbung der Datenstruktur stattfindet.

Schritt 7 kann bei der Entwicklung einer VIDEA-Instanz für die doppelt verkettete Liste weggelassen werden, da die unmittelbar zu überprüfenden Eigenschaften der Datenstruktur trivial sind und der einfache Algorithmus nur einen beispielhaften Ablauf zeigt, der nicht an sich gelernt werden soll. Dagegen haben wir bei den VIDEA-Instanzen für die AVL-Bäume und für die Algorithmen von Dijkstra und Kruskal auf Graphen verschiedene Tests spezifiziert und eingesetzt, wie in den Kapiteln 6 und 7 gezeigt wurde.

Nachdem ein Fahrplan vorliegt, mit dessen Hilfe die grundsätzliche Spezifikation für die Datenstruktur in unserem Beispiel bereits verfeinert werden konnte, kann es sinnvoll sein, besonders interaktive Teilabläufe bis auf die Ebene der Lernschritte hin zu analysieren. Ein Beispiel hierfür zeigt Abb. 98, wo ein Teil eines möglichen Lernpfades dargestellt wird, der sich aus dem bisher dargestellten Fahrplan ergibt. Abb. 98 stellt Lernschritte in einem Aktivitätsdiagramm dar. Die Aktionen des VIDEA-Systems und der innere Lernschritt (oder Bewusstseins-Schritt) des Lernenden sind als parallele Aktivitäten modelliert.

In Kapitel 5 (auf Seite 88) wurde ein einzelner Lernschritt definiert als ein Paar, bestehend aus einer atomaren Nutzeraktion und der darauffolgenden Beschäftigung des Lernenden bis zum Beginn des nächsten Lernschrittes. Der Einfachheit halber sind in der „Beschäftigung des Lernenden" innere Lernschritte und die Reaktionen des VIDEA-Systems auf die vorhergehende Aktion zusammengefasst.

Nach dem Aufbau der fehlerhaften Liste kommt der Lernende an einen Punkt, wo er die Liste reparieren muss. Nach dem Beobachten sind deshalb mehrere Lernschritte möglich, abhängig davon, welche Aktion der Lernende als nächste ausführt. Falls er zunächst die in der Liste fehlende *last*-Kante einfügen will, ist der nächste Lernschritt einer der beiden in Abb. 98 unten dargestellten. Entweder wird die *last*-Kante korrekt oder falsch platziert. In beiden Fällen soll das System geeignet reagieren. Im Falle des korrekten Einfügens der Kante könnte die Kante ohne weitere Reaktion des Systems eingefügt werden oder eine Meldung „Gut gemacht!" o.Ä. ausgegeben werden. In Abb. 98 wurde die erste Variante gewählt. Falls die Kante an einer falschen Stelle eingefügt wird, könnte eine Fehlermeldung ausgegeben oder die fehlerhafte Stelle gefärbt werden. Hier wurde die zweite Variante gewählt.

Wenn auf diese Weise an zunächst zweifelhaften Stellen des Gesamtablaufs verfahren wird, konkretisieren sich der Fahrplan und die Spezifikation der Datenstruktur immer mehr, bis eine ablauffähige Version der zu erstellenden VIDEA-Instanz zur Verfügung steht. Außerdem kann der Dozent seinen Fahrplan unmittelbar als Übungsaufgabe nutzen, wenn noch evtl. notwendige Hinführungen zum Thema und für das Verständnis der Lernenden notwendige Teile der

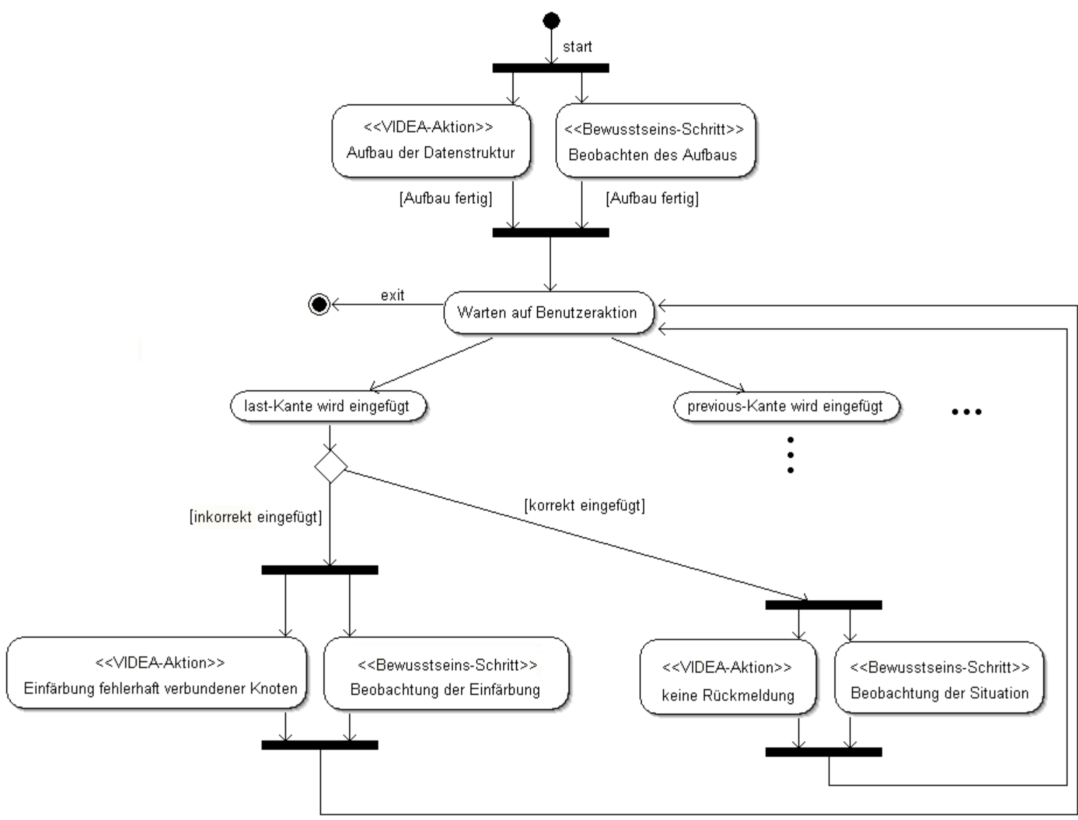


Abbildung 98: Drei Lernschritte der im Text beschriebenen, einfachen VIDEA-Listen-Instanz

Spezifikation hinzugefügt werden.

In einem sehr fordernden Szenario, wie dem in Kapitel 5 erwähnten Szenario *Spezifikation, Generierung und Test eigener Instanzen der Visualisierungsumgebung von Seiten der Lernenden* könnte es auch vorgesehen werden, dass die Lernenden eigene Spezifikationen für eine Datenstruktur erstellen. Dann würden sie die Rolle des Dozenten übernehmen und müssten mit PROGRES vertraut gemacht werden, was wohl nur für spezielle Lehrveranstaltungen sinnvoll ist.

In diesem Abschnitt wurde informell am Beispiel der doppelt verketteten, sortierten Ringliste gezeigt, wie die Erstellung von Fahrplänen und Lernpfaden in VIDEA abläuft. Zwei Beispiele von in der Praxis erprobten Fahrplänen für die VIDEA-AVL-Instanz und die VIDEA-Dijkstra-Instanz werden wir zusammen mit den gewonnenen Erkenntnissen in Kapitel 9 darstellen. Im den kommenden Abschnitten werden wir auf den weiteren Entscheidungsspielraum des Dozenten bei der Verfeinerung der Spezifikation eingehen.

8.1.3. Detaillierungsgrade bei der Algorithmenanimation

In den bisherigen Abschnitten wurden Themen behandelt, die bei jeder VIDEA-Instanz eine Rolle spielen. Wenn das Hauptaugenmerk einer VIDEA-Instanz auf der Lehre eines komplexen Algorithmus liegen soll, dann kommt zu den Datenstruktur-bezogenen Themen noch die Aufteilung des Algorithmus in Schritte hinzu, wie sie in Kapitel 5 eingeführt und in Kapitel 6 an mehreren Beispielen exemplarisch vorgeführt wurde.

Ein Sachverhalt, auf den bisher in diesem Zusammenhang nicht im Detail eingegangen wurde, ist die Wahlfreiheit des Dozenten bei der Granularität der entstehenden Schritte des Algorithmus. Bei der Spezifikation des Dijkstra-Algorithmus sind z. B. folgende drei Detaillierungsgrade möglich:

- Die Implementierung des Dijkstra-Algorithmus direkt in der PROGRES-Sprache; das würde bedeuten, dass es eine Produktion *dijkstraComplete* gäbe, in der der Algorithmus abläuft und an deren Ende der Ergebnisgraph mit allen berechneten kürzesten Pfaden und Entfernungen angezeigt werden würde. Selbst wenn es technisch möglich wäre, die Anzeige zwischen den Einzelschritten einer PROGRES-Programmeinheit in UPGRADE2 zu synchronisieren (was zum Entwicklungs-Zeitpunkt nicht möglich war) und somit einen flüssigen Gesamt-Ablauf zu sehen, würde nur ein passiver Film ohne jede Möglichkeit der Interaktion und des aktiven Lernens ablaufen.

- Das Aufbrechen der Hauptschleife des Algorithmus in Schritte, so dass in unserem Beispiel etwa eine Produktion *dijkstraStep* entstünde, deren sukzessiver Aufruf Schritt für Schritt den Algorithmus ausführen würde, so dass nach jedem Schritt eine Situation dargestellt ist, die für Experimente der Lernenden oder neue Handlungs-Stränge oder Verständnistests genutzt werden kann. In unserem Beispiel entspricht bei diesem mittleren Detaillierungsgrad dem *dijkstraStep* das Weiterlaufen der Dijkstra-Welle vom aktuellen Knoten U zu allen unmittelbar, also durch Folgen *einer* Kante erreichbaren Knoten V (siehe Kapitel 6, Seite 134 für den Algorithmus).

- Das Aufbrechen der Haupt- *und* Unter-Schleife in einen Fein-Schritt *dijkstraMiniStep*, der entsprechend fein-granularer als *dijkstraStep* ist und der den Algorithmus zwar nach und nach durchführt, aber dafür wesentlich häufiger ausgeführt werden muss. In unserem Beispiel könnte dem *dijkstraMiniStep* das Folgen *einer* Kante vom aktuellen Knoten U zu einem seiner unmittelbar benachbarten Knoten V entsprechen.

Wie in Kapitel 6 beschrieben, haben wir uns in der Dijkstra-VIDEA-Instanz für die mittlere beschriebene Lösung entschieden, weil sie uns für praktikable Zwecke am sinnvollsten erschien. Eine weitere sinnvolle Möglichkeit könnte eine Kombination der mittleren und der letzten Lösung sein – z. B. durch Implementierung zweier Produktionen *dijkstraStep und dijkstraMiniStep*. In der praktischen Anwendung würde dann laut Aufgabenstellung z. B. ein bestimmter, vorher festgelegter Haupt-Schritt des Algorithmus durch Experimente mit den Fein-Schritten erforscht werden, während der Gesamtalgorithmus weiterhin mit dem gröberen *dijkstraStep* erforscht werden würde.

An diesem Punkt des Designs einer VIDEA-Instanz steht fest,

- wie die Datenstruktur visuell spezifiziert werden soll,
- damit auch, durch welche Graph-Konstrukte sie in PROGRES dargestellt werden soll
- und wie fein die darstellbaren Schritte des Algorithmus sein sollen.

Nun ergibt sich die Frage, mit welchen Konstrukten die gewünschte Funktionalität modelliert werden soll. Im nächsten Abschnitt werden einige Anregungen zur Beantwortung dieser Fragestellung gegeben.

8.1.4. Auswahl der Konstrukte für Hilfsdatenstrukturen

Wie in Kapitel 6 zu sehen war, sind visuelle Spezifikationen oft eine ziemlich natürliche, naheliegende Beschreibung der Datenstruktur und des Algorithmus, so dass es, besonders für den mit UML vertrauten Dozenten, kein großes Problem sein sollte, die eigenen Vorstellungen in Graphtransformationen in PROGRES auszudrücken. Hier ist also wohl keine Hilfestellung allgemeiner Art notwendig.

Eine Fragestellung, auf die wir trotzdem kurz eingehen wollen, ist die Wahl der PROGRES-Konstrukte für eventuell notwendige Hilfsdatenstrukturen wie etwa beim Kruskal-Algorithmus. Dort gibt es in den gängigen Implementierungen zwei Hilfsdatenstrukturen, nämlich die Prioritäts-Warteschlange und die Partitionierung mit 'union' / 'find'-Funktionalität. Zwei Arten der Umsetzung in PROGRES bieten sich an:

Die erste Möglichkeit ist die Verwendung eigener, definierter Knotentypen mit eigens programmierten Zugriffs-Operationen. Das heißt im Fall der Prioritäts-Warteschlange, sie z. B. als explizite Liste eines neuen PROGRES-Knotentyps zu implementieren und ihre Umsortierung konsequenterweise während des Ablaufes des Algorithmus jedesmal zu visualisieren. Da gleichzeitig jeder Knoten im Umfeld der eigentlichen Datenstruktur sichtbar sein sollte, ist eine übersichtliche Visualisierung auf diese Weise schwer vorzustellen. Eine theoretisch mögliche Verwendung mehrerer Sichten existiert in der jetzigen Version von VIDEA nicht, ist aber als Erweiterung denkbar.

Eine andere Möglichkeit, für die wir uns im Rahmen der implementierten VIDEA-Instanzen überwiegend entschieden haben, ist die Verwendung zusätzlicher Attribute auf existierenden Knoten oder Kanten. Beim Dijkstra-Algorithmus zur kürzesten Wegesuche haben wir zum Beispiel gesehen, dass die Umfärbung derjenigen Knoten, die das Ziel der auslaufenden Welle durch den Graphen sind, in *recolorVs* durch eine gemischte Nutzung des bereits vorhandenen Attributs *color* und des als Datenspeicher genutzten Attributs *isOneOfCurrentVs* passiert. Abb. 99 zeigt nochmal die schon in Kapitel 6 gezeigte Implementierung von *recolorVs*.

Nachdem der Dozent die Spezifikation der Datenstruktur inklusive der wesentlichen Operationen und der Schritte des Algorithmus weitgehend umgesetzt hat, können abhängig von der zu unterrichtenden Datenstruktur und dem erstellten Fahrplan Verständnistests vorbereitet werden. Im nächsten Abschnitt beschreiben wir die Vorgehensweise.

8.1.5. Tests

Die im Fahrplan der VIDEA-Instanz vorgesehenen Verständnistests werden in VIDEA üblicherweise durch PROGRES-Tests implementiert. Beispiele waren die in Kapitel 6 gezeigten

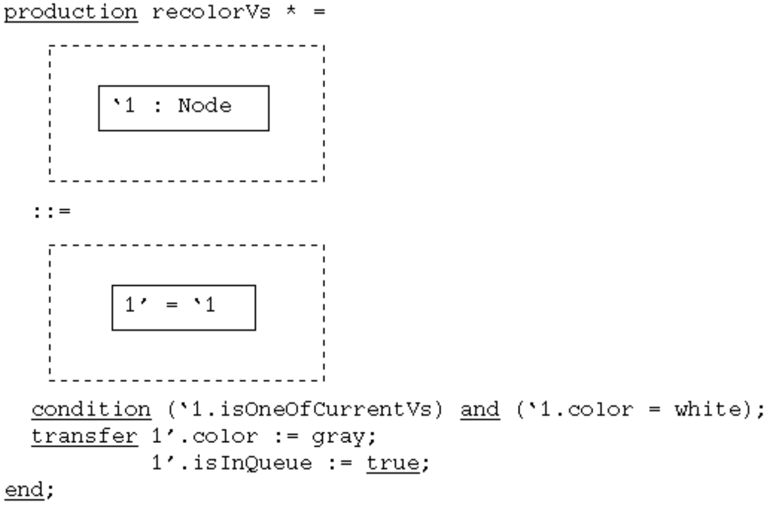


Abbildung 99: Wiederholung von *recolorVs*

Tests *checkBalance*, *checkHeight*, *isNextConsideredEdgeTaken*, *testNextConsideredEdge*, *testCurrentU* und *testNextU*. Die Struktur war dabei immer die gleiche: Es gibt ein richtiges Ergebnis, das ein Zahlenwert, ein boolescher Wert, ein Knoten oder eine Kante sein kann und das im Laufzeit-Graphen verborgen vorliegt als nicht angezeigtes Attribut oder Berechnungsvorschrift. Der Test prüft dann lediglich ab, ob der Eingabewert dem erwarteten Wert entspricht.

Die Schritte für den Dozenten sind also beim Erstellen neuer Tests:

- Wahl einer Eigenschaft der zu lehrenden Datenstruktur oder des zu lehrenden Algorithmus, deren Verständnis auf Seiten der Lernenden überprüft werden soll.
- Quantifizieren dieser Abhängigkeit in Form eines messbaren Wertes. Das kann z. B. ein ganzzahliger Wert, ein boolescher Wert, ein Knotentyp oder ein Kantentyp sein.
- Erstellen einer Berechnungsvorschrift für diesen Wert in Abhängigkeit vorhandener Knoten- oder Kantenattribute.
- Erstellung eines neuen Attributs, dessen Wert den zu überprüfenden Wert repräsentiert, falls keine Kombination bestehender Attribute sinnvoll genutzt werden kann.
- Schreiben eines Tests, der einen Parameter erwartet, dessen Wert mit dem zu berechnenden oder als Attributwert vorliegenden Wert verglichen wird. Das Ergebnis des Vergleichs ist die Bedingung für das Bestehen des Tests.

Um den Unterschied an zwei bereits gezeigten Beispielen zu zeigen, sind *checkBalance* und *textNextU* aus Kapitel 6 in Abb. 100 gegenübergestellt: In *checkBalance* wird der angegebene Balancefaktor mit dem Knotenattribut *balance* verglichen, also ein bereits vorhandenes Attribut genutzt. In *textNextU* wird eine Berechnungsvorschrift angegeben, die mithilfe der Attribute *distance* und *isInQueue* die geforderte Antwort auf die Frage berechnet, ob der übergebene Knoten nächstes ’U’ wird.

```
test InMenu_Exercise_checkBalance( node : AVLNode ; balanceFactor : integer)
=

    `1 = node

  condition `1.balance = balanceFactor;

end;
```

```
test InMenu_Exercise_testNextU( n : Node) =

        `1 : Node

          valid ((self.distance < n.distance) and (self.isInQueue))

  condition n.isInQueue;

end;
```

Abbildung 100: Oben: Wiederholung von *checkBalance*. Unten: Wiederholung von *textNextU*.

8.1.6. Vorbereiten der ferngesteuerten Algorithmenvisualisierung

Die Fähigkeit von VIDEA, ferngesteuerte Animationen anzubieten, wurde in Kapitel 7 aus der Sicht des Benutzers einer VIDEA-Instanz geschildert. Natürlich muss sich der Dozent auch für dieses Szenario zumindest einen grundlegenden Fahrplan überlegen.

Im Beispiel der beschriebenen AVL-Baum-Instanz kann ein grober Fahrplan zum Beispiel so aussehen:

Ziel: Es soll eine Java-Klasse `AVLNode` entwickelt werden, die AVL-Bäume modelliert und ausreichend Methoden beinhaltet, um einfache AVL-Bäume aufbauen zu können und Rotationsoperationen auf ihnen durchzuführen.

1. Der Lernende nutzt eine vorgegebene abstrakte Klasse `Node`, von der `AVLNode` erbt, und in der einfache Operationen wie *setLeft* und *setRight* schon vorgegeben sind.
2. Der Lernende öffnet sein Java-Programm in einer handelsüblichen Entwicklungsumgebung, die schrittweises Debugging anbietet (z. B. Together Control Center), und startet die VIDEA-Instanz.
3. Er führt sein Java-Programm schrittweise aus und beobachtet in der VIDEA-Instanz die animierten Veränderungen, die sein Programm an der Datenstruktur ausführt.
4. Falls sich Fehler zeigen, korrigiert der Lernende seinen Code und fährt mit Schritt 3 fort.

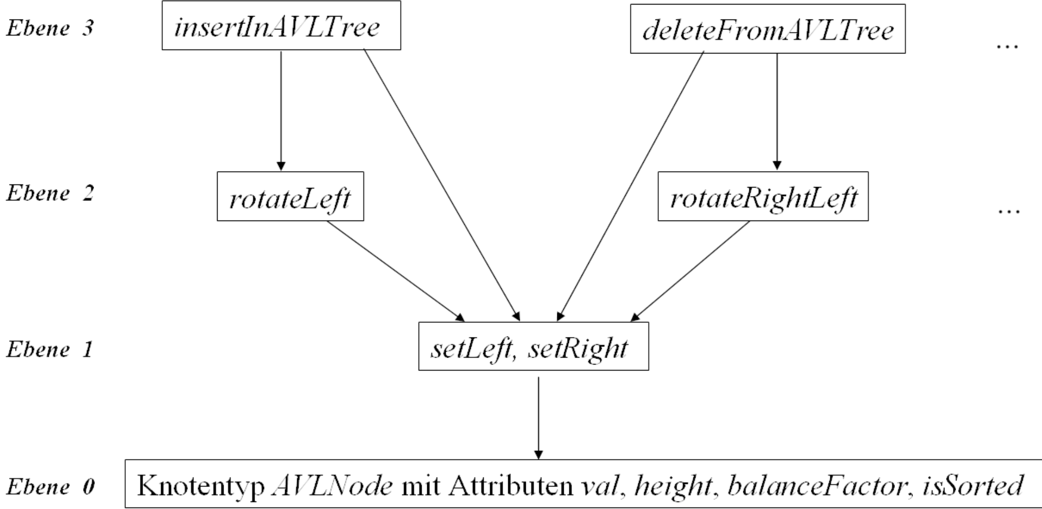


Abbildung 101: Aufrufhierarchie-Ebenen

In diesem Fahrplan ist wieder eine Designentscheidung des Dozenten enthalten, die im Folgenden explizit dargestellt wird: Beim Spezifizieren und Programmieren in PROGRES können sich – wie in textuellen Programmiersprachen auch – Programmeinheiten auf andere Programmeinheiten abstützen, z. B. Transaktionen auf Produktionen oder auf andere Transaktionen. Wir haben bei der Vorstellung der Spezifikationen in Kapitel 6 gesehen, dass die Programmeinheiten auf oberster Ebene auf Menüs verteilt werden müssen. Zusätzlich gibt es Programmeinheiten, die von anderen Programmeinheiten aufgerufen werden und reine Hilfs-Programmeinheiten sind. Ein Beispiel hierfür in der AVL-VIDEA-Instanz ist die Produktion *Primitive_getRoot* (siehe Seite 119), die nicht für den Benutzer sichtbar ist, aber von *InMenu_BinTree_insert* aufgerufen wird.

Eine weitere Entscheidung des Dozenten ist es nun, ob nur die Top-Level-Transaktionen bzw. -Produktionen für den Nutzer einer VIDEA-Instanz in einem bestimmten Szenario, z. B. in interaktiven Übungen, sichtbar sein sollen, oder auch die eine oder andere Hilfs-Transaktion oder -Produktion. Ein Beispiel aus unserer AVL-Instanz ist die Produktion *InMenu_Exercise_setLeft*, die von komplexeren Transaktionen wie *BinTree_insertInTree* sinnvoll genutzt wird, aber auch – wie der Name schon sagt – im Menü `Exercise` dem Nutzer direkt zur Verfügung gestellt wird. Der Dozent kann gemäß der Aufruf-Hierarchie der zur Verfügung stehenden PROGRES-Programmeinheiten mehrere Ebenen unterscheiden. Im Beispiel der AVL-Bäume kann man die Ebenen der reinen Knoten mit ihren Attributen, die Ebene der einfachen Operationen wie *setLeft* und *setRight*, die Ebene der Rotationsoperationen wie *rotateLeft* und *rotateRightLeft* und die Ebene der komplexesten Operationen auf AVL-Bäumen wie *insertInAVLTree* oder *deleteFromAVLTree* unterscheiden. Abb. 101 zeigt die Abhängigkeiten graphisch. Dabei stellen die Pfeile eine gerichtete „... stützt sich ab auf ...“-Relation dar.

Um die Lernenden eine Operation auf Ebene *n* implementieren zu lassen, ist es sinnvoll, die

Operationen bis einschließlich Ebene *n-1* bereits vorzugeben, damit die in der Java-Umgebung auszuführenden Debugging-Schritte nicht zu fein-granular werden.

In dem oben angegebenen Fahrplan wurden *setLeft* und *setRight* vorgegeben. Darauf aufbauend sollten Rotationsoperationen entwickelt werden. Die Lernenden entwickeln also in Abb. 101 auf Ebene 2. Genausogut hätten die Rotationsoperationen zur Verfügung gestellt und z. B. eine Implementierung für *insertInAVLTree* gefordert werden können.

Wenn der Dozent den Fahrplan systematisch durchgeht wie in Abschnitt 5.2.3 beschrieben, zeigen sich folgende Anforderungen an die Vorbereitung dieses Szenarios:

- Die zur Verfügung gestellte Klasse `Node.java` implementiert die zugrundegelegten Operationen *setLeft*, *setRight* etc. je nach Zielsetzung eventuell voll funktional. Zumindest müssen aber in jeder dieser Methoden geeignete Aufrufe an den `MusoftClient` stecken, die dann über CORBA an die VIDEA-Instanz weitergeleitet werden.
- Die VIDEA-Instanz muss deshalb für jede dieser grundsätzlichen Operationen eine Spezifikation aufweisen, die vom `MusoftClient` aufgerufen werden kann, z. B. *setLeftByNodeVal* und *setRightByNodeVal*.

Abb. 102 veranschaulicht das Szenario der ferngesteuerten Algorithmenanimation durch eine Übersichts-Graphik, in der die Hauptkomponenten von VIDEA und ihre Verbindungen untereinander und nach außen gezeigt werden. Man sieht, dass eine Art Rückkopplung entsteht, indem der Lernende bzw. Student die Bildschirmausgabe der ferngesteuerten Animation seines eigenen Programms sieht, davon beeinflusst evtl. Fehler in seinem Java-Code erkennt und korrigiert. Das korrigierte Programm nimmt dann beim nächsten Lauf wahrscheinlich im Hinblick auf die Aufgabenstellung bessere Manipulationen an der Datenstruktur vor, indem wieder Methoden des `MusoftClient` genutzt werden. Beispiele dafür sind beim AVL-Baum *setLeft* und *setRight*. Diese Methoden-Aufrufe werden vom `MusoftClient` über CORBA als Aufrufe an den `MusoftServer` und von da an den VIDEA-Kern weitergegeben, der über UPGRADE2 PROGRES-Transaktionen aufruft. Dadurch werden am Laufzeit-Graphen in der Graphdatenbank Änderungen verursacht. In Abb. 102 ist diese im PROGRES-Kasten enthalten. Die erwähnten Änderungen führen zu Ereignissen, die sich wiederum in der Visualisierung im Graphbrowser niederschlagen.

Nun sieht der Lernende das neue Verhalten seiner Datenstruktur und kann wieder entsprechend darauf reagieren. Natürlich wirkt er zusätzlich zur Anpassung des Programms auch über die übliche interaktive Schnittstelle auf VIDEA ein, indem aus dem Menü des VIDEA-Graphbrowsers Transaktionen, Produktionen und Tests ausgeführt oder Knoten oder Kanten selektiert oder verschoben werden. Diese zweite Rückkopplungs-Schleife ist aber unabhängig von der ferngesteuerten Algorithmenvisualisierung immer vorhanden und deshalb in Abb. 102 nicht explizit eingezeichnet.

Die wesentliche Nutzung dieses Szenarios besteht aber für den Lernenden darin, das eigene Programm nicht einfach durchlaufen, sondern mithilfe eines üblichen Debuggers schrittweise ausführen zu lassen. Er sieht dann die Aktionen seines Programms, da diese VIDEA-Aktionen auslösen, im VIDEA-Graphbrowser ablaufen, und erlebt somit ein unmittelbares graphisches Feedback auf die Schritte seines Programms, das wir als ferngesteuerte Algorithmenanimation oder als einfaches visuelles Debugging bezeichnen. Dabei ist es zusätzlich möglich, bei Bedarf die ferngesteuerte Ausführung eines eigenen Programms beliebig mit interaktiven Aufrufen von

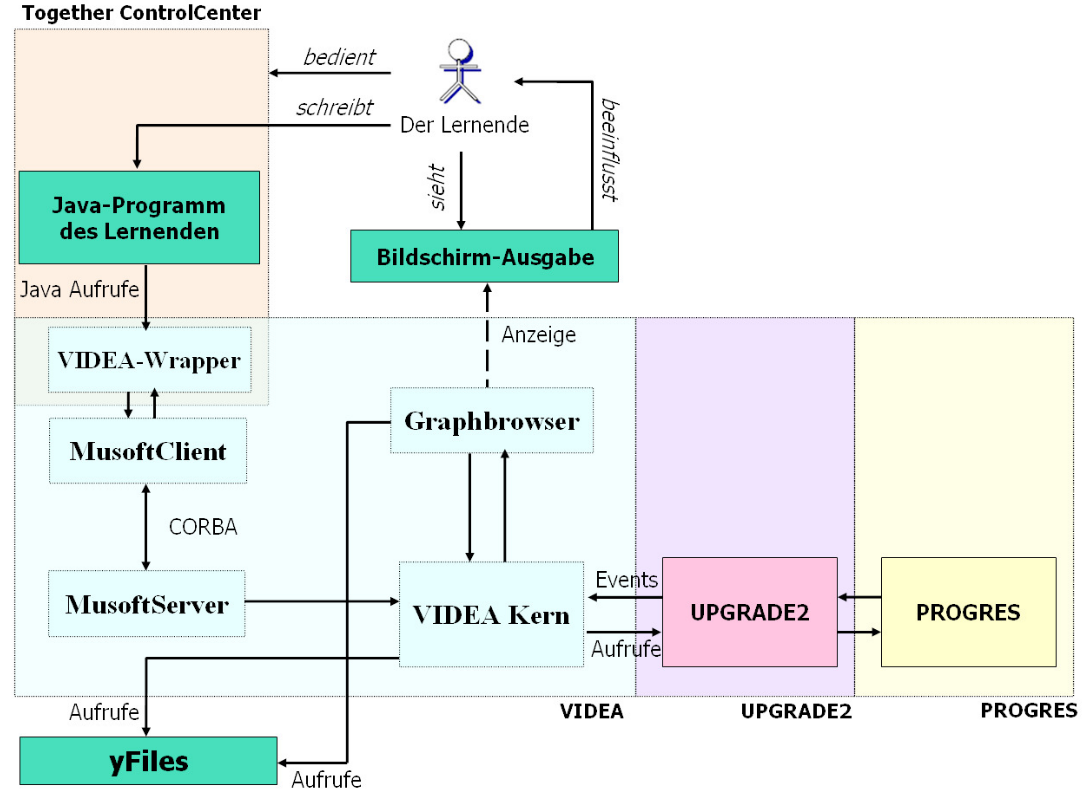


Abbildung 102: VIDEAs CORBA-Architektur

Aktionen der VIDEA-Instanz zu mischen und somit Teile des Programms, die noch fehlen, per Hand zu simulieren.

8.2. Generierung und grundsätzlicher Aufbau einer VIDEA-Instanz

Nachdem die PROGRES-Spezifikation erstellt ist, kann die Transformation in die UPGRADE2- und Java-Welt erfolgen. Zunächst wird in der PROGRES-Entwicklungsumgebung aus der Spezifikation ausführbarer Code generiert und dann mithilfe eines Skripts VIDEA-Funktionalität hinzugefügt und für eine lauffähige Instanz vorkonfiguriert.

Im Einzelnen wird der Generierungs-Prozess einer VIDEA-Instanz über die Generierung zuerst des C-Codes, dann des Java-Codes und schließlich zusätzlicher Komponenten gesteuert. Da VIDEA die Möglichkeit des ’rapid prototyping’ von UPGRADE2 nutzt, steht nach der Generierung bereits ein lauffähiger Graphbrowser zur Verfügung. Nach dessen Start können einfache Versuche mithilfe der generierten Menüs im Hinblick auf die gewünschte Bedienung und Funktionalität gemacht werden.

