Projektgruppe

Demonstrator InHouse-Energiemanagement

Gesamtdokument

Michael Belasus, Christian Bley, Ulrich Denecke, Mark Eilers,
Michael Erlmann, Moritz Eysholdt, Christian Hinrichs,
Malte Hurrelmann

Betreuer:

Prof. Dr. Michael Sonnenschein
Dr. Ute Vogel
Dipl.-Inform. Jörg Bremer
Dipl.-Inform. Barbara Rapp

Fk. II, Department für Informatik, Abt. Umweltinformatik
an der Carl-Von-Ossietzky Universität, Oldenburg

Stand: 31. März 2008

Inhaltsverzeichnis

. **Tabellenverzeichnis** x

. **Abbildungsverzeichnis** x

I. Einleitung 1

II. Seminararbeiten 2

1. Lastmanagement in privaten Haushalten 2
1.1. Einleitung . . . 2
1.2. Stromlast und Stromspitzen . . . 3
1.3. Stromverbrauch im Privathaushalt . . . 4
1.4. Möglichkeiten des Lastmangement . . . 6
1.5. Eckernförder Tarif . . . 8
1.6. Bidirektionales Energiemanagement Interface . . . 12
1.7. Resümee und Fazit . . . 14

2. Stromerzeugung und Strompreise 16
2.1. Einleitung . . . 16
2.2. Stromerzeugung . . . 16
2.2.1. Generator . . . 16
2.2.2. Dampfturbine . . . 16
2.2.3. Kohlekraftwerk . . . 16
2.2.4. Gasturbine . . . 18
2.2.5. Atomkraftwerk . . . 18
2.2.6. Wasserkraftwerk . . . 20
2.2.7. Windkraft . . . 22
2.2.8. Photovoltaik . . . 23
2.3. Strompreise . . . 24
2.3.1. Sonderverträge . . . 25
2.3.2. Tarifkunden . . . 25
2.4. Zusammenfassung und Fazit . . . 28

3. Klassifikation von Haushaltsgeräten 31
3.1. Einleitung . . . 31
3.2. Exkurs: Nachtspeicherheizungen . . . 31
3.2.1. Einleitung . . . 31
3.2.2. Technische Funktionsweise Nachtspeicherheizung . . . 32
3.2.3. Steuerungsarten . . . 33

3.3. Klassifikationen . . . 35
3.3.1. Klassifikation: Groß-/Kleinelektrogeräte . . . 35
3.3.2. Klassifikation: “weisse”,”braune”, “graue” und “rote” Ware . . . 35
3.3.3. Klassifikation: Energieeffizienzklassen . . . 35
3.3.4. Klassifikation nach [DHA05] . . . 37
3.4. Fazit . . . 39

4. Software-Qualitätsmanagement **41**
4.1. Einleitung . . . 41
4.2. Was ist Software-Qualitätsmanagement? . . . 41
4.2.1. Grundlegende Ansätze im Qualitätsmanagement . . . 42
4.2.2. Vorgehensmethoden in der Qualitätssicherung . . . 42
4.2.3. Validierung und Verifikation . . . 44
4.3. Testen von Software . . . 47
4.3.1. Testbereiche im V-Modell . . . 48
4.3.2. Bestimmen von Testfällen . . . 50
4.3.3. JUnit - Software-Unterstützung für das Testen von Java Code . . . 51
4.4. Inspektionen, Reviews und Walkthroughs . . . 51
4.5. Metriken zum Bewerten von Software . . . 52
4.6. Verbesserung der Prozessqualität . . . 53
4.7. Fazit . . . 53

5. Oberflächenframeworks **55**
5.1. Einleitung . . . 55
5.2. Basis Frameworks . . . 55
5.2.1. AWT - Abstract Window Toolkit . . . 55
5.2.2. Swing . . . 56
5.2.3. SWT - Standard Widget Toolkit . . . 56
5.3. OSGi Plattform . . . 57
5.3.1. Equinox . . . 59
5.4. Rich Client Konzepte . . . 60
5.4.1. Eclipse Rich Client Platform . . . 61
5.4.2. Netbeans platform . . . 63
5.4.3. Spring Rich Client Project . . . 64
5.5. Weitere Tools . . . 65
5.5.1. JFreeChart . . . 65
5.5.2. Jakarta POI . . . 66
5.5.3. SwingWT . . . 66
5.5.4. SWTSwing . . . 67
5.6. Fazit . . . 67

6. Hibernate und Spring **70**
6.1. Einleitung . . . 70

6.1.1. Abgrenzung . . . 70
6.1.2. Gliederung . . . 71
6.2. Hibernate . . . 71
6.2.1. O/R Mapping . . . 72
6.2.2. Implizites SQL . . . 74
6.2.3. Explizites SQL . . . 74
6.3. Spring . . . 75
6.3.1. Philosophie . . . 76
6.3.2. Übersicht . . . 76
6.3.3. Inversion of Control . . . 76
6.3.4. Der IoC Container . . . 77
6.3.5. Datenbankanbindung . . . 79
6.3.6. Wrapper für JDBC . . . 79
6.3.7. Wrapper für O/R Mapping . . . 80
6.3.8. Testen . . . 80
6.3.9. Beispielarchitektur . . . 80
6.3.10. Anforderungen . . . 80
6.3.11. Definitionen . . . 81
6.3.12. Die Architektur . . . 81
6.4. Fazit . . . 82
6.4.1. Hibernate . . . 82
6.4.2. Spring . . . 83
6.4.3. Beispielarchitektur . . . 83

7. Groupware **85**
7.1. Einleitung . . . 85
7.2. Groupware . . . 85
7.2.1. Anforderungen an Groupware . . . 85
7.2.2. Architekturen . . . 86
7.2.3. Softwareentwicklung . . . 87
7.3. Open-Xchange . . . 87
7.3.1. Funktionsumfang . . . 88
7.3.2. Nutzen . . . 89
7.4. Trac . . . 90
7.4.1. Komponenten . . . 90
7.4.2. Entwicklung und Wartung . . . 93
7.4.3. Projektplanung . . . 94
7.4.4. Vorgehensmodelle . . . 94
7.5. Mylar . . . 95
7.6. Schlussfolgerung . . . 97

8. Optimierungsheuristiken **99**
8.1. Einleitung . . . 99

8.2. Kombinatorische Optimierungsprobleme . . . 99
8.3. Exakte Lösungsverfahren . . . 99
8.3.1. Branch & Bound . . . 100
8.3.2. Branch & Bound: Ein Beispiel . . . 100
8.4. Heuristische Lösungsverfahren . . . 104
8.5. Problemspezifische Heuristiken . . . 104
8.5.1. Konstruktionsheuristiken . . . 104
8.5.2. Verbesserungsheuristiken . . . 105
8.6. Metaheuristiken . . . 105
8.6.1. Tabu-Search . . . 106
8.6.2. Tabu-Search: Ein Beispiel . . . 107
8.7. Fazit . . . 109

III. Anforderungsdefinition 110

9. Problemstellung 110

10. Annahmen 110

11. Komponenten 117
11.1. Gerätesammlung . . . 118
11.2. Tarifsammlung . . . 118
11.3. Lastgangsammlung . . . 118
11.4. Profile . . . 118
11.5. Ablaufplanung . . . 119
11.6. Bibliotheken . . . 120
11.7. Server . . . 120

12. Funktionale Anforderungen 121
12.1. Gerätesammlung . . . 121
12.1.1. Geräte verwalten . . . 121
12.1.2. Betriebsmodi verwalten . . . 123
12.2. Tarifsammlung . . . 125
12.2.1. Tarife verwalten . . . 125
12.3. Lastgangsammlung . . . 127
12.3.1. Lastgänge verwalten . . . 127
12.4. Profile . . . 128
12.4.1. Profile verwalten . . . 128
12.4.2. Profil bearbeiten . . . 130
12.5. Ablaufplanung . . . 133
12.5.1. Constraints verwalten . . . 133
12.5.2. Gerätekompositionen verwalten . . . 134

12.5.3. Einsatzpläne verwalten . . . 136
12.5.4. Einsatzpläne vergleichen . . . 139
12.6. Bibliotheken . . . 140
12.6.1. Online-Gerätebibliothek . . . 141
12.6.2. Lokale Gerätebibliothek . . . 141
12.6.3. Online-Tarifbibliothek . . . 142
12.6.4. Lokale Tarifbibliothek . . . 142
12.7. Server . . . 143
12.7.1. Online-Gerätebibliothek verwalten . . . 143
12.7.2. Online-Tarifbibliothek verwalten . . . 145

13. Benutzungsschnittstelle 146
13.1. Profilverwaltung . . . 147
13.2. Einsatzplanung . . . 147
13.3. Statistik . . . 149

14. Nicht-Funktionale Anforderungen 149

IV. Entwurf 151

15. Einleitung 151

16. Frameworks und Technologien 151
16.1. Eclipse Rich Client Platform (RCP) . . . 152
16.2. Eclipse Modeling Framework (EMF) . . . 155
16.2.1. Modell . . . 156
16.2.2. Mögliche Eingaben . . . 156
16.2.3. Codegenerierung . . . 156
16.3. Graphical Editing Framework (GEF) . . . 158
16.4. Hibernate . . . 160
16.5. Teneo . . . 162
16.6. Automatische Einsatzplanoptimierung . . . 163
16.6.1. Ablaufplanung im Kontext . . . 163
16.6.2. Verwendeter Ansatz . . . 168

17. Architektur 169
17.1. Übersicht . . . 169
17.2. Client . . . 170
17.2.1. Plugins . . . 170
17.2.2. Eclipse Platform . . . 171
17.2.3. Persistenz . . . 171
17.2.4. Kommunikation mit dem Server . . . 171

17.3. Server . . . 173

18. Datenmodell **174**
18.1. Codegenerierung . . . 174
18.2. Persistenz . . . 175
18.3. Struktur . . . 176
18.4. Schema loadcurve.xsd . . . 177
18.4.1. Typ LoadCurve . . . 177
18.4.2. Typ Measurement . . . 178
18.5. Schema payscale.xsd . . . 178
18.5.1. Typ PayScale . . . 178
18.5.2. Typ TimePricePair . . . 179
18.6. Schema payscalelib.xsd . . . 180
18.6.1. Typ PayScaleFolder . . . 180
18.6.2. Typ PayScaleLib . . . 180
18.7. Schema device.xsd . . . 181
18.7.1. Typ Device . . . 181
18.7.2. Typ OperationalMode . . . 182
18.8. Schema devicelib.xsd . . . 183
18.8.1. Typ DeviceFolder . . . 183
18.8.2. Typ DeviceLib . . . 183
18.9. Schema devcompos.xsd . . . 184
18.9.1. Typ Constraint . . . 184
18.9.2. Typ DeviceComposition . . . 184
18.9.3. Typ OverlapType . . . 185
18.9.4. Typ Task . . . 185
18.9.5. Typ ConstAfterEachOther . . . 187
18.9.6. Typ ConstMaxLoad . . . 187
18.9.7. Typ ConstOverlap . . . 187
18.10Schema schedule.xsd . . . 188
18.10.1Typ Schedule . . . 188
18.11Schema profile.xsd . . . 188
18.11.1Typ Profile . . . 189
18.12Constraints . . . 190

19. Plugins **191**
19.1. diem.App . . . 192
19.1.1. Activator.java . . . 192
19.1.2. Application.java . . . 192
19.1.3. ApplicationWorkbenchAdvisor.java . . . 192
19.1.4. ApplicationWorkbenchWindowAdvisor.java . . . 192
19.1.5. ApplicationActionBarAdvisor.java . . . 192
19.2. diem.Device . . . 193

19.2.1. Editor . . . 193
19.2.2. Views . . . 193
19.2.3. Actions . . . 195
19.3. diem.PayScale . . . 198
19.3.1. Editor . . . 198
19.3.2. Views . . . 198
19.3.3. Actions . . . 198
19.4. diem.LoadCurve . . . 200
19.4.1. Editor . . . 200
19.4.2. Views . . . 200
19.4.3. Actions . . . 200
19.5. diem.Profile . . . 201
19.5.1. Views . . . 201
19.5.2. Actions . . . 201
19.6. diem.Schedule . . . 203
19.6.1. View . . . 204
19.6.2. Actions . . . 204
19.7. diem.Shared . . . 207
19.7.1. Grafische Editoren . . . 207
19.7.2. View . . . 207
19.7.3. Actions . . . 208
19.8. diem.Model . . . 208
19.9. diem.ModelEdit . . . 208
19.10diem.Logic . . . 209

20. Server **209**
20.1. Komponenten . . . 210
20.2. Management . . . 212
20.3. Sicherheit . . . 213
20.4. Client-Server-Kommunikation . . . 213

21. Programmablauf **214**
21.1. Gerät zur Online-Bibliothek hinzufügen . . . 214
21.2. Gerätekomposition erstellen . . . 218
21.3. Tarif bearbeiten . . . 222

22. Qualitätssicherung **225**
22.1. Client . . . 226
22.1.1. diem.App . . . 227
22.1.2. diem.Device . . . 228
22.1.3. diem.PayScale . . . 229
22.1.4. diem.LoadCurve . . . 230
22.1.5. diem.Shared . . . 230

22.1.6. diem.Profile . . . 231
22.1.7. diem.Schedule . . . 232
22.1.8. diem.Logic . . . 233
22.1.9. diem.ModelEdit . . . 234
22.1.10. diem.Model . . . 234
22.2. Server . . . 234
22.2.1. DeviceServlet . . . 235
22.2.2. PayScaleServlet . . . 235
22.2.3. ManagerServlet . . . 236
22.2.4. ResourceHandler . . . 236
22.2.5. Sicherheit und Verfügbarkeit . . . 236

V. Prototypen 238

23. Aufteilung der Prototypen 238
23.1. Erster Prototyp . . . 238
23.2. Zweiter Prototyp . . . 239
23.3. Dritter Prototyp . . . 239

24. Prototyp 1 240

25. Abweichungen des ersten Prototypen 242
25.1. Datenmodell . . . 242
25.2. GUI & Layout . . . 243
25.2.1. Aufteilung in Perspektiven . . . 243
25.2.2. Änderungem der Editoren und Views . . . 248
25.2.3. Implementierung des Layouts . . . 248
25.3. EMF.edit . . . 249
25.4. JFace Databinding . . . 250
25.5. Server . . . 251
25.5.1. Zusätzliche Funktionen . . . 251
25.5.2. Zusätzliche Komponenten . . . 252
25.5.3. Persistenz . . . 252
25.6. Graphical Editing Framework . . . 253

26. Prototyp 2 256
26.1. Einleitung . . . 256
26.2. Funktionale Anforderungen 2. Prototyp . . . 256
26.3. Ergänzende Funktionale Anforderungen aus dem 1. Prototyp . . . 257

27. Änderungen und Erweiterungen zum Entwurf durch Prototyp 2 258
27.1. Tarif . . . 258

27.2. Profilperspektive . . . 259
27.3. Visuelle Statistik . . . 260
27.3.1. View . . . 261
27.3.2. Aktions . . . 262
27.4. Automatische Einsatzplanoptimierung . . . 263
27.5. Dokumentation . . . 266
27.5.1. Technische Umsetzung . . . 267
27.5.2. Handbuch . . . 267
27.5.3. Tutorial . . . 267
27.6. Server . . . 269
27.6.1. Management . . . 270

28. Prototyp 3 **273**
28.1. Änderungen und Erweiterungen zum Entwurf durch Prototyp 3 . . . 273
28.2. Numerische Statistik . . . 273
28.3. Serien . . . 274
28.4. Gerätesubstitution . . . 275
28.5. Einsatzplan drucken . . . 276
28.6. Wizards . . . 277
28.7. Server . . . 278
28.7.1. Importer Schnittstelle . . . 279
28.7.2. Day-Ahead Tarife . . . 280
28.7.3. Protokollierung . . . 281

VI. Feinentwürfe **282**

29. Feinentwurf grafische Editoren **282**
29.1. Einleitung . . . 282
29.2. Paketübersicht . . . 282
29.3. Klassenübersicht . . . 283

30. Feinentwurf Logikpaket **291**
30.1. Einleitung . . . 291
30.2. Schnittstelle . . . 291
30.3. Automatische Optimierung . . . 292
30.3.1. Implementierung des Choco-Solvers . . . 293
30.4. Zustandsüberprüfung für Constraints . . . 294
30.5. Statistische Funktionen . . . 295
30.5.1. Numerische Statistik . . . 295
30.5.2. Graphische Statistik . . . 295

31. Strukturbeschreibung des Einsatzplaneditors **296**

31.1. Einleitung . . . 296
31.2. Paketübersicht . . . 296
31.3. Klassenübersicht . . . 302
31.4. Kommunikation mit dem Datenmodell . . . 310
31.5. Die Klasse `TimeOperationFigure` . . . 311
31.5.1. Attribute . . . 311
31.5.2. Methoden . . . 313
31.5.3. Innere Klasse `ConnectorLayouter` . . . 314
31.6. Abschlussbetrachtung . . . 314

VII. Testen **316**

32. Systemtests **316**

33. Unit-Tests **316**
33.1. Tests des Pakets `diem.logic` . . . 317
33.1.1. `deriveLoadcurveFromSchedule(Schedule, LoadCurve)` . 317
33.1.2. `getCostOfLoadCurveByPayScale(LoadCurve, PayScale, boolean)` . . . 317
33.1.3. `getMaxOfLoad(Schedule, LoadCurve)` . . . 319
33.1.4. `getMiddleCostOfPayScaleByLoadCurve(LoadCurve, PayScale, boolean)` . . . 319
33.1.5. `getMiddleLoadCurveValue(LoadCurve)` . . . 320
33.1.6. `getMiddlePayScaleValue(PayScale)` . . . 321
33.1.7. `getMaxPayScaleValue(PayScale)` . . . 321
33.1.8. `getMaxLoadCurveValue(LoadCurve)` . . . 322
33.1.9. `getMinPayScaleValue(PayScale)` . . . 322
33.1.10. `getMinLoadCurveValue(LoadCurve)` . . . 322
33.1.11. `getMostExpensiveTask(Schedule)` . . . 323
33.1.12. `getTotalLoadOfLoadCurve(LoadCurve)` . . . 324
33.1.13. `calcConstraintStatus(DeviceComposition)` . . . 324
33.1.14. `calcTaskPrice(Task, Schedule)` . . . 325

VIII. Gerätedaten **326**

34. Quellen für Gerätedaten **326**

35. Gerätegruppierungen **327**

IX. Ausblick **328**

Abbildungsverzeichnis

1. Winter-Sommer Spitzenlastverlauf in Eckernförde [Pra97] . . . 3
2. Spitzenlastverlauf ohne Lastmanagement (links), und mit Lastmanagement (rechts) [Hei01] . . . 4
3. Aufbau der Komponenten im Eckernförder Tarif [Pra97] . . . 9
4. Bereitschaft den Tagesablauf zu ändern [Pra97] . . . 10
5. Zusammenhang Abrechnungsergebnis und Wahltarif [Pra97] . . . 11
6. Kommunikation und Handel im liberalisierten Strommarkt mit Integration verteilter Erzeugung [CB06] . . . 13
7. Kommunikationsstruktur innerhalb BEMI [CB06] . . . 14
8. Generator zur Erzeugung von dreiphasigem Wechselstrom (Quelle: [Gen07]) 17
9. Aufbau einer Dampfturbine (Quelle: [Dam07]) . . . 18
10. Aufbau eines Kohlekraftwerkes (Quelle: [SCZ07]) . . . 19
11. Aufbau einer Gasturbine (Quelle: [Gas07]) . . . 20
12. Aufbau eines Druckwasserreaktors (Quelle: [SCZ07]) . . . 21
13. Prinzip Laufwasserkraftwerk (Quelle: [Was07]) . . . 22
14. Prinzip Speicherkraftwerk (Quelle: [Was07]) . . . 23
15. Prinzip Pumpspeicherkraftwerk (Quelle: [Was07]) . . . 24
16. Gondel einer Windkraftanlage(Quelle: [Win07]) . . . 25
17. Aufbau einer Solarzelle (Quelle: [Sol07]) . . . 26
18. Lastschwankungen (Quelle: [Las07]) . . . 27
19. Strompreis an der EEX im Tagesverlauf (Quelle: [Str05]) . . . 28
20. CD aus dem Beispiel . . . 32
21. Vorwärtssteuerung . . . 33
22. Rückwärtssteuerung . . . 34
23. Spreizsteuerung . . . 34
24. Energielabel . . . 36
25. Klassifikation . . . 37
26. Das V-Modell . . . 44
27. Validierung und Verifikation in der Qualitätssicherung . . . 44
28. Kontrollflussgraph im Whitebox-Verfahren . . . 46
29. Treiber und Stümpfe im Integrationstest . . . 49
30. AWT Beispielanwendung . . . 56
31. Swing Beispielanwendung . . . 57
32. SWT Beispielanwendung . . . 58
33. Struktur der Eclipse IDE . . . 59
34. Komponenten der Eclipse IDE . . . 60
35. Poseidon for UML als Beispiel einer Netbeans RCP Anwendung . . . 64
36. Teil einer Spring-Rich Anwendung . . . 65
37. JFreeChart Diagramm . . . 66
38. SWTSwing Beispiel . . . 68
39. HibernateLogo . . . 71

40. ORM . . . 72
41. SpringLogo . . . 75
42. SpringOverview . . . 77
43. ExampleArchitecture . . . 82
44. Ein Ticket in Trac - Screenshot von [Edg07a] . . . 92
45. Ein Meilenstein in Trac - Screenshot von [Edg07a] . . . 93
46. Tracs Timelinefunktion - Screenshot von [Edg07a] . . . 93
47. Mylar im Eclipseworkbench . . . 96
48. Problem des Handelsreisenden . . . 101
49. Branch & Bound: Der Anfangszustand . . . 102
50. Branch & Bound: Der Endzustand . . . 103
51. Komponenten . . . 117
52. Anwendungsfälle Geräteverwalten . . . 121
53. Anwendungsfälle Betriebsmodi verwalten . . . 123
54. Anwendungsfälle Tarife verwalten . . . 125
55. Anwendungsfälle Lastgang verwalten . . . 127
56. Anwendungsfälle Profile verwalten . . . 129
57. Anwendungsfälle Profil bearbeiten . . . 131
58. Anwendungsfälle Constraints verwalten . . . 133
59. Anwendungsfälle Gerätekompositionen verwalten . . . 134
60. Anwendungsfälle Einsatzpläne verwalten . . . 136
61. Anwendungsfälle Einsatzpläne vergleichen . . . 139
62. Anwendungsfall Online-Gerätebibliothek . . . 141
63. Anwendungsfall lokale Gerätebibliothek . . . 142
64. Anwendungsfall lokale Tarifbibliothek . . . 143
65. Anwendungsfälle Online-Gerätebibliothek verwalten . . . 144
66. Anwendungsfälle Online-Tarifbibliothek verwalten . . . 145
67. Profilverwaltung . . . 147
68. Einsatzplanung . . . 148
69. Statistik . . . 149
70. RCP Logo . . . 152
71. Struktur einer RCP Anwendung . . . 153
72. EMF Logo . . . 155
73. EMF Übersicht . . . 155
74. GEF Logo . . . 158
75. GEF Übersicht . . . 159
76. Hibernate Logo . . . 160
77. Stuktur einer Hibernate Anwendung . . . 161
78. Elver Logo . . . 162
79. Teneo Mapping . . . 163
80. Teneo Laufzeitstruktur . . . 164
81. Client Server Struktur . . . 169
82. Client Struktur . . . 172

83. Server Struktur . . . 173
84. Typ LoadCurve . . . 178
85. Typ PayScale . . . 179
86. Typ PayScaleLib . . . 180
87. Typ Device . . . 181
88. Typ DeviceLib . . . 183
89. Typ DeviceComposition . . . 185
90. Typ ConstOverlap . . . 188
91. Typ Schedule . . . 189
92. Typ Profile . . . 189
93. Plugin Klassenstruktur . . . 194
94. Der Gerätesammlung-View . . . 194
95. Der Geräte-Editor . . . 194
96. Der Profil-View . . . 202
97. Der Einsatzplan-View . . . 205
98. Der Statistik-View . . . 216
99. Serverstruktur . . . 216
100. Sequenzdiagramm Gerät zur Online-Bibliothek hinzufügen . . . 217
101. Sequenzdiagramm neue Gerätekomposition erstellen . . . 220
102. Sequenzdiagramm zu der Action "'Tarif bearbeiten"' . . . 223
103. Das V-Modell . . . 226
104. Die Profilperspektive . . . 245
105. Die Perspektivenleiste . . . 245
106. Speichern bei Perspektivenwechsel . . . 246
107. Die Einsatzplanperspektive . . . 247
108. Databinding zwischen Eingabefeld und Model . . . 251
109. Ein GEF-Standard Editor . . . 253
110. Der Einsatzplan-Editor . . . 255
111. Tarife . . . 258
112. Die geänderte Profilperspektive . . . 260
113. Die Statistik-Perspektive . . . 261
114. Die erweiterte Statistik-Perspektive . . . 274
115. Klassendiagramm der grafischen Editoren . . . 283
116. Schematischer Aufbau der grafischen Editoren . . . 298
117. Die Schnittstelle "'LogicFacade"' . . . 298
118. Interfaces für Lösungsalgorithmen . . . 299
119. Klassen-Diagramm zur Implementierung des Choco-Solvers . . . 299
120. Klassen-Diagramm zur Implementierung des Algorithmus zur Zustandsüberprüfung der Constraints . . . 300
121. Klassen-Diagramm zur Implementierung der numerischen Statistik . . . 300
122. Klassen-Diagramm zur Implementierung der grapischen Statistik . . . 300
123. Die Paketstruktur des Einsatzplaneditors . . . 301
124. Die verschiedenen Konnektorentypen . . . 307

125. Gesamtansicht der Klassen . . . 309
126. Quelltext des `Adapter` Interfaces . . . 310
127. Klassendiagramm zur TimeOperationFigure . . . 311

Tabellenverzeichnis

1. durchschnittlicher Stromverbrauch in unterschiedlichen Personenhaushalten . . . 5
2. Haushaltsstromverbrauch nach Anwendungsarten, Angaben in % . . . 5
3. Exchange Lösungen . . . 86
4. Funktionsumfang von Open-Xchange . . . 88
5. Felder eines Tickets . . . 91

12.1.1 Geräte verwalten **121**
12.1.1 a) Gerät erstellen . . . 122
12.1.1 b) Gerät bearbeiten . . . 122
12.1.1 c) Gerät kopieren . . . 122
12.1.1 d) Gerät löschen . . . 122
12.1.1 e) Gerätedaten anzeigen . . . 122

12.1.2 Betriebsmodi verwalten **123**
12.1.2 a) Betriebsmodus erstellen . . . 123
12.1.2 b) Betriebsmodus bearbeiten . . . 123
12.1.2 c) Betriebsmodus kopieren . . . 124
12.1.2 d) Betriebsmodus löschen . . . 124
12.1.2 e) Lastgang importieren . . . 124
12.1.2 f) Durchschnittsverbrauch aus Gerätelastgang berechnen . . . 124

12.2.1 Tarife verwalten **125**
12.2.1 a) Fixtarif erstellen . . . 125
12.2.1 b) Fixtarif bearbeiten . . . 125
12.2.1 c) Variablen Tarif erstellen . . . 126
12.2.1 d) Variablen Tarif bearbeiten . . . 126
12.2.1 e) Tarif anzeigen . . . 126
12.2.1 f) Tarif kopieren . . . 126
12.2.1 g) Tarif löschen . . . 126

12.3.1 Lastgänge verwalten **127**
12.3.1 a) Lastgang erstellen . . . 127
12.3.1 b) Lastgang bearbeiten . . . 127
12.3.1 c) Lastgang kopieren . . . 128
12.3.1 d) Lastgang löschen . . . 128

12.3.1 e) Lastgang anzeigen 128

12.4.1 Profile verwalten **128**
12.4.1 a) Profil erstellen 129
12.4.1 b) Profil laden 129
12.4.1 c) Profil löschen 130
12.4.1 d) Profil kopieren 130
12.4.1 e) Profileigenschaften bearbeiten 130

12.4.2 Profil bearbeiten **130**
12.4.2 a) Gerät hinzufügen 131
12.4.2 b) Gerät löschen 131
12.4.2 c) Tarif hinzufügen 132
12.4.2 d) Tarif löschen 132
12.4.2 e) Lastgang hinzufügen 132
12.4.2 f) Lastgang löschen 132

12.5.1 Constraints verwalten **133**
12.5.1 a) Constraint hinzufügen 133
12.5.1 b) Constraint bearbeiten 134
12.5.1 c) Constraint löschen 134

12.5.2 Gerätekompositionen verwalten **134**
12.5.2 a) Gerätekomposition erstellen 135
12.5.2 b) Gerätekomposition kopieren 135
12.5.2 c) Gerätekomposition löschen 135
12.5.2 d) Betriebsmodus hinzufügen 135
12.5.2 e) Betriebsmodus löschen 136

12.5.3 Einsatzpläne verwalten **136**
12.5.3 a) Einsatzplan erstellen 137
12.5.3 b) Einsatz planen 137
12.5.3 c) Tarif zum Einsatzplan hinzufügen 137
12.5.3 d) Tarif aus Einsatzplan löschen 137
12.5.3 e) Einsatzplan optimieren 138
12.5.3 f) Zyklus festlegen 138
12.5.3 g) Zyklus ändern 138
12.5.3 h) Zyklus löschen 138
12.5.3 i) Einsatzplan drucken 138

12.5.4 Einsatzpläne vergleichen **139**
12.5.4 a) Erwarteten mit gemessenem Stromverbrauch vergleichen 139
12.5.4 b) Zwei Einsatzpläne visuell vergleichen 140
12.5.4 c) Einsatzpläne preislich vergleichen 140

12.5.4 d) Geräte substituieren 140

12.6.1 Online-Gerätebibliothek **141**
12.6.1 a) Gerätedaten zur Online-Gerätebibliothek hinzufügen 141

12.6.2 Lokale Gerätebibliothek **141**
12.6.2 a) Gerätedaten aus Internet beziehen 142

12.6.4 Lokale Tarifbibliothek **142**
12.6.4 a) Tarifdaten aus dem Internet beziehen 143

12.7.1 Online-Gerätebibliothek verwalten **143**
12.7.1 a) Geräte auflisten 144
12.7.1 b) Gerät verifizieren 144
12.7.1 c) Gerät löschen 144
12.7.1 d) Gerät als Datei vom Server herunterladen 145
12.7.1 e) Gerät als Datei auf den Server übertragen 145

12.7.2 Online-Tarifbibliothek verwalten **145**
12.7.2 a) Tarife anzeigen 146
12.7.2 b) Tarif als Datei vom Server herunterladen 146
12.7.2 c) Tarif als Datei auf den Server übertragen 146
12.7.2 d) Tarif löschen 146
6. Erlaubte Zeichen in Ordnernamen 270
7. Eigenschaften eines `ImportJob` 279
8. Mögliche Aktionen des `JobServlets` über `action`-Parameter 281

Teil I.
Einleitung

Autor: Mark Eilers

Eines der großen Probleme der heutigen Zeit ist die zunehmende Verknappung der natürlichen Rohstoffe und ein gleichzeitig stetig wachsender Energiebedarf. Um dieses Problem zu minimieren, gibt es neben dem Finden neuer Wege in der Energiegewinnung auch die Möglichkeit den derzeitigen Energieverbrauch zu reduzieren. Hilfreich hierfür wäre eine Vermeidung von Lastspitzen durch die Umverteilung eines Anteils des privaten Stromverbrauchs in Spitzenzeiten auf weniger laststarke Zeiträume. Dies könnte beispielsweise durch die Einführung von last- und zeitabhängigen Stromtarifen geschehen, die den Verbrauchern die Nutzung weniger stark beanspruchter Zeiträume durch geringere Strompreise attraktiver erscheinen lassen könnten.

Autor: Michael Belasus und Christian Bley

Im Folgenden wird mit den Seminararbeiten der Projektgruppenmitglieder begonnen. Die Seminararbeiten behandeln Themen, die für den Einstieg in das Thema Energieverbrauch und Softwareentwicklung helfen.
In Teil III wird mit der Anforderungsdefinition auf die geforderten Kernelemente des Projektes eingegangen und das gewünschte Verhalten der Software definiert.
Der Entwurf in Teil IV definiert die Umsetzung der Anforderungsdefinition und erläutert die verwendeten Frameworks und Tools. Der Entwurf ist als Grobentwurf zur gesamten Software verfasst. Feinere Erläuterungen und genaue Definitionen werden in den einzelnen Entwurfszusätzen zu den Prototypen in Teil V dargelegt.
Die zentralen Elemente der Software werden in Feinentwürfen in Teil VI definiert.
Teil VII enthält die Testberichte zu den Unit- und Systemtests der Projektgruppe.
In Teil VIII wird auf die verwendeten Gerätedaten eingegangen, die als Grundlage für die Gerätebibliothek verwendet werden.
Zum Schluss folgt der Ausblick auf die mögliche Erweiterung und Nutzung der Software in einem anderem Kontext, also zuvor beabsichtigt wurde.

Teil II.
Seminararbeiten

1. Lastmanagement in privaten Haushalten

Autor: Michael Belasus

1.1. Einleitung

Diese Projektgruppe hat das Ziel eine Software zu entwickeln, die privaten Haushalten bei der Planung von Geräteeinsätzen behilflich ist. Dabei steht die Einsparung von Stromkosten für den Anwender im Vordergrund. Derzeit bieten Energieversorger keine zeitvariablen Tarife an, sodass private Anwender mit Hilfe von Geräteeinsatzplänen noch nicht sparen können. Es liegt aber im Interesse der Energieversorger die Auslastung des Energieversorgungsnetzes ohne Lastspitzen zu unterhalten. Dazu ist Lastmanagement, wie es bereits in der Industrie angewandt wird, auch in Privathaushalten notwendig. Dies kann vor allem dadurch erreicht werden, wenn lastschwächere Zeiten billiger sind. Unter der Vorraussetzung, dass zeitvariable Stromtarife angeboten werden, können auch private Haushalte durch Lastmanagement Stromkosten sparen.

Zu Beginn dieser Seminararbeit wird erläutert was Lastspitzen im Energieversorgungsnetz bedeuten und warum diese teuer und umweltschädlich sind. Dabei wird der Begriff Lastmanagement verdeutlicht. Es werden erste Vorschläge für Lastverlagerung gegeben. Im Anschluss wird auf den Stromverbrauch in privaten Haushalten eingegangen. Dabei wird betrachtet welche Anwendungsarten im Privathaushalt prozentual wie viel Strom verbrauchen, um Möglichkeiten für Lastverlagerung aufzuzeigen. Aus diesen Erkenntnissen lassen sich verschiedene Möglichkeiten für Lastmanagement ableiten. Diese werden anschließend näher betrachtet. Dabei werden auch Maßnahmen vorgestellt, die bereits umgesetzt worden sind. Zwei dieser Maßnahmen werden dann im Detail näher betrachtet. Unter anderem gab es in den neunziger Jahren einen großen Feldversuch in Eckernförde, der mit einem lastabhängigen Tarif Lastmanagement in privaten Haushalten untersucht hat. Das zweite Lastmanagementsystem, welches näher vorgestellt wird, ist das Bidirektionale Energiemanagement Interface (BEMI). Dabei handelt es sich um eine technische Umsetzung die Last im Haushalt zu messen und zu kontrollieren.

1.2. Stromlast und Stromspitzen

Heutzutage kann Energie auf verschiedene Art und Weise erzeugt, aber nicht in beliebigen Mengen gespeichert werden. Die Energieversorger müssen immer auf den aktuellen Bedarf der Verbraucher reagieren. Dieser Verbrauch ist tageszeit- und jahreszeitabhängig. Die Abbildung 1 zeigt deutlich, dass der Verbrauch im Winter höher als im Sommer ist. Auch über den Tag verteilt ist zu erkennen, dass zum Mittag und zum Abend Spitzen auftreten. Am Wochenende ist der Gesamtverbrauch deutlich geringer als in der Woche. Daraus lässt sich bereits Potenzial für die Verlagerung von Lasten in die Nacht, in den Abend und ins Wochenende ableiten.

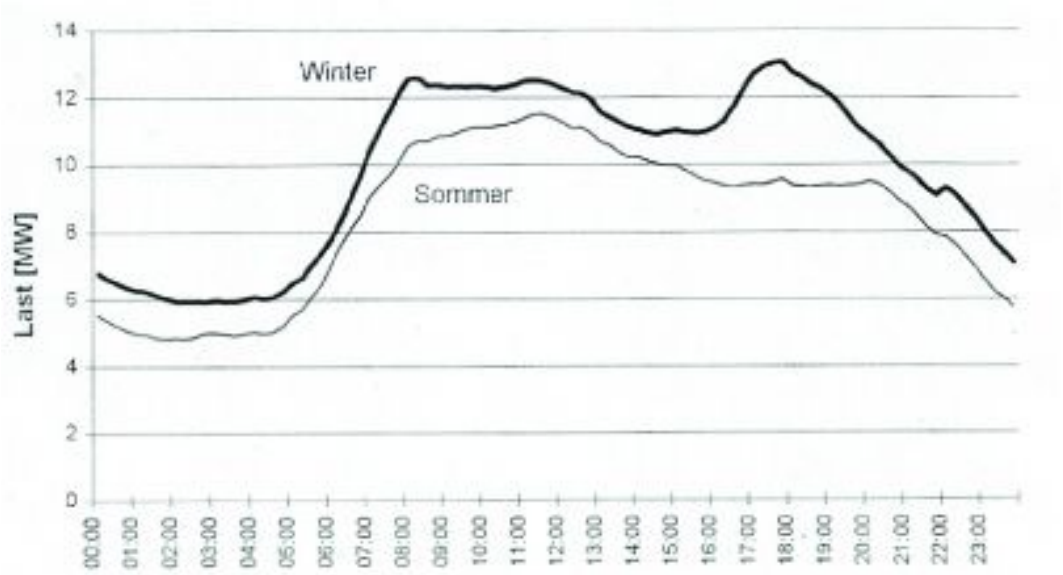

Abbildung 1: Winter-Sommer Spitzenlastverlauf in Eckernförde [Pra97]

Die Industrie hat an diesem Verbrauch den stärksten Anteil, daher gibt es bereits spezielle Tarife und Lastmanagementsysteme für die Industrie, damit Spitzen im Stromversorgungsnetz vermieden werden. Im Zusammenhang mit der Klimaerwärmung und dem sich daraus resultierenden Klimaschutz sind auch private Haushalte zum Strom sparen, insbesondere zur Vermeidung oder Verringerung von Lastspitzen aufgefordert. Zunächst soll erläutert werden, warum solche Spitzenlasten so teuer sind.
Die Stromversorgung erfolgt über drei Kraftwerkstypen mit unterschiedlichen Kosten, verschiedenen Umweltbeeinflussungen und Einsatzzeiten. Die so genannten Grundlastkraftwerke werden ganzjährig betrieben und stellen die Grundversorgung sicher. Sie sind im Verhältnis zu den anderen beiden Kraftwerkstypen kostengünstig und belasten die Umwelt am geringsten in Bezug auf den CO_2-Ausstoß. Wenn die Netzlast über die Grundversorgung steigt, werden zusätzlich Mittellastkraftwerke geschaltet. Diese haben höhere und variable Kosten. Sie belasten die Umwelt stärker als Grundlastkraftwerke. Um sehr hohe Spitzen im Netz abzufangen, gibt es als drittes die Spit-

zenlastkraftwerke. Diese verursachen sehr hohe Kosten und einen sehr hohen CO_2 Ausstoß. Sie werden daher nur kurzfristig eingesetzt, um Spitzen im Netzwerk abzudecken.

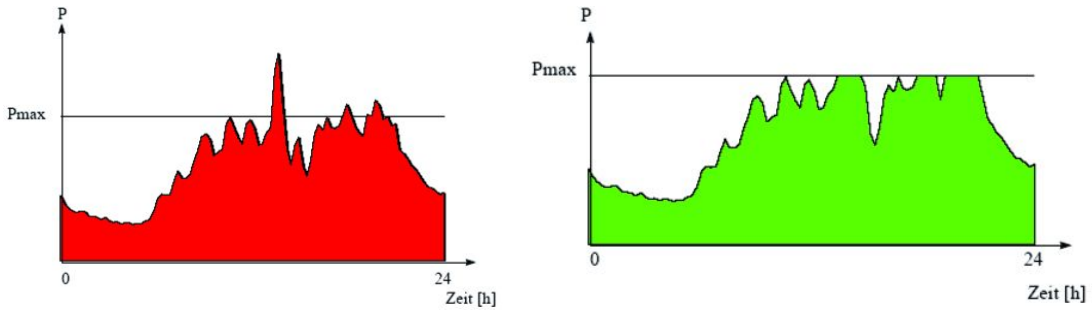

Abbildung 2: Spitzenlastverlauf ohne Lastmanagement (links), und mit Lastmanagement (rechts) [Hei01]

Solche Lastspitzen sind in der Abbildung 2 (links) dargestellt. Wünschenswert ist aber ein Kurvenverlauf, wie in Abbildung 2 (rechts) dargestellt. Die teuren und umweltschädlichen Lastspitzen sind abgeschnitten. Die Vermeidung solcher Spitzen bewirkt, dass die teuren und umweltschädigenden Spitzenlastwerke nicht oder nur in geringerem Ausmaß betrieben werden müssen. Dies kann erreicht werden, indem durch Lastmanagement die Netzlast kontrolliert und der Einsatz von Geräten auf eine andere Zeit verschoben wird. Damit kann eine gleichmäßigere Auslastung der Kraftwerke ereicht werden. Die Stromerzeugung wird kostengünstiger und vor allem umweltfreundlicher, da weniger CO_2 bei der Stromerzeugung produziert und an die Umwelt abgegeben wird.

1.3. Stromverbrauch im Privathaushalt

In privaten Haushalten steigt die Ausstattung an technischen Haushaltsgeräten jährlich. 1980 besaßen 8% der Haushalte einen Wäschetrockner. 2006 lag dieser Wert bei 44 %, damit hat fast jeder zweite Haushalt einen Wäschetrockner. Noch extremere Steigerungen gelten für die Anzahl von Computern und die gesamte Unterhaltungselektronik in privaten Haushalten. Daraus resultiert der Anstieg des Stromverbrauches. Verbrauchte ein Zweipersonenhaushalt in den achtziger Jahren durchschnittlich noch 1495 kWh im Jahr, so waren es 2006 bereits 1900 kWh im Jahr. In der Tabelle 1 ist ein zeitlicher Verlauf für den Anstieg des durchschnittlichen Stromverbrauches für verschiedene Haushaltsgrößen dargestellt. Die Werte von 2006 sind den Angaben der EnergieAgentur.NRW und dem Bund der Energieverbraucher [1] entnommen. Die Daten für die achtziger und

[1] www.energieverbraucher.de

neunziger Jahre wurden aufgrund der vom Bund der Enrgerieverbraucher angegebenen durchnittlichen Steigerungen berechnet.

	Personenhaushalte			
	Ein	Zwei	Drei	Vier
80-er Jahre	944 kWh	1495 kWh	1967 kWh	2675 kWh
90-er Jahre	1102 kWh	1745 kWh	2296 kWh	3123 kWh
2006	1200 kWh	1900 kWh	2500 kWh	3400 kWh

Tabelle 1: durchschnittlicher Stromverbrauch in unterschiedlichen Personenhaushalten

Es ist zu erkennen, dass der Stromverbrauch jeweils ansteigt. Laut dem Verband der Elektrizitätswirtschaft (VDEW) rechnen die Energieversorger zukünftig mit einem Anstieg von jährlich circa 0,5%. Der VDEW führt diesen Trend auf den Einsatz von effizienteren Geräten und dem wachsenden Energiebewusstsein der Verbraucher zurück. Selbst mit dem Einsatz von modernen energiesparenden Geräten können Lastspitzen auftreten, sofern viele Geräte zur selben Zeit eingeschaltet und betrieben werden. Daher ist Lastmanagement notwendig, um einer wirtschaftliche und umweltfreundliche Energieversorgung zu verwirklichen.

Anwendung	**1986**	**2001**	**2006**
Beleuchtung	8,8	7	11,05
kühlen & gefrieren	21,6	16,9	15,76
Kühlschrank		8,6	10,34
Gefriergerät		8,3	5,42
Elektroherd	9,1	7,7	8,38
waschen & trocknen	7,2	6	15,17
Waschmaschine		3,6	5,1
Wäschetrockner		2,4	10,07
Geschirrspüler	2,7	2,7	5,37
Warmwasser elektr.	14,1	11,2	11,5
TV, Audio, Video, PC	6,5	5,8	11,14
Büro			12,18
elektr. heizen	4,2	2,2	5,37
Sonstige Geräte	19,8	20,1	
Sonst. Verbrauch	6	4,8	3,9

Tabelle 2: Haushaltsstromverbrauch nach Anwendungsarten, Angaben in %

Im Folgenden wird genauer betrachtet, auf welche Anwendungsbereiche im privaten Haushalt sich der Stromverbrauch aufteilt. Die Tabelle 2 stellt eine Auflistung von Daten

aus den Jahren 1986, 2001 und 2006 dar. Die Angaben sind aus den Quellen [Ene06], [DTK07] und dem Bund der Energieverbraucher entnommen.
Auffällig ist die Klassifizierung von 1986, die im Vergleich zu den Auflistungen 2001 und 2006 allgemeiner gehalten wurde. So wurde nicht zwischen Kühlschrank und Gefrierschrank unterschieden, sowie zwischen Waschmaschine und Wäschetrockner. Auch ist der Bereich der Unterhaltungselektronik und Büro zusammengefasst worden. Erkennbar sind die fallenden Tendenzen im Bereich Kühlen & Gefrieren, Waschen & Trocknen und Kochen. Dies ist aufgrund steigender Gerätesättigung im Haushalt vor allem auf effizientere Geräte zurück zuführen. Ein sehr auffälliger Bereich für steigenden prozentuellen Stromverbrauch sind die Bereiche TV, Audio, Video und PC. Dieser lag 1986 bei 6,5% und beträgt 2006 bereits 11,14%, wobei hier sogar ein extra Bereich Büro mit 12,18% eingeführt wurde. Dies ist vor allem darauf zurück zuführen, dass immer mehr Unterhaltungselektronik entwickelt wird und die Preise im Vergleich zu 1986 gesunken sind. Dadurch schaffen sich immer mehr Haushalte Zweit- und auch Drittgeräte an.
Auf Grund der Daten von 2006 fallen circa 44% auf die Bereiche Waschen, Trocknen, Kühlen und Gefrieren. Die zugehörigen Geräte gehören in die Klassifikation Haushaltsgroßgeräte. Die Bereiche Unterhaltungselektronik und Büro liegen mit circa 33% an zweiter Stelle. Dies sind die Bereiche die für eine Einsparung oder Lastverlagerung näher betrachtet werden müssen, da sie circa zwei Drittel des Gesamtstromverbrauches umfassen. Innerhalb dieser Bereiche sind ebenfalls, die steigenden Geräteanzahlen im Privathaushalt zu finden. Bei der Unterhaltungs- und Büroelektronik sind die höchsten StandBy Stromverbräuche zu finden, da diese Geräte meistens per Fernbedienung eingeschaltet werden.

1.4. Möglichkeiten des Lastmangement

In den vorherigen Abschnitten wurde beschrieben, warum Lastspitzen im Energieversorgungsnetz vermieden werden sollten. Dies kann zum einen durch Verschiebung der Last in lastschwächere Zeiten und zum anderen durch Verringerung der Last geschehen. Die Abbildung 1 verdeutlicht in welche Zeiten die Lasten verschoben werden könnten. Eine Betrachtung des Stromverbrauches in privaten Haushalten zeigte die Bereiche mit den größten Anteilen am Gesamtstromverbrauch. Die Verschiebung von Last in eine andere Tageszeit und die Reduktion der Last als Reaktion auf die zu hohe Gesamtlast wird als Lastmanagement bezeichnet. Dies kann zum einem durch den Anwender erfolgen und zum anderen automatisiert werden.
Damit der Anwender die Nutzung eines Gerätes in eine lastschwächere Zeit verschiebt, muss er entsprechend motiviert werden. Dies wird erreicht indem attraktive und zeitabhängige Strompreise etabliert werden. Sofern der private Stromkunde weiß, dass am Abend oder am Wochenende der Strom billiger ist, verschiebt er bestimmte Stromverbräuche in diesen Zeitraum.

Eine einfache technische Lösung zur automatisierten Lastverlagerung ist der Maximumwächter oder auch Lastschalter genannt. Dieses Gerät vergleicht die Ist-Leistung mit der eingestellten Soll-Leistung. Beim Übersteigen der eingestellten Last wird der angeschlossene Verbraucher abgeschaltet. Sofern die Last wieder sinkt, wird der Verbraucher wieder eingeschaltet.
Dabei muss beachtet werden, dass nicht jeder beliebige Verbraucher einfach während des Betriebes abgeschaltet werden kann. Zum einen kann es zusätzliche Stromkosten verursachen und zum anderen können Defekte auftreten. Bei einer Unterbrechung des Wasch- oder Spülvorganges muss nach dem erneuten Einschalten beispielsweise das Wasser erneut angeheizt werden. Dadurch wird doppelt Strom verbraucht und dies wiederum belastet die Stromrechnung des Privathaushaltes unnötig. Ein plötzliches Abschalten kann darüber hinaus einen Gerätedefekt verursachen. So kann beispielsweise das abrupte Abschalten der Kühlung eines Beamers zum Defekt der Beamerlampe führen. Denkbar ist auch ein eventueller Datenverlust, falls die Stromversorgung für den Computer aufgrund des automatischen Lastmanagements abgeschaltet wird. Bei solchen möglichen Defekten befürchten viele Privathaushalte um ihre Geräte und in einigen Fällen auch um ihren Versicherungsschutz. Hinzu kommt der Wunsch nach Eigenständigkeit. Eine häufige Aussage ist dabei: ”Ich lasse mir doch nicht in meinen Haushalt hineinreden.”
Bei einer automatischen Lastverschiebung muss außerdem beachtet werden, dass der Betrieb einiger Verbraucher nicht in jede beliebige Tageszeit erfolgen kann. Eine Waschmaschine beispielsweise kann in einem Mehrfamilienhaus nicht immer beliebig benutzt werden. Viele Hausordnungen weisen aus, dass nach 22 Uhr solche und andere Tätigkeiten unterlassen werden sollen. Daher kann nicht in jede beliebige Zeit verschoben werden. Wie in der Abbildung 1 zu erkennen, sind in den Abendstunden und dem gesamtem Wochenende geringere Lastspitzen als in der Mittagszeit an den Werktagen vorhanden. In diese Zeiten lassen sich vor allem das Waschen und Trocknen von Wäsche, sowie das Geschirrspülen verschieben. Verbraucher aus der Unterhaltungselektronik lassen sich schlechter verschieben, da sie dann eingeschaltet werden, wenn der Anwender sie auch benutzen möchte. Der Privatanwender möchte Musik hören oder sich einen Film anschauen, wenn er Lust dazu hat und nicht wenn es von der Netzauslastung am besten wäre. Daher muss bei dem automatischen Lastmanagement darauf geachtet werden, welcher Verbraucher verschoben werden soll. Dem Anwender muss es möglich sein, selbst zu entscheiden, welcher Verbrauch verschoben werden darf und welcher nicht.
Automatische Lastmanagementsysteme können technisch im Zuge der Hausautomatisierung umgesetzt werden. Dabei werden Techniken wie der Europäische Installationsbus (EIB) eingesetzt. Dabei handelt es sich um spezielle programmierbare elektrische Verbraucher. Diese werden normal an das Haustromnetz angeschlossen und benötigen zusätzlich eine Steuerleitung. Diese Verbraucher können Schalter, Lampen oder auch ein Herd sein. Sie besitzen eindeutige Adressen und können über ein Computerprogramm entsprechend programmiert werden. So kann an einem Tag der Schalter das Licht im Flur oder am nächsten Tag das Licht im ganzen Haus schalten. Durch ein

zusätzliches Lastmesssystem kann bei Spitzenlast entsprechend der Stromverbrauch im Haushalt reduziert werden. Umgesetzt sind solche Systeme im intelligenten Haus [Dor95] oder in einem Projekt der Universität Kassel mit dem Namen BEMI - Bidirektionales Energiemanagement Interface [CB06], [DN06] und [DCB05]. Hier können Verbraucher im Haushalt automatisiert geschaltet werden. Gleichzeitig sendet der Energieversorger den aktuellen Strompreis und einen Forecast, so dass ein Geräteeinsatzplan der Haushaltsgeräte erstellt werden. Dadurch kann der Verbraucher Stromkosten sparen und die Energieversorger haben weniger Lastspitzen.
Eine weitere Motivation zum Stromsparen für Privatanwender ist die genaue Aufklärung bzgl. Stromverbrauchs und Ersparnis beim Austausch von alten Geräten gegen moderne, neuere Geräte oder Techniken. So kann allein durch das Austauschen von herkömmlichen Glühlampen mit Energiesparlampen bis zu 80% Stromkosten gespart werden, da eine 20W Energiesparlampe genauso hell leuchtet wie eine 100W Glühlampe bei entsprechend geringerem Stromverbrauch. Zu dieser Aufklärung gehört es ebenfalls den Verbraucher für das Thema StandBy zu sensibilisieren. Dazu gehört auch beispielsweise eine Kaffeemaschine, die den gekochten Kaffee drei oder vier Stunden warm hält. Eine Thermoskanne hält den Kaffee ebenfalls warm, aber verbraucht dafür keinen Strom. Diese Stromsparmaßnahmen fallen nicht unter den Begriff des Lastmanagement, aber durch die Senkung des Stromverbrauches in Haushalten wird die Last im Energieversorgungsnetz gesenkt. Daher sind auch solche Maßnahmen ein Beitrag zur Senkung des Energiebedarfes und der Umweltbelastungen.
Aktuell gibt es derzeit nur für die Industrie verschiedene Stromtarife. Es existieren sowohl verschiedene Verbrauchstarife und Spitzenlasttarife. Bei diesem Spitzenlasttarif wird im 15 Minutentakt die jeweilige Spitzenlast gemessen. Übersteigt diese Last den vereinbarten Wert des gewählten Tarifes wird dieser teurer berechnet. Mit Hilfe von Lastmanagementsystemen werden Verbraucher abgeschaltet, damit teure Spitzenlasten vermieden werden. Für Privathaushalte gibt es derzeit nur Festpreise. Einer davon ist der Nachttarif. Dieser wurde eingeführt, damit sich über Nacht die Nachtspeicheröfen aufladen und diese Last nicht am Tag auftritt. Automatische Lastmanagementsysteme sind derzeit nur in Testumgebungen bei verschiedenen Universitäten verwirklicht worden. Private Haushalte haben durchaus die Möglichkeit Systeme wie EIB einzusetzen. Derzeit ist diese Technik wenig verbreitet aufgrund der Anschaffungskosten.

1.5. Eckernförder Tarif

Wie bereits erläutert, sind die Netzbetreiber daran interessiert Lastspitzen zu vermeiden. In den neunziger Jahren wurde in einem umfangreichen Feldversuch mit einem lastabhängigen Echtzeit-Tarif erstmals untersucht, ob sich Lasten im Privathaushalt messbar verschieben lassen. Dazu wurden in einer zweijährigen Planungsphase ein lastabhängiger Echtzeit-Tarif und die notwendige Hardware entwickelt. Der Feldversuch

wurde unter dem Namen Eckernförder Tarif [Pra97] bekannt. Ziel des Tarifes war es anhand der aktuellen Netzlast den derzeitigen Stromerzeugungspreis zu berechnen und an den Endverbraucher weiter zu leiten. Durch die Anzeige des derzeitigen Tarifes konnten die Teilnehmer entscheiden, ob sie bereit sind, den geltenden Preis zu bezahlen, wenn sie z. B. beginnen wollten die Wäsche zu waschen. Dadurch, dass der Tarif zwischen 11,8 Pf/kWh (netto) und 70,8 Pf/kWh (netto) schwankte, sollte der Verbraucher motiviert werden, seinen geplanten Verbrauch in eine andere Tageszeit zu verlagern.

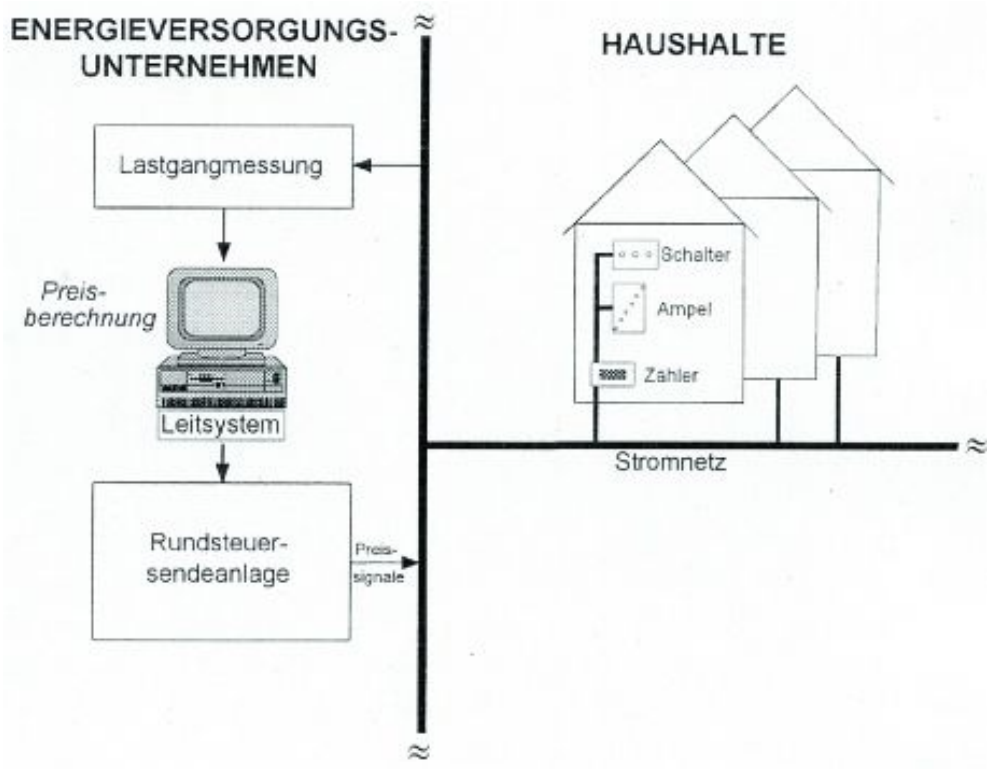

Abbildung 3: Aufbau der Komponenten im Eckernförder Tarif [Pra97]

Neben der Berechnung der aktuellen Last musste vor allem der aktuelle Preis an den Kunden übermittelt werden. Zu diesem Zweck war neue Hardware notwendig, da die normalen Stromzähler diese Aufgaben nicht übernehmen können. Dafür wurden entsprechende Zähler entwickelt. Die alten Zähler der Teilnehmer wurden mit diesen neuen Zählern ausgetauscht. Damit den Versuchsteilnehmern nicht zu gemutet werden musste, jedes Mal in den Keller gehen zu müssen, um den aktuellen Tarif zu erfahren, bekamen die Versuchsteilnehmer zusätzlich eine Stromampel und ein Lastschalter. Die verschiedenen Komponenten und der Aufbau sind in der Abbildung 3 dargestellt.
Die Stromampel bestand aus 9 Leuchtdioden (je 3 grüne, gelbe und rote LED´s). Diese LED´s signalisierten den aktuellen Preis. Grün bedeutete, dass zurzeit der Strompreis günstig ist. Sofern die roten LED leuchteten, war der Preis am höchsten. Der Lastschalter konnte durch die Versuchsteilnehmer auf einen bestimmten kWh Preis pro-

grammiert werden, sodass beim Erreichen dieses Preises der Verbraucher abgeschaltet wurde. Wenn der Preis fiel, wurde der Verbraucher wieder eingeschaltet. Für den Fall, dass die teure Strompreisphase sehr lange dauerte, konnte eine maximale Abschaltzeit eingestellt werden, damit z. B. bei einem Kühlschrank das Kühlgut nicht verdirbt. Die Stromampel und der Lastschalter konnten an jeder beliebigen Steckdose betrieben werden. Der aktuelle Strompreis wurde über eine Rundsteuersendeanlage übermittelt. Die Codierung wurde mittels der Frequenz des Signals realisiert, je kleiner die Frequenz, desto höher war der aktuelle Preis. Dieser wurde am Zähler über ein Display und über die LED´s an der Stromampel dem Testteilnehmer angezeigt.
Der Feldversuch in der Praxis wurde über zwei Jahre (August 94 – Ende 96) mit ungefähr 1000 zufällig ausgewählten Haushalten durchgeführt. Dabei bestanden die Haushalte aus Einperson- bis Vierpersonenhaushalte. Von den 1000 teilnehmenden Haushalte waren 57% Einfamilienhäuser und 43% Mehrfamilienhäuser beteiligt. Die Teilnehmer konnten über regelmäßige Befragungen ihrer Erfahrungen mitteilen. Der Rücklauf dieser Befragung lag, laut den Angaben der Projektleiter, bei ungefähr 40%. Bei den Befragungen stellte sich heraus, dass es keine Hemmnisse bei den Teilnehmern gab bzgl. einer zeitlichen Verschiebung für Waschmaschine, Trockner und Geschirrspüler. Dafür mussten die meisten Teilnehmer ihren gewohnten Tagesablauf ändern. Die Aussicht auf die Kostenersparnis war es den Teilnehmern wert.

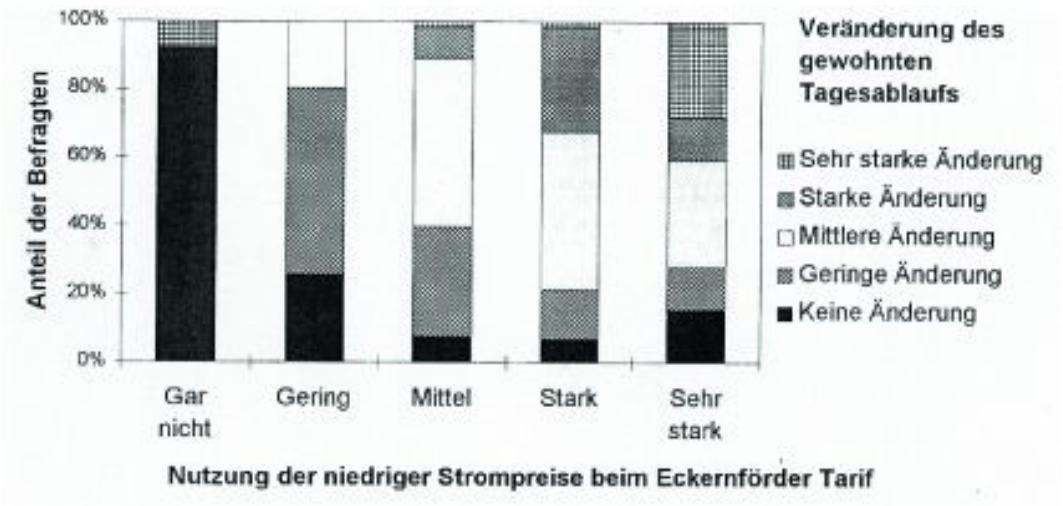

Abbildung 4: Bereitschaft den Tagesablauf zu ändern [Pra97]

Die Abbildung 4 zeigt den Zusammenhang, basierend auf den Aussagen der Untersuchungsergebnisse, zwischen der Bereitschaft den Tagesablauf zu ändern und der Ersparnis mit dem Eckernförder Tarif. Je mehr die Teilnehmer bereit waren ihren Verbrauch zu verschieben und damit den gewohnten Tagesablauf zu ändern, um so mehr konnte gespart werden. Die Ersparnisse lagen dabei zwischen 40 DM und 130 DM pro Jahr. Einige Teilnehmer hatten mögliche Mehrkosten von bis zu 50 DM, da sie sich nicht nach dem Eckernförder Tarif gerichtet hatten. Im Rahmen dieses Feldversuches muss-

ten diese Kosten nicht gezahlt werden.
Messtechnisch konnte eine Verlagerung von bis zu 100 Watt/a pro Haushalt nachgewiesen werden. Auf die Gesamtspitzenlast hatte der Feldversuch keinen messbaren Einfluss, da die Teilnehmer nur circa 4% der Gesamtlast ausmachten. Eine Verlagerung der Spitzen wurde nicht während der normalen Spitzenzeiten sondern zeitlich verschoben festgestellt. Die Befragungen ergaben, dass hauptsächlich das Wäsche waschen, trocknen und Geschirr spülen durch die Teilnehmer verlagert wurden. Dabei wurden diese Arbeiten in die preisgünstigeren Abendstunden und auf das Wochenende verschoben. Andere Verbraucher konnten nicht bzw. wurden weniger verschoben.

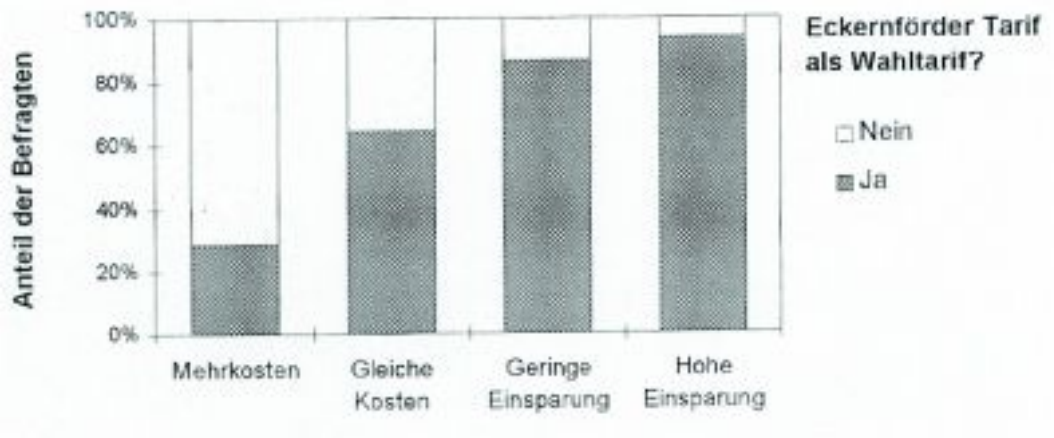

Abbildung 5: Zusammenhang Abrechnungsergebnis und Wahltarif [Pra97]

Die Befragung ergab, dass die Teilnehmer den Tarif und die verwendete Technik sehr gut fanden. Viele hatten schon nach kurzer Zeit ein Gefühl für den aktuellen Preis und mussten sich nicht mehr mit Hilfe der Stromampel orientieren. In der Abbildung 5 ist zu erkennen, dass sich trotz der möglichen Mehrkosten etwa 30% der befragten Teilnehmer (n=326 [Pra97]) für den Eckernförder Tarif entscheiden würden. 82% der Befragten gaben an, dass sie auf Grund der Teilnahme am Eckernförder Tarif zum bewussten Umgang mit Energie angeregt wurden. 78% sind davon überzeugt, dass sie durch solch einen Tarif umweltgerechter gehandelt haben. Zwei Drittel der Teilnehmer würden sich für solchen einen Tarif entscheiden, sofern ein solcher angeboten wird. Die Stromampel wurde begrüßt, während der Lastschalter im Allgemeinen abgelehnt wurde. Zu groß war die Befürchtung, dass die angeschlossenen Verbraucher schaden nehmen könnten und im Schadenfall eventuell der Versicherungsschutz erlischt. Die Teilnehmer gaben weiter an, dass sie selber über den Zeitpunkt des Ein- und Ausschalten entscheiden möchten.
Abschließend wurde von den Projektleitern mit dem Eckernförder Tarif festgestellt, dass es technisch realisierbar ist, einen lastabhängigen Echtzeit-Tarif umzusetzen. Die vorhanden Zähler müssen dazu ausgewechselt werden. Die Bereitschaft den Verbrauch zu verschieben, ist für viele Privathaushalte kein Problem, sofern ein finanzieller Anreiz

besteht.

1.6. Bidirektionales Energiemanagement Interface

Mit dem Projekt Bidirektionales Energiemanagement Interface (BEMI) der Universität Kassel wurden die technischen Grundlagen einer modernen Hausautomatisierung entwickelt. Diese entstand in Kooperation mit verschiedenen Partnern innerhalb des Projektes "Dezentrale regenerative Energieversorgungsanlagen: Technische und wirtschaftlichen Integration in den Netzbetrieb und Anpassung von Rahmenbedingungen - DINAR"[2]. Dieses Forschungsprojekt wurde 2004 von dem Kasseler Forschungsinstitut ISET und der Firma EUS GmbH aus Dortmund gestartet. Hauptziel des Projektes ist es, die Nutzung erneuerbarer Energien in bestehende Energieversorgungsanlagen zu integrieren.
Die Anzahl der verschiedenen dezentralen Energieversorgungsanlagen (DEA), wie z. B. Photovoltaik (PV)-, Windkraft- und Kraftwärmekopplungs (KWK)- Anlagen (z. B. Brennstoffzellen) nimmt immer mehr zu. Die Verwendung ist tageszeit- und wetterabhängig, daher kommt es zu einer schwankenden Energiebereitstellung. Der Anlagenbetreiber entscheidet, wie viel Strom eingespeist wird. Die Einspeisung erfolgt dabei "blind", den Energieversorgungsunternehmen fehlen die Möglichkeiten die Einspeisung der Energiemenge von herkömmlichen (Kohle- oder Atomkraftwerken) und den erneuerbaren dezentralen Energieversorgungsanlagen zu steuern und zu monitoren. Derzeit wird die Stromversorgung durch Großkraftwerke realisiert. Zukünftig wird der Anteil der Stromerzeugung durch DEA mehr zunehmen. Das Hauptproblem dabei ist die schwankende Verfügbarkeit, diese muss dann durch konventionelle Kraftwerke ausgeglichen werden. Daher ist es notwendig einen Überblick zwischen Netzlast und Netzversorgung zu haben. Die Netzversorgung unterteilt sich weiter in konventionelle Großkraftwerke und in DEA. Sowohl die Netzlast als auch die Versorgung durch DEA unterliegen immer den tagesaktuellen Situationen. Dabei handelt sich auf Seitens der Versorgung beispielsweise darum ob es windstill ist oder regnet. Die Netzlast, wie bereits beschrieben, ist ebenfalls tages- und jahreszeitabhängig. Daher muss rechtzeitig eine neue Strategie - "Optimierte Energieeinspeisung"- entwickelt werden.
Innerhalb dieses Projektes entstand das Bidirektionale Energiemanagement Interface. Da herkömmliche Zähler, wie auch bereits im Eckernförder Tarif festgestellt, nicht ausreichen, da sie nur den Stromverbrauch messen können, mussten diese zunächst ersetzt werden, damit eine Kommunikation zwischen Haushalt und Energieversorger ermöglicht werden kann. Dieser neue Zähler wird durch BEMI realisiert. BEMI fungiert als Energiemanager an der Anschlussstelle zum Niederspannungsnetz. Die Abbildung 6 zeigt wie die Kommunikation und der Aufbau erfolgt. Für die Kommunikation wird

[2] www.projektdinar.de

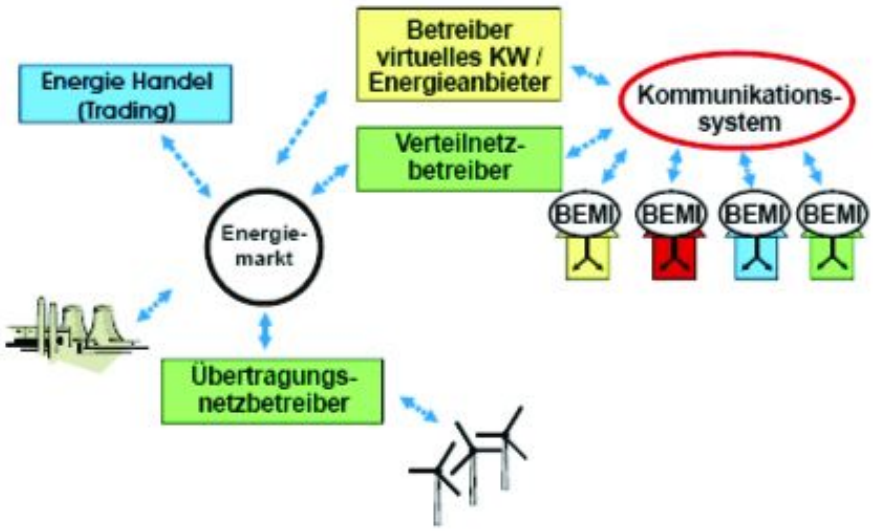

Abbildung 6: Kommunikation und Handel im liberalisierten Strommarkt mit Integration verteilter Erzeugung [CB06]

als Übertragungsweg das Internet verwendet. Als wichtigste Komponenten innerhalb BEMI sind Kommunikationsinterface, Rechner, verschiedene Bus-Schnittstellen, Sicherheitsinterface, Mehrtarifzähler und rechnergesteuerte Laststeuereinheiten implementiert worden. Mit Hilfe von BEMI wird der Energieverbrauch des Haushaltes gemanagt. Durch das integrierte Lastmanagement kann anhand der aktuellen Last der Stromverbrauch ermittelt werden. Bei zusätzlich angeschlossenen DEA kann die derzeitige Stromerzeugung mit einbezogen werden und damit wird der tatsächliche Verbrauch ermittelt. Die Leitstelle kann dies dann in die aktuelle Netzauslastung mit einbeziehen. Hauptsächlich wird über BEMI von der Leitstelle das Preisprofil für den nächsten Tag empfangen. Anhand dieser Informationen werden die angeschlossenen Geräte für den folgenden Tag geplant. Die Geräte wurden dazu in Gerätetypen eingeteilt und für jeden Gerätetyp wurde ein Managementalgorithmus entwickelt. Nach [DN06] wurden folgenden drei Gerätetypen unterschieden:

- Geräte mit einem Wärme, Kälte- oder Batteriespeicher, der in einem bestimmten Temperatur- bzw. Ladezustandsbereich gehalten werden muss
- Geräte, die nach dem Start ein festes Programm abarbeiten (z. B. Waschmaschine)
- Geräte, die bei hohem Strompreis abgeregelt werden können, aber sonst kein Management erlauben (z. B. eine dimmbare Beleuchtung)

Die Einsatzplanung der Geräte erfolgt unter der Berücksichtigung der Bedürfnisse des Nutzers. Dabei werden angeschlossene Verbraucher und Erzeuger innerhalb des Gebäudes beachtet. Die Abbildung 7 stellt einen möglichen Aufbau und die Kommunikationstruktur für Energiemanagement mit BEMI dar.