Mit den an dieser Stelle vorliegenden Informationen über VIDEA kann der Generierungs-Prozess in Abb. 103 genauer als bisher dargestellt werden. Im Kasten links sind alle wesentlichen Konfigurationsdateien angedeutet, die der Dozent braucht, um die VIDEA-Instanz nach dem

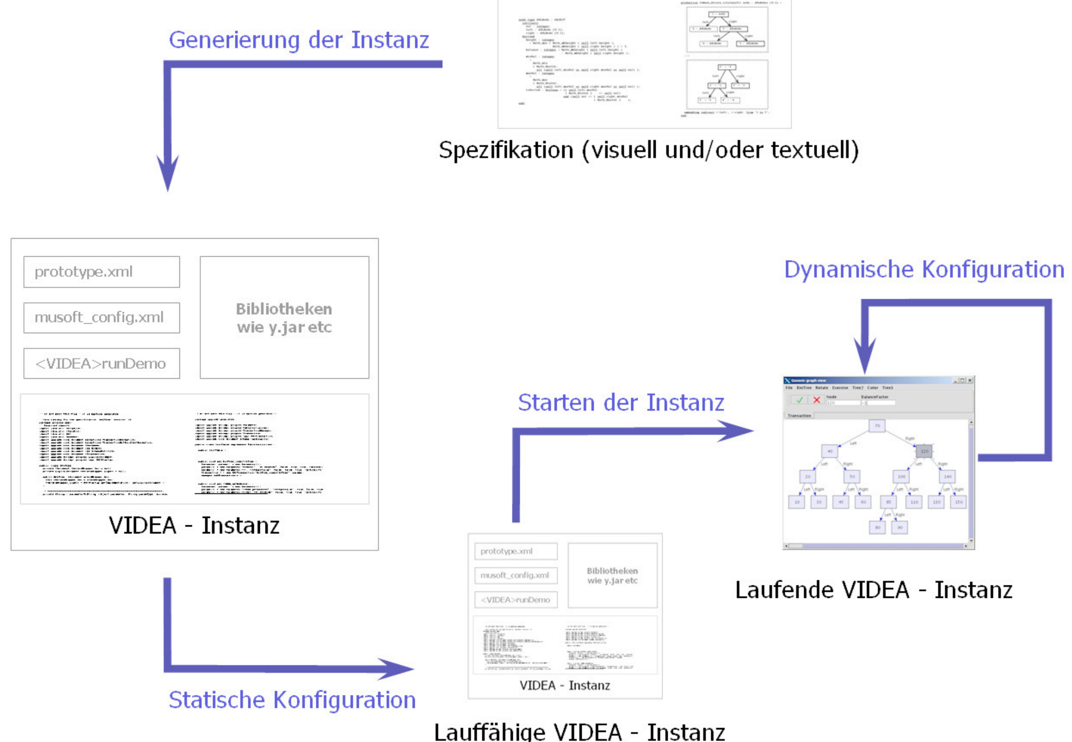


Abbildung 103: Die Generierung einer VIDEA-Instanz, genauer als in Abb. 31

Generierungs-Schritt sinnvoll konfigurieren zu können:

- Eine Konfigurationsdatei zur Steuerung der automatischen Erzeugung von Menüs aus Mengen von PROGRES-Programmeinheiten inklusive der Bindung von Aktionen an Menüpunkte (`prototype.xml`).
- Eine Konfigurationsdatei zur Vorbelegung aller wesentlichen VIDEA-Einstellungen mit Standardwerten für die entsprechende VIDEA-Instanz (`musoft_config.xml`).
- Eine Skript-Datei zur Steuerung automatischer Abläufe (namens `<VIDEA-Instanz>runDemo`, z. B. `avlRunDemo`).

Außerdem enthält die generierte VIDEA-Instanz wie im linken Kasten in Abb. 103 ebenso angedeutet

- Java-Klassen für die Grundfunktionalität des Prototypen, Schnittstellen zu den aus der PROGRES-Spezifikation generierten Programmeinheiten und den VIDEA-typischen Erweiterungen, die als Quellcode-Listings im unteren Bereich des Kastens „VIDEA - Instanz“ angedeutet sind und
- Bibliotheken, die u. a. für die Kommunikation mit der PROGRES-Laufzeitumgebung und den yFiles-Funktionen genutzt werden.

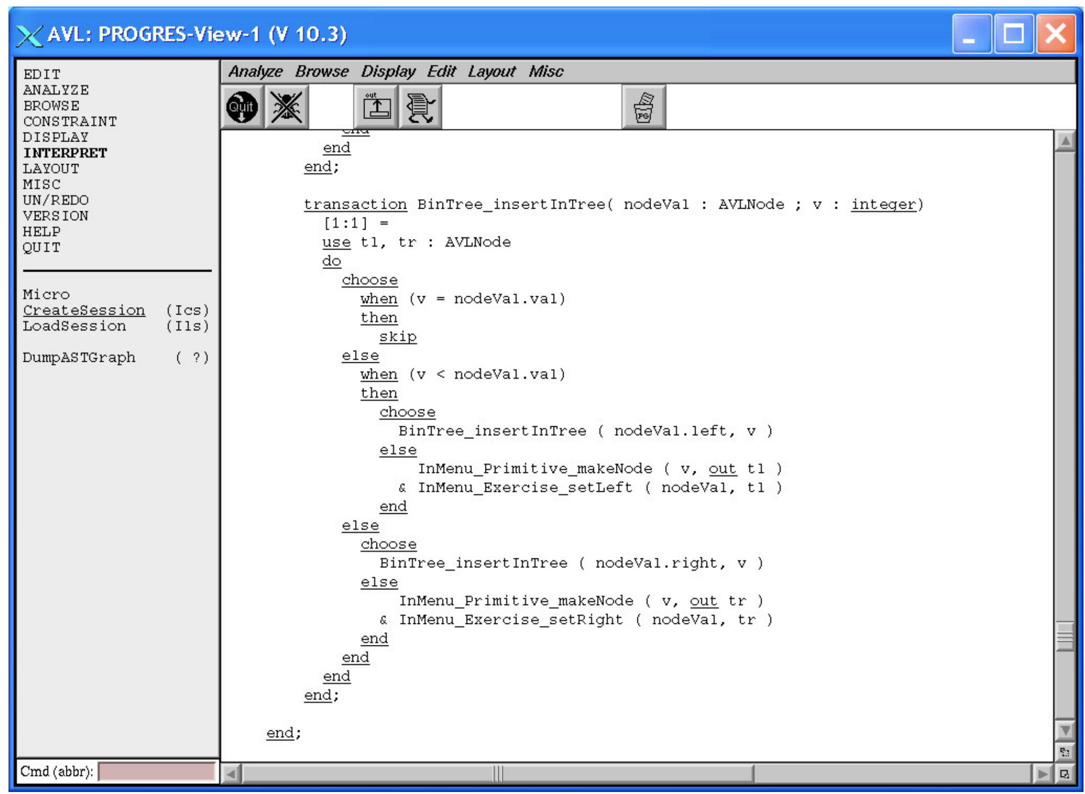


Abbildung 104: Die PROGRES-Entwicklungsumgebung, hier aus dem VIDEA-Generierungsskript gestartet

Es ist möglich, das Generierungsskript von VIDEA so anzupassen, dass nach dem Generierungsschritt grundsätzliche Einstellungen für die jeweilige Datenstruktur bereits korrekt voreingestellt sind. Zum Beispiel kann es sinnvoll sein, das zu verwendete Layout und Menüeinträge für den Start der VIDEA-Manager und wesentlicher Operationen geeignet vorzubelegen. In diesem Fall können nach der Generierung erste Tests durchgeführt werden, ohne die Konfigurationsdateien anpassen zu müssen.

Für die Konfiguration selbst sind hauptsächlich die erwähnten Konfigurationsdateien und die Skript-Datei wesentlich, zu denen in den folgenden Abschnitten alle wichtigen Informationen gegeben werden. Im Falle der Konfiguration eines neuen, d. h. noch nicht in VIDEA implementierten Layoutalgorithmus muss eine der Quelldateien von VIDEA angepasst werden, und zwar der `YFilesLayoutManager`, der sich unter den in obiger Liste erwähnten Java-Klassen für die Grundfunktionalität befindet.

Der für den Nutzer des Generierungs-Shellskripts offensichtlichste Schritt ist der Start der PROGRES-Entwicklungsumgebung, um die Generierung des C-Codes aus der visuellen Spezifikation anzustoßen. Abb. 104 zeigt einen Bildschirmausschnitt der PROGRES-Entwicklungsumgebung mit geöffneter Spezifikation für die VIDEA-AVL-Instanz. In dieser Situation wird zunächst C-Code generiert, indem der Benutzer bei geöffneter PROGRES-Spezifikation über die

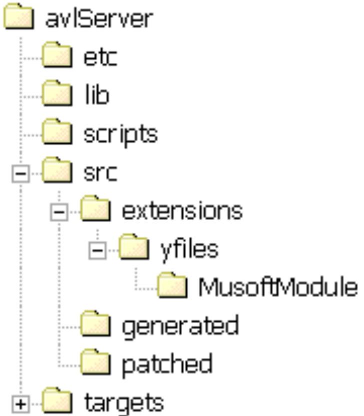


Abbildung 105: Die Verzeichnisstruktur eines Nutzers, der VIDEA-Instanzen generiert

PROGRES-Toolbar den Punkt `INTERPRET -> CreateSession -> Generate -> CCode` auswählt, woraufhin die Generierung in das neue Verzeichnis der VIDEA-Instanz erfolgt. In Abb. 104 wurde bereits `INTERPRET` links oben gewählt und deswegen `CreateSession` links unten sichtbar gemacht. Danach genügt es, PROGRES wieder zu schließen. Der Rest der Aktivitäten beim Generieren einer VIDEA-Instanz läuft im Hintergrund ab.

Nach der Ausführung der eben dargestellten Schritte liegt die in Abb. 105 gezeigte Verzeichnis-Struktur vor, wobei das Hauptverzeichnis gleichzeitig das vom Generierungsskript erzeugte Verzeichnis ist, dessen Name als erster Parameter dem Shellskript übergeben wurde (hier mit Namen `avlServer`). Diese Namensgebung rührt daher, dass es im Falle der Nutzung des visuellen Debuggings auch ein Verzeichnis `avlClient` gibt, das u. a. das Programm der Lernenden enthält, Details hierzu folgen am Ende dieses Kapitels.

Das VIDEA-Skript zum Erzeugen von VIDEA-Instanzen befindet sich natürlich nicht in der Verzeichnisstruktur einer VIDEA-Instanz selbst, sondern eine Ebene höher, typischerweise im Heimat-Verzeichnis eines speziell eingerichteten VIDEA-Benutzers, unter dem auch die VIDEA-Instanz läuft.

Im Hauptverzeichnis des Prototypen liegen u. a. das Start-Skript und das Skript zur Übersetzung des Prototypen. Da eine solche Übersetzung nur notwendig ist, wenn der Quellcode von VIDEA geändert worden ist, gehen wir an dieser Stelle nicht weiter auf dieses Thema ein und verweisen auf die VIDEA-Dokumentation [VID05].

Die Konfigurationsdateien `musoft_config.xml` und `prototype.xml` und das VIDEA-RunDemo-Skript befinden sich im Unterverzeichnis `etc`. Die Java-Quelldateien der für VIDEA neu entwickelten Klassen liegen im Verzeichnis `src/extensions/yfiles`, während sich die von PROGRES und UPGRADE2 generierten Dateien für die Implementierung der Schnittstelle zu den PROGRES-Unterprogrammen in `src/generated` und `src/patched` befinden. Das Unterverzeichnis `targets` enthält die kompilierten Versionen (hauptsächlich `class`-Dateien) der oben erwähnten Quelldateien. In `lib` liegen die für den jeweiligen UPGRADE2-Prototypen speziellen Java-Archive, im Moment nur `y.jar`. Das Unterverzeichnis `scripts` wird im Rahmen von VIDEA nicht genutzt.

Nach der Generierung einer VIDEA-Instanz kann diese prinzipiell sofort gestartet werden. Um sie aber sinnvoll einsetzen zu können, sind bestimmte Eigenschaften dieser Instanz zu konfigurieren. Dies kann statisch durch Anpassung von Konfigurationsdateien oder – teilweise – dynamisch zur Laufzeit geschehen. Im nächsten Abschnitt beschreiben wir zunächst die statische Konfiguration.

8.3. Statische Konfiguration

Die statische Konfiguration einer VIDEA-Instanz bezieht sich auf die Möglichkeiten der Einflussnahme des Nutzers vor dem Start dieser Instanz. Konfiguriert werden

- Skripte für die automatisierte Ausführung vordefinierter Abläufe,
- Menüeinträge zum Starten von Operationen der Datenstruktur, Managern und vordefinierten Abläufen,
- die Anzeige von Knoten und Kanten inklusive Größe, Farbschemata, Stilschemata, Layout und die Auswahl darzustellender Attribute
- und auszugebende Meldungen aller sichtbarer Operationen in vordefinierten Standardsituationen.

Diese Punkte werden in den folgenden Unterabschnitten näher erläutert.

8.3.1. Einbindung vordefinierter Skripte

Eine einfache Möglichkeit, vordefinierte Skripte ablaufen lassen zu können, bietet der VIDEA-Skriptmanager, der in der AVL-Instanz und in der Listen-Instanz eingesetzt wurde, um wesentliche Abläufe beim Aufbau einer Datenstruktur im Rahmen des passiven Vorführens als kontinuierlichen Ablauf darzustellen. Ein Beispiel für ein Skript zum Aufbau eines AVL-Baums wurde auf Seite 152 gezeigt.

Um eigene Skripte einzubinden, ist nichts weiter erforderlich, als

- geeignete Operationen auszuwählen, aus denen der zu visualisierende Ablauf bestehen soll,
- deren Namen gefolgt von eventuellen Parametern zeilenweise in eine Textdatei zu schreiben,
- diese Textdatei ins `etc`-Verzeichnis der VIDEA-Instanz zu kopieren und
- in `musoft_config.xml` einen geeigneten Eintrag zu machen.

Zusätzlich zum eigentlichen Ablauf kann auch die Geschwindigkeit von Skripten in `musoft_config.xml` einstellen, wie in Abschnitt 8.3.3 gezeigt wird.

8.3.2. Konfiguration der Schnittstelle für Nutzeraktionen

Im Gegensatz zu der im nächsten Unterabschnitt beschriebenen Konfigurationsdatei `musoft_config.xml`, die speziell für VIDEA entwickelt wurde, wird `prototype.xml` in allen UPGRADE2-Erweiterungen genutzt. Sie dient dazu,

- Menüs zu definieren,
- Aktionen für Menüs zu definieren,
- Menüs automatisch aus PROGRES-Produktions- und -Transaktions-Namen zu generieren
- und Aktionen an Menüs zu binden.

Abb. 106 zeigt eine beispielhafte `prototype.xml`-Datei für die VIDEA-AVL-Instanz. Die gesamte Grundstruktur dieser Datei inklusive der Namen von Tags und Attributen ist durch UPGRADE2 vorgegeben. Wie man sieht, werden in den `COMMAND`-Tags im oberen Teil der Datei Kommandos definiert, die weiter unten als Aktionen für ebenso hier deklarierte Menüeinträge dienen; die gleichen Kommandos werden im Java-Code der Top-Level-VIDEA-Klasse (`DefaultyFilesViewWindow`) an in UPGRADE2 definierte Aktions-Objekte gebunden, deren *actionPerformed*-Methode beim Anklicken des entsprechenden Menüpunktes ausgeführt wird. Der Quellcode für `DefaultyFilesViewWindow` befindet sich im gleichen Verzeichnis wie alle VIDEA-spezifischen Klassen, also wie erwähnt im Verzeichnis `src/extensions/yfiles` der VIDEA-Instanz. Normalerweise ist es nicht notwendig, Java-Quellcode anzupassen; in Ausnahmefällen kann es aber Vorteile bringen, eine eigene Aktion zu codieren, um einen VIDEA-Manager mit festen Parametern zu versorgen, die weder in `musoft_config.xml` statisch vordefiniert werden sollen, noch im interaktiven Dialog des jeweiligen Managers vom Benutzer eingestellt werden sollen. Dafür muss in der aktuellen VIDEA-Version manuell eine Aktion im `DefaultyFilesViewWindow` hinzugefügt werden; eine schönere Lösung wäre die Verwaltung mehrerer Konfigurationen von Parametern für jeden Manager, die z. B. über `musoft_config.xml` konfigurierbar sind. Allerdings ist die Verwendung mehrfacher Konfigurationen nur in Ausnahmefällen sinnvoll, z. B. wenn der Lernende an einer speziellen Stelle des Fahrplans nicht wissen soll, auf was für ein Farbschema gerade umgeschaltet wird. Denn normalerweise werden neue Farbschemata mithilfe der in Abschnitt 8.4.1 beschriebenen interaktiven Schnittstelle des Farbmanagers eingestellt.

Ein Beispiel für die Nutzung eines festen Parametersatzes für den Farbmanager ist in Abb. 106 als `ColorUnbalancedNodes` gezeigt. Durch die explizite Einbindung dieser Aktion kann der Lernende während der Bearbeitung seiner Aufgabe von dem standardmäßig für diese VIDEA-Instanz eingestellten Farbschema zur Rotfärbung aller fehlerhaften Zustände in einem AVL-Baum auf ein Farbschema umschalten, das nur die unbalancierten Stellen des Baums markiert.

Ein Beispiel für das Einbinden der interaktiven Oberfläche eines Managers ist die in Abb. 106 definierte Aktion `YFILES.Config.NodeColoring`, die dann weiter unten an einen Menüeintrag `NodeColoring...` im Menü `Options` gebunden wird.

Die gerade erwähnte Bindung einer Aktion an einen Menüeintrag findet unter dem `VIEWWINDOW`-Tag statt; hier werden alle Menü- und Toolbar-Einstellungen für das Hauptfenster der VIDEA-Instanz festgelegt. In unserem Beispiel wird in `prototype.xml` eine Menüleiste (`MENUBAR`) bestehend aus den festen Menüs `File`, `Options` und `AVL Specific` definiert. „Fest“ bedeutet in diesem Zusammenhang, dass die Funktionalität bereits durch die statische Konfiguration (XML+Java) determiniert ist. Folgende Menüs sind generell als Ergänzung zu den unten beschriebenen automatisch erzeugten Menüs sinnvoll:

- Das `File`-Menü, in dem generische Aktionen wie `Close` zum Schließen des Fensters zur Verfügung gestellt werden können.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONFIGURATION>
<CONFIGURATION>

  <COMMANDS>
    <COMMAND name="YFILES.Close" tooltip="Close VIDEA" binding="ctrl C"
             icon="upgrade/icons/common/close.gif" />
    <COMMAND name="YFILES.Config.NodeColoring" />
    <COMMAND name="YFILES.Config.EdgeColoring" />
    <COMMAND name="YFILES.Config.NodeAttributeDisplay" />
    <COMMAND name="YFILES.Config.EdgeAttributeDisplay" />
    <COMMAND name="YFILES.Config.ColorUnbalancedNodes" />
    <COMMAND name="YFILES.Config.ColorUnsortedNodes" />
    <COMMAND name="YFILES.Exercise.RunDemo" />
    <COMMAND name="YFILES.Execute script" />
  </COMMANDS>

  <VIEWWINDOW name="DefaultGraph" caption="Generic graph view">
    <MENUBAR>
      <MENU name="File">
        <MENUITEM name="Execute script..." action="UPGRADE.ScriptManager.Start" />
        <SEPERATOR />
        <MENUITEM name="Close" action="UPGRADE.Views.Close" />
      </MENU>
      <MENU name="Options">
        <MENUITEM name="Node Coloring..." action="YFILES.Config.NodeColoring" />
        <MENUITEM name="Edge Coloring..." action="YFILES.Config.EdgeColoring" />
        <MENUITEM name="Node Attribute Display..."
                            action="YFILES.Config.NodeAttributeDisplay" />
        <MENUITEM name="Edge Attribute Display..."
                            action="YFILES.Config.EdgeAttributeDisplay" />
      </MENU>
      <GEN_MENU prefix="InMenu_Primitive_" name="Primitive" />
      <GEN_MENU prefix="InMenu_BinTree_" name="BinTree" />
      <GEN_MENU prefix="InMenu_Rotate_" name="Rotate" />
      <GEN_MENU prefix="InMenu_Exercise_" name="Exercise" />
      <GEN_MENU prefix="InMenu_Predefined_" name="Trees" />
      <MENU name="AVL specific">
        <MENUITEM name="predefinedTree1" action="YFILES.Exercise.RunDemo" />
        <MENUITEM name="color all unbalanced nodes"
                            action="YFILES.Config.ColorUnbalancedNodes" />
        <MENUITEM name="color all unsorted nodes"
                            action="YFILES.Config.ColorUnsortedNodes" />
      </MENU>
    </MENUBAR>
      <TOOLBARS>
      <TOOLBAR name="Generic">
        <TOOLBARITEM name="Close" action="UPGRADE.Views.Close" />
      </TOOLBAR>
     </TOOLBARS>
  </VIEWWINDOW>
</CONFIGURATION>
```

Abbildung 106: Eine beispielhafte Ausprägung der Konfigurationsdatei `prototype.xml` für die VIDEA AVL-Instanz

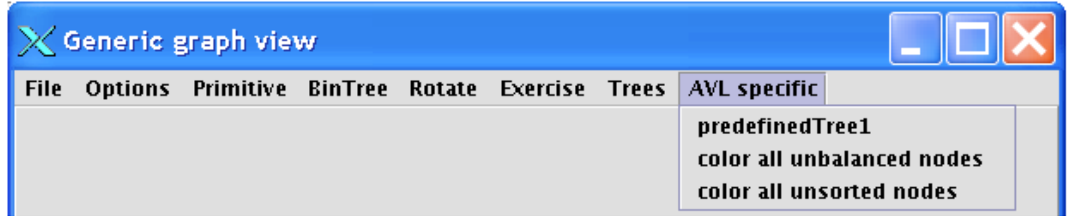


Abbildung 107: Die Menüleiste unserer Dozenten AVL-Instanz

- Das `Options`-Menü, in dem die Farb-, Stil- und Anzeigemanager von VIDEA angeboten werden können, wie wir das in den Dozenten-Versionen aller vier VIDEA-Instanzen getan haben.
- Einen Menüpunkt zum Aufruf vordefinierter Skripte.
- Einen instanzspezifischen Menüpunkt, in dem Aktionen, die an speziellen Java-Code innerhalb von VIDEA gebunden werden sollen, zusammengefasst werden.

Ein Beispiel für den dritten Punkt ist der Menüpunkt `AVL specific -> predefinedTree1` in unserer AVL-Instanz, der durch den Aufruf einer Klasse `YFilesRunDemo` den vordefinierten Aufbau des AVL-Baums durch den Ablauf des in `musoft_config.xml` konfigurierten Skripts `avlRunDemo` ermöglicht.

Ein Beispiel für den vierten Punkt obiger Liste sind in der AVL-VIDEA-Instanz im Menüpunkt `AVL specific` die zwei Aktionen (`color all unbalanced nodes` und `color all unsorted nodes`), die dazu dienen, die weiter oben beschriebenen festen, vordefinierten Farbschemata anzubieten.

Neben den beschriebenen direkt in `prototype.xml` konfigurierbaren Menüeinträgen gibt es noch eine zweite Art: Diese werden durch die PROGRES-Spezifikation festgelegt, indem die Füllung der Menüs mit Menüpunkten durch die Namen von PROGRES-Programmeinheiten bestimmt wird. Diese Menüs werden in der Konfigurationsdatei als `GEN_MENU` gekennzeichnet; ein Beispiel dafür in Abb. 106 ist das Menü `Primitive`, das mit allen Transaktionen und Produktionen der zugehörigen PROGRES-Spezifikation gefüllt wird, deren Namen mit *In-Menu_Primitive_* beginnen. Damit ist also ein Teil des Designs der VIDEA-Instanz bereits in der Namensgebung der visuellen Spezifikation enthalten.

Die ebenso vorhandene Möglichkeit, Toolbars anzugeben und anzulegen, wird durch den kleinen Block innerhalb des `TOOLBARS`-Tags angedeutet. In Abb. 106 wird nur eine Schaltfläche zum Schließen der Anwendung definiert (`TOOLBARITEM name="Close" ...`).

Abb. 107 zeigt die Menüleiste der VIDEA AVL-Instanz für die Präsenzlehre. Diese entspricht den gerade gegebenen Empfehlungen für die Aufteilung der Aktionen in Menüs und den Definitionen der Konfigurationsdatei aus Abb. 106.

8.3.3. Konfiguration des Standardverhaltens einer VIDEA-Instanz

In Abschnitt 8.4 werden wir u. a. Möglichkeiten beschreiben, zur Laufzeit mithilfe interaktiver VIDEA-Manager Farbschemata, Stilschemata und Attributanzeige-Schemata vorzugeben oder anzupassen. Es gibt jedoch einerseits Konfigurationsmöglichkeiten in VIDEA, die auf keiner interaktiven Oberfläche eines Managers konfigurierbar sind, andererseits ist es gerade für Fälle, in denen nur eine bestimmte Konfiguration benötigt wird, praktischer, diese vor dem Start definieren zu können. Man denke nur an den Fall eines vorbereiteten Szenarios in einer Präsenzveranstaltung oder an interaktive Übungen, in denen man den Lernenden gar keine Möglichkeit anbieten *will*, die Einstellungen anzupassen, damit z. B. versteckte Attribute auch versteckt bleiben. In solchen Fällen ist man auf eine Voreinstellung von Knotengrößen, angezeigten Attributen, Farbschemata etc. angewiesen.

Dies leistet die VIDEA-Konfigurationsdatei `musoft_config.xml`, die in VIDEA neu eingeführt wurde und in jeder VIDEA-Instanz im `etc`-Verzeichnis vorliegen muss. Abb. 108 zeigt einen Ausschnitt aus einer existierenden Datei für die VIDEA-Dijkstra-Instanz. Man sieht, dass es sich um eine XML-Datei handelt, die durch ein `ALL`-Tag umschlossen wird. Dazwischen können Einstellungen für das Layout, Grundeinstellungen für Knoten- und Kantentypen, Bedingungen für die Knoten- und Kanteneinfärbung, für die Strichelung von Kanten, abkürzende Aktionen für das Erzeugen und Löschen von Knoten und eine Menge von Abbildungsregeln von Transaktionsnamen und Rückgabe-Codes auf Meldungen vorgenommen werden.

Im Folgenden werden wir alle erwähnten Möglichkeiten beschreiben:

1. **Das zu verwendende Layout:**
   Das `LAYOUT`-Tag enthält die Angabe des Identifikators eines Layoutalgorithmus, so wie er im VIDEA-Layoutmanager bezeichnet wird. In Abb. 108 wird der Layoutalgorithmus mit Identifikator `Dijkstra` gewählt.
2. **Spezifikation der Knotentypen:**
   In Abb. 108 wird unter dem `NODETYPE`-Tag nur ein Knotentyp *Node* aus der zugehörigen PROGRES-Spezifikation (siehe Abb. 68 in Kapitel 6) definiert, der mit einer festen Breite von 150 Pixeln dargestellt werden soll. Die Höhen der Knoten werden in VIDEA aus der Anzahl der angezeigten Attribute berechnet und müssen deswegen nicht in `musoft_config.xml` angegeben werden. Von den Attributen des Typs *Node* sollen *id* und *distance* in dieser Reihenfolge übereinander angezeigt werden. Die Standardfarbe der Knoten vom Type *Node* ist *cyan*, falls sich keine andere Regel des `YFilesNodeColoringManager` auswirkt.

   Bei Dijkstras Algorithmus sind keine weiteren Knotentypen notwendig, anders als zum Beispiel bei der doppelt verketteten Liste, bei der wir zusätzlich zum Listenelement auch noch einen Listenkopf verwenden.
3. **Spezifikation der Kantentypen:**
   Normale PROGRES-Kanten wie *left* und *right* beim AVL-Baum müssen in VIDEA nicht spezifiziert werden. Was hier gemeint ist, sind Kantentypen mit Attributen, die wie schon in Kapitel 6 beschrieben mithilfe des Edge-Node-Edge-Filters durch spezielle Knotentypen emuliert werden.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<ALL>

  <LAYOUT algorithm="Dijkstra"/>

  <NODETYPE name="Node" nodeWidth="150" visibleAttributes="id,distance"
                                  defaultColor="cyan"/>

  <EDGETYPE name="Edge" visibleAttribute="weight" defaultColor="gray"/>

  <NODECOLORING>
    <CONDITION nodetype="Node" attributename="color"
                               lowvalue="0" highvalue="0" color="green"/>
    <CONDITION nodetype="Node" attributename="color"
                               lowvalue="1" highvalue="1" color="yellow"/>
    <CONDITION nodetype="Node" attributename="color"
                               lowvalue="2" highvalue="2" color="red"/>
  </NODECOLORING>

  <EDGECOLORING attributename="isOnTheWay" lowValue="1" highValue ="1" color="green"/>
  <EDGESTYLE attributename="" lowValue="" highValue ="" style=""/>
  <RUNDEMO file="dijkstraRunDemo" speed="2" showTransactionNames="1"/>
  <SPECIALPURPOSE makeNodeTransaction="makeNode" deleteNodeTransaction=""/>

  <MESSAGEMAPPING>
    <TRANSACTION transactionname="findRoot"
       noerrortxt=""
       normalerrortxt="Sorry, there is no valid root node. /A valid
                       root node is a node without incoming edges."
       parameterconversionerrortxt=""
       defaulttxt="Transaction failed"/>

    <TRANSACTION transactionname="nextStep"
       noerrortxt=""
       normalerrortxt="There is no (more) node in the queue. /
                       This means either that all nodes are green, so the
                       algorithm has finished / or that findRoot was not called
                       before nextStep or didn’t succeed."
       parameterconversionerrortxt=""
       defaulttxt="Transaction failed"/>

    <TRANSACTION transactionname="testCurrentU"
       noerrortxt="Yes!"
       normalerrortxt="No, this is wrong.  //
                       Remember that the current ’u’ is the node that was the
                       origin of all edges that were traversed in the last step. /
                       Try again."
       parameterconversionerrortxt="Please specify a valid node as parameter!"
       defaulttxt="Transaction failed"/>

    <TRANSACTION transactionname="testNextU"
       noerrortxt="Yes!"
       normalerrortxt="No, this is wrong.  //Remember that the next
                       ’u’ will be the yellow node with the smallest estimated
                       distance. /Try again."
       parameterconversionerrortxt="Please specify a valid node as parameter!"
       defaulttxt="Transaction failed"/>

    ...

  </MESSAGEMAPPING>

</ALL>
```

Abbildung 108: Ein Ausschnitt aus `musoft_config.xml` für die Dijkstra-Instanz

Der Kantentyp *Edge* in Abb. 108 ist also in der PROGRES-Spezifikation ein Knotentyp, der mindestens ein Attribut *weight* hat. In der VIDEA-Instanz wird dieser Knotentyp dann als Kante mit *weight*-Attribut und der Standardfarbe 'gray' dargestellt.

4. **Farbschemata für Knoten,** definiert durch das `NODECOLORING`-Tag**:**
   Die Semantik der durch die `CONDITION`-Tags gebildete Matrix wird in Abschnitt 8.4 bei der Beschreibung der interaktiven Oberfläche unserer Manager ausführlich dargestellt. Hier sei nur angedeutet, dass die Belegung in Abb. 108 bedeutet, dass jeder Knoten vom Typ *Node*, dessen Attribut *color* den Wert 0 (1, 2) hat, in grüner (gelber, roter) Farbe dargestellt werden soll.
5. **Farbschema für Kanten** (`EDGECOLORING`)**:**
   Hier gilt analoges wie beim Farbschema für Knoten. In unserem Fall werden diejenigen Kanten vom Knotentyp *Edge*, deren Attribut *isOnTheWay* den Wert *1* (für *true*) hat, grün dargestellt. Das sind gemäß Dijkstra-Algorithmus diejenigen Kanten, die bereits auf sicheren kürzesten Pfaden vom Startknoten zu anderen Knoten liegen.
6. **Stilschema für Kanten:**
   Im Beispiel in Abb. 108 wird kein Stilschema verwendet, wie man an den leeren Einträgen für `attributname`, `lowValue`, `highValue` und `style` im `EDGESTYLE`-Tag sieht. In der Kruskal-Instanz allerdings haben wir folgendes Stilschema verwendet, um die Kanten, die noch nicht vom Algorithmus betrachtet wurden, zu stricheln und damit auf übersichtliche Art von denjenigen zu unterscheiden, die im Spannbaum sind und farblich markiert wurden:
   ```
   <EDGESTYLE attributename="stillInQueue" lowValue="1"
    highValue ="1" style="dashed"/>
   ```
7. **Ablaufgeschwindigkeit eines vordefinierten Skriptes:**
   Im `RUNDEMO`-Tag werden drei Informationen für das Standard-Skript der zu dieser `musoft_config.xml` gehörigen VIDEA-Instanz angegeben: Der Name des Standardskriptes in `file`, die Geschwindigkeit des Skriptes in `speed` und der boolesche Wert, der angibt, ob eine Anzeige der Transaktionsnamen erwünscht ist, in `showTransactionNames`. Bei der Geschwindigkeits-Angabe bedeutet eine größere ganze Zahl eine niedrigere Geschwindigkeit. Die maximal erlaubte Geschwindigkeit ist 2, das heißt es sind Werte $>= 2$ erlaubt.
8. **Definition einfacher Aktionen** (Schnelltasten oder Mausklicks) **:**
   Im `SPECIALPURPOSE`-Tag können die Namen zweier PROGRES-Programmeinheiten angegeben werden, die das Erzeugen und das Löschen eines Knotens eines bevorzugten Knotentyps dieser VIDEA-Instanz durchführen. Die in VIDEA implementierten Aktionen hierfür sind der Klick auf die linke Maustaste für die über `makeNodeTransaction` spezifizierte Transaktion und der Klick auf die mittlere Maustaste für die über `deleteNodeTransaction` spezifizierte Transaktion.
9. **Die Zuweisung von Meldungen an Transaktionen:**
   Diese Funktionalität ist in UPGRADE2 in der gewünschten Form nicht vorhanden und wurde in VIDEA neu implementiert.

   Das Konzept für Meldungen stammt aus den internen Fehlercodes von UPGRADE2 und sieht vor, dass es für jede Transaktion oder Produktion bis zu drei Meldungen geben kann,

die im `TRANSACTION`-Tag konfiguriert werden:

- Erfolgreiche Ausführung als Wert des Attributs `noerrortxt`: In diesem Fall wird üblicherweise keine Meldung ausgegeben, aber für unser didaktisches Konzept ist es sehr hilfreich, wenn bei bestimmten Aktionen auch im Erfolgsfall eine Meldung ausgegeben werden kann. Ein Beispiel ist, im Rahmen unserer AVL-VIDEA-Instanz bei den Rotationsoperationen im Erfolgsfall eine Meldung 'Correct!' anstatt nichts auszugeben. Auf diese Weise erhalten die Lernenden ein verlässliches Feedback.
- Normaler Fehler als Wert des Attributs `normalerrortxt`: Dieser Fehler tritt auf, wenn die Transaktion aus einem nicht näher spezifizierten Grund fehlschlägt, ein Fall, in dem PROGRES normalerweise die Meldung 'Transaction Failed' ausgibt. In diesem Fall kann meistens eine Meldung ausgegeben werden, die auf die individuelle Transaktion zugeschnitten ist und somit den Lernenden wertvolle Hinweise in Richtung auf einen Lernschritt gibt.
- Parameter-Fehler als Wert des Attributs `parameterconversionerrortxt`: Es wurden die falsche Anzahl oder die falschen Typen von Parametern übergeben. Bei den meisten Transaktionen lohnt es sich nicht, hier eine spezielle Meldung außer 'wrong parameters' anzugeben; allerdings gibt es Ausnahmefälle wie *insert* bei der doppelt verketteten Liste, wo nochmal darauf hingewiesen wird, eine positive ganze Zahl zu spezifizieren oder knotenbasierte Transaktionen, wo man zurückmelden kann, dass doch bitte ein existierender Knoten anzugeben bzw. anzuklicken sei.

Die ebenso in `musoft_config.xml` anzugebende Meldung für das Attribut `defaulttxt` bezieht sich auf die Ausgabe einer Standardmeldung in einer möglicherweise unvorhergesehenen Fehlersituation von UPGRADE2 zur Laufzeit der VIDEA-Instanz.

Um also eigene Meldungen für eine Transaktion ausgeben zu können, genügt es, ein `TRANSACTION`-Tag zu erstellen und nach den gezeigten Beispielen in Abb. 108 den Transaktionsnamen und die drei Arten von Nachrichten (plus 'Transaction Failed' als Standardtext) einzugeben. Dabei können durch die Verwendung von Schrägstrichen, wie in Abb. 108 zu sehen, Zeilenumbrüche und damit mehrzeilige Nachrichten spezifiziert werden.

Die Nutzung geeigneter Layoutalgorithmen stand bei der Entwicklung von VIDEA von Anfang an sehr weit vorne auf der Prioritäten-Liste, weil ein nicht zu vernachlässigbarer Einfluss der Art der Visualisierung und des Morphings auf den Lerneffekt vermutet wird. Im kommenden Abschnitt wird die grobe Funktionsweise von Layoutmanagern in VIDEA beschrieben und gezeigt, wie auf der Basis von yFiles ein neuer Layoutmanager erstellt und in die VIDEA-Instanz eingebunden werden kann.

8.3.4. Layoutalgorithmen in VIDEA

Eine unserer grundsätzlichen Anforderungen in Kapitel 3 war „*Layoutalgorithmen sind unabhängig von Datenstruktur und Algorithmus konfigurierbar*" (Anforderung K4). Im letzten Unterabschnitt wurde gezeigt, wie das zu verwendende Layout einer VIDEA-Instanz in `musoft_config.xml` unabhängig von Datenstruktur und Algorithmus konfiguriert werden

kann. In diesem Abschnitt werden wir darauf eingehen, was zu tun ist, wenn ein Layoutalgorithmus benötigt wird, der in VIDEA noch nicht vorhanden ist. Im Wesentlichen muss dafür eine VIDEA-Klasse, nämlich der `YFilesLayoutManager`, angepasst werden. Dieser enthält eine öffentliche Methode *doLayout()*, in der wiederum eine Methode für das eigentliche, jeweilige Layout aufgerufen wird. Im Moment stehen im `YFilesLayoutManager` vier Layout-Methoden zur Verfügung:

- Eine Methode für Binärbäume,
- eine Methode für die doppelt verkettete Ringliste,
- eine Methode für allgemeine Graphen und
- eine Methode mit dem Verzicht auf Layout.

Prinzipiell sind zum Hinzufügen eines neuen Layouts folgende zwei Schritte notwendig:

1. Die Entscheidung, ob einer der mitgelieferten Layoutalgorithmen verwendet werden kann. Falls ja, genügt der entsprechende Eintrag in `musoft_config.xml`.

2. Falls nein, muss
   - eine neue Methode do<*eigenerName*>Layout() im `YFilesLayoutManager` implementiert werden
   - und das XML-Tag `<eigenerName>` in `musoft_config.xml` eingefügt werden:

```
<LAYOUT algorithm="<eigenerName>"/>
```

Für das Erstellen des eigentlichen Codes eines neuen Layoutmanagers können hier keine allgemeingültigen Regeln gegeben werden, da es sich dabei um eine von der Datenstruktur abhängige Funktionalität handelt. Allerdings stehen durch die Einbindung von yFiles in VIDEA mächtige Bausteine für die Erstellung neuer Layoutmanager zur Verfügung, die es erlauben, schneller als zum Beispiel direkt in Java die gewünschte Animation umzusetzen.

Um trotzdem einen Eindruck im Hinblick auf die Erstellung eines neuen Layoutmanagers mithilfe von yFiles und Java zu vermitteln, geben wir im Folgenden einen Überblick über die Implementierung der in VIDEA vorhandenen Layoutmanager.

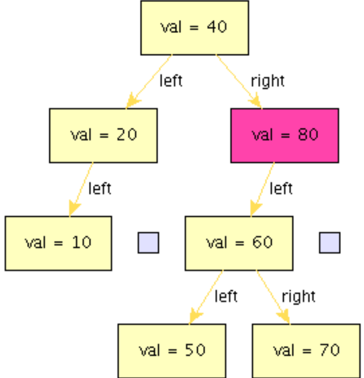
AVL-Bäume:

Beim Layouting von AVL-Bäumen ist es wichtig, dass

1. AVL-Bäume als binäre Bäume dargestellt werden,
2. die gegebene Unterscheidung der Kinder eines Knotens in linke und rechte Kinder seitenerhaltend dargestellt wird, das heißt linke Kinder links von rechten Kindern, und
3. Veränderungen am Baum, insbesondere Rotationsoperationen auf eine weiche, flüssige Art animiert werden, um unserem didaktischen Vorsatz treu zu bleiben.

Punkt 1 wurde erreicht durch Verwendung des yFiles `TreeLayouters`. Dieser Layouter stellt n-äre Bäume auf übliche Art dar.

Punkt 2 wurde eigens für die Visualisierung von AVL-Bäumen implementiert. Dazu wird vor dem eigentlichen Anwenden des TreeLayout auf den Baum oder die Bäume der Graph mit einer



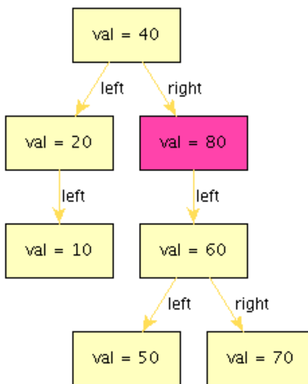


Abbildung 109: Unbalancierter AVL-Baum, links mit sichtbaren Pseudo-Knoten, rechts ohne

Hilfsfunktion *prepareAVLNodeCoord* vorbereitet. Dazu wird zum Einen dafür gesorgt, dass sich vor dem eigentlichen Layout-Prozess die Position des linken Kindes links von der Position des rechten Kindes befindet, wenn ein Elternknoten mindestens ein linkes und ein rechtes Kind hat. Notfalls wird das linke Kind etwas verschoben. Das reicht deswegen aus, weil das yFiles `TreeLayout` seitenerhaltend ist, d. h. linke Knoten *vor* Berechnung des Layout bleiben linke Knoten *nach* dem Layout-Prozess. Zum Anderen werden für Knoten, die nur ein Kind besitzen, Pseudo-Knoten der Größe Null eingefügt, die keine Bedeutung in der Datenstruktur haben und deshalb auch nicht im PROGRES-Graphen vorkommen, aber das Layout beeinflussen. Dadurch wird erreicht, dass z. B. im Falle eines Knotens, dessen einziges Kind an einer *left*-Kante hängt, dieser linke Kindknoten auch wirklich links unter seinem Elternknoten platziert wird.

Abb. 109 zeigt zur Veranschaulichung links ein Szenario mit einem zufälligerweise unbalancierten AVL-Baum, in dem die Pseudo-Knoten sichtbar gemacht wurden. Rechts ist derselbe Baum zu sehen, wie er ohne das Hilfsmittel der Pseudo-Knoten aussehen würde. Man sieht, dass Knoten 60 unschönerweise direkt unter seinem Elternknoten gezeichnet wird.

Punkt 3 wird sichergestellt, indem der Layout-Vorgang unter Nutzung eines `LayoutMorpher` durchgeführt wird, der vermöge einer Methode *setSmoothViewTransform* eines yFiles-`AnimationPlayer` in einem flüssigen Animationsschritt das geforderte Layout erzeugt.

Ringliste:

Die Methode *doDllLayout* für das Erstellen eines passenden Layouts für die doppelt verkettete Ringliste nutzt mehrere von yFiles angebotene Layout-Mechanismen:

- Den `SingleCycleLayouter` für die Darstellung auf einem Kreis.
- Den `ParallelEdgeLayouter`, damit die *previous*- und *next*- Kanten zwischen aufeinanderfolgenden Listenelementen (anti)parallel dargestellt werden.

```
...

SingleCycleLayouter  singleCycleLayouter;
ParallelEdgeLayouter parallelEdgeLayouter;
GreedyMISLabeling    greedyMISLabeling;
AnimationPlayer      player;
Graph2D              graph;

...



private void doDllLayout()  {

  ...

  singleCycleLayouter.setAutomaticRadius(true);
  singleCycleLayouter.setMinimalNodeDistance(100);
  singleCycleLayouter.setNodeSequencer(this);
  greedyMISLabeling.setPlaceNodeLabels(false);
  greedyMISLabeling.setPlaceEdgeLabels(true);

  try  {  singleCycleLayouter.setLabelLayouter(greedyMISLabeling);
          singleCycleLayouter.setLabelLayouterEnabled(true);
          singleCycleLayouter.setParallelEdgeLayouter(parallelEdgeLayouter);
          singleCycleLayouter.setParallelEdgeLayouterEnabled(true);
          GraphLayout graphLayout = singleCycleLayouter.calcLayout(graph);
          LayoutMorpher layoutMorpher = new LayoutMorpher(yview, graphLayout);
          layoutMorpher.setSmoothViewTransform(true);
          player.animate(layoutMorpher);
       }
  catch(Exception e)
    {  log.error(e);
    }

  yview.fitContent();
  yview.updateView();

  ...
}
```

Abbildung 110: Ein Ausschnitt aus doDllLayout()

- Den `GreedyMISLabeling`, damit sich die Beschriftungen dieser Kanten nicht überdecken.
- Einen `LayoutMorpher`, wie schon beim AVL-Baum, um z. B. beim Einfügen neuer Elemente eine flüssige Bewegung zu zeigen.

Mit diesen Hilfsmitteln werden die Listenelemente also in einem Ring angeordnet, mit parallelen Kanten, schön platzierten Beschriftungen und in flüssiger Animation. Abb. 110 zeigt oben einige Deklarationen auf Klassenebene des Layoutmanagers, unten den wesentlichen Teil des beschriebenen Layoutalgorithmus. Hier wird grundsätzlich ein `SingleCycleLayouter` zur Anordnung der Knoten auf einem Kreis verwendet. Die verschiedenen Voreinstellungen dienen

der Erhöhung der Übersichtlichkeit. Die Nutzung des Layoutmanagers selbst als `NodeSequencer` dient der weiter unten beschriebenen Möglichkeit, eine bestimmte Reihenfolge der Knoten auf der Liste vorzugeben. Ein Objekt *greedyMISLabeling* sorgt dafür, dass die Kantenbeschriftungen sich nicht überlappen, so wie der `ParallelEdgeLayouter` die parallele Darstellung der Kanten durchführt. Danach wird in den letzten vier Zeilen des `try`-Konstrukts in Abb. 110 das Layout auf dem yFiles-Graphen *graph* berechnet, einem Morpher zugewiesen und auf eine weiche, flüssige Art animiert. Schließlich wird der gesamte Inhalt des Graphen wieder auf die aktuelle Größe des Fensters angepasst und der View aktualisiert.

Leider ist der Layoutalgorithmus in dieser Form trotzdem noch nicht brauchbar, weil die Reihenfolge der Listenelemente auf dem Kreis zufällig ist; insbesondere werden logisch aufeinanderfolgende Knoten, die durch *next*- und / oder *previous*-Kanten verbunden sind, nur selten auch nebeneinander dargestellt, was dazu führt, dass Kanten quer durch die Liste gehen und ein ziemliches Durcheinander entsteht.

Da die korrekte und übersichtliche Darstellung einer Datenstruktur für eine Lernumgebung aber unbedingt erforderlich ist, ist es notwendig, einen zusätzlichen Algorithmus zu implementieren, der die Listenelemente in die korrekte logische Reihenfolge bringt, was nicht immer eindeutig ist; schließlich können beliebig viele Kanten in der doppelten Verkettung fehlen. Das liegt daran, dass wir in unserer Lernumgebung den Lernenden prinzipiell Fehler erlauben, um diese dann entsprechend zu visualisieren und deren Beseitigung durch Fehlermeldungen ans Herz zu legen. In manchen Fällen ist dann trotzdem eine eindeutige Reihenfolge feststellbar, da es mit den vier verwendeten Kantentypen *first*, *last*, *previous* und *next* eine erhebliche Redundanz in der Liste gibt, die natürlich beabsichtigt ist, damit überhaupt Fehler der Lernenden und damit ein Lerneffekt über das passive Betrachten hinaus möglich ist.

Abb. 111 zeigt eine doppelt verkettete Liste; die Reihenfolge der Listenelemente auf dem Kreis ist trotz vereinzelt fehlender Kanten und der zwei kompletten Unterbrechungen der Liste zwischen den Listenelementen 5 und 8 bzw. 1 und 9 eindeutig, weil durch *first*- und *last*-Kanten das erste und letzte Element klar definiert ist und zusätzlich das Listenelement 5 wegen der *next*-Kante unmittelbar nach dem Listenelement 4, und das Listenelement 8 wegen der *previous*-Kante unmittelbar vor 9 eingeordnet werden muss.

Unsere Verbesserung des beschriebenen Algorithmus nutzt die Möglichkeit des yFiles `SingleCycleLayouter`, einen `NodeSequencer` anzugeben, wie er schon in Abb. 110 angedeutet, dort aber noch nicht genutzt wird. Dieser `NodeSequencer` kennt dann die Reihenfolge der Listenelemente im Ring, derer sich der `SingleCycleLayouter` bedienen kann. Wir füllen den `NodeSequencer` nach einer Berechnungsvorschrift der folgenden Art:

1. Suche das erste Listenelement. Das gelingt entweder durch vorhandene *first*-Kante oder durch Suchen eines Elements, das keine eingehende *next*-Kante, aber eine ausgehende *next*-Kante und keine ausgehende *previous*-Kante hat.
2. Baue durch Folgen der *next*-Kanten soweit wie möglich eine Reihenfolge der Elemente ab dem ersten Element für den Beginn des Kreises auf.
3. Suche das letzte Listenelement als Ziel einer ggf. vorhandenen *last*-Kante (hier werden

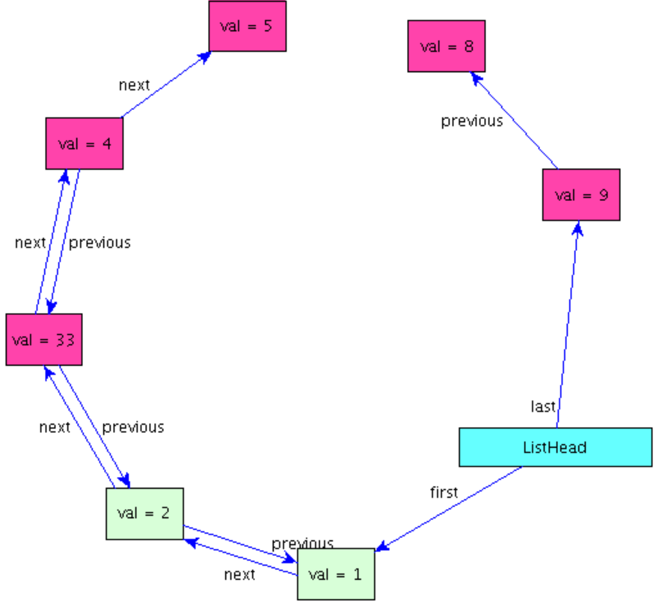


Abbildung 111: Beispiel einer Liste, die trotz Unterbrechungen eindeutig zu layouten ist

sonst keine Möglichkeiten geboten, wir wollen eine unsaubere Verkettung der Listenelemente am Listenende optisch bestrafen).

4. Falls das letzte Element nicht bereits in Schritt 2 gefunden wurde, baue durch Folgen der *previous*-Kanten soweit wie möglich eine Reihenfolge der Elemente vom letzten Element rückwärts für das Ende des Kreises auf.
5. Füge alle evtl. noch nicht betrachteten Elemente, ausgehend vom ersten erzeugten, zwischen den bisher gefundenen Anfangs- und Schluss-Elementen in der Reihenfolge des Folgens ihrer *next*-Kanten und dann, falls es noch nicht betrachtete Elemente gibt, in der Reihenfolge ihrer Erzeugung in den Kreis ein. Die Erzeugungsreihenfolge wird von yFiles automatisch verwaltet.

Bei diesem Schema ging es nicht darum, immer die bestmögliche darstellbare Ordnung zu finden. Denn einerseits muss in vielen Fällen die Reihenfolge inkonsistent sein, wenn die Lernenden in ihrem Algorithmus oder ihren interaktiven Versuchen zu viele oder zu gravierende Fehler machen. In welcher Reihenfolge stellt man z. B. zwei Listenelemente dar, die sich gegenseitig mit next-Kanten referenzieren? Und andererseits kann es aus didaktischen Gründen sinnvoll sein, die Lernenden bei Fehlern zusätzlich zu den bisher beschriebenen Mechanismen auch durch ein zerstörtes Layout auf grobe Inkonsistenzen aufmerksam zu machen.

Dieses Schema hat sich in unseren praktischen Versuchen bisher bewährt, obwohl es absichtlich nicht immer die beste mögliche darstellbare Ordnung findet. So könnte man z. B. beim Fehlen der *next*-Kanten in Schritt 2 alternativ eventuell vorhandenen *previous*-Kanten rückwärts folgen und vieles andere mehr. Hier wurde also aus Demonstrationsgründen eine willkürliche

Entscheidung getroffen. Bei einigen Versuchen zeigte es sich, dass es wichtig ist, noch zwei zusätzliche Eigenschaften des Layouts garantieren zu können: Erstens sollte das Layout nicht kippen, d. h. die Reihenfolge der Elemente im Kreis sollte stabil bleiben – entweder im Uhrzeigersinn oder gegen ihn. Zweitens sollten sowohl der Listenkopf als auch der Einstiegspunkt in die Liste nach Möglichkeit räumlich stabil bleiben. Beide Forderungen werden durch geeignete Parametrisierung der beschriebenen yFiles-Objekte erfüllt und durch die Eigenschaften des `SingleCycleLayouter` gewährleistet.

Algorithmen von Dijkstra und Kruskal:

Bei der Visualisierung der beiden Graphalgorithmen Dijkstra und Kruskal in VIDEA kommt es im Gegensatz zu den AVL-Bäumen und der doppelt verketteten Liste nicht darauf an, ein der Datenstruktur inhärentes Layout möglichst übersichtlich darzustellen, oder flüssige Bewegungen zu gewährleisten. Im Gegenteil könnten flüssige Animationen – oder genauer gesagt Ortveränderungen von Knoten und Kanten – hier eher verwirren, weil sich der Fortschritt der Berechnung wegen der unveränderlichen Struktur des Graphen in diesen Fällen nur in einer Änderung der Knoten- und Kanten-*Attribute* zeigen kann, die ja wie beschrieben u. a. als Änderungen der Farben oder des Stils dargestellt werden.

Deshalb wird in diesen beiden VIDEA-Instanzen sogar nur einmal ein Layout durchgeführt und danach abgeschaltet, allerdings mit der Option, über das Menü jederzeit zwischen einem automatischen Layout, einem einmaligen Layout und eben keinem Layout umschalten zu können. So kann gewährleistet werden, dass einerseits anfangs dem Benutzer ein „schönes" Layout angeboten wird, er dieses aber nach Belieben seinem Geschmack entsprechend abändern kann, ohne dass es wieder durch einen automatischen Layout-Prozess zerstört wird. Andererseits kann beim manuellen, strukturellen Ändern des Graphen, zum Beispiel nach dem Einfügen oder Löschen von Knoten oder Kanten, schnell wieder ein ansprechendes Layout hergestellt werden.

Das yFiles-Paket bietet zum Erstellen eines Layouts allgemeiner Graphen eine ganze Reihe interessanter Layoutalgorithmen, etwa orthogonale, zufällige oder organische Layouter (letztere simulieren anhand konfigurierbarer Eigenschaften jedes Knotenpaares anziehende oder abstoßende Kräfte und bewegen die Knoten in eine Position, in die sich entsprechende Teilchen in einem physikalischen Systemen bewegen würden, siehe [yfi05]). In unseren Versuchen haben sich diese aber teilweise als überfrachtet für kleine Lehrgraphen, wie sie in VIDEA verwendet werden, herausgestellt. In anderen Fällen gab es zum Entwicklungs-Zeitpunkt Probleme in den betrachteten Layoutern, die dazu führten, dass zwischen Knoten nicht soviel Abstand blieb, wie vom Platz her möglich, so dass Kantenbeschriftungen nicht mehr zu lesen waren oder Knoten sich sogar überlappten.

Deshalb wurde beschlossen, aus dem bewährten `SingleCycleLayouter` und einigen zusätzlichen Maßnahmen einen Layoutalgorithmus zusammenzusetzen, der die stark verzeigerten Knoten prinzipiell in einem Kreis anordnet, während schwach verzeigerte außerhalb des Kreises angeordnet werden können. Das hat sich bei allen von uns getesteten Beispielen als übersichtliches Anfangslayout bewährt (siehe z. B. Abb. 112).

Zusammenfassend lässt sich sagen, dass – wie erwartet – die zwei Algorithmen auf nicht näher spezifizierten Graphen mit Standard-Layoutverfahren einfach umgesetzt werden konnten. Die Visualisierung der Bäume und Listen forderte einen erheblich größeren Aufwand.

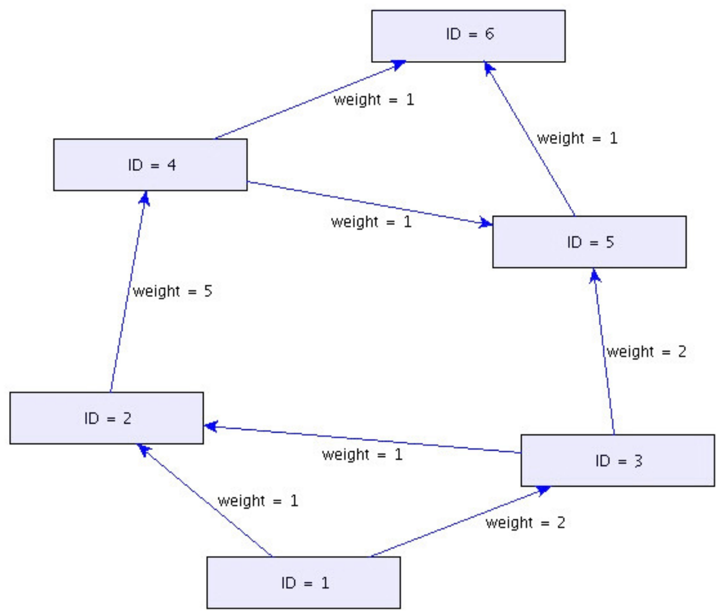


Abbildung 112: Wiederholung von Abb. 28 aus Kapitel 5

Diese Unterscheidung kann bei der Planung helfen, um den Aufwand abzuschätzen, der nötig sein wird, um eine neue Datenstruktur oder einen neuen Algorithmus passend und lehrreich darzustellen. Letztendlich ist aber für die Erstellung eines neuen Layoutalgorithmus im Allgemeinen die Programmierung einer Java-Methode notwendig, auch wenn auf mächtige vorhandene Klassen zurückgegriffen werden kann.

Nach der Erörterung dieser abschließenden Unterscheidungen zum Thema Layoutalgorithmen behandelt der nächste Abschnitt die Bedienung und Konfiguration einer VIDEA-Instanz.

8.4. Bedienung und dynamische Konfiguration

Die Bedienung einer VIDEA-Instanz erfordert keine ausführliche Beschreibung, da der Start einfach durch den Aufruf eines Shell-Skripts geschieht und dann eine graphische Oberfläche zur Verfügung steht, wie sie von anderen Werkzeugen bekannt ist.
Als hauptsächliche Benutzerschnittstelle dient das Menü, in dem

- Operationen zur Verfügung stehen, die als PROGRES-Programmeinheiten definiert wurden und zum Aufbau oder der Manipulation der Datenstruktur dienen, Schritte des Algorithmus ausführen oder Verständnistests initiieren,
- vordefinierte Skripte aufgerufen werden können,

- Schnittstellen zu den VIDEA-Managern zur Verfügung stehen, um zum Beispiel das automatische Layout ein- oder auszuschalten oder das in `musoft_config.xml` vordefinierte Farbschema zu verändern und
- die VIDEA-Instanz beendet werden kann.

Das Grundprinzip der Bedienung des interaktiven Graphbrowsers ist immer gleich: Zunächst wird eine Datenstruktur aufgebaut und danach bearbeitet, wobei der Aufbau durch Aktionen wie diese erfolgen kann:

- Die Ausführung einer speziellen vordefinierten Produktion auf den leeren Graphen wie *predefinedGraph1* im Dijkstra-Beispiel,
- die manuelle Ausführung gegebener Primitive wie *createNode* oder
- vordefinierte Skripte zum kontinuierlichen Aufbau der jeweiligen Datenstruktur wie *predefinedTree1* im AVL-Beispiel.

Die zweite Möglichkeit der Einflussnahme besteht in der unmittelbaren Manipulation der visualisierten Datenstruktur durch Bewegen, Markieren oder Verschieben der graphischen Elemente mit der Maus. Eine zusätzliche Möglichkeit in VIDEA ist es, eine Standardoperation für das Erzeugen eines spezifischen Knotentyps in `musoft_config.xml` vorzudefinieren. In diesem Fall kann durch einen Mausklick auf den leeren Hintergrund des Datenstruktur-Bereichs die vordefinierte Operation aufgerufen und damit ein Knoten des entsprechenden Knotentyps erzeugt werden.

Analog kann – bei entsprechender Konfiguration in `musoft_config.xml` – durch das Anklicken mit der mittleren Maustaste ein Knoten gelöscht werden.

Die dritte Möglichkeit der Einflussnahme besteht in der ferngesteuerten Animation laufender Programme der Lernenden. Hierzu genügt es, das eigene Programm, das natürlich die Schnittstelle des `MusoftClient` nutzen muss, zu starten, oder in einem beliebigen Debugger schrittweise auszuführen.

8.4.1. Interaktive Nutzung der VIDEA-Manager

Entschließt sich der Dozent, die interaktiven Stil-, Attributanzeige- oder Farbmanager im Menü anzubieten, dann hat der Nutzer die Möglichkeit, die in `musoft_config.xml` gespeicherten Anfangswerte für Stil-, Attributeanzeige- und Farbschemata zur Laufzeit der VIDEA-Instanz interaktiv zu verändern.

Die Vorgehensweise ist dabei immer dieselbe: Nach dem Start des jeweiligen Managers aus dem `Options`-Menü wird ein Dialog geöffnet (siehe Abb. 113), der oben in Form einer Auswahlliste die Wahl des Knotentyps bzw. Attributs ermöglicht, für den die beabsichtigten Einstellungen getätigt werden sollen.

Die in PROGRES vorgesehene Aufteilung der Knotentypen auf Knotenklassen lassen wir in der PROGRES-Spezifikation zwar zu (und verwenden sie z. B. in unserer Dijkstra-Instanz zur Aufteilung in Knoten und attributierte, simulierte Kanten auch selbst, siehe Kapitel 6), zeigen aber in der Auswahlliste der Knotenmanager einfach alle in `musoft_config.xml` definierten Knotentypen in einer flachen Struktur. Bei den Kantenmanagern setzen wir aus Gründen der

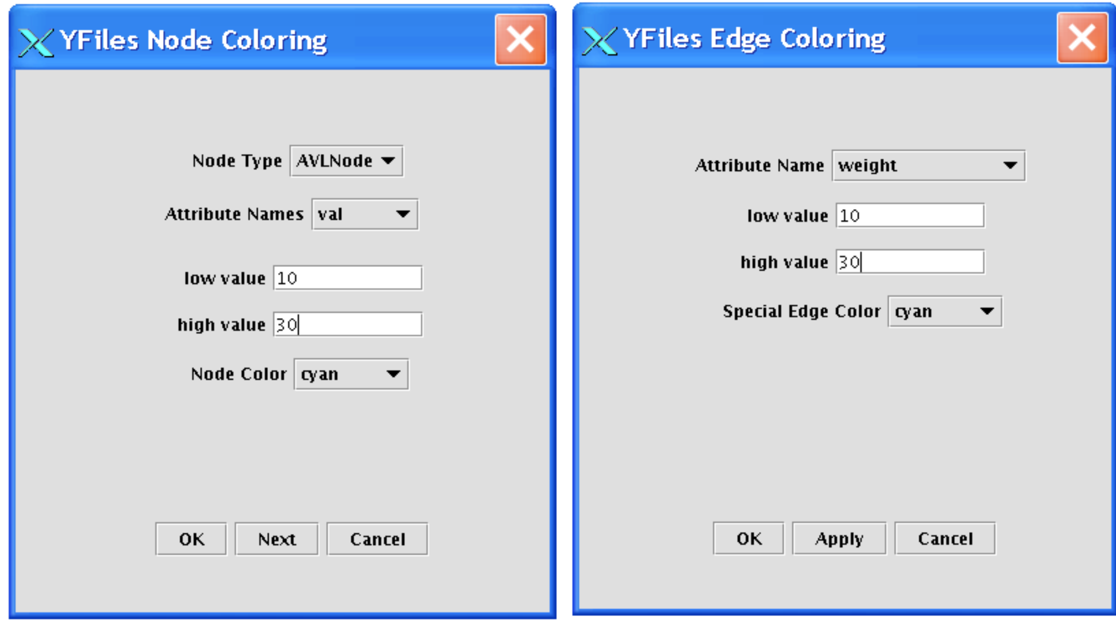


Abbildung 113: Die Oberfläche zweier Manager. Links: Der Node Coloring Manager der AVL-Instanz. Rechts: Der Edge Coloring Manager der Dijkstra-Instanz.

Übersichtlichkeit der entstehenden Visualisierung maximal einen attributierten Kantentyp voraus, der – falls vorhanden – ebenso in `musoft_config.xml` festgelegt ist, und zeigen keine Auswahlliste für den Kanten-Typ mehr an (siehe Abb. 113 rechts).

Im Fall der Stil- und der bisher beschriebenen Farbmanager kann dann in einer zweiten Combobox dasjenige Attribut des vorher spezifizierten Kanten- oder Knotentyps gewählt werden, auf dessen Werte sich die als Nächstes einzugebenden Intervalle beziehen. Abb. 113 links zeigt den `YFilesNodeColoringManager` der AVL-VIDEA-Instanz in Aktion. Als Knotentyp wurde der in diesem Fall einzige Typ *AVLNode* gewählt, als aktuelles Attribut *val*.