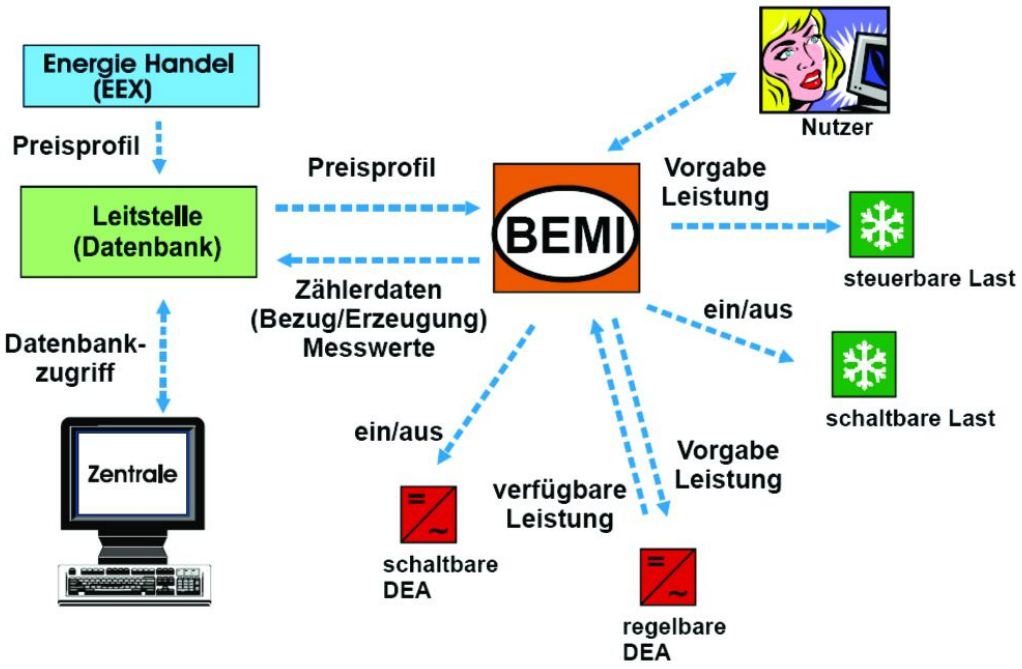

Abbildung 7: Kommunikationsstruktur innerhalb BEMI [CB06]

Der Aufbau und die Kommunikation wurde auf basierenden Standards entwickelt. So kommuniziert BEMI mit den Lasten und Erzeugern über die standardisierten Schnittstellen wie EIB und CAN. Der Aufbau erfolgte modular und lässt sich nach Bedarf weiteren Aufgaben anpassen.
Mit BEMI wurde ein effizientes Konzept entwickelt, mit dem Erzeugungs- und Lastmanagement in privaten Haushalten ermöglicht werden kann. Durch Integration vorhandener Techniken, Standards, Schnittstellen und Protokollen kann es erweitert und angepasst werden. Die Einteilung der Gerätetypen kann auch innerhalb dieser Projektgruppe mit in Betracht gezogen werden.

1.7. Resümee und Fazit

Aufgrund der schwankenden Nachfrage an Energie und der schwankenden Einspeisung der gesamten Energien müssen die Energieversorger bei Spitzenlasten i.d.R. teure und umweltschädlichere Kraftwerke zusätzlich schalten. Damit die Stromkosten moderat bleiben und die Stromerzeugung umweltschonender werden, müssen Lastspitzen im Energieversorgungsnetz vermieden werden. Mit Hilfe von Lastmanagement und geeigneten Tarifen kann auch im Privathaushalt der Stromverbrauch und damit die Lasten

im Energieversorgungsnetz verschoben werden. Durch den Einsatz von erneuerbaren Energien werden umweltbewusste Energieerzeugungsanlagen verwendet. Diese haben einen stark schwankenden Energiebeitrag, der ohne intelligente Einsatzplanung schwer zu steuern ist, mit dem Projekt BEMI wurde ein Konzept für Lastmanagement- und Erzeugungssysteme für den privaten Haushalt entwickelt.
In privaten Haushalten bedeutet Lastmanagement vor allem den Verbrauch zu verschieben und der Einsatz energieeffizienter Geräte. Der Eckernförder Feldversuch von vor 13 Jahren hat ergeben, dass private Verbraucher bereit sind, ihren gewohnten Tagesablauf zu ändern, wenn es sich finanziell für sie bezahlt macht.
Bei einem automatisierten Lastmanagement muss unterschieden werden, welcher Geräte verschoben werden können. Dabei kann innerhalb dieses Projektes die Gerätetypeneinteilung aus dem Projekt BEMI berücksichtigt und gegebenenfalls ergänzt werden. Ein weiterer wichtiger Aspekt ist, in welche Zeit die Verschiebung erfolgen soll. Diese kann nicht beliebig gewählt werden, sondern muss den Bedürfnissen des Anwenders Rechnung tragen. Der Anwender will diese Verschiebungen auf jeden Fall beeinflussen können.

Literatur

[CB06] CHRISTIAN BENDEL, MARTIN BRAUN, DAVID NESTLE JÜRGEN SCHMID PHILIPP STRAUSS: *Energiemanagement in der Niederspannungsversorgung mit dem Bidirektionalen Energiemanagement Interface (BEMI)*. Technischer Bericht, Universität Kassel, 2006.

[DCB05] DR. CHRISTIAN BENDEL, DAVID NESTLE, STEFAN MALCHER: *Dezentrale Energieeinspeisung ins Niederspannungsnetz - technische und wirtschaftliche Entwicklungslösungen* -. Technischer Bericht, Universität Kassel, 2005.

[DN06] DAVID NESTLE, DR.-ING CHRISTIAN BENDEL: *Energiemanagement in der Niederspannungsversorgung zur Integration dezentraler und fluktuierender Erzeuger*. Technischer Bericht, Universität Kassel, 2006.

[Dor95] DORTMUND, UNI: *Intelligentes Haus*. Technischer Bericht, Universität Dortmund, 1995.

[DTK07] DI THOMAS KÖNIGSTEIN, ENERGIE & BILDUNG: *Stromsparberatung*. 2007.

[Ene06] ENERGIEAGENTUR.NRW: *Prozentuale Anteile der 12 Stromverbrauchsbereiche im Haushalt*. 2006.

[Hei01] HEINDLER, FRANZ: *Energiemanagement und Stromeinkauf im liberalisierten Markt*. Diplomarbeit, Universität Graz, 2001.

[Pra97] PRAGER, W.: *Entwicklung und Test eines lastabhängigen Echtzeit-Tarifs in Eckernförde*. 1997.

2. Stromerzeugung und Strompreise

Autor: Ulrich Denecke

2.1. Einleitung

Da es das Ziel dieser Projektgruppe ist, einen Planer für den häuslichen Energieverbrauch zu erstellen, ist es interessant zu erfahren, wie elektrische Energie gewonnen wird und wie sich Strompreise zusammensetzen. In der folgenden Seminarausarbeitung wird einleitend das Prinzip der Stromerzeugung mittels eines Generators beschrieben, um anschließend unterschiedliche konventionelle Kraftwerkstypen und regenerative Energiequellen zu beleuchten. Im folgenden Abschnitt *Strompreise* wird anfangs der Unterschied zwischen Grund-, Mittel- und Spitzenkraftlastwerken und den daraus resultierenden Strompreisen erläutert. Im Anschluss daran werden unterschiedliche Tarifmodelle beschrieben.

2.2. Stromerzeugung

2.2.1. Generator

Bei den meisten Kraftwerkstypen wird zur Stromerzeugung ein Generator eingesetzt. Seine Aufgabe ist es, mechanische Energie, nämlich die von der Turbine bereitgestellte Rotationsenergie, in elektrische Energie umzusetzen. Er beruht auf dem Prinzip der elektrischen Induktion.
Wird ein elektrischer Leiter innerhalb eines Magnetfeldes bewegt, so wird in ihm eine Spannung induziert. Bei einem Generator führt ein (Elektro-)Magnet in Leiterschleifen eine Rotation aus, vergleiche Abbildung 8.
Bei dem abgebildeten Generator wird so dreiphasiger Wechselstrom mit einer Phasenverschiebung von jeweils 120° zwischen den einzelnen Phasen erzeugt.

2.2.2. Dampfturbine

Die für den Generator notwendige Rotationsenergie der Welle wird bei den meisten konventionellen Kraftwerkstypen durch eine Dampfturbine bereitgestellt. Durch sie wird die Energie des unter Druck stehenden Dampfes in Rotationsenergie umgewandelt, indem der heiße Dampf durch die Schaufeln der Turbine geleitet wird, vgl. Abbildung 9.

2.2.3. Kohlekraftwerk

Kohlekraftwerke verwenden - wie der Name schon sagt - Kohle als Energieträger. Der Aufbau eines Kohlekraftwerkes ist in Abbildung 10 dargestellt. Kohlekraftwerke kann man Grob in zwei Klassen der Befeuerung unterteilen. Die erste Klasse arbeitet mit

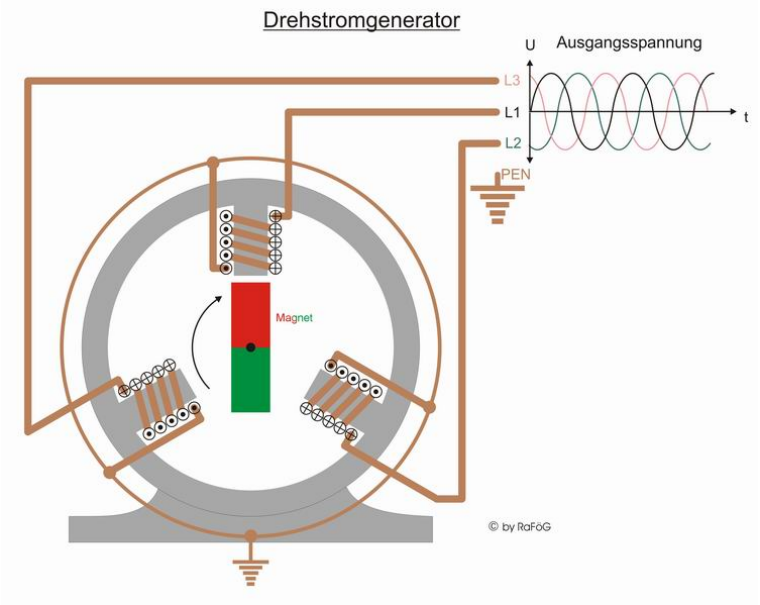

Abbildung 8: Generator zur Erzeugung von dreiphasigem Wechselstrom (Quelle: [Gen07])

Rostbeschickung. Das bedeutet, die Kohle wird über einen beweglichen Rost langsam durch die Brennkammer bewegt und dabei verbrannt. Dies bietet Vorteile, da die Kraftwerke einen relativ einfachen Aufbau haben und wartungsärmer sind. Rostbeschickung kommt vorzugsweise bei Industrie- und Heizkraftwerken verwendet. Die zweite Klasse ist die Staubbefeuerung. Hierfür wird die Kohle gemahlen und dabei getrocknet, bevor sie als Kohlenstaub-Luft-Gemisch in die Brennkammer geblasen und dort verbrannt wird. Vorteilhaft an dieser Methode der Befeuerung ist, dass das Staub-Luft-Gemisch wie ein Gas brennt und sich solche Kraftwerke daher leicht geregelt werden können. Weiterhin können sie dadurch schnell angefahren werden. Für Staubfeuerung macht es praktisch keinen Unterschied, welche Art von Kohle verfeuert wird (vgl. [Str92]).
Weiterhin können die Kraftwerke nach der Art der Kohle, mit der sie befeuert werden, aufgeteilt werden in beispielsweise Braunkohlekraftwerke und Steinkohlekraftwerke. Diese beiden Arten von Kohle zeichenen sich durch einen sehr unterschiedlichen Brennwert aus, der bei Rheinischer Braunkohle $7,9\frac{MJ}{kg}$ und bei Ruhr-Fettkohle $31,0\frac{MJ}{kg}$ beträgt.
Da Steinkohle im Abbau teurer ist als Braunkohle, werden Braunkohlekraftwerke hauptsächlich im Grundlastbereich eingesetzt, während Steinkohle vorwiegend im Mittel- und Spitzenlastbereich zum Einsatz kommt ([Str05]). Es ergeben sich Stromerzeugungskosten von etwa 2,4 Cent pro Kilowattstunde für Braun und 3,35 Cent pro Kilowattstunde für Steinkohle ([BMW07]).

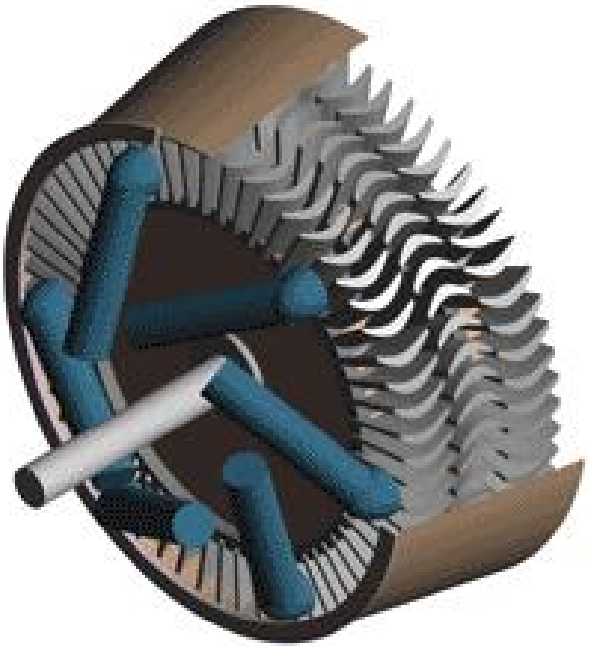

Abbildung 9: Aufbau einer Dampfturbine (Quelle: [Dam07])

2.2.4. Gasturbine

Gasturbinen bieten eine Möglichkeit, schwankenden Energiebedarf schnell auszugleichen, da sie sich schnell anfahren und abschalten lassen. Abbildung 11 zeigt den Aufbau einer Gasturbine. Die durch den Ansaugkanal angesaugte Luft wird im Verdichtetr komprimiert und in der Brennkammer mit dem Energieträger, meist Erdgas, aber auch sehr fein zerstäubtes Öl, verbrannt. Die Abgase treiben direkt die Turbine an, das heißt, es entfällt der Schritt der Dampferzeugung. Allerdings sind die Kosten pro Erzeugter Kilowattstunde mit 4,9 Cent (bei Betrieb mit Erdgas) relativ hoch (vgl. [BMW07]). Daher werden Gasturbinen laut [Str05] als Spitzenlastkraftwerke eingesetz. Der Wirkungsgrad beträgt bei Gasturbinen zwischen 38 und 56% (vgl. [Inn07]).

2.2.5. Atomkraftwerk

Kernreaktoren können in vier Klassen unterteilt werden. Dies sind Druckwasser-Reaktoren, Siedewasser-Reaktoren, Brutreaktoren und Hochtemperaturreaktoren. Einen schönen Überblick hierzu bietet [Str92]. Im folgenden Abschnitt werden die ersten drei Typen kurz erläutert.
Allen Reaktortypen ist gemein, das sie ihre Energie aus dem Zerfall von radioaktiven Isotopen beziehen. Die durch den radioaktiven Zerfall freiwerdende Energie wird zur Dampferzeugung (entweder direkt oder indirekt, siehe dazu die Reaktorbeschreibungen) verwendet. Der Dampf treibt wie beim Kohlekraftwerk eine Turbine an, die den

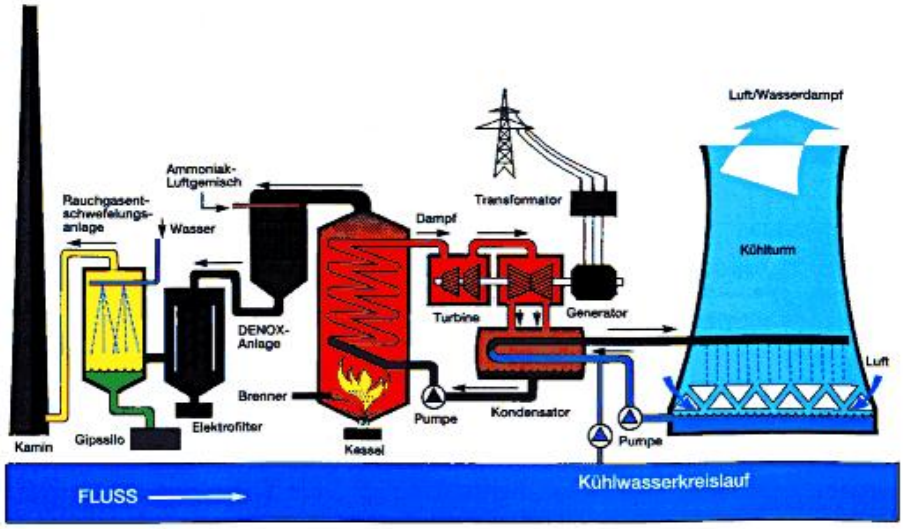

Abbildung 10: Aufbau eines Kohlekraftwerkes (Quelle: [SCZ07])

zur Stromerzeugung erforderlichen Generator antreibt. Abbildung 12 zeigt den Aufbau eines Atomkraftwerkes mit Druckwasser-Reaktor. Beim Druckwasser-Reaktor existieren zwei Stoffkreisläufe. Zum einen der Kühlmittel- bzw. Primärkreislauf und zum anderen der Arbeitsmittel- bzw. Sekundärkreislauf. Der Druck im Primärkreislauf wird so gewählt, dass das Kühlmedium nicht beginnt zu sieden, da beim Sieden der Wärmeübergang zwischen Brennstab und Kühlmedium stark eingeschränkt werden würde. Der Großteil der deutschen Atomkraftwerke ist vom Typ Druckwasser-Reaktor (vgl. [BfS07]).
Beim Siedewasser-Reaktor hingegen wird das Kühlmedium Wasser bewusst zum Sieden gebracht. Um die Wärmeabfuhr aus dem Kern sicher zu gewährleisten, muss im System ein Zwangsumlauf integriert sein. Der so erzeugte Dampf wird getrocknet und durch die Turbine geleitet.
Der dritte Typ ist der Brutreaktor. Als Spaltstoff wird mit Plutonium angereichertes Uran verwendet (80% UO_2 und 20% PuO_2). Durch die hohe Konzentration von spaltbarem Material ist eine hohe Wärmeabfuhr notwendig. Als Kühlmittel kann daher zum Beispiel flüssiges Natrium verwendet werden. Da die Errichtung von Brutreaktoren schwierig ist, sind weltweit nur wenige kommerzielle Anlagen dieses Typs im Einsatz (vgl [Str92]).

Atomkraftwerke ohne Kraft-Wärme-Kopplung haben einen Wirkungsgrad von etwa 30% und dienen als Grundlastkraftwerke (vgl. [Str05]). Laut [BMW07] decken die Kernkraftwerke in Deutschland rund 26% der allgemeinen Stromversorgung und in etwa 48% des Grundlastbedarfs. Die Kosten pro erzeugter Kilowattstunde liegen bei 2,65Cent.

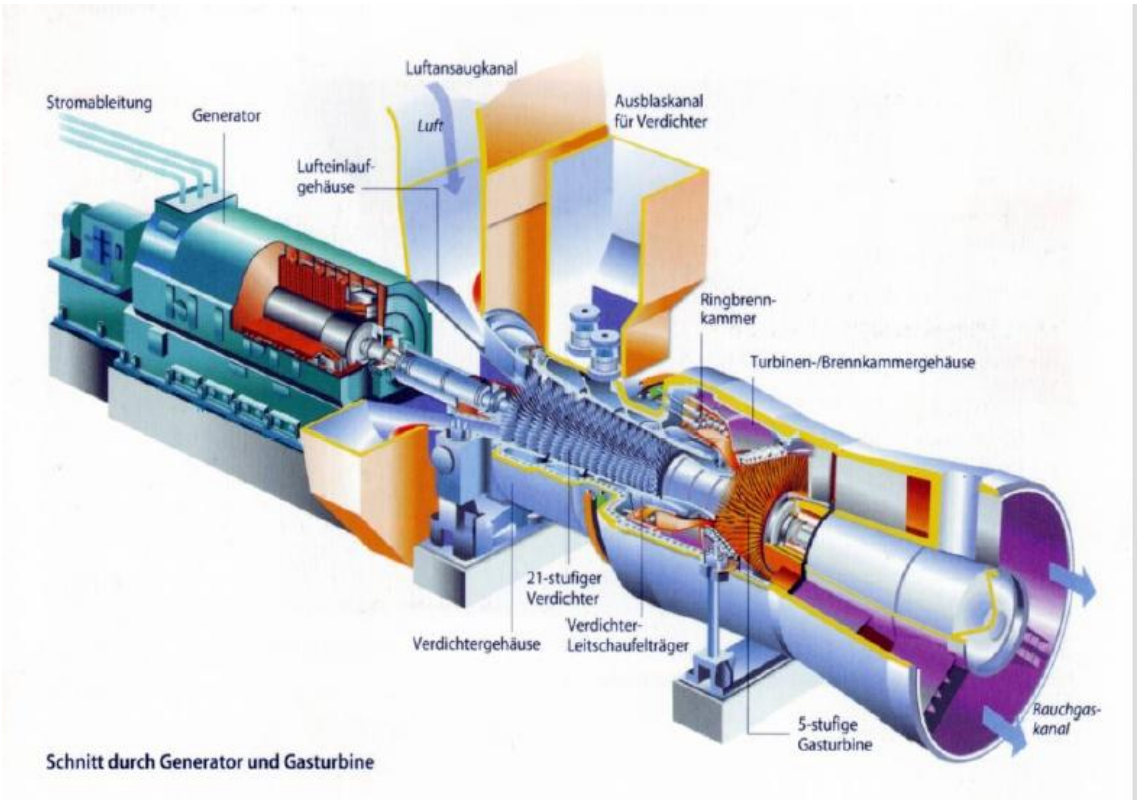

Abbildung 11: Aufbau einer Gasturbine (Quelle: [Gas07])

2.2.6. Wasserkraftwerk

Die Wasserkraftwerke können grob in drei unterschiedliche Typen eingeteilt werden, die Laufwasserkraftwerke, Speicherkraftwerke und Pumpspeicherkraftwerke. Weiterhin gibt es Wellen- und Gezeitenkraftwerke, wobei die Wellenkraftwerke bisher nur in Projekten erprobt werden. [Str92]
Laufwasserkraftwerke (dargestellt in Abbildung 13) arbeiten mit geringen Fallhöhen und großem Massedurchfluss. Für sie wird ein Wehr in einem Fluss mit ausreichend Gefälle installiert. Das Wasser fließt vom sogenannten Oberwasser durch eine Turbine ins Unterwasser und treibt diese an. Eine Durchflussregulierung bei Laufwasserkraftwerken bietet sich im Allgemeinen nicht an, da keine Möglichkeit zur Speicherung des Wassers und so der Energie existiert. Daher wäre eine Regulierung energetisch ineffizient, weil das Wasser an der Turbine vorbeigeleitet werden müsste und die Energie so verloren wäre. Aufgrund dieser Einschränkung werden Laufwasserkraftwerke zur Grundlastdeckung verwendet [Str05].
Speicherkraftwerke, deren Prinzip in Abbildung 14 arbeiten im Gegensatz zu Laufwasserkraftwerken mit großen Fallhöhen und einem geringen Massedurchfluss [Str92]. Bei ihnen fließt das Wasser durch eine Druckleitung aus einem Stausee durch die Turbine. Da bei Dauerbetrieb der Speichersee leer laufen könnte, werden Speicherkraftwerke zur

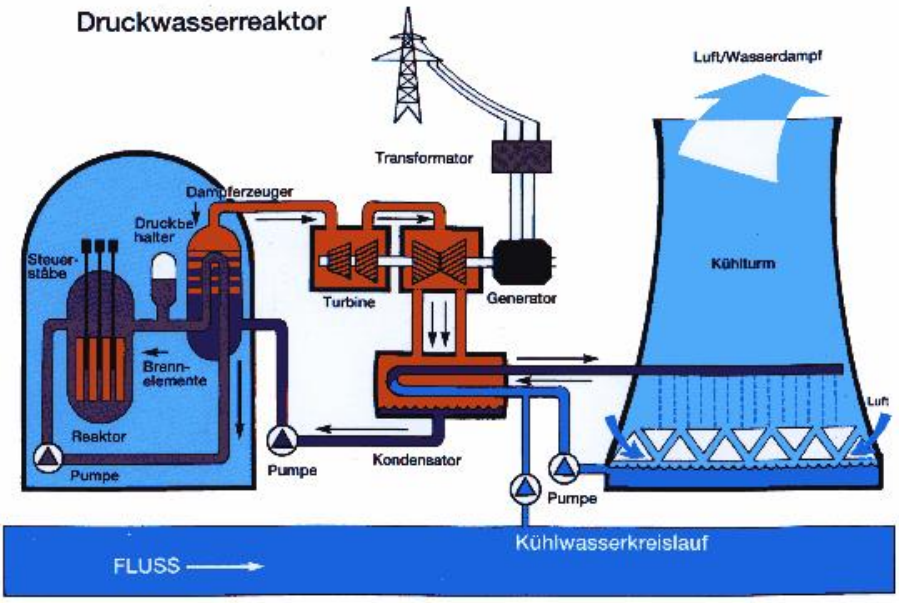

Abbildung 12: Aufbau eines Druckwasserreaktors (Quelle: [SCZ07])

Spitzenlastdeckung verwendet.
Pumpspeicherkraftwerke arbeiten wie Speicherkraftwerke, können jedoch im Gegensatz zu diesen nicht nur Strom erzeugen, sondern auch Strom speichern, indem sie mittels der Turbine, bei älteren Kraftwerken auch mittels einer Pumpe, Wasser zurück ins Speicherbecken pumpen. Daher brauchen sie nicht zwangsweise einen Zufluss zum Speicherbecken. Laut [Ada07] wird hierbei ein Wirkungsgrad von ca. 75% erreicht. Sie dienen als Spitzenlast- und Reservekraftwerke. Ihre volle Leistung erreichen Pumpspeicherkraftwerke innerhalb von 60 bis 100 Sekunden. Eine schematische Darstellung eines Punpspeicherkraftwerkes wird in Abbildung 15 gezeigt.
Gezeitenkraftwerke nutzen die Fallhöhe des Wassers zwischen mindestens einem Becken und dem Meer. Das Becken kann durch Eindämmen einer Bucht erzeugt werden. Beim ansteigender Flut wird das Wasser durch eine Turbine in das Becken gedrückt, bei einsetzender Ebbe fließt es durch die Turbine zurück ins Meer. Die bei Ebbe und Flut entstehenden Wartezeiten, in denen kein Strom erzeugt werden kann, können durch geeignete Meerbeckenanlagen verkürzt werden. Ein weiterer Vorteil der Gezeitenkraftwerke ist, dass sich die Becken als Pumpspeicher nutzen lassen. (nach [Str92]) Ein Beispiel für eine solche Anlage ist das La-Rance-Kraftwerk in Frankreich.
Ein anderer Ansatz für Gezeiten- beziehungsweise Meeresströmungskraftwerke wird beispielsweise in [Mey] beschrieben. Hierbei wird eine Säule im Meeresboden verankert, an der wie bei einer Windkraftanlage ein Rotor befestigt ist, der von den Meeresströmungen angetrieben wird. Vorteile seien hierbei, dass Meeresströmungen im Vergleich zu Wind besser vorhersagbar sind und dass anders als bei den im obigen Absatz beschriebenen Gezeitenkraftwerken keine großen Staumauern in geeigneten Buchten

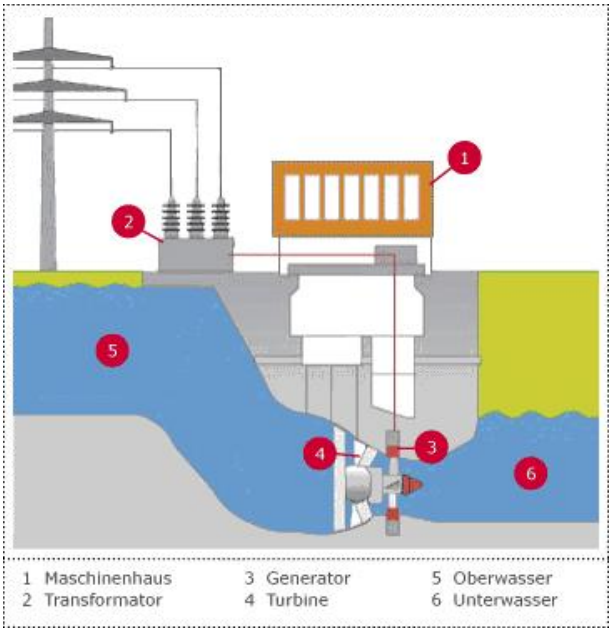

Abbildung 13: Prinzip Laufwasserkraftwerk (Quelle: [Was07])

errichtet werden müssen, was größere Einschnitte für die Umwelt bedeuten würde. Nachteilig sei jedoch die Anbindung ans Stromnetz, da hierfür Unterseekabel verlegt werden müssen. Durch geschickte Planung ließen sich jedoch solche Anlagen in die Nähe von Offshore-Windparks errichten, wodurch sich beide Anlagetypen die Anbindung ans Stromnetz teilen könnten. Als zu erwartende Kosten für die Stromerzeugung mit solchen Systemen gibt [Mey] pro Kilowattstunde 5 bis 10 Cent an.

2.2.7. Windkraft

Bei Windkraftanlagen wird die durch den Rotor strömende Luft in kinetische Energie der Welle umgesetzt, die dann den Generator antreibt. Hierbei kommt es zu einigen Problemen . Die durch Windkraftanlagen zu gewinnende Energiemenge lässt sich nicht vorhersagen. Weiterhin ist eine Speicherung des Energieträgers, wie es bei Wasserkraftwerken möglich ist, nicht realisierbar. Ein weiterer Nachteil ist, dass je nach Bodenform andere Windgeschwindigkeiten zu erwarten sind. Beispielsweise betrug die durchschnittliche Windgeschwindigkeit 1972 auf Norderney $7,4\frac{m}{s}$ und in Stuttgart-Hohenheim nur $2\frac{m}{s}$. Kleine Änderungen der Windgeschwindigkeit verursachen größere Leistungsschwankungen. Da diese unerwünscht sind, sind Windkraftanlagen so dimensioniert, dass sie ihre volle Leistung bereits bei mittleren Windgeschwindigkeiten erreichen. Bei höheren Geschwindigkeiten wird die Leistung mittels einer Regelung, die beispielsweise die Flügelanstellung verändert, konstant gehalten. Über- oder unterschreiten die Windgeschwindigkeiten die vertretbaren Grenzen, so wird die Anlage abgeschaltet. Abbildung 16 zeigt den Aufbau der Gondel einer Windkraftanlage. Zwi-

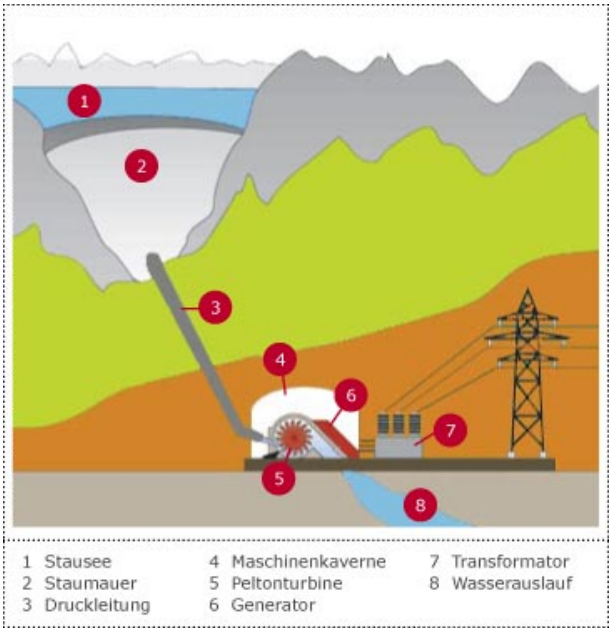

Abbildung 14: Prinzip Speicherkraftwerk (Quelle: [Was07])

schen Rotor und Generator befindet sich ein Getriebe, das die langsame Rotation der Nabe in eine schnellere für den Generator umsetzt. Die so erzeugte Spannung muss noch umgerichtet werden, um sie an die Netzfrequenz anzupassen. Hierfür wird sie erst gleichgerichtet, um anschließend mit einem Wechselrichter für das Stromnetz angepasst und synchron eingespeist wird ([Ene07]). Ein Vorteil der Windenergie ist, neben den Umweltaspekten, dass aufgrund von thermischen Gründen die Ausbeute von Windkraftanlagen zur Mittagszeit am höchsten ist. Daher lassen sich Windkraftanlagen zur Deckung der Spitzenlast nutzen ([Str05]). Das Problem hierbei ist jedoch, wie bereits oben erwähnt, das sich die Windstärke nicht vorhersagen lässt und daher konventionelle Kraftwerke in Reserve gehalten werden müssen. Der erzielte Wirkungsgrad für Windkraftanlagen liegt bei etwa 20%.

2.2.8. Photovoltaik

Photovoltaik beschreibt den Prozess, bei dem durch Lichteinstrahlung Ladungen in einem Halbleiter getrennt werden, vgl Abbildung 17. Es wird also die Energie der elektromagnetischen Welle des Lichts direkt und ohne Umwege über mechanische Komponenten in elektrische Energie umgewandelt.
Kraftwerke, die auf diesem Prinzip basieren sind laut [Str05] spitzenlastgeeignet, da sie aufgrund der erhöhten Sonneneinstrahlung zur Mittagszeit und der daraus resultierenden höheren Erträge besonders geeignet sind, um die Lastspitzen zur Mittagszeit zu kompensieren.

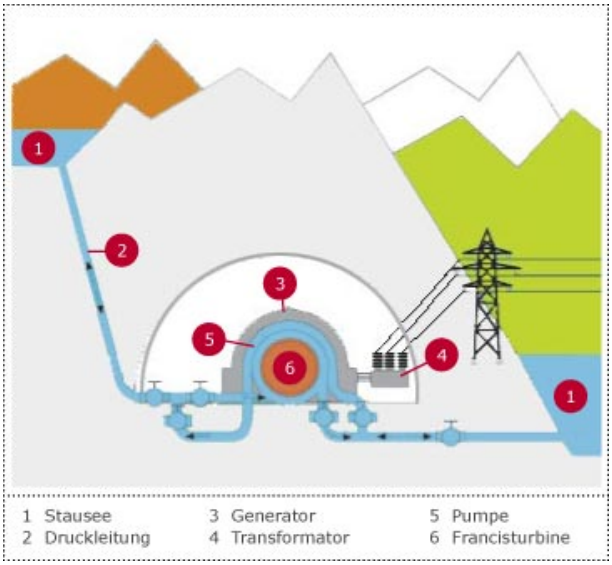

Abbildung 15: Prinzip Pumpspeicherkraftwerk (Quelle: [Was07])

2.3. Strompreise

Abbildung 18 zeigt den Stromverbrauch im Tagesverlauf. Man sieht deutlich, dass der Verbrauch über Nacht recht konstant gering ist und in den Morgenstunden drastisch ansteigt. Im weiteren Tagesverlauf kommt es zu mehreren Lastspitzen in der Mittagszeit und am frühen Abend. Um diese Spitzen ausgleichen zu können, müssen die Stromversorger sogenannte Sekunden-, Minuten- und Stundenreserven vorhalten, die entsprechend schnell Leistungsschwankungen kompensieren können. Wie bereits bei den verschiedenen Kraftwerkstypen aufgezeigt, ergibt sich durch die unterschiedlichen Kosten pro erzeugter Kilowattstunde eine Aufteilung, in welchen Bereichen ein Kraftwerk wirtschaftlich Arbeiten kann. Bei den Spitzenlastkraftwerken ist es zumeist so, dass die Investitionskosten für das Kraftwerk recht gering sind, die Betriebskosten hingegen recht hoch. Daher sind diese Kraftwerke nur bis zu einer gewissen Laufzeit rentabel. Hingegen ist es bei den Grundlastkraftwerken so, dass die Investitionskosten recht hoch sind, dafür jedoch die Betriebskosten recht niedrig. Solch ein Kraftwerk arbeitet am wirtschaftlichsten, wenn es eine möglichst lange Zeit unterbrechungsfrei arbeitet. Bei den regenerativen Energien kommt noch hinzu, dass die Energie nicht immer verfügbar ist beziehungsweise sich die erzeugbare Energiemenge nicht vorhersagen lässt. Zwischen Grund- und Spitzenlast ist die Mittellast eingeordnet.
Hieraus ergibt sich, dass der Strom in der Erzeugung nicht konstant viel kostet, sondern diese Kosten im Tagesverlauf schwanken. Wird gerade zu einer Lastspitze viel Strom benötigt, müssen viele der teuren Spitzenlastkraftwerke angefahren werden, wodurch die Stromerzeugungskosten steigen. Abbildung 19 zeigt den zeitlichen Verlauf der Strompreise an der EEX, der European Energy Exchange, über den Tag. Es ist zu sehen, dass auch die Kosten für Stromeinkäufe zu den Lastspitzen Spitzenpreise errei-

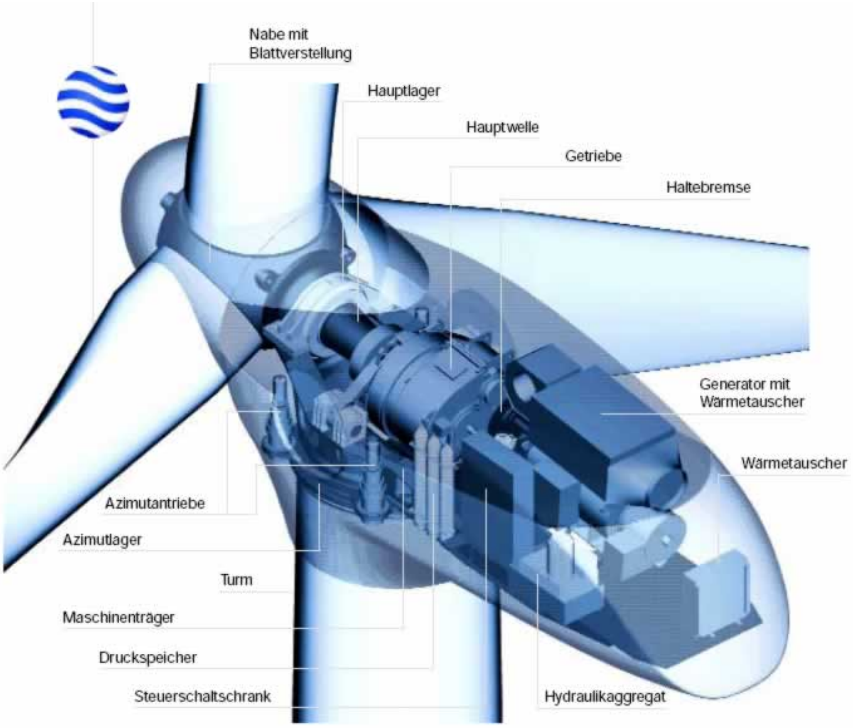

Abbildung 16: Gondel einer Windkraftanlage(Quelle: [Win07])

chen. Diese Kosten könnten theoretisch an den Verbraucher weitergegeben werden, um ihn zu animieren, die Stromkosten zu senken. Im folgenden Abschnitt werden einige Modelle für Stromtarife vorgestellt.

2.3.1. Sonderverträge

Sonderverträge kommen für Großkunden in Frage, deren Jahresverbrauch über 100.000 kWh liegt oder die eine Anschlussleistung von mehr als 30kW benötigen. Hierbei wird ein individueller Vertrag zwischen Verbraucher und Versorger geschlossen. Dieser Vertrag ist an die besonderen Anforderungen des Kunden angepasst. Die Vertragsdetails werden in der Regel vertraulich behandelt, da sie sich von Kunde zu Kunde sehr stark unterscheiden können. (Quelle: [Str05])

2.3.2. Tarifkunden

Bei den Tarifkunden setzt sich die Gebühr aus zwei Teilen zusammen. Dies ist zum einen ein Grundpreis, der die Zählermiete darstellt und zum anderen der Arbeitspreis für den eigentlichen Verbrauch. Hierbei gibt es nun unterschiedlichste Tarife. Als Beispiele für nicht zeitabhängige Tarife dienen die Tarife der EWE AG. Sie sind zu finden unter [EWE07].

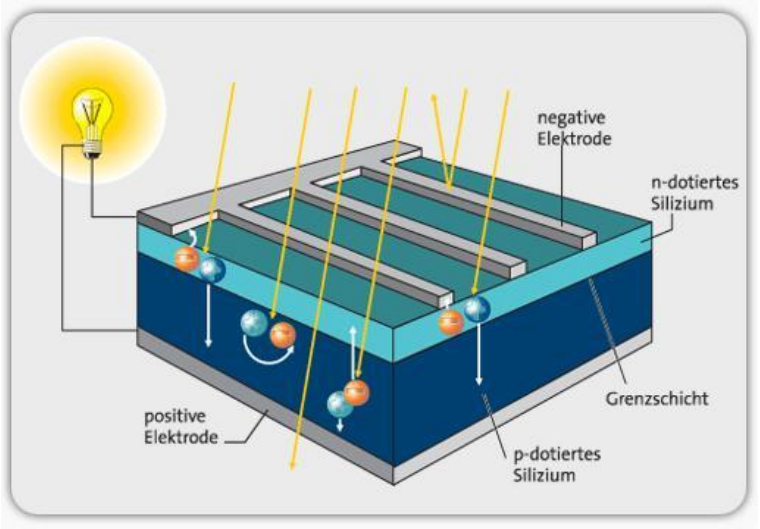

Abbildung 17: Aufbau einer Solarzelle (Quelle: [Sol07])

Der einfachste Tarif ist der *EWE Strom classic*. Bei ihm beträgt der Grundpreis 55,00EUR pro Jahr. Der Arbeitspreis beträgt unabhängig von der abgenommenen Arbeit 17,73 Cent pro Kilowattstunde. Für Kunden mit einem Verbrauch von mehr als 10.000 Kilowattstunden pro Jahr gibt es als günstigeren Tarif *EWE Strom classic XL*. Bei kleineren Abnahmemengen sind die Kosten wie bei *EWE Strom classic*. Ab 10.000 Kilowattstunden pro Jahr verteuert sich der Grundpreis auf 60,80EUR pro Jahr, dafür sinkt der Arbeitspreis auf 17,60 Cent pro Kilowattstunde. Der Strom wird also günstiger, je mehr man abnimmt.
Weiterhin gibt es Tarife, die die Erzeugungsart der elektrischen Energie festlegen. Dies sind zum Beispiel die Tarife *EWE NaturWatt Strom* und *EWE NaturWatt Strom plus*. *EWE NaturWatt Strom* gibt an, dass die Energie zu 100% aus regenerativen Energiequellen gewonnen wird. Der Grundpreis beträgt 79,00EUR pro Jahr. Der Arbeitspreis ist mit 17,73 Cent pro verbrauchter Kilowattstunde nicht teurer als im Tarif *EWE Strom classic*. *EWE NaturWatt Strom* gibt zusätzlich noch die Zusammensetzung des Stroms an (50% Wasserkraft, 49% Windenergie und 1% Sonnenenergie). Der Grundpreis beträgt ebenfalls 79,00EUR pro Jahr, der Arbeitspreis hingegen 20,60 Cent pro Kilowattstunde. Anhand dieser Tarife wird deutlich, dass es nicht immer deutlich teurer sein muss, Strom aus regenerativen Energiequellen zu bevorzugen. Weiterhin ist zu sehen, dass die hier vorgestellten Tarife nicht auf die Uhrzeit des Verbrauchs eingehen und so von den durchschnittlichen Erzeugungskosten abhängen.
Ein anderer interessanter Ansatz ist es, den Strompreis für eine gewisse Zeit festzulegen und eine Lieferung zu diesem Preis zu garantieren. Der entsprechende Tarif der EWE AG ist *EWE Strom classic fix*. Hierbei musste der Vertrag bis zum 31.03.2007 abgeschlossen werden. Die EWE AG garantiert dann bis zum Ende des Jahres 2008 bei einem Grundpreis von 60,80 EUR pro Jahr einen Arbeitspreis von 18,53 Cent pro Ki-

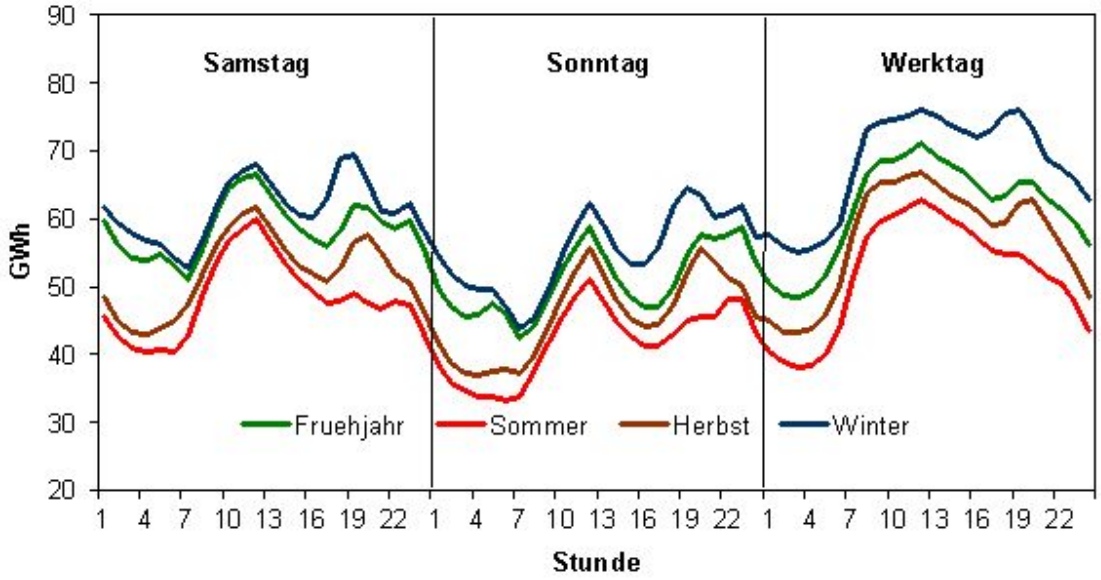

Abbildung 18: Lastschwankungen (Quelle: [Las07])

lowattstunde. So ein Ansatz bietet ein gewisses Risiko für beide Vertragspartner, wird durch irgendein Ereignis oder eine Gesetzesänderung der Strom billiger, so zahlt der Kunde mehr als notwendig. Wird der Strom jedoch teurer, was der wahrscheinlichere Fall sein dürfte, so muss der Anbieter gegebenenfalls den Strom unter Wert verkaufen.
In den USA, wo es immer wieder, besonders in den Sommermonaten wenn viele Klimanalagen laufen, zu erheblichen Energiemagel kommt, ist es interessant, den Kunden zu motivieren, seinen Verbrauch in Zeiten zu legen, in denen die Stromerzeugung günstiger ist und es nicht zu Ausfällen kommt. Als Beispiele hierfür wurden einige Privatkundentarife der Firma Georgia Power (vgl. [GPo07]) gewählt. Der erste hier vorgestellte Tarif ist der TIME OF USE - RESIDENTIAL ENERGY ONLY SCHEDULE: T̈OU-REO-3.̈ Bei ihm werden die Preise in festen Timeslots angegeben. So beträgt der Arbeitspreis bei diesem Tarif bei einem Grundpreis von 9,75 $ pro Monat während der Sommermonate Juni bis September On-Peak 17,086 und Off-Peak 5,278 Cent pro Kilowattstunde. On-Peak bedeutet hierbei zu den Zeiten, zu denen eine höhere Last erwartet wird. Dies ist in diesem Fall von 14:00 bis 19:00 von Montag bis Freitag in den oben genannten Monaten mit Ausnahme von einigen Feiertagen. Off-Peak beschreibt die übrige Zeit. Für die Zeit von Oktober bis Mai gilt ein vereinfachtes System bei dem die Kilowattstunde bis zu einem monatlichen Verbrauch von 650 kWh 5,278 Cent kostet, bei einem höheren Verbrauch nur noch 2,954 Cent.
Der Tarif REAL-TIME PRICING DAY-AHEAD WITH ADJUSTABLE CBL SCHEDULE: „RTP-DAA-2“ geht noch genauer auf die tatsächlichen Erzeugungskosten ein. Bei ihm ändern sich die Strompreise stündlich. Eine Preisübersicht für den Tag wird am Vortag bekannt gegeben. Die dort angegebenen Preise werden durch Lastprognosen abge-

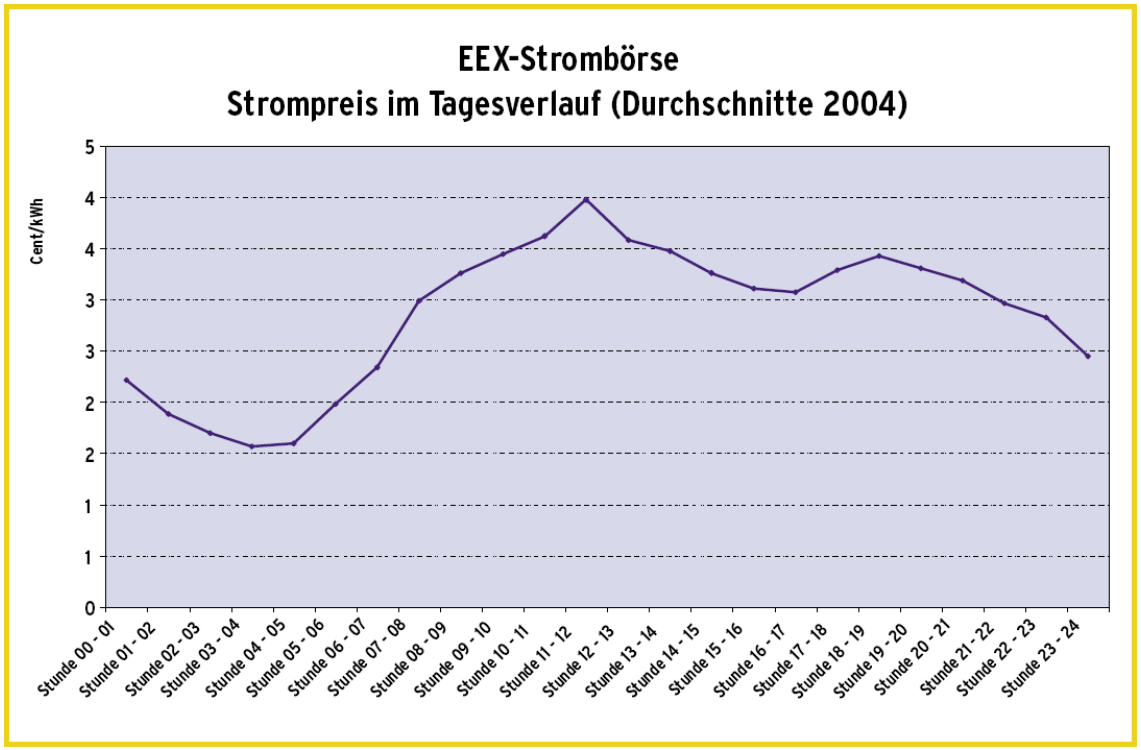

Abbildung 19: Strompreis an der EEX im Tagesverlauf (Quelle: [Str05])

schätzt.

2.4. Zusammenfassung und Fazit

Im ersten Abschnitt *Stromerzeugung* wurde kurz auf die verschiedenen Arten der konventionellen Stromerzeugung und auf die Stromerzeugung mittels regenerativer Energiequellen eingegangen. Hierbei wurden die ungefähren Kosten der Stromerzeugung aufgezeigt und darauf eingegangen, welchem Lastbereich ein Kraftwerkstyp angehört. Im folgenden Abschnitt *Strompreise* wurden unterschiedliche Tarifmodelle vorgestellt wie Sondertarife, Tarife mit festen Arbeitspreisen, sinkende Arbeitspreise ab einer bestimmten Abnahmemenge, für längere Zeit vertraglich festgelegte Arbeitspreise und zwei unterschiedliche Modelle die zeitlich über den Tag veränderbare Strompreise anbieten.

Anhand der Kraftwerksbeschreibungen ist zu sehen, dass sich beispielsweise Laufwasser- und Gezeitenkraftwerke gut für die Grundlastdeckung verwenden lassen, da bei Laufwasserkraftwerken eine Speicherung der Energie nicht wirtschaftlich ist und sich Gezeitenkraftwerke gut einplanen lassen und zuverlässig zu bestimmten Zeiten Energie liefern. Weiterhin wurde deutlich gemacht, dass sich Solar und Windenergie zur Spitzenlastdeckung eignen, da sie sowieso in der Mittagszeit, in der es zu einer Lastspitze

kommt, die besten Erträge liefern. Allerdings müssen, da Erträge aus Solar- und Windenergie nicht präzise vorhersagbar sind, ausreichend Reserve in Form von schnell anfahrbaren konventionellen Kraftwerken beziehungsweise (Pump-)Speicherkraftwerken zur Verfügung stehen.

Durch die unterschiedlichen Erzeugungskosten in den verschiedenen Lastbereichen scheint es sinnvoll zu sein, zeitlich variable Stromtarife einzuführen, wie dies schon beispielsweise Georgia Power in den USA eingerichtet hat. Der Verbraucher könnte um Kosten zu sparen seine stromverbrauchenden Aktivitäten in Zeiten geringerer Last und so auch geringerer Stromkosten legen. Ist das System der Preismitteilung schnell genug, das man nicht im Voraus berechnete, sondern aktuelle Strompreise bekommt, so könnte sehr flexibel auf regenerative Energien eingegangen werden. Die Erzeuger wären also in der Lage, Preisvorteile durch Windenergie an windigen Tagen direkt an den Verbraucher weiterzugeben und ihn so zu animieren, regenerative Energien verstärkt zu nutzen.

Literatur

[Ada07] ADAM, MARIO: *Stromerzeugung mit Wasserkraft - Übersicht*, 05 2007. http://tww.fh-duesseldorf.de/DOCS/FB/MUV/adam/downloads/skript_fb4/regen/520Wasserkraft.pdf.

[BfS07] BFS: *Kernkraftwerke in der Bundesrepublik Deutschland*, 05 2007. http://www.bfs.de/kerntechnik/ereignisse/standorte/karte_kw.html.

[BMW07] BMWI: *Kernenergie in Deutschland*, 05 2007. http://www.bmwi.de/BMWi/Navigation/Energie/kernenergie,did=156016,render=renderPrint.html.

[Dam07] 05 2007. http://library.thinkquest.org/C006011/german/sites/dampfturbine.php3?v=2.

[Ene07] *Enercon Netzanbindungssystem*, 05 2007. http://www.enercon.de/www/de/na_netzanbindungssystem.nsf/mainView/1?OpenDocumen.

[EWE07] *EWE Tarifübersicht Privatkunden*, 05 2007. http://www.ewe.de/ewe_6197.php.

[Gas07] 05 2007. http://www.hagelstein-consult.de/images/gasturbine.jpg.

[Gen07] 05 2007. http://www.bswiki.de/mediawiki/index.php/Drehstromgenerator:_Aufbau_und_Funktion.

[GPo07] *Georgia Power pricing information*, 05 2007. http://www.georgiapower.com/pricing/gpc_resrates.asp.

[Inn07] *Höherer Wirkungsgrad durch Siemens-Gasturbine*, 05 2007. http://www.innovations-report.de/html/berichte/energie_elektrotechnik/bericht-4946.html.

[Las07] 05 2007. http://www.ewi.uni-koeln.de/content/forschung_und_beratung/ewi_modelle/gems/anforderungen/index_ger.html.

[Mey] MEYER, FRANZ: *Seaflow - Strom aus Meeresströmungen*. Technischer Bericht, BINE Informationsdienst.

[SCZ07] 05 2007. http://www.stadtwerke.clausthal.harz.de.

[Sol07] 05 2007. http://www.solar-is-future.de/uploads/pics/0205_Prinzip_0852_01.gif.

[Str92] STRAUSS, KARL: *Kraftwerkstechnik zur Nutzung fossiler, regenerativer und nuklearer Energiequellen*. Springer-Verlag, 1992.

[Str05] *Was Strom aus erneuerbaren Energien wirklich kostet*, 2005.

[Was07] 05 2007. http://solaratlas.htwk-leipzig.de.

[Win07] 05 2007. http://www.windpark.at/de/c_technik/technik_010.htm.

3. Klassifikation von Haushaltsgeräten

Autor: Malte Hurrelmann

3.1. Einleitung

Im Rahmen der Projektgruppe "Demonstrator InHouse Energiemanagement" an der Uni Oldenburg, welche im SS 2007 und WS 2007/2008 stattfindet, soll eine Software entwickelt werden, welche dem "Privatmann" die Möglichkeit geben soll, seinen individuellen Stromverbrauch transparenter zu machen. Hierzu sollen (möglichst) alle Stromverbraucher in seinem Haushalt aufgeführt und die jeweils einstehenden Stromverbräuche und damit auch Stromkosten dargestellt werden, sowie Einsparpotential durch z.B. Erneuerung eines Gerätes durch ein sparsameres aufgezeigt werden. Des weiteren soll die Software die Möglichkeit bieten, mögliche (zukünftige) variable Stromtarifmodelle mit einzubeziehen und die Benutzung der Haushaltsgeräte damit besser planbar zu machen. Im Vorfeld der Softwareentwicklung muss zunächst geklärt werden, mit welchen Geräten sich eine Kostenersparnis bei Benutzung variabler Stromtarife erreichen lässt.

Stromtarifmodelle mit variablen Stromkosten basieren auf der Grundlage, dass ein Stromerzeuger versucht den Stromverbrauch seiner Abnehmer und damit die von ihm zu erzeugende elektrische Energie möglichst vorhersehbar und gleichmäßig zu gestalten. Hierbei ist es denkbar, dem Abnehmer über einen niedrigen Preis den Anreiz dazu zu geben, die elektrische Energie nicht in den Spitzenlastzeiten zu verbrauchen, sondern die Benutzung der entsprechenden Geräte auf eine andere Tageszeit zu verschieben.
Betrachtet man hierbei einen typischen Tageslastgang der Bundesrepublik Deutschland und insbesondere der Privathaushalte, so ist leicht erkennbar, dass Lastspitzen und -täler durchaus am durchschnittlichen Tagesablauf der Bevölkerung auszumachen sind. Lastspitzen sind um die Mittagszeit sowie in den frühen Abendstunden zu verzeichnen und werden insbesondere durch die Essenszubereitung, Fernsehgeräte und die Beleuchtung hervorgerufen.
Weiterhin ist besonders in der Zeit von 22Uhr abends bis 6Uhr morgens ein Lasttal zu beobachten, welches sich dadurch erklären lässt, dass der Großteil der Bevölkerung schläft und somit außer den üblichen Geräten mit Stand-By Modus sowie Kühlgeräte alle anderen Geräte i.d.R. ausgeschaltet sind. Dieses Lasttal ist insbesondere der Zielzeitraum für die Benutzung von geeigneten Geräten dar.

3.2. Exkurs: Nachtspeicherheizungen

3.2.1. Einleitung

Ab den 1950er und den 1960er Jahren wurde seitens der Energiebetreiber bereits im großen Stil versucht, eine Einebnung der Lastkurve, bzw. eine Lastverschiebung hin-

ein in die Lasttäler zu erreichen (vergleiche hierzu [Bre05]), indem Speicherheizungen gekoppelt mit einem extra "Nachttarif´´ propagiert wurden. Dieser Nachttarif war sehr viel günstiger als der Tagtarif. Dieser Nachttarif wurde dann dazu genutzt, um das vorhandene Lasttal in der Nacht künstlich mit der Erzeugung von Wärme zu füllen. Somit wurden diese Speicherheizungen auch als Nachtspeicherheizungen oder Niedertarifheizungen bekannt. Hierbei wird mit Hilfe elektrischer Widerstände Wärme erzeugt, diese gespeichert und dann gleichmäßig wieder abgegeben. Dieser Vorgang der Ladung geschieht i.d.R. in einem vorgegebenen Zeitraum, der oftmals auch als Ladefreigabe bezeichnet wird, z.B. von 22Uhr bis 6Uhr. In den Wintermonaten ist oftmals noch ein weiterer, jedoch kürzerer Ladezeitraum z.B. von 14Uhr bis 16Uhr vorgesehen.

3.2.2. Technische Funktionsweise Nachtspeicherheizung

Die generelle technische Funktionsweise einer Nachtspeicherheizung soll im folgenden in Anlehnung an [Sta05] näher erklärt werden.

Abbildung 20: Nachtspeicherheizung

Ein Kern aus einem wärmespeichernden Material wird zu meist in der Nacht durch entsprechend gewählte elektrische Widerstände auf eine Temperatur von bis zu 600°C aufgeheizt, um die so gespeicherte Wärmeenergie mehr oder minder gleichmäßig über den Tag wieder abzugeben. Der erwähnte Kern besteht hierbei meist aus mehreren Einheiten von der Größe eines handelsüblichen Backsteins welche in der Regel aus Eisenoxid gegossen werden. Dieses Material wurde gewählt, da es im Gegensatz zu anderen untersuchten potentiellen Wärmespeichern (z.B. Magnesit, Wasser) relativ günstig in der Herstellung ist und/oder sehr viel bessere Wärmespeichereigenschaften besitzt. Da der Kern im Vergleich zur Umgebung eine recht hohe Temperatur erreichen kann ist eine gute Wärmeisolation (Dämmstoff) von Nöten um die Brand- und Verletzungsgefahr zu minimieren, sowie die erreichte Wärme so lange wie möglich zu speichern und kontrolliert an die Umgebung abgeben zu können. Verwendete Materialien sind hierbei z.B. Steinwolle und Keramikfaser.

Zusätzlich zur passiven Entladung der Speicherelemente ist es bei den meisten handelsüblichen Speicherheizungen möglich, die Entladung durch die Verwendung von eingebauten Lüftern zu beschleunigen und somit die Umgebungstemperatur zu erhöhen. Die Regelung dieses Lüfters kann zum einen manuell, sowie bei neueren Modellen in der Regel auch per Thermostat passieren, welcher eine vom Nutzer definierte "Soll"-Temperatur mit der "Ist"-Temperatur vergleicht und dementsprechend den Lüfter steuert.

3.2.3. Steuerungsarten

Bis 1996 ist die geschätzte Anzahl installierter Nachtspeicherheizungen auf 2,5 Millionen angestiegen. Da seit dieser Zeit die Vergünstigungen beim Vertrieb von Nachtstrom kontinuierlich gesunken sind, ist zu erwarten das Neuinstallationen von Nachtspeicherheizungen generell abgenommen haben. Somit ist ferner anzunehmen, dass sich die Anzahl verbauter Geräte seit 1996 nicht nennenswert verändert haben wird. Laut [Sta05] beträgt die Summe der installierten Leistung aller Geräte in etwa 40GW, welche bei gleichzeitigem Ladebeginn eine weitere ungewollte Lastspitze in der Nacht erzeugen würde, was genau entgegengesetzt der Ausgangsidee der Einebnung der Lastkurve wäre. Somit wurden Steuerungstechniken entwickelt, welche die Last möglichst gleichmäßig auf die gesamte Ladefreigabe verteilen. Sie heißen Vorwärts-, Rückwärts- und Spreizsteuerung. Die Vorwärtssteuerung veranlasst die Speicherheizungen gleich zu Beginn der Ladefreigabe den Speicherkern mit maximaler Leistung zu laden, bis die Zieltemperatur erreicht ist. Danach wird dieser Temperatur mit geringerer Leistung gehalten.

Somit ergibt sich in der Summe folgende Leistungskurve:

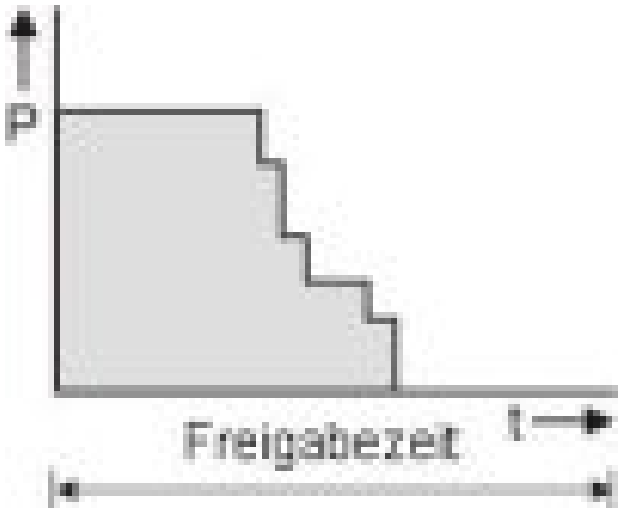

Abbildung 21: Vorwärtssteuerung

Analog zur Vorwärtssteuerung wird bei der Rückwätssteuerung die Maximallast erst kurz vor Ende der Ladefreigabe erreicht:

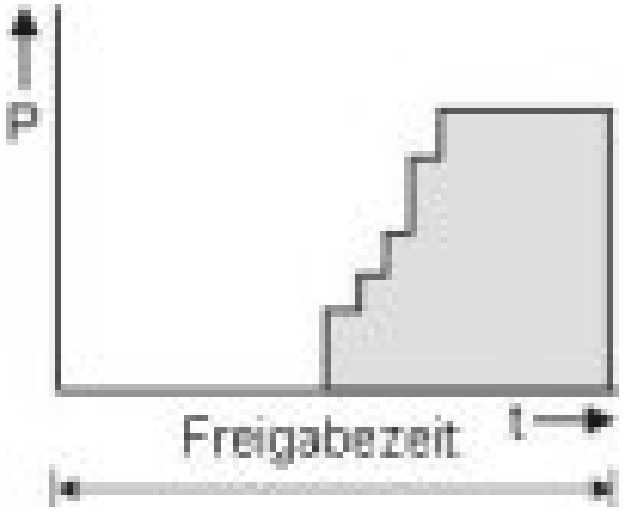

Abbildung 22: Rückwärtssteuerung

In der Spreizsteuerung wird die Maximallast etwa in der Mitte der Ladefreigabe erreicht, danach wird wie bei der Vorwärtssteuerung die Zieltemperatur bis zum Ende der Freigabedauer gehalten:

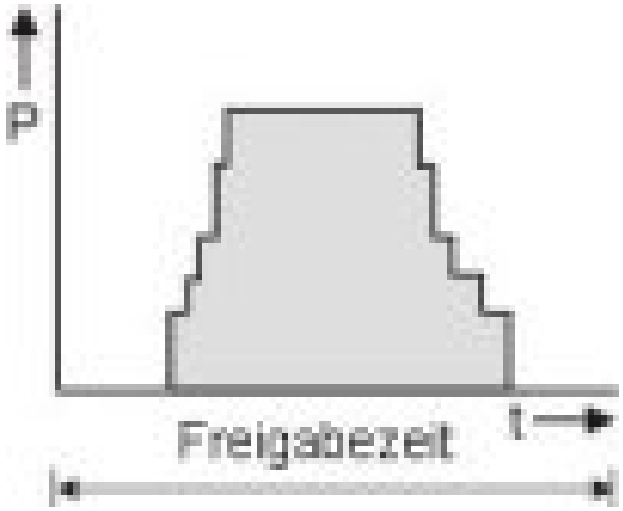

Abbildung 23: Spreizsteuerung

Mittels Rundsteuertechnik werden etwa je ein drittel der Speicherheizungen mit einer dieser Steuerungstechnik geladen, wodurch in der Summe eine Leistungsverteilung erreicht wird, die über den Freigabezeitraum in etwa gleich ist.

3.3. Klassifikationen

3.3.1. Klassifikation: Groß-/Kleinelektrogeräte

Bevor eine Klassifikation gewählt werden kann, ist es sinnvoll, vorhandene Klassifikationen von elektrischen Haushaltsgeräten näher zu betrachten.
Die einfachste Klassifikation wird durch die reinen Abmessungen der Energieverbraucher vorgenommen. Somit wird in Elektrogroßgeräte und Elektrokleingeräte unterteilt. Die Klasse "Elektrogroßgeräte" beinhaltet also große Geräte wie Kühlschrank, Herd, Spülmaschine, usw. Die Klasse "Elektrokleingeräte" beinhaltet somit kleine Geräte wie Kaffeemaschine, Wasserkocher, DVDPlayer, usw. Obwohl die Leistungsaufnahme der Elektrogroßgeräte i.d.R. höher ist als jene von Elektrokleingeräten wird sie in der Klassifikation nicht betrachtet.

3.3.2. Klassifikation: "weisse","braune", "graue" und "rote" Ware

Die eine weitere einfache Klassifikation geschieht anhand der Farbe der Geräte. Historisch bedingt hatten Geräte zur Erfüllung eines Aufgabenbereichs die gleiche Farbe. So waren Elektrische Geräte die zur Erfüllung des Haushalts dienen, wie z.B. Herd, Waschmaschine, Spülmaschine usw. Weiß und wurden somit als "weisse Ware" bezeichnet. Unterhaltungselektronik, wie der Fernseher und das Radio besaßen für gewöhnlich einen Korpus aus Holz und wurden somit als "braune Ware" bezeichnet. Da Holz als Verkleidungsmaterial von Unterhaltungselektronik immer mehr in den Hintergrund gedrängt wurde, kam die "graue Ware" hinzu. Diese Klasse beinhaltet z.B. DVDPlayer, Computer, usw. Die Klasse der "grauen Ware" hat somit die "braune Ware" abgelöst. Weiterhin wurden Heizgeräte, wie z.B. elektrische Heizungen unabhängig von der eigentlichen Farbe des Gerätes als "rote Ware" bezeichnet um den Charakter von Wärme wiederzugeben. Beide vorangegangenen Klassifikationen berücksichtigen die Verbrauchs- oder Lasteigenschaften der Geräte ihrer Klassen kaum oder gar nicht. Aus diesem Aspekt sind sie für die Verwendung in der Projektgruppe kaum geeignet.

3.3.3. Klassifikation: Energieeffizienzklassen

Seit dem 01.01.1997 ist des weiteren das Energieverbrauchskennzeichnungsgesetz in Kraft getreten, welches einen Händler für Haushaltselektrogeräte dazu verpflichtet, die von ihm verkauften Geräte mit einem EU-Energielabel zu versehen um den vom Hersteller angegebenen Verbrauch für den Kunden abschätzbar und somit verschiedene Geräte miteinander vergleichbar zu machen. Im Zuge des Inkrafttretens wurden für viele Gerätetypen durch die Deutsche Energie-Agentur ([Deu98]) eine Effizienzklassifikationen entwickelt. Die Vorgehensweise hierbei soll am Beispiel des Gerätetyps "Kühl- und Gefriergeräte", welche seit dem 01.01.1998 gekennzeichnet werden muss erläutert werden. Kühl- und Gefriergeräte wurden in Effizienzklassen von "A" bis "G" eingeteilt, wobei "A" die Geräte enthält, welche am effizientesten mit elektrischer Energie haushalten. Die Klassen wurden wie folgt definiert: Zunächst wurde ein Referenzzeitraum

im Jahre 1992 gewählt und daraufhin ein durchschnittlicher Verbrauch für damalige auf dem Markt erhältliche Kühlschranke, Gefrierschränke, Gefriertruhen sowie Kühl-/Gefrierkombinationen bestimmt. Nun wurden Faktoren festgelegt, welche das Verhältnis der Energieeffizienzklassen zu diesen ermittelten durchschnittlichen Verbräuchen darstellen. So wurde der Klasse "A" der Faktor 0.55 zugeordnet, was bedeutet, dass ein Kühlschrank der ab dem 01.01.1998 gekauft worden ist und somit mit dem Energie-Label ausgezeichnet worden sein muss, 55% oder weniger der elektrischen Energie verbraucht als ein vom Volumen und der Kühlleistung gleichwertiger Kühlschrank aus dem Jahre 1992.