Insgesamt bedeutet das in Abb. 113 links dargestellte Szenario folgendes: Alle Knoten vom Typ *AVLNode*, für die nicht schon eine vor dem letzten Drücken der Schaltfläche ’Next’ angegebene Regel zutrifft und deren *val*-Attribut einen Wert zwischen *10* und *30* besitzt, sollen ab sofort in der Farbe *cyan* dargestellt werden. Die Schaltfläche ’Next’ schaltet eine Regel bzw. eine Zeile in der internen Datenstruktur, entsprechend den Zeilen im `CONDITION`-Tag in `musoft_config.xml`, weiter und zeigt die Änderungen gleich im Graphen an; dadurch wird es ermöglicht, eine neue Farb-Bedingung einzugeben und mit einem weiteren Mausklick auf ’Next’ zu testen, wie sich der aktuelle Graph mit dieser zusätzlichen Änderung darstellt. Beim Bestätigen mit ’OK’ werden die eingegebenen Farbregeln übernommen und der Dialog beendet. Beim Abbruch mit ’Cancel’ wird ebenfalls der Dialog beendet, ohne allerdings die Farbregeln zu übernehmen. Das bezieht sich auch auf alle ggf. bereits mit ’Next’ eingegebenen Regeln, so dass die vorhergehende Farbgebung der Datenstruktur wiederhergestellt wird.

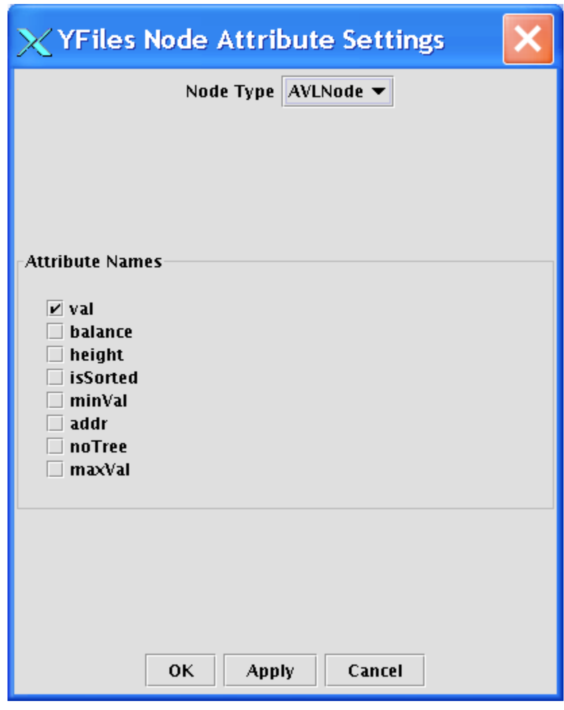


Abbildung 114: Der Node Attribute Display Manager

Nachdem es beim bisherigen Einsatz von VIDEA nie notwendig war, mehr als vier Regeln für einen Knotentyp anzugeben, wurde kein Mechanismus zum Löschen einer Regel eines aktuell editierten Regelsatzes eingeführt, da sich die unvermeidliche Komplizierung der Schnittstelle für den Lernenden durch eine ggf. auftretende minimale Arbeits-Ersparnis bei der Eingabe nicht lohnen würde. Sollte es zukünftig neue Erkenntnisse geben, wäre es natürlich leicht möglich, eine entsprechende Erweiterung in VIDEA einzubauen.

Wichtig ist, dass die zuerst eingegebenen Zeilen Vorrang haben vor den später eingegebenen; das heißt, dass folgende Reihenfolge von Werten als Parameter des `YFilesNodeColoringManagers` (ob in `musoft_config.xml` oder interaktiv im Dialog, ist natürlich egal) in einem AVL-Baum einen Knoten mit `val=10` *rot* färben würde:

```
AVLNode,  val,  10,  20,  red.
AVLNode,  val,   5,  10,  green.
```

Die umgekehrte Reihenfolge

```
AVLNode,  val,   5,  10,  green.
AVLNode,  val,  10,  20,  red.
```

dagegen würde den Knoten mit Wert 10 *grün* färben.

Durch dieses Konzept, Farbschemata für Knoten anzugeben, kann man bei entsprechender Anzahl von Regeln jede denkbare Kolorierung erreichen; denn im Extremfall ist es ja möglich,

über ein Schlüsselattribut für jeden Knoten eine eigene Regel anzugeben. In unseren Erfahrungen mit den bisher erzeugten vier Typen von VIDEA-Instanzen wurden maximal vier Regeln bei im Moment bis zu zwei Knotentypen benötigt. VIDEA erlaubt die Angabe beliebig vieler Regeln.

Die Funktionsweise des Managers für Kantenfarbschemata ist eine vereinfachte Variante der gerade beschriebenen Funktionsweise des Managers für Knotenfarbschemata. Da die attributierten Kanten – und nur für diese Kanten macht ein Farbschema nach unserer Philosophie einen Sinn – in VIDEA und UPGRADE2 durch genau einen Knotentyp simuliert werden, wird dieser Knotentyp zugrunde gelegt. Ebenso lassen wir nach einigen Versuchen mit der Einfärbung von Datenstrukturen aus Gründen der Übersichtlichkeit für die Kanten nur *eine* Farbregel zu, weswegen es in Abb. 113 rechts statt der 'Next'-Schaltfläche eine 'Apply'-Schaltfläche gibt, die die Änderungen unmittelbar im Graphen anzeigt, ohne die Eingabe einer zusätzlichen Regel zu erlauben. Dafür gibt es für die Darstellung der Kanten zusätzlich einen Stilmanager, dessen Oberfläche hier nicht noch extra gezeigt wird, der aber durch eine zusätzliche Regel eine Strichelung bestimmter Kanten durchführt.

Die dritte Art der Manager in VIDEA nach den Stil- und den Farbmanagern sind die Attributanzeigemanager, wo die darzustellenden Attribute von Knoten und Kanten ausgewählt werden können. Hier wird nach der Auswahl des Knotentyps eine Menge von 'checkboxes' für die zur Verfügung stehenden Attribute dieses Knotentyps angeboten, durch deren Markierung der Nutzer angibt, ob das jeweilige Attribut für alle Instanzen dieses Knoten- oder Kantentyps angezeigt werden soll. Abb. 114 illustriert beispielhaft für den Knotentyp *AVLNode* des AVL-Baums die Auswahl nur eines einzigen Attributs *val*. Bei 'Apply' wirkt sich die jeweilige Auswahl sofort im Graphen aus, 'OK' beendet zusätzlich den Manager, 'Cancel' beendet den Manager mit dem Zustand nach dem letzten 'OK'.

In diesem Kapitel wurde anhand von Beispielen die Vorgehensweise beschrieben, wie VIDEA im Detail vom Dozenten eingesetzt wird – inklusive Generierung, Konfiguration und Benutzung einer VIDEA-Instanz. In Kapitel 9 werden Ergebnisse einer Evaluation zweier VIDEA-Instanzen vorgestellt.

9. Evaluation

Als neues didaktisches Konzept für die Lehre von Algorithmen und Datenstrukturen wurde VIDEA nicht nur konzipiert und realisiert, sondern auch eingesetzt und evaluiert. Die entsprechenden Szenarien und Ergebnisse werden in diesem Kapitel vorgestellt.

Der erste praktische Einsatz von VIDEA erfolgte im Rahmen der Übungen zur Vorlesung „Einführung in die Informatik II“ im Wintertrimester 2003 an der Universität der Bundeswehr München. Dort wurde an zwei Tagen jeweils ca. eine Stunde damit verbracht, ein aus der Vorlesung bekanntes Grundthema mit einer Gruppe von Lernenden anhand einer interaktiven, graphischen Übungsaufgabe mithilfe von VIDEA zu vertiefen. Parallel dazu beschäftigte sich jeweils eine Kontrollgruppe mit einer Papier-Aufgabe desselben Lernstoffes.

Seitdem wurde VIDEA zwar weiterentwickelt (nähere Informationen hierzu folgen in Kapitel 10), aber aus Zeit- und organisatorischen Gründen nur vereinzelt eingesetzt. Deshalb fand bisher auch keine weitere vergleichende Evaluation statt.

Neben der Betreuung der Lernenden beim Einsatz von VIDEA erfolgte eine beobachtende Evaluierung der VIDEA-Gruppe. Nach Beendigung der Aufgabe füllten an beiden Terminen beide Gruppen einen Evaluationsbogen aus, mit dessen Hilfe der Lernfortschritt gemessen werden sollte. Die VIDEA-Gruppe füllte zusätzlich einen Fragebogen aus, der Selbsteinschätzung und Motivation nach der Übung überprüfte. In den folgenden Abschnitten wird diese Vorgehensweise ausführlicher dargestellt und begründet. Außerdem werden die konkreten Erfahrungen beschrieben.

9.1. Begriffsklärung und Ablauf

Wenn im Folgenden von „Evaluation“ gesprochen wird, ist das Ziel keine formale Bewertung von VIDEA mit mathematischer Signifikanz bezüglich spezifischer isolierbarer Kriterien wie objektivem Lernfortschritt der Lernenden oder Nutzbarkeit der VIDEA-Oberfläche. Stattdessen war unser Ziel ein systematischer Einsatz von VIDEA im Evaluationsszenario zur Ermittlung einer tendenziellen Wirkung bezüglich Motivation und Lernerfolg an einer typischen Gruppe von Lernenden. Die Übungssituation entsprach den üblichen Rahmenbedingungen im Übungsbetrieb an der Universität der Bundeswehr München, mit Ausnahme der Nutzung des interaktiven Werkzeugs VIDEA.

Obwohl wegen der geringen Größe der Vergleichsgruppen keine signifikanten Ergebnisse erwartet werden können, verwenden wir den Begriff *Evaluation*, der in Wörterbüchern als *Bewertung* (siehe [DUD86]) oder als *Wertbestimmung* (siehe [Ber96]) definiert wird. Etwas ausführlicher ist die Definition in [Bro78]: *„... die Auswertung einer Erfahrung durch eine oder mehrere Personen; der Begriff ist bes. gebräuchlich in den Sozialwiss., die soziale oder pädagog. Aktionsprogramme auf den in ihnen angestrebten Erfolg hin untersuchen ...*“.

Auch im Bereich der Untersuchung von Computer-Mensch-Schnittstellen wird Evaluation oft eher allgemein definiert, z. B. in [PRS+94]:

- *evaluation*: A process through which information about the usability of a system is gathered in order to improve the system or to assess a completed interface.

- *evaluation method*: A procedure for collecting relevant data about the operation and usability of a computer system.

Zusätzlich zur inhaltlichen Evaluation hatten wir uns entschlossen, auch Daten über den subjektiven Lernfortschritt der Lernenden, Effekte auf die Motivation usw. zu sammeln. Um diese Art der nicht-inhaltlichen Evaluation einordnen zu können, verwendeten wir eine einfache Einteilung der *Mittel* einer Software-Evaluation wie vorgeschlagen in [OR94] in

- *Subjektive Evaluationsmittel*, zu denen beispielweise mündliche Befragung, schriftliche Befragung, lautes Denken zählen und

- *Objektive Evaluationsmittel*, zu denen anwesende Beobachtung und abwesende Beobachtung, wie z. B. bei einer Videoaufzeichnung oder beim 'logfile-recording', gehören.

Beim im ersten Punkt genannten „lauten Denken“ wird der Benutzer aufgefordert, während einer typischen Aufgabenbewältigung seine Überlegungen, Probleme und Handlungs-Alternativen laut vorzutragen.

Wir hatten uns dafür entschieden, sowohl Motivation als auch Lernfortschritt mithilfe jeweils eines Fragebogens zu bewerten. Der Fragebogen zur Ermittlung des Lernfortschritts wurde auch für eine Kontrollgruppe eingesetzt, die VIDEA nicht nutzte. Danach konnte der Lernfortschritt beider Gruppen verglichen werden.

Bezüglich obiger Unterscheidung in subjektive und objektive Evaluationsmittel setzten wir sowohl subjektive Evaluationsmittel durch die Befragungen der Lernenden in Form der Motivationsfragebögen als auch objektive Evaluationsmittel durch die anwesende Beobachtung während der Bedienung von VIDEA ein.

Den inhaltlichen Fragebogen sahen wir als objektive Überprüfung des kollektiven Wissensstandes nach der Übung, ähnlich wie die Überprüfung des individuellen Wissensstandes bei einer Klausur. Der gemessene Wissensstand konnte dann mit demjenigen der Kontrollgruppe verglichen werden.

Für die Lernenden der Kontrollgruppe fand weder eine beobachtende Evaluation noch eine Überprüfung der Motivation statt, sondern eine normale Übung im Rahmen des Übungszyklus für die Vorlesung „Einführung in die Informatik II“. Aus diesem Grund und weil die jeweilige Übungsaufgabe zum Lernstoff passte, der zum fraglichen Zeitpunkt geübt wurde, bemerkten die Teilnehmer dieser Gruppen gar nicht, dass sie Teil einer Evaluation waren. Erst am Ende der Übungsstunde wurden sie gebeten, den inhaltlichen Fragebogen auszufüllen. Insgesamt hatten wir daher sowohl für die AVL-Übung als auch für die Dijkstra-Übung folgende Aufteilung:

- Die VIDEA-Gruppe und die Kontrollgruppe lösten jeweils eine Übungsaufgabe und füllten danach einen Evaluationsfragebogen aus.
- Nur die VIDEA-Gruppe füllte zusätzlich einen Motivationsfragebogen aus und wurde durch eine beobachtende Evaluation begleitet.

Im Folgenden stellen wir zunächst die nicht-inhaltliche Evaluation der beiden VIDEA-Gruppen und die entsprechenden Ergebnisse vor. Danach beschreiben wir die Übungsaufgaben beider Gruppen, die zugehörige inhaltliche Evaluation und ihre Ergebnisse.

Evaluation: AVL-Bäume mit MuSofT

Feedback-Fragebogen

1. Bitte schildern Sie Ihren Eindruck des benutzten Tools (Zutreffendes bitte umkreisen):

   a. Die Benutzung des Tools hat mir – abgesehen von evtl. technischen Problemen - Spaß gemacht:

   Ja | Nein

   b. Die Benutzung des Tools hat mir alles in allem Spaß gemacht:

   Ja | Nein

   c. Besonders gefallen hat mir:

   ....
   ....
   ....

   d. Besonders gestört hat mich:

   ....
   ....
   ....

   e. Folgende Funktionalität fehlt oder sollte verstärkt werden:

   ....
   ....
   ....

   f. Folgende Verbesserungsvorschläge habe ich:

   ....
   ....
   ....

2. Bitte schildern Sie Ihren Eindruck des benutzten Konzeptes (Zutreffendes bitte umkreisen):

   a. Glauben Sie, dass Sie mehr gelernt haben, als bei einer vergleichbaren Übungs-Aufgabe auf dem Papier ?

   Ja | Nein

   b. Als wie nützlich in Punkten von 1-9 würden Sie die interaktive AVL-Übung für sich persönlich bewerten ? (1 = nutzlos, 9 = unglaublich wertvoll)

   1 2 3 4 5 6 7 8 9

   c. Würden Sie eine solche Übung einem Studienkollegen empfehlen ?

   Ja | Nein

   i. Wenn ja : Warum ?
   ii. Wenn nein: Warum nicht ?

   ....
   ....
   ....

3. Bitte schildern Sie Ihre Meinung von Ihrem Wissensstand (bitte eine der Zahlen von 1-9 umkreisen, höher bedeutet besser):

   a. Wie schätzen Sie Ihre momentanen Kenntnisse und Fähigkeiten im Bereich der Algorithmen und Datenstrukturen und verglichen mit Ihren Studienkollegen generell ein?

   1 2 3 4 5 6 7 8 9

   b. Wie schätzen Sie Ihre Kenntnisse von AVL-Bäumen vor dieser Übung ein ?

   1 2 3 4 5 6 7 8 9

   c. Wie schätzen Sie Ihre Kenntnisse von AVL-Bäumen nach dieser Übung ein ?

   1 2 3 4 5 6 7 8 9

Abbildung 115: Der verwendete Motivationsfragebogen

9.2. Nicht-inhaltliche Evaluation

Zur nicht-inhaltlichen Evaluation wurde ein Motivationsfragebogen eingesetzt, der im Folgenden kurz vorgestellt wird, bevor die Ergebnisse präsentiert werden.

9.2.1. Der Motivationsfragebogen

Abb. 115 zeigt den Motivationsfragebogen, der in gleicher Form für die AVL-Übung und für die Dijkstra-Übung – außer natürlich bei expliziter Nennung des Lernthemas der AVL-Bäume bei den Fragen 2 und 3 in Abb. 115 – genutzt wurde, um neben der beobachtenden Evaluation einen Eindruck zu bekommen,

- wie sich der Einsatz von VIDEA auf die Motivation auswirkte,
- welche Eigenschaften von VIDEA besonders motivierend oder demotivierend wirkten und damit, welche Änderungen möglicherweise sinnvoll wären und
- wie sich die subjektive Bewertung des eigenen Wissensstandes der Lernenden durch die Übung veränderte.

Die „eventuellen technischen Probleme“, von denen in Frage 1a die Rede ist, beziehen sich auf temporäre Instabilität der Implementierung der Ereigniswarteschlange in UPGRADE2 und ihrer Nutzung in VIDEA, was zum Zeitpunkt der hier beschriebenen Evaluation zu gelegentlichen Abstürzen führte. Durch die explizite Aufnahme dieser Tatsache in den Motivationsfragebogen hofften wir, emotionale Reaktionen auf eventuelle Stabilitätsprobleme weitgehend von den emotionalen Reaktionen auf die Arbeit mit VIDEA und dessen Konzept zu entkoppeln. Es zeigte sich jedoch sowohl in der beobachtenden Evaluation als auch in der Auswertung der Fragebögen, dass die positiven Reaktionen auf das Arbeiten mit VIDEA nur wenig von den Stabilitätsproblemen beeinträchtigt wurden.

Bei der eigenen Einschätzung des Wissensstandes in Frage 2 und 3 ging es natürlich nicht darum, eine verlässliche Bewertung zu erlangen, sondern, wie eingangs in diesem Kapitel erklärt, um eine rein subjektive Einschätzung, in der sich letztlich wieder die individuelle Motivation und die Akzeptanz der VIDEA-Methode zeigen konnten. Um im schlimmsten Fall nicht nur nichtssagende Aussagen wie „mittelzufrieden“ zu erhalten, wurde eine Kombination aus Ja / Nein-Aussagen, fein-granularer zu bewertender Aussagen und selbst-formuliertem Text der Lernenden genutzt.

Um sowohl Selbst- als auch Fremdbild-Einschätzung der Lernenden mit einzubeziehen, wurde sowohl die Frage nach der Bewertung des eigenen Lernfortschritts als auch nach einer möglichen Empfehlung an Kommilitonen gestellt.

Um herauszufinden, ob die Motivation und damit auch der Lerneffekt durch eine negative Belegung des Lernthemas schon vorher beschränkt waren, wurde die Frage nach der Selbsteinschätzung vor der Übung gestellt, obwohl es uns klar ist, dass die rückwirkende Betrachtung der Lernenden zumindest zum Teil auch vom momentanen Motivations-Zustand beeinflusst werden musste. Die Antworten zeigen eine moderate mittlere Selbsteinschätzung (4.37) in Bezug auf die eigenen Kenntnisse von Algorithmen und Datenstrukturen und eine hohe Einschätzung der Steigerung der eigenen Kenntnisse von AVL-Bäumen (von 4.06 auf 6.81, das ist eine Steigerung um fast 68%).

Benutzung hat Spaß gemacht Abgesehen von ... Alles in allem	 Ja: 100% Nein: 0% Ja: 100% Nein: 0%
Besonders gefallen	Die graphische Darstellung (Balance, Sortierung), optisch nachvollziehbare Rotationen, bessere Darstellung als an der Tafel (auch bei Unbalancierung), die Möglichkeit, selber Dinge ausprobieren zu können, Balance und Höhen sehr anschaulich erkennbar
Besonders gestört	Instabilität, Geschwindigkeit nicht einstellbar
Es fehlt	'undo', Anzeige der internen IDs, zusätzliche Produktionen ('insert' an Knoten etc.), Zoom - Funktion für große Bäume, Refresh-Button, Tastaturkürzel für DeleteNode, Lustiges Maskottchen – z. B. eine Kuh, die durch das Programm führt, 'redo'
Verbesserungsvorschläge	3D-Graphik, Hilfe-Funktion, Tooltips , keinen Unterschied der Reihenfolge des Anklickens bei bei setLeft, setRight, gleiche Knoten im Baum nicht zulassen
Mehr gelernt als bei Papier-Übung	Ja: 100% Nein: 0%
Nützlich in Punkten von 1-9	Durchschnitt: 6.3
Weiterempfehlen	Ja: 100% Nein: 0%
Warum?	Graphisch anschaulich, macht Spaß, verständlich, individuell einstellbar, bessere Lernmethode als Papier (weil übersichtlicher), intensiveres Lernen, selber machen
Eigene Kenntnisse in Pkt von 1-9 v. * Algorithmen und Datenstrukturen vergl. mit Kollegen generell * AVL-Bäumen vor dieser Übung * AVL-Bäumen nach dieser Übung	 Durchschnitt 4.37 Durchschnitt 4.06 Durchschnitt 6.81

Tabelle 1: Ergebnisse des Motivationsfragebogens für die AVL-Übung

9.2.2. Ergebnisse der AVL-Übung

Die beobachtende Evaluation während der AVL-Übung zeigte, dass die Lernenden schnell lernten, das Werkzeug zu bedienen. Sie hatten Spaß und arbeiteten konzentriert. Andererseits lasen sie Fehlermeldungen nicht richtig, und es bestand die Tendenz, das gewünschte Ergebnis unter Umgehung der geforderten Schritte zu erreichen. So wurde etwa mehrfach versucht, den bestehenden AVL-Baum durch Einfügen neuer Knoten zu rebalancieren, anstatt durch explizite Rotationen. Diese Beobachtung floss in die Erstellung unserer zweiten Übungsaufgabe für

den Dijkstra-Algorithmus ein, wo wir weniger Spielraum in der Arbeitsanweisung ließen (siehe Abschnitt 9.4).

Die Auswertung des Motivationsfragebogens zeigte, dass die Lernenden in der VIDEA-Gruppe die graphische Darstellung und die Animation ansprechend und den Lernvorgang intensiv fanden, dass es ihnen Spaß machte und als verständlich empfunden wurde.

Die Antworten der Lernenden sind in Tabelle 1 zu sehen; dabei wurde wörtlich zitiert. Ähnliche Antworten wurden zusammengefasst. Wir wollen deswegen zu jedem Punkt nur kurz etwas sagen. Zunächst war der hohe Motivationsfaktor erfreulich - schließlich würden 100% der Befragten eine Übung mit VIDEA weiterempfehlen. Die Antworten auf die Fragen, was gefallen hat, drehen sich, wie zu vermuten war, vor allem um die Visualisierung an sich. Die Verbesserungsvorschläge bezogen sich auf eine Erhöhung der Stabilität, was mittlerweile in UPGRADE2 und VIDEA umgesetzt wurde, und den Wunsch, interaktiv die Geschwindigkeit von Rotationsoperationen regulieren zu können. Über den Sinn einer solchen Möglichkeit kann man streiten. Die Rotationsoperationen konzeptionell herauszuheben, ist eher nicht sinnvoll. Allerdings wäre es eine Verbesserungsmöglichkeit, in VIDEA *jeder* Operation die Morphing-Geschwindigkeit als zusätzliches Attribut hinzuzufügen. Auf diese Art wäre es neben der gewünschten interaktiven Schnittstelle denkbar, die Geschwindigkeit in Skripten variabel zu gestalten. Die Einstellung der grundsätzlichen Geschwindigkeit von Skripten, ist – wie in Kapitel 8 beschrieben – bereits einstellbar.

Zu den restlichen Wünschen sei gesagt, dass 'undo' / 'redo' in VIDEA vorbereitet und somit leicht zu ergänzen ist, Zoom ohne großen Aufwand zu implementieren wäre, aber im Moment nicht vorgesehen ist. DeleteNode mit der mittleren Maustaste aufzurufen, ist bereits Teil der Funktionalität von VIDEA. Ein Teil der Anregungen erscheint fragwürdig. So soll die Reihenfolge des Anklickens von Knoten beim Aufruf einer Transaktion in voller Absicht von Bedeutung sein, da dadurch die Reihenfolge der Parameter festgelegt wird. Ein anderer Teil ist einfach eine Frage des Designs der VIDEA-Instanz, wie z. B. Nicht-Zulassen gleicher Knotenwerte, Einfügen an der Wurzel des Unterbaums ermöglichen etc.

3D-Graphik und Tooltips sind genauso wenig geplant wie eine ausführliche Hilfefunktion. Allerdings haben wir, wie in manchen Bildschirmausschnitten in dieser Arbeit schon zu sehen war, die problemlos konfigurierbaren Meldungen dazu genutzt, kontextsensitive Vorschläge zu machen und / oder Definitionen zu wiederholen, wie etwa die Meldung der falsch aufgerufenen Linksrechtsrotation in Abb. 85 in Kapitel 7.

Das Feedback mit den Knoten-IDs beruht darauf, dass Knoten in UPGRADE2 und damit auch in VIDEA eine interne ID haben, die z. B. dann zu sehen ist, wenn der Benutzer einen knotenwertigen Parameter nicht durch Selektion an eine Transaktion übergibt, sondern abwartet, bis das Parameterfenster sich öffnet. So ist es vorgekommen, dass ein Lernender das im Knoten angezeigte *val*-Attribut mit dieser Knoten-ID verwechselte. Durch die persönliche Betreuung konnten solche Probleme schnell geklärt werden, aber wir haben diese Anregung als wertvollen Verbesserungsvorschlag in die Liste der sinnvollen Erweiterungen für VIDEA aufgenommen.

Zu empfehlen ist in diesem Zusammenhang, entweder bei der Einführung von VIDEA kurz darauf hinweisen, was es mit diesen internen IDs auf sich hat, da das Verständnis dieses Konzeptes einem Informatik-Lernenden sicher nicht schadet. Oder der Dozent lässt die internen IDs in den Knoten anzeigen, wenn das in der Datenstruktur sinnvoll ist. Eine dritte Möglichkeit ist, die infrage kommenden Transaktionen so zu spezifizieren, dass sie als Parameter Schlüsselattribute

Benutzung hat Spaß gemacht Abgesehen von ... Alles in allem	 Ja: 100% Nein: 0% Ja: 95% Nein: 5%
Besonders gefallen	Graphische Darstellung, farbliche Darstellung, die Überprüfungsfunktionen, ausführliche Beschreibung auf dem Übungsblatt, Verständnistest des nächsten Schrittes
Besonders gestört	Stabilität, Knotenwerte stehen nicht an den Knoten
Es fehlt	Hilfe, selber Knoten erzeugen können, eigentlich nichts
Verbesserungsvorschläge	'undo', Dijkstra-Algorithmus ändern, Name von 'FindNextU' etwas missverständlich Buttonleiste für wichtige Funktionen
Mehr gelernt als bei Papier-Übung	Ja: 100% Nein: 0%
Nützlich in Punkten von 1-9	Durchschnitt: 6.5
Weiterempfehlen	Ja: 100% Nein: 0%
Warum?	Verständlicher, Anschaulichkeit um Klassen höher als auf dem Papier, kein Schreibaufwand, schnelle Auswertung, übersichtliches Abarbeiten der Schritte, ausführliche Erklärung des Algorithmus, spart Zeit, leichtere Vorstellung der Problematik, nicht so abstrakt wie reine Sprache , Arbeitsweise des Algorithmus brennt sich ein, schadet nicht
Eigene Kenntnisse (in Pkt von 1-9) von Algorithmen und Datenstrukturen vergl. mit Kollegen generell	Durchschnitt 5.0

Tabelle 2: Ergebnisse des Motivationsfragebogens für die Dijkstra-Übung

der Knoten, wie z. B. *val* erwarten, und nicht die Knoten selbst.

Der Vorschlag der Kuh, die durch das Programm führt, hat eher für Heiterkeit gesorgt als zu konkreten Plänen für die Weiterentwicklung von VIDEA.

Die Einschätzung des Lerneffekts von Seiten der Lernenden ist merkbar positiv; so gehen sie anfangs von einem leicht unterdurchschnittlichen Wissen im Verhältnis zu ihren Studienkollegen aus, empfinden ihr Wissen nach der Übung aber deutlich besser. Dazu passt auch, dass sie die Nutzung von VIDEA weiterempfehlen würden.

9.2.3. Ergebnisse der Dijkstra-Übung

Die Lernenden, die an der Dijkstra-Übung teilgenommen haben, waren zum Großteil dieselben, die eine Woche zuvor an der AVL-Übung teilgenommen hatten. Da die Anwesenheit bei Übun-

gen nicht verpflichtend vorgeschrieben ist, hatten wir hierauf keinen unmittelbaren Einfluss und beschränkten uns auf die Ankündigung einer Besonderheit in der Übung.

Die beobachtende Evaluation während der Dijkstra-Übung ergab ähnliche Erkenntnisse wie bei der AVL-Übung: Hohe Motivation, keine Probleme mit dem Umgang mit VIDEA und Freude am Ausprobieren, wobei wir diesmal die einzelnen notwendigen Schritte noch unmissverständlicher vorgegeben hatten und – vermutlich deswegen – eine sehr nahe Bearbeitung an den Vorgaben beobachteten. Nach unseren Erfahrungen gilt es an dieser Stelle abzuwägen zwischen viel Spielraum für die Lernenden, mit dem vermuteten Vorteil all der Möglichkeiten der Interaktion generell und des Lernens aus Fehlern im Besonderen, und weniger Spielraum, mit dem Vorteil einer leichteren Koordinierbarkeit der einzelnen Gruppen und einer besseren Vergleichbarkeit der Ergebnisse.

Die Auswertung des Motivationsfragebogens zeigte dieselbe hohe Motivation und fast die gleichen vergebenen Pluspunkte wie bei der AVL-Übung, wobei diesmal die Einfärbefunktion der Knoten des Graphen sehr den Gefallen der Lernenden fand. Ebenso beliebt waren die Überprüfungsfunktionen zum Test des momentanen und nächsten 'U' im Algorithmus, also desjenigen Knotens, der als nächster der Ausgangspunkt der auslaufenden Dijkstra-Welle wird (siehe Kapitel 5, Seite 80). Als störend empfunden wurden am ehesten wieder die auftretenden Abstürze und auch die Tatsache, dass die internen Knoten-IDs nicht am Knoten standen. Da die Beschriftung der Knoten beim Dijkstra-Algorithmus zur kürzesten Wegesuche keine Rolle spielt, ist die Erfüllung dieses Wunsches kein Problem. Wir hatten uns beim Entwurf der Übung für fortlaufende Beschriftungen entschieden und würden in Zukunft einfach die internen Knoten-IDs als Beschriftung verwenden.

Die Verbesserungsvorschläge umfassten wieder die 'undo'-Funktion, von der schon im letzten Abschnitt die Rede war, und diesmal außerdem den Wunsch, den Dijkstra-Algorithmus zu verändern, was allerdings nicht im Sinn einer Lerneinheit ist. Außerdem wurde ein anderer Name für einen PROGRES-Test *FindNextU* gewünscht, der prüft, ob der von den Lernenden identifizierte nächste Knoten wirklich das nächste vom Algorithmus betrachtete *U* ist. Es ist wohl unmissverständlicher, diesen Test z. B. in *testNextU* umzubenennen. Ebenso wurde diesmal eine Leiste mit Schaltflächen für wichtige Funktionen gewünscht, was technisch möglich ist, aber noch nicht getestet wurde.

Wieder glaubten 100% der Lernenden, dass sie mehr gelernt hatten als es mit einer Papier-Übung der Fall gewesen wäre, und waren sich darin einig, diese Übungsform weiterzuempfehlen. Die Nützlichkeit in Punkten von 1 bis 9 wurde durchschnittlich mit 6.5 bewertet. Die Gründe, eine Übung mit VIDEA weiterzuempfehlen, waren zahlreich und positiv und sprechen wohl für sich.

Da die Zeit knapp bemessen war, erwarteten wir in der Frage nach der Selbsteinschätzung diesmal nur die Antwort auf den Vergleich der aktuellen Kenntnisse von Algorithmen und Datenstrukturen generell im Vergleich mit den Kollegen. Denn diesmal waren wir besonders interessiert an der Entwicklung des Selbstbewusstseins gegenüber dem Thema Datenstrukturen und Algorithmen nach zwei VIDEA-Übungen in zwei Wochen. Das Resultat lag mit durchschnittlich 5.0 genau in der Mitte der möglichen Antworten. Die Ergebnisse des Motivationsfragebogens für die Dijkstra-Übung sind in Tabelle 2 zusammengefasst.

Nach dieser Beschreibung der subjektiven Auswirkungen des Einsatzes von VIDEA kom-

men wir in den nächsten Abschnitten zur Messung des objektiven, inhaltlichen Lernfortschritts nach der Nutzung der zwei VIDEA-Instanzen für AVL-Bäume und den Dijkstra-Algorithmus.

9.3. Inhaltliche Evaluation der Übung „AVL-Bäume“

Im Folgenden werden wir zunächst die Übungsaufgaben und anschließend die Ergebnisse der Evaluation der AVL-Übung vorstellen.

9.3.1. Die Übung der VIDEA-Gruppe

Die VIDEA-Gruppe bestand an diesem Termin aus 17 Lernenden. Abb. 116 zeigt das verwendete Übungsblatt der AVL-Übung für die VIDEA-Gruppe. Die Lernenden erhielten zusätzlich Beiblätter mit Illustrationen der Rotationsoperationen in PROGRES, wie sie im Rahmen dieser Arbeit in Kapitel 6 gezeigt worden sind. Die gelösten Aufgaben gehen aus dem Übungsblatt klar hervor und umfassen

- das Beobachten eines Beispielablaufs von Einfüge-Operationen, die in flüssiger Animation alle Rotationen illustrierten,
- das Angeben von Höhe und Balancierungs-Faktor mehrerer innerer Knoten,
- das Finden eines falsch einsortierten Knotens,
- das Prüfen des Ergebnisses,
- das „Reparieren“ des Baumes durch Umhängen des fehlerhaften Knotens,
- das Prüfen und Finden der nun unbalancierten Stelle,
- das Prüfen des Ergebnisses,
- das „Reparieren“ durch manuellen Aufruf geeigneter Rotationsoperationen und
- das Testen des Verhaltens bei Einfügen weiterer Knoten.

Alle wesentlichen Teile dieses Einsatzes von VIDEA wurden bereits bei der Beschreibung der typischen VIDEA-Szenarien in Kapitel 7 vorgestellt.

Die Lernenden arbeiteten in Zweiergruppen an jeweils einem Rechner. Jede Zweiergruppe verbrachte ca. 30 Minuten mit der Bearbeitung der Aufgaben 1-6 in Abb. 116 und 5-10 Minuten mit dem freien Experimentieren wie in Aufgabe 7 beschrieben. Da am Anfang der Stunde die Bedienung von VIDEA ca. 5-10 Minuten lang erklärt und vorgeführt wurde und am Ende der Motivationsfragebogen in ca. 5 Minuten und der Evaluationsfragebogen in ca. 10 Minuten ausgefüllt wurden, hatte die Gruppe insgesamt eine Stunde Zeit. Das ist die gleiche Zeitdauer, die die Kontrollgruppe insgesamt, also für Übung und inhaltlichen Fragebogen zur Verfügung hatte. Das heißt, die Netto-Übungszeit für die Kontrollgruppe war ca. 10-15 Minuten länger als für die VIDEA-Gruppe. Dies wurde teilweise dadurch ausgeglichen, dass die Kontrollgruppe zusätzlich das Löschen in einem AVL-Baum übte.
Im nächsten Unterabschnitt werden wir die Übung der Kontrollgruppe beschreiben, bevor wir den Evaluationsfragebogen und dessen Auswertung vorstellen.

Aufgabe 27: AVL-Bäume mit VIDEA

1. **Lassen Sie den Beispiel-Baum „Example_Tree_1" aufbauen, nach Belieben gleich mit Einfärbung der unbalancierten Knoten. Beobachten Sie, wie nacheinander die Knoten 10, 20, 30 ... 70 eingefügt werden und danach die Knoten 150, 140 ... 80 und schließlich noch 85. Beachten Sie, dass jedes Mal höchstens eine Rotationsoperation notwendig ist, um den Baum wieder zu balancieren.**
   Vergleichen sie die auf dem Beiblatt gegebene Spezifikation der Rotations-Operationen, die jeweils maximal 4 Zuweisungen benötigen.

2. **Geben Sie für die Knoten ‚60' und ‚70' und 2 weitere frei gewählte innere Knoten des Baumes die Höhe und den Balancierungs-Faktor an und prüfen Sie Ihr Ergebnis mit *checkHeight* bzw. *checkBalance*.**

3. **Stellen Sie einen veränderten Baum her durch Aufruf der Aktion „Example_Tree_2": Der neue Baum verletzt für zwei Knoten die *Sortierungs-Eigenschaft* (Definition siehe Beiblatt). Finden Sie die Wurzeln der Teilbäume, die die Sortierungs-Eigenschaft verletzen und prüfen Sie Ihr Ergebnis mit *Color->Unsorted*.**

4. **Nun soll die *Sortierungs-Eigenschaft* hergestellt werden:**
   Ermitteln Sie den Knoten, der zur Herstellung der Sortierungs-Eigenschaft umgehängt werden müsste, entfernen Sie ihn von seinem aktuellen Vater-Knoten (mithilfe von *DeleteInEdges*) und fügen Sie ihn an einer geeigneten Stelle wieder ein (mithilfe von *SetLeft* bzw. *SetRight*). Wenn nun kein Knoten mehr rot dargestellt wird, haben Sie die richtige Stelle gefunden, sonst probieren Sie es noch mal.

5. **Der Baum verletzt nun ebenso die *Balancierungs-Eigenschaft*. Finden Sie die Wurzel des kleinsten Teilbaumes, der die Balancierungs-Eigenschaft verletzt (Definition siehe Beiblatt) und prüfen Sie Ihr Ergebnis mit *Color->Unbalanced*.**

6. **Nun soll die *Balancierungs-Eigenschaft* hergestellt werden:**

   Ermitteln Sie die dafür notwendige Rotationsoperation und führen Sie sie aus, so dass der Baum wieder sortiert ist.

7. **Fügen Sie mithilfe von *insert* weitere Knoten ein und spielen Sie mit den Rotationsoperationen.**

Beiblatt: Der VIDEA-Prototyp

Bedienung:

- Knoten können mit der linken Maustaste selektiert und / oder bewegt werden.
- Um einen zweiten Knoten in die Selektion mit aufzunehmen, selektiert man diesen bei gedrückter Shift-Taste.
- Wenn ein Kommando einen oder mehrere zusätzliche Parameter erwartet, können diese in einer extra Eingabezeile gefüllt werden (Bestätigung mit RETURN-Taste oder Mausklick auf grünes Häkchen)
- Werden Knoten als Parameter erwartet, können diese durch die momentane Selektion übergeben werden.

Erklärung der Kommandos:

Menü	Unter-Menü	Aktion	Parameter
File	Close	Beendet die Anwendung	Keiner
BinTree	Insert	Fügt einen neuen Knoten in den Baum ein	1 Integer
Rotate	LeftRightRotate	Führt Links-Rechts-Rotation aus	1 Knoten
	LeftRotate	Führt Links-Rotation aus	1 Knoten
	RightLeftRotate	Führt Rechts-Links-Rotation aus	1 Knoten
	RightRotate	Führt Rechts-Rotation aus	1 Knoten
Exercise	CheckBalance	Prüft Balance-Faktor eines Knotens	1 Knoten, 1 Integer
	CheckHeight	Prüft Höhen-Wert eines Knotens	1 Knoten, 1 Integer
	DeleteInEdges	Löscht einlaufende Kanten e. Knotens	1 Knoten
	SetLeft	Erzeugt Links-Kante zwischen 2 Knoten	2 Knoten
	SetRight	Erzeugt Rechts-Kante zwischen 2 Knoten	2 Knoten
Tree1	Example_Tree_1	Demonstriert den Aufbau eines AVL-Baums	Keiner
Tree2	Example_Tree_2	Demonstriert Probleme in einem AVL-Baum	Keiner
Color	Unbalanced	Zeichnet unbalancierte Knoten (s.u.) rot	Keiner
	Unsorted	Zeichnet unsortierte Knoten (s.u.) rot	Keiner

Definitionen:

Ein Knoten heißt *unbalanciert*, wenn die Höhen seiner 2 unmittelbaren Teilbäume sich um einen Wert größer als 1 unterscheiden, d.h. wenn sein Balance-Faktor nicht im Bereich –1, 0, 1 liegt.

Ein Knoten heißt *unsortiert*, wenn es in seinem linken unmittelbaren Teilbaum einen Knoten mit größerem Wert oder in seinem rechten unmittelbaren Teilbaum einen Knoten mit kleinerem Wert gibt.

Abbildung 116: Ausschnitte aus dem Übungsblatt für die AVL-Baum-Übung

9.3.2. Die Übung der Kontrollgruppe

Die Kontrollgruppe bestand an diesem Termin aus 16 Lernenden. Abb. 117 zeigt das Übungsblatt der Kontrollgruppe für AVL-Bäume. Da die Lernenden an der Universität der Bundeswehr München die imperative Programmiersprache Ada lernen, wurden die AVL-Bäume in Ada umgesetzt. Es handelt sich um einen anderen Tutor als in der VIDEA-Gruppe, weil beide Übungen wegen der herrschenden Rahmenbedingungen zeitgleich durchgeführt werden mussten. Es wurde zunächst das Einfügen in einen AVL-Baum von Hand geübt und danach – zusammen mit Berechnung des Balancefaktors – in Ada implementiert. Dann übten die Lernenden das Löschen in einem AVL-Baum, was die VIDEA-Gruppe nicht übte. Wenn man nun voraussetzt, dass keine der beiden Gruppen a priori ein besseres Verständnis von AVL-Bäumen hatte und der Effekt einer Ein-Stunden-Übung sichtbare Auswirkungen in einer inhaltlichen Evaluation wie unserer zeigt, müsste sich die beschriebene ungleiche Verteilung der Aufgaben im Resultat der inhaltlichen Evaluation auswirken. Tatsächlich ist bei der Auswertung der Ergebnisse des inhaltlichen Fragebogens zu sehen, dass die Kontrollgruppe beim Einfügen in AVL-Bäume ungefähr gleich gut abschnitt, beim Löschen aber wesentlich besser. Es sei an dieser Stelle noch erwähnt, dass der Tutor der Kontrollgruppe bei der Durchführung der Übung den Fokus sehr stark auf das Lernen der Sortierungseigenschaft der AVL-Bäume legte, was aus dem Übungsblatt selbst nicht ersichtlich ist.

Alles in allem fand die Übung der Kontrollgruppe mit mehr Unterstützung durch den Tutor statt, während wir uns in der VIDEA-Gruppe auf die beobachtende Evaluation und evtl. Fragen der Lernenden zur Bedienung von VIDEA beschränkten und bezüglich der inhaltlichen Lehre weitgehend VIDEA vertrauten.

9.3.3. Der Evaluationsfragebogen

Im inhaltlichen Evaluationsfragebogen für beide Gruppen wurden systematisch abgeprüft:

- Die Einordnung des Themas der AVL-Bäume: Ihre Definition, wo werden sie gebraucht?
- Statisches Wissen: Definition der Höhe, der Sortierungs- und Balancierungseigenschaft.
- Strukturelles Wissen: Wie viele Rotationen sind nach einem Einfügen maximal nötig? Wo werden Knoten eingefügt, um bestimmte Rotationen zu erzwingen? Hat die Sortierungseigenschaft etwas mit Balancierungen zu tun?
- Praktisches Wissen: Wie rebalanciert man? Wie fügt man ein? Wie löscht man?

Diese Einteilung ist zwar willkürlich getroffen und fließend, vermittelt uns aber doch den Eindruck, dass wir das Thema unter den gegebenen Rahmenbedingungen gründlich genug evaluiert haben. Der Fragebogen ist in Abb. 118 und 119 zu sehen.

9.3.4. Die Auswertung des inhaltlichen Fragebogens

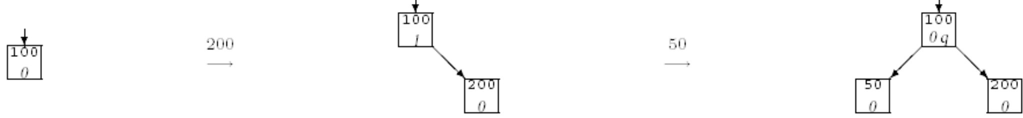
Die Auswertung der Fragebögen zeigte, dass beide Gruppen ein gutes Verständnis von AVL-Bäumen gewonnen hatten, wobei die VIDEA-Gruppe z. B. besser darin abschnitt, die Höhe eines Beispiel-Baums anzugeben oder die Stelle, wo ein Beispiel-Baum unbalanciert ist. Die Stärken der Kontrollgruppe lagen bei der Beantwortung von Fragen wie derjenigen nach der Höhe des leeren Baums oder, was beim Löschen eines Knotens passiert – ein Thema, das die VIDEA-

Aufgabe 27 (Ada: AVL-Bäume)

Gegeben Sei folgende Definition für einen AVL-Baum für Integer:

```
type AVL_Node;
type AVL_Tree is access AVL_Node;
type AVL_Node is record
                   Elem        : Integer;
                   -- evtl. weitere oder generisch
                   Left,Right  : AVL_Tree;
                   Balance     : Integer range -1..1;
                 end record;
```

a) Simulieren Sie das Einfügen der Zahlen 150, 100, 50, 130, 140, 110, 160, 105, 120, 125 in AVL-Bäume. Schreiben Sie an jeden Knoten zusätzlich die jeweilige *Balance*, z.B.:



b) Was sind die Bedingungen für die 4 Rotationsarten beim Einfügen? Erstellen Sie eine Prozedur zum Einfügen eines bislang nicht enthaltenen Elements in einen AVL-Baum.

c) Simulieren Sie jetzt auf die gleiche Weise das Löschen der Zahlen 140, 160, 110, 105, 130, 125, 150, 100, 50, 120. Erstellen Sie eine Prozedur zum Löschen eines Elements aus einem AVL-Baum.

d) **Zusatz:** Schreiben Sie eine Druckprozedur, die die AVL-Bäume ausdruckt. Der in Teilaufgabe (a) entstandene AVL-Baum soll dann etwa so aussehen:

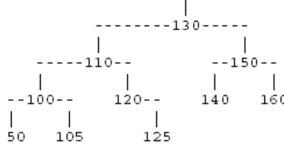
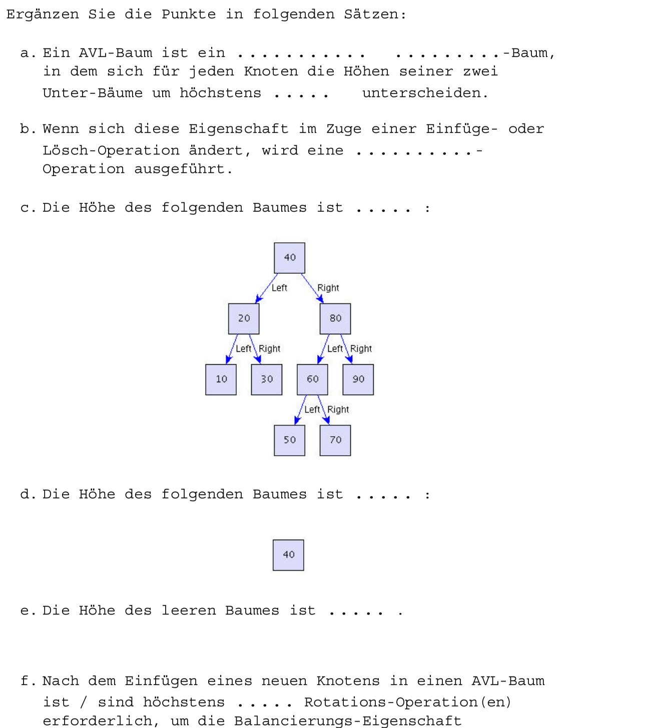


Abbildung 117: AVL-Übung der Kontrollgruppe

Gruppe ausgeklammert hatte. Auch wenn durch die eng beieinander liegenden Resultate und die Kleinheit der Gruppen kein signifikantes Ergebnis hergeleitet werden kann, kann ein Vorteil der VIDEA-Gruppe bei Fragen beobachtet werden, die mit der visuellen Darstellung am Bildschirm zusammenhingen. Eine Stärke der anderen Gruppe war feststellbar bei eher theoretischen Fragen, wie derjenigen nach der Höhe des leeren Baums. Ausnahmen davon gab es bei Teilthemen, die unterschiedlich intensiv geübt worden waren, wie dem Löschen im Baum oder dem Finden unsortierter Stellen.

Wie in der Einleitung zu diesem Kapitel gesagt, sind die weiter unten gezeigten Ergebnisse in Tabelle 3 vorsichtig zu interpretieren. Die '>'-Zeichen, die an manchen Stellen in den Auswertungen auftauchen, bedeuten, dass die jeweilige Gruppe aus nachvollziehbaren Gründen die jeweilige Frage besser beantworten hätte können als in den Prozent-Zahlen ausgedrückt. Gründe waren fehlende Bearbeitung einer zugehörigen Aufgabe, nachträglich nachzuvollziehender, abweichender Wissensstand der Kontrollgruppe oder nur teilweise korrekte Erklärungen, die wir



Evaluation: AVL-Bäume

Test-Aufgaben für die Überprüfung der gelernten Inhalte

1. Ergänzen Sie die Punkte in folgenden Sätzen:

a. Ein AVL-Baum ist ein ....-Baum, in dem sich für jeden Knoten die Höhen seiner zwei Unter-Bäume um höchstens ..... unterscheiden.

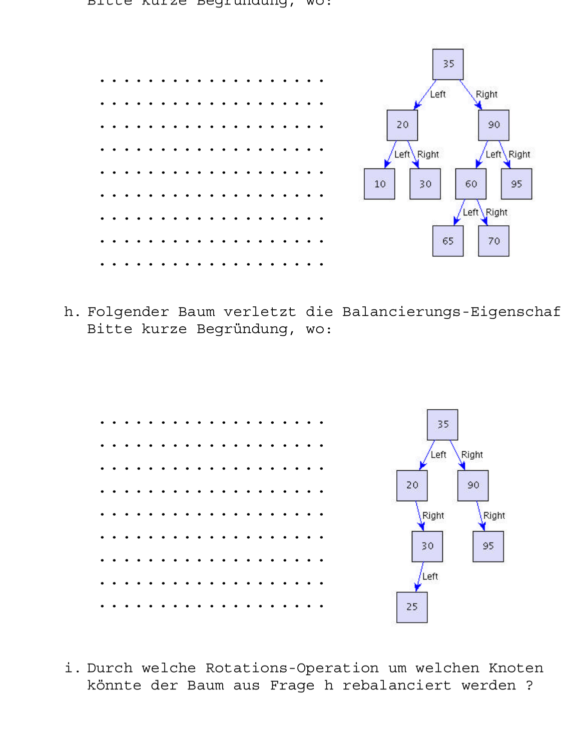
b. Wenn sich diese Eigenschaft im Zuge einer Einfüge- oder Lösch-Operation ändert, wird eine ....-Operation ausgeführt.

c. Die Höhe des folgenden Baumes ist ..... :

d. Die Höhe des folgenden Baumes ist ..... :

e. Die Höhe des leeren Baumes ist ..... .

f. Nach dem Einfügen eines neuen Knotens in einen AVL-Baum ist / sind höchstens ..... Rotations-Operation(en) erforderlich, um die Balancierungs-Eigenschaft wiederherzustellen.



g. Folgender Baum verletzt die Sortierungs-Eigenschaft. Bitte kurze Begründung, wo:

....
....
....
....
....
....
....
....
....

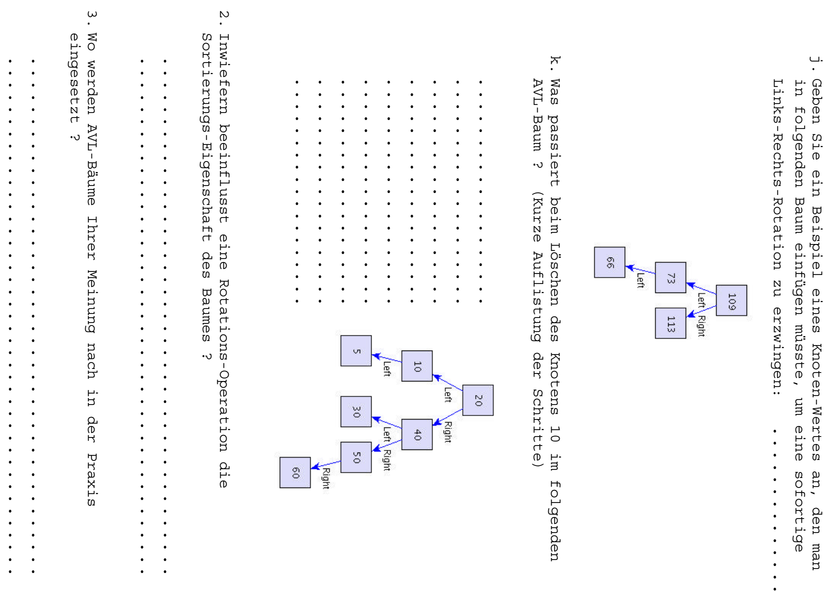
h. Folgender Baum verletzt die Balancierungs-Eigenschaft. Bitte kurze Begründung, wo:

....
....
....
....
....
....
....
....
....

i. Durch welche Rotations-Operation um welchen Knoten könnte der Baum aus Frage h rebalanciert werden ?

....
....

Abbildung 118: Inhaltlicher AVL-Evaluationsbogen


j. Geben Sie ein Beispiel eines Knoten-Wertes an, den man in folgenden Baum einfügen müsste, um eine sofortige Links-Rechts-Rotation zu erzwingen: ....

k. Was passiert beim Löschen des Knotens 10 im folgenden AVL-Baum ? (Kurze Auflistung der Schritte)

2. Inwiefern beeinflusst eine Rotations-Operation die Sortierungs-Eigenschaft des Baumes ?

3. Wo werden AVL-Bäume Ihrer Meinung nach in der Praxis eingesetzt ?

Abbildung 119: Inhaltlicher AVL-Evaluationsbogen, Fortsetzung

nicht eindeutig mit Punkten bewerten konnten. Es wurde bereits erwähnt, dass die Kontrollgruppe das Thema der Sortierungseigenschaft besonders intensiv geübt hatte. Damit sind die auf diese Frage bezogenen Ergebnisse des Vergleichs der beiden Gruppen nicht aussagekräftig und wurden nur der Vollständigkeit halber dokumentiert.

Die Frage nach dem praktischen Einsatz von AVL-Bäumen wurde aus Gründen der Untersuchung des Verständnisses des Themas gestellt, ist aber natürlich für VIDEA irrelevant.

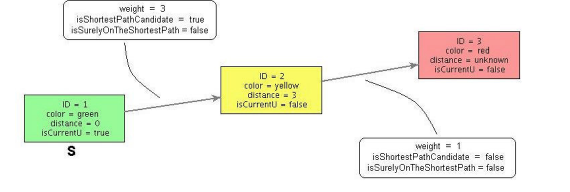
9.4. Inhaltliche Evaluation der Übung „Dijkstras kürzeste Wegesuche“

Der Dijkstra-Algorithmus zur kürzesten Wegesuche wurde im Rahmen der Übungsaufgaben eingeführt. Es wurde versucht, durch vorhergehende, genaue Absprache der Tutoren die Präsentation dieses Themas in beiden Gruppen so ähnlich wie möglich zu gestalten. Aus den gleichen Gründen wie bei der AVL-Übung war es nicht möglich, den gleichen Tutor für VIDEA-Gruppe und Kontrollgruppe zu verwenden. Wie schon bei den AVL-Bäumen, verließen wir uns in der VIDEA-Gruppe in dieser Hinsicht vor allem auf das Übungsblatt und die Führung durch VIDEA selbst, während in der Kontrollgruppe eine aktivere, inhaltliche Betreuung stattfand. Zum Beispiel wurden vom Tutor aktiv Hilfestellungen gegeben, wenn beim Lösen einer Aufgabe Fehler erkennbar waren.

Dijkstra's kürzeste Wegesuche mit VIDEA

Der Dijkstra-Algorithmus ist neben dem Floyd-Warshall-Algorithmus, den Sie bereits kennen, eine weitere Möglichkeit, kürzeste Wege in einem Graphen zu suchen. Dijkstra ist dabei meistens schneller, findet aber die Wege nur von einem gegebenen Startknoten aus (bei nicht-negativen Kanten-Gewichten).

Die Datenstrukturen Node und Edge kann man sich gemäß dem folgenden Bild deklariert denken:



Die Arbeitsweise des Algorithmus:

- Von einem gegebenen Startknoten S aus werden kürzeste Wege zu allen anderen Knoten berechnet (die Ergebnisse stehen dann in distance) und nebenbei der kürzeste Pfad von S zu jedem Knoten (im Attribut isSurelyOnTheShortestPath der auf dem jeweiligen Wege liegenden Kanten).
- Die Menge der Knoten des Graphen zerfällt zu jedem Zeitpunkt in 3 Teilmengen, die durch 3 Farben dargestellt werden:
  - o Die Knoten, die noch nicht betrachtet wurden, sind rot (da dies am Anfang die gesamte Menge ist, werden alle Knoten rot initialisiert),
  - o die Knoten, für die es schon eine Schätzung gibt, werden gelb eingefärbt
  - o und die Knoten, zu denen bereits der kürzeste Weg (von S aus) gefunden wurde, werden grün eingefärbt (im Attribut color).
- Alle gelben Knoten sind zu jedem Zeitpunkt in einer Prioritätswarteschlange gespeichert, aufsteigend sortiert nach der bisher bekannten Distanz zum Startknoten S.
- Zu Beginn werden alle Knoten als rot und mit unbekannter Distanz initialisiert und nur die Distanz des Startknotens wird auf 0 gesetzt, seine Farbe auf gelb. Schließlich wird S in die Prioritätswarteschlange gestellt.
- In jedem Schritt des Algorithmus passiert Folgendes: Der gelbe Knoten mit der kürzesten bekannten Distanz, im folgenden U genannt (gespeichert im Knotenattribut isCurrentU, wird aus der Warteschlange entnommen, grün eingefärbt und alle Ziele V auslaufender Kanten von U werden besucht mit den folgenden Aktionen:
  - o Ist V rot, wird er gelb eingefärbt und in die Prioritätswarteschlange übernommen.
  - o War oder wurde V nun gelb, wird geprüft, ob seine bisher bekannte Distanz durch die neue Kante verbessert werden kann; wenn ja, wird die Distanz angepasst und die Kante von U nach V als isSurelyOnTheShortestPath markiert.
- Das passiert so lange, bis entweder alle Knoten bearbeitet (= grün) sind, also die Prioritätswarteschlange leer ist oder bis ein bestimmter Knoten, dessen kürzeste Distanz gesucht war, bearbeitet ist.

In Pseudocode:

```
While es gibt noch gelbe Knoten loop
    • nimm den gelben Knoten mit der kleinsten Distanz
      (mithilfe der Prioritätswarteschlange)
    • nenne ihn U
    • färbe ihn grün,
    • setze seine einlaufende Kante, die Kandidat für den kürzesten Weg ist,
      als sicher auf dem kürzesten Weg
    • For all von U auslaufenden Kanten E loop
        o nenne den Zielknoten V
        o wenn V rot ist, setze ihn gelb
          (und füge ihn in die Prioritätswarteschlange ein)
        o wenn V nun gelb ist und sich seine berechnete Distanz über die
          neue Kante E verbessern lässt, setze seine Distanz neu und nimm
          E als neuen Kandidaten für den kürzesten Weg zu V
    end loop;
end loop;
```

Aufgabe: Dijkstra mit VIDEA

1. Lassen Sie den vordefinierten Graphen PredefinedGraph1 erzeugen. Startknoten ist natürlich der Knoten 1, da er als einziger keine einlaufenden Kanten hat. Überlegen Sie sich jetzt erst mal theoretisch, welches der kürzeste Pfad zu den Knoten 4 und 6 ist:

   Kürzester Pfad zu Knoten 4 ist: ....
   Kürzester Pfad zu Knoten 6 ist: ....

Abbildung 120: Dijkstra VIDEA-Uebung

2. Was müsste jetzt Ihrer Meinung nach in der Visualisierung passieren, wenn der Algorithmus die Initialisierung durchführt?

....

Testen Sie Ihre Vermutung durch Ausführen der Aktion FindRoot.

3. Was müsste jetzt passieren, wenn die Hauptschleife des Algorithmus, also der 'Schritt', einmal durchlaufen wird?

....

Testen Sie Ihre Vermutung durch Ausführen der Aktion NextStep.

4. Welcher Knoten ist nun U? ....

Testen Sie Ihre Vermutung durch Ausführen von FindCurrentU.

5. Welcher Knoten müsste im nächsten Schritt U werden? ....

Testen Sie Ihre Vermutung durch Ausführen von FindNextU.

6. Gehen Sie nun in der gleichen Weise weiter vor und überlegen Sie sich, was als nächstes passieren müsste, bevor Sie es mit NextStep überprüfen. Welche Bedeutung haben wohl die grün markierten Kanten?

....

7. Löschen Sie den Graphen mit DeleteAll und lassen Sie den nächsten Graphen PredefinedGraph2 entstehen. Sollten Sie das Layout geändert haben, führen Sie zumindest einmal doOneLayout aus. Gehen Sie wieder mit FindRoot und NextStep vor. Wann könnten Sie aufhören, wenn nur der kürzeste Pfad zu Knoten 3 gesucht wäre? War es bis zu diesem Zeitpunkt nötig, den Knoten 4 (und seine einlaufenden und auslaufenden Kanten) überhaupt in Betracht zu ziehen? Beachten Sie die Farbe des Knotens 4!

....

8. Wenn Sie noch Zeit haben, spielen Sie mit den Graphen PredefinedGraph3 und PredefinedGraph4.

Für ein schöneres Layout von PredefinedGraph4 ziehen Sie einfach dessen Knoten 7 außerhalb des Knotens 1 und schalten danach auf manuelles Layout.

Beiblatt: Der VIDEA - Prototyp

Bedienung:

- Knoten können mit der linken Maustaste selektiert und /oder bewegt werden.
- Das Layout ist auf automatisch voreingestellt. Bei Nicht-Gefallen kann es jederzeit auf manuell umgestellt werden, um nur bei Bedarf doOneLayout zu verwenden.
- Wenn ein Kommando einen Parameter erwartet, kann dieser in einer extra Eingabezeile gefüllt werden (Bestätigung mit RETURN-Taste oder Mausklick auf grünes Häkchen)
- Werden Knoten als Parameter erwartet, können diese durch die momentane Selektion übergeben werden.

Erklärung der Kommandos:

Menü	Unter-Menü	Aktion	Parameter
File	Close	Beendet die Anwendung.	Keiner
Layout	automaticLayout doOneLayout manualLayout	In jedem Schritt wird ein Layout berechnet. Nur einmal -jetzt- wird ein Layout berechnet. Der Benutzer macht sein Layout selbst.	Keiner
DeleteGraph	DeleteAll	Löscht den ganzen Graphen.	Keiner
Exercise	FindCurrentU FindNextU	Finden Sie den Knoten, der noch 'U' ist. Finden Sie den Knoten, der gleich zu 'U' wird.	1 Knoten 1 Knoten
Dijkstra	FindRoot NextStep	Initialisiert den Graphen gemäß Dijkstra. Führt einen Schritt gemäß Dijkstra aus.	Keiner
Predefined	PredefinedGraph1 PredefinedGraph2 PredefinedGraph3 PredefinedGraph4	Baut den jeweiligen Beispiel-Graphen auf.	Keiner

Abbildung 121: Dijkstra VIDEA-Uebung, Fortsetzung

1 Aufgabe ...(Ada: Shortestpath mit Dijkstra)

Der Dijkstra-Algorithmus ist neben dem Floyd-Warshall-Algorithmus, den Sie bereits kennen, eine weitere Möglichkeit, kürzeste Wege in einem Graphen zu suchen. Dijkstra ist dabei meistens schneller, findet aber die Wege nur von einem gegebenen Startknoten aus (bei nicht-negativen Kanten-Gewichten).
Im folgenden sei immer ein klar definierter Startknoten gegeben durch die Eigenschaft, als einziger keine einlaufenden Kanten zu besitzen.