Abbildung 24: Energielabel für Kühl-/Gefrierkombination (Quelle: [Deu98])

Da aufgrund immer während fortschreitender Entwicklung die Herstellung von immer sparsameren Geräten möglich ist, welche logischerweise immer in die Energieeffizienzklasse "A" einzuordnen wären, hat man sich dazu entschlossen weitere Klassen einzuführen und diese "A+" sowie "A++" benannt.

Die Klassifikation nach Energieeffizienskklassen berücksichtigt den Verbrauch der elektrischen Geräte und ist somit durchaus in die Projektgruppen Arbeit mit einzubeziehen.

3.3.4. Klassifikation nach [DHA05]

Neben den oben benannten Klassifikationen soll an dieser Stelle insbesondere eine Klassifikation betrachtet werden, die in sofern geeigneter für die Benutzung in der Projektgruppe erscheint, als dass sich mit ihrer Hilfe entscheiden lässt inwiefern die betrachteten Geräte an einem Lastmanagement teilnehmen können. Eine solche Klassifikation wird in Anlehnung an [DHA05] getätigt und soll im Folgenden Vorgestellt werden.
Zunächst wird der Verbrauch an elektrischer Energie in eine Ober- und eine Unterklasse gegliedert. Hierbei unterscheiden zwei Oberklassen "regelbar" und "nicht regelbar" erst einmal darüber, ob ein Haushaltsgerät überhaupt an einem Lastmanagement teilnehmen kann, oder nicht. Die Oberklasse "regelbar" wird ferner in die drei Unterklassen "speicherbar", "zeitlich verschiebbar" und "abschaltbar/substituierbar" unterteilt. Anhand dieser Gliederung lässt sich nun festmachen, in welcher Art das jeweilige Haushaltsgerät am Lastmanagement teilnehmen kann.

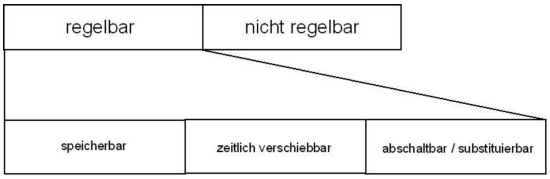

Abbildung 25: Klassifikation (Quelle: [DHA05])

Oberklassen:
regelbar: die Klasse "regelbar" enthält alle die Geräte, welche durch geeignete Maßnahmen an einem Lastmanagement teilnehmen können, sie enthält die Klassen "speicherbar","zeitlich verschiebbar" und "abschaltbar/substiuierbar".
nicht regelbar: In der Klasse "nicht regelbar" sind alle Geräte vorhanden, welche sich generell nicht in ein Lastenmanagementprogramm einpflegen lassen. Meist ist dies der Fall, da z.B. ein eine Verschiebung der Benutzung des Gerätes einen inakzeptablen Komfortverlust mit sich bringen würde. In die Klasse "nicht regelbar" fallen insbesondere Geräte zur Essenszubereitung und Beleuchtung, da diese immer dann elektrische Energie verbrauchen müssen, wenn sie auch benutzt werden. So wird z.B. ein Herd oder ein Mikrowellengerät meistens kurz vor dem Essen benutzt. Und da z.B. das Mittagessen in der Regel in einem festen Zeitraum geschieht, ist hier energietechnisch gesehen

eine Anhebung der Gesamtlast in der Mittagszeit nicht aus dem Wege zu gehen. Gleichermaßen verhält es sich mit Geräten zur Erzeugung von Licht. Sie werden immer dann benutzt, wenn das Tageslicht alleine nicht mehr ausreicht, also in den Abendstunden.

Unterklassen:
"speicherbar": in die Klasse "speicherbar" fallen alle Anwendungen, bei denen sich elektrische Energie in eine andere, speicherbare Energieform umwandeln lässt. Dieser Prozess der Umwandlung sollte nach Möglichkeit an jedem Zeitpunkt des Tages passieren können. Zwar sind Umwandlungen elektrischer in kinetische Energie (z.B. Schwungrad) oder direkte kurzzeitige Speicherung elektrischer Energie (z.B. Batterien, Kondensatoren) denkbar, jedoch angesichts des Anwendungszwecks (Haushalt) nicht praktikabel. Somit fallen die Klasse "speicherbar" besonders thermische Anwendungen. Dazu zählen die eingangs schon erwähnten Speicherheizungen aber auch Gefriergeräte. Bei letzteren ist es denkbar, das Kühlgut vor eintreten einer Lastspitze und somit eines relativ hohen Strompreises weiter als gewohnt herunterzukühlen. Durch diesen Vorgang würde das Kühlgut keinen nennbaren Schaden nehmen, jedoch wäre die Zeit, die es ohne weitere Kühlung gelagert werden könnte bis es eine kritische obere Temperatur erreicht hat, lang genug um die Zeitspanne mit dem relativ hohen Strompreis überbrücken zu können.

"zeitlich verschiebbar": als "zeitlich verschiebbar" würde ein Stromverbrauch dann gelten, wenn eine Leistung ohne mögliche Speicherung erzeugt werden muss, jedoch der Zeitpunkt des Stromverbrauchs frei wählbar ist. In diese Klasse sind insbesondere die oben genannten Elektrogroßgeräte oder "weisse Ware" einzuordnen. So ist es z.B. denkbar den Geschirrspüler mitten in der Nacht einzuschalten, nämlich dann, wenn der Strompreis ein Minimum erreicht hat. Gleiches gilt generell auch für Waschmaschinen, allerdings ist hierbei zu beachten, dass die gewaschene Wäsche nach Beendigung des Waschprogramms i.d.R eine gewisse Nacharbeit benötigt, da sie eine Restfeuchte besitzt und somit aus der Wäschetrommel entnommen und aufgehängt bzw. mit einem Wäschetrockner weiterbearbeitet werden sollte. Somit ist zum einen eine Reihenfolge definiert (Waschmaschine vor Wäschetrockner), zum anderen darf das Waschmaschinenprogramm nicht in den Abendstunden beginnen, da dann die Wäsche nach Beendigung recht lange in der Trommel verbleiben würde. Generell wäre auch eine Unterbrechung des Waschprogramms möglich. Betrachtet man ein übliches 30°C Waschprogramm einer Waschmaschine, so fällt zunächst erstmal auf, dass etwa die Hälfte der gesamten elektrischen Energie auf die Heizung des Waschwassers entfällt. Entsprechend höher ist dieser Anteil bei höher temperierten Waschgängen. Somit ist es nicht ratsam, das Waschprogramm an einer beliebigen Stelle zu unterbrechen, da dann das Waschwasser abkühlen würde und somit erneut Energie aufgebracht werden müsste, wenn das Programm fortgesetzt wird. So ist eine Unterbrechung erst dann möglich, sobald die reine thermische Anwendung beendet und das Laugenwasser abgepumpt ist. Somit gibt es im Prinzip nur einen möglichen Zeitpunkt das Waschprogramm zu unterbre-

chen, nämlich kurz vor dem Schleudervorgang. Viele Waschmaschinenprogramme tätigen an dieser besagten Stelle schon eine Unterbrechung des Waschgangs, somit müsste man hier nur einen automatischen Programmwechsel einfügen. Ein weitere Anwendungsbeispiel für zeitlich verschiebbaren Stromverbrauch ist ferner die Beheizung von Aquarien sowie die Belüftung von Gartenteichen durch Umwälzpumpen, denn beide Geräte könnten mit einer Verzögerung anlaufen, ohne Flora oder Fauna des betrachteten Wasserbassins nachhaltig negativ zu beeinflussen.

"abschaltbar/substituierbar": zuguterletzt beinhaltet die Klasse "abschaltbar/substituierbar" alle Geräte, die im Prinzip nur aus angeeigneter Bequemlichkeit immerwährend Strom verbrauchen, im Sinne des Lastmanagements jedoch einen erheblichen Teil der Grundlast abtragen könnten. Dieses gilt für alle Stand-By Geräte wie z.B. Videorecorder, Fernsehgeräte, HiFi-Komponenten. Hierbei wäre es denkbar, dass an Stelle vieler einzelner Stand-By Schaltungen, eine einzige Stand-By Steckdosenleiste für jeweils alle HiFi-Komponenten, eine weitere für alle Unterhaltungselektronik (Fernseher, Videorecorder, Spielekonsole, usw.), sowie eine für die Heimbüroelektronik (Computer, Monitor, Drucker, usw.) eingesetzt würde. Eine solche Steckdosenleiste könnte dann mit Hilfe einer weiteren, oder einer schon vorhandenen Fernbedienung alle an sie angeschlossenen Geräte auf Knopfdruck mit Strom versorgen und trotzdem nur selbst im Stand-By Modus sein. Somit ginge kaum Komfort verloren. Weiterhin gehören in die Klasse des substituierbaren Stromverbrauchs Warmwasserspeicher welche oftmals unter Küchenspülen oder in Gäste WCs montiert sind und hier für die Warmwassererzeugung sorgen. Diese Geräte sind so konzipiert, dass sie eine bestimmte Menge Wasser (z.B. 5Liter) immer auf einer vorher definierten Temperatur halten. Nun ist leicht einzusehen, dass gerade in den erwähnten Gäste WCs diese Vorgehensweise sowohl ökologisch als auch aus der Sicht des Lastenmanagements nicht tragbar ist, sobald das auf Temperatur gehaltene Wasser längere Zeit nicht benutzt wird. Um den Komfortverlust zu minimieren ist z.B. der Ersatz dieser Geräte durch Durchlauferhitzer denkbar. Sie verfügen über eine sehr viel höhere Leistungsaufnahme und erhitzen so das angeforderte Wasser erst dann, wenn es auch wirklich benutzt wird. Weiterhin seien hier auch so genannte Schein-Aus Geräte erwähnt, welche, obwohl sie keinerlei Stand-By Funktionalität bieten und nicht eingeschaltet sind trotzdem Strom verbrauchen. Hierzu gehören neben Computertechnik und vielen Netzteilen z.B. auch viele Toaster. Dieser unnötige Stromverbrauch kann nur durch das Trennen des Gerätes von der Stromleitung also Ziehen des Netzsteckers aus der Steckdose, oder durch eine Steckdose mit Kippschalter erreicht werden.
Auch in die Klasse der abschaltbaren Energieverbräuche zählen Beleuchtungen, welche nur zur

3.4. Fazit

Nach Betrachtung dieser vier Klassifikationen wird deutlich, dass sich eine Klassifikation nach Größe und Farbe von Geräten wenig mit dem Verbrauch und somit einem

Einsparpotential für die Benutzung eines Gerätes bietet. Die Klassifikation nach Energieeffizienzklassen ermöglicht einen ersten Anhaltspunkt zum Vergleich von gleichartigen Elektrogeräten und ist somit sowohl für den Endverbraucher, als auch für die Projektgruppenarbeit als wichtig einzuschätzen. Insbesondere die Klassifikation nach [DHA05] bietet die Möglichkeit, bei der Benutzung und nicht-Benutzung von elektrischen Haushaltsgeräten Einsparpotentiale zu erkennen und einen möglichen Komfortverlust gegen den gegenüberstehenden Einspareffekt abwägen zu können.

Literatur

[Bre05] BREMER, JÖRG: *Agentenbasierte Simulation des Planungsverhaltens adaptiver Verbraucher in Stromversorgungssystemen mit Real-Time-Pricing*. Diplomarbeit, 2005.

[Deu98] DEUTSCHE ENERGIE-ARGENTUR GMBH (DENA): *EU Energielabel*, 1998. Internet: http://www.eu-label.de/ (21.05.2007).

[DHA05] DR. HANS AUER, DR. CLAUS HUBER, DR. MICHAEL STADLER DI CARLO OBERSTEINER: *Endbericht: Modellierung von Kraftwerksbetrieb und Regelenergiebedarf bei verstärkter Einspeisung von Windenergie in verschiedene Energiesysteme unter Berücksichtigung des Lastmanagements*. Institut für Elektrische Anlagen und Energiewirtschaft, 2005.

[Sta05] STADLER, INGO: *Demand Response; Nichtelektrische Speicher für Elektrizitätsversorgungssysteme mit hohem Anteil erneuerbarer Energien*. Verlag im Internet; disertation.de, 2005.

4. Software-Qualitätsmanagement

Autor: Michael Erlmann

4.1. Einleitung

Moderne Softwareprojekte nehmen inzwischen bis zu ihrer Fertigstellung lange Zeiträume und sehr viele Ressourcen in Anspruch. Daher ist es notwendig, sicher zu stellen, dass ein Projekt auch fertig gestellt wird, da nicht selten das Überleben der ganzen Firma an diesen Projekten hängt. Häufig scheiterten diese Projekte aber daran, dass das entwickelte Produkt zu viele Fehler enthält, oder Fehler, welche in einer frühen Entwicklungsphase aufgetreten sind, das Projekt so weit beeinflussen, dass eine Fertigstellung innerhalb des zeitlichen oder budgettechnischen Rahmens nicht mehr möglich ist. Um diesem entgegen zu wirken gibt es im Software Engineering den Teilbereich des Software-Qualitätsmanagements, um methodisch Fehler zu Korrigieren und zu Vermeiden.

4.2. Was ist Software-Qualitätsmanagement?

Um sich mit dem Begriff des Qualitätsmanagement auseinander setzen zu können benötigt man eine genaue Definition davon, was Qualität im Bezug auf Software bedeutet. Dazu kann die Beschreibung aus der DIN ISO 9126 herangezogen werden:

”Software-Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Software-Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.”[91291]

Das Software-Qualitätsmanagement befasst sich nun damit, ob die entwickelte Software diesem Qualitätsbegriff gerecht wird. Hierzu gliedert es sich in drei Unterpunkte: Qualitätsmanagement, Qualitätsplanung und Qualitätssicherung (nach [Wir03]).

Qualitätsmanagement befasst sich mit den managementbezogenen Aktivitäten der Qualitätssicherung. Sie stellt einen Katalog von Tätigkeiten und Werkzeugen zum Erreichen der Qualität bereit.

Qualitätsplanung legt die Vorgehensweise fest und bestimmt Ziele und Verantwortungen. Auch die notwendigen Maßnahmen werden in der Planung festgelegt.

Qualitätssicherung verwendet die vom Qualitätsmanagement angebotenen Tätigkeiten und Werkzeuge, um die im Qualitätsplan festgelegten Ziele zu erreichen.

4.2.1. Grundlegende Ansätze im Qualitätsmanagement

Im Qualitätsmanagement gibt es zwei grundlegende Ansätze zur Sicherstellung der Qualität. Das sind zum einen Maßnahmen für die konstruktive Qualitätssicherung und zum anderen für analytische Qualitätssicherung (vergleiche [Wir03]).

Konstruktive Qualitätssicherungs-Maßnahmen unterstützen den Entwickler bei der Umsetzung. Hierzu werden ihm Vorgehensweisen und Werkzeuge bereitgestellt. Dabei wird nicht nur darauf geachtet, dass das entwickelte Produkt eine hohe Übereinstimmung mit den Anforderungen erhält, sondern auch, dass der Erstellungsprozess bestimmte Eigenschaften besitzt. Beispiele hierfür sind ganze Entwicklungsumgebungen wie z.B. Eclipse oder aber auch schon das Einrücken von Programmtext im Editor. Auf diese Weise versucht die konstruktive Qualitätssicherung Fehler von Anfang an zu vermeiden.

Analytische Qualitätssicherungs-Maßnahmen befassen sich dagegen mit dem direkten Vergleich erarbeiteter Ergebnisse mit den geforderten Funktionalitäten. Dabei erhält man ein Wert für die Qualität des untersuchten Produktes. Das Produkt muss jedoch bereits entsprechend weit fertig gestellt worden sein, um diesen Wert ermitteln zu können. Daher wird analytische Qualitätssicherung eher gegen Ende der Entwicklung ausgeführt. Probleme hierbei sind, dass Fehler, welche nicht früh erkannt worden sind, sich nun um so stärker auswirken können, und eine Behebung erheblichen Aufwand bedeuten kann. Dies führt weiter dazu, dass möglicherweise nicht genügend Zeit eingeplant worden ist, um einen solchen Aufwand betreiben zu können. Daher kann wie zum Beispiel im V-Modell angedacht das Ergebnis einer Phase analytisch gegen seine jeweilige Spezifikation abgeglichen werden.

4.2.2. Vorgehensmethoden in der Qualitätssicherung

Es gibt fünf grundsätzliche Vorgehensmethoden in der Qualitätssicherung. Dabei stellen diese nicht einzeln stehende Verfahren dar sondern befassen sich mit Teilaspekten der Entwicklung. Sie sollten somit bei der Entwicklung kombiniert verwendet werden (nach [Fäh02]).

Quantitative Qualitätssicherung geht davon aus, dass ein Vergleich von Anforderungen und Ergebnissen nur dann erfolgen kann, wenn quantifizierbare Soll- und Istwerte vorliegen. Somit wäre eine Bedingung wie 'Das System muss performant sein' nicht überprüfbar. Statt dessen braucht man in der Anforderung bereits zum Beispiel eine Bedingung 'Das System darf zum Verarbeiten von Vorgang X maximal Y Sekunden brauchen', was im Programm messbar ist. Jedoch kann es eine Schwierigkeit sein entsprechende Metriken zur Quantifizierung von Soll- und Istwerten zu erhalten, da viele Anforderungen einen subjektiven Hintergrund haben können. Ebenso kann sich

ein Programm auf unterschiedlichen Rechnern jeweils unterschiedlich verhalten. Metriken und ihre Rollen werden in Abschnitt 4.5 genauer betrachtet.

Maximale Konstruktive Qualitätssicherung hat als Ziel, die analytischen Maßnahmen so gut wie möglich durch konstruktive Maßnahmen zu entlasten. Durch vorausblickende konstruktive Maßnahmen sollen Fehler vermieden werden, so dass diese gar nicht erst bedacht werden müssen, wenn die analytischen Maßnahmen durchgeführt werden. So überprüft zum Beispiel Eclipse bei der Zuweisung von Variablen automatisch den Typ der Variable während des Implementierens. Somit kann auf eine komplette Typprüfung während der Laufzeit verzichtet werden, da Typfehler nicht auftreten können.

Frühe Fehlererkennung und -behebung geht davon aus, dass Fehler, welche früh in der Entwicklung stattfinden diejenigen sind, die am teuersten sind. Je länger ein Fehler unerkannt bleibt, desto schwerer wird es ihn nachträglich zu beseitigen. Daher soll besonders in den frühen Phasen der Entwicklung ein besonderes Augenmerk auf das Auftreten und Verhindern von Fehlern gerichtet sein. Es heißt in [Fäh02]: ”55% aller Fehler entstehen in der Anforderungs- und Entwurfsphase”. Insgesamt sollen Fehler zum frühstmöglichen Zeitpunkt erkannt und behoben werden, damit diese nicht weitere Folgefehler in späteren Phasen erzeugen können. Hier spielen besonders die konstruktiven Qualitätssicherungsmaßnahmen eine große Rolle.

Unabhängige Qualitätssicherung trennt die Entwicklung von der Qualitätssicherung. Es wird davon ausgegangen, dass der Entwickler nicht gleichzeitig konstruktiv sowie destruktiv denken kann. Demnach übersieht der Entwickler mögliche Fehlerquellen in seinem Produkt. Daher sollte der Entwickler nicht die analytische Qualitätssicherung seines eigenen Entwurfs oder Codes durchführen. Auch der neutrale Blickpunkt einer Person die nicht mit dem Code verbunden ist, ist von Vorteil.

Entwicklungsbegleitende Qualitätssicherung sieht die Qualitätssicherung als Teil der Entwicklung an und bettet sie dazu fest in die Entwicklung ein. Hierzu wird das V-Modell verwendet (Siehe Abbildung 26).
Im V-Modell wird die Zeit auf der horizontalen Achse angenommen und die Schritte werden den schwarzen Pfeilen nach ausgeführt. Im linken Arm sind dabei entsprechend dem erweiterten Wasserfallmodell Rückschritte zum Prüfen und Ausbessern der vorherigen Phasen angedacht. Der rechte Arm beschreibt die expliziten Testphasen. Diese werden in Abschnitt 4.3.1 genauer betrachtet. Die beiden Phasen, welche im V-Modell die gleiche Tiefe haben, bearbeiten die gleichen Dokumente. Im linken Teilarm werden sie erstellt und im rechten Teilarm getestet. Zu beachten ist, dass in jeder Phase im linken Teilarm bereits für die dazugehörigen Testphasen Anwendungsszenarien und Testfälle generiert werden müssen.

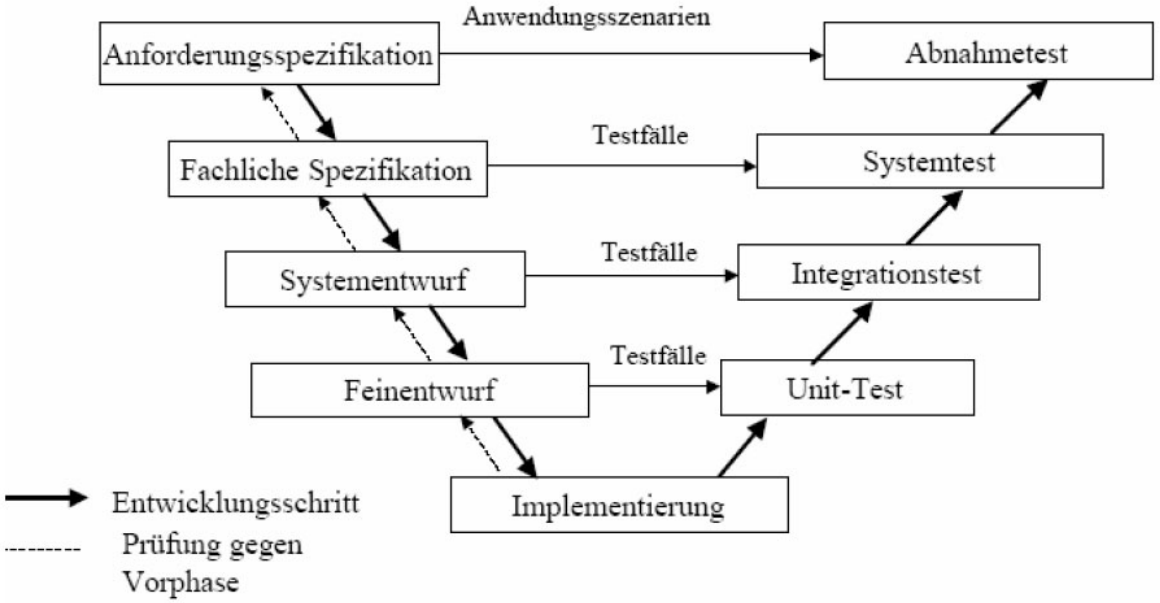

Abbildung 26: Das V-Modell. Quelle: [Wir03]

4.2.3. Validierung und Verifikation

Abbildung 27: Validierung und Verifikation im Kontext der Software-Qualitätssicherung

Die Qualitätssicherung muss sich mit zwei Fragen beschäftigen: 'Wird das richtige System gebaut?' (Validierung) und 'Wird das System richtig gebaut?' (Verifikation). Die jeweiligen Positionen werden in Abbildung 27 dargestellt. Bei der Validierung muss betrachtet werden, ob die Anforderungen, welche in der Anforderungsdefinition aufgenommen wurden, auch jene sind, welche der Kunde vom Produkt wünscht. Die Verifikation beschäftigt sich mit dem Überprüfen, ob das Produkt die spezifizierten Anforderungen umsetzt. Es sollen jedoch nicht nur die in der Anforderungsdefinition festgelegten Eigenschaften überprüft werden, sondern auch weitere nichtfunktionale Anforderungen bedacht werden, welche die Qualität des Produktes beeinflussen, wie Effizienz, Erweiterbarkeit, oder Robustheit. Dabei wird die Verifikation hier nicht als formaler

mathematischer Beweis angestrebt, sondern eine des zu verifizierenden Elementes entsprechende nichtformale Methode angewendet ([Has04]).

Methoden zur Validierung

Ein häufiges Hindernis bei der Validierung sind Kommunikationsschwierigkeiten zwischen Kunde und Entwickler. Erst, wenn beide Seiten sich sicher sind, das gleiche Produkt zu meinen, kann mit der Entwicklung begonnen werden. Für den Abgleich der Vorstellungen von Entwicklern und Kunden ist es sinnvoll, **Explorative Prototypen** bereit zu stellen, welche dem Kunden zeigen, was der Entwickler unter seinen Ausführungen verstanden hat. Diese Prototypen müssen so lange angepasst werden, bis der Kunde mit der Funktionalität einverstanden ist.

Methoden zur Verifikation

Verifikation in der betrachteten Art und Weise erreicht der Entwickler entweder experimentell durch Testen, oder analytisch, was durch Reviews und Inspektionen geschieht. Dabei gliedert sich der experimentelle Teil in strukturorientierte Verfahren (Whitebox) und funktionsorientierte Verfahren (Blackbox).

Whitebox

Die Whitebox-Verfahren gehen davon aus, dass der Code eines untersuchten Objektes einsehbar ist. Der Quelltext der zu testenden Methode wird genommen und auf einen gerichteten Graphen, einen sogenannten Kontrollflussgraphen, abgebildet (Siehe Abbildung 28). Dabei entsprechen die Knoten des Graphen genau einer Anweisung und die Kanten entsprechen dem Pfad durch den Code. Bei einer Schleife oder einer Bedingung gehen demnach mehrere Kanten aus einem Knoten. Der Test des Programmcodes findet dann entweder kontrollflussorientiert oder datenflussorientiert statt.
Beim **kontrollflussorientierten Test** ist ein Testfall gleich einem einzelnen Pfad durch den Graphen. Dabei können dann unterschiedliche Überdeckungsmaße zur Auswahl der Testfälle bestimmt werden (nach [Sch04]):

1. Anweisungsüberdeckung: Jede Anweisung muss durch mindestens einen Testfall abgedeckt sein. Das heißt es müssen so lange neue Testfälle entwickelt werden, bis jeder Knoten im Kontrollflussgraphen (28) mindestens einmal erreicht wurde. Anweisungsüberdeckung wird oft als Maß zur Testüberdeckung verwendet, findet (laut [Sch04]) jedoch nur 18% der Fehler.

2. Zweigüberdeckung: Jede Kante muss durch mindestens einen Testfall abgedeckt sein. Hierbei reicht es, wenn Schleifen nur einmal durchlaufen werden, was bei manchen Programmen jedoch keine ausrechende Testzahl sein kann, da hier weitere Bedingungen von der jeweiligen Kante abhängen können. Zweigüberdeckung ist gut geeignet logische Fehler aufzudecken, jedoch findet es selten Datenfehler.

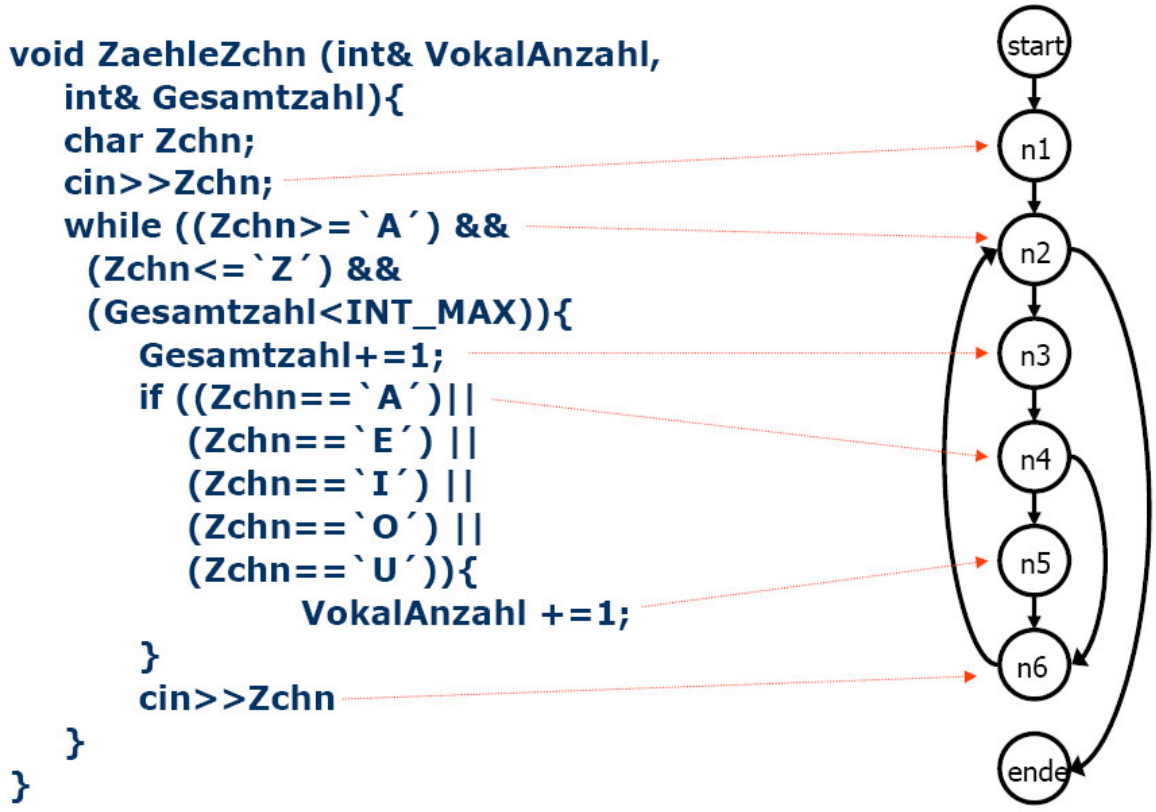

Abbildung 28: Abbildung eines Programmes auf einen Kontrollflussgraphen. Quelle: [Sch04]

3. Bedingungsüberdeckung: Bei jeder Verzweigung werden alle möglichen Bedingungen verwendet, welche auftreten können. Damit erreicht man mehr Zustände als mit der Zweigüberdeckung. Bei der einfachen Bedingungsüberdeckung werden alle atomaren Bedingungen mindestens in einem Testfall wahr und in einem Testfall falsch angenommen. Bei der Mehrfachbedingungsüberdeckung werden alle Kombinationen von atomaren Bedingungen in einem Testfall abgedeckt, wobei die Komplexität jedoch zur Anzahl der Variablen exponentiell wächst.

4. Pfadüberdeckung: Jeder denkbare Pfad im Kontrollflussgraphen erhält einen Testfall. Problematisch hierbei ist, dass in der Regel unendlich viele Pfade existieren. Daher werden die Pfade beschränkt darauf, dass Schleifen in einem Testfall keinmal, in einem weiteren einmal und in einem dritten Testfall mehr als zweimal durchgeführt werden.

Der **datenflussorientierte Test** betrachtet den Lebenszyklus von Variablen im Programmtext, betrachtet also das Erzeugen, Schreiben, Lesen und Vernichten einer Variable. Dazu werden dem Kontrollflussgraphen Parameter angehängt, die jeweils beschreiben, welche Variablen in diesem Knoten wie verwendet werden. Testfälle ergeben sich dann daraus, Pfade zwischen der Erstellung und der Verwendung von Variablen anhand bestimmter Kriterien zu berechnen. Diese sollen hier jedoch nicht weiter betrachtet werden. Für genauere Studien kann [Bal98] herangezogen werden.

Blackbox

In Blackbox-Verfahren ist der Programmtext nicht einsehbar und das Objekt muss anhand seiner Schnittstellen getestet werden. (nach [Has04]) Daher muss auf andere Weise eine Testabdeckung erreicht werden. Hierzu kann nur die Spezifikation des untersuchten Objektes herangezogen werden. Daher ist es zur Durchführung von Blackbox-Tests erforderlich, eine genaue Anforderungsdefinition und einen sehr genauen Software-Entwurf zu haben, damit anhand von diesen geeignete Testfälle generiert werden können. Wie diese Testfälle bestimmt werden können wird in Abschnitt 4.3.2 genauer betrachtet.

4.3. Testen von Software

Das experimentelle Ermitteln der Qualität von Software geschieht durch Tests auf dieser Software. Zumeist wird dazu die Vorgehensweise aus dem V-Modell gewählt (siehe Abschnitt 4.2.2), da dieses Modell durch den Ansatz der entwicklungsbegleitenden Qualitätssicherung das Testen von Software nicht nur voraussetzt, sondern die Rolle einzelner Testarten an bestimmten Stellen in der Entwicklung beschreibt und erfordert. Auch geht es von Anfang an darauf ein, dass für die spätere Qualitätssicherung Testfälle frühzeitig erstellt werden müssen.

4.3.1. Testbereiche im V-Modell

Das Testen teilt sich in die Bereiche Unit-Test, Integrationstest, Systemtest und Abnahmetest. Diese Reihenfolge entspricht der Bottom-Up Reihenfolge im V-Modell (nach [Wir03]).

Unit-Test

Die erste Teststufe im V-Modell sind die Tests der einzelnen Komponenten für sich. Diese Stufe wird daher Unit-Test genannt. Im Blickpunkt ist dabei besonders die Funktionalität jeder einzelnen Prozedur und Funktion, bei der objektorientierten Programmierung entsprechend der Methoden, Klassen, Interfaces und zusammenhängenden Komponenten. Es sollen besonders Fehler in der entsprechenden Komponente aufgedeckt werden, nichtfunktionale Anforderungen wie zum Beispiel Effizienz, Wartbarkeit oder eine Dokumentation werden bei diesen Tests noch nicht beachtet. Es wird üblicherweise eine Bottom-Up-Strategie verfolgt. Angefangen von den kleinsten Teilen, den Methoden, wird ein immer größerer Teil zusammenhängender Module getestet, nach den Methoden die Klassen und anschließend die komplette Komponente. Beim Test-First Ansatz werden zuerst alle Testfälle für eine komplette Komponente geschrieben und erst anschließend werden die Klassen derart implementiert, so dass sie alle diese Testfälle abdecken. Die Testfälle müssen dafür aus der Spezifikation abgeleitet werden, was einen sehr ausführlichen Feinentwurf voraussetzt.
Üblicherweise wird beim Unit-Test bei Java das Testframework JUnit verwendet (siehe Abschnitt 4.3.3).

Integrationstest

Beim Integrationstest werden die Komponenten systematisch auf ihre Zusammenarbeit getestet. Der Fokus liegt somit auf den Schnittstellen zwischen den Komponenten. Vorbedingung ist, dass die Module an sich alle bereits getestet sein müssen und man auftretende Fehler somit eindeutig Inkompatibilitäten der Schnittstellen zuweisen kann.
Um diese Komponenten miteinander testen zu können benötigt, man sogenannte Treiber und Stümpfe (siehe Abbildung 29).
Ein Stumpf ist dabei ein der getesteten Hierarchie untergeordneter Teil, welche die Funktionalität der Komponenten, die es ersetzt, so gut wie möglich simuliert. In der Regel bestehen sie aus kleinen Prototypen, welche mit wenig Aufwand erstellt wurden. Sie geben zumeist feste Werte zurück statt diese zu berechnen und geben die Eingabewerte, welche zum Aufrufen verwendet wurden, an den Tester aus, um dem Tester aufzuzeigen, welche Aufrufe ”nach unten” von der getesteten Teilhierarchie erzeugt wurden.
Ein Treiber liegt oberhalb der zu testenden Teilhierarchie. Es ruft die Komponenten auf und versorgt sie mit Eingabewerten. Die Testfälle werden an den Treiber übergeben, welcher entsprechende Eingabewerte für die zu testenden Komponenten übergibt.

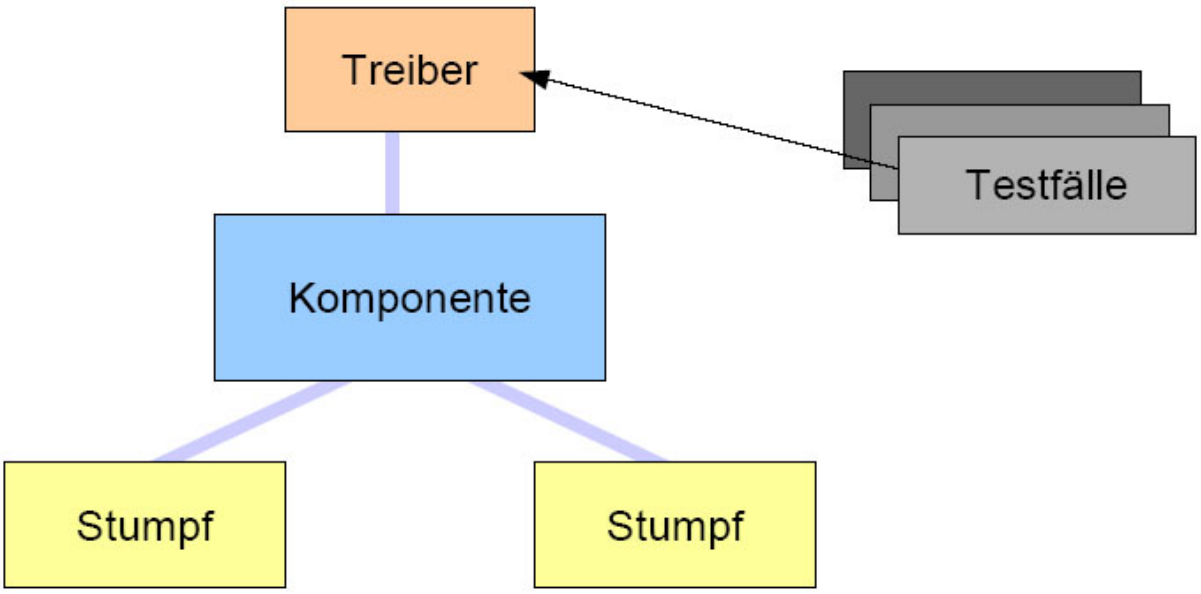

Abbildung 29: Treiber und Stümpfe im Integrationstest

Zur Durchführung des Integrationstests gibt es vier grundlegende Vorgehensweisen.
Beim **Big-Bang** werden alle Komponenten zusammen gesetzt und betrieben. Die Testfälle werden dann von Anfang an auf der kompletten Struktur ausgeführt. Ein Vorteil ist, dass der Zeitaufwand zum Erstellen der Treiber und Stümpfe entfällt. Dagegen ist das Auffinden von Fehlern erschwert, da der Test unsystematisch statt findet.
Beim **Top-Down Ansatz** beginnt man mit der am höchsten liegenden Komponente, diejenige, die keine Treiber benötigt und auf andere Klassen nur Aufrufend wirkt. Die tiefer liegenden Komponenten werden durch Stümpfe ersetzt. Die Stümpfe werden nach und nach durch die entsprechenden untergeordneten Komponenten ersetzt, wenn alle bestimmten Testfälle positiv verlaufen. Einige der Testfälle müssen beim ersetzen weiterer Stümpfe durch untergeordnete Komponenten erneut durchgeführt werden, da durch das Ersetzen sich das Ergebnis des jeweiligen Tests verändert haben könnte.
Der **Bottom-Up Ansatz** geht von den tiefsten Komponenten aus. Jede dieser Komponente bekommt einen Treiber und wird getestet. Falls ein Test gelingt wird die Komponente mit einer anderen Komponente zu einem Build kombiniert und erneut getestet. Darauf wird der Build mit einer weiteren Komponente verbunden und ein neuer Build erstellt. Dies wird solange wiederholt, bis das komplette System zusammen gesetzt wurde.
Bei der **Sandwichmethode** werden alle Komponenten entsprechend mit Treibern und Stümpfen versehen. Dann wird das System entsprechend der Testphase bei bestandennen Tests bedarfsgerecht zusammengesetzt. Dabei wird abwechselnd von Oben und Unten zusammen wachsen gelassen.

Systemtest

Der Systemtest betrachtet ähnlich dem Big-Bang Ansatz das Gesamtsystem, jedoch liegt hier das Augenmerk nicht auf der Funktionalität des Systems, da diese bereits im Unit- und Integrationstest übeprüft wurde. Statt dessen werden folgende Punkte betrachtet: Vollständigkeit, Leistung, Zuverlässigkeit, Sicherheit und Benutzbarkeit. Diese nicht-funktionalen Anforderungen an das Produkt werden unter Aufsicht der einzelnen Entwickler nachgeprüft. Auch Handbücher und Dokumentation stehen im Fokus dieser Untersuchung. Er findet in kontrollierter Testumgebung statt, so dass auftredende Fehler leichter nachvollziehbar sind.

Abnahmetest

Beim Abnahmetest wird das System dem Kunden gegeben, so dass dieser das System auf seine Sichtweisen und Anforderungen hin testen kann. Getestet wird mit echten Daten und unter normalen Betriebsbedingungen. Dies geschieht sowohl bei den Entwicklern auf extra für den Kunden eingerichteten Plattformen (Alpha-Test) als auch direkt beim Kunden auf seinen eigenen Maschinen (Beta-Test). Bei Bestehen dieser Tests gibt es eine Abnahmebescheinigung, ansonsten kommt es zu Nachbesserungsforderungen oder sogar zur Rückgabe des Produktes.

4.3.2. Bestimmen von Testfällen

Eine Schwierigkeit bei der Sicherstellung der Qualität von Software ist das Ermitteln von Testfällen. Da es aus Komplexitätsgründen nicht möglich ist, alle Eingabewerte und -folgen für jedes denkbare Programm als Testfälle zu erstellen, muss man entsprechende Untermengen erstellen, welche Gruppen von Testfällen repräsentieren. Weiter bleibt zu betrachten, welche dieser Untergruppen eine höhere Fehlerwahrscheinlichkeit haben. Somit benötigen wir Techniken zur Testdaten- und Testfallbestimmung.
Eine solche Technik ist die Äquivalenzklassenbildung (siehe [Sch04]). Dabei wird der Testraum in eine endliche Anzahl äquivalenter Klassen mit gleichen erwarteten Testergebnisses geteilt. Eine solche Klasse beinhaltet zum Beispiel alle Eingaben, welche eine bestimmte Fehlermeldung als Ausgabe erzeugen würden. Ein anderes Beispiel sind alle Eingaben, welche eine erwartete Ausgabe größer Null ergeben. Testfälle werden dann für jeweils einen Repräsentanten aus jeder Äquivalenzklasse erzeugt. Falls zur Testzeit Grund zur Annahme besteht, dass nicht alle Repräsentanten einer Äquivalenzklasse gleich behandelt werden, zum Beispiel aufgrund unerwarteter Testergebnisse, wird diese weiter aufgeteilt.
Der Ansatz der "Grenzwertanalyse" geht davon aus, dass Fehler besonders im Zusammenhang mit den Grenzen von Wertebereichen auftreten (siehe ebenfalls [Sch04]). Somit werden diese Fehler besonders an den Rändern der gebildeten Äquivalenzklassen sichtbar. Daher ist es sinnvoll außer einem beliebigen Repräsentaten aus der Äquivalenzklasse zu wählen, eher jeweils einen Testfall für einen Wert in der Nähe zu jedem Rand der Klasse zu erzeugen. Ein Beislpiel hierfür ist eine Klasse, welche eine Division

durchführt. Es ergeben sich unter anderem die drei Äquivalenzklassen ”Divisor < 0”, ”Divisor = 0” und ”Divisor > 0”. Die Ränder der dritten betrachteten Äquivalenzklasse wären somit eine sehr kleine positive Zahl, welche einen Überlauf bei Durchführung der Rechnung erzeugen könnte, oder eine sehr große positive Zahl, welche einen Unterlauf erzeugen könnte. Bei Betrachtung nur eines Repräsentanten wären diese Sondefälle nicht bedacht worden. Grenzwertanalyse soll sowohl für Eingabewerte geschehen, als auch bei den erwarteten Ausgabewerten, es sollen also Eingabewerte erzeugt werden, die Ausgaben erzeugen, welche an den Rändern von Äquivalenzklassen über die Ausgabewerte liegen.

4.3.3. JUnit - Software-Unterstützung für das Testen von Java Code

JUnit ist ein Testframework zum erstellen automatischer Unit-tests in Java. Es wurde geschrieben von Erich Gamma und Kent Beck. JUnit ist Open Source und steht unter der Common Public License Version 1.0 [Ope06]. Das Framework entstammt dem Extreme Programming und ist deswegen besonders für den Test-First Ansatz geeignet, kann jedoch auch zum nachträglichen Testen eingesetzt werden. JUnit ist erhältlich bei `http://www.junit.org/` .
Beim Einsatz von JUnit wird für jede zu testende Klasse eine weitere Klasse implementiert, welche diese Klasse testet. Dabei wird von JUnit eine einzelne Testmethode aufgerufen, in der mehrere Testfälle, die üblicherweise einen Zusammenhang haben, enthalten sind. Diese Testmethode ruft darauf, entsprechend den implementierten Testfällen, die zu testende Klasse auf und führt die Testfälle aus. Zusätzlich hat man die Möglichkeit alle Testmethoden einer Testklasse zu einer Testsuite zusammen zu fassen, welche dann von JUnit automatisch ausgeführt werden. Weiter bietet JUnit eine Methode setUP(), in welcher die Objekte initialisiert werden, die zum Test benötigt werden, sowie eine Methode tearDown(), um alle erstellten Objekte zu entfernen. Diese beiden Methoden werden vor und nach jeder Durchführung eines Testfalles ausgeführt.
Wenn ein Testfall ausgeführt wird und der Test gelingt, so passiert garnichts. Nur falls der Test fehlschlägt und die Bedingungen des Testfalles nicht eintreten, so wird ein AssertionFailedError geworfen. Dieser wird vom JUnit Framework abgefangen und behandelt. Der Entwickler enthält anschließend eine Benachrichtigung über die fehlerhaften Testfälle. Mit diesen Informationen kann er nun seine Klasse überarbeiten und den Test erneut ausführen. Dies wird solange wiederholt, bis alle Testfälle bestanden sind, da eine Änderung schon bestandene Testfälle wieder verwerfen kann.

4.4. Inspektionen, Reviews und Walkthroughs

Syntax, Konsistenz und Vollständigkeit können durch konstruktive Qualitätssicherungsmaßnahmen automatisch kontrolliert werden. Die Semantik kann jedoch nicht automatisch geprüft werden, dazu sind analytische Verfahren notwendig. Dabei gibt es drei wesentliche manuelle Prüfmethoden: Inspektionen, Reviews und Walkthroughs (vergleiche [Fäh02] und [Sch04]). Ihr gemeinsames Ziel ist es, Fehler und Inkonsistenten in

allen erstellten Dokumenten aufzudecken. Dazu werden Gruppen gebildet, in welchen die Teilnehmer entsprechend den benötigten Rollen aufgeteilt werden. Die Gruppen bestehen zum größten Teil aus den Entwicklern des Projektes. Dabei sollen weder Vorgesetzte noch Zuhörer anwesend sein. Von Vorteil bei diesen Methoden ist, dass alle Teilnehmer ein fundiertes Wissen über die untersuchten Dokumente erhalten sowie ein Verständnis für die Arbeitsmethoden der Kollegen entwickeln. Dem gegenüber steht ein sehr hoher Aufwand, der bis zu 20% der Erstellungskosten des Gesamtprojektes einnehmen kann (laut [Fäh02]).
Bei einer Inspektion wird geprüft, ob Teilprodukte mit der Spezifikation übereinstimmen und die verwendeten Standards und Paradigmen einhalten. Erst nach der Inspektion werden die Teilprodukte für eine weitere Verwendung im folgenden Entwicklungsprozess freigegeben. Bei einer Inspektion teilen sich die Rollen auf in einen unabhängigen Moderator, Gutachter bestehend aus Kollegen des Autors, sowie einen Protokollführer. Der Autor des Teilproduktes selbst soll an einer Inspektion nicht teilnehmen. Die Inspektion wirdanhand eines vom Moderator erstellten Inspektionsplans in einer Sitzung durchgeführt. Dabei soll maximal eine Seite pro Stunde bearbeitet werden, um eine entsprechende Genauigkeit der Untersuchung zu erreichen.
Bei einem Review wird das Teilprodukt auf seine Stärken und Schwächen hin untersucht. Anders als bei der Inspektion wird der Reviewplan jedoch nicht vom Moderator erstellt, sondern vom Autor vorgeschlagen. Dieser wird zusammen mit dem Teilprodukt vom Moderator und dem Autor einer Eingangsprüfung unterzogen und danach in einer Sitzung reviewed. Hierbei soll jedoch anders als bei der Inspektion der Autor ebenfalls anwesend sein, aber eine passive Rolle einnehmen, um bei möglichen Rückfragen Auskunft geben zu können. Bei einem Review sollen keine Verbesserungen ermittelt und diskutiert werden, sondern nur die Stärken und Schwächen erkannt und markiert werden.
Ein Walkthrough ist eine abgeschwächte Form des Review, bei der der Autor selbst auch der Moderator ist. Ein Walkthrough ist dazu gedacht das Wissen über das Teilprodukt an die Kollegen weiter zu geben. Die Untersuchung auf Stärken und Schwächen findet dazu parallel statt, ist aber wesentlich uneffektiver als im Review. Bei einem Walkthrough können auch Endbenutzer eingeladen werden, um Missverständnisse und Unvollständigkeiten aufzudecken.

4.5. Metriken zum Bewerten von Software

Eine Softwaremetrik ist eine Funktion, welche die Eigenschaften der Software auf eine Zahl abbildet. Anhand dieser Zahl soll dann interpretierbar sein, wie hoch der Erfüllungsgrad der Software mit den Anforderungen ist (nach [Sch04]). Das Hauptproblem hierbei ist, eine Funktion zu bestimmen, welche eine solche Zahl sinnvoll ermitteln kann. Einfachstes aber auch schlechtes Beispiel ist das Betrachten der Zeilen an Code des Quelltextes. Hierbei werden Eigenschaften wie Kommentare, Zeilenlänge, Strukturiertheit, Vererbung und Programmierstil unbewertet mit in die Zahl auf-

genommen. Auch Metriken wie das Zählen der Zeichen, Klassen, Variablen, Funktionen, Schachtelungstiefen oder Schnittstellen verbessern den Informationsgehalt kaum. Andere Ansätze betrachten den Lebenszyklus der Software, zum Beispiel Anrufe von Kunden beim Kundendienst pro Monat, Verhältnis zwischen gefundenen und behobenen Fehlern oder der Testüberdeckungsgrad. Das Ziel beim Einsatz einer Metrik ist es, bestimmte betrachtete Qualitätsmerkmale zu untersuchen. Daher muss eine Metrik entsprechend des betrachteten Qualitätsmerkmales bestimmt werden. Hierbei muss ermittelt werden, welche Zahlenwerte erfasst werden können, um die entsprechende Eigenschaft wieder zu spiegeln. Diese gemessenen Zahlen dienen dann dazu den Code an der entsprechenden Stelle, an der die Qualität betrachtet wird, zu verbessern.

4.6. Verbesserung der Prozessqualität

Ein Ansatz bei der Verbesserung der Qualität von Software ist es, die Qualität des Erstellungsprozesses dieser Software zu verbessern.
Von der International Standard Organisation gibt es die ISO 9000 sowie ISO 9001 (nach [Fäh02] und [Sch04]). Sie ist ein allgemeiner Qualitätssicherungsstandard, welcher nicht speziell auf Softwareentwicklung abgestimmt ist. Dazu dient der Abschnitt ISO 9000-3, welcher die ISO 9001 auf Anwendung zur Softwareentwicklung beschreibt. ISO 9001 beschäftigt sich mit Qualitätsmanagementsystemen in Bezug auf Produktion, Montage, Endprüfung und Wartung. Es werden Maßnahmen sowohl für die Geschäftsleitung als auch für die Mitarbeiter festgelegt, welche sie zur Qualitätssicherung beitragen müssen. Anhand der ISO 9000 können sich Unternehmen zertifizieren lassen.
Weitere Standards zur Verbesserung der Qualität sind TQM (Total Quality Management) oder CMM (Capability Maturity Model) (vergleiche [Fäh02]). Bei TQM wird das gesamte Unternehmen mit in die Qualitätssicherung einbezogen und was Qualität im einzelnen bedeutet wird direkt vom Kunden bestimmt. Ständige Verbesserung des Prozesses sowie eine kundenorientierte Sichtweise auf die Entwicklung sind die Hauptaspekte. CMM betrachtet die Erstellung von Software anhand von fünf Reifegraden. Dabei müssen alle Anforderungen einer Stufe im Prozess berücksichtigt werden, um die nächst höhere Stufe zu erreichen. Jede Stufe hat dabei Hauptkriterien, deren Erfüllung anhand von Fragebogen ermittelt werden soll. Beim erreichen einer höheren Stufe sinken gegenüber der vorherigen Stufe die Risiken wogegen Produktivität und Qualität steigen sollen.

4.7. Fazit

Um eine hohe Qualität bei Software-Produkten zu erhalten, ist es notwendig Qualitätssicherung als wichtiges Element der Entwicklung anzusehen. Die konstruktiven Qualitätssicherungsmaßnahmen laufen dabei entwicklungsbegleitend, während analytische Maßnahmen sowohl zwischendurch den Entwicklungsprozess unterbrechen als auch am Ende durchgeführt werden müssen. Um ordentliche Testfälle ermitteln zu können,

sind zudem eine sehr gut ausgearbeitete Anforderungsdefinition sowie ein sehr präziser Entwurf nötig. Qualitätsmanagement bietet den Entwicklern dazu Verfahren und Methoden, welche während und nach der Implementierung verwendet werden können.

Literatur

[91291] 9126, DIN ISO: *Informationstechnik - Beurteilen von Software-Produkten, Qualitätsmerkmale und Leitfaden zu deren Verwendung*. Beuth Verlag Berlin, 1991.

[Bal98] BALZERT, H: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 1998.

[Fäh02] FÄHNRICH, KLAUS PETER: *Software-Qualitätsmanagement*. Skript zur Vorlesung, 2002.

[Has04] HASSELBRING, WILHELM: *Software Engineering*. Skript zur Vorlesung, 2004.

[Ope06] THE OPEN SOURCE INITIATIVE (OSI): *Common Public License Version 1.0*, 2006. Internet: http://www.opensource.org/ (29.05.2007).

[Sch04] SCHLINGLOFF, HOLGER: *Qualitätssicherung von Software*. Skript zur Vorlesung, 2004.

[Wir03] WIRSING, MARTIN: *Methoden des Software-Engineering*. Skript zur Vorlesung, 2003.

5. Oberflächenframeworks

Autor: Christian Hinrichs

5.1. Einleitung

Noch vor einigen Jahrzehnten wurden Computerprogramme ausschließlich über die direkte Befehlseingabe auf der Konsole gesteuert. Grafische Oberflächen waren nur sehr umständlich zu implementieren und boten auch dann noch keinen nennenswerten Komfort. Heutzutage besteht ein wesentlicher Teil einer Anwendung aus dem Design und der Implementierung einer ansprechenden grafischen Oberfläche. Der Benutzer möchte intuitiv durch die Bedienung geleitet werden. Daten sollen auf elegante Weise repräsentiert werden. Und nicht zuletzt möchte der Benutzer einen gewissen Einfluss auf die Oberfläche ausüben können, um sie seinen persönlichen Wünschen anzupassen. Diese Arbeit soll einen Einblick in aktuelle Werzeuge zur Erstellung komplexer Oberflächen in Java bieten und verschiedene Alternativen aufzeigen, welche in dem jeweiligen Szenario einer Anwendungsentwicklung mehr oder weniger sinnvoll sind.

5.2. Basis Frameworks

Die Basis Frameworks, oder auch *Window Toolkits*, stellen die Grundlage einer grafischen Repräsentation von Elementen zur Steuerung von Anwendungen dar. Sie definieren Schaltflächen, Texte, Eingabefelder, Dekorationen, Fenster usw. Außerdem erlauben sie das Setzen von Schriftarten, Farben sowie das Zeichnen von Linien, Polygonen etc. Zusätzlich ist jeweils eine Ereignisbehandlung enthalten, wodurch auf Benutzeraktionen reagiert werden kann. Im Folgenden werden die drei bekanntesten Programmierschnittstellen (APIs) vorgestellt.

5.2.1. AWT - Abstract Window Toolkit

Das AWT ist in den *Java Foundation Classes* (JFC) enthalten, welche mit der Java Platform (J2SE, J2EE) ausgeliefert werden. Es existiert seit ca. 1996.
Diejenigen Elemente des AWT, welche Betriebssystemkomponenten darstellen, also Schaltflächen und ähnliches, werden durch sogenannte *Peer Driven Widgets* realisiert (vgl. [Ull07]). Das bedeutet, dass diese Komponenten nicht von Java gerendert, sondern vom Betriebssystem selbst gezeichnet und verwaltet werden. Dies hat den Vorteil, dass die Anwender sich bei der Bedienung derartiger Oberflächen nicht umgewöhnen müssen, was das Aussehen der Bedienelemente betrifft. Außerdem besitzen diese schwergewichtigen Elemente eine sehr schnelle Reaktionszeit, weil das Betriebssystem die Eingaben direkt verarbeitet (auch *heavyweight* genannt), anstatt diese Ereignisse der Java Platform weiterzuleiten. Im AWT sind Peer Komponenten für alle Betriebssysteme, auf denen Java läuft, implementiert. Da jedoch nicht alle Systeme die gleichen Widgets

besitzen existiert im AWT nur die Schnittmenge der Gui-Elemente über die Betriebssysteme. Zum Beispiel gibt es unter Motif keine Listen, und damit auch nicht im AWT. In Abbildung 30 ist exemplarisch das typische Aussehen einer AWT Oberfläche unter Windows dargestellt[3].

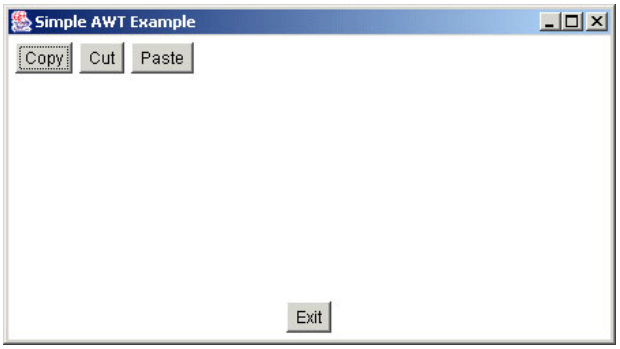

Abbildung 30: AWT Oberfläche

5.2.2. Swing

Swing besteht seit 1998 und ist seitdem wesentlicher Bestandteil der JFC (vgl. Abschnitt 5.2.1). Swing bietet zusätzlich zu AWT neben nützlichen Komponenten wie Listen und Bäumen außerdem einige komfortable Neuerungen. So wird Transparenz unterstützt, ebenso wie Drag'n'Drop und Tooltips. Auch die direkten Zeichenmöglichkeiten wurden mit Java2D stark erweitert. So können nun komplexe grafische Objekte durch Pfade definiert und darauf verschiedentliche Bewegungs- und Verschiebeoperationen durchgeführt werden (siehe [Ull07]).
Der größte Unterschied zu AWT besteht jedoch darin, dass die Komponenten nicht mehr durch das Betriebssystem verwaltet werden. Alle Elemente wie Knöpfe, Auswahllisten etc. werden direkt von Swing gerendert, das Framework ist somit leichtgewichtig (*lightweight*, das Gegenteil zu *heavyweight* aus Abschnitt 5.2.3) und absolut portabel. Damit die Benutzer sich jedoch nicht komplett auf ein neues Aussehen einlassen müssen wurden sogenannte *Look & Feels* implementiert, welche das Aussehen diverser Betriebssysteme nachahmen und sich im laufenden Betrieb austauschen lassen. In Abbildung 31 ist ein typisches Swing Fenster unter Windows XP zu sehen[4].

5.2.3. SWT - Standard Widget Toolkit

Das SWT existiert seit 2001 und wurde ursprünglich für die Entwicklungsumgebung Eclipse entwickelt. SWT ist eine Hybridlösung zwischen schwer- und leichtgewichtig.

[3] http://www.idevelopment.info/data/Programming/java/swing/AWTExample.html
[4] http://www.attainware.com/beanview/beanview101/docs/screenshots.html

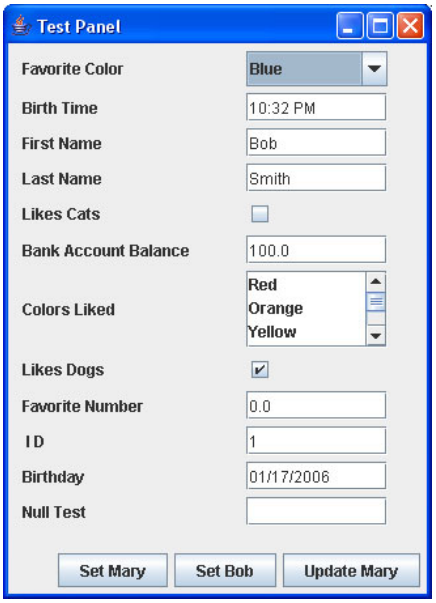

Abbildung 31: Swing Oberfläche

Es setzt auf einem kleinen nativen Kern auf, welcher in der Lage ist, die auf den Betriebssystem vorhandenen Widgets nativ zeichnen zu lassen. Auf der Java Seite werden für grafische Komponenten möglichst kleine *Wrapper* (Hüllenklasse) eingesetzt, welche auf die native Bibliothek zugreifen. Diese Bibliothek ist für das jeweilige Betriebssystem unterschiedlich, auf dem die Anwendung läuft. Sie ist für die meisten populären Systeme vorhanden, darunter Windows, Linux, BSD, Solaris, MacOS, QNX sowie diverse mobile Plattformen wie Windows CE etc. (vgl. [The07d]). Ist ein benötigtes Widget auf der Zielplattform nicht vorhanden, so wird es vom SWT gerendert.
Die Widgets des SWT sind, wie bei AWT und Swing, hierarchisch aufgebaut, sodass sich eigene Kompositionen bestehender Elemente erstellen lassen. Es existiert eine große Vielfalt an Widgets für viele denkbare Anwendungsbereiche, so etwa Elemente für Datumseingaben, TrayItems, stilisierter Text usw. Der Funktionsumfang deckt sich in etwa mit Swing. Daduch und durch die Breite der verfügbaren nativen Kerne vereint SWT nahezu alle Vorteile von AWT und Swing, ohne ihre Nachteile zu erben. In Abbildung 32 ist eine SWT Anwendung unter Windows Vista dargestellt[5].

5.3. OSGi Plattform

Die OSGi Alliance ist eine non-profit Organisation, bestehend aus einem Konsortium von Technologie-Innovatoren, darunter beispielsweise die Deutsche Telekom AG, IBM, Intel, Oracle, Samsung und viele andere (entnommen [OSG07b]). In diesem Konsortium wurde mit der Open Services Gateway Initiative (*OSGi*) eine Spezifikation für eine

[5] http://www.eclipse.org/swt/examples.php#insideEclipse

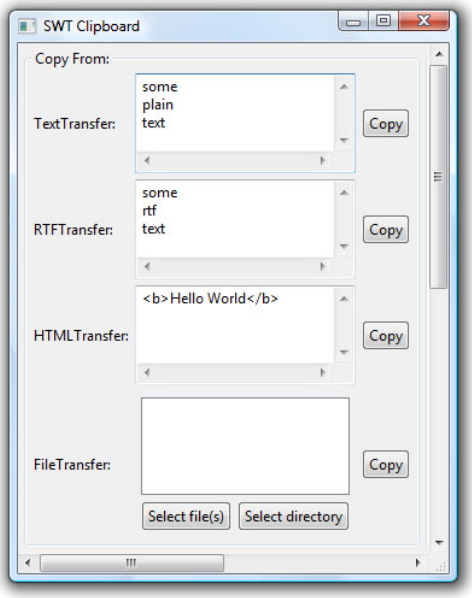

Abbildung 32: SWT Oberfläche

Service-orientierte Architektur erstellt. In dieser Architektur stellen einzelne *Services* die Funktionalitäten dar, bereitgestellt in sogenannten *Bundles*. OSGi spezifiziert somit eine Laufzeitumgebung für Services, welche auf einer Vielzahl von Plattformen, in überwiegend vernetzten Umgebungen, lauffähig sind. Dazu zählen:

- Telematik (Fahrzeugtechnik),
- Mobile Endgeräte (Handy, PDA),
- Residential Gateways (Router),
- Unterhaltungselektronik (Verteilung von Multimediainhalten)

und seit dem Release R4 (Oktober 2005) auch

- Enterprise Anwendungen (Eclipse).

Eine Plattform, welche nach der OSGi Spezifikation implementiert wurde bietet folgende Möglichkeiten:

- *Software Component Management*: Die Plattform steuert den Lebenszyklus aller auf ihr gestarteten Anwendungen und Komponenten.
- *Remote Component Management*: Da OSGi für vorwiegend vernetze Systeme gedacht ist, werden nahezu alle gebräuchlichen Protokolle unterstützt, wodurch sich die einzelnen Komponenten per Fernzugriff steuern lassen.

- *Secure Execution Environment*: Die Sicherheit der Daten und der Ausführung wird durch Mechanismen der JVM, sowie durch eigene Funktionen garantiert.
- *Commercial Off The Shelf Components*: Die zahlreichen Mitgliedsorganisationen erstellen fortlaufend neue Komponenten für viele Anwendungsbereiche, wodurch eine Palette an vorgefertigten Bundles verfügbar ist.
- *Simplified Deployment, Interopability, and Dynamic Functionalities*: Die Spezifikiation einer OSGi Plattform ist so gestaltet, dass alle Bundles unabhängig von der konkreten Implementierung der Plattform gestartet, dynamisch nachgeladen und geupdated werden können.

Quelle: OSGi Frequently Asked Questions [OSG07a]

5.3.1. Equinox

Equinox ist ein Projekt der Eclipse Community und stellt eine konkrete Implementierung der OSGi Spezifikation dar (siehe [The07b]). Dieses Projekt existiert seit etwa 2005 und hat die Eclipse IDE insofern verändert, dass sie als Menge von Bundles verstanden werden kann, welche auf Equinox gestartet werden.

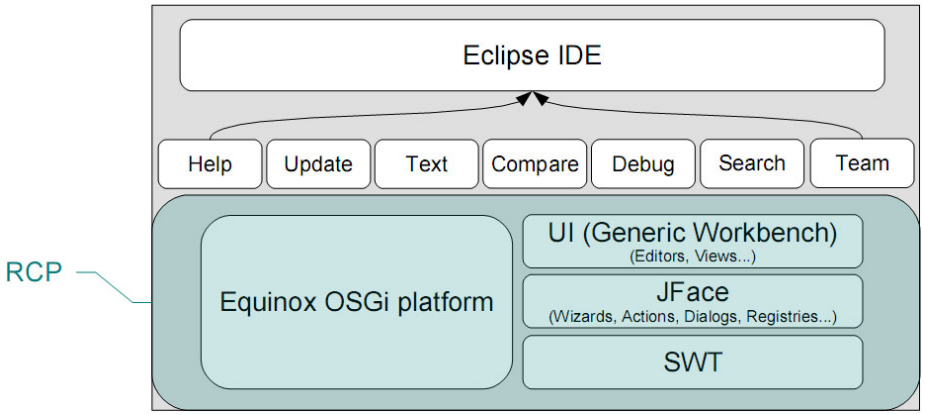

Abbildung 33: Struktur der Eclipse IDE

In Abbildung 33 ist schematisch der Aufbau der Eclipse IDE dargestellt. Die unterste Ebene (bezeichnet mit RCP, siehe Abschnitt 5.4.1) besteht einerseits aus Equinox (der OSGi Plattform) und parallel aus einer Vereinigung von drei grafischen Paketen: Zunächst die Basisbibliothek SWT, welche die grundlegenden visuellen Elemente bereitstellt. Darauf setzt dann das JFace auf, welches SWT-Komponenten nutzt, um Elemente wie Wizards und Dialoge, aber auch abstraktere Funktionalitäten wie Actions und Registries anzubieten (eine *registry* ist im JFace ein Datenpool, welcher betriebssystemlastige Resourcen wie Bilder und Farbobjekte verwaltet). Dieses wiederum wird vom Paket UI verwendet, um den *Workbench* zu erstellen, welcher aus Editoren, Views etc. besteht

(siehe auch Abschnitt 5.4.1 bzw. [Dau05]). Diese beiden Komponenten der untersten Ebene liefern somit den nötigen Kern, um die Eclipse IDE aufzubauen: In Equinox werden nun einzelne Bundles gestartet, welche jeweils eine Funktion der IDE darstellen und intensiv Gebrauch von den grafischen Paketen machen, dazu zählen die dargestellten Komponenten *Help*, *Update*, *Text* usw. In der Abbildung 34 ist dieser Zusammenhang noch einmal am Bild einer Eclipse IDE dargestellt.

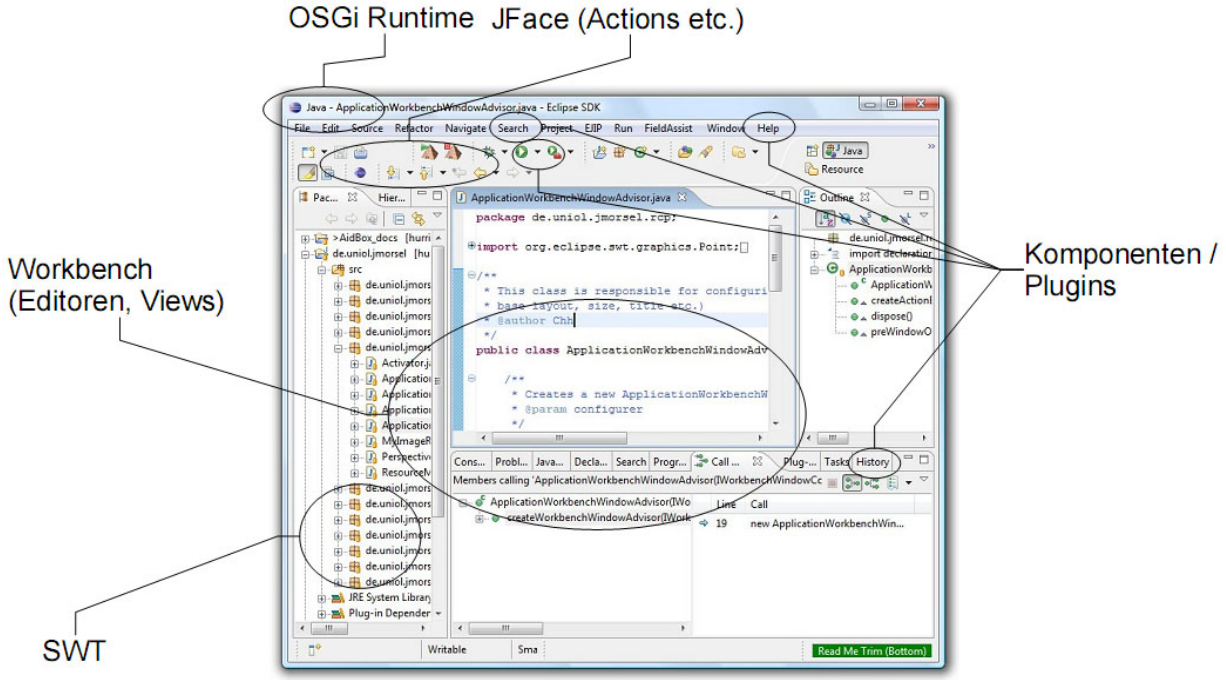

Abbildung 34: Komponenten der Eclipse IDE

5.4. Rich Client Konzepte

Ein Rich Client bezeichnet einen Client, der sowohl die grafische Repräsentation, als auch die Verarbeitung der Daten lokal vornimmt. Er besteht meist aus einer Sammlung von Plugins, welche seine eigentlichen Funktionalitäten definieren. Zur Erstellung von Rich Clients gibt es Frameworks, welche alle eine Fülle von vorgefertigten Modulen zur freien Verwendung bereitstellen. Die folgende Liste zeigt eine exemplarische Auswahl:

- Dialoge
- Wizards
- Validierung von Eingaben
- Toolbars

- Internationalisierung
- Registries
- Progress monitoring
- GUI Builder
- Hilfesystem

Im Folgenden werden einige populäre Frameworks vorgestellt.

5.4.1. Eclipse Rich Client Platform

Das Eclipse RCP Framework bietet dem Entwickler ein mächtiges Werkzeug zur Erstellung von Rich Client Architekturen. Es besteht aus dem mit RCP bezeichneten Teil in Abbildung 33. Man kann somit als Entwickler direkt die Sourcen der Eclipse IDE verwenden, um einen Rich Client zu erstellen. Minimal muss ein solcher Client aus eben den beiden markierten Paketen bestehen, also `org.eclise.core.runtime` und `org.eclipse.ui`. Wahlweise können dann beliebige andere Bundles hinzugefügt werden, unter anderem alle in der Eclipse IDE enthaltenen Komponenten.
Die Vorteile der Eclipse RCP sind:

- sehr schnelle Entwicklung einer professionell aussehenden Applikation
- natives Look & Feel
- Kompilierung für diverse Plattformen: Windows, Linux, BSD, Solaris, MacOS, QNX...
- Konfiguration über XML-Dateien
- Workflow durch Implementierung von Actions

Das Eclipse RCP Framework gestaltet die Arbeitsfläche des Rich Clients prinzipiell nach einem Containerprinzip. Das bedeutet, dass es verschiedene Klassen von Containern gibt, welche visuelle Elemente enthalten können. Diese Container besitzen dann abhängig von ihrer Art Features, ohne dass der Entwickler diese selbst implementieren muss. Beispielsweise gibt es den Container *Editor*, welcher die Eigenschaft hat, eine Datenquelle wie etwa eine Datei als Basis zu besitzen und auf dieser zu arbeiten. So ist z.B. der mittlere Container in Abbildung 34, welcher einen Java-Quelltext anzeigt, vom Typ Editor, da er eine .java Datei darstellt und bearbeitet. Daneben gibt es *Views*. Diese Container sind abhängig von anderen Gegebenheiten der Anwendung. So ist etwa die *Outline* (in der Abbildung rechts) ein View, welcher sich immer auf den gerade aktiven Editor bezieht und die Struktur der geöffneten Java Datei anzeigt.

Die Besonderheit dieses Containerprinzips ist die dahinter liegende Grundfunktionalität. So lassen sich die einzelnen Views, Editoren etc. durch den Benutzer beliebig durch Drag'n'Drop in der Arbeitsfläche anordnen, vergrößern, deaktivieren etc. Der Entwickler muss diese Funktionen einfach nur in der entsprechenden Konfigurationsdatei seiner Anwendung aktivieren.
Die Eclipse IDE unterstützt den Entwickler bei der Erstellung eines RCP Projektes durch Wizards, welche die grundlegende Code-Struktur selbst generieren. Diese Struktur muss dann vom Entwickler mit Inhalt gefüllt und kann bei Bedarf erweitert werden. Für die Konfiguration der XML Dateien stellt Eclipse Hilfsmittel in Form von Editoren zur Verfügung, sodass kein XML-Code von Hand getippt werden muss. Dazu zählen Entwicklungsschritte wie das Einbinden neuer Views und Editoren als auch erweiterte Einstellungen wie Internationalisierung oder Branding des Produktes durch Namen, Versionsnummern, About-Informationen, Icons und ähnlichem.

Üblicherweise besteht ein Eclipse RCP Projekt aus folgenden Klassen:

- *Activator*, abgeleitet von *AbstractUIPlugin*: Diese Klasse definiert das eigentliche Bundle bzw. Plugin, welches man entwickelt. Sie ist für den gesamten Lebenszyklus des Plugins sowie Bundle-spezifische Komponenten wie Preferences oder die Image Registry verantwortlich.

- *Application* implementiert *IPlatformRunnable*: Die Application Klasse stellt den Einstiegspunkt dar. Sie besitzt eine run()-Methode, welche zum Start der Anwendung vom OSGi Starter aufgerufen wird und das *Display* sowie den Workbench initialisiert. Die Klasse *Display* repräsentiert die Betriebssystemressource, auf der gezeichnet werden soll, also üblicherweise den Bildschirm.

- *ApplicationActionBarAdvisor*: Dieser Advisor erstellt und konfiguriert die ActionBar der Anwendung, also den Bereich, welcher die Menüs sowie die *Coolbar* enthalten soll. Die *Coolbar* ist das Eclipse-Äquivalent einer Toolbar mit direkt zugänglichen Menükomponenten in Form von Buttons.

- *ApplicationWorkbenchAdvisor*: Der Workbench Advisor erstellt den Workbench, indem Methoden zu speziellen Zeitpunkten des Anwendungs-Lebenszyklus aufgerufen werden (z.B. preWindowOpen() - wird direkt vor dem Anzeigen des Fensters ausgeführt) sowie der zu benutzende Window Advisor sowie die initiale Perspektive festgelegt wird.

- *ApplicationWorkbenchWindowAdvisor*: Der Window Advisor konfiguriert das Layout des Fensters der Anwendung. Dazu zählen Größe, Anzeige der Statuszeile etc.

- *Perspective*: Eine Perspektive legt fest, welche Views und Editoren vom System angezeigt werden, sowie in welcher Größe und Position dies geschehen soll.

Daneben gibt es dann Editoren, welche wie bereits oben beschrieben auf externen Daten arbeiten, und Views, welche alle anderen Arten der visuellen Repräsentation vornehmen, sowie Actions: diese sind ein wichtiger Bestandteil des RCP Designs, denn sie stellen den Arbeitsfluss der Anwendung dar. Jedem Menüpunkt und Button sollte eine Action zugeordnet sein, in der Action wird dann die Ereignisbehandlung zu diesem Element implementiert.
Das Framework bietet auch hier dem Benutzer viele fertige Klassen zur Benutzung an. So gibt es Editoren für bestimmte Arten von Dateien (für Texteditoren gibt es im Framework sogar ein ganzes Paket), Views für Standardaufgaben wie etwa die Anzeige einer Baumstruktur und eine gewisse Menge von standardisierten Actions, wie etwa *Save* und *Close*, welche sich auf einen Editor beziehen, oder *About* und *Help* zur Anzeige von erweiterten Informationen.

Ein weiteres wichtiges Feature ist der Produktexport der Plattform. In diesem Prozess wird eine minimale Equinox Umgebung für ein Zielsystem erstellt, welcher dann die erforderlichen Bundles für den entwickelten Rich Client (einschließlich des Clients selbst, denn jeder Rich Client in Eclipse stellt ja ein eigenständiges Plugin dar) hinzugefügt werden. Der Anwender kann dieses Produkt auf dem Zielsystem dann direkt starten, ohne weitere Abhängigkeiten beachten zu müssen, abgesehen von der Java Runtime Environment, welche aber auch sehr einfach unter Einhaltung der Lizenzbestimmungen mit in das Produkt integriert werden kann.
Auch für den Export steht ein Wizard zur Hilfe zur Verfügung, welcher dem Entwickler bei der Auslösung des Produktes aus der Entwicklungsumgebung in ein eigenständig lauffähiges Paket behilflich ist.

Weitere Informationen, Tutorials etc. finden sich in [The07c] und [Dau05].

5.4.2. Netbeans platform

Netbeans ist ebenso wie die Eclipse IDE eine Entwicklungsumgebung für Java Anwendungen. Mit *platform* (vgl. [Net07]) wurde in Netbeans ein Unterprojekt realisiert, welches im Funktionsumfang nahezu äquivalent zum Eclipse RCP ist.
In Abbildung 35 ist exemplarisch die Anwendung *Poseidon for UML* dargestellt, welche mit Netbeans *platform* erstellt wurde[6]. Die konzeptuellen Unterschiede zu einem typischen Eclipse Rich Client sind kaum vorhanden: auch hier finden sich Menüs, eine Toolbar, sowie diverse Editoren und Views. Auch Features wie Actions, das Plugin-Prinzip, der Produktexport sowie das Branding sind vorhanden. Der konkrete Unterschied ist jedoch das zugrunde liegende Basis Framework, in diesem Fall ist es Swing.

[6] http://www.netbeans.org/images/screenshots/platform/poseidon_uml.png

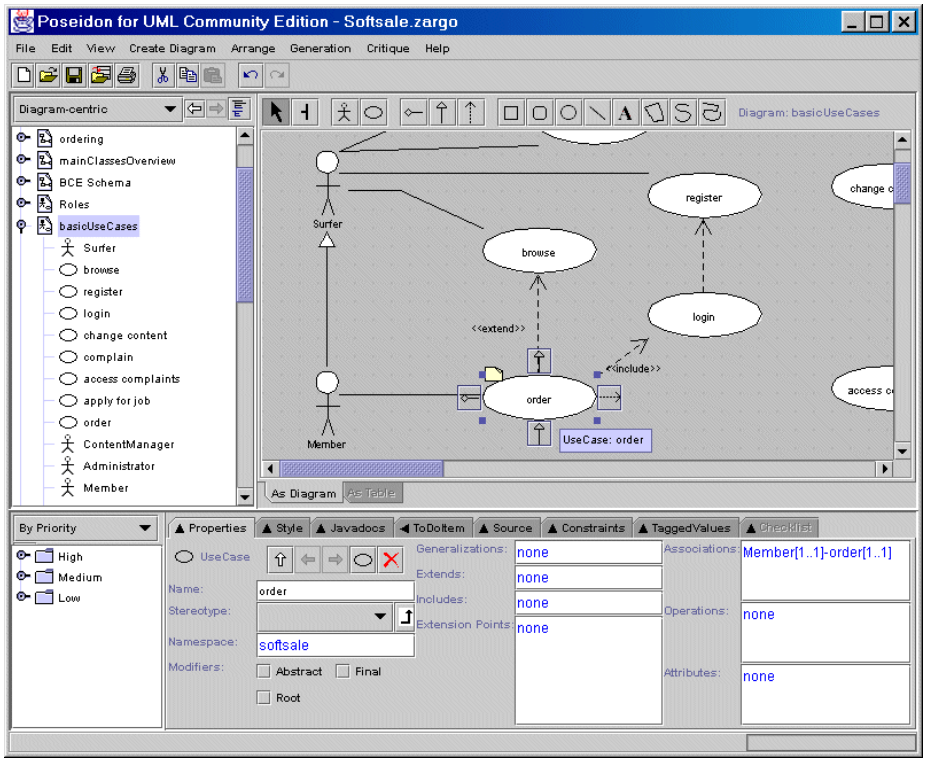

Abbildung 35: Poseidon for UML als Beispiel einer Netbeans RCP Anwendung

5.4.3. Spring Rich Client Project

Das Spring Framework ist ähnlich wie Eclipse RCP ein freies Framework zur Erstellung von Anwendungen mit Java. Wie auch bei Eclipse RCP enthält Spring diverse Subprojekte, darunter Pakete zur Unterstützung der Entwicklung von Webanwendungen, eine eigene Entwicklungsumgebung, und auch ein Projekt zur Erstellung von Rich Clients. Dieses wird in der Kurzform *Spring Rich* genannt und hat ähnliche Eigenschaften wie Netbeans *platform*. Es basiert ebenfalls auf Swing, wobei die Entwickler sich in diesem Punkt ganz deutlich von der nativen Lösung mittels SWT abgrenzen. Sie sind der Meinung, dass Swing inzwischen so gut weiterentwickelt wurde, dass es einer nativen Alternative in nichts nachsteht, sei es die Reaktionszeit, das Aussehen oder sonstiges (Quelle: [The07e]). In Abbildung 36 ist deutlich der Unterschied zur Abbildung des Netbeans Frameworks in Abschnitt 5.4.2 zu erkennen[7], dies ist jedoch teilweise in dem verwendeten *Look & Feel* begründet.
Der wesentliche Vorteil von Spring Rich liegt darin, dass es in das Spring Framework integriert ist und die Verwendung dieses Frameworks sich anbietet, wenn das Spring Framework bei der Entwicklung der Anwendung zum Einsatz kommt.

[7] http://spring-rich-c.sourceforge.net/screenshots/simple2.png,

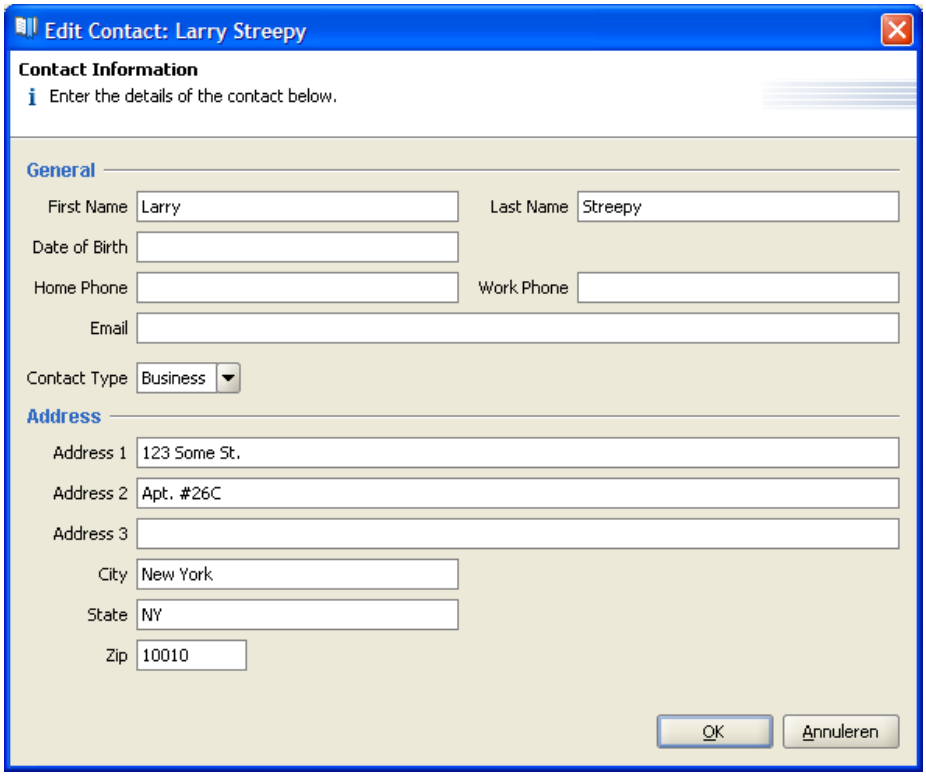

Abbildung 36: Teil einer Spring-Rich Anwendung

5.5. Weitere Tools

Neben den oben dargestellten Frameworks gibt es eine Reihe von kleineren nützlichen Tools und Bibliotheken, welche die Erstellung und Wartung von ansprechenden Oberflächen erleichtern. Im Folgenden werden einige populäre Beispiele vorgestellt.

5.5.1. JFreeChart

Die von der Projektgruppe zu entwickelnde Software soll unter anderem die verarbeiteten Daten grafisch aufbereiten und präsentieren. JFreeChart ist eine Java Bibliothek zur Erstellung von einfachen bis sehr komplexen Diagrammen. Durch das objektorientierte Design und die Anlehnung an Swing lassen sich die Diagramme recht einfach wie GUI-Elemente zusammenbauen. Die möglichen Ausgabeformate sind einerseits pixelbasierte Bilder wie JPEG und PNG, aber auch vektorbasierte Formate wie PDF, EPS und SVG. Zudem lassen sich die Diagrammkonstrukte direkt in einer Swing-Umgebung als GUI-Elemente verwenden, was die Einbettung insbesondere in eine Swing basierte Anwendung stark vereinfacht, und mit Hilfe von Wrapper Klassen auch für andere Basis Frameworks nützlich ist. Die Datenquellen der Diagramme können auf unterschiedlichem Wege importiert werden, darunter die Datenbankanbindung über JDBC und auch das Einlesen von Dateien im XML Format (entnommen [Obj07]). In Abbildung 37 ist ein

Screenshot eines typischen mit JFreeChart erstellten Diagrammes zu sehen[8].

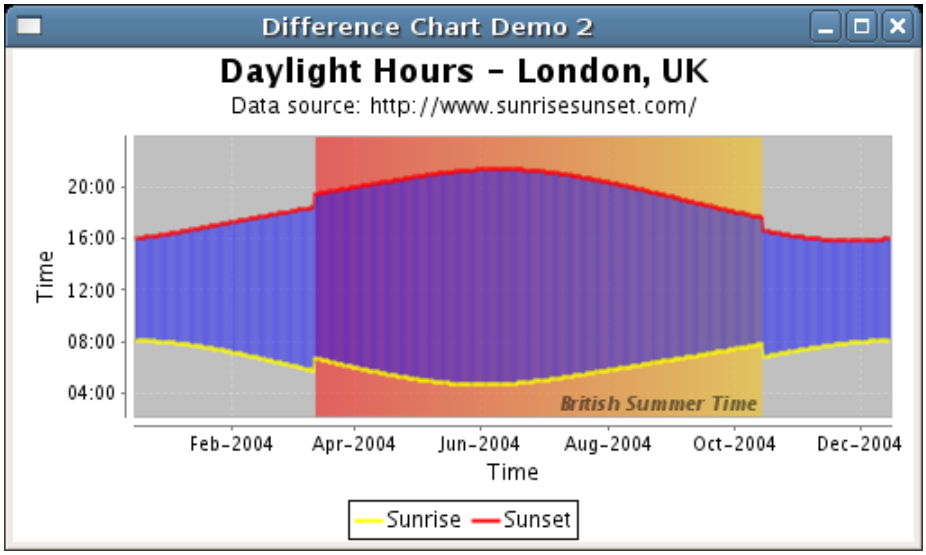

Abbildung 37: JFreeChart Diagramm

5.5.2. Jakarta POI

POI ist ein Subprojekt des Apache Jakarta Projektes und stellt den Versuch dar, das Microsoft OLE 2 Compound Document format für Java zugänglich zu machen. POI steht dabei für *poor obfuscation implementation*, was darauf aufmerksam machen soll, dass das OLE2 Format laut den Entwicklern „einfach schlecht durchdacht“ ist[9], da es im Prinzip ein Archiv mit einer eigenen Dateistruktur darstellt, welche dem DOS-FAT-Dateisystem ähnelt und weiterhin keine Features besitzt. Das erste bekannte Release von POI war im Mai 2005 (siehe [The07a]). Das Projekt ist in verschiedene Teilprojekte untergliedert. Als Basis für alle weiteren Projekte dient das POIFS, welches die Implementierung des o.g. OLE2 Formates darstellt, da dieses ein Containerformat ist, welches die zu parsenden Nutzdaten enthält. Darauf aufsetzend arbeiten dann Subprojekte wie etwa HSSF, welches das Lesen und Schreiben von Excel-Dateien (XLS) ermöglicht. Dieses Projekt ist neben POIFS das am meisten fortgeschrittene, alle anderen befinden sich noch im frühen Entwicklungsstadium: HWPF (Word-Dokumente), HSLF (PowerPoint Präsentationen) und HPSF (Manipulation von Dokumenteneigenschaften).

5.5.3. SwingWT

SwingWT stellt eine Brücke zwischen Swing und SWT dar. Die Entwickler haben es sich zum Ziel gesetzt, Swing-Komponenten zu schaffen, welche intern SWT-Elemente

[8] http://www.jfree.org/jfreechart/images/DifferenceChartDemo2.png,
[9] http://jakarta.apache.org/poi/trans/de/index.html,

nutzen (vgl. [RT07]). Das bedeutet, dass man mit Hilfe dieser Bibliothek eine Swing-Anwendung bauen kann, welche jedoch zur Laufzeit SWT Elemente darstellt. So soll es zum Beispiel ohne großen Aufwand möglich sein, eine bestehende Swing-Anwendung auf SWT umzurüsten, indem einfach alle Gui Elemente mit den Gegenstücken aus der SwingWT Bibliothek ersetzt werden. Im Code werden dann weiterhin (abgeleitete) Swing-Klassen benutzt, welche allerdings intern auf der SWT Bibliothek basieren und native Elemente zeichnen. Ein weiteres Feature ist, dass die zugrunde liegenden SWT Klassen durch ihre Wrapperklassen hindurch direkt ansprechbar sind, sodass sowohl Swing-, als auch SWT-spezifische Funktionen genutzt werden können.

5.5.4. SWTSwing

SWTSwing stellt genau das Gegenteil zu SwingWT dar (vgl. 5.5.3 und [Dec07]). Mit dieser Bibliothek werden Swing Inhalte in SWT Klassen gekapselt, sodass ein Wechsel von SWT zu Swing ohne größere Codemodifikation möglich ist. In Abbildung 38 ist beispielhaft der Torrent-Client Azureus zu sehen, welcher mittels dieser Bibliothek von SWT auf Swing umgerüstet wurde[10].

5.6. Fazit

Als Fazit lässt sich festhalten, dass der Bedarf an Hilfsmitteln zur Erstellung von komplexen Oberflächen in Java durch die zur Verfügung stehenden Frameworks und Tools sehr gut abgedeckt wird. Mit OSGi steht ein einfach zu handhabendes Grundgerüst zur Verfügung, welches durch Eclipse Equinox effizient umgesetzt wird und dem Entwickler mit der Eclipse RCP ein mächtiges Werkzeug zur Erstellung von funktionsreichen und eleganten Benutzeroberflächen liefert. Auch im Spring Framework existiert eine Implementierung der OSGi Spezifikation.
Das Rich Client Konzept stellt durch seine komponentenorientierte Herangehensweise ein einfach zu nutzendes und dennoch beliebig komplexes Modell für anspruchsvolle Oberflächen dar. Durch die native Anbindung an verschiedene Betriebssysteme, bzw. die erfolgreiche Nachahmung nativer Elemente kann Java durchaus als Programmiersprache für Desktopanwendungen betrachtet werden. Zudem wird der Entwicklungsprozess durch die große Auswahl an freien Bibliotheken und Tools stark vereinfacht.
Für die Projektgruppe wäre meiner Meinung nach ein Rich Client empfehlenswert, da er ohne zu großen Aufwand eine sehr anwenderfreundliche Oberfläche ermöglicht. Die Wahl des zu verwendenden Frameworks hängt dabei von Nebenbedingungen ab, die sich erst in weiteren Gesprächen herausstellen werden. Meine persönliche Erfahrung mit der Eclipse RCP ist jedoch durchweg positiv ausgefallen, weswegen ich diese Plattform empfehlen würde.

[10] http://swtswing.sourceforge.net/screenshots/images/AzureusDownloadPeersMetalL&F.png

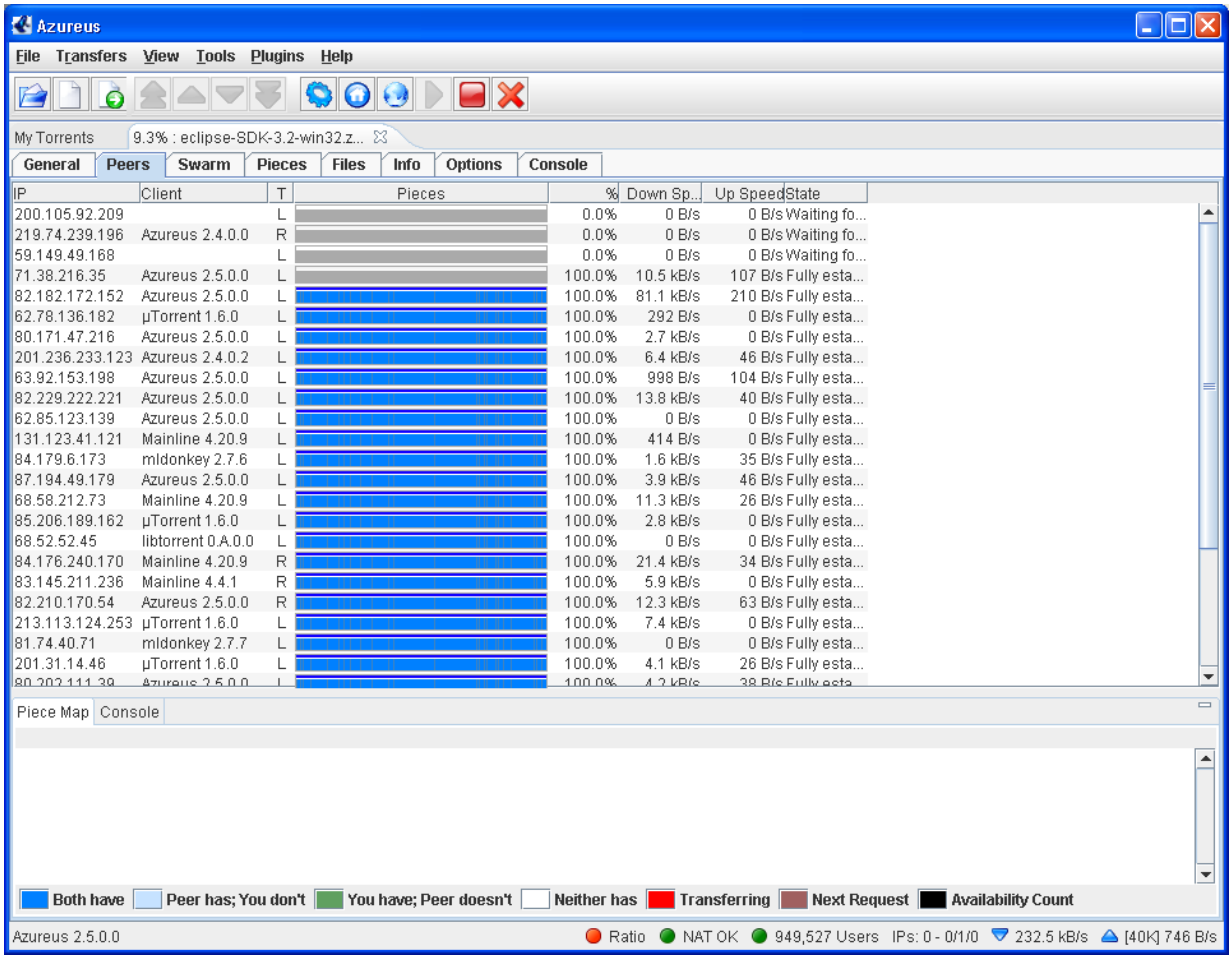

Abbildung 38: SWTSwing Beispiel

Literatur

[Dau05] DAUM, BERTHOLD: *Rich-Client-Entwicklung mit Eclipse 3.1*. dpunkt.verlag, 2005.

[Dec07] DECKERS, CHRISTOPHER: *The SWTSwing project*, 05 2007. http://swtswing.sourceforge.net/main/index.html

[Net07] NETBEANSCOMMUNITY: *platform: Netbeans Module (Plug-in) and Rich Client Application Development*, 05 2007. http://www.netbeans.org/about/press/reviews.html

[Obj07] OBJECTREFINERYLIMITED: *JFreeChart*, 05 2007. http://www.jfree.org/jfreechart/

[OSG07a] OSGIALLIANCE: *OSGi FAQ*, 05 2007. http://www.osgi.org/about/faqs.asp

[OSG07b] OSGiAlliance: *OSGi Members*, 05 2007. http://www.osgi.org/about/member_list.asp

[RT07] Rawson-Tetley, R.: *SwingWT - The Swing/AWT API over SWT library*, 05 2007. http://swingwt.sourceforge.net/

[The07a] TheApacheSoftwareFoundation: *Jakarta POI - Java API To Access Microsoft Format Files*, 05 2007. http://jakarta.apache.org/poi/

[The07b] TheEclipseFoundation: *Equinox*, 05 2007. http://www.eclipse.org/equinox/

[The07c] TheEclipseFoundation: *Rich Client Platform*, 05 2007. http://wiki.eclipse.org/index.php/Rich_Client_Platform

[The07d] TheEclipseFoundation: *SWT: The Standard Widget Toolkit*, 05 2007. http://www.eclipse.org/swt/

[The07e] TheSpringFramework: *Spring richclient*, 05 2007. http://spring-rich-c.sourceforge.net/

[Ull07] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Computing, 2007. http://www.galileocomputing.de/openbook/javainsel6/

6. Hibernate und Spring

Autor: Moritz Eysholdt

6.1. Einleitung

In Zeiten von immer komplexer werdenden Anwendungen und Forderungen nach immer kürzeren Entwicklungszeiten ist es sinnvoll, möglichst viel Wissen und Quellcode aus bereits entwickelten Anwendungen wiederzuverwenden. Ein gutes Framework bietet beides: Für eine spezifische Anwendungsdomäne beinhaltet es den Quellcode zur Lösung von Standardproblemen. Ebenso enthält es Wissen über eine bewährte Architektur bzw. Entwurfsmuster und unterstützt den Entwickler bei der Implementierung dieser Architektur. Des Weiteren hilft ein gutes Framework dabei, das Abstraktionsniveau während der Entwicklung möglichst hoch zu halten und den Entwickler mit weniger, simpleren Zeilen Quellcode schneller zum Ziel zu bringen.
Das Standardproblem der Datenhaltung wurde in der Softwareentwicklung ausreichend mit der Einführung von relationalen Datenbanken gelöst, jedoch war ihre Verwendung durch den Zwang zur Nutzung von SQL umständlich und die Vermischung von SQL-Code und regulärem Quellcode führte insgesamt zu unübersichtlichem und schwer wartbarem Quellcode. Das Entwurfsmuster des Objekt Relationalen Mappings (ORM) verspricht, diese Probleme zu lösen. Das Framework Hibernate hilft dem Entwickler, ORM mit minimalem Aufwand zu nutzen.
Ein weiteres Standardproblem ist die Implementierung der Programmlogik. Für kleine Anwendungen leicht handhabbar, gibt es bei Größeren den Wunsch nach Modularität, Wiederverwendbarkeit, flexibler Konfigurierbarkeit, einfacher Kommunikation mit anderen Komponenten, Logging, Zugriffskontrolle, einfacher Testbarkeit und mehr. Das Spring-Framework adressiert diese Bedürfnisse durch die Einführung des Inversion-of-Control (IoC)-Containers und die Kapselung vieler bereits existierender Frameworks und Bibliotheken.

6.1.1. Abgrenzung

Das Ziel dieser Seminarausarbeitung ist es, die grundsätzliche Funktionsweise von Hibernate und dem Spring-Framework verständlich zu machen. Es ist somit als Einführung in das Thema gedacht. Für detaillierte Beispiele sei auf die jeweilige Originaldokumentation ([Teaa], [Teab]) verwiesen.
Des Weiteren werden Hibernate und das Spring-Framework nur im Kontext einer Standalone-Anwendung betrachtet. Auf Web-spezifische Funktionalitäten wird nicht weiter eingegangen.

6.1.2. Gliederung

Zuerst wird Hibernate und danach das Spring-Framework behandelt, da die Komponente zur Datenhaltung von Spring bei entsprechender Konfiguration auf Hibernate aufbaut. Im Rahmen von Hibernate wird das Konzept des Objekt Relationalen Mappings (ORM) erklärt. Ebenso wird darauf eingegangen, wie sich ohne SQL-Code (Implizites SQL) und mit explizitem SQL auf die Datenbank zugreifen lässt.
Für das Spring-Framework soll die Philosophie, das Konzept des Inversion-of-Control (IoC)-Containers, die Integration von JDBC/ORM sowie die Unterstützung beim Testen angesprochen werden.

6.2. Hibernate

Abbildung 39: Hibernate Logo

Es lässt sich sagen, dass das Standardproblem der Datenhaltung durch die Einführung von relationalen Datenbanken für einen Großteil der Szenarien gelöst wurde. Die für den Zugriff auf die Daten notwendige Sprache SQL wird jedoch trotz Standardisierungsversuchen von jeder Datenbank leicht verschieden interpretiert. Ebenso hat es sich in der Praxis als unkomfortabel erwiesen, für jeden Datenbankzugriff SQL-Code schreiben zu müssen. Die Vermischung von SQL-Code und regulärem Programmcode in derselben Datei führte leicht zu unübersichtlichem Quellcode insgesamt. Das Objekt Relationale Mapping, welches im folgenden Abschnitt erläutert wird, kann diese Probleme lösen.
Hibernate ist als ein Framework zu begreifen, welches es dem Entwickler so einfach wie möglich macht, ORM zu nutzen. Dabei werden eine Vielzahl von Datenbanken und ihre SQL-Dialekte unterstützt. Hibernate bringt eine eigene Abfragesprache (Hibernate Query Language, HQL) mit, welche zur Laufzeit in den jeweils benötigten SQL-Dialekt übersetzt wird. Damit kann der Entwickler seine Anwendung vollständig ohne Abhängigkeit auf ein spezifisches Datenbanksystem implementieren. Um beispielsweise von MySQL auf Oracle umzusteigen, ist lediglich eine Änderung der Hibernate-Konfigurationsdatei notwendig. Der Quellcode inklusive dem HQL-Code der Anwendung muss jedoch nicht angepasst werden. [Teaa]
Hibernate ist open source und wird als Teil des Jboss Enterprise Middleware Systems (JEMS), welches zu RedHat gehört, entwickelt. Es existiert als Framework für Java und als NHibernate auch für .NET. [HBN]

Intern verwendet Hibernate überwiegend JDBC, um sich mit einer Datenbank zu verbinden. Die Verbindungsdaten werden in einer zentralen Konfigurationsdatei verwaltet.

6.2.1. O/R Mapping

Beim Objekt Relationalen Mapping (O/R Mapping, ORM) wird typischerweise jeder Datenbank-Tabelle eine Java-Klasse zugeordnet. Diese Klasse hat für jede Spalte der Tabelle eine Membervariable mit zugehörigen Getter/Setter-Methoden. Eine Instanz der Java-Klasse, also ein Objekt, repräsentiert dann eine Zeile der entsprechenden Datenbanktabelle. Die Aufgabe des ORM-Frameworks ist es dann, bei Bedarf die angeforderten Daten aus der Datenbank zu laden und entsprechende Java-Objekte zu instantiieren und zu initialisieren. Wenn die Daten innerhalb der Java-Objekte geändert oder Neue erstellt wurden, ist es ebenso die Aufgabe des ORM-Frameworks, diese Änderungen ggf. in der Datenbank zu speichern. ORM selbst ist prinzipiell unabhängig von einer konkreten Programmiersprache. Da Hibernate in Java implementiert ist, wird hier direkt von Java-Objekten und Klassen gesprochen.
Um das Mapping zu beschreiben, verwendet Hibernate eine XML-Datei ein oder mehrere Tabellen. Sie definiert beispielsweise, welche Membervariable zu welcher Spalte gehört, ob eine Spalte ein Schlüssel ist und welchen Typ eine Spalte hat. Abbildung 40 veranschaulicht diese Zusammenhänge.

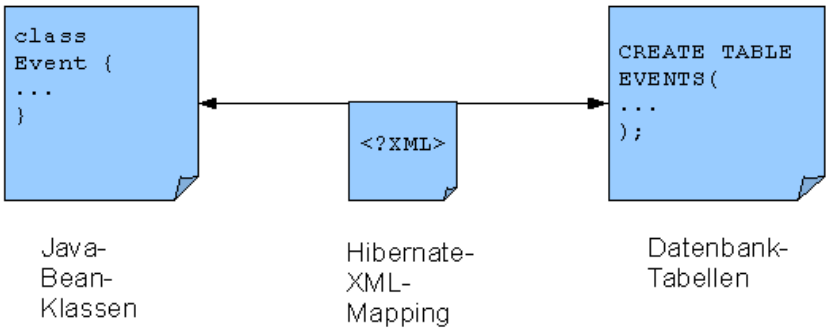

Abbildung 40: Objekt Relationales Mapping (ORM)

Am besten veranschaulichen lassen sich all diese Zusammenhänge mit einem Beispiel. Die Java-Bean in Listing 1 ist die Java-Implementierung des Datenmodells. Es sind Werkzeuge verfügbar, um dieses automatisch aus dem Hibernate-Mapping zu generieren.
Diese Java-Bean wird vom Hibernate-XML-Mapping in Listing 2 auf die Datenbanktabelle *EVENTS* gemappt. Der XML-Tag *id* definiert den Primärschlüssel der Tabelle und *generator class=ïnative¨* gibt an, dass die Generierung von neuen Schlüsselwerten

der Datenbank überlassen werden soll. In SQL-DDL wird dies typischerweise dadurch ausgedrückt, dass die entsprechende Spalte auf *auto_increment* gesetzt wird. Welche Typen zu verwenden sind, kann sich Hibernate in den meisten Fällen aus den Java-Typen aus dem Java-Daten-Modell herleiten. In einigen Fällen jedoch, beispielsweise beim Behandeln von Zeitwerten, stehen in der Datenbank verschiedene Typen zur Verfügung. Hibernate bringt hier sein eigenes Typensystem mit, von dem aus sowohl auf Java-Typen als auch auf SQL-Typen gemapped werden kann. *type=T̈imestamp¨* gibt also einen Hibernate-Type an. In einigen Fällen kann eine Spalte in der Datenbank nicht den gleichen Namen haben wie im Java-Objekt, beispielsweise weil dieser Name in der Datenbank ein reserviertes Schlüsselwort ist. In diesem Falle kann für die Spalte, wie mit *column=ËVENT_DATE¨* geschehen, ein separater Name angegeben werden.
Aus dem XML-Mapping kann Hibernate den notwendigen SQL-DDL-Code vollständig generieren um die Tabelle zu erstellen (das CREATE TABLE Statement).

Listing 1: Java Bean des Java-Daten-Modells

```
1  package de.uni-ol.mypackage;
2
3  import java.util.Date;
4
5  public class Event {
6    private Long id;
7    private String title;
8    private Date date;
9
10   public Event() {}
11   public Long getId() {                return id;          }
12   private void setId(Long id) {        this.id = id;       }
13   public Date getDate() {              return date;        }
14   public void setDate(Date date) {     this.date = date;   }
15   public String getTitle() {           return title;       }
16   public void setTitle(String title) { this.title = title; }
17 }
```

Listing 2: Hibernate-XML-Mapping

```
1  <hibernate-mapping>
2    <class name="de.uni-ol.mypackage.Event" table="EVENTS">
3      <id name="id" column="EVENT\_ID">
4        <generator class="native" />
5      </id>
6      <property name="date" type="timestamp" column="EVENT_DATE" />
7      <property name="title" />
8    </class>
9  </hibernate-mapping>
```

6.2.2. Implizites SQL

Die einfachste Möglichkeit, um auf die Datenbank zuzugreifen ist einfach mit dem Java-Datenmodell zu arbeiten. Das Listing 3 zeigt am Beispiel, wie ein *Event*-Objekt instantiiert und initialisiert wird, um die Werte dann mit *session.save(theEvent)* in der Datenbank zu speichern.
Für sämtliche Interaktion mit der Datenbank wird eine Hibernate-Session benötigt. Die Klasse *HibernateUtil* ist nicht Teil des Hibernate-Frameworks, sondern ein selbst implementiertes Singleton, welches eine Instanz der Hibernate *SessionFactory* beherbergt. Die Methode *getCurrentSession()* gibt die aktuelle Session zurück, wenn eine verfügbar ist oder erstellt eine Neue. Hibernate schließt eine Session automatisch, wenn eine Transaktion beendet wird. Der Aufruf von *session.save()* bringt Hibernate dazu, das Event-Objekt mit folgendem Query in der Datenbank abzulegen: *insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)*. Ebenso können mit *session.get(Class c, Serializable id)* und anderen Methoden Objekte aus der Datenbank geladen werden. Wird eine Getter-Methode aufgerufen und die gemappte Spalte ist ein Fremdschlüssel, so lädt Hibernate automatisch das referenzierte Objekt aus der Datenbank. Anstelle eines bloßen Schlüsselwertes hat der Entwickler sofort das vollständige Objekt zur Verfügung.

Listing 3: Implizites SQL

```
1  private void createAndStoreEvent(String title, Date theDate) {
2    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
3    session.beginTransaction();
4
5    Event theEvent = new Event();
6    theEvent.setTitle(title);
7    theEvent.setDate(theDate);
8
9    session.save(theEvent);
10   session.getTransaction().commit();
11 }
```

6.2.3. Explizites SQL

In einigen Situationen ist es möglicherweise nicht performant genug, sich nur auf implizites SQL zu verlassen. Beispielsweise ist es günstiger, einen *JOIN* zu verwenden, anstatt erst den Inhalt der ersten Tabelle zu laden und dann über die Fremdschlüssel durch die Zweite zu iterieren. Um die Unabhängigkeit von einer einzelnen Datenbankimplementierung zu wahren, bietet Hibernate die Hibernate Query Language (HQL). Sie ist an SQL angelehnt und wird von Hibernate zur Laufzeit in den jeweils benötigten SQL-Dialekt übersetzt. Listing 4 zeigt einige Beispiele, welche durch die Nähe der HQL zur SQL intuitiv verständlich sein sollten. Für eine vollständige Erläuterung sei auf die Originaldokumentation verwiesen. Beachtenswert ist, dass durch die Kenntnis

von Hibernate über die Fremdschlüsselbeziehungen diese nicht explizit in *JOIN*s mit angegeben werden müssen. Neben Abfragen in HQL können mit Hibernate auch Abfragen in klassischem SQL durchgeführt werden, womit jedoch die Unabhängigkeit von einer Datenbankimplementierung verspielt wird.

Listing 4: Explizites SQL

```
List cats = session.createQuery(
  "from Cat as cat where cat.birthdate < ?")
  .setDate(0, date)
  .list();

List mothers = session.createQuery(
  "select mother from Cat as cat join cat.mother as mother where " +
  "cat.name = ?")
  .setString(0, name)
  .list();

List kittens = session.createQuery(
  "from Cat as cat where cat.mother = ?")
  .setEntity(0, pk)
  .list();

Cat mother = (Cat) session.createQuery(
  "select cat.mother from Cat as cat where cat = ?")
  .setEntity(0, izi)
  .uniqueResult();
```

6.3. Spring

Abbildung 41: Spring Logo

Das Spring-Framework wird von seinen Entwicklern selbst als “the leading full-stack Java/JEE application framework” [SPG] bezeichnet. Die Tatsache, dass es bis jetzt über eine Millionen mal heruntergeladen wurde [Joh07], untermauert die Führungsposition. Das Projekt ist seit dem Jahre 2003 open source und wird unter der Schirmherrschaft

von Interface21 [INT] entwickelt. Der Fokus des Spring-Frameworks liegt auf der Verwaltung der Geschäftsobjekte mit dem Ziel, diese möglichst unabhängig voneinander und anderen Komponenten zu halten. Geschäftsobjekte sind die Objekte, welche die Anwendungslogik implementieren. Des Weiteren integriert Spring eine Vielzahl weiterer Technologien, Frameworks und Bibliotheken zur einfachen Verwendung durch den Entwickler. Die Originaldokumentation findet sich unter: [Teab]

6.3.1. Philosophie

Die Entwicklung von Spring wird von folgenden Philosophien geführt [JOH05]. Sie zu kennen ist eine große Hilfe, die Struktur des Framewoks besser zu begreifen.

- Spring ist sowohl umfassend als auch modular: Es besteht aus vielen Komponenten mit konsistenter Architektur, welche aber weitestgehend unabhängig voneinander sind. Damit wird der Entwickler nicht gezwungen, das gesamte Framework zu nutzen, sondern er kann sich die Komponenten auswählen, die er tatsächlich benötigt.
- Es ist nicht das Ziel von Spring, das Rad neu zu erfinden. Eine Komponente wird nur dann neu entwickelt, wenn etwas signifikant besser gemacht werden kann als bei allem bisher Existierenden. Beispiel: Das Spring-MVC-Web-Framework kann Struts ersetzen. Auf der anderen Seite werden existierende Lösungen integriert. Bespiele: Commons Logging, Hibernate, TopLink etc.
- Inversion of Control (IoC): Nicht der eigene Quellcode ruft das Framework auf, sondern das Framework ruft den eigenen Quellcode auf. Details zu diesem Konzept werden in einem der folgenden Kapitel erläutert.
- Das Spring-Framework und darauf aufbauender Quellcode soll einfach zu testen sein. Auch hierzu folgt ein eigenes Kapitel.

6.3.2. Übersicht

Abbildung 42 zeigt den tatsächlichen Umfang des Spring-Frameworks. In diesem Artikel soll lediglich auf Data Access Objects (DAO), Objekt Relationales Mapping (ORM) und Core (der IoC Container) eingegangen werden. Für die Komponenten zur aspektorientierten Programmierung (AOP), der Java Enterprise Edition (JEE) und zur Web-Programmierung sei auf die Originaldokumentation verwiesen.

6.3.3. Inversion of Control

Das Konzept des Inversion of Control (IoC) sieht vor, dass ganz nach dem Hollywood-Prinzip “Don’t call me, I’ll call you” nicht der eigene Quellcode das Framework aufruft, sondern das Framework den eigenen Code. Der Hauptzweck ist, dass der Entwickler POJOs (Plain Old Java Objects) schreiben kann. POJOs erben nicht von komplexen

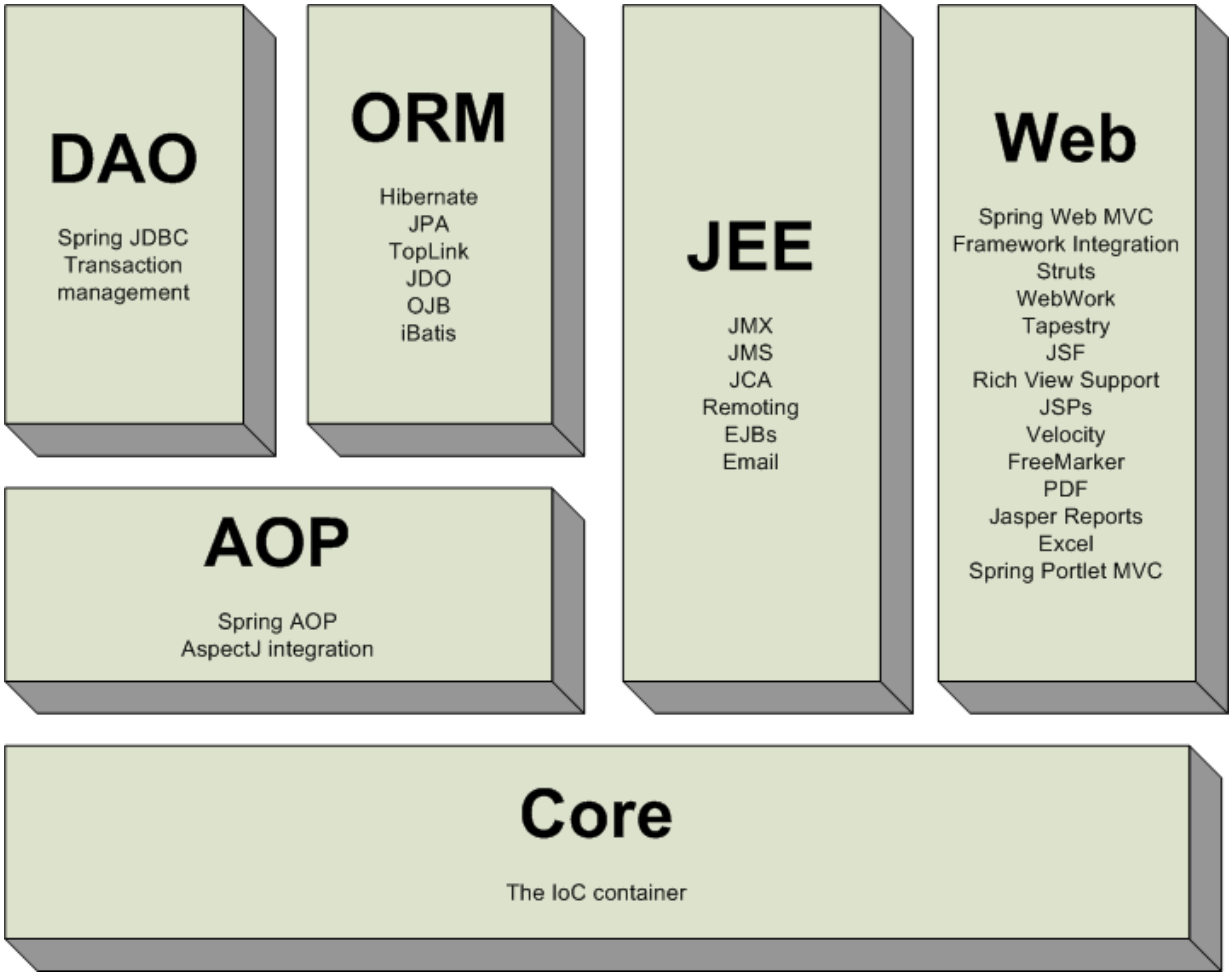

Abbildung 42: Spring-Framework: Übersicht (Quelle: [Teab])

Klassen eines Frameworks oder einer Bibliothek und haben keine direkten externen Abhängigkeiten. Im Falle von Spring werden die externen Abhängigkeiten vom Spring-Framework aufgelöst. Die damit gewonnene Unabhängigkeit bietet einige Vorteile: Die POJOs

- sind einfach zu testen, da die Anforderungen an die Testumgebung minimal sind.
- lassen sich über Spring sehr flexibel kombinieren und konfigurieren.
- lassen sich sehr gut wieder verwenden.

6.3.4. Der IoC Container

Listing 5: Beispielhaftes Geschäftsobjekt für den Spring-IoC-Container

```
1 public class ExampleBusinessObject implements MyBusinessObject {
```

```
2   private ExampleDataAccessObject dao;
3   private int exampleParam;
4
5   public void setDataAccessObject(ExampleDataAccessObject dao) {
6     this.dao = dao;
7   }
8
9   public void setExampleParam(int exampleParam) {
10    this.exampleParam = exampleParam;
11  }
12
13  public void myBusinessMethod() {
14    // do stuff using dao
15  }
16 }
```

Der IoC Container des Spring-Frameworks ist seine zentrale Komponente. Da er die POJOs instantiiert, müssen diese dem Java-Bean-Standard entsprechen, weshalb die POJOs im Folgenden öfters als Java-Beans bezeichnet werden. Der Java-Bean-Standard schreibt vor, dass eine Java-Klasse einen öffentlichen parameterlosen Konstruktor haben, serialisierbar sein und via Getter/Setter-Methoden konfigurierbar sein muss, um sich als Bean zu bezeichnen. Im Kontext von Spring wird eine Bean, die einen Teil der Anwendungslogik implementiert, als Geschäftsobjekt (Business Object) bezeichnet. Das Listing 5 zeigt beispielhaft ein solches Geschäftsobjekt.

Listing 6: Konfiguration für den Spring-IoC-Container

```
1  <beans>
2    <bean id="myDataSource"
3          class="org.apache.commons.dbcp.BasicDataSource"
4          destroy-method="close">
5      <property name="driverClassName" value="com.mysql.jdbc.Driver" />
6      <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
7      <property name="username" value="someone" />
8    </bean>
9    <bean id="exampleDataAccessObject"
10         class="example.ExampleDataAccessObject">
11     <property name="dataSource" ref="myDataSource" />
12   </bean>
13   <bean id="exampleBusinessObject"
14         class="example.ExampleBusinessObject">
15     <property name="dataAccessObject">
16       <ref bean="exampleDataAccessObject" />
17     </property>
18     <property name="exampleParam">
19       <value>10</value>
```

```
20      </property>
21    </bean>
22  </beans>
```

Instantiiert werden die Java-Beans von Springs *BeanFactory*, welche die Beans anhand einer zentralen Konfiguration kombiniert und konfiguriert. Typischerweise wird die *XMLBeanFactory* verwendet, welche, ihre Konfiguration aus einer XML-Datei bezieht, wie der Name erahnen lässt. Das Listing 6 zeigt eine solche XML-Konfiguration. Das Kombinieren (Dependency-Injection) und Konfigurieren der Beans vollzieht Spring, indem es Setter-Methoden der Objekte aufruft oder Werte per Konstruktor-Parameter übergibt. Die Konfiguration in Listing 6 beschreibt die drei Beans mit den Namen *exampleBusinessObject*, *exampleDataAccessObject* und *myDataSource*. Die drei Beans laufen, da nicht anders definiert, im Application-Scope. Klassen im Application-Scope werden genau einmal instantiiert pro Anwendungsinstanz. Damit verhalten sie sich genau so als würden sie das Singeleton-Entwurfsmuster implementieren. Mögliche andere Scopes sind Session oder Prototype. Letztere werden bei jedem Aufruf neu instantiiert. Wenn die BeanFactory das *exampleBusinessObject* instantiiert, wird die Methode *setExampleParam()* mit dem Wert 10 und die Methode *setDataAccessObject()* mit der Instanz des *exampleDataAccessObjects* aufgerufen. Auffällig ist, dass die Bean *myDataSource* nicht Teil des eigenen Quellcodes ist, sondern aus dem Apache-Projekt stammt. Hier zeigt sich die freie Kombinierbarkeit der Beans. Kompatibilität wird über Java-Interfaces sichergestellt.

6.3.5. Datenbankanbindung

Spring stellt verschiedene Möglichkeiten der Datenbankanbindung bereit: Es kann als Wrapper für JDBC oder für ein ORM-Framework dienen. Für beide Varianten übersetzt es die datenbankspezifischen Fehlercodes in datenbankunabhängige Exceptions aus einer Spring-eigenen Exception-Hierarchie. Dem Entwickler muss somit nicht mehr im Datenbankhandbuch die Fehlercodes heraussuchen, wenn beispielsweise ein Unique-Constraint verletzt wird und der Anwender darüber benachrichtigt werden soll, dass sein gewählter Benutzername bereits vergeben ist.

6.3.6. Wrapper für JDBC

Listing 7: Absetzens eines Querys mit dem JDBC-Wrapper von Spring

```
1 int i = simpleJdbcTemplate.queryForInt("SELECT␣COUNT(0)␣" +
2     "FROM␣T_CLIENT␣WHERE␣TYPE=?␣AND␣CURRENCY=?", 13, "GBP");
```

Wie Listing 7 zeigt, verhilft Springs Wrapper für JDBC hauptsächlich zu deutlich kompakterem Quellcode. Das Iterieren über *ResultSet*s entfällt und dank Java 5 können auf einfache Weise beliebig viele Parameter übergeben werden.

6.3.7. Wrapper für O/R Mapping

Spring kann eine Vielzahl von ORM-Frameworks wrappen. Zu ihnen zählen Hibernate, TopLink, JDO und andere. Der Quellcode der eigenen Anwendung wird dabei vom ORM-Framework entkoppelt. Spring kann dabei die Resourcen-Verwaltung übernehmen, beispielsweise kann es sich um Connectionpools kümmern. Des Weiteren kann es die Sessions und Transaktionen verwalten und zum Beispiel automatisch eine gültige Session bereitstellen, wenn eine Methode eines Geschäftsobjekts aufgerufen wird. Ebenso kann die Transaktion mit einem Rollback beendet werden, wenn die Methode eine Exception wirft.

6.3.8. Testen

Spring unterstützt das Testen mit JUnit. Wie bereits erwähnt wird beim Entwickeln des Spring-Frameworks großen Wert darauf gelegt, dass sowohl das Framework selbst, als auch der das Framework verwendende Quellcode einfach testbar ist. Sehr vorteilhaft für das Testen von Geschäftsobjekten ist das Konzept der IoC, da somit die Laufzeitumgebung von einzelnen Geschäftsobjekten minimal ist und flexibel konfiguriert werden kann. Das Paket *org.springframework.test* hilft dabei,

- die bereits vorhandene XML-Konfiguration zum Instantiieren von Geschäftsobjekten innerhalb von JUnit-Test zu verwenden.
- Tests in Transaktionen zu kapseln und den Ausgangszustand der Datenbank nach dem Test wieder herzustellen (rollback).

6.3.9. Beispielarchitektur

Im Folgenden wird beispielhaft eine Architektur für eine Anwendung beschrieben. Die Anwendung kann dabei sowohl eine Standalone-, als auch eine Webanwendung sein. Der Fokus dieser Architektur liegt auf einer sauberen Trennung zwischen Persistenz-, Logik- und GUI-Ebene sowie einer optimalen Verwendung des Spring-Frameworks.

6.3.10. Anforderungen

Im Interesse des guten Software-Engineering sollen folgende Anforderungen berücksichtigt werden:

- Die Anwendungslogik muss unabhängig von der Benutzungsschnittstelle sein. Dabei spielt es keine Rolle, ob die Benutzungsschnittstelle mit einem Widget-Toolkit, einem Webinterface oder einer Konsole realisiert wird.
- Die Anwendungslogik soll der Benutzungsschnittstelle ein klar definiertes Interface zur Verfügung stellen. Alle über dieses Interface ausgetauschten Objekte dürfen keine Abhängigkeiten auf die Anwendungslogik haben und müssen serialisierbar sein, sodass sie sich auch über ein Netzwerk übertragen lassen.

- Die Anwendungslogik soll nicht an eine bestimmte Datenbankimplementierung gebunden sein. Es sollen beispielsweise keine Änderungen am Quellcode notwendig sein, um von MySQL auf Oracle umzusteigen.
- Quellcode der Anwendungslogik und Quellcode der Datenbankabfrage soll sauber voneinander getrennt sein.

6.3.11. Definitionen

Folgende weitere Fachbegriffe, die bisher noch nicht erklärt wurden, finden in der Beschreibung der Architektur Verwendung:

- Eine Entität ist eine Datenbankentität, so wie sie für Entity-Relationship-Diagramme definiert ist.
- Ein Data Access Object (DAO) ist ein Singelton, welches einer Datenbankentität zugeordnet ist und alle Datenbankzugriffe kapselt. Jede Entity-Klasse aus dem Java-Datenmodell des ORM hat somit ihr eigens DAO.
- Ein Value Object (VO) ist eine Java-Bean, welche keine Methoden außer Getter und Setter besitzt und keine Abhängigkeiten auf andere Java-Klassen außer anderen VOs hat. VOs lassen sich serialisieren und dienen als Container für den Datenaustausch.
- Ein Service ist ein Geschäftsobjekt, welches eine Fassade für die Anwendungslogik oder einen Teil von ihr bildet und dessen öffentlich verfügbare Methoden ausschließlich VOs und primitive Datentypen als Parameter und Rückgabewerte haben.

6.3.12. Die Architektur

Die Abbildung 43 veranschaulicht die Beispielarchitektur, welche allen genannten Anforderungen gerecht wird. Die Grafik ist vertikal unterteilt in Benutzungsschnittstelle (User Interface, UI), Logik und Persistenz und unterscheidet auf der horizontalen Ebene zwischen selbst geschriebenem und fremdem Quellcode. Jeder Pfeil steht für eine Abhängigkeit.
In der Benutzungsschnittstelle wird beispielhaft davon ausgegangen, dass ein Graphical User Interface (GUI) zusammen mit einem GUI-Toolkit verwendet wird. Eine andere UI-Lösung (Webinterface, Konsole) würde jedoch ebenso gut mit der Anwendungslogik zusammenarbeiten können. Die Services bilden die Fassade für die Anwendungslogik, was dazu führt, dass die GUI ausschließlich Abhängigkeiten auf die Services und VOs hat. Die Geschäftsobjekte, welche aufgrund der Services vor der GUI verborgen bleiben, implementieren die eigentliche Anwendungslogik. Sie greifen auf die vom ORM-Framework bereitgestellten Entity-Objekte und die zugehörigen DAOs zu. Die Ergebnisse der Berechnungen der Geschäftsobjekte werden entweder wieder in der

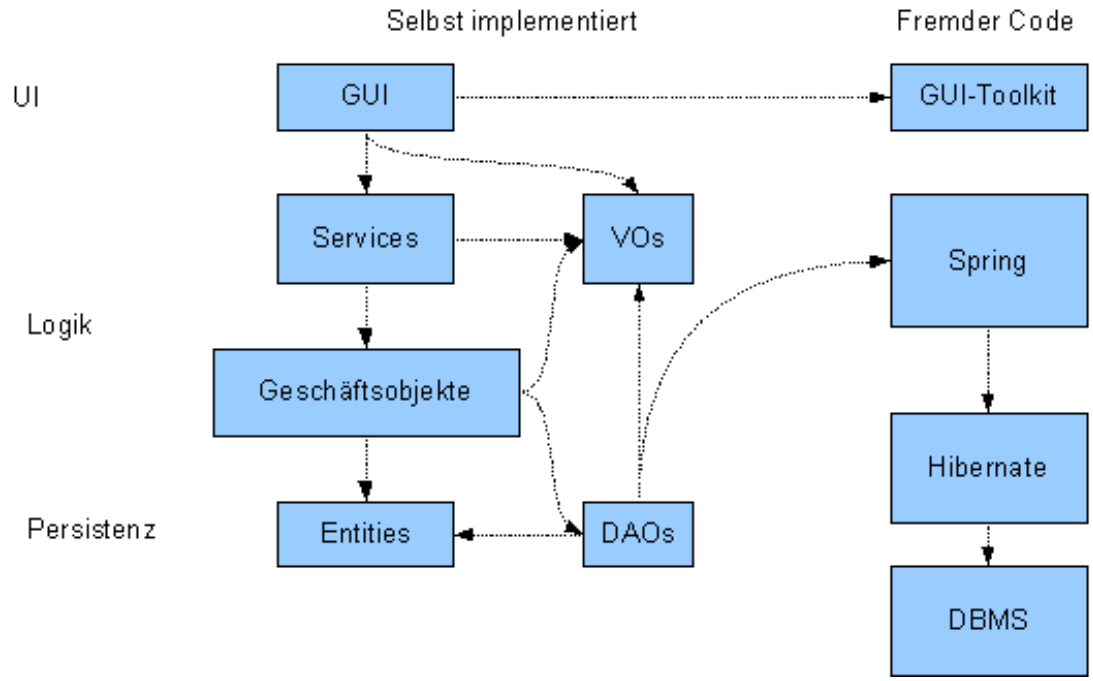

Abbildung 43: Beispielarchitektur

Datenbank gespeichert oder in Form von VOs von den Services genutzt. Die DAOs tätigen die Datenbankabfragen, welche trotz des ORM noch nötig sind und übernehmen die Konvertierung zwischen VOs und Entity-Objekten. Aufgrund des IoC-Konzeptes haben weder die Services und Geschäftsobjekte, noch die Entity-Objekte Abhängigkeiten auf das Spring-Framework. Da Spring als Wrapper für Hibernate fungiert, haben auch die DAOs keine direkten Abhängigkeiten auf Hibernate. Dank der Abstraktion von Spring und Hibernate bleiben die DAOs vollständig unabhängig von einer spezifischen Datenbankimplementierung.

6.4. Fazit

An dieser Stelle soll eine kurze Einschätzung zu Hibernate, Spring und der vorgestellten Beispielarchitektur gegeben werden.

6.4.1. Hibernate

Nach einer kurzen Eingewöhnungszeit beschert Hibernate dank dem ORM dem Entwickler das Erlebnis, nicht mehr explizit mit der Datenbank zu kommunizieren, sondern die Daten direkt in der Anwendung verfügbar zu haben. Die Abstraktionsebene und der Entwicklungskomfort werden erhöht sowie die Verständlichkeit des Quellcodes verbessert. Dem steht das Risiko gegenüber, dass der Entwickler den Blick für tat-

sächlich getätigte Datenbankabfragen verliert und sei Quellcode unnötig viele davon produziert.

6.4.2. Spring

Der IoC-Container ermöglicht eine phantastische Einfachheit und Autonomie der einzelnen Geschäftsobjekte der Anwendungslogik. Diese wirkt fehlervorbeugend während der Entwicklung, vereinfacht das Testen und erhöht die Wiederverwendbarkeit des Quellcodes. Darüber hinaus kapselt Spring eine Vielzahl von Fremdframeworks und -bibliotheken, so dass der Entwickler diese nahtlos in seine Spring-Anwendung integrieren kann. Dem steht der hohe Einarbeitungsaufwand gegenüber, welcher sich auf dem Umfang und der Komplexität des Frameworks begründet.

6.4.3. Beispielarchitektur

Die Vorteile dieser Architektur teilen sich wie folgt auf:

- Planbarkeit: Durch ihre klare Struktur gibt es einen direkten Top-Down-Prozess von Anwendungsfällen (Anforderungsdokument) zu einem Objektorientierten-/Serviceorientierten Design (Designdokument) und weiter zu den konkreten Geschäftsobjekten, Entities und Services.
- Skalierbarkeit: Die Architektur funktioniert unabhängig davon, ob es 5, 50 oder 500 oder mehr Geschäftsobjekte gibt. Erst wenn die Leistungsfähigkeit der Hardware nicht mehr ausreicht, muss ein verteiltes System in Betracht gezogen werden.
- Beherrschbarkeit: Dank ihrer klaren Struktur und dem generell hohen Abstraktionslevel ist die Architektur leicht beherrschbar.

Als Nachteil ist die hohe Einarbeitungszeit zu nennen: Der Umfang von Spring und Hibernate ist immens und auch das Konzept von Geschäftsobjekten, Services, ORM, VOs, DAOs etc. bedarf einer gewissen Gewöhnung.
Ob die vorgeschlagene Architektur sinnvoll ist, hängt stark von der Größe des geplanten Projektes ab. Wenn das Projekt eher klein (weniger als 15 Entitäten + Geschäftsobjekte) ist oder einen prototypischen Charakter hat, mag die Architektur überdimensioniert sein.
Die Architektur wird sehr gut von Werkzeugen der modellgetriebenen Softwareentwicklung unterstützt: Mit AndroMDA ist es möglich, aus einem UML-Diagramm die sämtliche VOs, DAOs, Entity-Objekte, Hibernate-mappings und Spring-Konfiguration zu generieren. Ebenso können Services und Geschäftsobjekte mit den Stümpfen der zu implementierenden Methoden generiert werden. Das automatische Generieren des Quellcodes hilft ebenso, die Einarbeitungszeit niedrig zu halten, da der Codegenerator dem Entwickler viele Implementierungsfragen abnimmt.

Literatur

[HBN] *Hibernate Website*. http://www.hibernate.org/

[INT] *Interface21 Webseite*. http://www.interface21.com/

[Joh07] JOHNSON, ROD: *Spring 2.0: What's New and Why it Matters, InfoQ*, 01 2007. http://www.infoq.com/articles/spring-2-intro

[SPG] *Spring Framework Webseite*. http://www.springframework.org/

[Teaa] TEAM, HIBERNATE DEVELOPER: *Hibernate Reference Documentation Version 3.2.2*. http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf

[Teab] TEAM, SPRING DEVELOPER: *The Spring Framework - Reference Documentation Version 2.0.5*. http://static.springframework.org/spring/docs/2.0.x/reference/index.html

7. Groupware

Autor: Christian Bley

7.1. Einleitung

In der heutigen Softwareentwicklung geht der Trend immer mehr in Richtung ingenieurmäßige Entwicklung. Das bedeutet weniger Alleinentwicklung und mehr Teamarbeit und somit mehr Kommunikation und Zusammenarbeit zwischen den Entwicklern. Hier ist eine unterstützende Software sinnvoll, deren Funktionsumfang hilft, solche Gruppenorientierte Projekte besser planen und durchführen zu können.
Ich werde in diesem Seminar zunächst die Eigenschaften und Anforderungen von solcher Software in Abschnitt 7.2 darstellen. Dann werde ich zwei verbreitete Vertreter von Groupware vorstellen und auf ihre Eignung hin untersuchen, ob sie den in Abschnitt 7.2.1 genannten Anforderungen entsprechen.
Der Schwerpunkt dieses Seminars liegt auf Groupwareanwendungen, die sich speziell für die Software-Entwicklung eignen, wobei ich hier auf die Software Trac im Abschnitt 7.4 als typischen Vertreter eingehen werde. In Abschnitt 7.4.4 wird auf die Verträglichkeit mit verbreiteten Vorgehensmodellen zur Softwareentwicklung geprüft, wobei dann am Schluss ein Fazit zur Nutzbarkeit der Groupware steht.

7.2. Groupware

Der Name *Groupware* leitet sich aus den Wörtern *Group* und *Software* ab. Wie Wulf in [Wul97] erklärt, steht der Begriff für Anwendungen deren Aufgabe die Unterstützung von Kommunikation, Kooperation und Koordination bei gruppenorientierten Arbeiten ist. Hierbei kann es sich um Hardware oder Software handeln. In der Forschung wird der Bereich auch als *Computer Supported Cooperative Work*, also Computer unterstützte Zusammenarbeit bezeichnet.

7.2.1. Anforderungen an Groupware

Wie Roth in [Rot00] darlegt, deckt eine Groupware in der Regel einen oder mehrere der drei Funktionsbereiche Kommunikation, Koordination und Kooperation ab.
Für die Kommunikation ist es notwendig, Nachrichten austauschen zu können, beispielsweise über einen Email-Verteiler oder ein Forum. Dies bietet die Möglichkeit, den Fokus der Gruppe schnell auf ein Problem oder eine Aufgabe umzulenken und so eine schnelle Behebung des Fehlers sicherzustellen. Auf diese Weise wird auch die Kooperation des Teams gefördert.
Damit alle Mitarbeiter möglichst effektiv zusammenarbeiten können, ist es auch wichtig, den Wissensstand möglichst anzugleichen. Es ist meist weniger effektiv einige Spezialisten zu haben. Wenn zu jedem Projektzweig mehrere Entwickler mit grundlegendem Wissen zur Verfügung stehen, bietet dies Sicherheit bei Ausfall eines Mitarbeiters.

So sollte auch die Speicherung und Verbreitung von Wissen eine grundlegende Eigenschaft einer Groupwareanwendung sein. Dies ist gut über Wikis, auch über Foren oder gemeinsame Dokumentenverwaltung zu erreichen.
Um den Fortschritt eines Projekts gut einschätzen zu können und so den weiteren Weg zu planen, ist ein Feedback für die Projektleitung sehr wichtig. Sie muss erkennen können, wie die einzelnen Komponenten vorankommen und wo es Engpässe gibt. Hier greifen wieder gute Kommunikation und Elemente, wie ein Memo- oder Berichtssystem, mit dem der Fortschritt, beziehungsweise Blockaden erkannt werden können. Auch sollten Mechanismen bestehen, mit denen Ressourcen und Mitarbeiter dirigiert werden können um solche Probleme zu lösen oder zu verhindern.

7.2.2. Architekturen

Für alle Anforderungen existieren bereits Lösungen, meistens einzeln oder in Aggregation einiger Funktionen. Um eine effektive Nutzbarkeit zu erzielen, sollten möglichst viele Anforderungen in einer Anwendung umgesetzt werden.
Sehr verbreitet ist das Exchangesystem. Hier handelt es sich um eine Client-Server Lösung, deren Clients meist auf bereits bestehenden Personal Information Managern basieren. Dies ist Software, die persönliche Termine oder Kontakte verwaltet. Die Serverseite stellt der Exchangeserver. Dieser besteht aus einer Datenbank in der alle Daten des Benutzers hinterlegt und mit anderen Daten verknüpft werden. Meistens sind diese Server noch mit weiteren Modulen, wie Mailserver ausgestattet.
Bei Clients gibt es zwei gängige Architekturen. Häufig handelt es sich um Fatclients die auch losgelöst vom Exchangeserver funktionieren, wie die bereits genannten Personal Information Manager. Einige Systeme bieten auch Kompatibilität für unterschiedliche Clients an. Die zweite Architektur ist der Zugriff über einen Browser durch ein Web-Frontend. Dies ist bei einigen Lösungen möglich und bei Open-Xchange, das in Abschnitt 7.3 vorgestellt wird sogar der Standard. Einige weitere Lösungen habe ich in Tabelle 3 zusammengefasst.
Auch existieren Erweiterungen für bestimmte Personal Information Manager, um auf eine andere Exchangelösung zugreifen zu können. So gibt es zum Beispiel für Microsoft Outlook Konnektoren für Opengroupware.org oder Kolab.

Exchangeserver	Nativer Client
Microsoft Exchange	Microsoft Outlook
IBM Lotus Domino	Lotus Notes
Open-Xchange	Webclient
Opengroupware.org	Webclient
Kolab	ohne eigene Clientlösung

Tabelle 3: Exchange Lösungen

Ein einfacherer Ansatz ist ein einziger Server, der die Groupware-Funktionen übernimmt und gleichzeitig auch den Webclient stellt. Ein möglicher Zugang mit anderen

Clients erfolgt hier über Webservices. Implementiert wird solche Groupware meist als Programmzusatz zu Webservern, die auf eine separate Datenbank im Hintergrund zugreifen um ihre Daten zu verwalten. Diese Systeme sind kompakter und können auch schneller Arbeiten. Da sie aber meistens für bestimmte Nutzungsbereiche konzipiert werden, sind sie dafür eingeschränkter in den Funktionen.

7.2.3. Softwareentwicklung

Speziell für Softwareentwicklung im Opensource-Bereich gelten zwar auch dieselben Anforderungen, wie in Abschnitt 7.2.1 dargestellt, jedoch hat man hier noch engere Bedingungen.
Da ein Informationsaustausch zwischen Entwicklern häufig über das Internet statt findet, stehen hier Webbasierte Anwendungen im Vordergrund, da diese eine vom Client unabhängige Nutzung ermöglichen.
Der Quellcode wird in öffentlich zugänglichen Repositories hinterlegt, weshalb Änderungen oder Fehler übersichtlich angezeigt werden müssen.
Da das Projekt andauernden Änderungen unterworfen ist, muss die Dokumentation öffentlich und auch leicht anpassbar sein. Auch eine Verknüpfung mit automatischen Dokumentationsprogrammen, wie Doxygen wäre hier hilfreich.
Das Projekt sollte in mehrere Teilkomponenten zerlegt werden können, für die jeweils getrennt der Fortschritt geplant und überwacht werden kann. Gruppenverwaltung und Zuordnung zu den Teilaufgaben sind aus Sicht der Planung unverzichtbar.
Zusätzlich sollte die Groupware es auch ermöglichen, eine leichtere Wartbarkeit der Software zu erreichen. Bei Bugtrackingtools wie Bugzilla ist dies eine der Hauptfunktionen.
Wenn Fehler entdeckt und gemeldet werden, sollten diese Meldungen gleich als Arbeitsauftrag weiter genutzt werden können. So ist ein schnelles Beheben des Fehlers möglich. Über das selbe System können dann auch Erweiterungswünsche mitgeteilt werden.
Testumgebungen und Testberichte sind in modernen Entwicklungen sehr wichtig und eine Integration von solchen Frameworks ist hilfreich für die Qualitätssicherung.
Diese Anforderungen sind nicht die einzigen und je nach Projektziel gibt es unterschiedliche Prioritäten. Dennoch sollte eine Groupwareanwendung diese Anforderungen umsetzen, um ein breites Anwendungsspektrum zu erreichen.