Gegeben seien folgende Typ-Deklarationen für einen Graphen:

```
V : Positive := 20;

type Color is (White, Gray, Black);

type Node;
type Vertex is access Node;

type EdgeElem;
type Edge is access EdgeElem;
type EdgeElem is record
    TargetNode : Vertex;
    Weight : Natural;
    NextEdge : Edge  := null;
end record;

type Node is record
    NodeID : Positive range 1..V;
    NodeColor : Color := White;
    Distance : Natural := 0;
    Pred : Vertex := null;
    OutgoingEdges :  Edge := null;
end record;
```

Der Dijkstra-Algorithmus geht folgendermaßen vor:

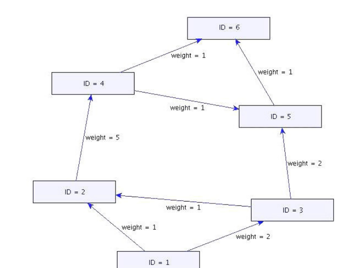
- Von einem gegebenen Startknoten S aus werden kürzeste Wege zu allen anderen Knoten berechnet (die Ergebnisse stehen dann in *Distance*) und nebenbei der kürzeste Pfad von S zu jedem Knoten (auffindbar durch Zurücklaufen durch die *Pred*-Verweise).
- Die Menge der Knoten des Graphen zerfällt zu jedem Zeitpunkt in 3 Teilmengen, die durch 3 Farben dargestellt werden:
  1. Die Knoten, die noch nicht betrachtet wurden, sind rot (da dies am Anfang die gesamte Menge ist, werden alle Knoten rot initialisiert),
  2. die Knoten, für die es schon eine Schätzung gibt, werden gelb eingefärbt

1

  3. und die Knoten, zu denen bereits der kürzeste Weg (von S aus) gefunden wurde, werden grün eingefärbt (im Attribut *NodeColor*),
- Alle gelben Knoten sind zu jedem Zeitpunkt in einer Prioritätswarteschlange gespeichert, aufsteigend sortiert nach der bisher bekannten Distanz zum Startknoten S.
- Zu Beginn werden alle Knoten als rot und mit quasi-unendlicher Distanz initialisiert (siehe obige Typ-Deklaration von *Node*) und nur die Distanz des Startknotens wird auf 0 gesetzt, seine Farbe auf gelb. Schließlich wird S in die Prioritätswarteschlange gestellt.
- In jedem Schritt des Algorithmus passiert Folgendes:

  Der gelbe Knoten mit der kürzesten bekannten Distanz, im folgenden U genannt, wird aus der Warteschlange entnommen, grün eingefärbt und alle Ziele V auslaufender Kanten von U werden besucht mit den folgenden Aktionen:
  - Ist V rot, wird er gelb eingefärbt und in die Prioritätswarteschlange übernommen.
  - War oder wurde V nun gelb, wird geprüft, ob seine bisher bekannte Distanz durch die neue Kante verbessert werden kann; wenn ja, wird die Distanz angepaßt und U als Vorläuferknoten von V eingetragen.
- Das passiert solange, bis entweder alle Knoten bearbeitet (= grün) sind, also die Prioritätswarteschlange leer ist oder bis ein bestimmter Knoten, dessen kürzeste Distanz gesucht war, bearbeitet ist.

1. Gegeben sei folgender Graph:



2

Abbildung 122: Dijkstra-Übung der Kontrollgruppe

Frage	VIDEA-Gruppe	Kontrollgruppe
	richtige Antworten in %	
Ein AVL-Baum ist ein ...-...-Baum	74	91
Die Höhen zweier Teil-Bäume unterscheiden sich höchstens um ...	100	94
Wenn sich die Balance ändert, wird eine ...-Operation ausgeführt	94	94
Die Höhe des folgenden Baumes <1> ist...	94	75
Die Höhe des folgenden Baumes <2> ist...	94	88
Die Höhe des leeren Baumes ist...	65	75
Nach dem Einfügen eines Knotens sind höchstens ... Rotationen erforderlich	88	>0
Folgender Baum verletzt die Sortierungs-Eigenschaft. Bitte kurze Begründung, wo	82	100
Folgender Baum verletzt die Balancierungs-Eigenschaft. Bitte kurze Begründung, wo	94	88
Durch welche Rotation um welchen Knoten könnte der Baum aus der vorigen Frage rebalanciert werden?	>47	>31
Geben Sie ein Beispiel eines Knoten-Wertes an, den man im folgenden Baum einfügen müsste, um eine sofortige Links-Rechts-Rotation zu erzwingen	71	75
Was passiert beim Löschen des Knotens 10 im AVL-Baum (Kurze Auflistung der Schritte):	>29	88
Inwieweit beeinflusst eine Rotations-Operation die Sortierungs-Eigenschaft des Baums?	65	94
Wo werden AVL-Bäume Ihrer Meinung nach in der Praxis eingesetzt?	47	56

Tabelle 3: Ergebnisse der Auswertung der Kontroll-Fragebögen zum inhaltlichen Verständnis beider Gruppen bezüglich AVL-Bäumen. Die mit '>' gekennzeichneten Zahlen bedeuten, dass es externe Faktoren gab, durch die die Zahlen möglicherweise verringert wurden (siehe Beschreibung im Text).

9.4.1. Die Übung der VIDEA-Gruppe

Die VIDEA-Gruppe bestand an diesem Termin aus 19 Lernenden. Die Übungsaufgaben der VIDEA-Gruppe waren, wie in Abb. 120 und Abb. 121 zu sehen, u. a.:

- Theoretisches Überlegen kürzester Pfade von einem gegebenen Startknoten zu zwei anderen Knoten,
- Überlegen und darauf folgendes Testen von Fragestellungen der folgenden Art:
  - Was passiert am Anfang des Algorithmus, also etwa, welcher Knoten wird als erster betrachtet?

Simulieren Sie unter der Bedingung, dass Knoten 1 der Startknoten ist, schrittweise (gemäß dem oben definierten 'Schritt') die Ausführung des Algorithmus, bis für alle Knoten der kürzeste Pfad von S aus bekannt ist. Überlegen Sie sich besonders in jedem Schritt, welcher Knoten gerade U ist.

2. Welche Knoten sind gelb, wenn Knoten 5 gerade U ist ?
3. Wann könnte man eigentlich anhalten, wenn der kürzeste Weg zu Knoten 6 gesucht ist ?
4. Schreiben Sie unter Verwendung der oben gegebenen Typdeklarationen eine Prozedur

Dijkstra(S : in out Vertex)

die als Parameter den Startknoten S bekommt und gemäß dem beschriebenen Algorithmus für alle Knoten die kürzesten Wege berechnet und diese in den Knoten-Attributen *Distance* und *Pred* ablegt.
Stützen Sie sich dabei auf die Prozeduren

procedure Enqueue (Q : in out PriorityNodeQueue; V : Vertex)

zum Einfügen des Knotens V in Q und

DequeueMinimal (Q : in out PriorityNodeQueue; U : out Vertex)

zum Entnehmen des kleinsten Knotens U aus Q in einer gegebenen Datenstruktur *PriorityNodeQueue* ab.

3

Abbildung 123: Dijkstra-Übung der Kontrollgruppe, Fortsetzung

– Welche neue Farbgebung entsteht nach dem nächsten Schritt, welcher Knoten ist nun der Ausgangsknoten des nächsten Schrittes?

Danach wurde anhand dreier weiterer Beispielgraphen u. a. gefragt, wann genau der Algorithmus halten würde, wenn der kürzeste Weg zu einem angegebenen Knoten gesucht wäre.

9.4.2. Die Übung der Kontrollgruppe

Die Kontrollgruppe bestand an diesem Termin aus 14 Lernenden. Abb. 122 und 123 zeigen die Übungsaufgabe der Kontrollgruppe. Ähnlich wie schon bei der Kontrollgruppen-Aufgabe für AVL-Bäume sind auch hier wesentliche Teile der Datenstruktur in Ada vorgegeben. Nachdem das Prinzip des Algorithmus erklärt wurde, erhielten die Lernenden die Chance, sich den Ablauf des Algorithmus in den Aufgaben 1,2 und 3 nochmal selbst ähnlich klarzumachen, wie er auch in VIDEA visualisiert würde. Danach konnten sie ihn in Aufgabe 4 als Ada-Prozedur `Dijkstra` implementieren.

Evaluation: Dijkstra's Shortest Path

Test-Aufgaben für die Überprüfung der gelernten Inhalte

1. Wie ist der kürzeste Pfad in folgendem Graphen

   a. Zum Knoten 3 ? ....

   b. Zum Knoten 5 ? ....

2. In welcher Reihenfolge würden die Knoten des obigen Graphen im Rahmen der Berechnung des Dijkstra-Algorithmus (bis zum Ende) als 'U' besucht werden ?

   ....

3. Kann es passieren, dass der Algorithmus manche Knoten (und also auch Kanten) gar nicht besuchen muss, wenn die kürzesten Wege zu einem bestimmten Knoten gesucht sind ?
   Bitte kurze Begründung.

   ....
   ....
   ....
   ....

4. Warum ist es nicht sinnvoll, die Prioritätswarteschlange in Ada als geordnete Liste zu implementieren ?

   ....
   ....
   ....

5. Nun soll der Algorithmus minimal erweitert werden, um bei der Suche nach einem kürzesten Pfad zu einem bestimmten Zielknoten Z bei sehr starker Vernetzung des Graphen nicht zu oft in die falsche Richtung zu laufen.

   Dazu soll jeder Knoten 2 zusätzliche Attribute

   X_Position :Float; Y_Position :Float;

   bekommen, die zu Beginn initialisiert werden, und es sollen bevorzugt solche Kanten traversiert werden, die eher in Richtung auf die Koordinaten des Zielknotens Z hin führen.

   a. Welche Aktion in der Hauptschleife des Dijkstra-Algorithmus müsste dafür ein anderes Ergebnis liefern ?

   ....
   ....

   b. Welche Funktion bzw. Aktion muss also geändert werden ?

   ....
   ....

   c. Beschreiben Sie kurz die notwendige Änderung:

   ....
   ....

   d. Ist es in Ihrer Lösung immer noch gewährleistet, dass der jeweils kürzeste Weg gefunden wird ?

   ....

   Wenn ja, warum? Wenn nein, warum nicht ?

   ....
   ....
   ....

Abbildung 124: Inhaltlicher Dijkstra-Evaluationsbogen

Frage	VIDEA-Gruppe	Kontrollgruppe
	richtige Antworten in %	
Wie ist der kürzeste Weg in folgendem Graphen zum Knoten 3?	89	71
zum Knoten 5?	89	79
In welcher Reihenfolge würden die Knoten des obigen Graphen im Rahmen der Berechnung des Dijkstra-Algorithmus (bis zum Ende) als 'U' besucht werden?	100	79
Kann es passieren, dass der Algorithmus manche Knoten (und also auch Kanten) gar nicht besuchen muss, wenn die kürzesten Wege zu einem bestimmten Knoten gesucht sind?	100	93
Bitte kurze Begründung:	>47	>7
Warum ist es nicht sinnvoll, die Prioritätswarteschlange in Ada als geordnete Liste zu implementieren?	53	0
Nun soll der Algorithmus minimal erweitert werden, um bei der Suche nach einem kürzesten Pfad zu einem bestimmten Zielknoten Z bei sehr starker Vernetzung des Graphen nicht zu oft in die falsche Richtung zu laufen. Dazu soll jeder Knoten zwei zusätzliche Attribute XPosition und YPosition vom Typ Float bekommen, die zu Beginn initialisiert werden, und es sollen bevorzugt solche Kanten traversiert werden, die eher in Richtung auf die Koordinaten des Zielknotens Z hin führen. Welche Aktion in der Hauptschleife des Dijkstra-Algorithmus müsste dafür ein anderes Ergebnis liefern?	63	36
Welche Funktion bzw. Aktion muss also geändert werden?	>32	>14
Beschreiben Sie kurz die notwendige Änderung:	58	50
Ist es in Ihrer Lösung immer noch gewährleistet, dass der jeweils kürzeste Weg gefunden wird?	47	21
Wenn ja, warum? Wenn nein, warum nicht?	47	29

Tabelle 4: Ergebnisse der Auswertung der Kontroll-Fragebögen zum inhaltlichen Verständnis beider Gruppen bezüglich Dijkstra-Algorithmus. Die mit '>' gekennzeichneten Zahlen bedeuten, dass es externe Faktore n gab, durch die die Zahlen möglicherweise verringert wurden (siehe Beschreibung im Text).

9.4.3. Der Evaluationsfragebogen

Der Evaluationsfragebogen für beide Gruppen, also VIDEA- und Kontrollgruppe, ist in Abb. 124 zu sehen. Es wurde den Lernenden die Möglichkeit gegeben,

- ihr Verständnis über den Ablauf und die Arbeitsweise des Algorithmus anhand eines Beispiels in den Fragen 1 und 2,
- ein etwas tieferes Verständnis des Algorithmus in den Fragen 3 und 4 und
- die Fähigkeit für eine kleine Transferleistung in Frage 5

unter Beweis zu stellen.

9.4.4. Die Auswertung des inhaltlichen Fragebogens

Die Auswertung der Fragebögen zeigte ein leicht bis viel besseres Abschneiden der VIDEA-Gruppe in allen Fragen. Besonders deutlich zeigte sich das

- beim Finden der kürzesten Pfade in zwei Beispiel-Graphen,
- bei der Frage und Begründung, ob es sein kann, dass bestimmte Knoten nicht besucht werden müssen und
- bei der Frage, welche Grundaktionen des Algorithmus in Ada wie zu ändern wären, wenn zusätzliche Koordinateninformationen in den Knoten gespeichert werden würden.

Warum die Kontrollgruppe so schlecht abschnitt in Fragen nach der Implementierung der Prioritäts-Warteschlange, ließ sich nicht mehr rekonstruieren.

Trotz allem, was über die fehlende Signifikanz gesagt wurde, haben wir uns über das deutlich bessere Abschneiden der VIDEA-Gruppe in unserer zweiten Evaluation gefreut, in der wir bereits die Erfahrungen aus der AVL-Evaluation einbringen konnten. Das gilt in besonderem Maße, weil die Kontrollgruppe einerseits – wie schon bei der AVL-Übung – etwas mehr Zeit zur reinen Bearbeitung der Übungsaufgabe zur Verfügung hatte; andererseits, weil die Kontrollgruppe im Gegensatz zur VIDEA-Gruppe eine Ada-Aufgabe übte und wir deshalb bei den Ada-Fragen ein besseres Abschneiden der Kontrollgruppe erwartet hatten.

9.5. Zusammenfassung

Es wurden zwei VIDEA-Instanzen an jeweils einem Termin evaluiert. Dabei bearbeitete an beiden Terminen jeweils eine VIDEA- und eine Kontrollgruppe Aufgaben zum gleichen Lernthema auf unterschiedliche Weise, nämlich die eine Gruppe mit VIDEA-Unterstützung und die andere ohne.

Beide Gruppen füllten danach einen inhaltlichen Fragebogen ähnlich einer Probeklausur aus, dessen Auswertung wir vorstellten: Bei den AVL-Bäumen stellten wir vergleichbare Lernfortschritte bei beiden Gruppen fest mit der Tendenz der VIDEA-Gruppe zu besseren Ergebnissen bei am Bildschirm geübten Fragestellungen. Bei der Übung zur kürzesten Wegesuche auf Graphen mit Dijkstra erzielte die VIDEA-Gruppe generell die besseren Ergebnisse.

Zusätzlich werteten wir an beiden Terminen für die VIDEA-Gruppe einen Motivationsfragebogen aus und führten eine beobachtende Evaluation durch. Beide lieferten Feedback für die Weiterentwicklung von VIDEA. Außerdem ließen sie eine hohe Motivation beim Einsatz von VIDEA erkennen.

10. Abschließende Bemerkungen

In dieser Arbeit wurde der neuer Ansatz VIDEA für die Lehre von Algorithmen und Datenstrukturen vorgestellt. VIDEA verbindet diejenigen Eigenschaften multimedialer Lehrwerkzeuge, die nach heutigem Kenntnisstand einen maximalen Lerneffekt ermöglichen, in den Bereichen Visualisierung, Interaktivität und aktivem Lernen auf konsequente und neuartige Weise. VIDEA ist entstanden im Rahmen von MuSofT (siehe [DE02]) und kann über [VID04] bezogen werden.

Um die Probleme ähnlich zielführender, bestehender Ansätze zu vermeiden, die oft auf einen kleinen Bereich von Datenstrukturen festgelegt sind oder dem nutzenden Dozenten unnötigen Mehraufwand bei der Erstellung oder Anpassung neuer Lerneinheiten verursachen, wurde in unserem Ansatz eine zweistufige Parametrisierbarkeit des Systems entworfen:

Auf der ersten Stufe werden zu lehrende Datenstrukturen und Algorithmen inklusive Operationen visuell spezifiziert. Aus dieser Spezifikation wird in einem automatisierten Verfahren eine Lerneinheit generiert, die bereits ablauffähig ist und deren Verhalten auf der zweiten Stufe durch Konfiguration weiter konkretisiert wird. Auf diese Weise können durch Anpassung existierender Spezifikationen schnell neue Lerneinheiten mit den in VIDEA garantierten Eigenschaften zur Erreichung eines hohen Lerneffekts erstellt werden. Ebenso können durch Konfiguration einer bestehenden Lerneinheit weitere Varianten dieser Lerneinheit in anderen Lernszenarien genutzt werden.

In diesem abschließenden Kapitel fassen wir unsere Arbeit zusammen, geben einen Überblick über das erstellte System und die erreichten Ziele und einen Ausblick auf mögliche Weiterentwicklungen des vorgestellten Ansatzes.

10.1. Zusammenfassung

Das Ziel bei der Entwicklung von VIDEA war die Erforschung der Möglichkeiten eines Ansatzes der multimedialen Lehre von Algorithmen und Datenstrukturen, der zwei Arten von Erkenntnissen umsetzt: Erkenntnisse über die Gründe mangelnder Nutzung multimedialer Lehrwerkzeuge und Erkenntnisse über wesentliche didaktische und lernpsychologische Eigenschaften solcher Systeme.

Nachforschungen haben ergeben, dass multimediale Lehrwerkzeuge in einem viel geringeren Maße im Lehrbetrieb von Universitäten eingesetzt werden, als es in Anbetracht positiver Forschungsergebnisse in diesem Bereich und einer positiven Grundhaltung der entsprechenden Dozenten eigentlich zu erwarten wäre. Das trifft sogar im Bereich der Lehre von Algorithmen und Datenstrukturen zu, der sich besonders für die animierte Darstellung der Lerninhalte anbietet. Umfragen unter den entsprechenden Dozenten ergeben als Hauptgrund den Mehrfachaufwand, den ein Dozent hat, um verschiedene Datenstrukturen multimedial zu unterrichten. Denn nutzt er eine Visualisierung, die wesentliche Fähigkeiten wie Interaktivität, flüssige Animationen usw. abgestimmt auf die jeweilige Datenstruktur besitzt, dann muss er für die Lehre einer anderen, vielleicht sogar ähnlichen Datenstruktur auf eine neue Visualisierung zurückgreifen. Oder er entscheidet sich für ein Rahmenwerk zur Erstellung von Animationen. Diese bieten heute allerdings bei weitem nicht die interaktiven und visuellen Möglichkeiten, die gebraucht werden, um die möglichen Resultate im Lernfortschritt zu erzielen.

Die Antwort von VIDEA hierauf ist die Entwicklung und Nutzung eines Grundszenarios,

das es dem Dozenten ermöglicht, sich bei der Erstellung neuer Instanzen der Visualisierungsumgebung zunächst auf die Spezifikation der Datenstruktur bzw. des Algorithmus und des didaktischen Konzeptes zu konzentrieren. Liegt bereits eine VIDEA-Instanz für eine ähnliche Datenstruktur vor, kann unabhängig von Erwägungen über Visualisierung und Oberfläche die zugehörige Spezifikation auf einem hohen Abstraktionsniveau geändert und daraus eine neue VIDEA-Instanz generiert werden. In der generierten Instanz können nun Details zur Visualisierung konfiguriert werden – im Allgemeinen ohne Code schreiben oder übersetzen zu müssen.

Dieses Grundszenario wurde angereichert mit denjenigen Möglichkeiten, die die größten Chancen auf eine Maximierung des Lerneffekts bieten. Dazu wurden aus verschiedenen Bereichen, u. a. aus allgemeinen Erkenntnissen der Lernpsychologie und aus Ergebnissen von Studien über die Wirksamkeit von Algorithmenanimationen, Eigenschaften multimedialer Lehrwerkzeuge zusammengetragen, die als wesentlich identifiziert worden sind. Zur Umsetzung dieser Eigenschaften in einem Ansatz, der das skizzierte Grundszenario unterstützt, wurde ein ausführlicher Katalog technischer Anforderungen entwickelt. Die Anforderungen wurden in Kategorien aufgeteilt und priorisiert. Durch die Nutzung einer großen Menge an Informationen und Details aus einem breiten Feld von Quellen wurde die Anforderungsliste so aussagekräftig, dass sie auch zukünftigen Forschungen als Bewertungsschema für hochinteraktive Datenstruktur-Animationssysteme dienen kann.

Bestehende Ansätze und Werkzeuge, die zur Spezifikation und Visualisierung von Algorithmen und Datenstrukturen dienen können, wurden analysiert und anhand der wichtigsten Punkte der erstellten Anforderungsliste bewertet. Dabei zeigte es sich, dass keines der bestehenden Systeme alleine die wesentlichen Forderungen erfüllen konnte. Eine Kombination von Werkzeugen und Paradigmen konnte allerdings die meisten Anforderungen erfüllen und wird deshalb im Rahmen der Implementierung des VIDEA-Ansatzes genutzt, der speziell auf die Umsetzung des genannten Anforderungskatalogs abzielt: UML-Klassen- und UML-Objektdiagramme und Graphtransformationen werden für die Spezifikation von Datenstrukturen und Algorithmen eingesetzt, während die Animation mithilfe einer Java-basierten Graphbibliothek erfolgt.

Insgesamt wurde eine Methodik entwickelt und vorgeführt, um

- mit Graphtransformationen Datenstrukturen zu spezifizieren, ohne die Interaktivität oder die Konfigurierbarkeit der späteren Lerneinheiten zu beschränken,
- algorithmische Abläufe in Schritte verschiedener Schrittweite zu zerlegen, die es erlauben, für die Lehre geeignete interaktive und visuelle Schnittstellen anzubieten,
- diese Schritte ebenso mit Graphtransformationen zu spezifizieren,
- Verständnistests mit Graph-Tests zu spezifizieren,
- Teile des didaktischen Konzepts, wie angebotene Operationen, angebotene Informationen, verbotene Situationen, erlaubte, aber markierte Fehlersituationen, Reaktionen auf Fehler etc. sinnvoll in die Spezifikation mit aufzunehmen,
- die entstehenden Spezifikationen automatisiert in VIDEA-Instanzen umzusetzen, so dass per Konstruktion die wichtigsten Forderungen des Anforderungskatalogs erfüllt sind,
- in einem neu entwickelten, einfachen, aber mächtigen Konzept Anzeigeschemata, Farbschemata und Meldungen in den VIDEA-Instanzen zu verwalten und zu konfigurieren,

- die entstehenden VIDEA-Instanzen in externe Fahrpläne und ggf. Lernschritte einzubinden und
- die entstehenden VIDEA-Instanzen in eine Reihe identifizierter, nach Interaktivität geordneter Einsatzszenarien einzuordnen.

Fahrpläne ergänzen eine generierte VIDEA-Instanz und dienen dazu, die Nutzung der VIDEA-Instanz durch den Lernenden zu planen und in gewissen Bahnen zu lenken. Es hat sich gezeigt, dass es nicht sinnvoll ist, diese Arbeitsanweisung in das Werkzeug mit aufzunehmen, da hierbei fast zwangsläufig die Interaktivität beeinträchtigt wird. Nach unseren Erfahrungen ist es vorzuziehen, dass der Dozent den Fahrplan extern erstellt, die wesentlichen Abläufe in die Spezifikation der VIDEA-Instanz einfließen lässt und – je nach verwendetem Lernszenario – einen Teil des Fahrplans dem Lernenden z. B. als Übungsblatt zur Verfügung stellt. Damit behält der Lernende die Kontrolle über seine Aktivitäten und kann z. B. Fehler machen, aus denen er wiederum lernen kann. Obwohl ein solch freies Experimentieren seine Berechtigung hat, da es immer wieder als besonders wertvoll für den Lerneffekt genannt wird, kann es vom Dozenten leicht vermieden werden, dass der Lernende zu weit von den vorgesehenen Lernpfaden abweicht; dies kann z. B. erreicht werden, indem Synchronisationspunkte im Fahrplan existieren, die das Erreichen bestimmter Punkte durch den Lernenden überprüfbar machen, wie schriftlich zu fixierende Antworten auf kontextspezifische Fragen zu einem speziellen Szenario oder die Diskussion der Antworten verschiedener Gruppen bei der Leitung einer Übungsgruppe durch einen Tutor.

VIDEA überlässt also die Erstellung eines Fahrplans prinzipiell dem Dozenten, macht aber Vorschläge für sinnvolle Elemente von Fahrplänen und ihre sinnvolle Zusammensetzung. Dieses Konzept der Erstellung von Fahrplänen hat sich bisher bewährt, wie sich insbesondere auch in einer beobachtenden Evaluation bei der Nutzung von VIDEA durch zwei Studentengruppen zeigte.

Um die Mächtigkeit des Ansatzes zu untersuchen, wurden eine Reihe von Einsatzszenarien für mit VIDEA generierte Lerneinheiten formuliert, die viele in der Literatur immer wieder gewünschte didaktische Konstellationen umsetzen. Die vier interessantesten Szenarien wurden herausgegriffen und konzeptionell ausführlich beschrieben. Es handelt sich dabei um die Szenarien:

- Vorführen im Rahmen einer Präsenzveranstaltung.
- Passives Betrachten vorgegebener Abläufe.
- Interaktives Erforschen und Aufgaben-Lösen.
- Ferngesteuerte Animation eigener Programme der Lernenden.

Anhand dieser Szenarien wurde der konkrete Einsatz mehrerer VIDEA-Instanzen in der Lehre gezeigt. Bei den zu lehrenden Algorithmen und Datenstrukturen handelt es sich um AVL-Bäume, die doppelt verkettete Ringliste, den Dijkstra-Algorithmus zur kürzesten Wegesuche und den Kruskal-Algorithmus zur Berechnung minimaler Spannbäume.

In einer empirischen Untersuchung des Lerneffekts bei Nutzung zweier VIDEA-Instanzen im Vergleich zu zwei Vergleichsgruppen, die an einer klassischen Übung mit Tafel und Papier teilnahmen, wurde demonstriert, dass mit VIDEA erzeugte hochinteraktive Datenstruktur-Animationen einen Lernvorteil bringen können.

Insgesamt konnte der gewählte Ansatz bei der bisherigen Nutzung die in ihn gestellten Erwartungen größtenteils erfüllen:

Die Spezifikationen von Datenstrukturen und ihren Operationen kann durch die prinzipiell visuelle Form auf weitgehend natürliche Art erfolgen, und es wurde gezeigt, dass sie leicht wiederverwendbar und anpassbar sind.

Die in unserer Anforderungsliste geforderte Entkopplung der konfigurierbaren Eigenschaften (Farbschemata, Stilschemata, Meldungen, Skripte, angezeigte Attribute, Layoutalgorithmen, Aufteilung auf Menüs usw.) von der Spezifikation der Datenstruktur konnte erfolgreich umgesetzt werden. Alle Eigenschaften können in XML-Konfigurationsdateien konfiguriert werden.

Im Falle der Notwendigkeit der Erstellung eines neuen Layoutalgorithmus zeigte es sich, dass zwar mächtige Grundbausteine in Form von Java-Klassen zur Verfügung gestellt werden konnten, aber die Implementierung von Java-Code notwendig bleibt, da Layoutalgorithmen individuell an die jeweilige Datenstruktur angepasst werden müssen.

Für die Spezifikation von Datenstrukturen und Algorithmen und die Verarbeitung der Graphtransformationen wird PROGRES [Sch94] verwendet. Die Erzeugung von Menüs und Schnittstellen zu Java werden von einem eigenen Rahmenwerk namens UPGRADE2 [JSW00] zur Verfügung gestellt. Für die Darstellung von Layoutalgorithmen und die flüssige Animation allgemein wird yFiles [WEK03] eingesetzt. Die Werkzeuge werden durch neu entwickelten Java-Code verbunden und in VIDEA genutzt.

Die Implementierung des VIDEA-Rahmenwerks besteht aus einer Reihe von Java-Klassen und zugehörigen XML-Konfigurationsdateien. Für die Oberfläche der VIDEA-Manager wurde Swing verwendet, für die Darstellung der Datenstrukturen yFiles. Mithilfe der Reflection-API des Java Development Kits wurde ein im Moment nicht in UPGRADE2 vorhandenes Delegation-Pattern für die Behandlung eigener Fehlermeldungen simuliert. Zusätzlich wurde Einfluss auf die weitere Entwicklung von UPGRADE2 genommen.

Für die notwendigen Konfigurationsdateien wurde XML-Code erstellt. Außerdem wurden mehrere PROGRES-Spezifikationen für Algorithmen und Datenstrukturen entwickelt, die alle Aspekte der in Kapitel 7 ausführlich dargestellten wichtigsten VIDEA-Szenarien unterstützen. Bei den durchgeführten Teilaufgaben handelt es sich u. a. um folgende:

- Der Standardprototyp zur Verarbeitung einfacher Graphen wurde um Basisfunktionalität für die Nutzerschnittstellen von Lernenden und Dozenten erweitert.
- Es wurde die Möglichkeit flüssiger Animationen hinzugefügt.
- UPGRADE2 selbst wurde um die Möglichkeit der Ausgabe konfigurierbarer Meldungen – auch in Situationen, in denen kein Fehler vorliegt – erweitert.
- Der in UPGRADE2 genutzte Graph-Prototyp wurde komplett von JViews entkoppelt.
- Zu konfigurierende Eigenschaften wurden vom Standardverhalten entkoppelt.
- Es wurde eine neue Knoten-, Kanten- und Selektions-Verwaltung entworfen und implementiert.

Das Hauptfenster in VIDEA ist ein Graphbrowser, der syntaxgesteuertes Erstellen und Bearbeiten von Datenstrukturen ermöglicht. Die hierfür zur Verfügung stehenden Operationen werden als Graphtransformationen spezifiziert, über zwei XML-Dateien konfiguriert und über das Menü oder eine Toolbar ausgeführt. Die Übergabe von Parametern an Operationen erfolgt im Falle knoten- oder kantenwertiger Parameter durch Markierung der zu übergebenden Objekte in der passenden Reihenfolge oder durch ein sich öffnendes Parameter-Fenster, in dem z. B. ganzzahlige Parameter in einem Textfeld oder boolesche Parameter durch Anklicken einer 'checkbox' übergeben werden können.

Zusätzlich können abkürzende Aktionen für das Erzeugen und Löschen von Knoten eines ausgezeichneten Knotentyps vor Start der VIDEA-Instanz festgelegt werden, indem jeweils der Name einer PROGRES-Programmeinheit für die jeweilige Aktion konfiguriert wird.

Der Benutzer kann beliebig zwischen manuellem Layout und automatischem Layout, das spezifisch für die zu lernende Datenstruktur ist, wechseln, falls der Dozent dieses Verhalten in der Konfiguration der VIDEA-Instanz vorgesehen hat. Die graphischen Elemente der Datenstruktur, also Knoten und Kanten, können mit der Maus beliebig markiert und verschoben werden.

Bei Nutzung des manuellen Layouts bleiben die Positionen aller Teile der Datenstruktur erhalten, bis sie vom Benutzer neu festgelegt werden; sonst wird bei Ausführung der nächsten Operation die Position aller graphischen Elemente neu berechnet und gesetzt.

Alternativ zum manuellen Aufbauen einer Datenstruktur kann dies auch durch ein vom Dozenten vorbereitetes Skript geschehen, das schrittweise einen ganzen Ablauf ausführt und in einstellbarer Geschwindigkeit visualisiert. Der Aufruf des Skripts erfolgt ebenso wie der Aufruf einzelner Operationen über das Menü.

Um Fehlersituationen besonders darzustellen, können vom Dozenten Farbschemata in einer der Konfigurationsdateien vorgegeben werden, die jedem Knotentyp in Abhängigkeit von den Werten seiner Attribute verschiedene Farben zuweisen – im einfachsten Fall eine Fehlerfarbe für fehlerhafte Situationen. Dasselbe gilt für Kanten, bei denen zusätzlich zwischen den Stilen *durchgezeichnet* und *gestrichelt* unterschieden wird.

Genauso wie die Farb- und Stil-Schemata können die angezeigten Attribute vordefiniert werden, und zwar für Knoten eine beliebige Kombination der verfügbaren Attribute, und für Kanten ein ausgezeichnetes Attribut.

Die Verwaltung der Farbschemata, Stilschemata und der angezeigten Attribute wird von so genannten VIDEA-Managern durchgeführt, die jeweils auch eine graphische Schnittstelle anbieten. Der Dozent kann für jeden Manager entscheiden, ob die graphische Schnittstelle für den Lernenden verfügbar sein soll oder nicht. Falls ja, kann der Nutzer die graphische Oberfläche des jeweiligen Managers zur Laufzeit der VIDEA-Instanz aus dem Menü heraus aufrufen und dort jederzeit alle Einstellungen ändern, die vom Dozenten vorkonfiguriert worden sind.