7.3. Open-Xchange

Eine Lösung für Groupwareanwendungen ist Open-Xchange. Entwickelt wird Open-Xchange als Opensourceprojekt. Den Server gibt es als freie Community-Edition in der Opensourceversion oder auch vermarktet als Open-Xchange Server 5 mit zusätzlichen Diensten unter [Ope].
Basis ist der Exchangeserver, Zugriff auf die Funktionen bietet ein Web-Frontend. Per Zusatzsoftware, des sogenannten XTenders ist es möglich den Server mit MS Outlook,

Palm OS, Samba oder SyncML, ein Standard zur Synchronisation mobiler Geräte, zu kombinieren.

7.3.1. Funktionsumfang

Bereich	- Funktionen
Kommunikation	- Webmail
	- Kontaktdatenbank
	- Forum
Kooperation	- Dokumentenpool
	- Wissensdatenbank
	- Gruppenfunktionen
Koordination	- Projektmanagement
	- Kalender
	- Aufgaben
	- Notizen / Bookmarks

Tabelle 4: Funktionsumfang von Open-Xchange

Eine kurze Übersicht über die vorhanden Funktionen von Open-Xchange zeigt Tabelle 7.3.1.
Open-Xchange besteht eigentlich aus zwei Weboberflächen, den Groupware- und den Webmailteil. Der Webmailclient ist aus dem Groupwareteil, wie der Kontaktliste aufrufbar, kann aber auch losgelöst benutzt werden. So hat man auch von fremden Rechnern Zugriff auf eigene Emails. Neben einer globalen Adressliste können auch persönliche Kontakte für andere Personen freigegeben werden.
Die Kalenderfunktion ist stark in andere Komponenten integriert. Termine können persönlich oder für eine Gruppe bestimmt sein. Kalender sind frei durchsuchbar und können freigegeben werden. Eine Benachrichtigung per Email kann bei allen Terminen eingerichtet werden.
Im Aufgabenplaner werden alle hier erstellten oder von anderen Stellen übernommenen Aufgaben aufgelistet. Eine Aufgabe kann auch mehreren Personen oder Gruppen zugewiesen werden. Man kann sie auch als Serie anlegen, damit sie regelmäßig wiederholt wird. Es ist möglich einen Zeitplan, sowie eine Reise- und Kostenplanung zu einzelnen Aufgaben zu erstellen.
Bei Anlegen eines Projektes sind Angaben wie Start-, End- und Abgabedatum, Typ, Leiter und Auftraggeber vorgesehen. Es kann ein Budget und der erwartete Arbeitsaufwand gespeichert werden. Durch zugeordnete Aufgaben werden die Budget und Zeitangaben stets aktualisiert. Meilensteine ermöglichen eine Ordnung und Planung des Projektes. Sie dienen jedoch nur als Anhaltspunkt für die Zeitplanung und zur Beschreibung von Teilzielen, es können keine Aufgaben Meilensteinen zugeordnet werden. Sämtliche Aufgaben und Meilensteine werden in einem Projektdiagramm übersichtlich dargestellt.

Über die Dokumentenfunktion können eigene Dokumente hinterlegt werden. Es sind private, globale oder individuelle Ordner möglich. Für die letzteren können die Zugriffsrechte speziell zugeschnitten werden. Es existiert auch eine Versionskontrolle für die Dateien, die jedoch Unterschiede nur anhand der Beschreibung darstellt.
Ähnlich funktioniert die Wissensdatenbank, wobei man hier keine Dateien hochlädt, sondern Wissenseinträge erstellt. Auch hier gibt es eine Ordnerstruktur mit dem gleichen Rechtesystem für Schreib- und Leserechte. Alle Einträge können mit anderen verknüpft werden und es besteht die Möglichkeit eigene Notizen zu den Einträgen zu erstellen.
Als weiteres Kommunikationsinstrument dient das Forum, welches wie herkömmliche Foren im Internet aufgebaut ist. Außerdem bietet es die Möglichkeit, Beiträge oder Themen zu verknüpfen, Lese- und Schreibrechte zu erteilen und bei Antworten per Email zu benachrichtigen.

7.3.2. Nutzen

Open-Xchange erfüllt alle unter Abschnitt 7.2.1 aufgeführten Anforderungen und bietet gleichzeitig die Funktionen eines Exchangesystems auf Basis eines frei verfügbaren Opensource Servers. Somit wäre dieses System auch für freie Softwareprojekte eine Option.
Dem Nutzer ermöglicht die Suite eine komfortable Ordnung und Planung der persönlichen Arbeit, durch den Webzugriff auch plattformunabhängig. Die Möglichkeit Kalender, Dokumente und Wissen mit anderen teilen zu können ist der Zusammenarbeit sehr förderlich. Auch das Projektmanagement kann hier einfach durchgeführt werden, da einzelne Abschnitte über die Meilensteine gut darstellbar sind. Die Aufgabenverteilung lässt aus Sicht der Projektverwaltung ebenfalls wenig zu bemängeln.
Für die Nutzung im Umfeld einer Firma, in der mehrere Projekte verwaltet werden müssen, ist dies sicher eine gute Lösung. Im Falle einer Softwareentwicklung, deren spezifische Anforderungen in Abschnitt 7.2.3 beschrieben wurden, gibt es aber gewisse Defizite.
Die Tatsache, dass nur angemeldete Nutzer Zugriff auf Dokumente haben können, schränkt die Fähigkeit stark ein, Dokumentationen frei zu verteilen. Das Erstellen eines Fehlerberichtes ist für Außenstehende ebenfalls nicht möglich. Wichtig erscheint hier auch der Mangel, auf den Quellcode in einem Repository zugreifen zu können. Die Teammitglieder können sich zwar über ein Forum austauschen, Dokumente und Wissen teilen. Dies ist in dieser Form aber sehr umständlich, da auf neue Einträge nicht automatisch hingewiesen wird. Auch sind viele Funktionen zu umständlich implementiert oder auch einfach überflüssig. Für den Zweck der Softwareentwicklung erscheint dieses System als nicht optimal.

7.4. Trac

Für den Einsatz in der Opensourceentwicklung haben sich schmalere Projektumgebungen bewährt, wie Bugzilla oder große Projektsammlungen wie SourceForge.net, die ihren Teilnehmern Dienste wie Quellcode-Version-Kontrolle, Kommunikationsunterstützung oder Projektverwaltung anbieten. Ein solch schmales, aber umfangreiches Werkzeug ist Trac.
Trac kombiniert die Bugtrackingeigenschaften von Tools wie Bugzilla mit Groupwareelementen wie Wiki, Roadmap und der Möglichkeit Änderungen am Quellcode direkt aus einem Repository heraus einzusehen. Die Handhabung ist dabei intuitiv und einfach zu lernen. Implementiert ist Trac als Opensource in Python und ist frei verfügbar.

7.4.1. Komponenten

Es werden hier die einzelnen Komponenten von Trac beschrieben, wobei ich die Einzelheiten aus der offiziellen Dokumentation [Edg07b] entnommen habe.
Trac kann an einen Webserver angebunden oder auch als Standalone-Server genutzt werden. Benötigt wird eine SQLite, PostgreSQL oder MySQL-Datenbank. Die Konfiguration Tracs geschieht über die zentrale `trac.ini`-Datei oder über ein Admintool von der Kommandozeile aus. Die Darstellung Tracs basiert auf einem HTML-Templatesystem, dessen Erscheinung frei angepasst werden kann. So kann Trac nicht nur als Verwaltung sondern auch als Präsentation des Projekts genutzt werden.
Trac unterstützt Nutzer- und Gruppenmanagement. Es können auch anonyme Aktionen durchgeführt werden, solange die Rechte hierfür freigegeben worden sind.
Trac ist durch Plugins leicht erweiterbar. Diese können unterschiedliche Funktionen von vereinfachter Konfiguration über den Webbrowser, bis zur veränderten Nutzeranmeldung implementieren. Auch wenn ein Großteil des möglichen Funktionsumfangs von Trac durch Plugins entsteht, gehe ich hier hauptsächlich auf die Grundfunktionen von Trac ein.
Ein wichtiges Element ist das TracWiki. Es entspricht von der Handhabung anderen gängigen Wikis und ist in viele andere Elemente von Trac eingebunden. Die Nutzung ist denkbar einfach. Es gibt zahlreiche Formatierungsmöglichkeiten wie Tabellen, Schriftformen, Listen oder Überschriften. Außerdem gibt es weitere Makros um Ausgaben zu erleichtern oder Eingaben weiterzuverarbeiten. Letzteres sind sogenannte WikiProzessoren, die unter anderem zum Beispiel Syntax-Hervorhebung bei Codeblöcken oder verlinkten Quelldateien ermöglichen.
Einige WikiProzessoren arbeiten ständig im Hintergrund. So wird Upper-Camel-Case-Schreibweise in Links zum Artikel mit dem entsprechenden Titel umgewandelt. Ein Artikel kann auch direkt verlinkt werden, falls diese Schreibweise nicht erwünscht ist. Die direkte Linkfunktion ermöglicht es auch auf andere Elemente von Trac zu verweisen. Der WikiProzessor erkennt hierbei die gewünschte Zielkomponente an der Syntax des Links.

Tickets sind ein zentrales Element von Trac. Sie bieten die Möglichkeit schnell Fehler zu melden, Aufgaben oder Anfragen für Erweiterungen zu erstellen.
Tickets haben einen der vier Zustände `new`, `assigned`, `closed` und `reopened`. Neue Tickets sind `new`, `asigned` bedeutet, dass sich jemand diesem Problem angenommen hat. Der Zustand `closed` beschreibt, dass das Ticket geschlossen wurde, wobei der Grund im Feld Resolution angegeben wird. Ein geschlossenes Ticket kann wieder geöffnet werden und ist dann im Zustand `reopened`.
Jedes Ticket hat einen Typ. Mögliche Typen können in der Konfigurationsdatei festgelegt werden. Der Typ entscheidet über die Rolle des Tickets, zum Beispiel als Aufgabe oder gemeldeter Fehler. Ein Ticket erhält als Namen eine kurze Zusammenfassung des Problems. Die genaue Beschreibung erfolgt dann mit Hilfe der WikiEngine und kann alle möglichen Wikiformatierungen umfassen. So können mit TracLinks auch Querverweise auf andere Tickets erfolgen.
Zur Einordnung des Tickets können Priorität oder Schwierigkeit ausgewählt werden. Man kann Komponenten oder Versionen per Auswahlliste zuordnen, die über die Konfiguration voreingestellt werden. Dort können auch bestimmte Felder ausgeblendet werden.
Sogenannte Reports ordnen alle Tickets nach gewünschten Kriterien. Alle Tickets können durchsucht und durch die Suchfunktion TracQuery über den Feldern geordnet werden. Hilfreich ist hier das Keywordfeld um eine Auflistung nach Interessenbereichen zu erreichen, wie Sicherheit oder Stabilität. Da Tickets einzelnen Komponenten, Meilensteinen und Versionen zugeordnet werden können, ist so eine genaue Übersicht über mögliche Problembereiche möglich.

Feld	Beschreibung
Reporter	Urheber des Tickets
Type	Der Zweck des Tickets
Komponente	Der betroffene Teil des Projekts
Version	Betroffene Version der Software
Keywords	Schlüsselworte für Suche und Reportsystem
Priority	Die Dringlichkeit dieser Meldung
Severity	Die Schwierigkeit des Problems
Assigned to/Owner	Primärer Bearbeiter
CC	Benachrichtigung an Emailadressen oder Benutzer
Status	Aktueller Zustand des Tickets
Resolution	Grund für das Schließen des Tickets
Summary	Eine Zusammenfassung des Inhalts
Description	Die genaue Beschreibung des Problems

Tabelle 5: Felder eines Tickets

Tickets können kommentiert und wie Wikiartikel verändert werden. So ist eine Diskussion über das Problem möglich und eine bessere Zusammenarbeit gewährleistet. Auch

das Anhängen von Dateien ist erlaubt.
TracTickets sind also ein einfach zu bedienendes Multifunktionssystem. In Tabelle 5 habe ich alle Eigenschaften eines Tickets aufgeführt, die Trac mitbringt, selbst erstellte Felder sind auch möglich. Abbildung 44 zeigt ein Ticket, wie es auf der Projektseite von Trac dargestellt wird.

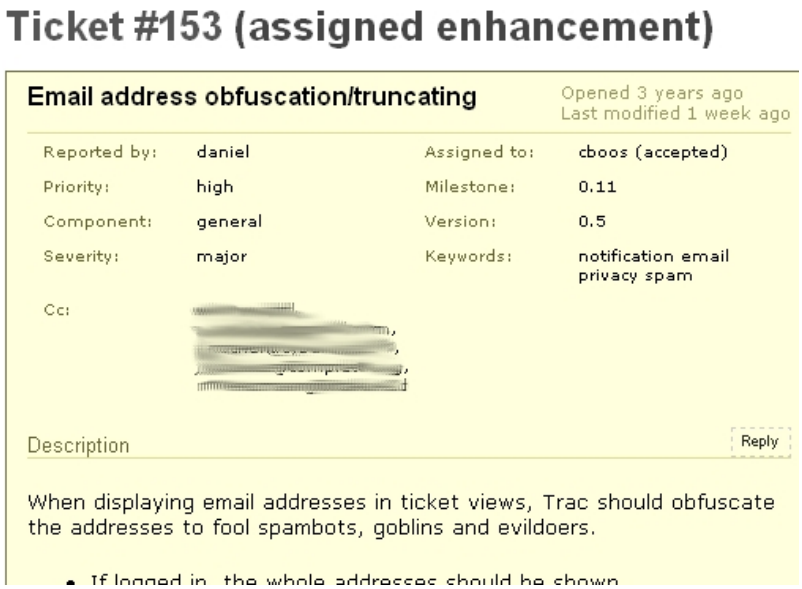

Abbildung 44: Ein Ticket in Trac - Screenshot von [Edg07a]

Die nächste Komponente ist die Roadmap, wo beliebige Meilensteine angelegt werden können. Diese können mit einem Zieldatum, einem Namen und einer Beschreibung in Wikiformversehen werden. Meilensteinen können mit sogenannten Tickets verknüpft werden. In dem Fall wird angezeigt wie viele Tickets noch offen sind, also bearbeitet werden oder bereits geschlossen wurden. Zusätzlich gibt es eine Exportfunktion ins iCalendar-Format. Die Darstellung eines Meilensteins wird in Abbildung 45 gezeigt.
Eine wichtige Eigenschaft von Trac ist, direkt auf das Repository zugreifen zu können. Trac enthält eine sehr enge Schnittstelle zum Versionsverwaltungssystem, die bei jedem Check in ein sogenanntes Changeset erstellt, über das man für die betroffenen Dateien einen schnellen Änderungsvergleich durchführen kann. Das Changeset enthält eine Liste aller betroffenen Dateien. Die Änderungen werden farblich markiert und gleichzeitauch direkt im Quellcode hervorgehoben.
Für einzelne Dateien ist es auch möglich, über den Repository-Browser das Repository manuell zu durchsuchen. So können auch für eine einzelne Datei alle Revisionen durchsucht und verglichen werden. Soll der Vergleich über mehrere Revisionen erfolgen, so bietet Trac hier die RevisionLog-Funktion. Hier kann man die Start- und Endrevision auswählen und erhält dann ein Changeset über alle Änderungen, die zwischen diesen beiden Revisionen liegen.

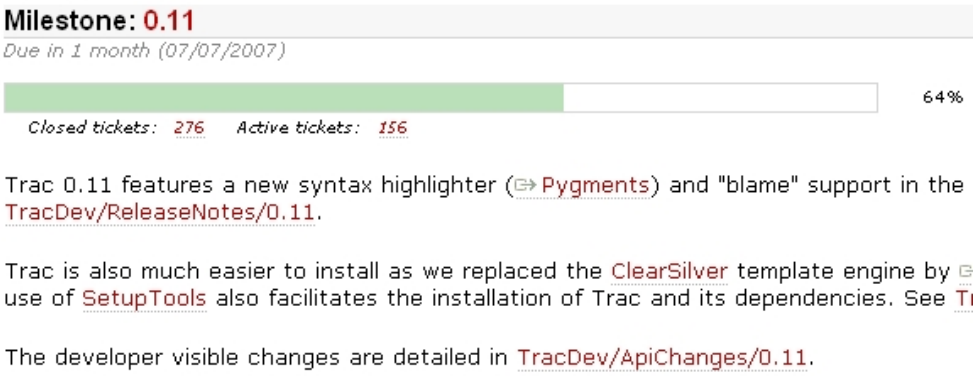

Abbildung 45: Ein Meilenstein in Trac - Screenshot von [Edg07a]

Eine der einfachsten, aber auch nützlichsten Komponenten ist die Timeline, die in Abbildung 46 abgebildet ist. Hier werden alle wichtigen Änderungen im Tracsystem eingetragen und nach Tagen geordnet angezeigt. Dazu gehören Erstellung und Schließen von Meilensteinen und Tickets, das Erstellen und Verändern von Wikiartikeln und zuletzt alle Changesets. So können auf einem Blick alle Änderungen überblickt werden und man ist auf dem neuesten Stand.

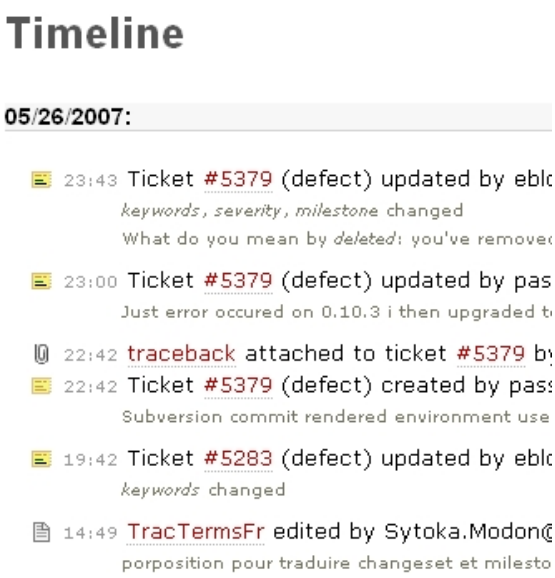

Abbildung 46: Tracs Timelinefunktion - Screenshot von [Edg07a]

7.4.2. Entwicklung und Wartung

Wie im Abschnitt 7.4.1 zu erkennen war, sind die einzelnen Komponenten einfach gehalten und vielseitig. Für Entwickler bietet Trac die Möglichkeit, sich schnell über Pro-

jektangelegenheiten auszutauschen. Man erhält einen schnellen Überblick über neue Tickets und Checkins und kann überprüfen, ob der bevorzugte Arbeitsbereich davon Betroffen ist.
Der Wissensaustausch über das Wiki ist ebenfalls nicht zu unterschätzen. Hier können Anleitungen oder Entwicklungsrichtlinien hinterlegt werden. Es existieren Plugins, die automatisch erstellte Dokumentationen wie JavaDoc oder Pythondoc ins TracWiki integrieren. Auch ein Plugin zur automatischen Ticketerstellung für Testfälle ist verfügbar. Leider gibt es in Trac noch keine engere Verknüpfung zu Testumgebungen, wie etwa JUnit.
Für die Wartung des Projektes sind die Tickets geradezu notwendig. Ohne diese zentrale Rückmeldung der Benutzer könnte man nicht so geordnet Fehlersuche und -korrektur praktizieren. Da auch nicht angemeldete User eine Benachrichtigung bei Kommentaren zum Ticket erhalten können, ist es möglich auf diesem Wege auch eine gewisse Nutzerunterstützung anzubieten.

7.4.3. Projektplanung

Besonders in Hinblick auf die Projektplanung und -Verwaltung sind die Meilensteine sehr nützlich. Da sie bei Trac auch mit Tickets verknüpft werden können, entsteht so ein mächtiges Werkzeug, um den weiteren Fortschritt durch Aufgaben zu planen und eventuelle Blockaden durch Fehlermeldungen zu erkennen. Ein Projektleiter kann ein Ticket einem anderen Teammitglied zuordnen. Auf diese Weise lassen sich auch Aufgaben delegieren.
Es besteht die Möglichkeit, relevante Kategorien durch speziell vereinbarte Schlüsselwörter festzulegen, losgelöst von Meilensteinen oder Versionen. So könnte man beispielsweise einen Report erstellen, der alle Tickets zum Schlüsselwort *Stabilität* auflistet um sich so einen Überblick über das Programmverhalten zu verschaffen.
Je nach Vorgehensweise können auch Gruppen mit speziellen Zugriffsrechten erstellt werden. Da alles in Trac stets geordnet dargestellt wird, bleibt das Vorhaben übersichtlich und kontrollierbar. Außerdem sind keine Vorkehrungen für das Verwalten mehrerer Projekte nötig, da eine Tracinstanz in der Regel nur für ein Projekt verwendet wird.

7.4.4. Vorgehensmodelle

Es gibt mehrere Vorgehensweisen um ein Projekt zu gestalten. Diese kollidieren in der Regel nicht mit Trac.
Nehmen wir als Beispiel das V-Modell, so können wir über Meilensteine und Aufgabentickets die einzelnen Schritte vorausplanen. Während jedem Abschnitt des V-Modells kann ein Ticket mit Testanforderungen zu jeder Komponente erstellt und an den Meilenstein der entsprechenden Testphase geknüpft werden. Ein Plugin für eine automatisierte Testumgebung ist das TestCaseManagementPlugin.
Für Vorgehensmodelle, die früh auf praktische Arbeit setzen, wie das Prototyping und das Extreme-Programming ist Trac gut nutzbar. Einzelne Prototypen sind zum Beispiel

entweder über Meilensteine und Komponenten oder über Versionen und Schlüsselwörter darstellbar.
Für das Extreme-Programming, das stark auf universelle Zusammenarbeit setzt, ist Trac ebenfalls geeignet, da hier die Entwickler nicht zu festen Gruppen zusammengeschlossen werden. Jeder Entwickler kann alle Tickets einsehen und bearbeiten. Bei Engpässen können Aufgaben von Anderen übernommen werden. Da hierfür alle Teammitglieder ein vergleichbares Wissen über das Projekt haben sollten, dient das Wiki als Informationsquelle.
Ein weniger optimale Lösung stellt Trac für Projekte dar, an deren Anfang viel theoretische Arbeit, wie UML-Diagramme, Use-Cases oder Analysen stehen. Für diese Arbeit bietet Trac nur die Möglichkeit, diese Dokumente als Anhänge von Tickets oder im Repository zu hinterlegen. Letzteres bietet noch den Vorteil der Versionierung. Fertiggestellte Dokumente können im Wiki verlinkt oder angezeigt werden. Auch gibt es Möglichkeiten, die Wikieinträge als verschiedene Dokumententypen zu exportieren und so in die Projektdokumentation einzufügen.
Zusammenfassend stellt Trac für die meisten verbreiteten Vorgehensmodelle eine gute Unterstützung dar.

7.5. Mylar

Für die Entwickler bietet die Möglichkeit deutliche Vorteile, Funktionen von Groupware wie Bugzilla oder Trac direkt aus der Entwicklungsumgebung heraus zu nutzen. Man bedarf nur noch der einen Software um das Projekt zu bearbeiten und Aufgaben können in direkten Zusammenhang mit dem Quellcode geplant werden. Auch die Möglichkeit, offline auf die Tasks zugreifen zu können kann in Fällen hilfreich sein, wenn man unterwegs arbeiten möchte und kein Internet verfügbar ist.
Genau hier setzt das Eclipseplugin Mylar an, indem es die Funktionalität von Bugtrackingsystemen, wie Bugzilla, Trac und JIRA in die Arbeitsumgebung von Eclipse einbindet. Es funktioniert mit Eclipseversion 3.2 und Java 5. Welche Versionen noch Unterstützt werden, findet man unter [The].
Zur Zeit werden nur die drei genannten Systeme unterstützt, es ist aber möglich, andere durch einen generischen Konnektor zu verwenden, der wichtige Daten aus HTML-Code heraus liest. Je nach genutztem System wird unterschiedlich auf die Tasks zugegriffen, so dass bei Trac das XmlRpcPlugin installiert sein sollte, damit die Tickets nicht über den eingebetteten Webbrowser angezeigt werden, sondern direkt bearbeitet werden können.
Die Nutzung von Mylar ist schnell erlernt, wie Kersten in [Ker06] zeigt. Um auf Tasks zugreifen zu können, wird zuerst ein Taskrepository eingerichtet. Nun werden je nach Zielrichtung Queries erstellt, in denen man Kategorien wie Meilensteine oder Ticketstatus, aber auch freie Suchbegriffe speichern kann. Unter diesen Queries werden dann die Tasks, die unter diese Bedingungen fallen, angezeigt und können bearbeitet werden.
Statt Onlinetasks ist es auch möglich persönliche, lokale Tasks einzurichten. Statt einem

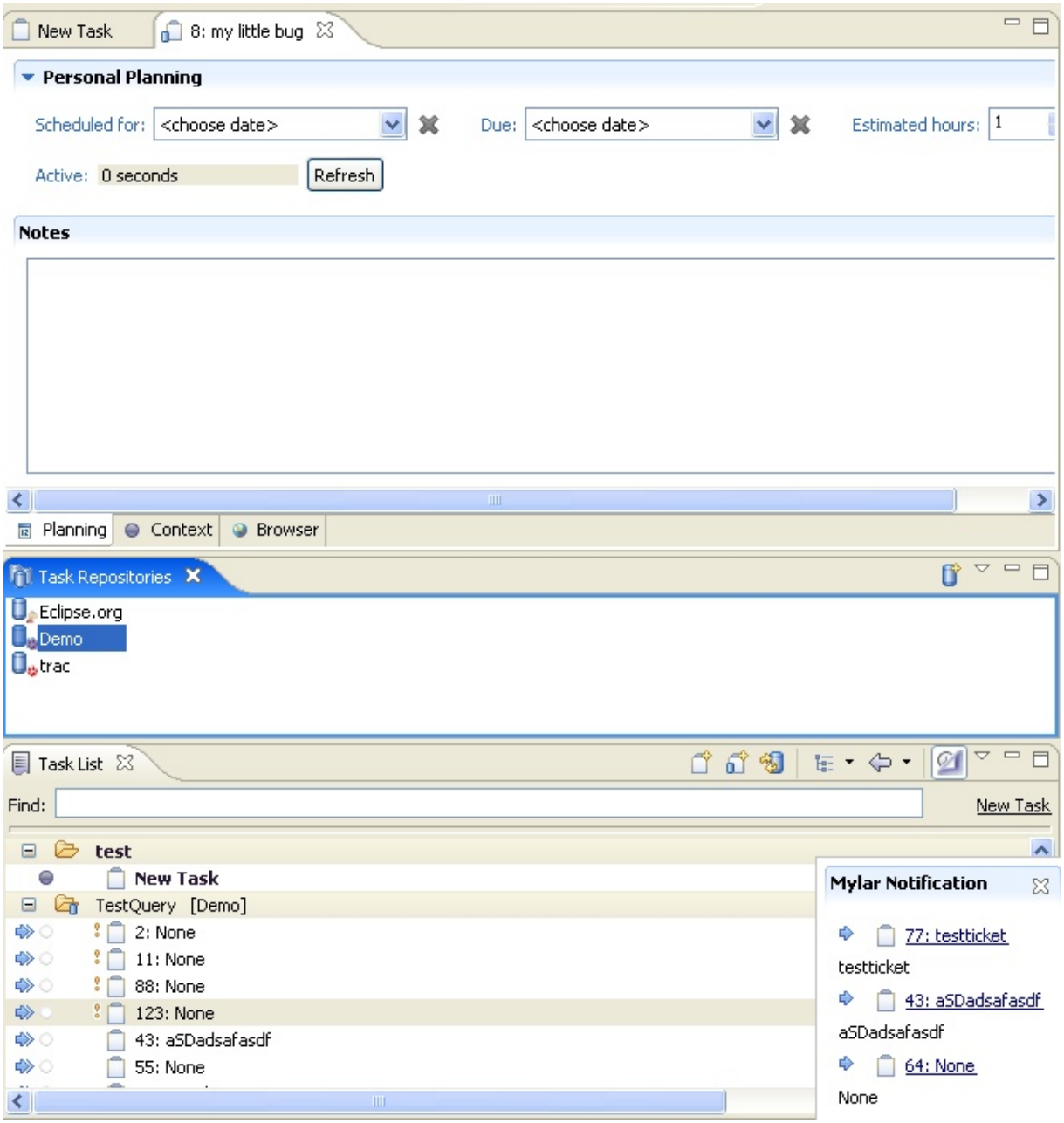

Abbildung 47: Mylar im Eclipseworkbench

Repository wird hier eine Kategorie erstellt. Ansonsten unterscheiden sich die Tasks nur in den Besonderheiten der einzelnen Systeme. Alle Tasks können persönlich geplant und kommentiert werden. Einen Eindruck vermittelt ein Screenshot meiner Eclipseversion in Abbildung 47. Tasks können auch als aktiv markiert werden, wodurch die Bearbeitungszeit festgestellt wird. Dies kann helfen, den Arbeitsaufwand bei ähnlichen Aufgaben besser einzuschätzen. Wenn ein Task abgeschlossen ist, so wird er ebenfalls auf dem Repository geschlossen.
Diese Einbindung in die Entwicklungsumgebung scheint zwar nicht notwendigerweise den Entwicklungsprozess zu verbessern, aber sie erleichtert die persönliche Planung und Nutzung von vorhandenen Projektmanagementsystemen. Ist man in mehreren Projekten involviert, so ist die Möglichkeit sehr hilfreich, mehrere oder verschiedene Online-Repositories parallel verwenden zu können.

7.6. Schlussfolgerung

Wie in den Abschnitten 7.3 und 7.4 gezeigt wurde, sind beide Groupwareanwendungen dazu geeignet, sowohl die Arbeit für den Einzelnen, als auch für Gruppen zu verwalten. Beide erfüllen die Anforderungen nach Kommunikation, Koordination und Kooperation. Auch ein Projektmanagement ist möglich, jeweils in unterschiedlichen Varianten. Dennoch sind beide nicht für den gleichen Einsatzbereich geeignet.
Open-Xchange bietet viele Funktionen, die man von einer Groupware erwarten kann, eignet sich jedoch wenig für die Verwendung in einer Softwareentwicklung, wie in Abschnitt 7.3.2 dargelegt wurde.
Trac bietet weniger Funktionen als Open-Xchange, ist aber auch enger an die Erfordernisse einer Softwareentwicklung angepasst. Die Fähigkeit, Änderungen am Quellcode automatisch anzuzeigen, ist in dieser Hinsicht eines der wichtigsten Elemente. Auch das Ticketsystem bietet viele Funktionen, während es einfach zu verwenden ist. So steht Trac nicht in Konflikt mit verbreiteten Vorgehensmodellen, wie dem V-Modell oder dem Extreme-Programming. Für viele Probleme und Wünsche gibt es Lösungen mit Trac. Allgemein betrachtet sind die einfache Nutzbarkeit und Übersichtlichkeit Tracs aber die wichtigsten Eigenschaften, um es als Entwicklungsgroupware in Betracht zu ziehen.
Es fehlt zwar bei einigen Bereichen noch die Integration, aber wie Mylar zeigt, geht auch hier die Entwicklung voran. Man sollte nicht vergessen, dass Trac auch ein Opensourceprojekt ist, das unentwegt erweitert und verbessert wird. So komme ich zu dem Schluss, dass Trac und ähnliche Anwendungen und Tools wie Mylar eine wichtige, sinnvolle Unterstützung für solche Vorhaben wie eine Projektgruppe darstellen.

Literatur

[Edg07a] EDGEWALLSOFTWARE: *Trac*, 2007. http://trac.edgewall.org/, Letzter Besuch: 29.05.2007.

[Edg07b] EDGEWALLSOFTWARE: *Trac User Guide*, 05 2007. http://trac.edgewall.org/wiki/TracGuide, Letzter Besuch: 25.05.2007.

[Ker06] KERSTEN, M.: *Task-focused programming with Mylar, Part 1*. IBM developerWorks, 2006. http://www.ibm.com/developerworks/java/library/j-mylar1/, Letzter Besuch: 15.05.2007.

[Ope] OPENXCHANGE: *Open-Xchange Server 5*. http://www.open-xchange.com/DE/header/products/, Letzter Besuch: 26.05.2007.

[Rot00] ROTH, J.: *Entwicklungs- und Laufzeitunterstützung für synchrone Groupware*. dissertation.de - Verlag im Internet GmbH, 2000.

[The] THEECLIPSEFOUNDATION: *Mylar Project*. http://www.eclipse.org/mylar/, Letzter Besuch: 15.05.2007.

[Wul97] WULF, V.: *Konfliktmanagement bei Groupware*. Vieweg Verlag, Braunschweig/Wiesbaden, 1997.

8. Optimierungsheuristiken

Autor: Mark Eilers

8.1. Einleitung

Im Rahmen der Projektgruppe *Demonstrator InHouse-Energiemanagement* stellt sich die Aufgabe einen Geräteeinsatzplan zu optimieren. Ein Geräteeinsatzplan kann als Zeitplan über den Einsatz von Haushaltsgeräten in einem Haushalt angesehen werden. Der Einsatz dieser Haushaltsgeräte soll so geplant werden, dass die anfallenden Stromkosten minimiert werden. Da theoretisch jedes betrachtete Haushaltsgerät an nahezu jeden beliebigen Zeitpunkt verschoben werden kann, führt dies zu einer solch gewaltigen Anzahl verschiedener Kombinationen, dass deren vollständige Überprüfung von heutigen Computern nicht in angemessener Zeit bewältigt werden kann. Diese Arbeit versucht daher einen Überblick über einige bekannte Verfahren zu geben, die angewandt werden können, um solch ein Optimierungsproblem zu lösen. Natürlich können nicht alle Lösungsverfahren in dieser Arbeit Beachtung finden. Der Fokus liegt daher auf Verfahren, die für das Lösen des gegebenen Problems angemessen erscheinen.

8.2. Kombinatorische Optimierungsprobleme

Ein Optimierungsproblem ist definiert als die Suche nach einer optimalen Lösung für ein meist mathematisches Problem. Um eine Lösung eines Optimierungsproblems überhaupt bewerten zu können, wird eine Zielfunktion definiert, die jeder Lösung des Problems eine meist reelle Zahl zuordnet.
Kombinatorische Optimierungsprobleme stellen eine Unterart von Optimierungsproblemen dar, bei denen diskrete Elemente einer endlichen Lösungsmenge so kombiniert oder angeordnet werden müssen, dass die dazugehörige Zielfunktion optimiert wird (siehe [SM07]). Die meisten kombinatorischen Optimierungsprobleme gehören zu den *NP-schweren* Problemen, das heißt, sie können von keinem deterministischen Algorithmus gelöst werden, der nicht mindestens eine exponentielle Zeitkomplexität $O(d^n)$ mit $d > 1$und Problemmgröße n besitzt (siehe [Far04]).
Die Optimierung des Geräteeinsatzplans kann als kombinatorisches Optimierungsproblem verstanden werden, bei dem eine Lösung den Einsatz der eingeplanten Geräte zu einem möglichen Zeitpunkt darstellt. Es gibt zwei unterschiedliche Ansätze um kombinatorische Optimierungsprobleme zu lösen, die *exakten* und die *heuristischen Lösungsverfahren*.

8.3. Exakte Lösungsverfahren

Exakte Lösungsverfahren, oder auch Optimierungsverfahren genannt, zeichnen sich dadurch aus, dass sie garantiert ein optimales Ergebnis liefern oder zeigen, dass es keine

Lösung gibt (siehe [WD06]). Im Allgemeinen gibt es zwei verschiedene Ansätze für exakte Lösungsverfahren, die sich in der Art der Lösungsaufzählung unterscheiden. Bei Verfahren mit expliziter Enumeration wird der gesamte Lösungsraum überprüft und die beste gefundene Lösung zurückgegeben. Voraussetzung hierfür ist natürlich die, bei kombinatorischen Optimierungsproblemen gegebene, Endlichkeit des Suchraums. Ein bekanntes Beispiel hierfür ist *Brute-Force*, das Ausprobieren aller Möglichkeiten einer Problemstellung. Obwohl Optimierungsverfahren mit expliziter Enumeration ein garantiert optimales Ergebnis liefern sind sie nur für kleine Problemgrößen interessant, da sie eine garantiert exponentielle Laufzeitkomplexität $O(d^n)\ mit\ d > 1$ besitzen.
Im Gegensatz dazu wird bei Optimierungsverfahren mit impliziter Enumeration versucht durch implizites und während der Berechnung gewonnenes Wissen über das Optimierungsproblem und den Suchraum, Teile des Lösungsraums, noch vor deren Berechnung, als nicht mehr zu der optimalen Lösung führend, verwerfen zu können. Ein bekanntes Beispiel dafür ist das Optimierungsverfahren Branch & Bound.

8.3.1. Branch & Bound

Die grundlegende Idee hinter Branch & Bound ist, den Lösungsraum nach und nach in einen Suchbaum einzuteilen und dabei Teilbäume, die zu keiner optimalen Lösung mehr führen können, zu eliminieren. Das Erstellen des Suchbaums wird als *branching* bezeichnet, die Überprüfung, ob ein Teilbaum weiter untersucht werden muss, als *bounding*. Das Verfahren soll nun anhand eines Beispiels (leicht verändert aus [Amb05] übernommen) zum *Problem des Handelsreisenden*, im Folgenden TSP (Traveling Salesman Problem) genannt, genauer erläutert werden.

8.3.2. Branch & Bound: Ein Beispiel

Das TSP-Problem ist ein kombinatorisches Optimierungsproblem von NP-äquivalenter Komplexität (siehe [WD06]). Eine Anzahl von n Knoten ist jeweils untereinander durch Kanten verschiedener Länge verbunden. Das Problem besteht nun darin eine Rundreise, vom Ausgangsknoten über jeden Knoten wieder zurück zum Ausgangsknoten, so zu wählen, dass die zurückgelegte Gesamtkantenlänge minimal wird. Für ein vollständiges, symmetrisches TSP-Problem wie in Abb. 48 gibt es $n!$ Lösungen, die verschiedenen Kombination aller Knoten. Das erste implizite Wissen, das wir für Branch & Bound benutzen wollen, ist nun Folgendes: Die Hälfte der Lösungen kann vernachlässigt werden, da es für die Lösung keinen Unterschied macht, ob eine Rundreise vorwärts oder rückwärts antreten wird. Die Lösung (A B C D E A) ist also gleich der Lösung (A E D C B A). Da außerdem gilt, dass die Lösungen (A B C D E A) und (B C D E A B) die gleiche Lösungsgüte besitzen, kann auch festgelegt werden, dass alle Rundreisen im Knoten A starten. Diese Einschränkungen führen dann zu

$$\frac{(n-1)!}{2}$$

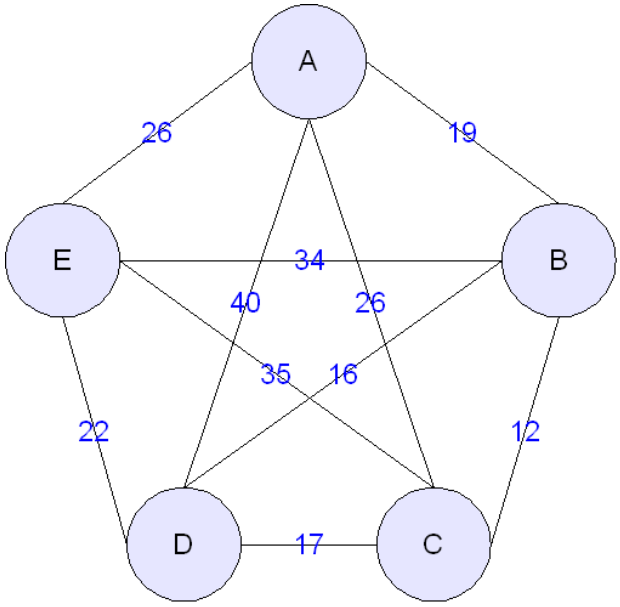

Abbildung 48: Problem des Handelsreisenden

unterschiedlichen Lösungen, im konkreten Beispiel also nur noch 12 zu betrachtende Rundreisen.
Beim branching spannt Branch & Bound einen Suchbaum auf, das Optimierungsproblem wird dabei in Teilprobleme zerlegt. In jedem Knoten des Suchbaums wird nun beim bounding berechnet, ob der noch zu berechnende Teilbaum vernachlässigt werden kann. Dazu benötigt Branch & Bound so genannte Schranken. In jedem Knoten des Suchbaums wird nun überprüft, ob das Teilproblem, gemessen an einer unteren Schranke (der günstigsten Vervollständigung der Lösung), eine obere Schranke (die beste bisherige Lösung) übertritt. Ist das der Fall, bedeutet dies, dass alle Teillösungen, die durch diesen Knoten führen, selbst im günstigsten Fall garantiert schlechter sein müssen als die bisher beste gefundene Lösung. Der Teilbaum muss also nicht weiter betrachtet werden.
Für das TSP-Problem in Abb.48 bietet sich die folgende Umsetzung an: Die Knoten und Kanten des Suchbaums entsprechen den Knoten und Kanten der Problemstellung, die Güte einer Lösung entspricht der Gesamtlänge einer Rundreise. Jeder komplette Abstieg durch den Suchbaum entspricht daher genau einer Rundreise des Handelsreisenden. Da zumindest eine Lösung als obere Schranke bekannt sein muss, wird eine Rundreise von Branch & Bound berechnet, nämlich (A B C D E A) mit einer Länge von 96. Die untere Schranke ist die günstigste Vervollständigung einer Lösung, also einer Rundreise. Aus der Tiefe eines Knotens im Suchbaum kann berechnet werden, wie viele Knoten noch für eine komplette Rundreise besucht werden müssen. Als untere Schranke bietet sich also die Entfernung an, die für die Durchquerung der fehlenden Knoten noch mindestens zurückgelegt werden muss. Zu diesem Zweck wird eine Tabelle mit den vier kürzesten Kanten des Graphen aufgestellt.

Verbleibende Knoten	Kürzeste Strecke	Summe
1	12	12
2	16	28
3	17	45
4	19	64

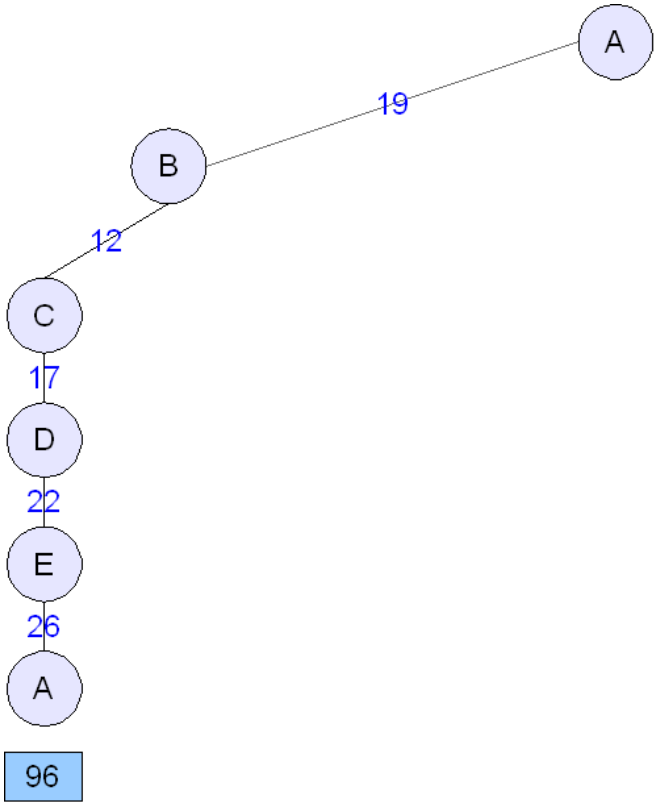

Abbildung 49: Branch & Bound: Der Anfangszustand

Die Konstruktion des Suchbaums beginnt mit der Konstruktion der oberen Schranke, der Rundreise (A B C D E A) mit der Lösungsgüte 96 (siehe Abb.49). Im Knoten B wird nun überprüft ob ein Teilbaum vernachlässigt werden kann (bounding) (Dass diese Prüfung erst im Knoten B beginnt und nicht im Knoten C, liegt an der impliziten Verkleinerung des Lösungsraum auf 12 zu betrachtende Lösungen). Nach dem Knoten B müssen für eine komplette Rundreise noch mindestens vier Knoten besucht werden. Die bisherige Teillösung (A B) ergibt eine Strecke von 19. Die Tabelle zeigt, dass für vier verbleibende Knoten, eine Strecke von mindestens 64 zurückgelegt werden muss. Die günstigste Vervollständigung der Lösung ergibt also: $19 + 64 = 83 < 96$. Die obere Schranke wird nicht überschritten, der Teilbaum wird daher um den Knoten D erweitert (branching) und muss weiterhin untersucht werden. Die Teillösung (A B D) ergibt eine Strecke von 35, drei weitere Knoten müssen besucht werden. Die günstigste Vervollständigung der Lösung lautet also $35 + 45 = 80 < 96$ Die obere Schranke wird wiederum

nicht überschritten, der Teilbaum wird um den Knoten E erweitert. Aus der Teillösung (A B D E) und noch zwei fehlenden Knoten lässt sich wiederum $57 + 28 = 85 < 96$ folgern, der Teilbaum wird um den Knoten C erweitert. Die Teillösung (A B D E C) ergibt nun eine Strecke von 92 und ein weiterer Knoten muss besucht werden. Die günstigste Vervollständigung dieser Lösung lautet jetzt allerdings $92 + 12 = 104 > 96$. Die obere Grenze ist überschritten. Die Teillösung würde also auch bei günstigster Vervollständigung eine schlechtere Lösungsgüte besitzen als unsere bisher beste Lösung. Daher kann die Überprüfung des Teilbaums hier abgebrochen werden und der nächste Teilbaum wird überprüft. Nach dem selben Regeln wird jetzt Schritt für Schritt der komplette Suchbaum ermittelt und Branch & Bound beendet (siehe Abb.50).

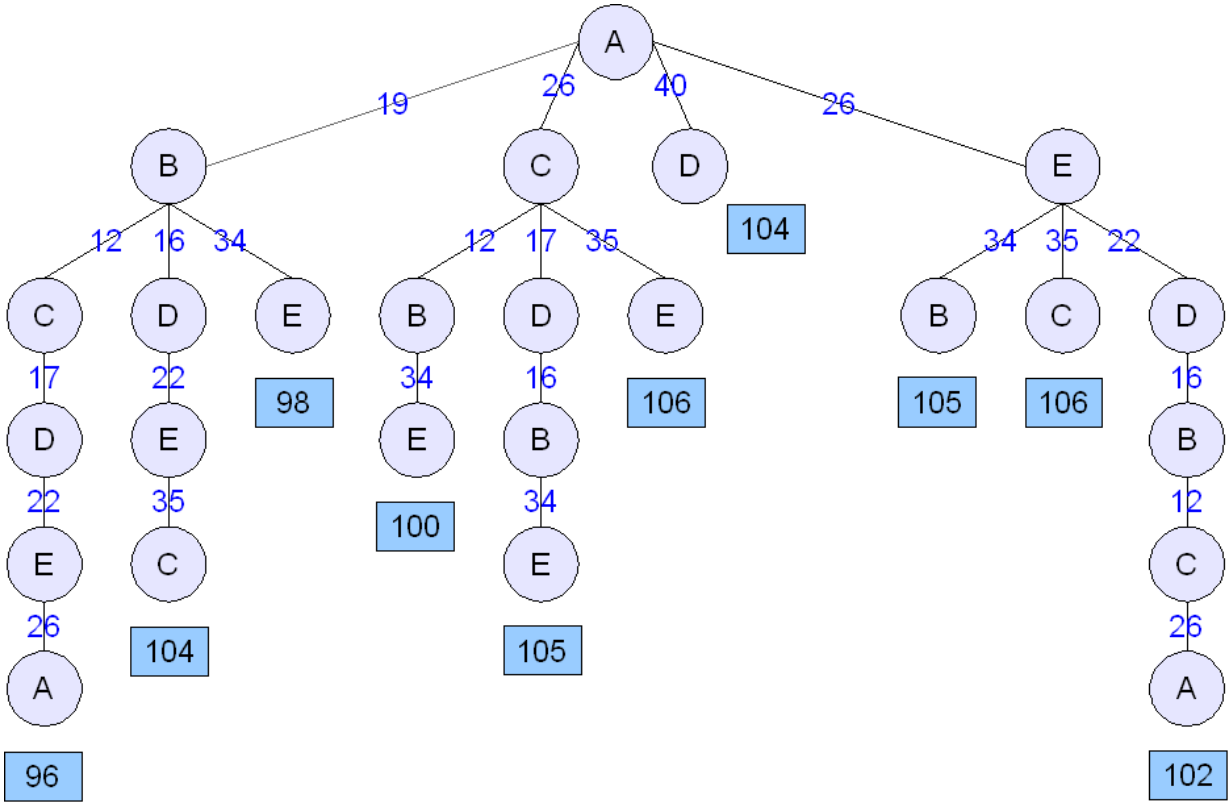

Abbildung 50: Branch & Bound: Der Endzustand

In diesem Beispiel konnte durch Branch & Bound schon über die Hälfte des Lösungsraums vernachlässigt werden, was durch eine genauere Festlegung der unteren Grenze, zum Beispiel eine Überprüfung, ob die vier kürzesten Strecken in einer einzelnen Rundreise überhaupt benutzt werden dürfen, noch verbessert werden könnte.
Branch & Bound ist in der Lage einen Lösungsraum stark einzugrenzen und dadurch in relativ schneller Zeit eine garantiert optimale Lösung zu finden. Die Schwierigkeit bei der Verwendung von Branch & Bound für abstraktere kombinatorische Optimierungsprobleme liegt jedoch in der Auswahl geeigneter Grenzen. Sie sollten so gewählt

werden, dass ausreichend viele Teilbäume eliminiert werden können, aber die optimale Lösung auf jeden Fall gefunden wird.
Unabhängig davon können gute Grenzen das Verfahren zwar extrem beschleunigen, dennoch besitzt Branch & Bound, wie alle exakten Lösungsverfahren, im schlechtesten Fall aber leider eine exponentielle Laufzeitkomplexität.

8.4. Heuristische Lösungsverfahren

Die Verwendung von exakten Lösungsverfahren erweist sich bei praktischen Problemen meist als zu zeitaufwändig. Eine Alternative zu den Optimierungsverfahren bieten hier die so genannten Heuristischen Lösungsverfahren oder Heuristiken. Im Gegensatz zu den exakten Lösungsverfahren garantieren Heuristiken nicht das optimale Ergebnis sondern nähern sich diesem Optimum nur an. Heuristiken liefern in kurzer Zeit ein relativ gutes Ergebnis. Über die tatsächliche Güte der Lösung kann im Allgemeinen jedoch keine Aussage getroffen werden. Heuristiken, die eine Gütegarantie geben nennt man Approximationsverfahren, sie werden in dieser Arbeit nicht weiter behandelt. Heuristische Lösungsverfahren bieten sich immer dann an, wenn für die Lösung eines Optimierungsproblems nicht zwangsläufig die optimale Lösung erforderlich ist oder wenn eine kurze Bearbeitungszeit für eine relativ gute Lösung wichtiger ist als die optimale Lösung. Es gibt unterschiedliche Klassifikationen von Heuristiken, generell muss man aber zwischen den *problemspezifischen Heuristiken* und den *Metaheuristiken* unterscheiden (siehe [Mer03]).

8.5. Problemspezifische Heuristiken

Im Rahmen dieser Arbeit wird eine Heuristik, die für das Lösen eines bestimmten Optimierungsproblems entwickelt wurde, und nur auf dieses Optimierungsproblem angewandt werden kann, als problemspezifisch bezeichnet. Innerhalb der Problemspezifischen Heuristiken muss zwischen Konstruktions- und Verbesserungsheuristiken unterschieden werden (siehe [GR06]).

8.5.1. Konstruktionsheuristiken

Konstruktionsheuristiken konstruieren eine und nur eine Lösung. Viele von ihnen basieren auf dem *Greedy*-Prinzip. Auf diesem Verfahren basierende Heuristiken wählen ausgehend von einem Startpunkt jeweils die momentan günstigste Erweiterung der Lösung. So beginnt die *Nearest-Neighbour*-Konstruktionsheuristik für das TSP-Problem mit einem beliebigen Knoten und vervollständigt die Lösung durch die Wahl des jeweils am nächsten liegenden Knoten. Sie vervollständigt also "'gierig"', nicht vorausschauend. Die nach dem Greedy-Prinzip benannte *Greedy*-Heuristik beginnt für das TSP-Problem mit der kürzesten Kante und vervollständigt die Lösung dann jeweils mit der angrenzenden kürzesten Kante. Obwohl diese beiden Heuristiken nach dem gleichen Prinzip

arbeiten und recht ähnlich scheinen, sind sie doch eigenständige problemspezifische Konstruktionsheuristiken, die nur für bestimmte Probleme Anwendung finden.

8.5.2. Verbesserungsheuristiken

Verbesserungsheuristiken können im Gegensatz zu den Konstruktionsheuristiken keine initiale Lösung konstruieren, sondern benötigen eine bereits bestehende Lösung, die dann schrittweise verbessert wird. Typische Vertreter für Verbeserungsheuristiken sind die *k-opt*-Heuristiken, bei denen jeweils *k* Elemente der Lösung vertauscht werden. Die so entstehende neue Lösung wird bewertet und, wenn sich diese Lösung als besser herausstellt, das Verfahren wiederholt. So werden bei einer bestehenden Lösung zum TSP-Problem, also einer konstruierten Rundreise, bei der 2-opt-Optimierung jeweils 2 Knoten getauscht. Zahlreiche Versuche haben gezeigt, dass 3-Opt-Heuristiken, gefolgt von 2-Opt-Heuristiken, für verschiedene Probleme bei festgelegter Laufzeit gegenüber k-opt-Heuristiken bessere Ergebnisse liefern. Die Berechnungen der Tauschoperationen höherer k-opt-Heuristiken sind bei gleicher Laufzeit meist zu aufwändig als das gute Ergebnisse möglich wären.
Die Berechnung von geringen Änderungen in einer Lösung lässt sich bei wachsender Problemgröße schneller berechnen als die Neuberechnung der Lösung. Dies ist auch leicht nachvollziehbar. Angenommen die Rundreise eines Handelsreisenden geht über *n* Knoten. Von einer bekannten Lösung ausgehend werden jetzt zwei Knoten getauscht. Die komplette Berechnung der neuen Lösungen würde genau *n* Rechenoperationen verlangen, die Additionen der Längen aller bereisten Kanten. Die Berechnung der neuen Lösung durch die alleinige Berechnung der Änderung benötigt dagegen genau acht Rechenoperationen, unabhängig von *n*, nämlich die Subtraktion der vier bereisten Kanten, die die Tour mit den zu tauschenden Knoten verbindet, und die Addition der vier neuen Kanten.

8.6. Metaheuristiken

Metaheuristiken sind im Gegensatz zu den problemspezifischen Heuristiken nicht auf ein spezielles Optimierungsproblem zugeschnitten. Sie beschreiben Vorgehensweisen, die bei einer großen Bandbreite von Optimierungsproblemen angewandt werden können. So eine Vorgehensweise besteht meist aus einer Abfolge von Verfahren, die dann für eine spezielle Aufgabe aber problemspezifisch umgesetzt werden müssen. Ein Beispiel dafür ist die Metaheuristik *Local Search*. Der Name Local Search resultiert aus der Tatsache, dass Local Search die *Nachbarschaft* einer gegebenen Lösung auf eine bessere Lösungsgüte untersucht. Lösungen, die sich voneinander in nur wenigen Punkten unterscheiden, nennt man *benachbart*. Eine *Nachbarschaft* ist genauer definiert als

> Die Menge der Lösungen, die von einer gegebenen Lösung durch eine einfache Veränderungsoperation erreicht werden können (siehe [Mer03]).

Nachbarschaften lassen sich häufig relativ schnell berechnen. Für das Untersuchen der Nachbarschaft kennt Local Search verschiedene Varianten. Beim *Best Improvement* wird die komplette Nachbarschaft untersucht und die Lösung mit der größten Verbesserung gewählt. Dagegen wird beim First Improvement die Nachbarschaft nur so lange untersucht bis sich eine bessere Lösung als die Ausgangslösung findet. Die Reihenfolge, mit der die Nachbarschaft untersucht wird kann dabei jeweils zufällig oder festgelegt sein. Unabhängig von der Art der Auswahl wird das Verfahren mit der verbesserten Lösung so lange wiederholt, bis in der Nachbarschaft keine bessere Lösung mehr gefunden wird, worauf Local Search terminiert.
Local Search ist ein Verfahren, dass sich, Metaheuristiken gemäß, in seinem Grundgedanken bei einem sehr großes Spektrum an Optimierungsproblem anwenden lässt. So sind die problemspezifischen k-opt-Heuristiken im Grunde umgesetzte Local Search-Heuristiken mit unterschiedlichen Nachbarschaftsgrößen.
Local Search findet garantiert ein mindestens lokales Optimum. Aber ein lokales Optimum kann gemessen am globalen Optimum dennoch beliebig schlecht sein. Nun erheben Heuristiken auch nicht den Anspruch das globale Optimum einer Zielfunktion zu finden, wünschenswert wäre jedoch ein Wert der die Lösungsgüte betreffend zumindest in der Nähe des globalen Optimums liegt. Möglichkeiten zur Lösung dieses Problems wären, Local Search von verschiedenen Startpunkten durchzuführen oder für Best Improvement die zu überprüfende Nachbarschaft zu vergrößern. Beides ist jedoch mit einem stark erhöhten Aufwand verbunden, der in der Praxis meist nicht vertretbar ist.
Ein anderer Ansatz sind Metaheuristiken die in der Lage sind lokale Optima zu überwinden. Die wohl bekannteste dieser Metaheuristiken ist *Tabu-Search*.

8.6.1. Tabu-Search

Tabu-Search ist eine Erweiterung der Local Search Heuristik. Tabu-Search untersucht die Nachbarschaft einer bekannten Lösung. Die Initiallösung wird meist von einer problemspezifischen Konstruktionsheuristik geliefert, nach dem Best-Improvement-Verfahren. Anders als bei Local Search wird allerdings auf jeden Fall die beste Lösung der Nachbarschaft ausgewählt, auch wenn diese **schlechter** als die Ausgangslösung sein sollte. Danach wird das ganze Verfahren mit der neuen Lösung wiederholt. Tabu-Search ist es dadurch möglich lokale Optima zu verlassen. Dies führt jedoch zu Zyklen. Zum Beispiel würde Tabu-Search nach dem Verlassen eines lokalen Optimums im nächsten Schritt ja sehr wahrscheinlich das gerade verlassene Optimum als beste Lösung der Nachbarschaft entdecken und zurückkehren. Aber auch größere Zyklen treten häufig auf. Um solche Zyklen zu vermeiden, führt Tabu-Search so genannte *Tabu-Listen*, in denen die bisher besuchten Lösungen gespeichert werden. Bei jedem Schritt wird die zuletzt ausgewählte Lösung in die Tabu-Liste aufgenommen. Diese Lösungen sind dann *tabu*, also für eine weitere Auswahl gesperrt. Durch die Tabu-Liste ist Tabu-Search theoretisch in der Lage jedes Optimum auch wieder zu verlassen. Dies gilt auch für ein mögliches globales Optimum. Daher ist es nötig geeignete Abbruchbedingungen aus-

zuwählen und die beste gefundene Lösung zu speichern.
Wie diese Tabu-Liste genau aufgebaut ist, muss wieder problemspezifisch entschieden werden. Es ist wie oben beschrieben möglich die bisher besuchten Lösungen komplett in der Tabu-Liste zu speichern. Bei größeren Optimierungsproblemen stellt dies aber meistens einen nicht unerheblichen Speicher- und Zeitaufwand dar. Häufiger werden daher nur Lösungsattribute in die Tabu-Liste geschrieben. Beim Untersuchen der Nachbarschaft sind dann alle Lösungen verboten, die Tabu-Attribute enthalten. Beim Problem des Handelsreisenden könnte so ein Tabu-Attribut zum Beispiel das Tauschen spezieller Knoten sein. Wird mit Tabu-Attributen gearbeitet, so muss festgehalten werden, wie lange und wie viele Tabu-Attribute gespeichert werden. Ist eine Tabu-Liste zu kurz kann dies zu Zyklen führen, ist sie zu lang, wird meistens ein zu großer Teil des Lösungsraums verboten. Insbesondere kann das Verbieten von Lösungsattributen dazu führen, dass eine Lösung verboten ist, obwohl sie besser ist als alle bisher Gefundenen. Zu diesem Zweck werden häufig Aspirationskriterien aufgestellt, die eine Auswahl trotz Tabu-Status erlauben. Für die Auswahl einer geeigneten Tabu-Liste gibt es kein allgemein gültiges Konzept, die geeignete Art und Länge einer Tabu-Liste muss problemspezifisch gewählt und durch Experimentieren geprüft und gegebenenfalls verändert werden.
Die problemspezifische Anwendung der Metaheuristik Tabu-Search soll nun wieder an einem kleinen Beispiel verdeutlicht werden.

8.6.2. Tabu-Search: Ein Beispiel

Gegeben sei das bereits bekannte TSP-Problem (siehe Abb.48). Tabu-Search benötigt eine Startlösung, angenommen eine für dieses Problem ausgewählte Konstruktionsheuristik liefert die Rundreise (A E D B C A) mit einer Länge von 102. Als benachbart werden alle Rundreisen definiert, die sich an genau zwei Stellen unterscheiden, also alle Rundreisen, die man durch den Tausch zweier Knoten konstruieren kann. Die Tabu-Liste soll als Tabu-Attribute das Vertauschen zweier Knoten enthalten, von unendlicher Länge sein und die Tabu-Attribute unendlich lange halten. Weiter soll Tabu-Search nach drei Durchgängen abbrechen und das beste bisher gefundene Ergebnis liefern.

Aktuelle Lösung	Nachbarschaft	Tabu-Liste	Beste Lösung
A E D B C A = 102	{ }	{ }	A E D B C A = 102

Nach der Initialisierung sind Nachbarschaft und Tabu-Liste noch leer. Nun wird die Nachbarschaft der aktuellen Lösung berechnet, die vertauschten Knoten sind jeweils hervorgehoben.

Aktuelle Lösung	Nachbarschaft	Tabu-Liste	Beste Lösung
A E D B C A = 102	A ***C*** D B ***E*** A = 119	{ }	A E D B C A = 102
	A ***B*** D ***E*** C A = 118		
	A E ***C*** B ***D*** A = 129		
	A ***D*** ***E*** B C A = 135		
	A E ***B*** ***D*** C A = 119		
	A E D ***C*** ***B*** A = 96		

Die Rundreise (A E D C B A) ist mit einer Länge von 96 die beste Lösung aus der Nachbarschaft sowie die beste bisher gefundene Lösung und wird als nächste Startlösung ausgewählt. Der Tausch der Knoten B und C wird in die Tabu-Liste aufgenommen und die neue Nachbarschaft berechnet.

Aktuelle Lösung	Nachbarschaft	Tabu-Liste	Beste Lösung
A E D C B A = 96	A ***B*** D C ***E*** A = 113	{B<->C}	A E D C B A = 96
	A ***C*** D ***E*** B A = 118		
	A E ***B*** C ***D*** A = 129		
	A ***D*** ***E*** C B A = 128		
	A E ***C*** ***D*** B A = 113		
	A E D ***B*** ***C*** A = 102		

Die beste Lösung der Nachbarschaft ist die Rundreise (A E D B C A) mit einer Länge von 102. Tabu-Search würde sie auswählen, allerdings enthält diese Lösung die Tabu-Attribute "'Knoten B und C getauscht"'. Daher wird die nächstbeste Lösung gewählt. Die Rundreisen (A B D C E A) und (A E C D B A) haben beide die Länge 103, dieses Tabu-Search wählt die erste aus. (A B D C E A) ist die neue Lösung, in die Tabu-Liste wird der Tausch der Knoten B und E aufgenommen und die neue Nachbarschaft wird berechnet.

Aktuelle Lösung	Nachbarschaft	Tabu-Liste	Beste Lösung
A B D C E A = 113	A ***E*** D C ***B*** A = 96	{B<->C	A E D C B A = 96
	A ***C*** D ***B*** E A = 119	B<->E}	
	A B ***E*** C ***D*** A = 145		
	A ***D*** ***B*** C E A = 129		
	A B ***C*** ***D*** E A = 96		
	A E D ***B*** ***C*** A = 118		

Wiederum gibt es zwei beste Lösungen, die Rundreisen (A E D C B A) und (A B C D E A) mit der Länge 96. Die erste Lösung enthält das Tabu-Attribut "'Knoten E und B getauscht"' daher wählt Tabu-Search die zweite Lösung. Diese ist identisch mit der bisher besten Lösung, nur in umgekehrter Richtung. Da dies der dritte Durchgang war ist unser Abbruchkriterium erfüllt und Tabu-Search terminiert.

8.7. Fazit

Das exakte Lösungsverfahren Branch & Bound bietet den Vorteil, garantiert das beste Ergebnis zu liefern, jedoch zu dem Preis einer längeren Laufzeit. Die Metaheuristik Tabu-Search kann in kurzer Zeit sehr gute Ergebnisse liefern, dafür ist jedoch die Implementierung sehr viel aufwendiger.
Auch eine Verknüpfung beider Verfahren ist möglich, so könnte Tabu-Search eine gute obere Schranke für Branch & Bound berechnen und dieses Verfahren dadurch stark beschleunigen.
Bevor jedoch eine klare Empfehlung gegeben werden kann, muss das tatsächliche Problem genauer eingegrenzt werden. Es ist zu erwarten, dass die bei der Modellierung eines Haushalts auftretenden Optimierungsaufgaben in einer für Optimierungsprobleme eher geringen Größenordnung liegen werden. Metaheuristiken werden normalerweise erst dann eingesetzt wenn die Problemgröße für die Berechnung einer exakten Lösung zu groß wird. Gleichzeitig ist aber die exakte Lösung für die Optimierung eines Geräteeinsatzplans nicht unbedingt erforderlich. Daher sollte in der Projektgruppe *Demonstrator InHouse-Energiemanagement* zuerst eine einfache Konstruktionsheuristik verwendet werden. Sollte sich diese als nicht ausreichend erweisen, kann sie im Nachhinein um eine Verbesserungsheuristik erweitert werden.

Literatur

[Amb05] AMBRASS, CHRISTIAN: *Implementierung und Evaluierung von Optimierungsverfahren am Beispiel der Zuordnung von ProseminarteilnehmerInnen im Softwareprojekt*. 2005.

[Far04] FARWER, BERNDT: *Formale Grundlagen der Informatik F3: Berechenbarkeit und Komplexität*, 2004. http://www.informatik.uni-hamburg.de/TGI/lehre/vl/WS0405/F3/F3-slides041214.pdf

[GR06] GÜNTHER RAIDL, ANDREAS CHWATAL: *Heuristische Optimierungsverfahren*, 2006. http://www.ads.tuwien.ac.at/teaching/ws/HeuOpt/folien.pdf

[Mer03] MERZ, DR. PETER: *Moderne heuristische Optimierungsverfahren: Meta-Heuristiken*, 2003. http://www-ra.informatik.uni-tuebingen.de/lehre/ss03

[SM07] STEPHAN MEHLHASE, JENS PFAU: *Evolutionsstrategien*, 2007. http://www.lernen-is-doof.de/files/einfuehrung.pdf

[WD06] WOLFGANG DOMSCHKE, ARMIN SCHOLL: *Heuristische Verfahren*. Arbeits- und Diskussionspapiere der Wirtschaftswissenschaftlichen Fakultät der Friedrich-Schiller-Universität Jena, 2006.

Teil III. Anforderungsdefinition

9. Problemstellung

Autor: Mark Eilers

Ziel der Projektgruppe "'Demonstrator InHouse-Energiemanagement"' ist die Entwicklung einer Software, die dem Anwender Möglichkeiten zum Senken von Stromkosten bei verschiedenen Tarifmodellen zeigt und den Anwender bei der Optimierung seines privaten Lastgangs unterstützt, um langfristig einen bewussteren Umgang mit dem eigenen Stromverbrauch zu schaffen.

Die zu entwickelnde Software soll dem Anwender die Möglichkeit bieten:

- Ein Profil seines Haushaltes durch eine Auswahl von elektrischen Geräten zu modellieren. Dabei werden dem Anwender neben einer Gerätebibliothek Werkzeuge und Hilfen zur Verfügung gestellt, um die in seinem Haushalt vorkommenden Geräte möglichst vollständig zu erfassen und, zum Beispiel durch die Verwendung eines effizienteren Geräts, zu optimieren.
- Stromtarife aus dem Internet zu beziehen oder eigene Stromtarife für die Software zu modellieren. Dabei sollen Standard-, Time-Of-Use- und Day-Ahead-Tarife unterstützt werden.
- Einsatzpläne für Geräte und sein Verbrauchsprofil zu erstellen. Diese Einsatzpläne sollen möglichst einfach modelliert werden und durch die Software hinsichtlich der Stromkosten, bezogen auf einen gewählten Stromtarif, optimiert werden.
- Die Auswirkungen von Änderungen, wie zum Beispiel der Austausch von elektrischen Geräten, oder das Ändern eines Einsatzplans, möglichst schnell und unkompliziert zu erfassen.
- Den Verbrauch verschiedener Geräte, Profile, Einsatzpläne, Tarife und auch selbst gemessene Lastgänge visuell miteinander und, soweit möglich, untereinander vergleichen zu können.

10. Annahmen

Autor: Moritz Eysholdt

Die Annahmen definieren domänenspezifische Begriffe und Datenstrukturen.

Gerät Ein Gerät ist ein beliebiger elektrischer Verbraucher. Es kann in Dateien gespeichert und aus ihnen geladen werden.

Attribute:

- Hersteller
- Produktbezeichnung
- Beschreibung
- Datensatz-Erstellungsdatum
- Gerätetyp: Ein hierarchischer Baum in welchen die Geräte einsortiert werden.
- Planbar: Wahr/Falsch. Planbare Geräte können vom Einsatzplan-Optimierungsalgorithmus eingeplant werden.
- Liste von Betriebsmodi
- Preis
- Anschaffungsdatum
- Baujahr
- Verifiziert: Wahr, wenn die Daten von einem Administrator überprüft wurden.
- Standby-Betriebsmodus: Dieser Betriebsmodus beschreibt die Leistungsaufnahme des Gerätes im Standby-Modus.

Betriebsmodus Ein Gerät kann beliebig viele Betriebsmodi haben. Ein Betriebsmodus beschreibt die charakteristische Leistungsaufnahme des Gerätes für eine bestimmte Gerätekonfiguration.

Attribute:

- Name
- Beschreibung
- Leistungsaufnahme: Die Leistungsaufnahme wird in Watt (W) angegeben und ist üblicher Weise der Durchschnitt des Gerätelastganges.
- Gerätelastgang: Der Lastgang beschreibt eine über die Zeit variierende Leistungsaufnahme des Gerätes als eine Menge von Messungen, wobei jede Messung aus dem gemessenen Stromverbrauch und der verstrichenen Zeit seit der ersten Messung besteht.
- Laufzeit: Wenn es für diesen Betriebsmodus eine charakteristische Zeitdauer gibt, nach welcher er abgeschlossen ist, so wird diese hier angegeben.

Einsatz Ein Einsatz (auch: Geräteeinsatz, Task) beschreibt die konkrete Verwendung eines Gerätes oder einer Gerätekomposition in einem Einsatzplan oder einer Gerätekomposition.

Attribute:

- Startzeit: Der Zeitpunkt des Aktivierens (Einschaltens) eines Gerätes relativ zum Anfang des Einsatzplanes oder der Gerätekomposition
- Einsatzdauer: Bei nicht gegebener Zyklusdefinition ist das Gerät während der gesamten Einsatzdauer aktiviert und wechselt danach in den Standardbetriebsmodus. Bei gegebener Zyklusdefinition beschreibt diese, wann ein Gerät während der Einsatzdauer aktiviert ist und wann nicht. Nach verstreichen der Einsatzdauer wechselt das Gerät auch hier in den Standardbetriebsmodus.
- Betriebsmodus oder Gerätekomposition: Referenziert den Betriebsmodus des eingesetzten Gerätes oder die eingesetzte Gerätekomposition.
- Zyklusdefinition (optional)

Zyklusdefinition Eine Zyklusdefinition beschreibt das mehrmalige stattfinden eines Einsatzes, ähnlich wie ein Serienelement in einer modernen Terminverwaltung. Die Zyklusdefinition ist wie folgt zu verstehen: Zur Startzeit (definiert in 'Einsatz') wird das Gerät für genau 'Betriebsdauer' Zeiteinheiten aktiviert. Danach verstreichen 'Wartezeit' Zeiteinheiten, worauf folgend das Gerät wieder für 'Betriebsdauer' Zeiteinheiten aktiviert wird. Dieser Vorgang wiederholt sich bis 'Einsatzdauer' Zeiteinheiten verstrichen sind. Constraints können so nicht den einzelnen Geräteaktivierungen zugeordnet werden, sondern nur dem Geräteeinsatz als Ganzen. Sind Constraints für einzelne Geräteaktivierungen erforderlich, so muss das Gerät in einer Gerätekomposition zusammen mit den benötigten Constraints abgelegt und diese Gerätekomposition zyklisch aufgerufen werden.