Durch die beschriebenen Fähigkeiten einer VIDEA-Instanz können alle in Kapitel 3 als sinnvoll identifizierten Ausprägungen des vorgestellten Grundszenarios unterstützt werden. Zugrunde liegen dabei die folgenden technischen Fähigkeiten:

- Graphartige Datenstrukturen können mit korrektem Layout dargestellt werden.
- Vordefinierte Abläufe können als Skripte definiert und abgespielt werden.

- Vordefinierte Abläufe können schrittweise, vorwärts oder rückwärts abgespielt werden (dieser Punkt ist in Vorbereitung).
- Es sind vom Dozenten vordefinierte, vom Lernenden frei erzeugbare und zufällige Eingabedatenmengen und Szenarien möglich.
- Die vordefinierten Abläufe können prinzipiell beliebig oft auf beliebig viele Instanzen von Datenstrukturen angewandt werden.
- Animationen können zu schrittweisen Veränderungen führen oder flüssig ablaufen.
- Es können harte Constraints definiert werden, die vom Lernenden nicht verletzt werden können, wobei Fehlermeldungen möglich sind.
- Es können weiche Constraints definiert werden, die der Lernende verletzen darf, deren Verletzung sich aber auf die Färbung der Datenstruktur oder auf die vom System zurückgegebenen Meldungen oder auf beide auswirkt.
- Es können Verständnistests spezifiziert und genutzt werden, die situationsbedingt vom Lernenden zu beantworten sind und konfigurierbare Meldungen in Abhängigkeit der Richtigkeit der Antwort zurückgeben. Das können zum Beispiel Fragen zu Messgrößen der aktuell angezeigten Datenstruktur oder zu nächsten Schritten des Algorithmus sein.
- Für jedes Attribut der Datenstruktur kann einzeln bestimmt werden, ob es angezeigt oder versteckt wird.
- Es existieren graphische Schnittstellen, in denen der Benutzer jederzeit zur Laufzeit des Systems die angezeigten Attribute von Knoten und Kanten und die genutzten Stil- und Farbschemata ändern kann. Diese Schnittstellen können individuell vom Dozenten angeboten werden oder nicht.
- Es kann im Hintergrund in andere Farbschemata oder Stilschemata gewechselt werden, ohne für den Lernenden ersichtlich zu machen, in welches.
- Es ist möglich, im Hinblick auf ein vorher definiertes Lernziel den Lernenden fortlaufend zu unterstützen, indem hilfreiches Feedback vom System in Form von ermunternden oder warnenden Meldungen, durch evtl. geändertes Layout und durch die selektive Nutzung von Fehlerfarben gegeben wird.
- Es kann eine freie Simulation eines Algorithmus in Schritten unterschiedlicher Granularität in einer vom Lernenden festgelegten Geschwindigkeit und Reihenfolge erfolgen.
- Es kann ein völlig freies Experimentieren des Lernenden durch beliebige Auswahl irgendwelcher angebotener Operationen auf der Datenstruktur erfolgen, wobei das System jederzeit durch Färbung der Datenstruktur oder durch Meldungen intelligent reagiert.
- Es können Programme der Studierenden in jeder Programmiersprache, die CORBA unterstützt, ferngesteuert animiert werden, wodurch ein einfaches visuelles Debugging erreicht werden kann.

Nachdem die wichtigsten Ergebnisse und Erkenntnisse, die sich aus der Entwicklung von VIDEA ergeben haben, informell beschrieben wurden, wird VIDEA im folgenden Abschnitt auf die Erfüllung derjenigen Anforderungen hin untersucht, die sich durch Auswertung einer Reihe von Studien ergeben haben.

10.2. Bewertung von VIDEA

In diesem Abschnitt prüfen wir VIDEA systematisch auf die Erfüllung zweier Listen von Anforderungen an multimediale Lernsoftware. Wir beginnen mit der in dieser Arbeit vorgeschlagenen Anforderungsliste aus Kapitel 3. Danach untersuchen wir, inwieweit VIDEA die Forderungen einer Anforderungliste, die für Rahmenwerke einer etwas anderen Zielsetzung entworfen wurde, erfüllen kann.

10.2.1. Bewertung von VIDEA anhand des eigenen Bewertungsschemas

In Kapitel 3 dieser Arbeit wurde ausgehend von Ergebnissen bisheriger Studien über die Wirksamkeit multimedialer Lehre eine ausführliche Liste von Anforderungen erarbeitet.

Da selbst die in Kapitel 4 vorgestellte kondensierte Teilliste von keinem der bestehenden, untersuchten Ansätze erfüllt werden konnte, wurde VIDEA entwickelt. In diesem Abschnitt ziehen wir Bilanz und prüfen VIDEA auf die Erfüllung der Anforderungen der ursprünglichen ausführlichen Liste.

Jede Kategorie ist absteigend in Bezug auf die Wichtigkeit ihrer Punkte sortiert (siehe Kapitel 3). In den Tabellen ist neben der Nummer der Anforderung ihr ggf. abgekürzter Text und der Grad der Erfüllung aufgeführt. Es gibt die Erfüllungsgrade JA (erfüllt), TEILS (teilweise erfüllt), MÖGL (Erfüllung ist konzeptionell leicht möglich, jedoch bisher nicht implementiert) und NEIN (nicht erfüllt).

Alle mit TEILS oder MÖGL markierten Anforderungen der Tabellen 5 bis 9 werden im Folgenden näher erläutert:

Anforderung		Erfüllung
SP 1	Die Spezifikationssprache besitzt eine formale Semantik ...	JA
SP 2	Es gibt eine Werkzeugunterstützung für die Erstellung der Spezifikation und die Codegenerierung	JA
SP 3	Die Spezifikationssprache ist natürlich, nach Möglichkeit auch bekannt und einfach	JA
SP 4	Die Spezifikationen sind einfach erweiterbar für kleine Änderungen an der Datenstruktur	JA
SP 5	Schnittstellenoperationen sind einfach zu spezifizieren bzw. nachzuspezifizieren	JA
SP 6	Invarianten sind einfach zu spezifizieren	JA
SP 7	Verbote in Abhängigkeit vom Zustand der Datenstruktur, z. B. verletzten Invarianten, sind einfach zu spezifizieren	JA
SP 8	Das Spezifizieren neuer Beispielszenarien ist einfach	JA
SP 9	Die Schritte eines Algorithmus sind in verschiedenen Granularitäten einfach zu spezifizieren	TEILS

Tabelle 5: Spezifikations-Anforderungen

Anforderung		Erfüllung
LO 1	Das Verhalten in Abhängigkeit vom Zustand der Datenstruktur ist steuerbar	JA
LO 2	Ausgabe- (Fehler-) Meldungen als Reaktionen auf Interaktionen sind möglich	JA
LO 3	Vom Nutzer eingegebene Datenmengen oder Instanzen einer Datenstruktur können zur Laufzeit verarbeitet werden	JA
LO 4	Es gibt eine interne Schnittstelle für den Aufruf von Operationen auf der Datenstruktur	JA
LO 5	Eine automatische Überprüfung von Invarianten ist möglich (z. B. bei interaktiven Änderungen durch den Benutzer)	JA
LO 6	Gerade bearbeitete, inkonsistente, wichtige Teile der Datenstruktur können ... einfach identifiziert werden	JA
LO 7	Die Laufzeitlogik ist fähig zu 'undo' / 'redo'	JA
LO 8	Ein schrittweises Weiterschalten der Zustände der Datenstruktur ist möglich	JA
LO 9	Es gibt eine Schnittstelle für eine beispielspezifische Taktung (nächste Schritte der Animation) von außen	JA
LO 10	Es gibt einen logischen Mechanismus, um Vorhersagen in einem bestimmten Kontext verlangen und testen zu können	JA
LO 11	Eine parallele Anzeige der zum aktiven Teil der Datenstruktur oder des Algorithmus passenden Programmteile ist synchronisierbar mit der Visualisierung	TEILS
LO 12	Zufällige Datenmengen, zufällige Instanzen einer Datenstruktur und zufällige Szenarien können zur Laufzeit generiert werden	JA
LO 13	Das Laden und Speichern verschiedener Beispiel-Szenarien ist möglich	JA
LO 14	Die Animation von Algorithmen kann zwischen verschiedenen Detaillierungsstufen wechseln	JA
LO 15	Eine Kenntnis über die Struktur von Datenstrukturen ist codierbar, etwa die Struktur von Graphen oder Bäumen	JA
LO 16	Es gibt eine Schnittstelle für den Start von Skripten	JA
LO 17	Es gibt eine Schnittstelle für den Aufruf von Befehlen in einer Skriptsprache	JA
LO 18	Verschiedene Geschwindigkeiten sind für die Animation möglich	JA
LO 19	Ein Fahrplan kann von der Laufzeitumgebung überwacht werden (ggf. mit Aufruf von Meldungen)	TEILS
LO 20	Es gibt einen Mechanismus für das Wechseln von Farbschemata zur Laufzeit	JA
LO 21	Es sind Strukturen für die Verarbeitung neuer Formen der Interaktion vorhanden	NEIN
LO 22	Ein Protokoll aufgerufener Operationen kann angezeigt werden	MÖGL

Tabelle 6: Anforderungen an die logische Verarbeitung

Anforderung		Erfüllung
O 1	Die entstehenden Animationen sind flüssig (kontinuierlich)	JA
O 2	Es gibt eine Schnittstelle für die Eingabe von Vorhersagen …	JA
O 3	Es gibt eine Schnittstelle für Fragen / Rückmeldungen an Studierende	JA
O 4	Es gibt eine Möglichkeit, Schnittstellenoperationen aufzurufen	JA
O 5	Explorative Lernszenarien mit genügend Wahlmöglichkeiten des Benutzers … sind möglich	JA
O 6	Das interaktive Erzeugen eigener Beispiele und eigener Datenmengen ist einfach	JA
O 7	Vordefinierte oder zur Laufzeit definierte Farb- und Stilschemata können dargestellt werden	JA
O 8	Es gibt einen Graphbrowser mit Layout …	JA
O 9	Graphartige Datenstrukturen … können dargestellt werden	JA
O 10	Das Einbinden von Layoutalgorithmen mit automatischer Berechnung eines stabilen Layouts ist möglich	JA
O 11	Eine Abspiel-Schnittstelle ist vorhanden oder zumindest einfach zu implementieren	JA
O 12	Animationsfilme können an beliebiger Stelle angehalten werden	TEILS
O 13	Mit dem Player können Abläufe vorwärts *und* rückwärts abgespielt werden	JA
O 14	Die Animation statischer Diagramme (Farben ändern etc.) ist möglich	JA
O 15	Es ist möglich, die Anzeige komplexer Zustände zur Laufzeit zu konfigurieren …	JA
O 16	Eine parallele Anzeige der zum aktiven Teil der Datenstruktur passenden Programmteile (beim Debugging) ist … möglich	TEILS
O 17	Die dargestellten Teile einer Datenstruktur können mit HTML-Links … versehen werden …	MÖGL
O 18	Es kann mehrere Fenster mit verschiedenen Sichten geben	MÖGL
O 19	Es gibt eine Schnittstelle für den Aufruf von Skripten	JA
O 20	Die Bedienung der Laufzeitumgebung ist intuitiv	JA
O 21	Das automatisch berechnete Layout kann manuell geändert werden	JA
O 22	Ein vorhandener Fahrplan für eine interaktive Übung in einem explorativen Szenario kann dargestellt werden	NEIN
O 23	Eine parallele Anzeige von zum aktiven Teil der Datenstruktur passendem Pseudocode ist … möglich	NEIN
O 24	Mischformen aus graphartigen Datenstr. können dargestellt werden	MÖGL
O 25	Nicht-graphartige Datenstrukturen … können dargestellt werden	NEIN
O 26	Eine asynchrone Anzeige erläuternder HTML-Seiten oder Filme ist als Hilfesystem möglich	MÖGL
O 27	Der Benutzer kann zoomen	MÖGL
O 28	Die Laufzeitumgebung kann als Applet … eingesetzt werden	NEIN
O 29	Sound ist – synchronisiert mit dem Ablauf der Animation – möglich	MÖGL
O 30	Es sind neue Formen der Interaktion vorgesehen …	NEIN

Tabelle 7: Anforderungen an die Oberfläche

	Anforderung	Erfüllung
K 1	Farbschemata sind in Abhängigkeit vom Zustand, von Invarianten, von aktuellen Geschehnissen usw. einfach konfigurierbar	JA
K 2	Meldungen sind in Abhängigkeit vom Zustand konfigurierbar	JA
K 3	Die Anzeige komplexer Zustände der Datenstruktur ist in verschiedenen Details konfigurierbar ...	JA
K 4	Layoutalgorithmen sind unabhängig von Datenstruktur und Algorithmus konfigurierbar	JA
K 5	Das Konfigurieren häufig genutzter Layoutalgorithmen wird dadurch unterstützt, dass wichtige Layoutalgorithmen bereits im System vorhanden sind	JA
K 6	Animationen sind über eine Datenstruktur-spezifische Bibliothek vorhandener Basisoperationen in einer Skriptsprache erstellbar	JA
K 7	Die in der Skriptsprache erstellten Animationen sind einfach auszuwechseln	JA
K 8	Verschiedene Geschwindigkeiten sind für die Animation konfigurierbar	JA
K 9	Stilschemata (z. B. für Kanten) sind konfigurierbar	JA
K 10	Die Umstellung zwischen erzwungenem, automatischen Layout und der Möglichkeit der manuellen Veränderung des Layouts ist konfigurierbar	JA
K 11	Es kann ein Fahrplan für interaktive Übungen konfiguriert werden	TEILS
K 12	Neue Formen der Interaktion sind konfigurierbar	NEIN

Tabelle 8: Anforderungen an die Konfigurierbarkeit

	Anforderung	Erfüllung
SO 1	Es gibt eine Schnittstelle für den externen Aufruf von Operationen auf der Datenstruktur, die von verschiedenen Programmiersprachen aus nutzbar ist	JA
SO 2	Das System ist stabil	JA
SO 3	Zumindest die Laufzeitumgebung ist möglichst plattformunabhängig	TEILS
SO 4	Es kann auf einen Netzwerkanschluss verzichtet, also rein lokal gearbeitet werden	JA
SO 5	Die Komponenten des Systems sind frei verfügbar	TEILS
SO 6	Das Werkzeug ist unabhängig von „evtl. bald nicht mehr verfügbaren Bibliotheken“	TEILS
SO 7	Das System ist einfach zu installieren	NEIN
SO 8	Man kann den Zustand der Animation speichern	NEIN

Tabelle 9: Sonstige Anforderungen

- Zu SP 9: Die geforderte Spezifizierbarkeit ist voll gegeben und in vielen Fällen einfach, wie in Kapitel 6 gezeigt; trotzdem wäre eine weitere Vereinfachung im Bereich der Spezifikation von Schleifen in einem Algorithmus wünschenswert, z. B. durch Einführung neuer Syntax-Konstrukte in PROGRES, siehe auch Abschnitt 10.3.

- Zu LO 11: Eine parallele Anzeige von Programmteilen und Visualisierung ist nur durch ferngesteuerte Algorithmenanimation möglich.

- Zu LO 19: Normalerweise wird in VIDEA ein Fahrplan nicht werkzeugunterstützt überwacht. Stattdessen dient er dem Lernenden als Richtschnur und dieser behält seine vollen interaktiven Möglichkeiten. Trotzdem kann der Dozent an manchen Stellen die Ausführung bestimmter Aktionen durch Spezifikation erzwingen (z. B. durch Vorbedingungen für Operationen).

- Zu LO 22: Die Protokollierung aufgerufener Operationen ist in der aktuellen VIDEA-Version nicht enthalten, aber leicht zu ergänzen.

- Zu O 12: Mit der leicht hinzuzufügenden neuen Player-Schnittstelle können „Filme“ jederzeit angehalten werden – aber natürlich nur zwischen zwei Transaktionen.

- Zu O 16: Hier gilt dasselbe wie bei LO 11.

- Zu O 17: Eine Verknüpfung der dargestellten Teile der Datenstruktur mit HTML-Links ist nicht implementiert, aber implementierbar.

- Zu O 18: Mehrere Fenster sind nicht vorgesehen, wären aber grundsätzlich implementierbar.

- Zu O 27: Zoom ist in VIDEA automatisiert, so dass die aktuell dargestellte Datenstruktur nach jedem Layoutvorgang die Größe des gesamten Fensters einnimmt. Manueller Zoom wäre leicht hinzuzufügen, ist aber im Moment nicht vorhanden.

- Zu K 11: Ein Fahrplan kann in VIDEA nicht werkzeugunterstützt konfiguriert werden, sondern durch Anpassung der externen, dem Lernenden vorliegenden Anleitung. Soll der Fahrplan von VIDEA überwacht werden, muss für die Änderung die Spezifikation geändert werden; das ist im Sinne dieser Arbeit keine Konfiguration.

- Zu SO 3: Die in der aktuellen VIDEA-Version verwendete PROGRES-Version ist nicht auf Windows ablauffähig, sondern nur auf Linux und einigen Unix-Derivaten. VIDEA wurde trotzdem auf Windows-Rechnern eingesetzt, indem eine X-Emulation (Exceed [Exc04]) verwendet wurde. Das ist natürlich keine Plattformunabhängigkeit im eigentlichen Sinne.

- Zu SO 5: Alle in VIDEA verwendeten Komponenten sind in ihren Laufzeitlizenzen frei verfügbar. Soll mithilfe von yFiles weiterentwickelt werden (zum Beispiel zur Entwicklung neuer Layoutalgorithmen) muss eine Entwicklungslizenz für yFiles erworben werden, die allerdings für die universitäre Lehre stark vergünstigt ist.

- Zu SO 6: Dieser Punkt ist naturgemäß etwas vage, wie bereits in Kapitel 3 klargestellt wurde. Trotzdem ist es sinnvoll, diese Überlegung durchzuführen. Das ganze VIDEA-Paket, wozu UPGRADE2, PROGRES und yFiles gehören, besteht zum Großteil aus Java. Des weiteren werden in PROGRES die Programmiersprachen C, TCL/TK und Modula3 genutzt. Solange keine neue, Java-basierte PROGRES-Version verfügbar ist, ist dieser Punkt also nur zum Teil erfüllt.

10.2.2. Bewertung von VIDEA anhand von *Best Practices*

Im Jahr 2002 identifizierte die 'Working Group on Improving the Educational Impact of Algorithm Visualization' in [NRA+02] elf Eigenschaften multimedialer Werkzeuge für die Lehre von Algorithmen, die sich im praktischen Einsatz besonders bewährt haben und inzwischen weitgehend akzeptiert sind. Der Fokus dieser so genannten Best Practices liegt nur auf einem Teil der Einsatzmöglichkeiten von VIDEA, nämlich auf der reinen Algorithmenanimation. Der Aspekt der hochinteraktiven Simulationsumgebung für die Lehre von Datenstrukturen wird also weitgehend vernachlässigt, genau wie bei den meisten bestehenden Ansätzen. Außerdem sind viele der hier genannten Punkte, soweit sie sich auf Ergebnisse durchgeführter Studien zum Lerneffekt multimedialer Lehrwerkzeuge beziehen, schon in unserer eigenen Anforderungsliste enthalten.

Da jedoch bei der reinen Algorithmenanimation nur ein Teil der Punkte unserer Liste eine Rolle spielt und außerdem der Aspekt der leichten Anpassbarkeit von Lerneinheiten komplett vernachlässigt wird, ergibt sich ein anderer Blickwinkel und eine andere Priorisierung der gewünschten Anforderungen. Trotzdem ist Algorithmenanimation ein Teil von VIDEA, und so nutzen wir die Gelegenheit, VIDEA auch auf die Erfüllung dieser Best Practices hin zu überprüfen. Die empfohlenen Vorgehensweisen sind:

1. Provide resources that help learners interpret the graphical representation.
2. Adapt to the knowledge level of the user.
3. Provide multiple views.
4. Include performance information.
5. Include execution history.
6. Support flexible execution control.
7. Support learner-built visualizations.
8. Support custom input data sets.
9. Support dynamic questions.
10. Support dynamic feedback.
11. Complement visualizations with explanations.

Da manche dieser Formulierungen in ihrer Kurzform nicht eindeutig einzuordnen sind, geben wir im Folgenden für jede der Vorgehensweisen eine kurze Erklärung ihrer Bedeutung und danach eine Schilderung ihrer Umsetzung oder fehlenden Umsetzung in VIDEA:

- Best practice: *Provide resources that help learners interpret the graphical representation.*

  Gemeint ist damit: Die Bedeutung der graphischen Repräsentation und ihrer Beziehung zu Programmelementen sollte entweder durch Text und Erklärungen im System oder durch explizite Erklärungen des zu lehrenden Algorithmus während der Unterrichtszeit klargestellt werden.

  Implementierung in VIDEA: Es wird eine externe Erklärung der grundsätzlichen Repräsentation im Fahrplan und eine kontextsensitive Erklärung durch frei definierbare Meldungen gegeben. Es werden geeignete Layoutalgorithmen und Knoten- und Kanten-Beschriftungen genutzt.

- Best practice: *Adapt to the knowledge level of the user.*

  Gemeint ist damit: Es besteht eine Gefahr, Anfänger mit Details oder Möglichkeiten der Software zu überlasten. Andererseits können Fortgeschrittene von mehr Kontrollmöglichkeiten und mehr Freiheitsgraden bei Darstellung und Interaktion, zum Beispiel mehreren Sichten oder frei wählbaren, auch großen, Eingabedatenmengen für Algorithmen und Operationen, profitieren.

  Implementierung in VIDEA: Es wird eine Überfrachtung der Oberfläche durch überflüssige Funktionalität vermieden. Sowohl vordefinierte Instanzen einer Datenstruktur als auch frei erzeugbare Szenarien zum Experimentieren sind für jeden Algorithmus möglich, da immer die gesamte Datenstruktur als Eingabedatenmenge für Operationen zur Verfügung steht. Dadurch entsteht für den Anfänger aber keine erhöhte Komplexität, da keine zusätzlichen Bedienungselemente notwendig sind. Unterschiedlich komplexe Möglichkeiten der Einflussnahme sind durch entsprechend definierte Aktionen möglich. Es können also mehr Operationen in der Fortgeschrittenen-Instanz angezeigt werden. Durch Konfiguration ist ein leichtes Umschalten zwischen Anfänger- und Fortgeschrittenen-Instanz möglich.

- Best practice: *Provide multiple views*.

  Gemeint ist damit: Das Angebot mehrerer Sichten kann dazu beitragen, das Verständnis des fortgeschrittenen Lernenden zu erhöhen. Zwei besonders aussichtsreiche Arten, mehrere Sichten anzubieten, sind laut der Autoren:

  - Die gleichzeitige Darstellung von Programm- oder Pseudocode, in dem die aktuelle Bearbeitungsstelle farblich markiert ist, und einer abstrakten Algorithmenanimationssicht.
  - Die nacheinander stattfindende Darstellung verschiedener Sichtweisen auf den Algorithmus, wie zum Beispiel mehrere Abläufe verschiedener Detaillierung.

  Implementierung in VIDEA: Die verteilte Darstellung in mehreren Fenstern ist in VIDEA nicht implementiert; allerdings wäre eine mögliche Erweiterung die Einführung zusätzlicher Sichten auf UPGRADE2-Ebene und die Zuweisung unterschiedlicher Layoutalgorithmen an diese. Schon jetzt ist die synchronisierte Sicht von Quellcode und Datenstruktur möglich durch die ferngesteuerte Algorithmenanimation, bei der der Lernende selbst mithilfe eines beliebigen Quellcode-Debuggers die Schrittgeschwindigkeit festlegt (siehe

die Beschreibung des Szenarios *Ferngesteuerte Animation eigener Programme der Lernenden* in Kapitel 7). Es liegt dann eine simultane Darstellung von Quellcode und Animation vor, wobei die gerade verarbeitete Programmstelle je nach verwendetem Debugger farblich markiert ist.

Die nacheinander stattfindende Darstellung mehrerer Sichtweisen auf den gleichen Algorithmus ist in VIDEA bereits integraler Bestandteil durch das Aufbrechen schleifenartiger Abläufe in Schritte. Diese Schritte, die verschiedenen Detaillierungsstufen entsprechen, können sowohl durch nacheinander ausgeführte Skripte als auch interaktiv vom Lernenden genutzt werden, um den Algorithmus in unterschiedlicher Ausführlichkeit ablaufen zu lassen.

- Best practice: *Include performance information.*

  Gemeint ist damit: Die Anzeige von gesammelten Messgrößen und Daten über den Algorithmus kann das Verständnis der Lernenden erhöhen. Ein spezielles Beispiel hierfür ist die gleichzeitige Animation mehrerer konkurrierender Algorithmen, z. B. mehrerer Sortieralgorithmen.

  Implementierung in VIDEA: Möglich ist die Darstellung zusätzlicher Informationen über einen Algorithmus im Sinne von visuellem Feedback durch entsprechend vorbereitete Meldungen. Nicht möglich ist die vergleichende Komplexitätsbetrachtung von Algorithmen durch Messung und Darstellung bestimmter Größen. Bei der ebenfalls vorgeschlagenen vergleichenden Betrachtung zweier oder mehrerer Algorithmen durch gleichzeitige Anzeige ist es bei den durch VIDEA unterstützten Algorithmen auf graphartigen Datenstrukturen fraglich, ob nicht eher eine Verwirrung der Lernenden stattfindet. Prinzipiell ist dies in VIDEA möglich durch den gleichzeitigen Start zweier VIDEA-Instanzen, z. B. für die kürzeste Wegesuche, mit jeweils einem Skript, dessen Operationen auf dem gleichen Ausgangsgraphen operieren und mit der gleichen Geschwindigkeit ablaufen. Diese Möglichkeit ist zwar nur zu einem kleinen Teil eine Eigenschaft des VIDEA-Systems, erfüllt aber prinzipiell den angegebenen Zweck. Problematisch für dieses Szenario ist natürlich, dass in VIDEA zur Zeit keine Synchronisation im eigentlichen Sinn zwischen zwei laufenden Instanzen möglich ist. Das verbleibende Problem, in welcher Detailliertheit in jedem der Skripte der Algorithmus animiert wird, so dass die unterschiedlichen Algorithmen und Konzepte tatsächlich vergleichbar sind, ist nicht VIDEA-spezifisch.

- Best practice: *Include execution history.*

  Gemeint ist damit: Wenn Lernende eine grobe Übersicht eines Algorithmus sehen wollen, oder frühere Schritte eines Algorithmus vergessen oder missverstanden haben, kann es hilfreich sein, dem Lernenden Ablaufinformationen zur Verfügung zu stellen.

  Implementierung in VIDEA: Eine Übersichtsdarstellung von Algorithmen ist nicht im Werkzeug implementiert, allerdings ist es möglich, frühere Schritte durch 'undo' / 'redo' ins Gedächtnis zu rufen. Eine werkzeugunabhängige Übersicht über die vorgegebenen Schritte einer Übungsaufgabe bietet auch der dem Lernenden sichtbare Teil des VIDEA-Fahrplans. Auf diesem navigiert der Lernende sozusagen, und kann jederzeit einordnen, wo er oder sie sich befindet.

- Best practice: *Support flexible execution control.*

  Gemeint ist damit: Als Kontrolle über die Algorithmenausführung werden u. a. eine schrittweise oder kontinuierliche Ausführung vorwärts und rückwärts und die Möglichkeiten, zum Anfang und zum Ende der Animation springen zu können, vorgeschlagen.

  Implementierung in VIDEA: Die Funktionalität und Schnittstelle eines simulierten Abspielgerätes für Animationen mit den Funktionen „vorwärts spielen", „Schritt vorwärts", „Schritt rückwärts", „Sprung vorwärts", „Sprung rückwärts" und „anhalten" ist möglich und technisch vorbereitet.

- Best practice: *Support learner-built visualizations.*

  Gemeint ist damit: Lernende gewinnen Einsichten über wesentliche Punkte eines Algorithmus und identifizieren sich mehr mit dem Lernthema, wenn sie ihre eigenen Animationen kreieren.

  Implementierung in VIDEA: Lernende können durch die ferngesteuerte Animation eigener Programme mit VIDEA de facto eigene Visualisierungen erstellen. Zusätzlich ist es möglich, Lernende in die Rolle des VIDEA-Dozenten zu versetzen, wie in Szenario 10 aus Kapitel 5 angedeutet (*Spezifikation, Generierung und Test eigener Instanzen der Visualisierungsumgebung von Seiten der Lernenden*). Dann erstellen sie selbst eine visuelle Spezifikation für die Erforschung einer Datenstruktur bzw. für die Visualisierung eines Algorithmus.

- Best practice: *Support custom input data sets.*

  Gemeint ist damit: Den Lernenden zu erlauben, eigene Eingabedatensätze zu spezifizieren, beschäftigt sie auf eine aktivere Art mit der Visualisierung, da sie die visualisierten Abläufe anhand einer Reihe verschiedener, selbst gewählter Daten verfolgen können.

  Implementierung in VIDEA: Die Eingabe eigener Datenmengen (Knoten, Kanten, Werte) und das Testen, Laufenlassen, Simulieren eines Algorithmus gegen diese Daten ist bereits Kern-Bestandteil von VIDEA.

- Best practice: *Support dynamic questions.*

  Gemeint ist damit: Um die Reflektion der Lernenden über eine Visualisierung zu erhöhen, wird vorgeschlagen, regelmäßig zwei Arten von Fragen auftauchen zu lassen: Zufällige Fragen an vorbereiteten Stellen und feste Fragen, die an kritischen Punkten immer gestellt werden und den Animationsfluss aufhalten, bis sie korrekt beantwortet werden. Beide Arten von Fragen sollen das Verständnis vertiefen.

  Implementierung in VIDEA: Diese Idee bezieht sich auf Algorithmenanimations-Systeme, bei denen die vorgeschlagene Art der Befragung ein Mittel ist, den ansonsten passiven Ablauf zu unterbrechen und Interaktivität einzubauen. Da in VIDEA der Lernende – mit Ausnahme des Ablaufs von Skripten – permanent aktiv ist und sein soll, gibt es immer mehrere Lernpfade, auf denen weiter geforscht werden soll. Deshalb passen die vorgeschlagenen Fragen nicht zu jeder VIDEA-Instanz.

Zufällige Fragen sind in VIDEA nicht implementiert. Stattdessen werden Verständnistests an geeigneter Stelle im VIDEA-Fahrplan eingebaut, die der Lernende an dieser Stelle beantworten muss oder kann – ja nach Spezifikation des Dozenten. Bei der Beantwortung wird der Lernende vom Werkzeug durch interaktives Feedback unterstützt.

- Best practice: *Support dynamic feedback*.

  Gemeint ist damit: Die von den Lernenden vorgenommenen Manipulationen der graphischen Repräsentationen sollten zu dynamischen Meldungen führen können, die z. B. darauf hinweisen, ob ein vom Lernenden vorgenommener Schritt in Richtung eines vorgegebenen Algorithmus geht.

  Implementierung in VIDEA: Dynamisches Feedback ist integraler Bestandteil von VIDEA. Es gibt einerseits Rückmeldungen des Systems durch eine Veränderung der Einfärbung bzw. der gestrichelten oder durchgehenden Zeichnung von Kanten. Andererseits kann bei Ausführung jedes Schrittes, hinter dem in VIDEA immer eine Operation steckt, zusätzlich oder alternativ eine geeignete Rückmeldung im Sinne dieser Best Practice an den Lernenden erfolgen, wenn dies vom Dozenten so spezifiziert wurde.

- Best practice: *Complement visualizations with explanations*.

  Gemeint ist damit: Gerade bei klassischen Algorithmenanimationen ist es hilfreich, parallel zur Visualisierung Erklärungen in irgendeiner Form anzubieten. Die Autoren nennen hier drei Beispiele: Textuelle Erklärungen in einem zweiten Fenster, das synchrone Abspielen vorbereiteter Tonspuren und die Nutzung eines Lehrbuchs.

  Implementierung in VIDEA: Wie in Kapitel 7 beschrieben, ist im Szenario *Vorführen im Rahmen einer Präsenzveranstaltung* eine synchrone, auditive Erklärung in VIDEA vorgesehen. Für diejenigen Szenarien, die beim Selbststudium Anwendung finden, wäre allerdings die Möglichkeit des Systems, eine synchrone Tonspur ablaufen zu lassen, wünschenswert. Die parallele Anzeige eines zweiten Fensters ist aktuell in VIDEA nicht vorgesehen. Es ist allerdings jederzeit möglich, an wichtigen Stellen eines Algorithmus Meldungen auszugeben, wenn das sinnvoll erscheint. Der dem Lernenden vorliegende VIDEA-Fahrplan zu einer interaktiven Übung dient neben der Führung der Lernenden auch als parallel nutzbare Informationsquelle und Erklärung der visuellen Aktivitäten auf dem Bildschirm (Beispiele für die Nutzung von Fahrplänen wurden in Kapitel 7, Beispiele für komplette Übungen mit VIDEA in Kapitel 9 angegeben).

Wie durch diesen Abgleich deutlich wird, kann VIDEA viele der von den Autoren als wesentlich genannten Punkte erfüllen. Allerdings gibt es auch offene Punkte; auf diejenigen, die für VIDEA sinnvoll erscheinen, werden wir nochmal in Abschnitt 10.3 eingehen.

10.3. Ausblick

Das VIDEA-Rahmenwerk in seiner jetzigen Form zeigte sich bisher im praktischen Einsatz den wesentlichen, an es gestellten Anforderungen gewachsen. Die vorgestellte Evaluation kann natürlich nur ein Anfang sein, da sich aufgrund der geringen Größe der Studentengruppen keine signifikanten Ergebnisse gewinnen ließen. Neue Untersuchungen über die Wirksamkeit der Nutzung von VIDEA-Instanzen in Bezug auf den Lerneffekt wären also sinnvoll.

Ebenso bietet es sich an, zu untersuchen, inwieweit verschiedene Dozenten mit der hier propagierten Art der Spezifikation von Algorithmen und Datenstrukturen zurechtkommen; bisher gibt es dazu vor allem eigene Erfahrungen bei der Erstellung der vorgestellten VIDEA-Instanzen. Hier könnten sich wertvolle Hinweise für die Weiterentwicklung von VIDEA ergeben.

Einige sinnvolle Erweiterungsmöglichkeiten von VIDEA haben sich bereits aus den bisherigen Erfahrungen und der durchgeführten inhaltlichen und nicht-inhaltlichen Evaluation ergeben und wurden an geeigneter Stelle erwähnt. Im Folgenden fassen wir noch einmal die wesentlichen Punkte zusammen:

Es wurden mehrere technische Verbesserungsmöglichkeiten identifiziert, von denen ein Teil bereits in VIDEA aufgenommen wurde. Beispiele von zusätzlichen Fähigkeiten, deren Nutzen zu untersuchen wäre, sind manueller Zoom, eine komfortablere Bedienung der Oberfläche der VIDEA-Manager für Spezialfälle, die Möglichkeit, mehrere Sichten anzuzeigen, eine pro Operation einstellbare Geschwindigkeit des Morphing-Vorgangs bei der flüssigen Animation und die Möglichkeit, eine vorbereitete Tonspur synchronisiert mit der Animation vorbereiteter Abläufe abzuspielen.

Bei der Parameter-Übergabe von Knoten und Kanten an PROGRES-Programmeinheiten wird zur Zeit ein interner UPGRADE2-Objekt-Identifikator erwartet, falls der Lernende nicht einfach durch Markieren des jeweiligen graphischen Objekts die automatisierte Parameterübergabe von VIDEA nutzt. Das kann zur Verwirrung des Lernenden führen, wenn die Lernumgebung nicht richtig eingeführt wurde. Es wurden mehrere Möglichkeiten genannt, dieses Standardverhalten von UPGRADE2 vor den Lernenden zu verbergen oder sogar für einen erweiterten Lerneffekt zu nutzen. Trotzdem wäre es im Sinne einer weiteren Automatisierung für den Dozenten hilfreich, wenn er zum Beispiel ein Attribut des zu übergebenden Knotens oder der zu übergebenden Kante auszeichnen könnte, dessen Wert dann repräsentativ für den Knoten oder die Kante im Parameterfenster eingegeben werden kann. Erforderlich hierfür wären Anpassungen sowohl im UPGRADE2- als auch im VIDEA-Code, und eine neue Schnittstelle für den Dozenten, um das jeweilige Attribut spezifizieren zu können.

Ein Punkt, der für die interaktive Nutzung einer VIDEA-Instanz sinnvoll ist, sich aber bei der Spezifikation von Algorithmen und bei der Nutzung von Skripten zum Teil unschön auswirkt, ist das in VIDEA notwendige Aufbrechen schleifenartiger Abläufe in Schritte verschiedener Schrittweite. Diese Notwendigkeit ergibt sich erstens daraus, dass die Visualisierung nur aktualisiert wird, wenn eine von „außen“, also von der Nutzerschnittstelle oder einem Skript aus aufgerufene PROGRES-Programmeinheit endet. Wenn also in einer PROGRES-Transaktion ein schleifenartiger Ablauf enthalten ist, ist nach dem Aufruf der Transaktion erst wieder der Zustand der Datenstruktur *nach* dem Ablauf der kompletten Schleife zu sehen. Deshalb nutzt

man Teilschritte, zwischen deren Aufruf jeweils die Anzeige aktualisiert werden kann. Zweitens entsprechen auch die Zustände der Datenstruktur, zwischen denen beim Ablaufenlassen von Skripten in beide Richtungen (durch 'undo' und 'redo') gewechselt werden kann, den Zuständen zwischen je zwei vom Skript aus aufgerufenen PROGRES-Programmeinheiten. Wenn also eine vom Skript aus aufgerufene Transaktion eine Schleife enthält, entspricht ein 'undo'-Schritt wieder einem kompletten Rückgängig-Machen der ganzen Transaktion.

Einerseits wird durch die Lösung, Schritte und Teilschritte für schleifenartige Abläufe festzulegen, die sowohl in Skripten als auch direkt vom Lernenden aufgerufen werden können, ein hohes Maß an Interaktivität gewährleistet; denn dadurch ist der Lernende nicht darauf festgelegt, vordefinierte Abläufe auf eine vorherbestimmte Art zu nutzen. Stattdessen hat er jederzeit die Wahl, einen kleinen oder großen Schritt weiterzuschalten, Zwischenfragen zum gerade angezeigten Zustand an das System zu stellen oder selbst zu beantworten, oder durch den Aufruf von zur Verfügung gestellten Grundoperationen den Algorithmus händisch zu simulieren und die Rückmeldungen des Systems (in Form von Veränderung von Struktur und Einfärbung der Datenstruktur oder ausgegebenen Meldungen) abzuwarten.

Andererseits führt die aus den genannten Gründen bestehende *Notwendigkeit*, die definierten Schritte zu benutzen, zu zwei Problemen: Erstens muss der Dozent bei der Spezifikation eines Algorithmus eine Aufschreibung nutzen, die ungewohnt ist, anstatt die in ausreichendem Maße existierenden Schleifenkonstrukte der PROGRES-Sprache verwenden zu können. Beispiele hierfür wurden in Kapitel 6 gezeigt. Und zweitens müssen bei einer Ausführung von schleifenartigen Abläufen in Skripten die zugehörigen Kontrollstrukturen *ins Skript* geschrieben werden, damit ein feingranulares 'undo' und 'redo' möglich bleibt. Damit ist aber die in VIDEA ansonsten sauber durchgeführte Trennung von Spezifikation (von Algorithmus und Datenstruktur) und Visualisierung nicht mehr zu 100% gegeben.

Die Frage ist nun, ob und wie das in VIDEA gegebene hohe Maß an Interaktivität, das sich an dieser Stelle vor allem in der Wahlfreiheit des Lernenden zeigt, in jedem Schleifendurchlauf zu entscheiden, in welcher Granularität und ob überhaupt der momentane Ablauf fortgesetzt werden soll, sich verbinden lässt mit einer einfacheren und konsequenteren Formulierung von Algorithmen komplett und direkt in der PROGRES-Sprache.

Der erste Schritt in diese Richtung wäre die Möglichkeit, Zwischenschritte in PROGRES-Schleifen visualisieren und vorwärts und rückwärts ausführen zu können. Hierzu bietet PROGRES bereits einige wesentliche Voraussetzungen, da auch Zwischenzustände aufgerufener PROGRES-Programmeinheiten protokolliert werden und Checkpoints in der genutzten Graphdatenbank gesetzt werden. Diese Zwischenzustände werden für das in PROGRES genutzte 'backtracking' sowieso verwendet und könnten mit nicht allzu hohem Aufwand zu entsprechend feinen 'undo'- und 'redo'-Möglichkeiten erweitert werden, da auch der von PROGRES generierte und in VIDEA genutzte Zugriffscode dahingehend erweitert werden kann, die Kontrolle zwischen zwei Teilschritten zurückzugeben.

Der zweite Schritt bestünde darin, die PROGRES-Sprache selbst zu erweitern, um spezifizieren zu können, wann die Ausführung jeweils angehalten werden soll. Eine zusätzliche Schwierigkeit besteht im Fall geschachtelter Schleifen, wenn man also verschiedene Granularitäten als „Einzelschritt“ anbieten will. Als Lösung denkbar wären Haltepunkte verschiedener Sorten.

Um zu klären, wie gewährleistet werden kann, dass der Lernende den Ablauf jederzeit verlassen und wieder betreten kann, ohne dass die Schnittstelle zwischen dem Lernenden und dem System oder die Spezifikation zu unhandlich wird, bedarf es allerdings weiterer Forschung.

Ein weiterer interessanter Forschungsgegenstand wäre die Einbindung des vorgestellten umfassenden Lehrkonzepts mit Fahrplänen und Lernschritten in die Werkzeugunterstützung. Es wurde in dieser Arbeit betont, dass die Auslagerung von Fahrplänen in den Verantwortungsbereich des Dozenten ohne Unterstützung des Werkzeugs aus gutem Grund erfolgte: Frühere Ansätze scheiterten an diesem Punkt, indem sie zugunsten einer stärkeren Führung der Lernenden entweder die interaktiven Möglichkeiten zu stark einschränkten oder den Aufwand zur Erstellung von neuen Animationen stark in die Höhe trieben (siehe Kapitel 3 und 4). Natürlich wäre es im Sinne einer umfassenderen Werkzeugunterstützung sinnvoll, zu untersuchen, inwieweit die Definition und Überprüfung von Fahrplänen ohne die beschriebenen Nachteile in das Werkzeug integrierbar sind. In den Kapiteln 5 und 6 wurde gezeigt, dass es schon jetzt in VIDEA möglich ist, für besondere Fälle sinnvolle Einschränkungen der vom Lernenden nutzbaren Lernpfade festzulegen. Zum Teil ist es dazu allerdings notwendig, diese in die PROGRES-Spezifikation mit aufzunehmen, wo sie sich zwar auf einer hohen logischen Spezifikationsebene befinden, aber mit der Spezifikation der Datenstruktur und des Algorithmus selbst vermengt werden. Die Erforschung von Möglichkeiten, Fahrpläne und Abläufe in die Werkzeugunterstützung aufzunehmen und dabei von der PROGRES-Spezifikation zu entkoppeln, wäre also zu untersuchen.

Ebenso wünschenswert wäre die Erweiterung von VIDEA um die Fähigkeit, eine größere Klasse von Datenstrukturen als die im Moment möglichen graphartigen Datenstrukturen (siehe Kapitel 6) auf die in Lehrbüchern übliche Art visualisieren zu können. Beispiele hierfür sind B-Bäume und Felder. Bei Feldern ist es fraglich, ob sie sinnvoll als Graphen spezifiziert werden können. Eine denkbare Lösung ist die Spezifikation eines Feldes als eine Menge von Knoten, deren Position im Feld durch ein zusätzliches zahlenwertiges Attribut pro Dimension des Feldes spezifiziert wird. Diese Attribute könnten dann als Grundlage für den Layoutalgorithmus dienen, ohne angezeigt zu werden. Zugriffe auf Feldelemente könnten dann ebenso über diese „Schlüsselattribute“ erfolgen. Um herauszufinden, ob dieser Ansatz zielführend ist, bedarf es weiterer Forschung.

B-Bäume können als Graphen spezifiziert werden, werden aber von VIDEA nicht auf die gewohnte Art visualisiert, da die Attribute von Knoten in VIDEA, abgesehen von den kantenwertigen Attributen, im Moment textuell untereinander geschrieben werden. Üblich ist aber eine Darstellung der inneren Knoten eines B-Baums als eine Menge horizontal aneinandergrenzender Rechtecke, die jeweils eine Beschriftung enthalten. Hier wäre es evtl. sinnvoll, ein mächtigeres Layoutkonzept einzubinden. Es gibt zwei naheliegende Lösungsmöglichkeiten, die allerdings beide auf den ersten Blick unbefriedigend sind: Es könnte entweder eine Anzahl fester Layout-Schemata vorgegeben werden, wie zum Beispiel das Zeichnen der anzuzeigenden Attribute als Text untereinander (also wie bisher) oder als getrennte Kästchen nebeneinander und so weiter. Auf diese Art erweitern sich die Möglichkeiten der Darstellung aber nur um diese fest implementierten Layout-Schemata und es entsteht nur eine eingeschränkte zusätzliche konzeptionelle Freiheit bei der Darstellung neuer Arten von Layouts. Oder es wird ein Teil der Layoutinformation in die visuelle Spezifikation der Datenstruktur mit aufgenommen, indem zum Beispiel ein

spezieller Kantentyp des Graphen als Layoutinformation für „nebeneinander zeichnen“ dient; diese Idee widerspricht allerdings der strikten Trennung von Datenstruktur und Layout, die aus den in Kapitel 3 genannten guten Gründen eingeführt wurde. Auch in diesem Punkt sind also weitere Forschungen notwendig.

Literatur

[AFJ+03] A. AKINGBADE, T. FINLEY, D. JACKSON, P. PATEL und S. H. RODGER: *JAWAA: Easy web-based animation from CS 0 to advanced CS courses*. In: *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Seiten 162–166. ACM Press, 2003.

[AGD03] http://www.ads.tuwien.ac.at/AGD/, März 2003.

[Amu03] http://www-2.cs.cmu.edu/afs/cs/project/amulet/www/amulet-home.html, März 2003.

[Ani03] http://www.informatik.uni-siegen.de/db/animations.php3?lang=en, März 2003.

[AS03] P. ASCHENBRENNER und A. SCHÜRR: *Generating Interactive Animations from Visual Specifications*. In: *2003 IEEE Symposium on Visual Languages and Formal Methods*, Auckland , New Zealand, Okt. 2003.

[Asc03] P. ASCHENBRENNER: *Eine Lerneinheit für die multimediale Lehre von Algorithmen und Datenstrukturen*. In: A. BODE, J. DESEL, S. RATHMAYER und M. WESSNER (Herausgeber): *Proceedings Delfi 2003*, Nummer P-37 in *GI-Edition Lecture Notes in Informatics*, Seiten 412–421, Bonn, 2003. Köllen Druck Verlag GmbH.

[BAD+01] M. BURNETT, J. ATWOOD, R. DJANG, H. GOTTFRIED, J. REICHWEIN und S. YANG: *Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm*. In: *Journal of Functional Programming, 11(2)*, Seiten 155–206, 2001.

[Bae98] R. BAECKER: *Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science*. In: *Stasko et al. (Hrsg.): Software Visualization, MIT Press*, Seiten 369–381, 1998.

[Bal90] S.-P. BALLSTAEDT: *Integrative Verarbeitung bei audiovisuellen Medien*. In: *Böhme-Dürr and K. Emig and J. Seel et al. (Hrsg.): Wissensveränderung durch Medien*, München, 1990.

[BCS96] M. BYRNE, R. CATRAMBONE und J. STASKO: *Do Algorithm Animations Aid Learning?* Technischer Bericht GIT-GVU-96-18, 1996.

[Ber96] *Die neue deutsche Rechtschreibung*. Bertelsmann, München, 1996.

[BG01] R. BANDLER und J. GRINDER: *Kommunikation und Veränderung. Die Struktur der Magie II*. Junfermann, Paderborn, 8. Aufl. 2001.

[BH98] M. BROWN und J. HERSHBERGER: *Fundamental Techniques for Algorithm Animation*. In: *Stasko et al. (Hrsg.): Software Visualization, MIT Press*, Seiten 81–102, 1998.

[BM00] R. BANDLER und W. MACDONALD: *Der feine Unterschied. NLP-Übungsbuch zu den Submodalitäten.* Junfermann, Paderborn, 4. Aufl. 2000.

[BM04] M. BIDOIT und P. D. MOSSES: *CASL User Manual.* LNCS 2900 (IFIP Series). Springer, 2004.

[BR97] M. H. BROWN und R. RAISAMO: *JCAT: collaborative active textbooks using Java.* In: *COMPUGRAPHICS '96: Proceedings of the fifth international conference on computational graphics and visualization techniques on Visualization and graphics on the World Wide Web*, Seiten 1577–1586. Elsevier Science Inc., 1997.

[Bro78] *Der Große Brockhaus*, Band 3, achtzehnte, völlig neu bearbeitete Auflage. Wiesbaden, 1978.

[Bro88] M. BROWN: *Exploring Algorithms Using Balsa-II.* IEEE Computer, 21(5):14–36, 1988.

[Bro91] M. H. BROWN: *Zeus: A System for Algorithm Animation and Multi-View Editing.* In: *Proceedings of IEEE Workshop on Visual Languages*, Seiten 4–9. IEEE Computer Society Press, New York, 1991.

[Bub03] http://olli.informatik.uni-oldenburg.de/fpsort/Animation.html, März 2003.

[Bud01] T. BUDD: *Classic Data Structures in Java.* Addison-Wesley, 2001.

[CCA03] http://www.cs.hope.edu/ alganim/ccaa/, März 2003.

[CDFP00] P. CRESCENZI, C. DEMETRESCU, I. FINOCCHI und R. PETRESCHI: *Reversible Execution and Visualization of Programs with LEONARDO.* Journal of Visual Languages and Computing, 11(2):125–150, 2000.

[Cha70] S.-K. CHANG: *The analysis of two-dimensional patterns using picture processing grammars.* In: *Proceedings of the second annual ACM symposium on Theory of computing*, Seiten 206–216. ACM Press, 1970.

[Cor05] http://skillstutor.com/index.cfm?fuseaction=skillsbank.cornerstone, Jan. 2005.

[DDD03] http://www.gnu.org/software/ddd/, März 2003.

[DE02] E.-E. DOBERKAT und G. ENGELS: *MuSofT - Multimedia in der Softwaretechnik.* Informatik Forschung und Entwicklung, 17(1):41–44, 2002.

[DHM96] S. DOUGLAS, C. HUNDHAUSEN und D. MCKEOWN: *Exploring Human Visualization of Algorithms.* In: *Proceedings of Graphics Interface '96*, Seiten 9–16, 1996.

[DUD86] DUDEN: *Die Rechtschreibung*, Band 1, 19., neu bearbeitete und erweiterte Auflage. Mannheim, 1986.

[Ein05a] http://ist.unibw-muenchen.de/Inst2/Lectures/WT2002/INF2/index.html, Sept. 2005.

[Ein05b] http://ist.unibw-muenchen.de/Inst2/Lectures/WT2002/INF2/index.html, Kapitel 9, Seite 329, März 2005.

[Ein05c] http://ist.unibw-muenchen.de/Inst2/Lectures/WT2002/INF2/index.html, Kapitel 10, Seite 365, März 2005.

[Ein05d] http://ist.unibw-muenchen.de/Inst2/Lectures/WT2002/INF2/index.html, Kapitel 9, Seite 338, März 2005.

[Exc04] http://www.hummingbird.com/products/nc/exceed/index.html, Nov. 2004.

[Fal02] N. FALTIN: *Strukturiertes aktives Lernen von Algorithmen mit interaktiven Visualisierungen*. Doktorarbeit, Universität Oldenburg, 2002.

[FSST86] M. L. FREDMAN, R. SEDGEWICK, D. D. SLEATOR und R. E. TARJAN: *The pairing heap: a new form of self-adjusting heap*. Algorithmica, 1(1):111–129, 1986.

[Gan05] *Ganimal. Project Homepage*. http://www.cs.uni-sb.de/GANIMAL, März 2005.

[Glo98] P. A. GLOOR: *User Interface Issues For Algorithm Animation.* In: *Stasko et al. (Hrsg.): Software Visualization, Kapitel 11, MIT Press*, Seiten 145–152, 1998.

[Güt92] R. H. GÜTING: *Datenstrukturen und Algorithmen, Kapitel 5*. Teubner, Stuttgart, 1992.

[HD98] A. HAUSNER und D. P. DOBKIN: *GAWAIN: Visualizing Geometric Algorithms with Web-Based Animation.* In: *Symposium on Computational Geometry*, Seiten 411–412, 1998.

[HNH02] S. HANSEN, N. NARAYANAN und M. HEGARTY: *Designing educationally effective algorithm visualizations*. Embedding Analogies and Animations in Hypermedia. Journal of Visual Languages and Computing, 13(2), Seiten 291–317, 2002.

[HSNH98] S. HANSEN, D. SCHRIMPSHER, N. H. NARAYANAN und M. HEGARTY: *Empirical Studies of Animation-embedded Hypermedia Algorithm Visualizations*. Technischer Bericht CSE98-06, Dept. of Computer Science & Software Engineering Technical Report Series, 1998.

[Jac97] J. JACKY: *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[JSW00] D. JÄGER, A. SCHLEICHER und B. WESTFECHTEL: *UPGRADE: A Framework for Building Graph-Based Software Engineering Tools*. In: *Aachener Informatik-Berichte Nr. 00-3*, 2000.

[JVi03] http://www.ilog.com/products/jviews/, März 2003.

[Kah05] W. KAHL: *Skript zur Vorlesung* Spezifikationstechniken *im Herbsttrimester 2001 an der UniBw München*. http://www2-data.informatik.unibw-muenchen.de/Lectures/HT2001/SpecTech/SpecTech.html, März 2005.

[Ker03] A. KERREN: *Exploratives Lernen mit partiell generierter Lehr- und Lernsoftware*. In: *Tagungsband zum Workshop Grundfragen multimedialer Lehre, GML 2003*, 2003.

[Kla83] H. A. KLAEREN: *Algebraische Spezifikation: Eine Einführung*. Springer, Berlin, Heidelberg, 1983.

[KS96] C. M. KEHOE und J. T. STASKO: *Using Animations to Learn about Algorithms: An Ethnographic Case Study*. Technischer Bericht GIT-GVU-96-20, Atlanta, Sept. 1996.

[KST01] C. M. KEHOE, J. T. STASKO und A. TALOR: *Rethinking the evaluation of algorithm animations as learning aids: an observational study*. International Journal of Human Computer Studies, 54(2):265–284, 2001.

[LBS94] A. LAWRENCE, A. BADRE und J. T. STASKO: *Empirically Evaluating the Use of Animations to Teach Algorithms*. In: *Proceedings of the 1994 IEEE Symposium on Visual Languages, St. Louis, MO*, Seiten 48–54, 1994.

[LF97] H. LIEBERMAN und C. FRY: *ZStep95: A Reversible, Animated Source Code Stepper*. In: *J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price (Hrsg.), Software Visualization: Programming as a Multimedia Experience*, Seiten 277–292. MIT Press, 1997.

[LG86] B. LISKOV und J. GUTTAG: *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, Cambridge, Mass., 1986.

[LNR03] J. LUCAS, T. L. NAPS und G. RÖSSLING: *VisualGraph: a graph class designed for both undergraduate students and educators*. In: *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Seiten 167–171. ACM Press, 2003.

[LOM05] http://ltsc.ieee.org/wg12/par1484-12-1.html, April 2005.

[MA91] R. MAYER und R. ANDERSON: *Animations need narrations: An experimental test of a dual-coding hypothesis*. In: *Journal of Educational Psychology*, Seiten 484–490, 1991.

[MA92] R. MAYER und R. ANDERSON: *The instructive animation: Helping students build connections between words and pictures in multimedia learning*. In: *Journal of Educational Psychology*, Seiten 444–452, 1992.

[Min01] M. MINAS: *Spezifikation und Generierung graphischer Diagrammeditoren*. Shaker-Verlag, Aachen, 2001.

[MM98] K. MARRIOTT und B. MEYER (Herausgeber): *A Survey of Visual Language Specification and Recognition*. Springer Verlag, 1998.

[MMSBA04] A. MORENO, N. MYLLER, E. SUTINEN und M. BEN-ARI: *Visualizing programs with Jeliot 3*. In: *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, Seiten 373–376. ACM Press, 2004.

[Moh88] T. G. MOHER: *PROVIDE: A Process Visualization and Debugging Environment*. IEEE Trans. Softw. Eng., 14(6):849–857, 1988.

[Mos] T. MOSSAKOWSKI: *CASL - From Semantics to Tools, TACAS 2000*. LNCS 1785. Springer.

[MRR95] H. MANDL und G. REINMANN-ROTHMEIER: *Unterrichten und Lernumgebungen Gestalten*. (Forschungsbericht Nr. 60. München), 1995.

[MS94] S. MUKHERJEA und J. T. STASKO: *Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger*. ACM Trans. Comput.-Hum. Interact., 1(3):215–244, 1994.

[Naj95] L. J. NAJJAR: *Does multimedia information help people learn?* Technischer Bericht GIT-GVU-95-28, 1995.

[Nap] T. NAPS: *Incorporating Algorithm Visualization into Educational Theory: A Challenge for the Future*. In: *Informatik / Informatique, Special Issue on Visualization of Software (Apr. 2001)*, Seiten 17–21.

[NBPFnVI00] F. NAHARRO-BERROCAL, C. PAREJA-FLORES und J. ÁNGEL VELÁZQUEZ-ITURBIDE: *Automatic generation of algorithm animations in a programming environment*. In: *Frontiers in Education Conference*, 2000.

[NEN00] T. L. NAPS, J. R. EAGAN und L. L. NORTON: *JHAVE, an environment to actively engage students in Web-based algorithm visualizations*. SIGCSE Bull., 32(1):109–113, 2000.

[NNZ00] U. NICKEL, J. NIERE und A. ZÜNDORF: *Tool demonstration: The FUJABA environment*. In: *Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland*, Seiten 742–745. ACM Press, 2000.

[NRA+02] T. L. NAPS, G. RÖSSLING, V. ALMSTRUM, W. DANN, R. FLEISCHER, C. HUNDHAUSEN, A. KORHONEN, L. MALMI, M. MCNALLY, S. RODGER und J. ÁNGEL VELÁZQUEZ-ITURBIDE: *Exploring the role of visualization and engagement in computer science education*. In: *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, Seiten 131–152. ACM Press, 2002.

[OR94] R. OPPERMANN und H. REITERER: *Software-ergonomische Evaluation*. In: *Edmund Eberleh (Hrsg.), Horst Oberquelle und Reinhard Oppermann: Einführung in die Software-Ergonomie*, Seiten 335–367, 1994.

[Pai86] A. PAIVIO: *Mental representations. A dual coding approach.* New York: Oxford University Press, 1986.

[PBS98] B. PRICE, R. BAECKER und I. SMALL: *An Introduction to Software Visualization*. In: *Stasko et al. (Hrsg.): Software Visualization, chapter 1*, Seiten 3–29, 1998.

[PH01] M. J. V. PEREIRA und P. R. HENRIQUES: *Visualization/Animation of Programs based on Abstract Representations and Formal Mappings*. In: *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, Seiten 373–381. IEEE Computer Society, 2001.

[PH03] M. J. V. PEREIRA und P. R. HENRIQUES: *Alma: a Generic Program Animation System*, 2003.

[PR69] J. L. PFALTZ und A. ROSENFELD: *Web Grammars.* In: *IJCAI*, Seiten 609–620, 1969.

[PRO05] http://musoft.cs.uni-dortmund.de:8080/musoft/auto?self=$dlv4fi9a, Sept. 2005.

[PRS+94] J. PREECE, Y. ROGERS, H. SHARP, D. BENYON, S. HOLLAND und T. CAREY: *Human-computer interaction*. Addison-Wesley, Wokingham, 1994.

[RCWP] G.-C. ROMAN, K. COX, C. WILCOX und J. PLUN: *Pavane: A System for Declarative Visualization of Concurreent Computations*. Technischer Bericht 91-26.

[Rie96] L. P. RIEBER: *Animation as feedback in a computer-based simulation: Representation matters.* Educational Technology Research & Development, 44(1):5–22, 1996.

[Röß02] G. RÖSSLING: ANIMAL-FARM*: An Extensible Framework for Algorithm Visualization*. Doktorarbeit, University of Siegen, Germany, 2002. Available online at `http://www.ub.uni-siegen.de/epub/diss/roessling.htm`.

[RSF00] G. RÖSSLING, M. SCHÜLER und B. FREISLEBEN: *The* ANIMAL *Algorithm Animation Tool*. 5th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland, Seiten 37–40, Juli 2000.

[SAL04] http://www.learninglab.de/~faltin/SALA/SALA.html, Januar 2004.

[Sch91] A. SCHÜRR: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag, 1991.

[Sch94] A. SCHÜRR: *PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems*. In: *Technical Report AIB 94-11, RWTH Aachen, Germany*, 1994.

[Sch96] A. SCHÜRR: *Introduction to the Specification Language PROGRES*. In: *M. Nagl (Hrsg.): Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach, LNCS 1170*, Seiten 248–279. Springer Verlag Berlin, 1996.

[SCO05] http://www.adlnet.org/index.cfm?fuseaction=scormabt, April 2005.

[SDBP98] J. STASKO, J. DOMINGUE, M. BROWN und B. PRICE: *Software Visualization*. MIT Press, 1998.

[Sha70] A. C. SHAW: *Parsing of Graph-Representable Pictures*. J. ACM, 17(3):453–481, 1970.

[Sim03] http://www.simul8.com/, März 2003.

[Sim05] http://www.mscsoftware.com/products/products_detail.cfm?PI=633 &Q=132&Z=149&Y=32, Feb. 2005.

[SK93] J. T. STASKO und E. KRAEMER: *A methodology for building application-specific visualizations of parallel programs*. J. Parallel Distrib. Comput., 18(2):258–264, 1993.

[Sof04] http://www.softwarevisualisierung.de, Dezember 2004.

[Sor03] http://www.db.fmi.uni-passau.de/Sommercamp2001/Unterlagen/SortDemo.html, März 2003.

[Sta90] J. T. STASKO: *Tango: A Framework and System for Algorithm Animation*. Computer, 23(9):27–39, 1990.

[Sta91] J. T. STASKO: *Using Direct Manipulation to Build Algorithm Animations by Demonstration*. In: *Proceedings of the ACM SIGCHI '91 Conference on Human Factors in Computing Systems*, Seiten 307–314, 1991.

[Sta92] J. STASKO: *Animating algorithms with XTANGO*. SIGACT News, 23(2):67–71, 1992.

[Sta97] J. STASKO: *Using Student-Built Algorithm Animations as Learning Aids*. SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education), 29, 1997.

[Sta03] http://www.stagecast.com/, März 2003.

[Tar77] A. TARSKI: *Einführung in die Mathematische Logik*. Vandenhoeck & Ruprecht, Göttingen, 5 Auflage, 1977.

[Ter01] M. TERADA: *Animating C programs in paper-slide-show.* In: *First International Program Visualization Workshop*, Seiten 79–88. University of Joensuu Press, 2001.

[Til03] http://www.tilcon.com/, März 2003.

[Tra03] P. L. TRAYNOR: *Effects of Computer-Assisted-Instruction On Different Learners*. Journal of Instructional Psychology, 825(30(2)):137–143, 2003.

[TZ01] J. A. TURNER und J. L. ZACHARY: *Javiva: a tool for visualizing and validating student-written Java programs*. In: *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, Seiten 45–49. ACM Press, 2001.

[UF04] T. ULLRICH und D. FELLNER: *AlgoViz - a Computer Graphics Algorithm Visualization Toolkit.* In: *World Conference on Educational Multimedia, Hypermedia and Telecommunications*, Seiten 941–948, 2004.

[UML03] UML REVISION TASK FORCE: *OMG Unified Modeling Language Specification v. 1.5*, März 2003. Document formal / 03-03-01.

[UPG03] http://www-i3.informatik.rwth-aachen.de/upgrade/, März 2003.

[VID04] http://www.musoft.org, Dezember 2004.

[VID05] http://www.es.tu-darmstadt.de, Jul. 2005.

[Wei95] B. WEIDENMANN: *Multimedia, Multicodierung und Multimodalität im Lernprozess*. In: *Arbeiten zur Empirischen Pädagogik u. Pädagogischen Psychologie, Gelbe Reihe, München*, 1995.

[Wei98] T. WEINGÄRTNER: *Eine funktionelle graphische Simulation des Kieferbereichs für die Operationsplanung maxillofacialer Eingriffe*. Doktorarbeit, IPR, 1998.

[WEK03] R. WIESE, M. EIGLSPERGER und M. KAUFMANN: *yFiles: Visualization and Automatic Layout of Graphs*. Chapter in M. Jünger and P. Mutzel (Hrsg.), Graph Drawing Software, Seiten 173–190, 2003.

[wik05] http://de.wikipedia.org/wiki/Computer_Based_Training, Jul. 2005.

[Wir83] N. WIRTH: *Algorithmen und Datenstrukturen*. Teubner Verlag, 1983.

[xyz05] http://www.ipn.uni-kiel.de/persons/michael/xyzet/xyzethome.html, Feb. 2005.

[yfi05] http://www.yworks.com/ger/products.htm, Sept. 2005.

[ZS92] A. ZÜNDORF und A. SCHÜRR: *Nondeterministic Control Structures for Graph Rewriting Systems*. In: *WG '91: Proceedings of the 17th International Workshop*, Seiten 48–62, London, UK, 1992. Springer-Verlag.

[ZWe05] http://vl.zuser.org/#tools, März 2005.