Attribute:

- Betriebsdauer
- Wartezeit

Einsatzplan Ein Einsatzplan beschreibt die Verwendung von Geräten und Gerätekompositionen in Abhängigkeit von der Zeit und zueinander in Form von Einsätzen. Durch einen dem Einsatzplan zugeordneten Tarif können die Stromkosten berechnet werden. Ein Einsatzplan kann in Dateien gespeichert und aus ihnen geladen werden. Ein Einsatzplan kann ausgedruckt werden.

Attribute:

- Name

- Beschreibung
- Datensatz-Erstellungsdatum
- Liste von Einsätzen
- Liste von Constraints
- Tarif
- Startzeit
- Dauer

Gerätekomposition Eine Gerätekomposition beschreibt wie ein Einsatzplan die Verwendung von Geräten und anderen Gerätekompositionen in Abhängigkeit von der Zeit, kennt aber keinen Tarif und keine fixen Start- und Endzeiten. Eine Gerätekomposition kann wie ein Gerät als planbares Element in einen Einsatzplan oder eine andere Gerätekomposition integriert werden. Eine Gerätekomposition kann in Dateien gespeichert und aus ihnen geladen werden.
Eine Gerätekomposition kann als Gruppe von Einsätzen und Constraints verstanden werden. Sie bietet dem Anwender die Möglichkeit, zusammengehörige Einsätze zu gliedern. Beispielsweise kann der Anwender Szenarien modellieren, in welchen mehrere Geräte involviert sind: Spielen an der Konsole würde aus Fernseher, Spielekonsole und Stereoanlage zusammengesetzt werden. Ebenso wäre es denkbar alle Geräte eines Zimmers in einer Gerätekomposition zusammenzufassen.

Attribute:

- Name
- Beschreibung
- Datensatz-Erstellungsdatum
- Liste von Einsätzen
- Liste von Constraints

Tarif Ein Tarif definiert einen vom Energielieferanten festgelegten Strompreis. Dieser kann von der Zeit abhängig sein. Ein Tarif kann in Dateien gespeichert und aus ihnen geladen werden.

Attribute:

- Tarifname
- Beschreibung
- Datensatz-Erstellungsdatum
- Anbietername

- Start- und Endzeitpunkt der Gültigkeitsdauer
- Strompreis in Abhängigkeit von der Zeit, dargestellt als Treppenfunktion.

Variabler Tarif Als variabler Tarif wird ein Tarif bezeichnet, welcher zu unterschiedlichen Zeiten unterschiedliche Strompreise definiert.

Fixtarif Im Gegensatz zum variablen Tarif sind beim Fixtarif die Stromkosten zu jedem Zeitpunkt gleich. Die Treppenfunktion des Strompreises ist hier eine waagerechte Gerade.

Constraint Ein Constraint ist eine Regel, welche eine Aussage über einen gültigen Einsatzzeitpunkt eines Gerätes oder mehrerer Geräte trifft.

Mögliche Constrainttypen sind:

- Zeitintervall: Es wird eine früheste Startzeit und eine späteste Endzeit definiert. Innerhalb dieses Zeitintervalls kann der Einsatz zu einem beliebigen Zeitpunkt stattfinden.
- Ein Einsatz A muss nach einem Einsatz B stattfinden: Der Einsatz B muss abgeschlossen sein, bevor Einsatz A beginnen kann.
- Ein Einsatz A muss zeitgleich mit einem Einsatz B laufen: Einsatz A und B müssen die selben Start- und Endzeiten haben.
- Die maximale Last ist begrenzt: Eine ausgewählte Gruppe von Einsätzen darf zusammen zu keinem Zeitpunkt einen bestimmten maximalen Stromverbrauch überschreiten.
- Zwei Einsätze dürfen nicht gleichzeitig stattfinden.
- Ein Einsatz A darf nicht starten oder enden, während Einsatz B läuft.

Lastgang Ein Lastgang beschreibt einen Stromverbrauch in Abhängigkeit von der Zeit. Er kann beispielsweise den Stromfluss durch eine Steckdosenleiste oder durch den hauszentralen Stromzähler über einen Zeitraum hinweg wiedergeben. Der hier beschriebene Lastgang ist unabhängig von dem zu einem Betriebsmodus gehörenden Gerätelastgang. Ein Lastgang kann in Dateien gespeichert und aus ihnen geladen werden.

Attribute:

- Name
- Beschreibung
- Datensatz-Erstellungsdatum
- Startzeitpunkt
- Dauer

- Messungen: Eine Liste von Messungen, wobei jede Messung aus dem gemessenen Stromverbrauch seit der letzten Messung und dem Zeitpunkt relativ zum Startzeitpunkt besteht.

Profil Ein Profil repräsentiert eine Unterauswahl von Geräten, Tarifen, Gerätekompositionen und Lastgängen. Der Anwender kann mit einem Profil beispielsweise alle Geräte eines Haushaltes oder Zimmers gruppieren. Ein Profil kann in Dateien gespeichert und aus ihnen geladen werden.

Attribute:

- Name
- Beschreibung
- Datensatz-Erstellungsdatum
- Liste von Geräten: Referenzen auf die lokalen Geräte.
- Liste von Tarifen: Referenzen auf die lokalen Tarife.
- Liste von Gerätekompositionen: Referenzen auf die Gerätekompositionen.
- Liste von Lastgängen: Referenzen auf die Lastgänge.
- Liste von Einsatzplänen

Online-Bibliothek Die Online-Bibliothek ist ein Online-Service, welcher eine Liste von Geräten, eine Liste von Tarifen oder beides vorrätig hält.

Attribute:

- Liste von Geräten
- Liste von Tarifen

Lokale Bibliothek Die lokale Bibliothek enthält eine vollständige Kopie der gewählten Online-Bibliothek. Geräte und Tarife in der lokalen Bibliothek können vom Anwender nicht direkt modifiziert werden. Er muss einzelne Geräte oder Tarife in seine lokale Gerätesammlung oder lokale Tarifsammlung kopieren, um sie bearbeiten zu können.

Attribute:

- Liste von Geräten
- Liste von Tarifen

Lokale Gerätesammlung Die lokale Gerätesammlung ist eine Liste aller Gerätekompositionen und Geräte, die der Anwender lokal erstellt oder aus der Bibliothek kopiert hat. Der Anwender kann jede Eigenschaft der Geräte und Gerätekompositionen dieser Liste bearbeiten.

Attribute:

- Liste von Geräten
- Liste von Gerätekompositionen

Lokale Tarifsammlung Die lokale Tarifsammlung ist eine Liste aller Tarife, die der Anwender lokal erstellt oder aus der Bibliothek kopiert hat. Der Anwender kann jede Eigenschaft der Tarife dieser Liste bearbeiten.

Attribute:

- Liste von Tarifen

Lokale Lastgangsammlung Die lokale Lastgangsammlung ist eine Liste von Lastgängen, die der Anwender selbst erstellt oder importiert hat.

Attribute:

- Liste von Lastgängen

Anwenderrollen Im Kontext dieser Anwendung wird zwischen zwei verschiedenen Rollen unterschieden:

- Anwender: Als Anwender wird der Benutzer der Desktopanwendung bezeichnet.
- Administrator: Als Administrator wird der Benutzer der Weboberfläche der Online-Datenbank bezeichnet, da diese für rein administrative Zwecke konzipiert ist.

11. Komponenten

Autor: Michael Erlmann

Die zu entwickelnde Software wird in die sieben hauptsächlichen Bereiche 'Lokale Gerätesammlung', 'Lokale Tarifsammlung', 'Lokale Lastgangsammlung', 'Profile', 'Ablaufplanung', 'Online-Bibliotheken' und 'Server'(vergleiche Abbildung 51) geteilt. Dies geschieht, um die unterschiedlichen Ansätze der einzelnen Bereiche voneinander zu trennen und ihre Rollen genau beleuchten zu können. Die Bereiche werden im Folgenden einzeln vorgestellt.

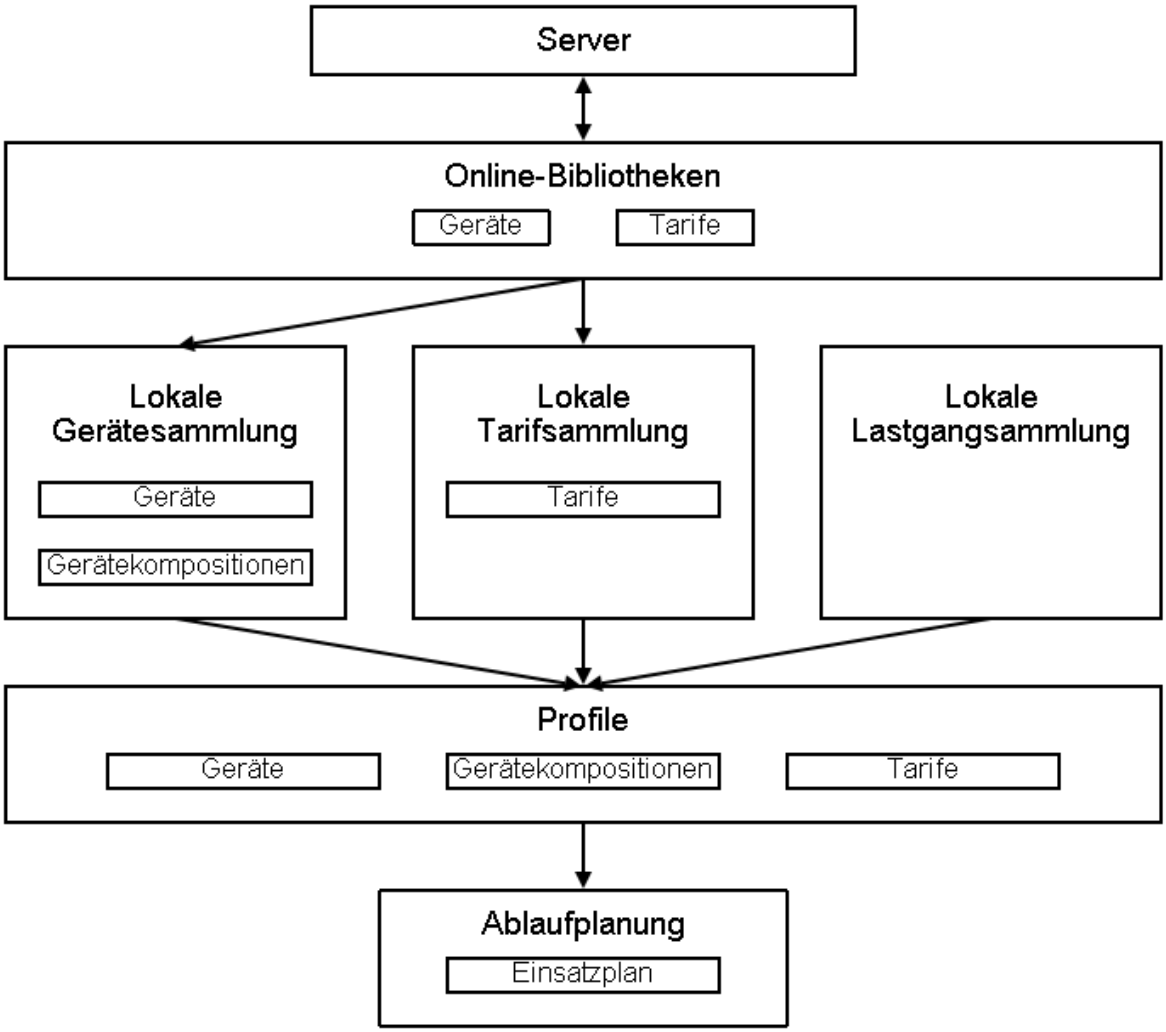

Abbildung 51: Komponenten

11.1. Gerätesammlung

Die lokale Gerätesammlung beinhaltet all jene Geräte, welche zum Planungsbereich des Anwenders, üblicherweise ein Haushalt, hinzugehören. Sie liegen in der lokalen Gerätesammlung als Instanzen und können hier nur heraus gelöscht werden, wenn sie von keinem Profil referenziert werden. Der Anwender kann beliebige Geräte aus der lokalen Gerätebibliothek in die lokale Gerätesammlung überführen und auch ganz neue Geräte erstellen und zu diesen Betriebsmodi erzeugen sowie bei bestehenden Geräten diese verändern.

Zusätzlich zu den Geräten können in der lokalen Gerätesammlung auch Gerätekompositionen gespeichert werden. Eine Gerätekomposition verbindet mehrere Betriebsmodi von unterschiedlichen Geräten anhand einer Menge von Constraints zu einer Gruppe, damit diese als ein zusammenhängendes planbares Element verwendet werden können.

11.2. Tarifsammlung

In der lokalen Tarifsammlung werden die verschiedenen Tarife gehalten. Der Anwender kann Tarife aus externen Quellen in die Tarifsammlung übernehmen, wie zum Beispiel aus der lokalen Tarifbibliothek.

Die Tarifsammlung kann drei Typen von Tarifen beinhalten: Standardtarif, Time-Of-Use-Tarif sowie Day-Ahead-Tarif.

11.3. Lastgangsammlung

In der lokalen Lastgangsammlung werden vom Anwender diejenigen Lastgänge gespeichert, welche mehrere Einsätze zugleich abdecken. Dabei kann es sich um den gesamten Lastgang des Haushaltes handeln, oder zum Beispiel um eine betrachtete Steckerleiste mit den daran angeschlossenen Geräten. Diese Lastgänge werden verwendet, um den im Profil erwarteten geplanten Verbrauch mit dem tatsächlichen Verbrauch vergleichen zu können. Man muss jedoch zwischen diesen Lastgängen und den Gerätelastgängen unterscheiden, welche in den Betriebsmodi direkt gespeichert werden und eine Durchführung des jeweiligen Betriebsmodus beschreiben. Lastgänge in der lokalen Lastgangsammlung sind nur löschbar, wenn sie von keinem Profil referenziert werden.

11.4. Profile

Die Modellierung seines Haushaltes unternimmt der Anwender anhand eines Profiles. Dabei kann er Profile beliebig definieren und zum Beispiel ein Profil nur für die Küche sowie eins für das Wohnzimmer anlegen, gleichzeitig aber auch eins erstellen, in dem Wohnzimmer und Küche innerhalb des gesamten Haushaltes gleichzeitig betrachtet werden. Ebenso kann er Profile mit Arbeiten verbinden und Geräte gruppie-

ren, welche nicht im gleichen Raum stehen, sondern zu dieser Arbeit gehören, aber auf verschiedene Räume verteilt sind.

In einem Profil sind alle Teile referenziert, welche man zum Erstellen eines Einsatzplanes benötigt. Demnach enthält das Profil sowohl Geräte und Gerätekompositionen, als auch Tarife. Weiter sind die Einsatzpläne selbst im Profil enthalten. Das Löschen von Geräten und Gerätekompositionen aus Profilen heraus ist nur dann möglich, wenn sie in keinem Einsatzplan innerhalb des Profiles mehr enthalten sind. Weiter besitzen Profile Referenzen auf Lastgänge aus der lokalen Lastgangsammlung, um diese mit den ermittelten Lastgängen der Einsatzpläne vergleichen zu können.

11.5. Ablaufplanung

Mit dem Erzeugen von Einsatzplänen, dem Optimieren und dem Vergleich von Plänen und auch Geräten beschäftigt sich die Komponente Ablaufplanung. Zusätzlich wird auch das Bilden von Gerätekompositionen wegen ihrer technischen Nähe zu den Einsatzplänen in der Ablaufplanung betrachtet.

Einsatzpläne werden aus einem Profil heraus erzeugt. Der Einsatzplan enthält Einsätze, bestehend aus Geräten und Gerätekompositionen verbunden mit ihren Einsatzzeiten, sowie Tarife. Dabei ist jedem Zeitpunkt im Einsatzplan nur ein Tarif zugeordnet. Sofern ein Standard- oder Time-Of-Use-Tarif verwendet wird enthält ein Einsatzplan somit insgesamt nur einen Tarif. Beim verwenden von Day-Ahead-Tarifen jedoch können mehrere Tarife unterschiedlicher Zeiten eingefügt werden, welche dann für die entsprechenden Zeitintervalle gültig sind. Alle in den Einsätzen vorhandenen Geräte und Gerätekompositionen sind immer in den zugehörigen Profilen und in der lokalen Gerätesammlung enthalten. Das gleiche gilt für die Tarife, welche zum Berechnen der Kosten verwendet werden sollen.

Die Optimierung von Plänen kann auf zwei Arten geschehen. Zum einen hat der Anwender die Möglichkeit den Einsatzplan direkt zu modifizieren. Zum anderen kann der Anwender einen solchen Plan auch automatisch generieren lassen. Dabei wird anhand einer Heuristik eine Optimierung des vom Anwender erzeugten Ausgangsplanes durchgeführt. Hierbei werden alle als planbar definierten Geräte unter Beachtung aller Constraints an jene Stellen verschoben, an denen sie die geringsten Kosten verursachen. Die automatische Optimierung von Plänen ist nur möglich, wenn für den generierten Zeitraum Tarife im Einsatzplan vorhanden sind.

Pläne kann man grundsätzlich mit zwei Dingen vergleichen. Zum einen kann man den Verbrauch eines Planes mit dem tatsächlichen Verbrauch in der selben Zeitspanne mittels eines Lastgangs aus der lokalen Lastgangsammlung vergleichen. Dies bietet die Möglichkeit im Haushalt versteckte Verbraucher ausfindig zu machen. Zusätzlich kann man einen Plan noch mit einem anderen Plan, welcher den gleichen Zeitraum abdeckt, vergleichen. Dabei gibt es zwei Zielfunktionen. Das eine ist der Vergleich der Pläne auf

verursachte Kosten, wobei derjenige Plan der bessere ist, welcher günstiger ist. Das andere ist ein visueller Vergleich der Pläne anhand der Start- und Endzeiten der Einsätze. Hierbei betrachtet der Anwender den Plan anhand seines Komforts und muss dann selber wissen, ob er bereit ist, zum Beispiel um vier Uhr morgens die Waschmaschine zu bedienen.

Zusätzlich bietet die Ablaufplanung auch den Vergleich von Geräten. Neben dem direkten Vergleich zweier Geräte anhand ihrer Eigenarten kann durch den Austausch nur eines Gerätes in einem fertigen Plan durch ein Gerät ähnlicher oder gleicher Funktion ein Nachbarplan erzeugt werden, welcher dann wie oben beschrieben optimiert und mit dem ursprünglichen Plan verglichen werden kann. Dabei kann ein Gerät sowohl den Preis beeinflussen als auch durch flexiblere Einsatzzeiten zu einem angenehmeren Plan anhand des Komforts führen.

Der Anwender hat die Möglichkeit Gerätekompositionen zu erstellen. Da die Gerätekomposition im Profil nur referenziert wird, wird bei der Erzeugung im Profil dort nur eine Referenz integriert und die Komposition in der lokalen Gerätesammlung als Instanz erzeugt. Beim Hinzufügen von Geräten in die Gerätekomposition wird ein bestimmter Betriebsmodus gewählt, welcher verwendet wird. Falls alle Betriebsmodi in der Komposition eine variable Länge haben, so ist die ganze Gerätekomposition variabel. Sofern jedoch mindestens ein Betriebsmodus eine fest definierte Länge hat, so hat die Gerätekomposition eine Mindestlänge, welche von den auf die Komposition gesetzten Constraints bestimmt wird.

11.6. Bibliotheken

Zusätzlich zur Datenhaltung in der lokalen Gerätesammlung und Tarifsammlung gibt es Online-Bibliotheken, welche Geräte und Tarife für den Anwender bereit halten. Dabei handelt es sich um einen Pool an Standard-Geräten und -Tarifen, die durschnittliche Haushalte ansatzweise darstellen können sollen. Die Online-Bibliotheken liegen auf einem zentralen Server und können vom Anwender abgefragt werden. Dabei werden dann die beiden Bibliotheken komplett auf den Rechner des Anwenders übertragen und liegen dort als lokale Instanz, auf welche der Anwender auch dann zugreifen kann, wenn er nicht mit dem Internet verbunden ist. Diese Instanzen kann der Anwender wie die lokalen Geräte- und Tarifsammlungen verwenden, jedoch kann er keine Veränderungen daran vornehmen.

11.7. Server

Der Server hält die Online-Bibliotheken bereit. Diese können durch den Anwender vom Server auf seinen eigenen Rechner herunter geladen werden. Ein Administrator kann zusätzlich Geräte und Tarife auf den Server herauf laden. Der Server besitzt eine Weboberfläche sowie eine Schnittstelle zur Kommunikation mit der Software.

12. Funktionale Anforderungen

Zu denen im vorherigen Abschnitt beschriebenen Komponenten werden die Funktionalen Anforderungen in Form von Anwendungsfällen beschrieben. Für die Anwendungsfälle 'löschen' ist vorgesehen, dass bevor tatsächlich gelöscht wird, eine Sicherheitsabfrage erfolgt. Nach dem der Anwender dieser zustimmt, wird die Löschung durchgeführt. Vor einer Löschung wird vom System das Element auf bestehende Referenzen geprüft. Falls Referenzen bestehen wird der Anwender darauf aufmerksam gemacht, er muss diese dann manuell lösen.

12.1. Gerätesammlung

Autor: Malte Hurrelmann

Die lokale Gerätesammlung enthält alle Geräte, die vom Anwender selbst angelegt wurden. Hierbei kann es sich um ein komplett neues Gerät handeln oder um eine modifizierte Kopie eines bestehenden Gerätes aus der Gerätebibliothek.

12.1.1. Geräte verwalten

Alle vom Nutzer angelegten Geräte sind im Unterpunkt Gerätesammlung vorhanden und können hier eingesehen werden. Im Gegensatz zur Online-Gerätebibliothek sind alle in der Gerätesammlung vorhandenen Geräte (Attribute wie Name, Leistungsaufnahme, Baujahr et cetera) über eine Eingabemaske editierbar. Weiterhin können hier Geräte hinzugefügt und gelöscht werden.

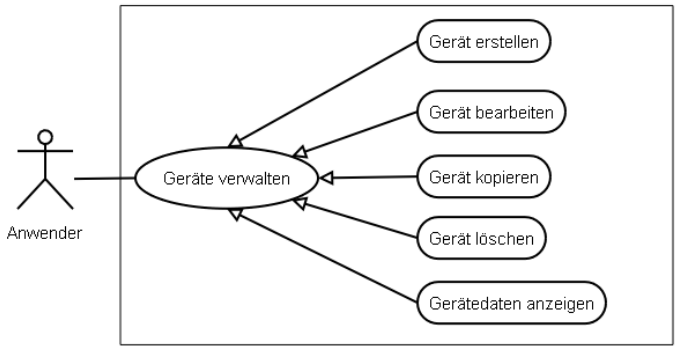

Abbildung 52: Anwendungsfälle Geräteverwalten

a)	Name:	Gerät erstellen
	Vorbedingung:	Keine.
	Beschreibung:	Über eine Eingabemaske werden die Geräteeigenschaften eingetragen und in der Gerätesammlung gespeichert.
	Nachbedingung:	Das Gerät ist in der Gerätesammlung vorhanden.

b)	Name:	Gerät bearbeiten
	Vorbedingung:	Mindestens ein Gerät muss in der lokalen Gerätesammlung vorhanden sein.
	Beschreibung:	Die Eigenschaften des Gerätes werden angezeigt und können durch den Anwender beliebig verändert werden.
	Nachbedingung:	Das Gerät mit allen Eigenschaften wurde in der lokalen Gerätesammlung abgespeichert.

c)	Name:	Gerät kopieren
	Vorbedingung:	Mindestens ein Gerät muss in der lokalen Gerätesammlung oder der Gerätebibliothek vorhanden sein.
	Beschreibung:	Der Anwender wählt ein Gerät aus der lokalen Gerätedatenbank oder der lokalen Gerätesammlung aus und kopiert sie in seine eigenen Gerätesammlung.
	Nachbedingung:	Das kopierte Gerät ist in der Gerätesammlung vorhanden.

d)	Name:	Gerät löschen
	Vorbedingung:	Ein Gerät muss in der lokalen Gerätesammlung vorhanden sein und darf von keinem Profil referenziert werden. Der Anwender wählt das zu löschende Gerät aus.
	Beschreibung:	Das gewählte Gerät wird aus der lokalen Gerätesammlung gelöscht.
	Nachbedingung:	Das Gerät ist nicht mehr in der lokalen Gerätesammlung vorhanden.

e)	Name:	Gerätedaten anzeigen
	Vorbedingung:	Es ist ein Gerät vorhanden.
	Beschreibung:	Es werden alle verfügbaren Daten zum Gerät angezeigt.
	Nachbedingung:	Keine.

12.1.2. Betriebsmodi verwalten

Der Anwender kann für ein Gerät beliebig viele Betriebsmodi definieren.

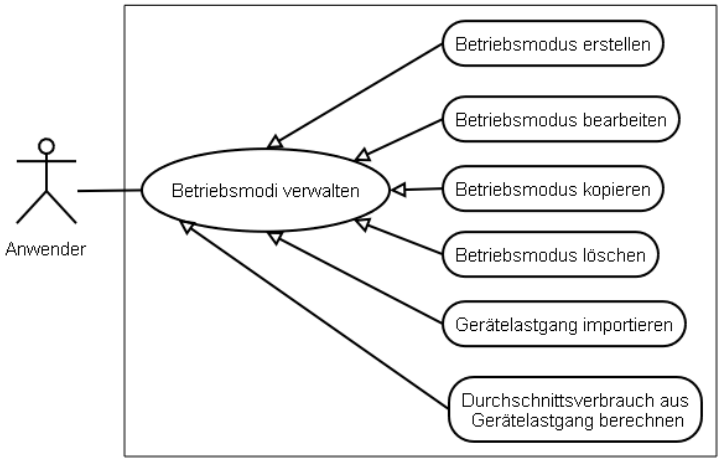

Abbildung 53: Anwendungsfälle Betriebsmodi verwalten

a)	Name:	Betriebsmodus erstellen
	Vorbedingung:	Gerät wurden aus der lokalen Gerätesammlung ausgewählt.
	Beschreibung:	Der Betriebsmodus für dieses Gerät wird über eine Eingabemaske erstellt.
	Nachbedingung:	Der neue Betriebsmodus ist dem Gerät zugeordnet worden.

b)	Name:	Betriebsmodus bearbeiten
	Vorbedingung:	Ein Gerät muss in der lokalen Gerätesammlung vorhanden sein und mindestens einen Betriebsmodus besitzen.
	Beschreibung:	Der gewählte Betriebsmodus wird bearbeitet.
	Nachbedingung:	Die geänderten Eigenschaften wurden gespeichert.

c)	Name:	Betriebsmodus kopieren
	Vorbedingung:	Ein Gerät muss in der lokalen Gerätesammlung vorhanden sein und mindestens einen Betriebsmodus besitzen.
	Beschreibung:	Betriebsmodi können innerhalb des Gerätes kopiert werden. Die Eigenschaften werden in die Eingabemaske zum Erstellen eines Betriebsmodus übertragen.
	Nachbedingung:	Der kopierte Betriebsmodus wurde gespeichert.

d)	Name:	Betriebsmodus löschen
	Vorbedingung:	Ein Betriebsmodus muss vorhanden sein und darf von keinem Einsatzplan referenziert werden.
	Beschreibung:	Ein Betriebsmodus wird aus dem Gerät gelöscht.
	Nachbedingung:	Der Betriebsmodus ist entfernt worden.

e)	Name:	Gerätelastgang importieren
	Vorbedingung:	Ein Betriebsmodus muss vorhanden sein.
	Beschreibung:	Der Lastgang wird im CSV-Format eingelesen oder manuell graphisch eingegeben.
	Nachbedingung:	Der Lastgang wurde als Eigenschaft des Betriebsmodus gespeichert. Der Durchschnittsverbrauch wird berechnet.

f)	Name:	Durchschnittsverbrauch aus Gerätelastgang berechnen
	Vorbedingung:	Ein Gerätelastgang ist vorhanden.
	Beschreibung:	Der Durchschnittsverbrauch wird aus dem eingelesenen Lastgang berechnet.
	Nachbedingung:	Der berechnete Durchschnittsverbrauch wird angezeigt und wird zum Speichern angeboten.

12.2. Tarifsammlung

Autor: Rachid Lacheheb

Der Anwender hat die Möglichkeit verschiedene Tarife zu nutzen (Fixtarife oder variable Tarife) um die Optimierung von Einsatzplänen zu steuern.

12.2.1. Tarife verwalten

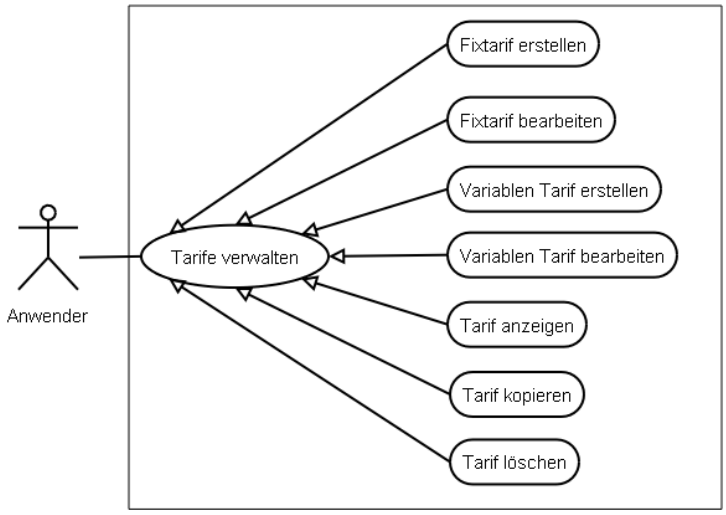

Abbildung 54: Anwendungsfälle Tarife verwalten

a)	Name:	Fixtarif erstellen
	Vorbedingung:	Keine.
	Beschreibung:	Ein Fixtarif wird vom Anwender mit konstantem kWh-Preis erstellt.
	Nachbedingung:	Der Tarif wurde gespeichert.

b)	Name:	Fixtarif bearbeiten
	Vorbedingung:	Ein Fixtarif muss in der Tarifsammlung vorhanden sein.
	Beschreibung:	Die Eigenschaften des Fixtarifs werden geändert.
	Nachbedingung:	Der Fixtarif mit allen geänderten Eigenschaften ist in der Tarifsammlung abgespeichert worden.

c)	Name:	Variablen Tarif erstellen
	Vorbedingung:	Keine.
	Beschreibung:	Der Anwender gibt einen variablen Tarif ein.
	Nachbedingung:	Der Tarif wurde gespeichert.

d)	Name:	Variablen Tarif bearbeiten
	Vorbedingung:	Ein variabler Tarif muss in der Tarifsammlung vorhanden sein.
	Beschreibung:	Die Eigenschaften des variablen Tarifes werden geändert.
	Nachbedingung:	Der variable Tarif mit allen geänderten Eigenschaften wurde in der Tarifsammlung abgespeichert.

e)	Name:	Tarif anzeigen
	Vorbedingung:	Ein Tarif ist vorhanden.
	Beschreibung:	Es werden die Eigenschaften des Tarifes angezeigt.
	Nachbedingung:	Keine.

f)	Name:	Tarif kopieren
	Vorbedingung:	Ein Tarif muss in der Tarifsammlung oder der Tarifbibliothek vorhanden sein.
	Beschreibung:	Der Anwender wählt einen Tarif aus der Tarifsammlung oder der Tarifbibliothek und kopiert ihn.
	Nachbedingung:	Der Tarif wurde in der Tarifsammlung abgelegt.

g)	Name:	Tarif löschen
	Vorbedingung:	Ein Tarif muss in der lokalen Tarifsammlung vorhanden sein und darf von keinem Profil referenziert werden. Der Anwender wählt den zu löschenden Tarif aus.
	Beschreibung:	Der gewählte Tarif wird aus der lokalen Tarifsammlung gelöscht.
	Nachbedingung:	Der Tarif ist nicht mehr in der lokalen Tarifsammlung vorhanden.

12.3. Lastgangsammlung

Autor: Mark Eilers

Die Lastgangsammlung enthält Lastgänge, die vom Anwender erstellt wurden, um sie mit von der Software berechneten Lastgängen vergleichen zu können.

12.3.1. Lastgänge verwalten

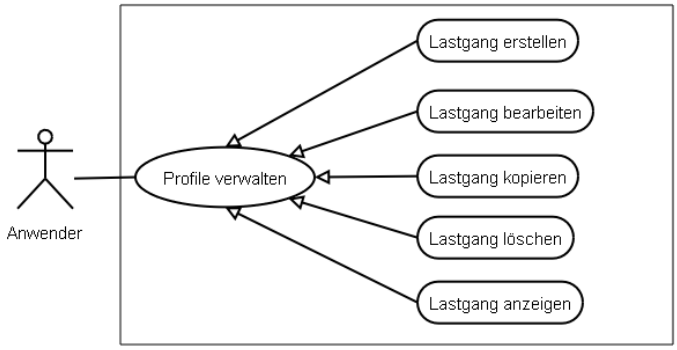

Abbildung 55: Anwendungsfälle Lastgang verwalten

a)	Name:	Lastgang erstellen
	Vorbedingung:	Keine.
	Beschreibung:	Ein Lastgang wird im CSV-Format eingelesen oder manuell eingegeben.
	Nachbedingung:	Der Lastgang wurde in der Lastgangsammlung gespeichert.

b)	Name:	Lastgang bearbeiten
	Vorbedingung:	Mindestens ein Lastgang muss in der Lastgangsammlung vorhanden sein.
	Beschreibung:	Der Lastgang und alle verfügbaren Daten werden angezeigt und können vom Anwender bearbeitet werden. Der Lastgang selbst kann graphisch bearbeitet werden oder im CSV-Format neu eingelesen werden.
	Nachbedingung:	Der Lastgang wurde gespeichert.

c)	Name:	Lastgang kopieren
	Vorbedingung:	Ein Lastgang muss in der Lastgangsammlung vorhanden sein.
	Beschreibung:	Der Anwender wählt einen Lastgang aus der Lastgangsammlung aus und kopiert ihn.
	Nachbedingung:	Der Lastgang wurde in der Lastgangsammlung abgelegt.

d)	Name:	Lastgang löschen
	Vorbedingung:	Ein Lastgang muss in der Lastgangsammlung vorhanden sein und darf von keinem Profil referenziert werden. Er muss vom Anwender ausgewählt werden.
	Beschreibung:	Der ausgewählte Lastgang wird aus der Lastgangsammlung entfernt.
	Nachbedingung:	Der Lastgang ist entfernt worden.

e)	Name:	Lastgang anzeigen
	Vorbedingung:	Es ist ein Lastgang vorhanden.
	Beschreibung:	Der Lastgang und alle verfügbaren Daten werden angezeigt.
	Nachbedingung:	Keine.

12.4. Profile

Autor: Michael Belasus

Über Profile kann der Anwender die Verwendung seiner Geräte innerhalb seines Haushaltes, einzelner Räume oder verschiedener Abläufe wie z. B. bestimmte Tage modellieren. Es können mehrere Profile vom Anwender angelegt werden. Jedes Profil wird unter einem eigenen Profilnamen gespeichert und enthält die innerhalb dieses Profils verwendeten Geräte und Gerätekompositionen aus der lokalen Gerätesammlung. Für das jeweilige Profil werden aus der lokalen Tarifsammlung die gewünschten Tarife ausgewählt.

Die Einrichtung eines Profils kann komplett manuell oder über einen Wizard erfolgen. Im Zusammenhang mit einem Wizard werden übliche Geräte abgefragt, sofern diese noch nicht eingegeben wurden, dadurch kann der Anwender versteckte Verbraucher innerhalb seines Haushaltes aufdecken.

12.4.1. Profile verwalten

Die Profilverwaltung umfasst, wie in Abbildung 56 dargestellt, das Anlegen, Auswählen, Bearbeiten, Kopieren und Löschen der Profile. Die Anwenderschnittstellen werden

sowohl über das Menü und als auch über ein Kontextmenü realisiert.

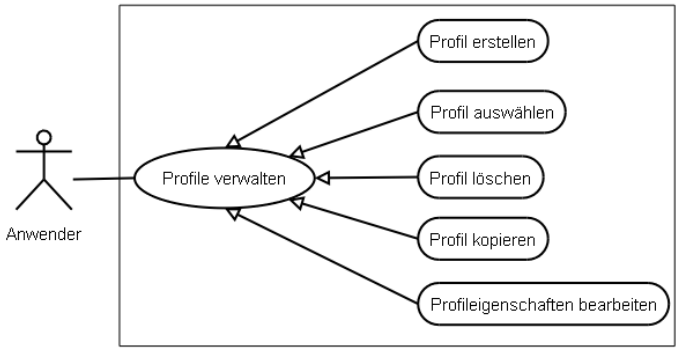

Abbildung 56: Anwendungsfälle Profile verwalten

a)	Name:	Profil erstellen
	Vorbedingung:	Ist kein Profil angelegt wird bei Programmstart das Anlegen eines Profils automatisch gestartet. Ist mindestens ein Profil angelegt, wird das zuletzt verwendete Profil geladen und der Anwender kann über das Menü ein neues Profil anlegen.
	Beschreibung:	Die Konfiguration des Haushalts wird mit Hilfe eines Wizards oder durch die Angabe eines Profilnamens manuell erstellt.
	Nachbedingung:	Das Profil ist im System enthalten.

b)	Name:	Profil laden
	Vorbedingung:	Mindestens ein Profil wurde vom Anwender angelegt.
	Beschreibung:	Bei Programmstart wird das zuletzt verwendete Profil geladen. Damit ein anderes Profil geladen werden kann, kann der Anwender über das Menü aus den erstellten Profilen das gewünschte Profil auswählen und öffnen.
	Nachbedingung:	Das Profil ist geladen.

c)	Name:	Profil löschen
	Vorbedingung:	Mindestens ein Profil wurde vom Anwender angelegt.
	Beschreibung:	Damit ein bereits angelegtes Profil gelöscht werden kann, kann der Anwender über das Menü aus den erstellten Profilen das gewünschte Profil auswählen und löschen.
	Nachbedingung:	Die zu dem Profil angelegten Referenzen auf die Geräte, Gerätekompositionen, Tarife und die Einsatzpläne werden ebenfalls gelöscht.

d)	Name:	Profil kopieren
	Vorbedingung:	Mindestens ein Profil wurde vom Anwender angelegt.
	Beschreibung:	Über das Menü wählt der Anwender das zu kopierende Profil aus, vergibt für das neue Profil einen neuen Namen und speichert es ab.
	Nachbedingung:	Das neue Profil enthält alle Referenzen auf die Geräte, Gerätekompositionen, Tarife und eine Kopie der Einsatzpläne wie das originale Profil.

e)	Name:	Profileigenschaften bearbeiten
	Vorbedingung:	Mindestens ein Profil wurde vom Anwender angelegt.
	Beschreibung:	Das entsprechende Profil wird über das Menü vom Anwender ausgewählt. Die Daten des Profils werden angezeigt. Der Anwender ändert die gewünschten Daten wie beispielsweise den Profilnamen und speichert das Profil.
	Nachbedingung:	Die Daten des Profils wurden gespeichert.

12.4.2. Profil bearbeiten

Profil bearbeiten beinhaltet, wie in Abbildung 57 dargestellt, die Anwendungsfälle, dass beliebig viele Geräte, Gerätekompositionen, Tarife und Lastgänge dem ausgewählten Profil hinzugefügt und daraus gelöscht werden können. Die Anwenderschnittstelle wird ebenfalls über ein Menü und ein Kontextmenü realisiert.

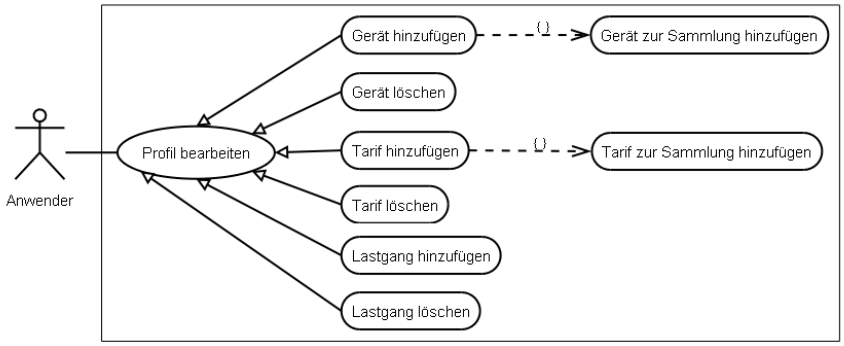

Abbildung 57: Anwendungsfälle Profil bearbeiten

a)	Name:	Gerät hinzufügen
	Vorbedingung:	Der Anwender hat ein Profil ausgewählt.
	Beschreibung:	Der Anwender kann Geräte direkt aus der lokalen Gerätebibliothek oder aus der lokalen Gerätesammlung in das Profil einfügen. Dies erfolgt über das Menü oder per Drag & Drop. Das Hinzufügen von Gerätekompositionen erfolgt ausschließlich aus der lokalen Gerätesammlung.
	Nachbedingung:	Die Referenzen auf die Geräte werden im Profil gespeichert. Erfolgt das Hinzufügen eines Gerätes aus der lokalen Gerätebibliothek so wird dieses Gerät gleichzeitig in der lokalen Gerätesammlung gespeichert.

b)	Name:	Gerät löschen
	Vorbedingung:	Der Anwender hat ein Profil ausgewählt. Dieses Profil enthält mindestens ein Gerät oder eine Gerätekomposition, welche in keinem Einsatzplan mehr vorhanden sind.
	Beschreibung:	Das zu löschende Gerät oder die Gerätekomposition werden ausgewählt und über das Menü gelöscht.
	Nachbedingung:	Die Referenz auf das Gerät oder die Gerätekomposition wird aus dem Profil entfernt.

c)	Name:	Tarif hinzufügen
	Vorbedingung:	Der Anwender hat ein Profil ausgewählt. Es wurde mindestens ein Tarif in der lokalen Tarifsammlung angelegt.
	Beschreibung:	Der Anwender kann Tarife direkt aus der lokalen Tarifbibliothek oder aus der lokalen Tarifsammlung in das Profil einfügen. Dies erfolgt über das Menü oder per Drag & Drop.
	Nachbedingung:	Die Referenzen auf die Tarife werden im Profil gespeichert. Erfolgt das Hinzufügen eines Tarifes aus der lokalen Tarifbibliothek so wird dieser Tarif gleichzeitig in der lokalen Tarifsammlung gespeichert.

d)	Name:	Tarif löschen
	Vorbedingung:	Der Anwender hat ein Profil ausgewählt. Es wurden mindestens ein Tarif dem ausgewählten Profil hinzugefügt.
	Beschreibung:	Der Anwender markiert den zu löschenden Tarif und entfernt diesen.
	Nachbedingung:	Die Referenz auf den Tarif wurde aus dem Profil entfernt.

e)	Name:	Lastgang hinzufügen
	Vorbedingung:	Der Anwender hat ein Profil ausgewählt. Es wurde mindestens ein Lastgang in der lokalen Lastgangsammlung gespeichert.
	Beschreibung:	Der Anwender fügt aus der lokalen Lastgangsammlung den gewünschten Lastgang über das Menü in das ausgewählte Profil hinzu.
	Nachbedingung:	Die Referenz auf den Lastgang wird im Profil gespeichert.

f)	Name:	Lastgang löschen
	Vorbedingung:	Der Anwender hat ein Profil ausgewählt. Es wurde mindestens ein Lastgang dem ausgewählten Profil hinzugefügt.
	Beschreibung:	Der Anwender markiert den zu löschenden Lastgang und entfernt diesen.
	Nachbedingung:	Die Referenz auf den Lastgang wird aus dem Profil entfernt. Die Änderung wird im Profil gespeichert.

12.5. Ablaufplanung

Autor: Christian Bley

Die Ablaufplanung bietet dem Anwender Funktionen, um die Nutzung seiner Geräte im Haushalt planen zu können. Dazu gehören der Zusammenschluss mehrerer Geräte zu einer Gerätekomposition, das Festlegen von Einschränkungen zu den Einsätzen im Einsatzplan und das Erstellen und Optimieren von Einsatzplänen. Zusätzlich bietet diese Komponente für den Anwender auch Vergleichsmöglichkeiten, um einen Überblick über die Effektivität der erstellten Einsatzpläne zu bekommen.

12.5.1. Constraints verwalten

Für Gerätekompositionen, Geräte und deren Betriebsmodi können Einschränkungen festgelegt werden, um die Planung und Optimierung von Einsatzplänen zu steuern.

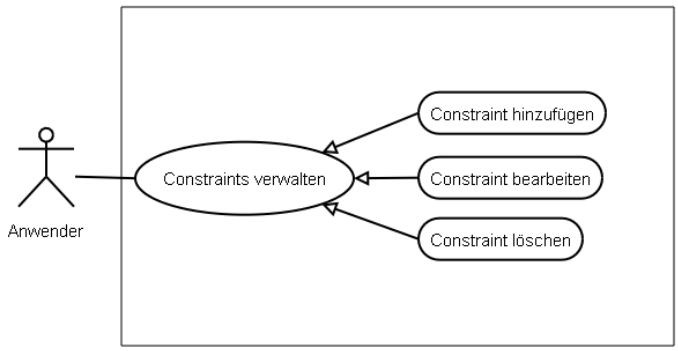

Abbildung 58: Anwendungsfälle Constraints verwalten

a)	Name:	Constraint hinzufügen
	Vorbedingung:	Einsätze müssen existieren.
	Beschreibung:	Der Anwender bestimmt eine Einschränkung für die Einsätze.
	Nachbedingung:	Der Constraint ist im System enthalten und eventuelle Kollisionen werden dem Anwender mitgeteilt.

b)	Name:	Constraint bearbeiten
	Vorbedingung:	Ein vorhandener Constraint muss vom Anwender ausgewählt werden.
	Beschreibung:	Der Anwender ändert die Einstellungen des Constraints.
	Nachbedingung:	Der geänderte Constraint wurde im System gespeichert. Kollisionen mit bestehenden Einstellungen werden dem Anwender mitgeteilt.

c)	Name:	Constraint löschen
	Vorbedingung:	Ein vorhandener Constraint muss vom Anwender ausgewählt werden.
	Beschreibung:	Der gewählte Constraint wird vom betreffenden Einsatz gelöscht.
	Nachbedingung:	Der Constraint wurde aus dem System entfernt.

12.5.2. Gerätekompositionen verwalten

Der Anwender kann mehrere Geräte als Gerätekompositionen zusammenfassen, um diese als Anwendungen im Einsatzplan zu verwenden. Eine Gerätekomposition kann dann wie ein einzelnes Gerät im Einsatzplan verwendet werden. Desweiteren kann er diese kopieren, bearbeiten und löschen.

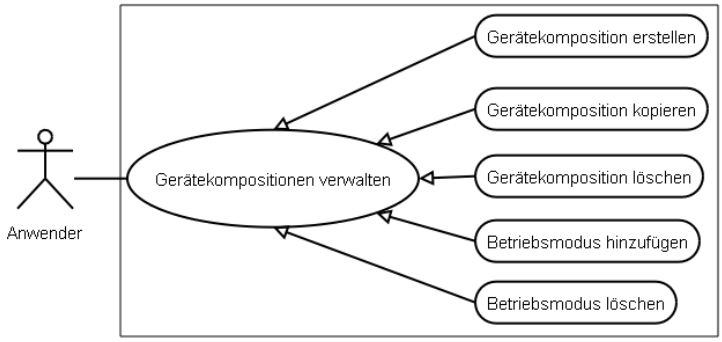

Abbildung 59: Anwendungsfälle Gerätekompositionen verwalten

a)	Name:	Gerätekomposition erstellen
	Vorbedingung:	Es müssen Geräte mit Betriebsmodi vorhanden sein.
	Beschreibung:	Der Anwender wählt die gewünschten Geräte und deren Betriebsmodi und fasst sie als Gruppe zusammen, die geschlossen angewendet werden soll, wofür eine Anwendungsdauer hinzugefügt werden kann.
	Nachbedingung:	Die neue Gerätekomposition ist in der lokalen Gerätesammlung enthalten.

b)	Name:	Gerätekomposition kopieren
	Vorbedingung:	Es muss eine Gerätekomposition ausgewählt werden.
	Beschreibung:	Die Gerätekomposition wird vom Anwender kopiert.
	Nachbedingung:	Eine Kopie der Gerätekomposition wurde in der lokalen Gerätesammlung gespeichert.

c)	Name:	Gerätekomposition löschen
	Vorbedingung:	Eine Gerätekomposition muss vorhanden sein. Der Anwender wählt die zu löschende Gerätekomposition aus.
	Beschreibung:	Das System löscht die gewählte Gerätekomposition, sofern keine Abhängigkeiten mehr bestehen.
	Nachbedingung:	Die gewählte Komposition wurde aus der lokalen Gerätesammlung entfernt.

d)	Name:	Betriebsmodus hinzufügen
	Vorbedingung:	Eine Gerätekomposition muss vorhanden sein. Der Anwender wählt den gewünschten Betriebsmodus aus.
	Beschreibung:	Ein gewählter Betriebsmodus wird zu einer vom Anwender bestimmten Gerätekomposition hinzugefügt.
	Nachbedingung:	Der Betriebsmodus wurde zu der Komposition hinzugefügt und die geänderte Komposition ist in der lokalen Gerätesammlung anstelle der alten Komposition enthalten. Alle bestehenden Einsätze dieser Gerätekomposition sind hiervon auch betroffen.

e)	Name:	Betriebsmodus löschen
	Vorbedingung:	Eine Gerätekomposition muss vorhanden sein und der Anwender muss einen Betriebsmodus hieraus auswählen.
	Beschreibung:	Der gewählte Betriebsmodus wird aus dieser Gerätekomposition gelöscht.
	Nachbedingung:	Der Betriebsmodus wurde aus der Komposition entfernt und die geänderte Komposition ist in der lokalen Gerätesammlung anstelle der alten Komposition enthalten. Alle bestehenden Einsätze dieser Gerätekomposition sind hiervon auch betroffen.

12.5.3. Einsatzpläne verwalten

Ein Einsatzplan bietet dem Anwender die Möglichkeit, den Ablauf seiner Geräte zu planen und zu beinflussen. Das Programm kann Einsatzpläne dahingehend optimieren, dass möglichst geringe Kosten entstehen. Desweiteren können für Einsätze, die regelmäßig geplant werden sollen, Zyklen erstellt werden. Diese fügen die Einsätze dann automatisch in die betroffenen Einsatzpläne ein.

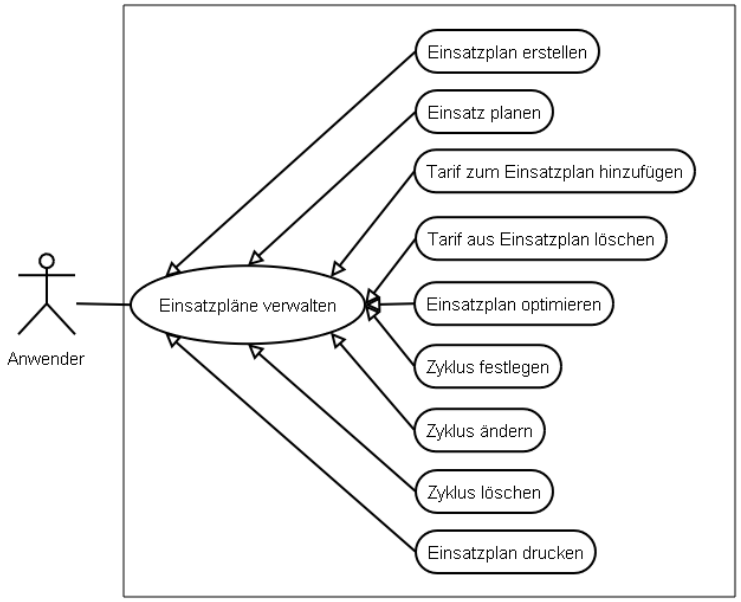

Abbildung 60: Anwendungsfälle Einsatzpläne verwalten

a)	Name:	Einsatzplan erstellen
	Vorbedingung:	Ein Profil muss existieren und es muss Geräte oder Gerätekompositionen enthalten.
	Beschreibung:	Der Anwender fügt die einzelnen Gerätekompositionen oder einzelne Betriebsmodi in den Einsatzplan ein, legt Start- und Endzeit des Einsatzes fest und kann weitere Einschränkungen zu den Komponenten erstellen (Siehe Bereich 12.5.1).
	Nachbedingung:	Der manuell erstellte Einsatzplan ist in dem gewählten Profil gespeichert. Die geplanten Geräteläufe sind als Einsätze in ihm enthalten.

b)	Name:	Einsatz planen
	Vorbedingung:	Ein Einsatzplan muss vorhanden sein und vom Anwender ausgewählt werden
	Beschreibung:	Das Zeitfenster für den Einsatz und der bevorzugte Zeitpunkt wird festgelegt oder geändert. Es wird der Betriebsmodus und der Folgemodus eingestellt. Bei Verletzung von Constraints wird der Anwender darauf aufmerksam gemacht.
	Nachbedingung:	Der gewählte Betriebsmodus wird in dem gewählten Zeitfenster zur Durchführung eingeplant und der geänderte Einsatzplan gespeichert

c)	Name:	Tarif zum Einsatzplan hinzufügen
	Vorbedingung:	Ein Einsatzplan muss vorhanden sein und vom Anwender ausgewählt werden. Der Anwender wählt einen vorhandenen Tarif aus.
	Beschreibung:	Der gewählte Tarif wird zu dem Einsatzplan hinzugefügt.
	Nachbedingung:	Der Einsatzplan enthält den gewählten Tarif.

d)	Name:	Tarif aus Einsatzplan löschen
	Vorbedingung:	Ein Einsatzplan mit Tarif muss vorhanden sein. Der Anwender muss den Tarif auswählen.
	Beschreibung:	Der Tarif wird aus dem Einsatzplan entfernt.
	Nachbedingung:	Der gewählte Tarif ist nicht mehr im Einsatzplan enthalten.

e)	Name:	Einsatzplan optimieren
	Vorbedingung:	Es muss ein Einsatzplan mit Tarif und Einsätzen existieren und der Anwender muss ihn auswählen. Der Anwender sollte die benötigten Constraints festgelegt haben.
	Beschreibung:	Der gewählte Einsatzplan wird durch eine Heuristik optimiert und an den aktuellen Tarif angepasst.
	Nachbedingung:	Der optimierte Einsatzplan hat den vorher ausgewählten Einsatzplan ersetzt.

f)	Name:	Zyklus festlegen
	Vorbedingung:	Es müssen Einsätze vorhanden sein. Der Anwender muss einen Einsatz auswählen.
	Beschreibung:	Für den gewählten Einsatz wird ein Zyklus erstellt, der diesen Einsatz zu festen Zeitpunkten in den betroffenen Einsatzplan automatisch einfügt.
	Nachbedingung:	Der erstellte Zyklus wurde gespeichert, und die betroffenen Elemente wurden in die entsprechenden Einsatzpläne eingefügt.

g)	Name:	Zyklus ändern
	Vorbedingung:	Es muss ein Zyklus existieren und vom Anwender ausgewählt werden.
	Beschreibung:	Der Anwender ändert den Zeitrahmen des Zyklus.
	Nachbedingung:	Das vom Zyklus betroffene Element wird aus dem vorher betroffenen Einsatzplan entfernt und an den neuen Zeitpunkten eingefügt.

h)	Name:	Zyklus löschen
	Vorbedingung:	Ein Zyklus muss existieren und er muss vom Anwender ausgewählt werden.
	Beschreibung:	Der gewählte Zyklus wird vom Anwender gelöscht.
	Nachbedingung:	Der Zyklus wurde aus dem Programm gelöscht.

i)	Name:	Einsatzplan drucken
	Vorbedingung:	Ein Einsatzplan muss existieren und ausgewählt werden. Ein Drucker muss am Rechner eingerichtet sein.
	Beschreibung:	Der gewählte Einsatzplan wird in eine druckbare Form exportiert und gedruckt.
	Nachbedingung:	Der Einsatzplan wird an den Drucker gesendet.

12.5.4. Einsatzpläne vergleichen

Dem Anwender werden Funktionen zur Verfügung gestellt, um Vergleiche und Einschätzungen auf Grund seiner Einsatzpläne durchführen zu können.

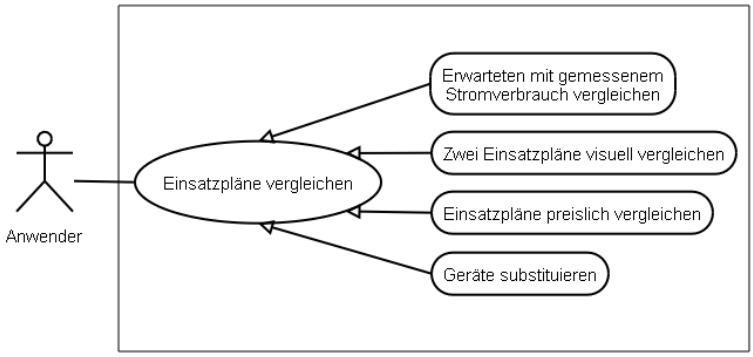

Abbildung 61: Anwendungsfälle Einsatzpläne vergleichen

a)	Name:	Erwarteten mit gemessenem Stromverbrauch vergleichen
	Vorbedingung:	Ein Einsatzplan muss vom Anwender komplett durchgeführt werden. Es muss ein Lastgang in der Lastgangsammlung existieren, der den Zeitraum des Einsatzplanes und die gleichen Geräte umfasst.
	Beschreibung:	Der Anwender trägt den realen Stromverbrauch zu einem durchgefühten Einsatzplan als Lastkurve oder Durchschnittswert ein.
	Nachbedingung:	Der Anwender wird auf Differenzen zwischen realen und errechneten Stromverbrauch aufmerksam gemacht. Wenn eine Lastkurve als realer Verbrauch angegeben wurde, werden reale und errechnete Kurve angezeigt, ansonsten gibt es einen Vergleich der absoluten Werte.

b)	Name:	Zwei Einsatzpläne visuell vergleichen
	Vorbedingung:	Zwei Einsatzpläne müssen existieren und ausgewählt werden.
	Beschreibung:	Die Strukturen der Einsatzpläne werden zum Vergleich übereinander gelegt und die errechneten Stromkosten angezeigt.
	Nachbedingung:	Es wird eine graphische Überlagerung angezeigt, womit der Anwender Unterschiede im Ablauf von Einsatzplänen erkennen und einschätzen kann.

c)	Name:	Einsatzpläne preislich vergleichen
	Vorbedingung:	Mindestens zwei Einsatzpläne müssen existieren und ausgewählt werden.
	Beschreibung:	Es werden mehrere Einsatzpläne ausgewählt und ein Vergleich der errechneten und realen Kosten durchgeführt.
	Nachbedingung:	Die Auswertung wird dem Anwender angezeigt.

d)	Name:	Geräte substituieren
	Vorbedingung:	Der Anwender muss einen Einsatzplan wählen.
	Beschreibung:	Der Anwender wählt ein Gerät aus, und es werden die möglichen Geräte mit gleicher Funktionalität zur Substitution angezeigt. Aus diesen wählt er das gewünschte aus. Das Gerät wird durch dieses ersetzt. Dies kann der Anwender für beliebige Geräte wiederholen. Bei Fertigstellung wird der Einsatzplan kopiert und mit diesen neuen Geräten substituiert und optimiert. Der ursprüngliche Einsatzplan bleibt erhalten. Anschließend werden die beiden Einsatzpläne visuell und preislich verglichen.
	Nachbedingung:	Der ursprüngliche und der neue Einsatzplan mit den substituierten Geräten sind im System enthalten. Der Anwender erhält einen visuellen und einen preislichen Vergleich der beiden Einsatzpläne.

12.6. Bibliotheken

Autor: Ulrich Denecke

Der Bereich Bibliotheken gliedert sich in vier Teile. Dies sind jeweils die online und lokalen Versionen der Gerätebibliothek und Tarifbibliothek. Ihre Funktionalitäten werden im folgenden Abschnitt beschrieben.

12.6.1. Online-Gerätebibliothek

Die Online-Gerätebibliothek umfasst eine Mischung aus verifizierten und nicht verifizierten Gerätedaten. Sie liegt auf einem Webserver und kann von dort vom Anwender mittels des Demonstrators ergänzt werden. Der entsprechende Anwendungsfall ist das Hinzufügen neuer Gerätedaten zur Online-Gerätebibliothek.

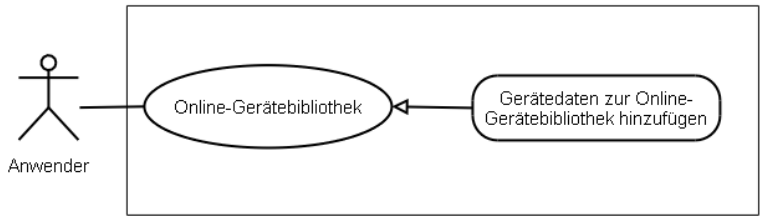

Abbildung 62: Anwendungsfall Online-Gerätebibliothek

a)	Name:	Gerätedaten zur Online-Gerätebibliothek hinzufügen
	Vorbedingung:	Der Anwender benutzt die Software auf einem Rechner mit korrekt konfiguriertem Internetzugang. Weiterhin wird mindestens ein Eintrag in der lokalen Gerätesammlung benötigt.
	Beschreibung:	Der Anwender wählt das Hinzufügen aller freigegebenen Einträge zur Online-Gerätebibliothek aus.
	Nachbedingung:	Die freigegebenen Datensätze aus der lokalen Gerätesammlung existieren in der Online-Gerätebibliothek.

12.6.2. Lokale Gerätebibliothek

Die lokale Gerätebibliothek besteht aus einer Kopie der Online-Gerätebibliothek. Sie ermöglicht das Einpflegen neuer, vorgegebener Geräte in die lokale Gerätesammlung auch im Offline-Betrieb. Der entsprechende Anwendungsfall ist das Beziehen der Gerätedaten aus dem Internet.

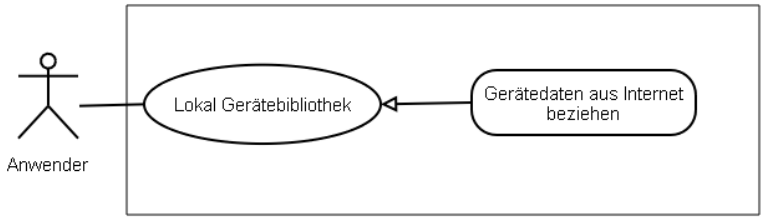

Abbildung 63: Anwendungsfall lokale Gerätebibliothek

a)	Name:	Gerätedaten aus Internet beziehen
	Vorbedingung:	Der Anwender benutzt die Software auf einem Rechner mit korrekt konfiguriertem Internetzugang.
	Beschreibung:	Der Anwender wählt das Beziehen der Online-Gerätebibliothek aus dem Internet aus.
	Nachbedingung:	Eine aktuelle Version der Online-Gerätebibliothek wurde in der Rubrik lokale Gerätebibliothek gesichert. Anschließend kann der Anwender Geräte aus der lokalen Bibliothek auswählen. Die vorherige, eventuell veraltete Version der lokalen Gerätebibliothek wurde überschrieben.

12.6.3. Online-Tarifbibliothek

Die Online-Tarifbibliothek stellt die Tarifdaten zum Download zur lokalen Tarifbibliothek zur Verfügung. Die einzige Interaktion des Anwenders ist das Beziehen der Tarifdaten aus dem Internet. Dieser Schritt wird im Bereich der lokalen Tarifbibliothek beschrieben. Da der Anwender jedoch keinen Einfluss auf diese Bibliothek nehmen kann, existiert in dieser Sektion kein Anwendungsfall. Ein Einstellen von Tarifdaten in die Online-Tarifbibliothek seitens des Anwenders macht keinen Sinn und wird daher nicht betrachtet.

12.6.4. Lokale Tarifbibliothek

Die lokale Tarifbibliothek hält nach dem Download der Online-Tarifbibliothek die aktuellen Tarifdaten zur Benutzung bereit. Der einzige Anwendungsfall für diese Sektion ist das Beziehen der aktuellen Tarifdaten aus dem Internet.

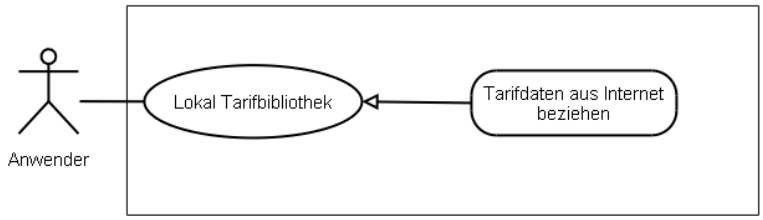

Abbildung 64: Anwendungsfall lokale Tarifbibliothek

a)	Name:	Tarifdaten aus dem Internet beziehen
	Vorbedingung:	Der Anwender benutzt die Software auf einem Rechner mit korrekt konfiguriertem Internetzugang.
	Beschreibung:	Der Anwender wählt das Beziehen der Online-Tarifbibliothek aus dem Internet aus.
	Nachbedingung:	Eine aktuelle Version der Online-Tarifbibliothek wurde in der Rubrik lokale Tarifbibliothek gesichert. Anschließend kann der Anwender Tarife aus der lokalen Bibliothek auswählen.

12.7. Server

Autor: Christian Bley

Der Server stellt die Online-Gerätebibliothek und die Online-Tarifbibliothek bereit. Diese können über eine Schnittstelle im Programm direkt vom Server bezogen werden, wie in den Abschnitten 12.6.2 und 12.6.4 beschrieben wird. Der Server bietet als Funktionalitäten die Verwaltung der Tarife und der Geräte innerhalb der Online-Bibliotheken. Er ist über eine einfache Weboberfläche mit dem Browser zu erreichen. Die Anwendungsfälle betreffen nur den Administrator.

12.7.1. Online-Gerätebibliothek verwalten

Über den Server können die in der Online-Gerätebibliothek gespeicherten Geräte mit ihren Betriebsmodi verwaltet werden.

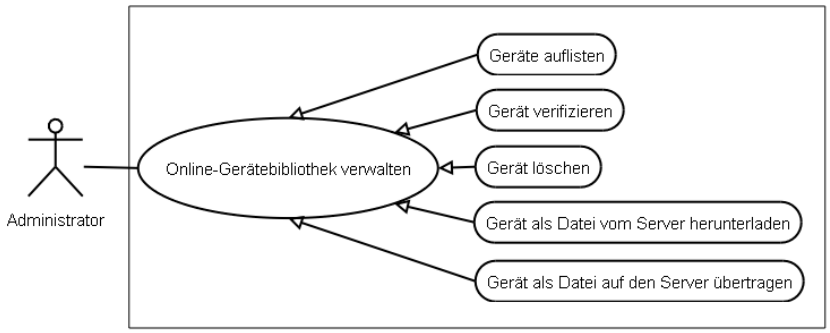

Abbildung 65: Anwendungsfälle Online-Gerätebibliothek verwalten

a)	Name:	Geräte auflisten
	Vorbedingung:	Es muss mindestens ein Gerät in der Online-Gerätebibliothek enthalten sein.
	Beschreibung:	Der Administrator kann die in der Online-Gerätebibliothek gespeicherten Geräte anzeigen lassen.
	Nachbedingung:	Es wird eine Liste der gespeicherten Geräte mit ihren Betriebsmodi angezeigt.

b)	Name:	Gerät verifizieren
	Vorbedingung:	Es muss mindestens ein Gerät in der Online-Gerätebibliothek enthalten sein. Der Administrator muss das Gerät aus der Geräteliste auswählen, das verifiziert werden soll.
	Beschreibung:	Wenn die Daten des Gerätes vertrauenswürdig sind, kann der Administrator dieses als vertrauenswürdig markieren und speichern.
	Nachbedingung:	Das Gerät wird in der Online-Gerätebibiliothek als vertrauenswürdig angezeigt.

c)	Name:	Gerät löschen
	Vorbedingung:	Es muss mindestens ein Gerät in der Online-Gerätebibliothek enthalten sein. Der Administrator muss das Gerät auswählen, das gelöscht werden soll.
	Beschreibung:	Das ausgewählte Gerät wird aus der Online-Gerätebibliothek entfernt.
	Nachbedingung:	Das gewählte Gerät ist nicht mehr in der Online-Gerätebibliothek enthalten.

d)	Name:	Gerät als Datei vom Server herunterladen
	Vorbedingung:	Es muss mindestens ein Gerät in der Online-Gerätebibliothek gespeichert sein. Der Administrator muss ein Gerät auswählen.
	Beschreibung:	Der Administrator wählt den Zielort auf dem eigenen Rechner aus und speichert das gewählte Gerät als Datei an dieser Stelle.
	Nachbedingung:	Das Gerät liegt als Datei auf dem lokalen Rechner vor.

e)	Name:	Gerät als Datei auf den Server übertragen
	Vorbedingung:	Es muss lokal eine Gerätedatei vorhanden sein.
	Beschreibung:	Der Administrator übergibt den Pfad der Datei an den Browser. Dieser überträgt die gewählte Datei in die Online-Gerätebibliothek.
	Nachbedingung:	Das Gerät ist in der Online-Gerätebibliothek enthalten.

12.7.2. Online-Tarifbibliothek verwalten

Über den Server können die in der Online-Tarifbibliothek gespeicherten Tarife verwaltet werden.

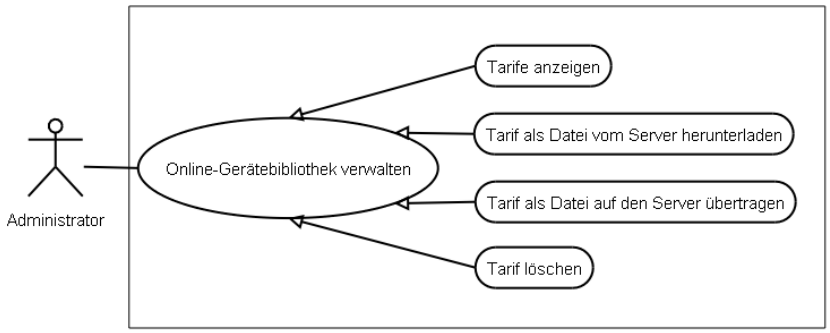

Abbildung 66: Anwendungsfälle Online-Tarifbibliothek verwalten

a)	Name:	Tarife anzeigen
	Vorbedingung:	Es muss mindestens ein Tarif in der Online-Tarifbibliothek gespeichert sein.
	Beschreibung:	Der Administrator kann die in der Online-Tarifbibliothek gespeicherten Tarife anzeigen lassen.
	Nachbedingung:	Es wird eine Liste mit den gespeicherten Tarifen angezeigt.

b)	Name:	Tarif als Datei vom Server herunterladen
	Vorbedingung:	Es muss mindestens ein Tarif in der Online-Tarifbibliothek gespeichert sein. Der Administrator muss einen Tarif auswählen.
	Beschreibung:	Der Administrator wählt den Zielort auf dem eigenen Rechner aus und speichert den gewählten Tarif als Datei an dieser Stelle.
	Nachbedingung:	Der Tarif liegt als Datei auf dem lokalen Rechner vor.

c)	Name:	Tarif als Datei auf den Server übertragen
	Vorbedingung:	Es muss lokal eine Tarifdatei vorhanden sein.
	Beschreibung:	Der Administrator übergibt den Pfad der Datei an den Browser. Dieser überträgt die gewählte Datei in die Online-Tarifbibliothek.
	Nachbedingung:	Der Tarif ist in der Online-Tarifbibliothek enthalten.

d)	Name:	Tarif löschen
	Vorbedingung:	Es muss mindestens ein Tarif in der Online-Tarifbibliothek gespeichert sein. Der Administrator wählt den gewünschten aus.
	Beschreibung:	Der gewünschte Tarif wird aus der Online-Tarifbibliothek gelöscht.
	Nachbedingung:	Der gewählte Tarif ist nicht mehr in der Online-Tarifbibliothek enthalten.

13. Benutzungsschnittstelle

Autor: Mark Eilers

Die Benutzungsschnittstelle verlangt eine einfache und intuitive Bedienung. Aus diesem Grund wird versucht dem Anwender möglichst eine Bedienbarkeit zu gewährleisten, die sich an die allgemeinen Standards, wie zum Beispiel Drag & Drop, hält. Die Benutzungsschnittstelle der Software wird in drei Ansichten unterteilt.

13.1. Profilverwaltung

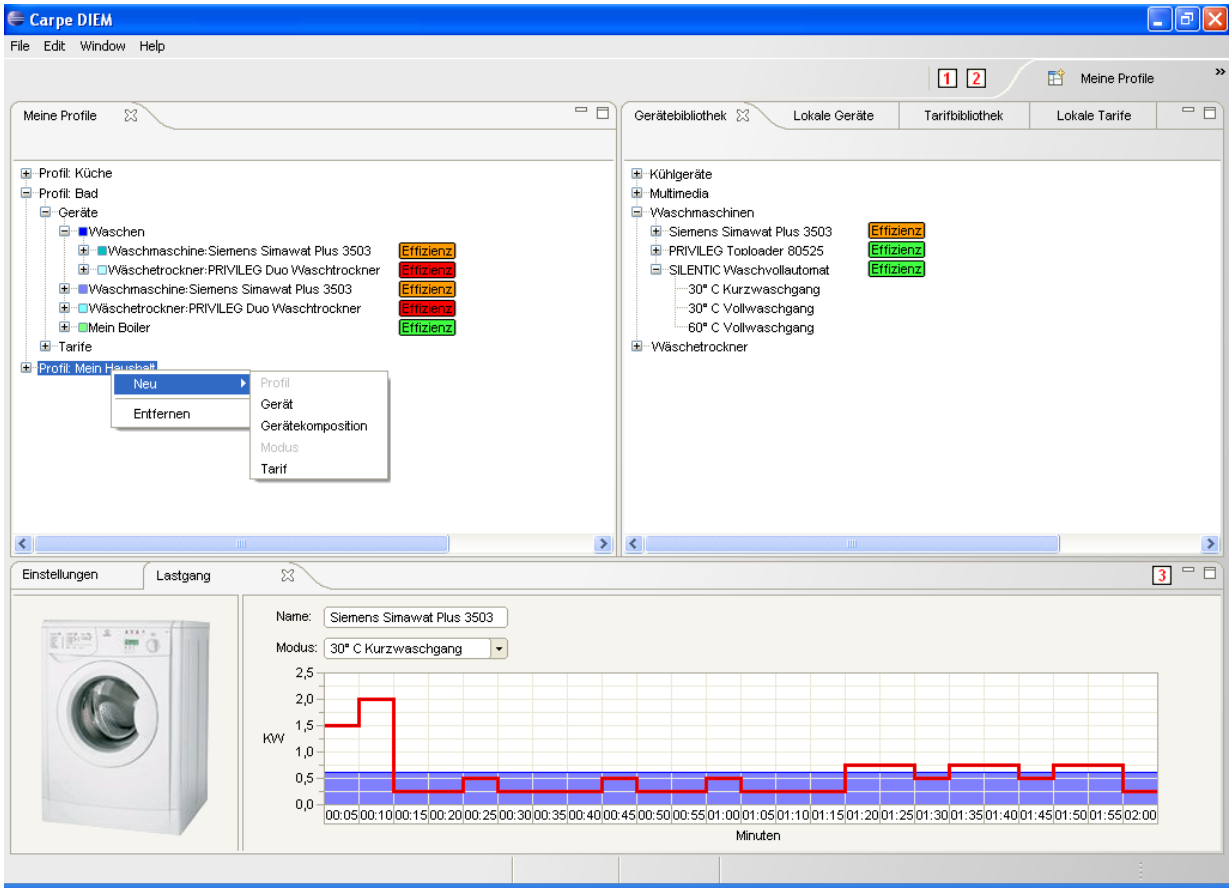

Abbildung 67: Profilverwaltung

Die Profilverwaltung (siehe Abb. 67) ist in drei große Bereiche unterteilt. Die Profilansicht im linken, die Gerätebibliothek und lokale Gerätesammlung samt Tarifen im rechten, sowie nähere Informationen und Einstellungen der ausgewählten Komponenten im unteren Bildbereich. In der Profilansicht kann der Anwender den Anwendungsfällen gemäß Profile, Geräte, Gerätekompositionen, Betriebsmodi und Tarife erstellen, bearbeiten, kopieren, entfernen und verwalten. Dies wird wie allgemein üblich durch Drag & Drop, sowie über Menüs und Kontextmenüs realisiert. Über Schaltflächen können Geräte- und Tarifdaten (1 und 2) heruntergeladen sowie Lastgänge (3) importiert werden.

13.2. Einsatzplanung

Die Einsatzplanung (siehe Abb. 68) ist ebenfalls in drei Bereiche unterteilt, in eine Profilansicht, wie in der Profilverwaltung, im linken, Einstellungen und Informationen über

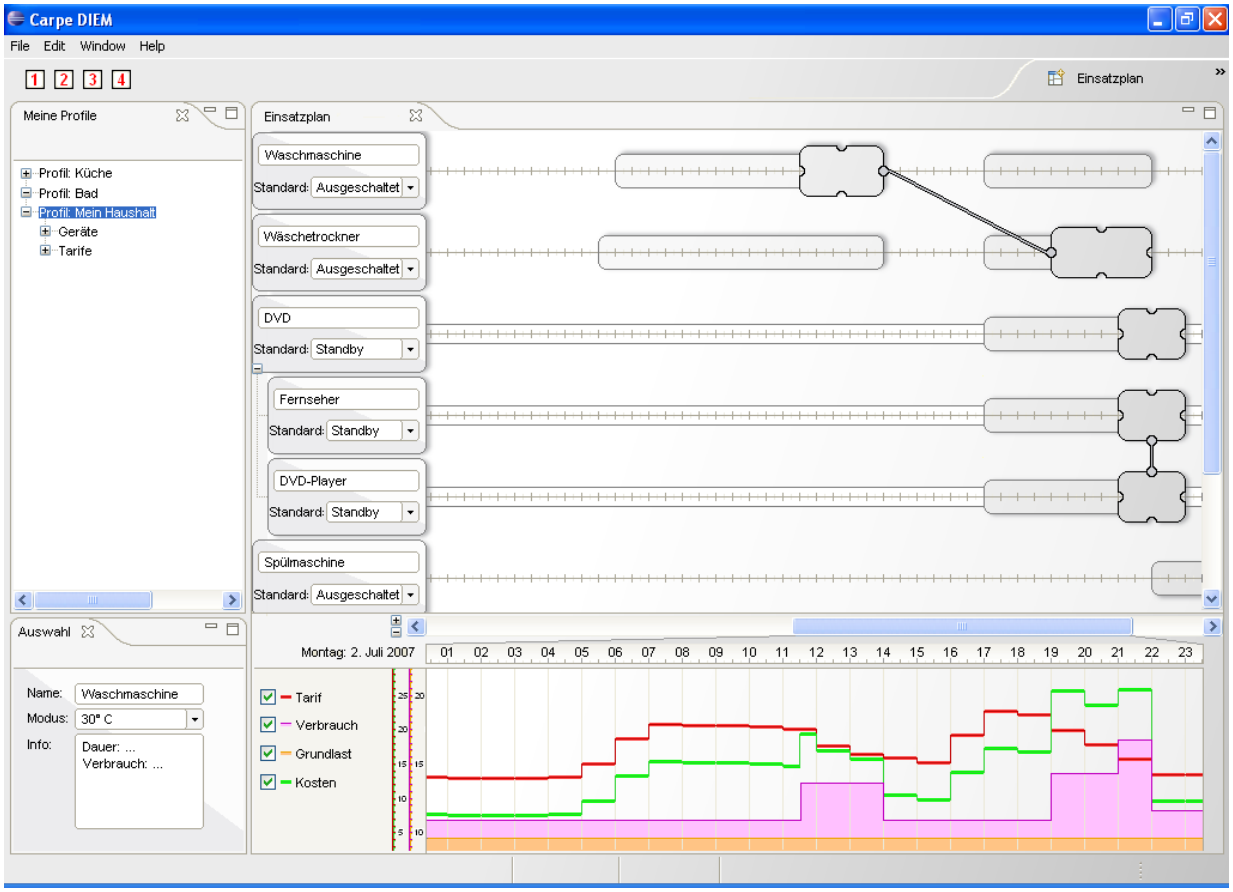

Abbildung 68: Einsatzplanung

die ausgewählten Komponenten im linken unteren und einen Einsatzplan im gesamten rechten Bildbereich. Der Einsatzplan zeigt neben den bisher geplanten Geräten und Gerätekompositionen mitsamt ihren Constraints auch verschiedene Graphen und Informationen, wie zum Beispiel den gewählten Tarif und den voraussichtlichen Stromverbrauch. Der Anwender kann per Drag & Drop Geräte, Gerätekompositionen oder Betriebsmodi in den Einsatzplan ziehen, dort über Auswahlmenüs den Standardbetriebsmodus oder den zu planenden Betriebsmodus festlegen, mit der Maus Constraints definieren und die Einsatzdauer bzw den Einsatzzeitpunkt festlegen und verändern. Der Einsatzplan soll im Hinblick auf die Zeitanzeige frei skalierbar sein. Über Schaltflächen können der Einsatzplan gedruckt (1), und optimiert (2), zeitliche Zyklen und Einstellungen vorgenommen (3), sowie weitere Constraints, wie zum Beispiel eine maximale Lastgrenze, festgelegt werden (4).

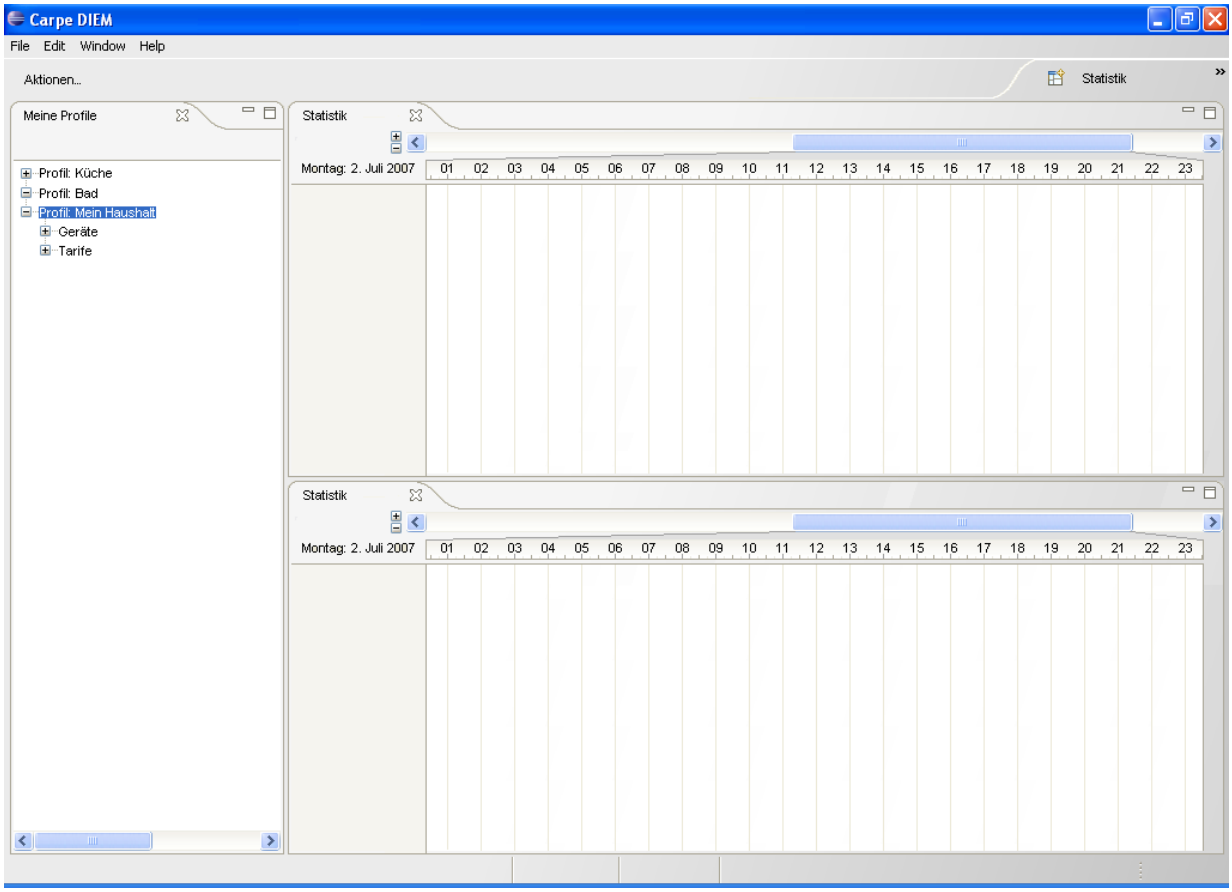

Abbildung 69: Statistik

13.3. Statistik

Die Statistik-Ansicht (siehe Abb. 113) ist in die Profilansicht im linken und vorläufig zwei Diagramme im rechten oberen und rechten unteren Bildbereich unterteilt. Der Anwender kann wiederum durch Drag & Drop verschiedene Komponenten aus der Profilansicht in die Diagramme ziehen und dann visuell vergleichen.

14. Nicht-Funktionale Anforderungen

Autor: Michael Belasus

Neben den bisher beschriebenen Funktionalen Anforderungen gibt es die im Folgenden beschriebenen Nicht-Funktionalen Anforderungen an die zu entwickelnde Software.

Hauptsächlich wird die Software zukünftig in privaten Haushalten angewendet, daher muss die Installation einfach und unkompliziert für den Privatanwender erfolgen. Die Installation des Programms wird mithilfe eines Installers durchgeführt. Mit der Instal-

lation werden alle für einen reinen Offline-Betrieb notwendigen Bibliotheken mitgeliefert.

Damit der Anwender die Funktionen und den Ablauf einer Modellierung von Haushaltsabläufen unkompliziert durchführen kann, muss das Programm intuitiv bedienbar sein. Durch entsprechende eindeutige Bezeichnungen von Ansichten, Buttons, Menü- und Kontexteinträgen kann sich der Anwender schnell und einfach zurechtfinden. Die Programmoberfläche wird derzeitigen Softwareanwendungen entsprechen. Fehler und Abhängigkeiten werden dem Anwender mithilfe von Pop-Up Fenstern mit klaren Anweisungen vermittelt. So wird die Löschung eines Gerätes im Profil nur erlaubt, wenn es in keinem Einsatzplan mehr vorhanden ist. Diese Abhängigkeit wird dann über ein Pop-Up angegeben. Eingaben, die der Anwender innerhalb des Programms vornimmt, werden auf Plausibilität geprüft in dem keine negativen Zahlen zugelassen werden. Angaben wie bei Gerätenamen oder Beschreibungen werden nicht weiter geprüft.

Der Ablauf einer Modellierung (z. B. das Anlegen des gesamten Haushaltes) ist klar strukturiert und auf die Funktion des Programms abgestimmt. Die Anzeigen und Beschreibungen innerhalb des Programms werden komplett in Deutsch gehalten. Das Programm wird modular aufgebaut, so dass spätere Erweiterungen und Ergänzungen möglich sind. Dazu gehört auch die Möglichkeit, dass weitere Sprachen hinzugefügt werden können.

Das Programm soll weitestgehend plattformunabhängig und robust sein. Das Öffnen des Menüs und die Abfrage der Bibliotheken sollen schnellstmöglich erfolgen. Bei der automatischen Optimierung wird dem Anwender visuell die Aktivität des Programms dargestellt, so dass auf eine Aktion des Anwenders immer eine erkennbare Reaktion des Programms erfolgt. Eine optionale Update-Funktion ermöglicht es dem Anwender neue Geräte zu laden. Day-Ahead-Tarife werden ebenfalls über eine Update-Funktion geladen.

Zusätzlich zu der übersichtlichen Benutzeroberfläche werden ein Handbuch und eine kurze Einführung über ein Tutorial verfügbar sein. Beispieldaten für Profile und Einsatzpläne helfen dem Anwender zusätzlich schnell und einfach sich in die Bedienung des Programms einzuarbeiten. Innerhalb des Programms werden über verschiedene Tooltips kurze Hilfen und Arbeitsanweisungen an den Anwender weitergegeben.

Die Software wird als Open Source unter der Eclipse Public License (EPL) entwickelt.

Teil IV.
Entwurf

15. Einleitung

Autor: Moritz Eysholdt

Das vorliegende Dokument beschreibt den Entwurf der Software DIEM. Der Entwurf setzt eine Abstraktionsebene unter der Anforderungsdefinition an und klärt die technischen Fragen der Implementierung.
Zunächst soll ein knapper Überblick über die zu entwickelde Software gebeben werden: DIEM besteht aus einer Desktopanwendung und einem Server, beide werden vollständig in Java implementiert. Die Desktopanwendung ermöglicht es dem Anwender, den Stromverbrauch seines Haushaltes zu simulieren. Der Anwender definiert hierzu die Einsäte seiner elektrischen Geräte in einem kalenderartigen Einsatzplan und gewährt ihnen mittels Constraints eine zeitliche Flexibilität. Die Desktopsoftware kann den Einsatzplan in Bezug auf einen zeitlich variablen Stromtarif nach den Kosten optimieren (Abschnitt 16.6). Zur Unterstützung wird dem Anwender eine Gerätebibiothek und eine Tarifbibliothek bereitgestellt. Beide werden vom Server vorrätig gehalten und von der Desktopsoftware vollständig gespiegelt. Die Desktopsoftware kann ihre Bibliotheken jederzeit vom Server aktualisieren, ist aber für den Betrieb nicht auf den Server angewiesen.
Die Desktopsoftware wird mittels der Eclipse Rich Client Platform (RCP, Abschnitt 16.1) und dem Eclipse Graphical Editing Framework (GEF, Abschnitt 25.6) realisiert. Das auf dem Eclipse Modeling Framework (EMF, Abschnitt 16.2) aufbauende Datenmodell nutzt sowohl die Desktopsoftware, als auch der Server. Die Anwendungssoftware speichert ihre Daten als XML, der Server mittels Teneo (Abschnitt 16.5) und Hibernate (Abschnitt 16.4) in einer relationalen Datenbank.
Nachdem die Frameworks (Kapitel 16) vorgestellt wurden, wird auf die Architektur (Kapitel 17) der Desktopsoftware (aka Client) und des Servers sowie das Datenmodell (Kapitel 18) eingegangen. Daraufhin werden die einzelnen Plugins (Kapitel 19) der Desktopsoftware spezifiziert und es wird detailliert auf den Server (Abschnitt 25.5) eingegangen. Abschließend veranschaulicht der Programmablauf (Kapitel 21) das Zusammenspiel der Komponenten und Qualitätssicherung (Kapitel 22) definiert die vorgehensweise zum Testen der Software.

16. Frameworks und Technologien

In diesem Abschnitt werden die einzelnen Frameworks und Technologien vorgestellt, die bei der Entwicklung der Software verwendet werden.

16.1. Eclipse Rich Client Platform (RCP)

Autor: Christian Hinrichs

Abbildung 70: RCP Logo

Das Eclipse RCP ist ein Projekt der Eclipse Foundation und definiert ein Framework zur Entwicklung von Rich Clients auf Basis der GUI-Bibliothek SWT. Es stellt eine Serviceorientierte Architektur dar, da es auf einer Implementierung der OSGi Spezifikation basiert, welche separat in dem Projekt Eclipse Equinox entwickelt wird. Die Projektgruppe hat sich für RCP als grundlegendes Framework entschieden, da es eine frei nutzbare Plattform bietet, mittels derer die strukturierte Entwicklung einer beliebig komplexen und erweiterbaren Anwendung mit anwenderfreundlichen Benutzungsoberflächen ermöglicht wird. Nach einer anfänglich steilen Lernkurve ist es den Entwicklern möglich, effizient und schnell professionelle Komponenten zu erstellen, sodass neben einer hohen Qualität des Endproduktes auch ein großer Lerneffekt für die Projektgruppenteilnehmer gegeben sein wird.
Das Konzept des RCP beinhaltet eine mehrschichtige, modularisierte bottom-up Struktur, in welcher einzelne Module in der Regel nur auf benachbarte Module der gleichen, oder auf Module in niederen Schichten zugreifen. In Abbildung 71 ist die Struktur einer auf diesem Framework basierenden Anwendung schematisch dargestellt. Die Equinox Laufzeitumgebung stellt die grundlegende Basis dar (unterste Schicht in der Abbildung), auf welcher die entwickelten Komponenten gestartet werden. Diese Komponenten werden als Bundles oder Plugins bezeichnet und stellen die mittlere Schicht in der Abbildung dar. Sofern die Bundles eine grafische Repräsentation innerhalb der Anwendung besitzen, wird diese durch die Verwendung der vom RCP bereit gestellten grafischen Subframeworks SWT, JFace sowie UI realisiert. Diese Pakete bauen selbst aufeinander auf, können aber je nach Bedarf einzeln angesprochen werden. Die Gesamtanwendung besteht dann aus einer Menge von Bundles, die die einzelnen funktionalen Komponenten der Anwendung umsetzen. Üblicherweise existiert dazu ein zentrales Bundle, an welches die weiteren Komponenten angeknüpft werden.
Der Aufbau der GUI einer solchen Anwendung erfolgt dabei durch einzelne Container, deren Basisfunktionalitäten bereits durch das Framework gegeben sind, und vom Entwickler nur noch mit Inhalt gefüllt werden müssen. Zu den Basisfunktionalitäten zählt dabei z.B. eine Titelleiste, in welcher der Titel des Containers angezeigt werden kann. Es werden zwei typische Container unterschieden. Zum einen existiert die Klasse `EditorPart`, welche einen sogenannten Editor bereitstellt. Dieser hat die Eigenschaft, dass er seinen Inhalt auf Basis einer Datenquelle darstellt, und diese damit editiert. Die Datenquelle wird durch ein ihm zugeordnetes Objekt von einer für die jeweilige Anwendung passende Implementierung des Interfaces `IEditorInput` definiert und verweist meist auf eine Datei im Dateisystem (etwa durch `IPathEditorInput`). Wird der dargestellte Inhalt durch den Anwender oder programmatisch verändert, so wechselt

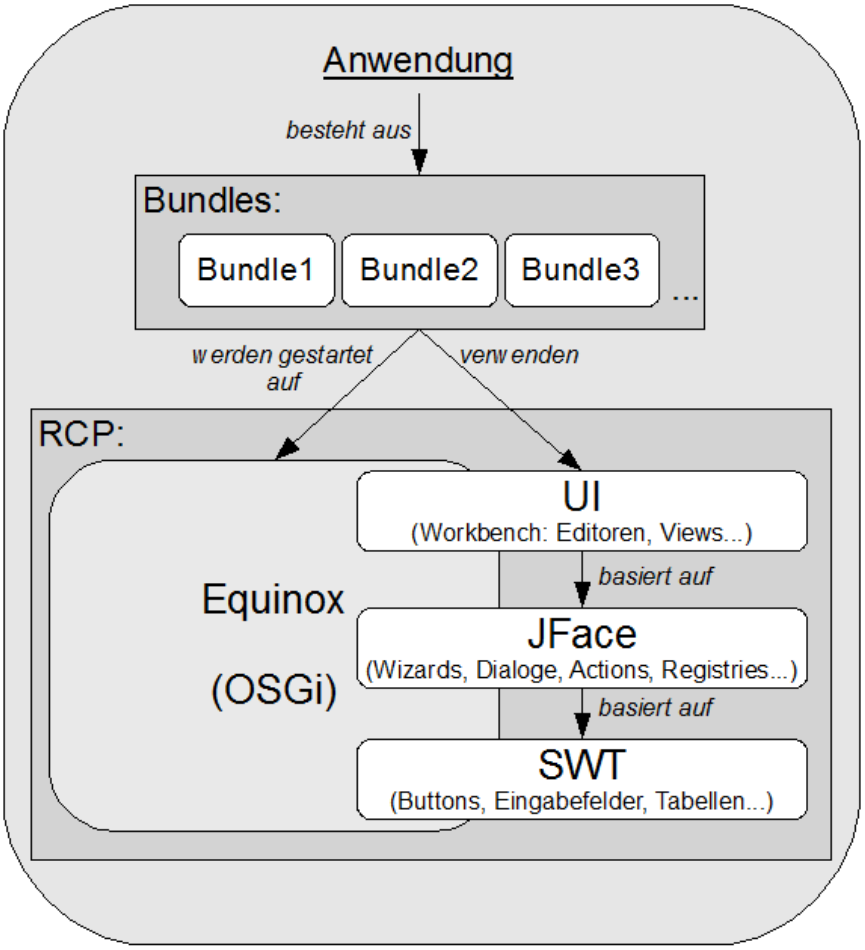

Abbildung 71: Struktur einer RCP Anwendung

der Editor in den Zustand *dirty*, was durch ein Sternsymbol neben dem Titel angezeigt wird. Nach einer Synchronisierung mit der Datenquelle (also z.B. durch das Übernehmen der vorgenommenen Änderungen in die geöffnete Datei) wird der Editor wieder in den Standardzustand versetzt und das Sternsymbol verschwindet. Der zweite Containertyp wird durch die Klasse `ViewPart` realisiert. Ein durch diese Klasse erstellter View arbeitet im Unterschied zum Editor nicht direkt auf einer Datenquelle, sondern stellt seinen Inhalt in Abhängigkeit von anderen Views oder Editoren dar. Existiert beispielsweise ein Editor zur Bearbeitung von XML-Dateien, so könnte man einen View implementieren, welcher die zur Zeit bearbeitete XML-Datei auf syntaktische Fehler überprüft und auf diese hinweist. Der Vorteil bei diesem dem Observer-Pattern nachempfundenen Prinzip liegt darin, dass ein View nicht einer bestimmten Instanz, son-

dern immer einem Typ eines Editors zugewiesen wird. Würden in obigem Beispiel etwa mehrere XML-Dokumente in jeweils einer Instanz des Editors parallel geöffnet sein, so würde der View immer gerade den Editor überprüfen, welcher zur Zeit den Fokus hat, also im Vordergrund steht.
Die Durchführung von durch den Anwender initiierten Vorgängen kann in einer RCP-Anwendung auf drei Arten vorgenommen werden. Für kleine Aktionen, wie etwa die Aktivierung einer grafischen Komponente auf Knopfdruck, würde man die entsprechenden Codezeilen direkt in den dem Knopf zugeordneten Listener schreiben und unmittelbar ausführen. Aufwändigere Aufgaben werden durch Ableitungen der Klasse `Action` realisiert. So ist typischerweise jedem Button der Toolbar sowie jedem Menüeintrag eine Action zugewiesen, welche bei Aktivierung des Buttons bzw. Menüs ausgeführt wird. Für Vorgänge, welche eine längere Zeit in Anspruch nehmen, wie etwa das parsen einer geladenen Datei, bietet RCP den Job-Mechanismus. Ein Job wird ähnlich wie eine Action gehandhabt, nur dass seine Ausführung dabei vom Framework geplant und durchgeführt wird. Die Planung besteht dabei im Wesentlichen aus der Überprüfung auf Inkonsistenzen mit anderen laufenden Jobs, da diese auf Wunsch parallel ausgeführt werden können. Zudem stellt das Framework grafische Komponenten zur Darstellung des Ablaufes eines Jobs bereit (z.B. Fortschrittsbalken).

Die Aufgaben der Entwickler in dieser Projektgruppe liegen also im Wesentlichen darin, die einzelnen Anforderungen an die zu erstellende Software durch Kombinationen von Editoren und Views in Verbindung mit entsprechenden Actions zu realisieren. In Kapitel 19 werden die zu implementierenden Komponenten daher nach genau dieser Struktur aufgeteilt und beschrieben. Die Einbettung in Equinox sowie der optionale Produktexport in eine für bestimmte Betriebssysteme fertig kompilierte Form werden komplett vom RCP übernommen und müssen daher an dieser Stelle nicht weiter betrachtet werden.

Weiterführende Literatur:

- Offizielle Wikiseite des RCP:
  http://wiki.eclipse.org/index.php/Rich_Client_Platform
- Getting started: RCP Tutorial von Ed Burnette:
  http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html
- Rich-Client-Entwicklung mit Eclipse 3.1, Dr. Berthold Daum, dpunkt.verlag GmbH, 1. Auflage 2005

16.2. Eclipse Modeling Framework (EMF)

Autor: Moritz Eysholdt

Das Eclipse Modeling Framework (EMF) ist ein Java-Framework und ein Codegenerator zur Entwicklung von Werkzeugen und Anwendungen basierend auf einem formalen Modell. Für dieses Modell, das EMF-Modell, bietet Eclipse zwar einen Editor an, üblicher Weise wird es jedoch aus anderen formalen Beschreibungen abgeleitet. Ist dieses Modell vorhanden, kann es zur Codegenerierung für fast beliebige Zwecke verwendet werden. Abbildung 73 bietet hierzu eine Übersicht. Die genaueren Zusammenhänge werden in den folgenden Kapiteln erläutert.

Abbildung 72: EMF Logo

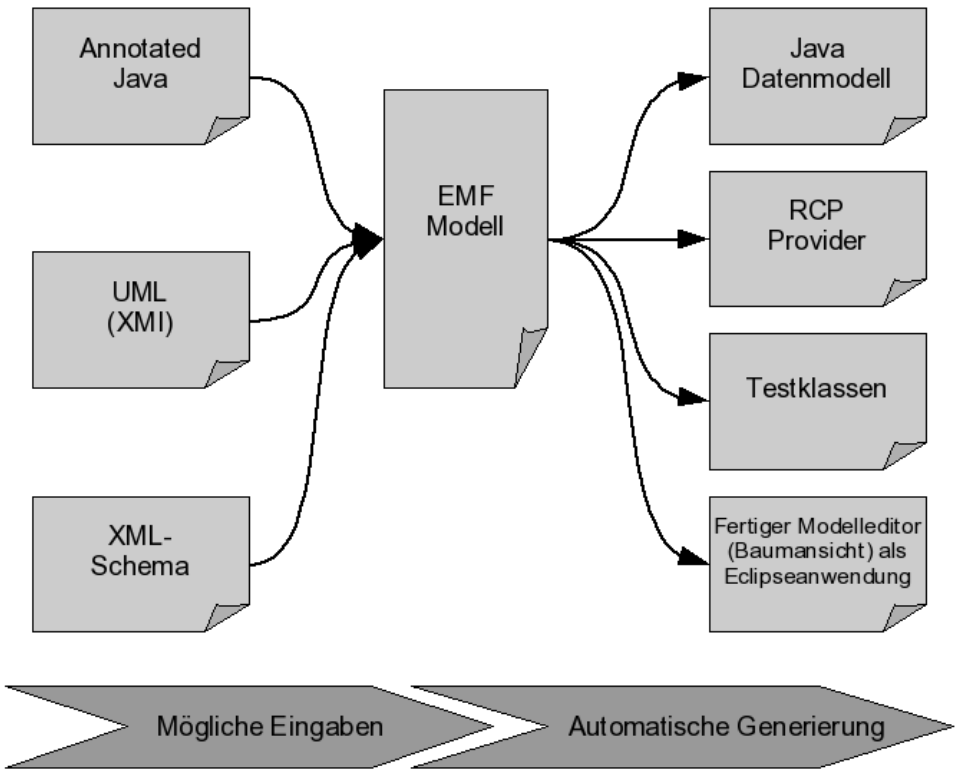

Abbildung 73: EMF Übersicht

16.2.1. Modell

Zunächst stellt sich die Frage, was überhaupt ein Modell ist. Prominente Beispiele sind ein UML-Modell, welches durch eine Vielzahl von UML-Diagrammen visualisiert wird, oder ein Datenmodell für eine Datenbank, welches durch ein Entity Relationship Diagramm (ER) dargestellt wird. Grob zusammengefasst ist ein Modell in diesem Kontext eine Datenstruktur, die durch ihre Objekt-Typen und deren Eigenschaften und Beziehungen zueinander beschrieben wird.
Gleiches gilt auch für ein EMF-Modell. Es enthält einige vordefinierte Datentypen, aber es können auch eigene Datentypen mit Eigenschaften, Vererbungsbeziehungen, Referenzbeziehungen etc. definiert werden. Ebenso kann ein EMF-Modell, gespeichert in einer Datei, ein Modell in einer anderen Datei referenzieren.
Das EMF-eigene Metamodell Ecore, welchem all seine Modelle genügen müssen, besitzt eine gewisse Ähnlichkeit mit dem Metamodell der UML, dem MOF. EMF begann als eine Implementierung der UML, jedoch zeigte die Praxiserfahrung, dass nur Teile des Standards sinnvoll waren. Damit wurde ein eigener Weg eingeschlagen. Inzwischen wurde das MOF 2.0 veröffentlicht und mit ihm das EMOF (Essential MOF). Ecore und EMOF sind bis auf einige Namensunterschiede weitestgehend identisch.

16.2.2. Mögliche Eingaben

Wie schon angedeutet wird ein EMF-Modell üblicher Weise nicht direkt erstellt, sondern aus einer anderen formalen Notation abgeleitet. Hierzu bringt das EMF von Haus aus Unterstützung für drei verschieden Formate mit.

- **Annotated Java** Die Datenstruktur muss in Form von Java-Interfaces beschrieben werden. Jede Get-Methode wird hierbei auf eine Eigenschaft gemapped.
- **UML (XMI)** Das EMF-Framework kann XMI-Dateien von Rational Rose direkt importieren, diverse andere UML-Editoren ,wie beispielsweise MagicDraw der Firma NoMagic, unterstützen den Export in EMF auch direkt.
- **XML Schema** XML Schemas können vom EMF direkt importiert werden. Aus Erweiterungsbeziehungen der XML-Typen werden dabei Vererbungen und aus den Attributen und Unterelementen werden die Eigenschaften eines Datentyps. Werden XML-Schemas verwendet, kann EMF ein Datenmodell direkt in eine dem Schema genügende XML-Datei speichern und auch wieder einlesen. Auf die Verwendung von XML-Schemas wird in Kapitel 18 weiter eingegangen.

16.2.3. Codegenerierung

Ist das Modell vorhanden, kann es zur Codegenerierung verwendet werden. Bestimmte Szenarien werden bereits von EMF unterstützt, der generierte Code kann jedoch prinzipiell jegliche Art von Textdatei sein. Zur Codegenerierung verwendet EMF JETs (Java

Emitter Templates), welche das Modell als Eingabe bekommen und daraufhin jegliche Art von Text generieren können. Hat der Entwickler im generierten Text Änderungen vorgenommen, sorgt JMerge dafür, dass diese Änderungen bei der nächsten Generierung nicht überschrieben werden.
EMF kann standardmäßig die folgenden Artefakte einer Anwendung generieren. Ist es noch nicht vorhanden, wird hierbei für jedes ein eigenes Eclipse-Projekt automatisch im Workspace erstellt.

- **Java-Datenmodell** Für jeden Datentyp im EMF-Modell wird eine Java-Bean erstellt. Diese besteht aus einem Interface, mit allen Getter- und Setter-Methoden, sowie der zugehörigen Implementierung.
- **RCP Provider** Für jeden Datentyp werden Klassen erstellt, welche zum RCP gehörige Provider-Interfaces implementieren. Diese stehen dem Entwickler als Adapter zwischen GUI-Elementen und Datenmodell zur Verfügung.
- **Testklassen** Es werden Helferklassen generiert um eigenen Quellcode zu testen, welcher das Datenmodell verwendendet.
- **Dateneditor** Der Dateneditor ist eine auf dem Java-Datenmodell und den RCP Providern aufbauende vollständige Eclipse-Anwendung. Er lässt sich in einem Eclipse Workbench als Plugin installieren und erlaubt es dann, die im Datenmodell spezifizierten Dateien im XML-Format zu laden, zu bearbeiten und zu speichern. Die Bearbeitung erfolgt über eine Baumansicht und den Eigenschafteneditor.

Weiterführende Literatur:

- Webseite des Eclipse Modeling Framework Projekts (EMF)
  http://www.eclipse.org/modeling/emf/
- Getting Started: The Eclipse Modeling Framework (EMF) Overview
  http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html
- XML Schema to Ecore Mapping
  http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf

16.3. Graphical Editing Framework (GEF)

Autor: Mark Eilers

Abbildung 74: GEF Logo

Das Graphical Editing Framework, ein Projekt der Eclipse Foundation, erlaubt die Entwicklung von grafischen Editoren zur Darstellung und Bearbeitung eines Datenmodells, die als Plugin in die Eclipse-Umgebung integriert werden können. Das GEF ist in zwei Plugins unterteilt, dem Draw2d (org.eclipse.draw2d), einem Toolkit zum Zeichnen und Anordnen von Grafiken auf einem SWT Canvas, und dem Plugin GEF (org.eclipse.gef) selbst, das ein auf Draw2d aufbauendes Model-View-Controller-Framework definiert.
Das GEF umfasst unter anderem:

- Tools wie Selection, Marque, Resize, Creation und Connections
- Eine Palette zur Darstellung von Tools
- "'Handles"' zur Verwaltung von Connections
- Unterstützung von Undo- und Redo-Funktionalitäten durch Commands und einem internen CommandStack

Das GEF wurde entwickelt um ein existierendes Datenmodell zu visualisieren und editieren zu können. Zu diesem Zweck stellt GEF Viewer bereit, die überall im Eclipse Workbench benutzt werden können. Diese Viewer basieren auf dem Model-View-Controller-Prinzip (MVC), der strikten Dreiteilung eines Softwaresystems in das Datenmodell (Model), die Darstellung (View) und der Programmsteuerung (Controller). Das Model enthält dabei alle Daten die visualisiert und editiert werden sollen, besitzt aber selbst keine Kenntnis darüber wie dies realisiert wird. Es muss nur in der Lage sein, Listener zu implementieren um Änderungen des Models an den Controller zu melden. Der View (nicht zu verwechseln mit einem View des RCP) darf wiederum kein Wissen über das Model oder den Controller besitzen und auch über keine eigene Logik verfügen. Der Controller ist dafür verantwortlich dem View mitzuteilen, was visualisiert werden soll und Änderungen am Model vorzunehmen.
Ein GEF Viewer realisiert Controller und View für ein Datenmodell. Für jedes zu visualisierende und editierbare Objekt des Datenmodells kann nun der entsprechende View aus Draw2d-*figures* erstellt werden. Einige von Draw2d bereitgestellte figures können direkt Teile eines Models visualisieren, so zum Beispiel das Label-figure zur Darstellung eines Strings, umfangreichere Objekte können aus Kompositionen von figures, borders und Layout Managern zusammengesetzt werden. Der Controller wird durch *EditParts* realisiert. Ein EditPart hat die Aufgabe den entsprechenden View, also die Draw2d-figures, zu einem Objekt des Models zu generieren, zu erneuern und das Model selbst

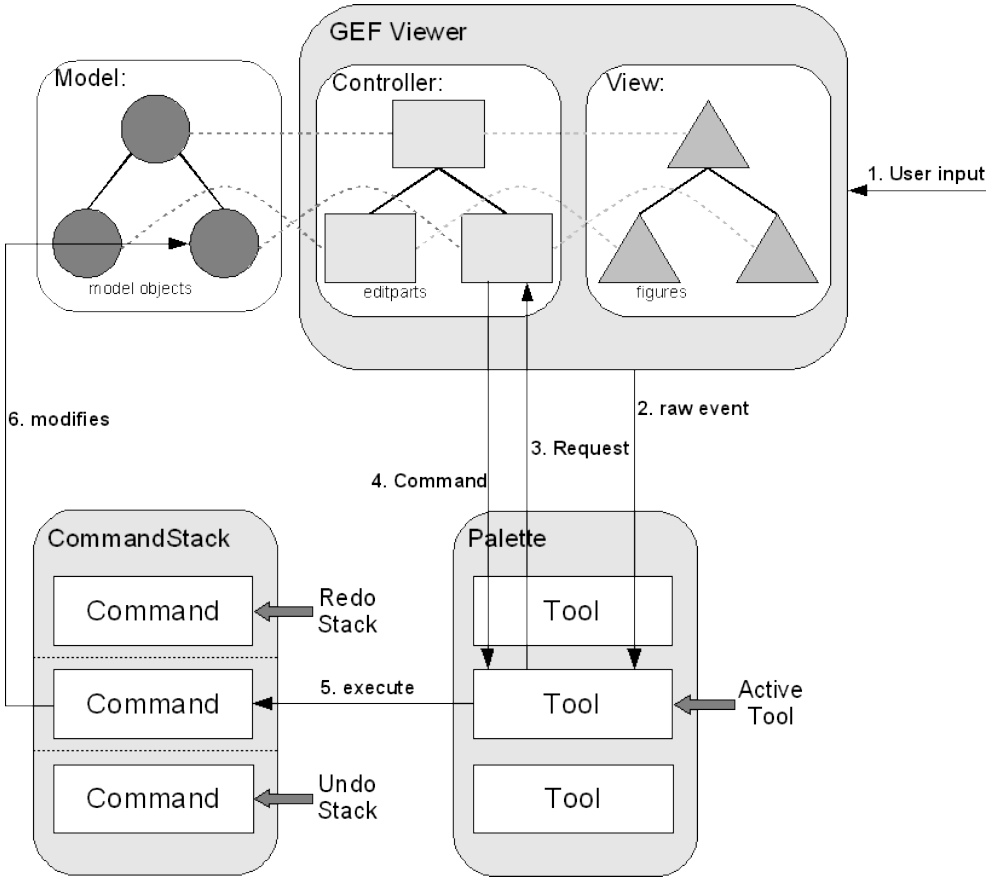

Abbildung 75: GEF Übersicht

zu editieren.

Abbildung 75 gibt einen Überblick über die Funktionsweise des GEF's:

- 1. Der Anwender kann auf der GUI, zum Beispiel mit Maus und Tastatur, agieren.
- 2. Eine Aktion des Anwenders erzeugt ein Event, das von dem GEF Viewer an das aktive Tool weitergeleitet wird.
- 3. Das aktive Tool erstellt aus solchen Events Request-Objekte und übergibt sie den entsprechenden EditParts. Requests werden von GEF verwendet um festzulegen welche Operationen auf dem Datenmodell durchgeführt werden müssen.
- 4. Die EditParts generieren aus den Requests ein Feedback an die GUI und ein Command-Objekt, das wieder an das aktive Tool übergegeben wird. Das Command-Objekt besitzt *execute-*, *undo-* und *redo*-Methoden um die im Request geforderten Modifikationen am Datenmodell zu implementieren.

- 5. Das Command-Objekt wird vom aktiven Tool an den CommandStack übergeben.
- 6. Der CommandStack verarbeitet das Command-Objekt und modifiziert entsprechend das Datenmodell. Diese Änderungen werden dann vom Datenmodell wieder an die EditParts gemeldet, die ihre Views entsprechend aktualisieren.

Für die Projektgruppe bietet es sich an, das GEF zur Realisierung aller grafischen Editoren der zu entwickelnden Software zu verwenden, da es zum einen das von EMF generierte Datenmodell explizit unterstützt und zum anderen genau die Funktionalitäten bereitstellt, welche die in dieser Software verwendeten grafischen Editoren implementieren müssen.

Weiterführende Literatur:

- Offizielle Webseite des GEF:
  http://www.eclipse.org/gef/
- Getting started: Create an Eclipse-based application using the Graphical Editing Framework:
  (http://www.ibm.com/developerworks/opensource/library/os-gef/)

16.4. Hibernate

Autor: Christian Bley

Abbildung 76: Hibernate Logo

Hibernate ist eine Persistenzlösung für Java und .NET, welche von JBossLabs entwickelt wurde. Hibernate automatisiert das Mapping eines in einer Programmiersprache implementierten objektorientierten Datenmodells auf eine relationale Datenbank. Dieses Mapping wird auch als Objektrelationales-Mapping bezeichnet. Entwickelt wird Hibernate als Opensource unter der Lesser GNU Public License (LGPL).
Hibernate bietet die Möglichkeit auch komplexe Objekte in einer Datenbank zu speichern, Suchanfragen mit der eigens hierfür entwickelten Hibernate Query Language durchzuführen oder auf geladenen Objekten zu arbeiten, während Hibernate für die Konsistenz der Daten sorgt. Es ist in mehrere Module unterteilt, wovon das Core Modul sämtliche beschriebenen Funktionen bietet. Unter anderem gibt es ein Modul zur Verwendung der Java Persistence API oder zur Unterstützung von Java Annotations.
Die Projektgruppe hat sich für Hibernate entschieden, weil es einfach zu konfigurieren ist, während es die Möglichkeit bietet das Datenmodell ohne Modifikation in Datenbanken zu speichern. Zu diesem Zweck verwendet die Projektgruppe das Hibernate

Core-Modul mit einem Konnektor namens Teneo, der im Abschnitt 16.5 näher beschrieben wird.
Hibernate benötigt zum Betrieb die Zugriffsdaten für die Datenbank, den Zugriffspfad und den Datenbankdialekt. Zugriff wird über einen JDBC-Treiber hergestellt, dessen Treiberklasse Hibernate angegeben werden muss. Diese lädt Hibernate zur Laufzeit und steuert sämtliche Transaktionen mit der Datenbank. Hibernate liefert des weiteren Verbindungsmanager mit, welche mehrere JDBC-Verbindungen bereithalten und bei Bedarf zuteilen können.

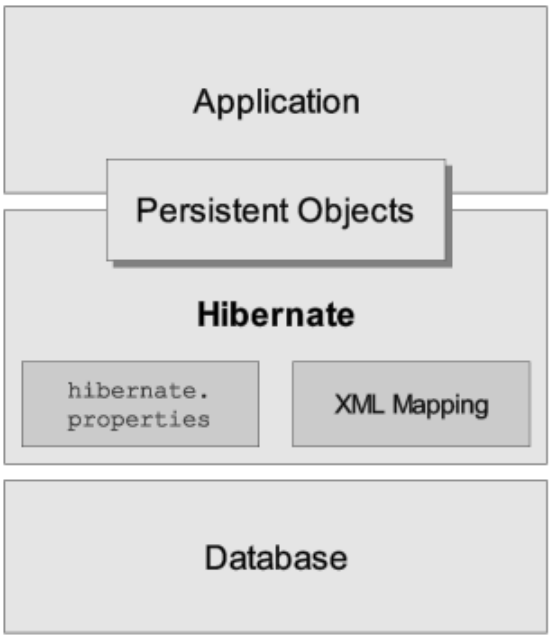

Abbildung 77: Stuktur einer Hibernate Anwendung

Konfiguriert wird Hibernate über XML-Dateien, den so genannten Mappings. In diesen wird festgelegt, welche Objektattribute in welchem Datenbankformat gespeichert werden sollen, welches das Schlüsselattribut ist, sowie Referenzen auf andere Objekte. Wenn Hibernate mit den nötigen Konfigurationen initiiert wurde, kann man über eine Hibernatesession auf die Datenhaltung zugreifen, Objekte speichern und laden oder nach bestimmten Kriterien suchen lassen. Abbildung 77 zeigt das Schema einer typischen Hibernateanwendung.
Da in der Projektgruppe ein aus einem XML-Schema generiertes Datenmodell verwendet wird, liegt es nahe, die Mappingdateien für Hibernate ebenfalls generieren zu lassen. Hierfür wird ein Tool namens Teneo (Siehe Abschnitt 16.5) verwendet, um den Funktionsumfang des Eclipse Modelling Frameworks (EMF) auch mit Hibernate voll nutzen zu können. Dieses erstellt die Mappings aus den vom EMF generierten Packageklassen. Somit muss nur die Konfiguration des Datenbankzugriffs von Hibernate durch

die Projektgruppe konfiguriert werden.

Weiterführende Literatur:

- Offizielle Hibernate-Webseite:
  http://www.hibernate.org/
- Getting started: I want to learn Hibernate for Java:
  http://www.hibernate.org/152.html
- Dokumentation: Hibernate Core 3.2.2 Reference Guide:
  http://www.hibernate.org/hib_docs/v3/reference/en/html/

16.5. Teneo

Autor: Christian Bley

Abbildung 78: Elver Logo

Das Werkzeug Teneo wird vom Elver.org entwickelt und ist Teil des Eclipse Modelling Framework Technologies Projekts (EMFT). Es stellt eine Verbindung zwischen dem Eclipse Modelling Framework (EMF) und den Persistenzlösungen Hibernate und JDO/JPOX her.
Da für die zu entwickelnden Software Hibernate und EMF verwenden wollen, bietet Teneo eine optimale Lösung, um unser Datenmodell mit geringen Aufwand in einer Datenbank zu speichern.
Sämtliche Konfiguration für Hibernate und Teneo geschieht über ein Datastoreobjekt, welches die Verbindungsdaten für Hibernate übergeben bekommt. Um die EMF-Modelle verwenden zu können müssen zuerst deren, vom EMF generierte EPackage-Objekte im Datastore registriert werden. Abbildung 79 zeigt den Prozess vom EMF-Modell zum Datastore. Wird Teneo mit diesen Daten initialisiert, erzeugt es die Hibernate-Mappings aus den EPackage-Klassen und richtet die nötigen Tabellen und Indizes in der Datenbank ein. Zugriff auf die Persistenzebene erhält man über die Hibenernatesession. Auch das Speichern und Laden geschieht über die herkömlichen Hibernatemethoden. Die Laufzeitstruktur einer Teneoanwendung zeigt Abbildung 80. Beim Instantiieren und Modifizieren der Objekte sorgt Teneo dafür, das die Konsistenz des EMF-Modells gewahrt bleibt.
Teneo gestattet auch die Verwendung des EMF-Ressourcesets. Bei der Registrierung einer Ressource wird ein Teneospezifischer Unified Resource Identifier (URI) verwendet, unter dem diese Teneokonfiguration zur Laufzeit erreicht werden kann. Die Ressource verwaltet dann sämtliche Sessions und Transaktionen. Es kann somit losgelöst von Hibernate und Teneo programmiert werden und die Datenhaltung bleibt transparent.

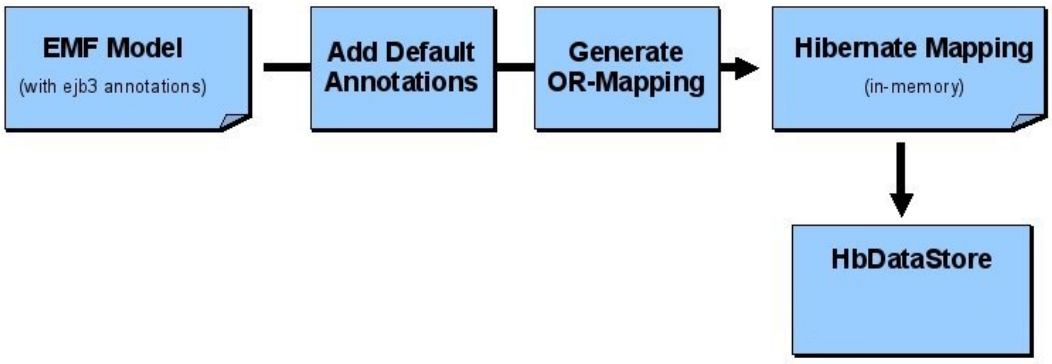

Abbildung 79: Teneo Mapping

Weiterführende Literatur:

- Teneo-Webseite des Eclipse Modeling Framework Technologies (EMFT) Projektes: http://www.eclipse.org/emft/projects/teneo/
- Elver persistency für Hibernate aka Teneo: http://www.elver.org/hibernate/index.html
- Getting started: Teneo Quick Tutorial: http://www.elver.org/hibernate/quick_tutorial.html

16.6. Automatische Einsatzplanoptimierung

Autor: Michael Erlmann

Einer der Teilaspekte der zu entwickelnden Software ist die automatische Optimierung eines Einsatzplans durch den Computer. Dies soll durch einen Ablaufplanungsalgorithmus realisiert werden. Dieses Kapitel betrachtet zunächst unterschiedliche Ansätze und legt sich dann auf eine Realisierung fest.

16.6.1. Ablaufplanung im Kontext

Bei der automatischen Einsatzplanung in dieser Software ergibt sich ein Ablaufplanungsproblem mit Constraints. Dieser wird beschrieben durch ein 7-Tupel (**A**ufträge, **R**essourcen, **P**rodukte, **H**ard-**C**onstraints, **S**oft-**C**onstraints, **E**reignisse, **Z**iel**f**unktion). Im Kontext der Optimierung der Einsätze von Geräten in einem Haushalt entsprechen sie dem Folgenden:

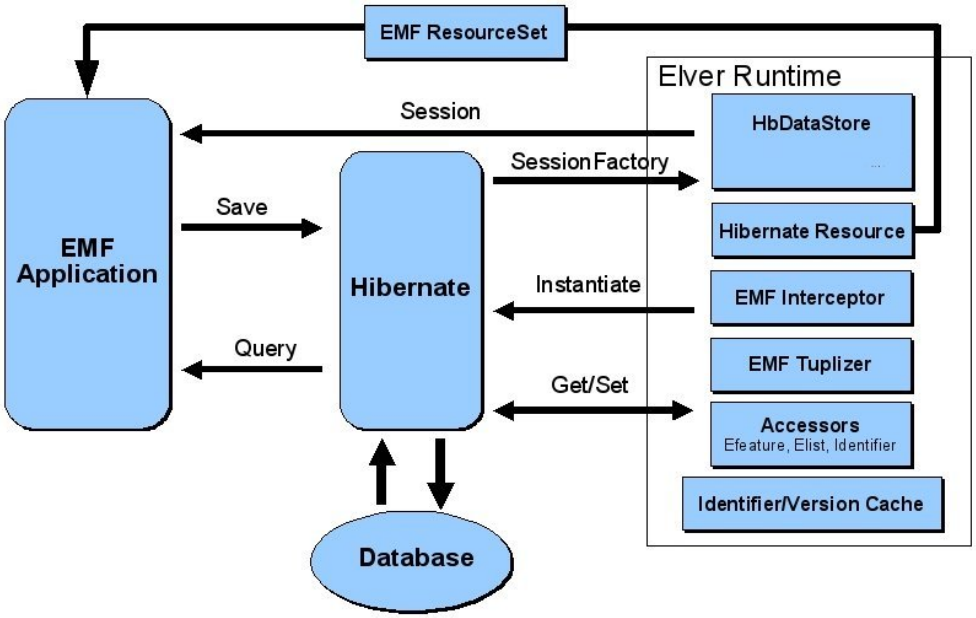

Abbildung 80: Teneo Laufzeitstruktur

Aufträge sind die Einsätze, welche eingeplant werden sollen.

Ressourcen entsprechen den Geräten, auf welchen die Betriebsmodi ausgeführt werden.

Produkte sind akute Ausführungen von Betriebsmodi auf Geräten zu bestimmten Zeitpunkten.

Hard-Constraints sind die Beschränkungen, die nicht verletzt werden dürfen, wie zum Beispiel dem gleichzeitigen Benutzen eines Gerätes in zwei Betriebsmodi.

Soft-Constraints sind Regeln, bei denen eine Einhaltung wünschenswert, jedoch nicht zwingend erforderlich ist. Sie sollten in unserem Kontext nicht auftreten.

Ereignisse treten auf, wenn der Anwender einen Einsatz nicht durchführen will oder kann. Da es dem Anwender jedoch überlassen bleibt den Plan anzupassen oder gar neu zu planen, wird eine automatische Reaktion auf Ereignisse nicht beachtet.

Zielfunktion ist die Minimierung der verursachenden Kosten anhand des verwendeten Tarifes.

Durch die Verwendung dieses Modellierungsansatzes ist es möglich verschiedene Ansätze der Ablaufplanung zum Generieren von Plänen heranzuziehen. Neben den exakten Verfahren gibt es heuristische Ansätze, welche initial gute Pläne erstellen oder

bereits erstellte Pläne verbessern. Diese einzelnen Ansätze werden in Bezug auf den Kontext im Folgenden betrachtet.

Exakte Lösungsverfahren

Die exakten Lösungsverfahren finden garantiert ein optimales Ergebnis oder zeigen, dass keine Lösung gefunden werden kann. Dabei werden unter anderem zwei Ansätze genutzt, die *Brute-Force* und *Branch & Bound* genannt werden. Im Folgenden wird betrachtet, ob sich diese zwei Ansätze sinnvoll auf die Planungsaufgabe anwenden lassen.

Brute-Force versucht durch explizite Enumeration jede mögliche Lösung zu erstellen. Für die zu entwickelnde Software bedeutet dies für jeden Einsatz jede mögliche Stelle zu besetzen, in jeder möglichen Kombination, mit allen anderen Einsätzen im Einsatzplan. Da ein Einsatz um eine Minute genau geplant werden kann, können hierbei sehr schnell sehr viele Kombinationen entstehen, abhängig von der Größe der Zeitabschnitte, in welchen die Einsätze verschoben werden können. Da diese je nach Einsatz mehrere Stunden betragen können ist hier mit einer sehr großen Zahl verschiedener Pläne und damit auch einer langen Laufzeit zu rechnen.

Branch & Bound beschreibt einen Suchbaum und zerlegt damit das Optimierungsproblem in Teilprobleme, um somit das Erstellen aller Pläne zu Umgehen. Dabei stellen die Knoten eine konkrete Verplanung dar und ein Pfad von der Wurzel bis zum tiefsten Ast entspricht einem kompletten Einsatzplan. Beim Erstellen der Knoten wird jeweils betrachtet, ob es sich noch lohnt den Teilbaum weiter zu führen, oder ob bereits abgebrochen werden kann, da eine bessere Gesamtlösung existiert, als der aktuelle Teilbaum noch bieten kann. Theoretisch kann damit das vollständige Erstellen vieler Pläne eingespart werden. Eine genauere Beschreibung zu Branch & Bound findet sich in der Seminararbeit Optimierungsheuristiken von Mark Eilers zur Projektgruppe Demonstrator InHouse Energie-Management. Beim Szenario der zu entwickelnden Software entspricht ein Knoten der zeitlichen Festlegung eines Einsatzes. Die Bewertung eines Knotens sind die verursachenden Kosten anhand des Tarifes. Für jeden zu verplanenden Einsatz muss ein eigener Baum erstellt werden, bei dem dieser Einsatz als Wurzelknoten fungiert. Dabei ist zu erwarten, dass Tarife im Kontext der Verplanung für lange Zeiträume den gleichen Wert annehmen. Daher ist auch hier eine sehr große Anzahl von Plänen und damit eine lange Laufzeit zu erwarten, da eine minutengenaue Planung durchgeführt wird, und daher sehr viele Teilbäume die gleiche Güte besitzen aber dennoch komplett aufgebaut werden müssen. Dadurch wird der theoretische Vor-

teil des Branch & Bound vorraussichtlich entfallen.

Heuristische Ablaufplanung

Anders als bei den exakten Lösungsverfahren erstellen die heuristischen Verfahren nicht mehrere Pläne. Statt dessen wird durch eine Konstruktionsheuristik ein möglichst guter Plan erstellt oder durch eine Verbesserungsheuristik ein initialer Plan bis zum Erreichen eines gewissen Abbruchkriteriums verbessert. Daher können die Heuristiken keinen optimalen Plan garantieren. Der Vorteil ist, dass diese Verfahren in der Regel eine wesentlich geringere Laufzeit aufweisen, jedoch nur angewendet werden können, wenn kein optimaler Plan notwendig ist.

Konstruktionsheuristiken müssen für das Planungsproblem entsprechend erstellt werden. Dabei kann man sich an bereits existierenden Heuristiken orientieren. Für das beschriebene 7-Tupel (A,R,P,HC,SC,E,ZF) existieren hierbei unter anderem die vier Ansätze *Auftragsbasiert*, *Operationsbasiert*, *Zeit- und Auftragsbasiert* sowie *Ressourcenbasiert*. Diese arbeiten jeweils mit einem Auswahlkriterium für das als nächstes zu verplanende Element, unterscheiden sich jedoch darin, worauf diese Auswahlkriterien angewendet werden sollen. Diese Auswahlkriterien sind zudem Kontextabhängig. Da es, wie bereits beschrieben, möglich ist, das Planungsproblem der zu entwickelnden Software auf das 7-Tupel abzubilden, können diese Ansätze als Beispiele zur Konstruktion einer Konstruktionsheuristik dienen:

Auftragsbasierte Planung verplant nacheinander die verschiedenen Aufträge an jeweils optimaler verbliebener Stelle auf die Ressourcen. Dabei werden die Aufträge anhand bestimmter Kriterien ausgewählt und nacheinander verplant. Hierbei werden demnach die unterschiedlichen Einsätze mit den dazugehörigen Constraints betrachtet und in eine der Heuristik entsprechenden, sinnvollen Reihenfolge gebracht. Das Resultat hängt hierbei stark von der ermittelten Reihenfolge, sowie den Regeln bei der Einplanung einzelner Einsätze in den Einsatzplan ab.

Operationsbasierte Planung teilt die Aufträge in die einzelnen Operationen der Aufträge auf und plant diese dann direkt ein ohne die Aufträge weiter zu betrachten. Da jedoch die Einsätze im gegebenen Kontext jeweils nur eine Operation haben, entspricht dieser Ansatz der auftragsbasierten Planung und soll nicht weiter betrachtet werden.

Zeit- und Auftragsbasierte Planung betrachtet nacheinander vorhandene Zeitpunkte und wählt dazu jeweils noch nicht verplante Aufträge aus, die zu diesem Zeitpunkt eingeplant werden können. Bei Anwendung auf den gegebenen Kontext würde die Heuristik somit einen ersten Zeitpunkt wählen und solange auf diesen, sich gegenseitig nicht durch Constraints ausschließende Einsätze verplanen, bis die Maximallast erreicht würde. Das Resultat wäre eine hohe Konzentration von Einsätzen auf wenigen günstigen Zeitpunkten, die alle parallel laufen.

Ressourcenbasierte Planung wählt die jeweiligen Ressourcen und verplant dann die Aufträge auf sie. Da jeder Auftrag nur auf einer Ressource laufen kann, da die Betriebsmodi in den Einsätzen den Geräten direkt zugeordnet sind, würde eine abgeschwächte auftragsbasierte Planung durchgeführt werden. Der Unterschied läge darin, dass nicht die Aufträge, also die Einsätze mit ihren Constraints, betrachtet werden würden, um eine Planungsreihenfolge zu erstellen, sondern die Ressourcen, also die Geräte selbst.

Weitere Heuristiken betrachten opportunistische Ansätze, bei denen die Reihenfolge anhand der Menge der Constraints festgelegt wird. Diese sollen jedoch nicht weiter betrachtet werden, da der Einsatz von Constraint-Solvern für die Anforderung als Überdimensioniert erscheint.

Verbesserungsheuristiken erzeugen im Gegensatz zu den Konstruktionsheuristiken nicht eine initial gute Lösung sondern arbeiten auf einem bereits bestehenden initialen Plan, welcher dann durch unterschiedliche Verfahren bis zu der Erfüllung eines gewissen Abbruchkriterium verbessert werden soll. Dazu ermitteln diese Heuristiken anhand von möglichst kleinen Veränderungen am Originalplan Nachbarpläne von denen dann einer ausgewählt wird. Die einfachste Art hierbei ist das Hill-Climbing, bei dem nur Nachbarpläne verwendet werden, die besser sind als der zuvor betrachtete Plan. Da hierbei jedoch lokale Optima nicht verlassen werden können, werden häufig andere Verfahren verwendet, die schlechtere Pläne bis zu einem gewissen Grad zulassen. Die bekanntesten dieser Verfahren sind Simulated Annealing, Treshold Accepting oder der Sintflutalgorithmus. All diese Verfahren können für das betrachtete Planungsproblem in der Software verwendet werden, jedoch tritt hier wieder das gleiche Problem wie bereits bei den exakten Lösungsverfahren auf: Dadurch, dass eine minutengenaue Verplanung möglich ist, sich jedoch die Einplanung über große Zeiträume verteilen kann, werden sehr viele der Lösungen die gleiche Güte haben, so dass der Lösungsraum eine große Bereiche an Nachbarlösungen der gleichen Güte besitzt, in denen sich die Verfahren lange aufhalten würden. Daher ist nicht sicher gestellt, dass die Verfahren diese Gebiete bis zur Erfüllung des Abbruchkriteriums verlassen könnten, selbst wenn eine Hilfestellung wie eine Tabu-Liste verwendet würde.

Weitere Ansätze

Andere Ansätze der Ablaufplanung betrachten zur Lösung noch neuronale Netze, Schwarm-Intelligenz, Genetische Algorithmen oder verteilte Ablaufplanung mit Agentensystemen. Diese Ansätze betrachten zumeist sehr komplexe Strukturen und bieten dazu gute Methoden an. Da es sich bei dem Planungsproblem in der zu entwickelnden Software jedoch um eine überschaubare Komplexität handelt, sollen diese Ansätze nicht genauer betrachtet werden.

16.6.2. Verwendeter Ansatz

Im Folgenden wird dargestellt, welches der vorgestellten Lösungsverfahren verwendet werden soll.
Als erstes wird betrachtet, ob es sinnvoll wäre, eines der exakten Lösungsverfahren zu verwenden. Dazu eine kurze Rechnung: Falls im Einsatzplan nur zehn Einsätze vorhanden sind und jeder dieser Einsätze um eine Stunde verschoben werden kann, was einem sehr kleinen Szenario entspricht, so ergeben sich dadurch, dass jeder Einsatz innerhalb dieser Stunde minutengenau verschoben werden kann und gleichzeitig Tarife über längere Zeit einen konstanten Wert haben, bereits 10^{60} verschiedene Lösungen. Beim Verfahren *Brute-Force* müssten diese alle erzeugt und verglichen werden. Bei *Branch & Bound* würde erwartungsgemäß nur ein kleiner Teil nicht erzeugt werden müssen, und dies auch nur dann, wenn bei mindestens einem der Einsätze in seiner Laufzeit unterschiedliche Preise anhand des Tarifs gelten, wobei alle anderen Teilbäume erst bei den Blättern erkennen, dass sie eine identische Qualität haben wie bereits erstellte Teilbäume. Demnach sind diese Ansätze nicht geeignet, um dem Anwender in angemessener Laufzeit eine Lösung zu bieten.
Weiter werden die Konstruktionsheuristiken betrachtet, wobei nur auf die *auftragsbasierten* und *zeit- und auftragsbasierten* eingegangen werden soll. Bei der *auftragsbasierten Planung* sind der Analyse nach, bei Verwendung eines geeigneten Auswahlkriteriums gute Lösungen in sehr kurzer Zeit zu erwarten. Da große Zeiträume gleicher Preise im Tarif zu erwarten sind und durch die globalen Constraints, nach vielen Einplanungsschritten nur geringe Bereiche ausgeschlossen werden sollten, ist bei diesem Ansatz bereits eine sehr starke Annäherung an die optimale Lösung zu erwarten. Im Gegensatz dazu sind wie schon beschrieben die wenigen Ausschlüsse durch globale Constraints ein Hindernis beim Verwenden der *zeit- und auftragsbasierten Planung*. Dies könnte der Anwender zwar durch Vergabe von weiteren Constraints an die Einsätze umgehen. Da jedoch mit der auftragsbasierten Planung eine funktionierende Heuristik vorhanden ist, sollte dies nicht notwendig sein.
Zuletzt sollen die Verbesserungsheuristiken betrachtet werden. Wie beschrieben tritt hier das Problem auf, dass sich die Verfahren in Bereiche großer Plateaus suchen würden. Da bereits durch die Verwendung einer Konstruktionsheuristik eine sehr gute Lösung zu erwarten ist, würden durch diese Verfahren in den meisten Fällen lediglich die Laufzeiten erhöht werden, ohne dass merkliche Verbesserungen am Plan zu erwarten wären. Daher soll bei der Planung auf die Verwendung von Verbesserungsheuristiken verzichtet werden.
Insgesamt wird demnach eine *auftragsbasierte Planung* eingesetzt werden, zu der im Speziellen das Auswahlkriterium spezifiziert werden muss. Da an dieser Stelle die Qualität der Heuristik entscheidend beeinflusst werden kann, sollen hier verschiedene Ansätze umgesetzt und miteinander verglichen werden. Durch einen Vergleich der unterschiedlichen implementierten Auswahlkriterien kann die am besten Geeignete ausgewählt werden.

Weiterführende Literatur:

- Sauer, Jürgen: Intelligente Ablaufplanung in lokalen und verteilten Anwendungsszenarien, Teubner, 2004.
- Eilers, Mark: Optimierungsheuristiken, Seminar zur Projektgruppe: Demonstrator InHouse Energie-Management, Oldenburg, 2007.

17. Architektur

Autoren: Moritz Eysholdt, Michael Belasus und Malte Hurrelmann

Die Architektur der Anwendung identifiziert ihre Komponenten, deren Zusammenspiel und die Einsatzorte der verwendeten Frameworks.

17.1. Übersicht

Im Folgenden wird die Architektur der zu entwickelnden Software vorgestellt. Diese teilt sich auf in einen lokal laufenden Rich Client und einen über Netzwerk erreichbaren Server. Obwohl der Client auch Out-of-the-box und ohne jegliche Netzwerkanbindung funktionstüchtig ist, besteht dennoch die Möglichkeit, über Netzwerk vom Server die Geräte- und Tarifdaten zu aktualisieren. Außerdem ist es möglich selbst definierte Geräte an den Server zu übermitteln und so die Gerätebibliothek des Servers zu erweitern.

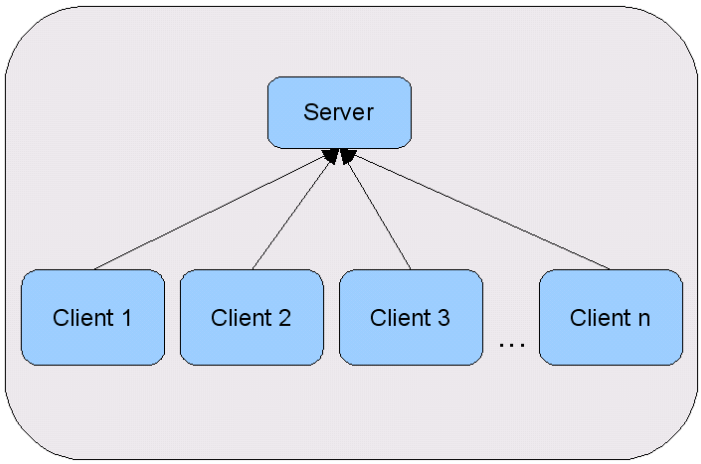

Abbildung 81: Client Server Struktur

Die Abbildung 81 stellt das Zusammenspiel zwischen Client und Server schematisch dar.

17.2. Client

Zunächst wird die Struktur des Clients vorgestellt, sie ist in Abbildung 82 visualisiert. Verwendet wird die bereits im Abschnitt 16.1 beschriebene Eclipse Rich Client Platform. Abbildung 82 zeigt diese und die verwendeten Komponenten im unteren Bildabschnitt (Eclipse Platform). Das Eclipse Modeling Framework (EMF) wurde in Abschnitt 16.2 und das Graphical Editing Framework (GEF) in Abschnitt 25.6 vorgestellt. XSD ist ein auf dem EMF aufbauendes Eclipse-Plugin, welches dem EMF die Verarbeitung von XML Schemas ermöglicht.

17.2.1. Plugins

Die zu entwickelnde Software wird aus mehreren zu implementierenden Eclipse-Plugins bestehen, welche auf den genannten Eclipse-Komponenten aufbauen. Diese sind in der Abbildung 82 unter DiemPlugins zusammengefasst worden. Jedes Plugin entspricht in etwa einer bereits in der Anforderungsdefinition identifizierten Komponente. Dieser modulare Aufbau fördert eine klare Aufgabenteilung bei der Implementierung. Jede gerichtete Kante in Abbildung 82 definiert eine Abhängigkeit. Ein Plugin kann auf sämtliche Funktionalität der Plugins zugreifen, von denen es abhängig ist. Die in Abbildung 82 genannten Plugins werden im kommenden Abschnitt kurz vorgestellt. Eine detailliertere Beschreibung der Plugins findet sich in Kapitel 19 “Plugins“. Über die im Folgenden genannten Abhängigkeiten hinaus verwendet jedes Plugin diem.ModelEdit und diem.Model, sofern es nicht anders beschrieben wird.

diem.App (Abschnitt 19.1) ist der Einstiegspunkt in die Anwendung. Es definiert die Perspektiven des Workbenches und läd alle weiteren Plugins. Dadurch ist diem.App effektiv von allen anderen Plugins abhängig.

diem.Schedule (Abschnitt 19.6) enthält die Benutzungsschnittstelle zur Verwaltung von Einsatzplänen und verwendet die Plugins diem.Device, diem.PayScale und diem.LoadCurve. Zur automatischen optimierung des Einsatzplanes wird diem.Logic verwendet.

diem.Profile (Abschnitt 19.5) enthält die Benutzungsschnittstelle zur Verwaltung von Profilen und verwendet die Plugins auf diem.Device, diem.PayScale, diem.LoadCurve und diem.Schedule.

diem.Logic (Abschnitt 19.10) enthält die Logik für die automatische Einsatzplanoptimierung und ist ausschließlich abhängig von diem.Model.

diem.Device (Abschnitt 19.2) enthält die Benutzungsschnittstelle zur Verwaltung von Geräten, Gerätesammlung und Gerätebibliothek.

diem.PayScale (Abschnitt 19.3) enthält die Benutzungsschnittstelle zur Verwaltung von Tarifen, der Tarifsammlung und der Tarifbibliothek.

diem.LoadCurve (Abschnitt 19.4) enthält die Benutzungsschnittstelle zur Verwaltung von Lastkurven und der Lastkurvensammlung.

diem.Model (Kapitel 18 und Abschnitt 19.8) enthält das Datenmodell und hat auf kein anderes Plugin Abhängigkeiten. Das Datenmodell wird basierend auf EMF (Abschnitt 16.2) generiert.

diem.ModelEdit (Kapitel 18 und Abschnitt 19.9) enthält automatisch generierte Adapter-Klassen, welche grafische Elemente mit dem Datenmodell verbinden können.

diem.Shared (Abschnitt 19.7) enthält von mehreren Plugins genutzte Funktionalität (Widgets, Helferklassen), hat aber selbst keine Abhängigkeiten außer auf das Datenmodell.

17.2.2. Eclipse Platform

Der untere größere Block in Abbildung 82 zeigt die verwendeten Plugins der Eclipse-Platform.

Equinox ist die Laufzeitumgebung aller Eclipse-Plugins und wird unter Abschnitt 16.1 erläutert.

SWT, JFace und UI bilden die RCP und werden ebenso unter Abschnitt 16.1 erläutert.

GEF ist ein in Abschnitt 25.6 beschriebenes Framework zur Erstellung von grafischen Editoren.

EMF bildet das Grundgerüst für das Datenmodell und wird in Abschnitt 16.2 näher beschrieben.

XSD liefert die Unterstützung für XML-Schemas zur Beschreibung von Metamodellen für das EMF.

17.2.3. Persistenz

Wie es Abbildung 82 zeigt, können die Daten des Datenmodelles auf einem Speichermedium (Harddisk) persistiert werden. Hierzu werden sie als XML serialisiert. Abschnitt 18.2 und Abschnitt 19.8 beschreiben diesen Vorgang genauer.

17.2.4. Kommunikation mit dem Server

Der Client kann seine Gerätebibliothek und seine Tarifbibliothek über den Server aktualisieren, eigene Geräte zum Server hochladen und so die Gerätebibliothek ergänzen. Die Client-Server-Kommunikation wird in Abschnitt 20.4 erläutert.

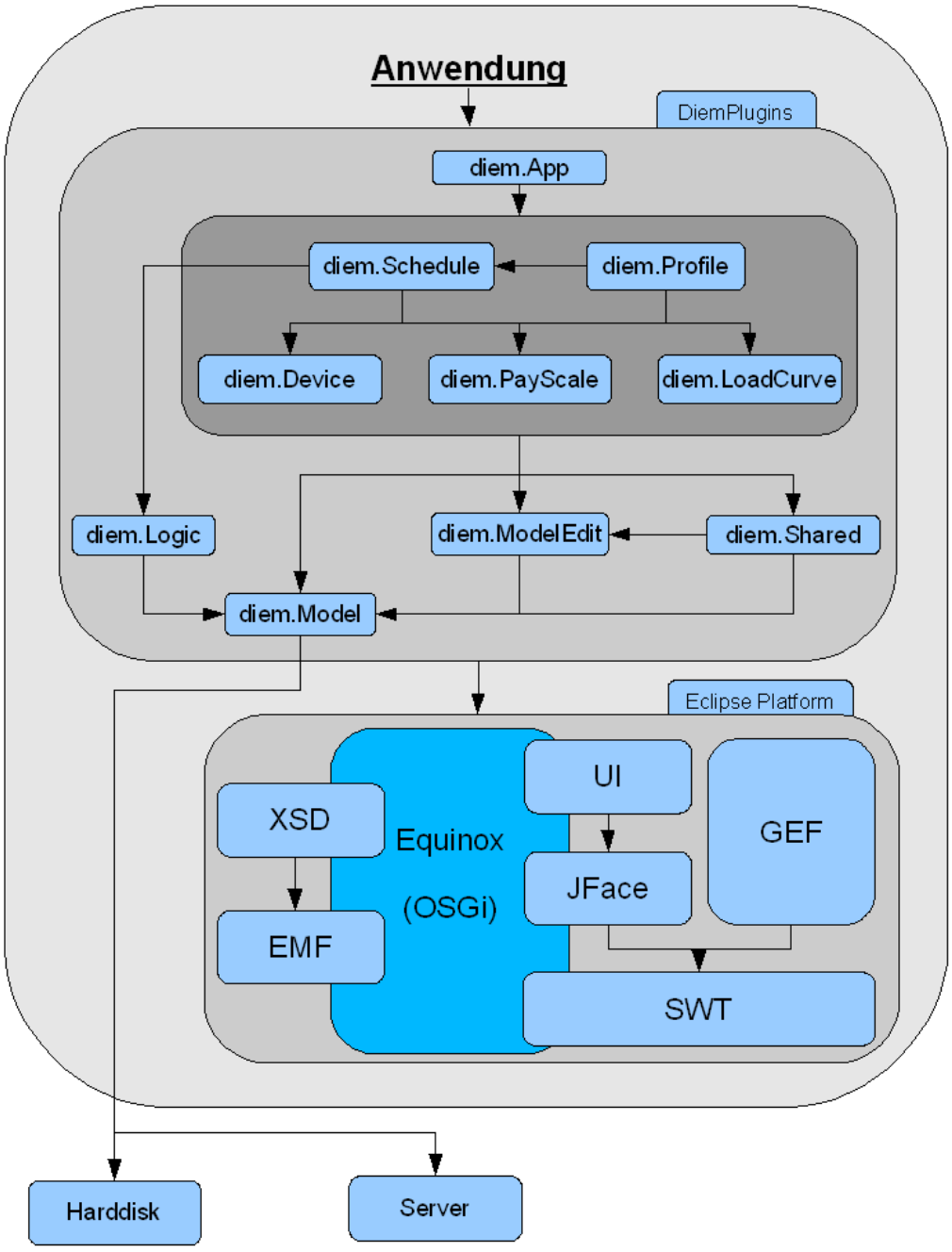

Abbildung 82: Client Struktur

17.3. Server

Dieses Kapitel gibt einen knappen Überblick über die Architektur des Servers, Details werden im Abschnitt 25.5 erläutert.

Der Server ist eine in einem Servlet-Container laufende Webanwendung. Da der Fokus auf dem Client liegt und der Server lediglich zur Aktualisierung der Geräte- und Tarifbibliothek benötigt wird, soll der Server bewusst einfach gehalten werden.

Sowohl Client als auch Server arbeiten auf dem gleichen Datenmodell, was die Datenhaltung vereinheitlicht und die Kommunikation zwischen beiden vereinfacht.

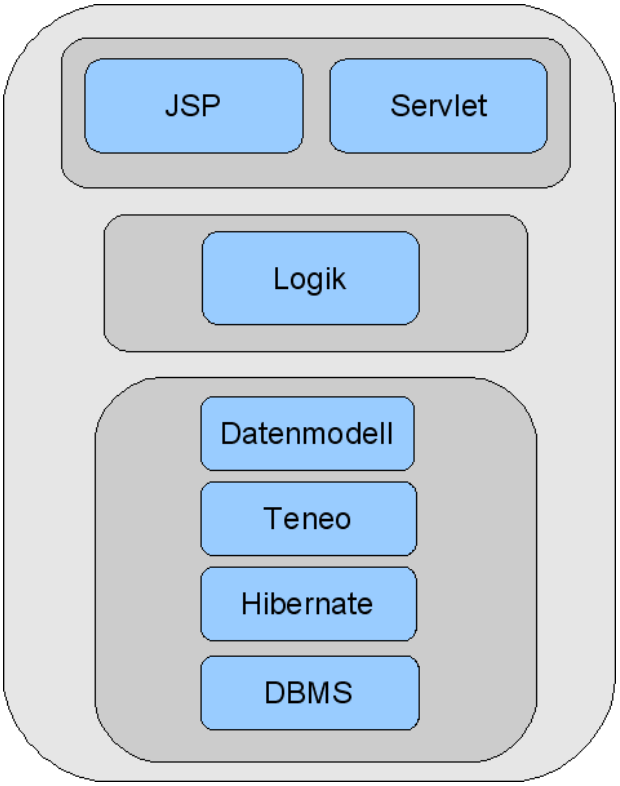

Abbildung 83: Server Struktur

Die Abbildung 83 zeigt die Struktur des Servers. JSPs produzieren die Webseiten und Servlets stehen für die Kommunikation mit dem Rich-Client bereit, verarbeiten POST-Requests und liefern XML aus. Frameworks wie Java Server Faces (JSF), Struts und Spring wurden aufgrund der einfachen Anforderungen für den Server als Überdimensioniert befunden.
Eine simple Logik organisiert den Datenaustausch zwischen Datenmodell und Servlets/JSPs. Es wird das selbe Datenmodell wie im Client verwendet, jedoch werden seine Daten nicht im Arbeitsspeicher vorrätig gehalten, sondern in einer Relationalen Daten-

bank abgelegt. Dazu generiert Teneo aus dem EMF-Modell Hibernate-Mappings und Hibernate verwaltet den Zugriff auf die Datenbank. Hibernate ist dabei in der Lage, die aus der Datenbank angefragen Daten direkt in den Objekten des EMF-Datenmodells abzulegen.
Dank Hibernate ist es nicht notwendig sich auf einen speziellen Datenbankserver fest zu legen. In diesem Projekt wird voraussichtlich PosgreSQL zum Einsatz kommen, da es bereits auf dem für das Projekt angedachten Server verfügbar ist.
Alternativ zu Teneo, Hibernate und einer relationalen Datenbank wäre es ebenso möglich, JPOX als Persistenzframework und damit als Alternative zu Hibernate einzusetzen. Anstelle von Teneo würde dann der EMF-JPOX-Connector treten, welcher ebenso wie Teneo von der Firma Elver entwickelt wird. Mit dem Einsatz von JPOX würde sich die Möglichkeit erschließen, nicht nur relationale Datenbanken, sondern auch Objektorientierte, wie z.B. db4o, einzusetzen. Die Hibernate-Lösung wird jedoch den Ansprüchen eines Demonstrators vollständig gerecht werden und bietet den Vorteil, dass während der Entwicklung bereits im Team vorhandene Kompetenzen bzgl. relationale Datenbanken und Hibernate genutzt werden können.

18. Datenmodell

Autor: Moritz Eysholdt
Das Datenmodell ist als XML-Schema spezifiziert, woraus sich unter Verwendung des Eclipse Modeling Frameworks (EMF) folgende Vorteile ergeben: Ein beachtlicher Anteil des benötigten Quellcodes und der Dokumentation kann automatisch generiert werden und die Daten lassen sich sowohl als XML serialisieren sowie in einer relationalen Datenbank persistieren. Des Weiteren ist die Beschreibung des Datenmodells mittels XML-Schemas vollständig. Sie enthält sowohl die formale Spezifikation des Datenmodells als auch die informale Dokumentation. Damit sind Code und Dokumentation nach dem bewährten Konzept von JavaDoc am selben Ort abgelegt und laufen nicht Gefahr über die Zeit inkonsistent zu werden.

18.1. Codegenerierung

Aus den XML-Schemas werden folgende Artefakte des Projektes generiert:

- **Latex**
  Alle Kapitel des Entwurfes welche den Namen “Schema *.xsd” tragen, wurden automatisch und vollständig aus den XML-Schemas generiert. Lediglich die Grafiken wurden aus dem Eclipse-XSD-Editor der Web Tool Platform (WTP) exportiert. Für die Generierung wurde die selbst geschriebene Java-Anwendung XSD2Latex verwendet, sie befindet sich im Subversion-Repository des Projektes.

- **Java-Datenmodell**
  Mit dem EMF wird ein vollständiges Java-Datenmodell aus den XML-Schemas

generiert. Jeder im XML-Schema definierte Datentyp wird hierzu auf eine Java-Bean abgebildet und aus jedem Element des Datentyps wird eine Eigenschaft der Bean, für welche Getter- und Setter-Methoden zur Verfügung gestellt werden. Für jede Bean wird sowohl ein Java-Interface als auch die zugehörige Java-Implementierung generiert. Des Weiteren ist es möglich sich mittels Observer-Patterns in das Datenmodell einzuklinken und so über Änderungen automatisch informiert zu werden.

- **RCP-Helferklassen**
  Für alle Typen des Datenmodells werden Helferklassen für die Integration des Datenmodells in eine Eclipse-RCP-Benutzungsschnittstelle generiert. Sie implementieren eine Vielzahl von Provider-Interfaces der RCP.

- **Datenmodelleditor**
  Der vom EMF generierte Datenmodelleditor erlaubt es, XML-Dateien, welche den XML-Schemas des Datenmodells genügen, visuell in einer Baumansicht zu bearbeiten. Er wird nicht Teil des Diem-Produktes sein, ist jedoch hilfreich um schnell Testdaten zu erstellen und die Entwickler durch Beispielcode zu inspirieren.

- **Hibernate-Mappings**
  Das Projekt Teneo erlaubt es, zur Laufzeit aus einem EMF-Datenmodell Hibernate-Mappings zu generieren welche die Persistierung des Datenmodells in einer relationalen Datenbank erlauben. Diese Technologie soll für den Server des Projektes verwendet werden.

- **DDL-Code**
  Aus den Hibernate-Mappings wiederum generiert Hibernate SQL-Code in der Data Definition Language (DDL) um in einer unterstützten relationalen Datenbank die benötigten Tabellen zu erstellen.

Neben der ersparten Arbeit ist ein nicht zu unterschätzender Vorteil der Codegenerierung die spätere Flexibilität für Änderungen. Sollten Modifikationen am Datenmodell durchgeführt werden, so müssen lediglich die XML-Schemas bearbeitet werden. Die Änderungen werden durch den automatischen Generierungsprozess direkt in alle soeben aufgeführten Artefakte übernommen. Inkonsistenzen zwischen Datenmodell und Anwendungslogik bzw. Benutzungsschnittstelle wird Eclipse detailliert mit Fehlermeldungen berichten. Somit ist es ausgesprochen einfach ein Refactoring am Datenmodell durchzuführen, ohne Gefahr zu laufen, unbemerkt Inkonsistenzen verursacht zu haben.

18.2. Persistenz

Die Daten werden auf zwei verschiedene Arten persistiert:

- **XML**
  Auf der Clientseite werden die Daten in XML-Dateien abgelegt. Jedes XML-Schema

spezifiziert hierbei genau ein Dateiformat. XML-Dateien können sich gegenseitig referenzieren, beispielsweise referenziert ein Einsatzplan die in ihm verwendeten Geräte und Gerätekompositionen. Die für die Serialisierung in XML und die Deserialisierung benötigte Funktionalität wird vom EMF bereitgestellt.

- **Relationale Datenbank**
  Wie bereits unter Codegenerierung beschrieben bildet das Projekt Teneo das Bindeglied zwischen dem EMF-Datenmodell und Hibernate. Damit ist es möglich die Daten in jeder von Hibernate unterstützten Datenbank zu halten.

18.3. Struktur

Von den folgenden Kapiteln (bis auf das Kapitel Constraints) ist jedes genau einem XML-Schema gewidmet, wobei jedes Schema eine Menge vom Typen und jeder Typ eine Menge von Elementen und Attributen enthalten.

- **Schema**
  Ein Schema definiert genau ein Dateiformat, enthält eine Menge von Datentypen und kann andere Schemas referenzieren um deren Datentypen zu verwenden. Das Wurzelelement gibt den Namen und den Datentypen des Wurzel-Knotens im XML-Dokument an.

- **Typ**
  Hat ein Typ Elemente und/oder Attribute, so entspricht er dem ComplexType eines Schemas, ansonsten ist er ein SimpleType. Ein Typ kann andere Typen erweitern, was im Java-Code dem Konzept der Vererbung entspricht. Wenn ein Typ einen direkten Bezug zu einer im Anforderungsdokument definierten Datenstruktur hat, so ist dieser Bezug angegeben.

- **Attribut**
  Ein Datentyp kann beliebig viele Attribute haben. Im Java-Datenmodell sind sie Eigenschaften einer Bean.

- **Element**
  Ebenso kann ein Datentyp beliebig viele Elemente haben, auch diese werden zu Eigenschaften einer Java-Bean. Zwischen Attributen und Elementen wird im Java-Datenmodell also nicht unterschieden.
  Elemente können andere Elemente referenzieren, wofür zwei verschiedene Verfahren verwendet werden:
  - **IDREF**
    Das Referenzierte Element (bzw. sein Datentyp) muss ein Attribut oder Element mit dem Datentypen ID haben, welches ihn eindeutig identifiziert. IDREF verweist auf genau diese ID und es ist nur eine Referenzierung innerhalb derselben XML-Datei möglich.

- **anyURI**
  Die Referenz wird als relativer Dateiname ergänzt um eine XPath-Beschreibung definiert. Somit sind auch dateiübergreifende Referenzen möglich.

Im Java-Quellcode sind Referenzen nicht als solche erkennbar, das EMF erstellt Bean-Eigenschaften welche direkt und typensicher auf die referenzierte Bean verweisen. Befindet sich das referenzierte Element in einer anderen Datei, so wird dieses dynamisch nachgeladen.
Elemente und Attribute mit den folgenden Namen werden in diesem Entwurf immer wieder verwendet, sind jedoch an ihren Verwendungsorten nicht dokumentiert, da sie immer den gleichen Zweck verfolgen:

- **Name**
  Das Element hat einen vom Nutzer spezifizierten Namen, welcher nicht eindeutig sein muss, solange es nicht anders angegeben ist. Er kann dafür verwendet werden, die Dateinamen von zu speichernden XML-Dateien abzuleiten.
- **CreationDate**
  Das Erstellungsdatum speichert den Zeitstempel, wann der Datensatz erstellt wurde.
- **Description**
  Die Beschreibung ist ein beliebiger vom Nutzer vergebener Kommentar.
- **id**
  Die ID ist ein auf Basis eines Zufallsgenerators und eines Zeitstempels automatisch generierter String, welche global eindeutig ist. Die ID beginnt mit den ersten drei Buchstaben des Datentyps.

18.4. Schema loadcurve.xsd

Das Schema loadcurve.xsd beschreibt alle Datenstrukturen eines Lastgangs. Ein Lastgang beschreibt einen Stromverbrauch in Abhängigkeit von der Zeit.

Wurzelelement: `LoadCurve:LoadCurve`

18.4.1. Typ LoadCurve

Quelle im Anforderungsdokument: Annahmen->Lastgang
LoadCurve beschreibst einen Lastgang. Der Lastgang wird angegeben als eine Liste von Stromverbrauchsmessungen, welche in beliebigen Abständen von einander stattgefunden haben können. Die Länge eines Lastgangs ergibt sich aus dem Zeitpunkt der letzten Stromverbrauchsmessung. (siehe Abbildung 84)

- Element `Name:string[1]`

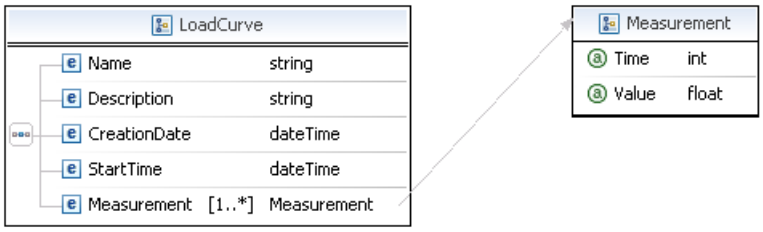

Abbildung 84: Typ LoadCurve

- Element `Description:string[1]`
- Element `CreationDate:dateTime[1]`
- Element `StartTime:dateTime[1]` StartTime gibt den absoluten Zeitpunkt des Beginns der Lastgangmessung an.
- Element `Measurement:Measurement[1..*]` Ein Lastgang muss mindestends eine Verbrauchsmessung enthalten.

18.4.2. Typ Measurement

Ein Stromverbrauchsmessung ist eine Wertepaar bestehend aus einem Zeitpunkt und einem Messwert.

- Attribut `Time:int` Time gibt die Anzahl der vergangenen Sekunden seit StartTime an. Constraint: Ein Lastgang darf nicht zwei Messungen für den selben Zeitpunkt haben, d.h. Time muss unique sein.
- Attribut `Value:float` Der Messwert gibt die durchschnittliche Leistungsaufnahme seit der letzten Messung in Watt [W] an.

18.5. Schema payscale.xsd

Das Schema payscale.xsd definiert alle zu einem Tarif gehörigen Datenstrukturen.

Wurzelelement: `PayScale:PayScale`

18.5.1. Typ PayScale

Quelle im Anforderungsdokument: Annahmen->Tarif, Annahmen->Fixtarif, Annahmen->Variabler Tarif
PayScale definiert einen Tarif, d.h. einen von einem Energielieferanten festgelegten Strompreis. PayScale kann sowohl einen Fixtarif als auch einen variablen Tarif definieren. (siehe Abbildung 85)

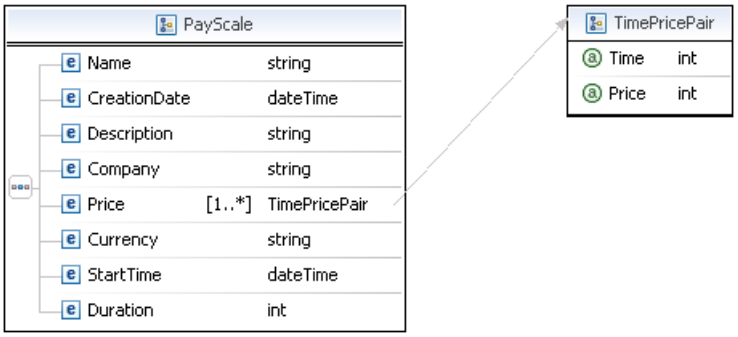

Abbildung 85: Typ PayScale

- Element `Name:string[1]`
- Element `CreationDate:dateTime[1]`
- Element `Description:string[1]`
- Element `Company:string[1]` Company gibt den Firmennamen des Energielieferanten an.
- Element `Price:TimePricePair[1..*]` Ein Fixtarif enthält genau ein TimePricePair, ein variabler Tarif mehr als ein TimePricePair.
- Element `Currency:string[1]` Currency gibt die Währung der in TimePricePair angegebenen Strompreise an.
- Element `StartTime:dateTime[1]` StartTime definiert den Zeitpunkt, ab welchem der Tarif gültig ist.
- Element `Duration:int[0..1]` Duration definiert die Gültigkeitsdauer des Tarifs in Sekunden. Der Endzeitpunkt der Gültikeit des Tarifs lässt sich berechnen, indem StartTime um Duration Sekunden inkrementiert wird. Die Gültigkeitsdauer muss nicht festgelegt werden.

18.5.2. Typ TimePricePair

Ein TimePricePair beschreibt ein Tupel bestehend aus einer Zeit und einem Preis.

- Attribut `Time:int` Time gibt die Anzahl der vergangenen Sekunden seit StartTime an. Constraints: Der Wert von Time des ersten TimePricePairs eines Tarifs muss 0 sein; Time ist über alle TimePricePairs eines Tarifes unique; Der Wert von Time muss immer kleiner sein als Duration des Tarifs.

- Attribut `Price:int` Price gibt den Strompreis an, welcher ab dem durch Time definierten Zeitpunkt gültig ist, bis entweder ein anderes TimePricePair einen anderen Tarif definiert oder der Gültigkeitszeitraum des Tarifes vorüber ist. Damit ist das durch ein TimePricePair definierte Intervall nach oben offen und nach unten geschlossen.

18.6. Schema payscalelib.xsd

Die Tarifbibliothek ist eine Sammlung von Tarifen, welche in einer hierarchischen Baumstruktur einsortiert werden. Im Gegensatz zum Profil enthält die Tarifbibliothek direkt die Instanzen der Tarife und referenziert sie nicht nur.

Wurzelelement: `PayScaleLib:PayScaleLib`
Referenzierte Schemas: `payscale.xsd`

18.6.1. Typ PayScaleFolder

Ein Tarifordner ist ein Knoten in der hierarchischen Baumstruktur der Tarifbibliothek.

- Attribut `Name:string` Der Name identifiziert einen Tarifordner. Durch die hierarchische Struktur kann ein Tarifordner im Baum durch eine Pfadangabe identifiziert werden, welche sich aus den Namen aller Väterknoten des Tarifordners und seinem eigenen Namen zusammensetzt. Constraint: Der Name muss eindeutig gegenüber den Geschwisterknoten sein. Geschwisterkonoten sind alle Knoten die den selben Vaterknoten haben.
- Element `PayScaleFolder:PayScaleFolder[0..*]` Ein Tarifordner kann beliebig viele Unterordner erhalten.
- Element `PayScale:PayScale[0..*]` Ein Tarifordner kann beliebig viele Tarife enthalten. Constraint: Der Name eines Tarifs muss eindeutig sein innerhalb eines Tarifordners.

18.6.2. Typ PayScaleLib

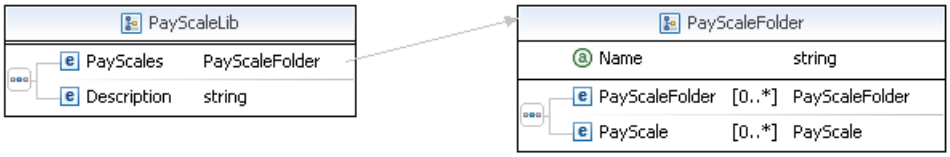

Abbildung 86: Typ PayScaleLib

Quelle im Anforderungsdokument: Annahmen->Online-Bibliothek, Annahmen->Lokale Bibliothek
PayScaleLib ist der Typ des Wurzelelementes der Tarifbibliothek. (siehe Abbildung 86)

- Element `PayScales:PayScaleFolder[1]` PayScales beinhaltet alle Tarife dieser Tarifbibliothek, hierarchisch sortiert in eine Baumstruktur.
- Element `Description:string[1]`

18.7. Schema device.xsd

Das Schema device.xsd spezifiziert die Datenstrukturen für ein Gerät.

Wurzelelement: `Device:Device`
Referenzierte Schemas: `loadcurve.xsd`

18.7.1. Typ Device

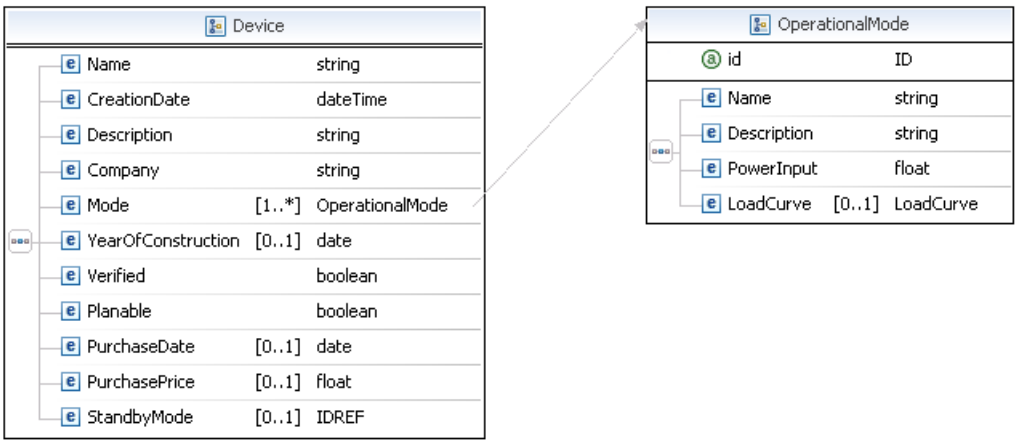

Abbildung 87: Typ Device

Quelle im Anforderungsdokument: Annahmen->Gerät
Ein Gerät definiert einen beliebigen elektrischen Verbraucher. (siehe Abbildung 87)

- Element `Name:string[1]`
- Element `CreationDate:dateTime[1]`
- Element `Description:string[1]`
- Element `Company:string[1]` Company gibt die Herstellerfirma dieses Gerätes an.

- Element `Mode:OperationalMode[1..*]` Ein Gerät kann beliebig viele Betriebsmodi haben, muss jedoch mindestens einen besitzen. Wenn das Gerät über einen Standby-Modus verfügt, so ist dieser auch hier aufgeführt.
- Element `YearOfConstruction:date[0..1]` Gibt das Jahr an, in welchem das Gerät hergestellt wurde.
- Element `Verified:boolean[1]` Dieser Wert ist true, wenn die Korrektheit der in diesem Datensatz gespeicherten Werte von einer Autorität bestätigt wurde.
- Element `Planable:boolean[1]` Dieser Wert ist true, wenn der Anwender die Einsatzplanung dieses Gerätes zum Teil oder ganz der Software überlassen kann und will. Planbare Geräte können vom Einsatzplan-Optimierungsalgorithmus im Einsatzplan verschoben werden.
- Element `PurchaseDate:date[0..1]` PurchaseDate ist das Datum, zu welchem das Gerät gekauft wurde.
- Element `PurchasePrice:float[0..1]` PurchasePrice ist der Preis, welcher für das Gerät beim Kauf bezahlt wurde.
- Element `StandbyMode:IDREF[0..1]` Besitzt das Gerät einen Standby-Modus, so wird er hier referenziert.

18.7.2. Typ OperationalMode

Quelle im Anforderungsdokument: Annahmen->Betriebsmodus
Ein Betriebsmodus ist einem Gerät zugeordnet und beschreibt dessen Leistungsaufnahme bei einer bestimmten Gerätekonfiguration.

- Attribut `id:ID`
- Element `Name:string[1]`
- Element `Description:string[1]`
- Element `PowerInput:float[1]` Die durchschnittliche Leistungsaufnahme des Gerätes in diesem Betriebsmodus, angegeben in Watt [W].
- Element `LoadCurve:LoadCurve[0..1]` Der Lastgang für diesen Betriebsmodus. Wenn er angegeben ist, sollte PowerInput seinem Durchschnittswert entsprechen. Für Details siehe loadcurve.xsd.

18.8. Schema devicelib.xsd

Die Gerätebibliothek ist eine Sammlung von Geräten, welche in einer hierarchischen Baumstruktur einsortiert sind. Im Gegensatz zum Profil enthält die Gerätebibliothek direkt die Instanzen der Geräte und referenziert sie nicht nur.

Wurzelelement: `DeviceLib:DeviceLib`
Referenzierte Schemas: `device.xsd`

18.8.1. Typ DeviceFolder

Ein Geräteordner ist ein Knoten in der hierarchischen Baumstruktur der Gerätebibliothek.

- Attribut `Name:string` Der Name identifiziert einen Geräteordner. Durch die hierarchische Struktur kann ein Geräteordner im Baum durch eine Pfadangabe identifiziert werden, welche sich aus den Namen aller Väterknoten des Geräteordners und seinem eigenen Namen zusammensetzt. Constraint: Der Name muss eindeutig gegenüber den Geschwisterknoten sein. Geschwisterkonoten sind alle Knoten die den selben Vaterknoten haben.
- Element `Folder:DeviceFolder[0..*]` Ein Geräteordner kann beliebig viele Unterordner erhalten.
- Element `Device:Device[0..*]` Ein Geräteordner kann beliebig viele Geräte enthalten. Constraint: Der Name eines Gerätes muss eindeutig sein innerhalb eines Geräteordners.

18.8.2. Typ DeviceLib

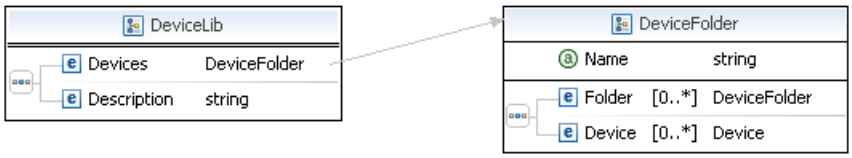

Abbildung 88: Typ DeviceLib

Quelle im Anforderungsdokument: Annahmen->Online-Bibliothek, Annahmen->Lokale Bibliothek
DeviceLib ist der Typ des Wurzelelementes der Gerätebibliothek. (siehe Abbildung 88)

- Element `Devices:DeviceFolder[1]` Devices beinhaltet alle Geräte dieser Gerätebibliothek, hierarchisch sortiert in eine Baumstruktur.
- Element `Description:string[1]`

18.9. Schema devcompos.xsd

Das Schema devcompos.xsd (DeviceComposition) enthält alle Datenstrukturen für die Gerätekomposition.

Wurzelelement: `DeviceComposition:DeviceComposition`
Referenzierte Schemas: `device.xsd`

18.9.1. Typ Constraint

Quelle im Anforderungsdokument: Annahmen->Constraint
Constraint ist der abstrakte Basistyp für alle Constraint-Typen. Er sorgt dafür, dass alle Constaints einen Namen haben und er ermöglicht in der Programmlogik den Einsatz von Polymorphie.

- Element `Name:string[1]`

18.9.2. Typ DeviceComposition

Quelle im Anforderungsdokument: Annahmen->Gerätekomposition
Eine Gerätekomposition beschreibt den Einsatz von Geräten und Gerätekompositionen in Abhängigkeit von der Zeit, wobei die Zeit nur relativ zum Beginn der Gerätekomposition definiert wird. Eine Gerätekomposition kennt keine festen Start- und Endzeiten. Neben Einsätzen (Tasks) kann die Gerätekomposition Constraints enthalten, welche als Regeln für die Einsatzplanoptimierung zu verstehen sind. (siehe Abbildung 89)

- Element `Name:string[1]`
- Element `CreationDate:dateTime[1]`
- Element `Description:string[1]`
- Element `Duration:int[1]` Die zeitliche Länge der Gerätekomposition angegeben in Sekunden (s).
- Element `Task:Task[0..*]` Eine Gerätekomposition kann beliebig viele Einsätze enthalten.
- Element `ConstAfterEachOther:ConstAfterEachOther[0..*]` Eine Gerätekomposition kann beliebig viele Constraints vom Typ ConstAfterEachOther enthalten.
- Element `ConstMaxLoad:ConstMaxLoad[0..*]` Eine Gerätekomposition kann beliebig viele Constraints vom Typ ConstMaxLoad enthalten.
- Element `ConstOverlap:ConstOverlap[0..*]` Eine Gerätekomposition kann beliebig viele Constraints vom Typ ConstOverlap enthalten.

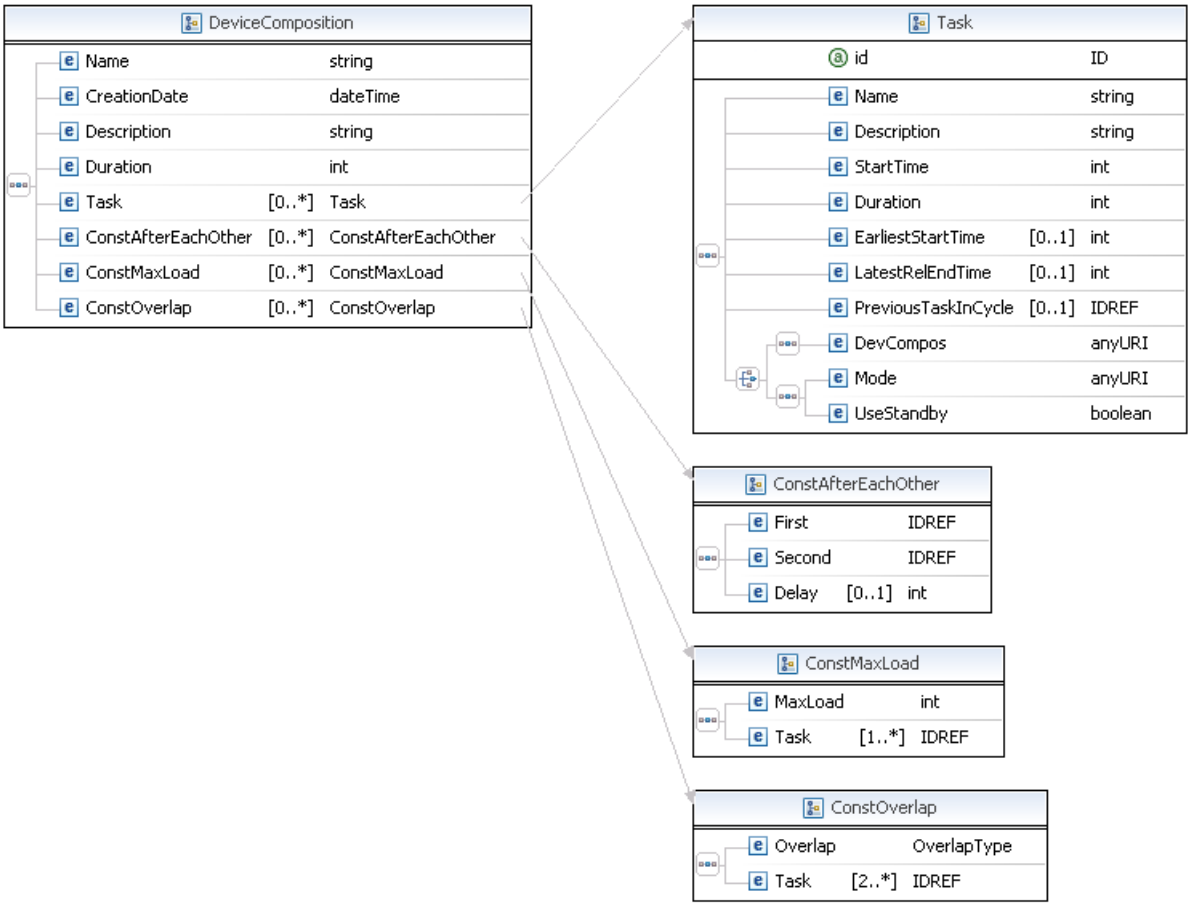

Abbildung 89: Typ DeviceComposition

18.9.3. Typ OverlapType

Der OverlapType definiert, dass zwei oder mehr Einsätze folgendermaßen zu einander liegen können: Identical: Sie liegen deckungsgleich. Prohibited: Sie dürfen sich nicht überschneiden. SameStart: Sie haben den gleichen Startzeitpunkt. SameEnd: Sie haben den gleichen Endzeitpunkt.

18.9.4. Typ Task

Quelle im Anforderungsdokument: Annahmen->Einsatz, Annahmen->Zyklusdefinition Ein Einsatz (auch: Geräteeinsatz) beschreibt die konkrete Verwendung eines Gerätes oder einer Gerätekomposition in einem Einsatzplan oder einer Gerätekomposition.

- Attribut `id:ID`
- Element `Name:string[1]`
- Element `Description:string[1]`

- Element `StartTime:int[1]` Die Startzeit gibt den Zeitpunkt der Aktivierung (Einschalten) des Gerätes relativ zum Beginn der Gerätekomposition in Sekunden (s) an.
- Element `Duration:int[1]` Die Dauer definiert, wie lange das Gerät im eingeschalteten Zustand verbleibt. Die Dauer wird in Sekunden (s) angegeben.
- Element `EarliestStartTime:int[1]` Der Wert früheste Startzeit gibt an, wie früh die automatische Einsatzplanoptimierung den Einsatz maximal einplanen darf. Die Zeit wird in Sekunden (s) angegeben. Ist der Einsatz nicht Teil eines Zyklusses, beschreibt der Wert die Anzahl der Sekunden die seit dem Beginn der Gerätekomposition vergangen sind. Gehört der Einsatz zu einem Zyklus (Was daran zu erkennen ist, dass PreviousTaskInCycle einen Einsatz referenziert), so gibt der Wert die vergangenen Sekunden nach LatestRelEndTime des unter PreviousTaskInCycle referenzierten Einsatzes an.
- Element `LatestRelEndTime:int[1]` Spätester relativer Endzeitpunkt gibt an, wie weit die automatische Einsatzplanoptimierung einen Task nach hinten schieben darf. Der Wert gibt die Anzahl der vergangenen Sekunden seit EarliestStartTime an. LatestRelEndTime kann auch als die Länge des Slots in Sekunden betrachtet werden, in welchem der Einsatz eingeplant werden muss.
- Element `PreviousTaskInCycle:IDREF[0..1]` Zyklen werden dadurch definiert, dass alle beteiligten Einsätze sich im Stile einer einfach verketteten Liste referenzieren. In einem Zyklus enthält jeder Einsatz eine Referenz auf den zeitlich davor liegenden Einsatz. Der erste Einsatz in einem Zyklus weiss nur dadurch, dass er zu einem Zyklus gehört, dass es einen anderen Einsatz gibt, welcher ihn referenziert. Ein Einsatz in einem Zyklus zeichnet sich dadurch aus, dass die früheste Startzeit (EarliestStartTime) nicht mehr relativ zum Beginn der Gerätekomposition ist, sondern relativ zum nächsten zeitlich früherliegendem Einsatz im Zyklus. Für den ersten Einsatz in einem Zyklus macht es keinen Unterschied, ob er sich in einem Zyklus befindet oder nicht. Constraint: Die verkettete Liste darf keine Zyklen enthalten.
- Element `DevCompos:anyURI[1]` Wird im Einsatz das Einplanen einer Gerätekomposition beschrieben, so ist diese hier verlinkt. Ein Einsatz kann entweder eine Gerätekomposition oder ein Gerät referenzieren. Constraints: DevCompos darf nicht gesetzt sein, wenn Mode angegeben wurde; Eine Gerätekomposition darf sich nicht selbst enthalten, auch nicht verschachtelt.
- Element `Mode:anyURI[1]` Beschreibt der Einsatz den Einsatz eines Gerätes, so wird hier der ausgewählte Betriebsmodus dieses Gerätes referenziert. Constraint: Mode darf nicht angegeben werden, wenn DevCompos gesetzt wurde.

- Element `UseStandby:boolean[1]` Handelt es sich um einen Geräteeinsatz, so gibt dieser Wert an, ob das Gerät im nicht aktivierten Zustand vollständig ausgeschaltet ist oder im Standby-Mode verweilt. Im Gerät selbst ist definiert, was der Standby-Modus ist. Constraint: UseStandby nur angegeben werden, wenn Mode gesetzt wurde.

18.9.5. Typ ConstAfterEachOther

Quelle im Anforderungsdokument: Annahmen->Constraint
Dieser Constraint definiert, dass der unter First referenzierte Einsatz abgeschlossen sein muss, bevor der unter Second referenzierte Einsatz beginnt.

- Erbt von `Constraint`
- Element `First:IDREF[1]` Der erste referenzierte Einsatz. Constraint: First muss einen anderen Task referenzieren als Second.
- Element `Second:IDREF[1]` Der zweite referenzierte Einsatz. Constraint: Second muss einen anderen Task referenzieren als First.
- Element `Delay:int[1]` Wenn Pause (Delay) angegeben ist, beschreibt sie die Zeit in Sekunden, die nach dem Abschluss des ersten Einsatzes verstreichen muss bevor der Zweite beginnen kann.

18.9.6. Typ ConstMaxLoad

Quelle im Anforderungsdokument: Annahmen->Constraint
Dieser Constraint definiert eine Maximallast welche zu keinem Zeitpunkt überschritten werden darf.

- Erbt von `Constraint`
- Element `MaxLoad:int[1]` Die Maximallast wird in Watt (W) angegeben.
- Element `Task:IDREF[1..*]` Die Beschränkung der maximalen Last gilt für alle hier referenzierten Einsätze. Constraint: Die Liste darf keinen Task mehrmals enthalten.

18.9.7. Typ ConstOverlap

Quelle im Anforderungsdokument: Annahmen->Constraint
Dieser Constraint definiert, wie eine Menge von Einsätzen zu einander liegen müssen. Details beschreibt der Typ OverlapType. (siehe Abbildung 90)

- Erbt von `Constraint`

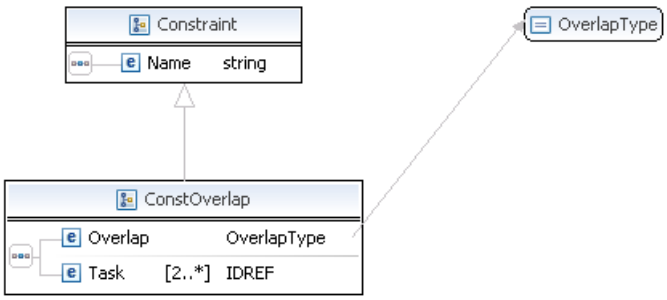

Abbildung 90: Typ ConstOverlap

- Element `Overlap:OverlapType[1]` Overlap gibt an, welcher Regel die referenzierten Einsätze genügen müssen.
- Element `Task:IDREF[2..*]` Diese referenzierten Einsätze müssen dem Constraint genügen.Constraint: Die Liste darf keinen Task mehrmals enthalten.

18.10. Schema schedule.xsd

Das Schema schedule.xsd enthält alle Datenstrukturen für den Einsatzplan.

Wurzelelement: `Schedule:Schedule`
Referenzierte Schemas: `devcompos.xsd`, `payscale.xsd`

18.10.1. Typ Schedule

Quelle im Anforderungsdokument: Annahmen->Einsatzplan
Der Einsatzplan (Schedule) beschreibt den Einsatz von Geräten und Gerätekompositionen in Abhängigkeit von der Zeit. Er erbt die Funktionalität der Gerätekomposition, kennt darüber hinaus jedoch einen absoluten Startzeitpunkt und ihm können Stromtarife zugeordnet werden. (siehe Abbildung 91)

- Erbt von `DeviceComposition`
- Element `StartTime:dateTime[1]` Der Startzeitpunkt des Einsatzplanes.
- Element `PayScale:anyURI[0..*]` Ein Einsatzplan kann beliebig viele Tarife enthalten. Constraint: Zu keinem Zeitpunkt darf mehr als ein Tarif gültig sein.

18.11. Schema profile.xsd

Das Schema profile.xsd definiert alle Datenstrukturen für ein Profil.

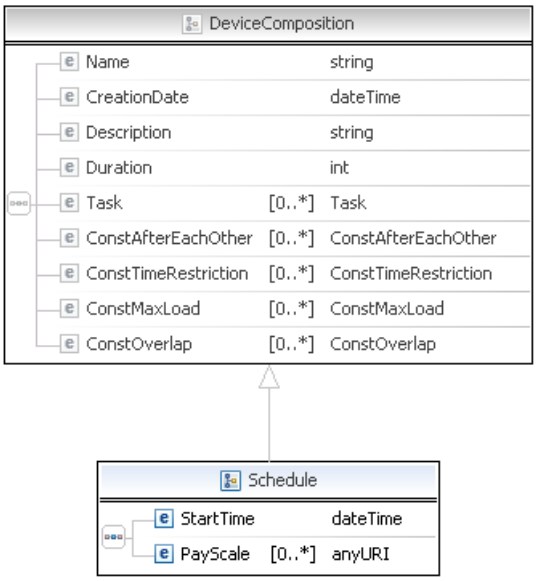

Abbildung 91: Typ Schedule

Wurzelelement: `Profile:Profile`
Referenzierte Schemas: `devcompos.xsd`, `schedule.xsd`, `payscale.xsd`, `device.xsd`, `loadcurve.xsd`

18.11.1. Typ Profile

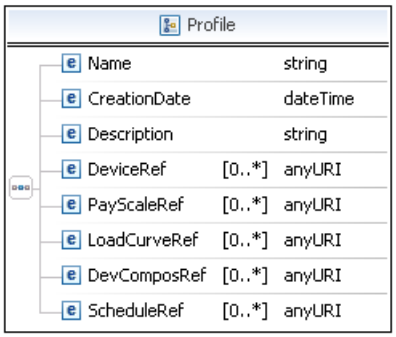

Abbildung 92: Typ Profile

Quelle im Anforderungsdokument: Annahmen->Profil
Ein Profil ist eine Sammlung von Referenzen auf Geräte, Tarife, Lastgänge, Gerätekompositionen und Einsatzpläne. (siehe Abbildung 92)

- Element `Name:string[1]`

- Element `CreationDate:dateTime[1]`
- Element `Description:string[1]`
- Element `DeviceRef:anyURI[0..*]` Ein Profil kann beliebig viele Referenzen auf verschiedene Geräte enthalten.
- Element `PayScaleRef:anyURI[0..*]` Ein Profil kann beliebig viele Referenzen auf verschiedene Tarife enthalten.
- Element `LoadCurveRef:anyURI[0..*]` Ein Profil kann beliebig viele Referenzen auf verschiedene Lastgänge enthalten.
- Element `DevComposRef:anyURI[0..*]` Ein Profil kann beliebig viele Referenzen auf verschiedene Gerätekompositionen enthalten.
- Element `ScheduleRef:anyURI[0..*]` Ein Profil kann beliebig viele Referenzen auf verschiedene Einsatzpläne enthalten.

18.12. Constraints

Dieses Kapitel bietet eine Übersicht über alle Constraints.

- `loadcurve.xsd#Measurement->Time`:
  Ein Lastgang darf nicht zwei Messungen für den selben Zeitpunkt haben, d.h. Time muss unique sein.
- `payscale.xsd#TimePricePair->Time`:
  Der Wert von Time des ersten TimePricePairs eines Tarifs muss 0 sein.
- `payscale.xsd#TimePricePair->Time`:
  Time ist über alle TimePricePairs eines Tarifes unique.
- `payscale.xsd#TimePricePair->Time`:
  Der Wert von Time muss immer kleiner sein als Duration des Tarifs.
- `payscalelib.xsd#PayScaleFolder->Name`:
  Der Name muss eindeutig gegenüber den Geschwisterknoten sein. Geschwisterkonoten sind alle Knoten die den selben Vaterknoten haben.
- `payscalelib.xsd#PayScaleFolder->PayScale`:
  Der Name eines Tarifs muss eindeutig sein innerhalb eines Tarifordners.
- `devicelib.xsd#DeviceFolder->Name`:
  Der Name muss eindeutig gegenüber den Geschwisterknoten sein. Geschwisterkonoten sind alle Knoten die den selben Vaterknoten haben.

- `devicelib.xsd#DeviceFolder->Device`:
  Der Name eines Gerätes muss eindeutig sein innerhalb eines Geräteordners.
- `devcompos.xsd#Task->PreviousTaskInCycle`:
  Die verkettete Liste darf keine Zyklen enthalten.
- `devcompos.xsd#Task->DevCompos`:
  DevCompos darf nicht gesetzt sein, wenn Mode angegeben wurde.
- `devcompos.xsd#Task->DevCompos`:
  Eine Gerätekomposition darf sich nicht selbst enthalten, auch nicht verschachtelt.
- `devcompos.xsd#Task->Mode`:
  Mode darf nicht angegeben werden, wenn DevCompos gesetzt wurde.
- `devcompos.xsd#Task->UseStandby`:
  UseStandby nur angegeben werden, wenn Mode gesetzt wurde.
- `devcompos.xsd#ConstAfterEachOther->First`:
  First muss einen anderen Task referenzieren als Second.
- `devcompos.xsd#ConstAfterEachOther->Second`:
  Second muss einen anderen Task referenzieren als First.
- `devcompos.xsd#ConstMaxLoad->Task`:
  Die Liste darf keinen Task mehrmals enthalten.
- `devcompos.xsd#ConstOverlap->Task`:
  Die Liste darf keinen Task mehrmals enthalten.
- `schedule.xsd#Schedule->PayScale`:
  Zu keinem Zeitpunkt darf mehr als ein Tarif gültig sein.

19. Plugins

Die einzelnen Komponenten der Client-Anwendung wurden in Kapitel 17 als zu realisierende RCP Plugins vorgestellt. Diese Plugins unterscheiden sich hauptsächlich in ihren Funktionalitäten, jedoch nicht im Aufbau. Sie enthalten jeweils grafische Komponenten (Editoren und Views) sowie dazugehörige Actions. Die grafischen Komponenten werden durchgängig je nach Komplexität mit Hilfe von SWT, JFace, Draw2D und/oder GEF erstellt. Sie werden in der Lage sein, Drag&Drop Ereignisse zu erkennen und entsprechend zu behandeln.
In den folgenden Abschnitten sollen diese Plugins nun näher betrachtet und spezifiziert werden.

19.1. diem.App

Autor: Christian Hinrichs

In Abschnitt 16.1 wurde bereits erwähnt, dass eine eigenständige, modularisierte RCP-Anwendung aus mehreren Plugins besteht, von denen eines als zentral angesehen werden kann, da es den eigentlichen Lebenszyklus der Anwendung kontrolliert und durch die anderen Plugins mit Funktionalität erweitert wird. Dieses zentrale Plugin wird in der zu erstellenden Anwendung durch `diem.app` realisiert und beinhaltet ein Grundgerüst an Klassen, welche mit einer Basisausstattung an Funktionalität vom RCP Framework generiert werden. Diese Klassen können dann vom Entwickler konfiguriert und erweitert werden, um die Anwendung an die jeweiligen Bedürfnisse anzupassen. Im Folgenden wird ein grober Überblick über dieses Gerüst gegeben.

19.1.1. Activator.java

Diese Klasse ist von `AbstractUIPlugin` abgeleitet und definiert ein einzelnes Plugin für die Equinox-Umgebung. Sie bietet Funktionalitäten wie etwa eine `ImageRegistry`, welche die Verwaltung von in der Anwendung verwendeten Bilddateien übernimmt, oder einen `PreferenceStore`, in welchem Benutzerpräferenzen persistent gespeichert werden können.

19.1.2. Application.java

Diese Klasse implementiert das Interface `IPlatformRunnable` und stellt den Einstiegspunkt der Anwendung dar. Hier wird der Workbench hochgefahren und damit die grafische Benutzungsoberfläche gestartet.

19.1.3. ApplicationWorkbenchAdvisor.java

Diese Klasse übernimmt das Initialisieren des Workbench, indem die initiale Perspektive sowie der `ApplicationWorkbenchWindowAdvisor` erstellt werden.

19.1.4. ApplicationWorkbenchWindowAdvisor.java

Diese Klasse ist für die allgemeinen Layout-Einstellungen des Anwendungsfensters wie Fenstergröße und Erstellung der Menüs und Toolbars zuständig.

19.1.5. ApplicationActionBarAdvisor.java

Diese Klasse definiert den Inhalt der Menüs und Toolbars der Anwendung, indem üblicherweise die implementierten Actions instantiiert, registriert und die darauf verweisenden Buttons platziert werden.

19.2. diem.Device

Autor: Mark Eilers

19.2.1. Editor

- Der Geräte-Editor `[DeviceEditor]` (siehe. Abbildung 95)

  Zur Bearbeitung von Geräteeigenschaften wird ein Editor erstellt. Der Geräte-Editor stellt immer genau das Gerät dar, welches in der Gerätesammlung (siehe Abschnitt 19.2.2) bzw. im Profil (siehe Abschnitt 19.5) ausgewählt ist. Er besteht aus drei Teilen: Auf der linken Seite ist eine exemplarische Abbildung des ausgewählten Gerätes dargestellt, die rechte Seite enthält zwei Registerkarten. Die Registerkarte "'Eigenschaften"' erlaubt die Konfiguration der Attribute des Gerätes (Hersteller, Produktbezeichnung etc.) mittels geeigneter Widgets, die Registerkarte "'Lastgang"' ermöglicht die Darstellung und das Editieren der einzelnen Betriebsmodi des Gerätes.

- Der Gerätekompositions-Editor `[DeviceCompositionEditor]` (siehe Abbildung 95)

  Der Gerätekompositions-Editor dient der Bearbeitung von Gerätekompositionen und stellt immer die Gerätekomposition dar, die in der Gerätesammlung (siehe Abschnitt 19.2.2) bzw. im Profil (siehe Abschnitt 19.5) ausgewählt ist. Er besteht aus einem grafischen Editor, der alle einer Gerätekomposition zugeordneten Betriebsmodi grafisch darstellt und dem per Drag&Drop auch weitere Betriebsmodi hinzufügt werden können, und einer skalierbaren Zeitachse, die die Länge der Einsätze relativ zu einem unbestimmten Startzeitpunkt anzeigt. In dem grafischen Editor kann der Anwender mit der Maus Constraints zwischen den einzenlen Betriebsmodi setzen und die Einsatzzeiten der Betriebsmodi festlegen.

19.2.2. Views

- Gerätesammlung `[DeviceCollectionView]` (siehe Abbildung 94)

  Der View "'Gerätesammlung"' zeigt alle Geräte und Gerätekompositionen, die sich in der Gerätesammlung befinden, samt der dazugehörigen Betriebsmodi als Knoten einer hierarchischen Baumstruktur an. Ist einem Gerät oder einem zugeordneten Betriebsmodus ein Durchschnittsverbrauch zugeordnet, so wird aus diesem ein farbiges Effizienzsymbol gebildet und neben dem Gerät bzw. neben dem Betriebsmodus dargestellt. In der Gerätesammlung kann der Anwender Geräte, Gerätekompositionen und Betriebsmodi erstellen, löschen, kopieren und in den entsprechenden Editoren anzeigen lassen. Beim Erstellen und Kopieren von Geräten und Gerätekompositionen werden entsprechende Dateien im Dateisystem

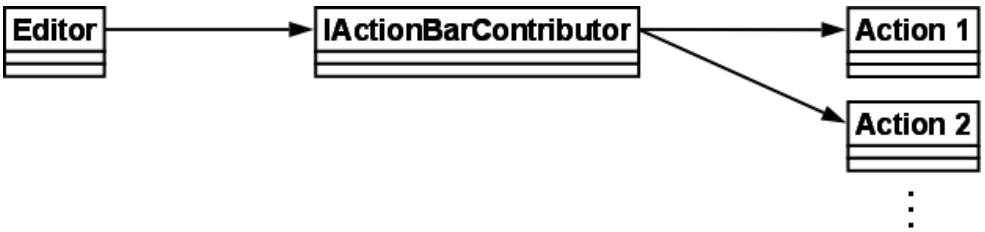

Abbildung 93: Plugin Klassenstruktur

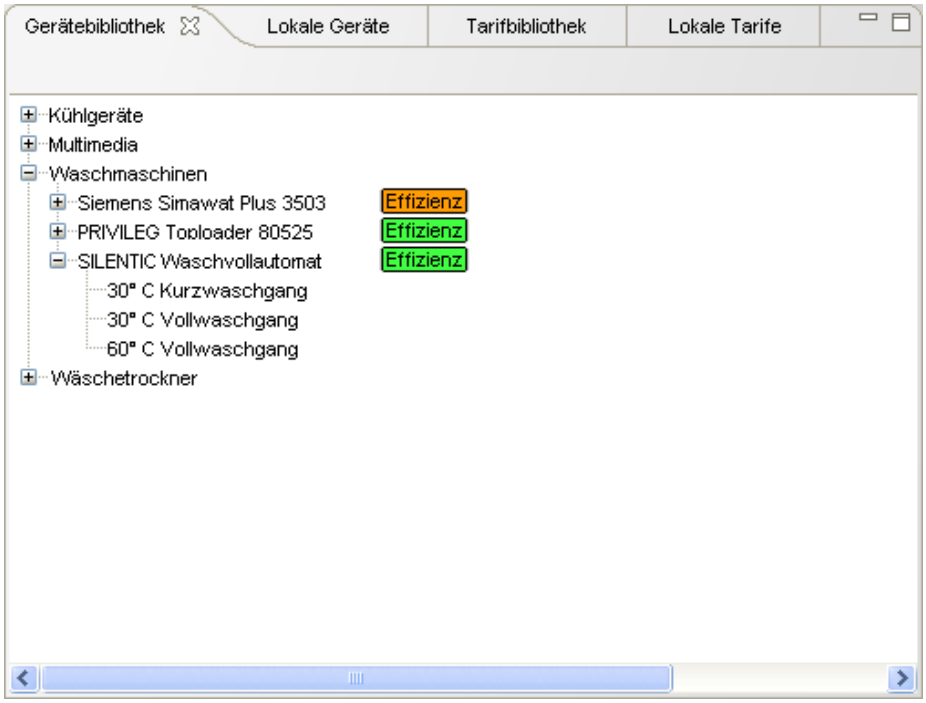

Abbildung 94: Der Gerätesammlung-View

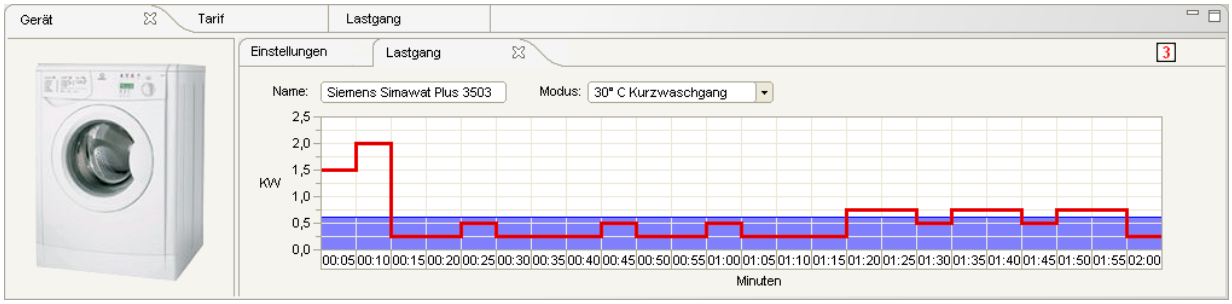

Abbildung 95: Der Geräte-Editor

sowie Knoten in der hierarchischen Baumstruktur erzeugt, beim Löschen entsprechend wieder entfernt. Beim Erstellen und Kopieren von Betriebsmodi werden dem ausgewählten Gerät im Dateisystem und in der hierarchischen Baumstruktur ein Betriebsmodus hinzugefügt, beim Löschen entsprechend wieder entfernt. Außerdem kann der Anwender Referenzen von Geräten oder Gerätekompositionen aus diesem View per Drag&Drop an den Profil-View übergeben.

- Gerätebibliothek `[DeviceLibraryView]` (siehe Abbildung 94)

  Der View "'Gerätebibliothek"' zeigt alle Geräte, die sich in der Gerätebibliothek befinden, samt der dazugehörigen Betriebsmodi als Knoten einer hierarchischen Baumstruktur an. Ist einem Gerät, oder einem zugeordneten Betriebsmodus ein Durchschnittsverbrauch zugeordnet, so wird aus diesem ein farbiges Effizienzsymbol gebildet und neben dem Gerät bzw. neben dem Betriebsmodus dargestellt. In der Gerätebibliothek kann der Anwender Geräte und Betriebsmodi in den entsprechenden Editoren anzeigen lassen oder in die Gerätesammlung kopieren. Außerdem kann kann der Anwender Referenzen von Geräten aus diesem View per Drag&Drop an den Profil-View übergeben, dabei wird das entsprechende Gerät automatisch in die Gerätesammlung kopiert.

19.2.3. Actions

- Actions des Geräte-Editors
  - **Speichern** `[SaveChanges]`
    Die Ausführung dieser Action bewirkt die Übernahme der im Geräteeigenschaften-Editor eingetragenen Werte in die entsprechende Datei im Dateisystem.
  - **Betriebsmodus auswählen** `[SelectOperationalMode]`
    Nach der Auswahl eines Betriebsmodus aus einer Drop-Down-Box werden durch diese Action die Daten und die Lastkurve des ausgewählten Betriebsmodus angezeigt.
  - **Durchschnittsverbrauch berechnen** `[CalculatePowerInput]`
    Diese Action berechnet den Durchschnittsverbrauch des ausgewählten Betriebsmodus aus den im Lastgang enthaltenen Daten.
  - **Lastgang importieren** `[ImportLoadCurve]`
    Diese Action ermöglicht es dem Nutzer einen im Dateisystem gespeicherten Lastgang zu importieren.
- Actions des Gerätekompositions-Editors
  - **Betriebsmodus in die Gerätekomposition hinzufügen** `[AddOperationalMode]`

Diese Action fügt den Betriebsmodus in die entsprechende Datei im Dateisystem, den Profil- und Gerätesammlungs-View und den Gerätekompositions-Editor ein. Der Einsatzzeitpunkt des Betriebsmodus wird dabei auf den Anfang der Gerätekomposition gelegt.

  - **Betriebsmodus aus der Gerätekomposition entfernen** `[RemoveOperationalMode]`
    Diese Action entfernt den Betriebsmodus aus der entsprechende Datei im Dateisystem, dem Profil- und Gerätesammlungs-View und dem Gerätekompositions-Editor.
  - **Einsatzzeitpunkt ändern** `[ChangeStartTime]`
    Ändert den erlaubten relativen Einsatzzeitpunkt eines verplanten Betriebsmodus.
  - **Constraints setzen** `[CreateConstraint]`
    Fügt einen Constraint zwischen zwei verplanten Betriebsmodi in die Gerätekomposition ein.
  - **Zyklen festlegen?** `[CreateCycle]`
    Definiert eine zyklische Wiederholung eines verplanten Betriebsmodus.
  - **Zeitachse ändern** `[ScaleTimeAxis]`
    Ändert die dargestellten Intervalle der Zeitachse und bewirkt die Skalierung der davon betroffenen grafischen Elemente.

- Actions des Gerätesammlung-Views
  - **Gerät erstellen** `[CreateDevice]`
    Diese Action erstellt eine Datei im Dateisytem sowie einen entsprechenden Knoten in der hierarchischen Baumstruktur des Views und öffnet den Geräte-Editor.
  - **Gerät bearbeiten** `[EditDevice]`
    Diese Action fokussiert den Geräte-Editor, welcher wiederum die betreffende Datei öffnet.
  - **Gerät kopieren** `[CopyDevice]`
    Diese Action erstellt eine Datei im Dateisystem, mit den den gleichen Inhalten, aber anderem Namen als die kopierte Datei, sowie einen entsprechenden Knoten in der hierarchischen Baumstruktur und öffnet den Geräte-Editor.
  - **Gerät entfernen** `[DeleteDevice]`
    Diese Action entfernt die entsprechende Datei aus dem Dateisystem und der hierarchischen Baumstruktur.
  - **Gerät anzeigen** `[ShowDevice]`
    Diese Action öffnet den Geräte-Editor, welcher wiederum die betreffende Datei öffnet.

  - **Betriebsmodus erstellen** `[CreateOperationalMode]`
    Diese Action erweitert das ausgewählte Gerät im Dateisystem und in der Baumstruktur um einen Betriebsmodus und öffnet den Geräte-Editor.
  - **Betriebsmodus bearbeiten** `[EditOperationalMode]`
    Diese Action fokussiert den Geräte-Editor, welcher wiederum die betreffende Datei öffnet.
  - **Betriebsmodus kopieren** `[CopyOperationalMode]`
    Diese Action erweitert das ausgewählte Gerät um einen Betriebsmodus, mit den gleichen Inhalten, aber anderem Namen als dem kopierten Betriebsmodus, sowie einen entsprechenden Knoten in der hierarchischen Baumstruktur und öffnet den Geräte-Editor.
  - **Betriebsmodus entfernen** `[DeleteOperationalMode]`
    Diese Action entfernt den Betriebsmodus aus der entsprechende Datei im Dateisystem und der hierarchischen Baumstruktur.
  - **Betriebsmodus anzeigen** `[ShowOperationalMode]`
    Diese Action öffnet den Geräte-Editor, welcher wiederum die betreffende Datei öffnet.
  - **Gerätekomposition erstellen** `[CreateDeviceComposition]`
    Diese Action erstellt eine Datei im Dateisytem, sowie einen entsprechenden Knoten in der hierarchischen Baumstruktur des Views und öffnet den Gerätekompositions-Editor.
  - **Gerätekomposition bearbeiten** `[EditDeviceCopmposition]`
    Diese Action fokussiert den Gerätekompositions-Editor, welcher wiederum die betreffende Datei öffnet.
  - **Gerätekomposition entfernen** `[DeleteDeviceComposition]`
    Diese Action entfernt die entsprechende Datei aus dem Dateisystem und der hierarchischen Baumstruktur.
  - **Betriebsmodus zu einer Gerätekomposition hinzufügen** `[AddOperationalMode]`
    Diese Action erweitert die betreffende Gerätekomposition im Dateisystem und in der Baumstruktur um einen Betriebsmodus.
  - **Betriebsmodus aus einer Gerätekomposition entfernen** `[DeleteOperationalMode]`
    Diese Action entfernt den Betriebsmodus aus der entsprechende Datei im Dateisystem und der hierarchischen Baumstruktur.

- Actions des Gerätebibliothek-Views
  - **Gerät anzeigen** `[ShowDevice]`
    Diese Action öffnet den Geräte-Editor, welcher wiederum die betreffende Da-

tei öffnet. Änderungen des Anwenders werden nicht in dieser Datei gespeichert, sondern es wird ein Kopie in der Gerätesammlung erzeugt.

- **Update der Gerätebibliothek** `[UpdateDeviceLib]`
  Diese Action ersetzt die Gerätebibliothek durch die Gerätebibliothek des Servers.
- **Gerät an den Server senden** `[SendDevice]`
  Diese Action schickt ein Gerät an den Server.

19.3. diem.PayScale

Autor: Ulrich Denecke

19.3.1. Editor

- Der Tarif-Editor `[PayScaleEditor]` (siehe Abbildung 95)

  Um Tarife zu erstellen oder zu bearbeiten, wird ein Editor erstellt. Der Tarif-Editor enthält einen grafischen Funktions-Editor (siehe Abschnitt 19.7.1), die Tarife können daher graphisch erstellt und bearbeitet werden. Alternativ dazu können die Tarife auch in eine Tabelle eingetragen werden, wobei jeweils Zeitpunkt und Strompreis angegeben werden. Weiterhin enthält der Editor noch Textfelder, um Tarifname, Tarifbeschreibung und die Gültigkeitsdauer des Tarifes einzugeben.

19.3.2. Views

- Der Tarifsammlung-View `[PayScaleCollectionView]` (siehe Abbildung 94)

  Der View "'Tarifsammlung"' zeigt alle Tarife der Tarifsammlung als Baumstruktur an. In der Tarifsammlung kann der Anwender durch Auswahl des Editors Tarife erstellen, löschen, kopieren und in dem Editor anzeigen lassen.

- Der Tarifbibliothek-View `[PayScaleLibraryView]` (siehe Abbildung 94)

  Der View "'Tarifbibliothek"' zeigt alle Tarife in der Tarifbibliothek als Baumstruktur an. Der Anwender kann Tarife im entsprechenden Editor anzeigen lassen oder in die Tarifsammlung kopieren.

19.3.3. Actions

- Actions des Tarif-Editors
  - **Tarif importieren** `[ImportPayScale]`
    Diese Action öffnet einen Dialog zur Auswahl einer csv-Datei. Wenn die Daten in der Datei den Vorgaben entsprechen, wird diese importiert und dabei eine xml-Datei angelegt, die die Tarifinformationen enthält. Weiterhin wird

ein entsprechender Knoten in der Baumstruktur des Views erstellt und der Tarif selbst als Graph angezeigt.

  - **Tarifwert hinzufügen** `[AddPayScaleValue]`
    Durch diese Action wird die Änderung dem Funktions-Editor gemeldet und ein entsprechender Wert in die Tabelle eingefügt.
  - **Tarifwert entfernen** `[RemovePayScaleValue]`
    Durch diese Action wird die Änderung dem Funktions-Editor gemeldet und der entsprechende Wert aus der Tabelle entfernt.
  - **Tarifwert verändern** `[ChangePayScaleValue]`
    Durch diese Action wird die Änderung dem Funktions-Editor gemeldet und der entsprechende Wert in der Tabelle verändert.

- Actions des Tarifsammlung-Views
  - **Tarif erstellen** `[CreatePayScale]`
    Diese Action erstellt eine Datei im Dateisystem, sowie einen entsprechenden Knoten in der Baumstruktur des Views und öffnet den Tarif-Editor.
  - **Tarif bearbeiten** `[EditPayScale]`
    Diese Action fokussiert den Tarif-Editor, welcher wiederum die betreffende Datei öffnet.
  - **Tarif kopieren** `[CopyPayScale]`
    Diese Action erstellt eine Datei im Dateisystem, mit den gleichen Inhalten, aber anderem Namen als die kopierte Datei. Weiterhin erstellt sie einen entsprechenden Knoten in der Baumstruktur und öffnet den Tarif-Editor.
  - **Tarif entfernen** `[DeletePayScale]`
    Diese Action entfernt die entsprechende Datei aus dem Dateisystem und den dazugehörigen Knoten aus der Baumstruktur.
  - **Tarif anzeigen** `[ShowPayScale]`
    Diese Action öffnet den Tarif-Editor, welcher wiederum die betreffende Datei öffnet.

- Actions des Tarifbibliothek-Views
  - **Tarif anzeigen** `[ShowPayScale]`
    Diese Action öffnet den Tarif-Editor, welcher wiederum die betreffende Datei öffnet. Änderungen des Anwenders werden nicht in dieser Datei gespeichert, sondern es wird eine Kopie in der Tarifsammlung erzeugt.
  - **Update der Tarifbibliothek** `[UpdatePayScaleLib]`
    Diese Action ersetzt die Tarifbibliothek durch die Tarifbibliothek des Servers.
  - **Tarif an den Server senden** `[sendPayScale]`
    Diese Action schickt ein Tarif an den Server.

19.4. diem.LoadCurve

Autor: Ulrich Denecke

19.4.1. Editor

- Der Lastgang-Editor `[LoadCurveEditor]` (siehe Abbildung 95)

  Um Lastgänge zu erstellen oder zu bearbeiten, wird ein Editor erstellt. Der Lastgang-Editor enthält einen grafischen Funktions-Editor (siehe Abschnitt 19.7.1), die Lastgänge können daher graphisch erstellt und bearbeitet werden. Alternativ dazu können die Lastgänge auch in eine Tabelle eingetragen werden, wobei jeweils Zeitpunkt und Messwert angegeben werden. Weiterhin enthält der Editor noch Textfelder, um Lastgangname, -beschreibung und die Startzeitpunkt des Lastgangs einzugeben.

19.4.2. Views

- Die Lastgangsammlung `[LoadCurveCollectionView]` (siehe Abbildung 94)
  Der View "'Lastgangsammlung"' zeigt alle Lastgänge, die sich in der Lastgangsammlung befinden als Knoten einer hierarchischen Baumstruktur an. In der Lastgangsammlung kann der Anwender Lastgänge erstellen, löschen, kopieren und in dem Lastgang-Editor anzeigen lassen. Beim Erstellen und Kopieren von Lastgängen werden entsprechende Dateien im Dateisystem und Knoten in der hierarchischen Baumstruktur erzeugt, beim Löschen entsprechend wieder entfernt. Außerdem kann der Anwender Referenzen von Lastgängen aus diesem View per Drag&Drop an den Profil-View übergeben.

19.4.3. Actions

- Actions des Lastgang-Editors
  - **Lastgang importieren** `[ImportLoadCurve]`
    Diese Action öffnet einen Dialog zur Auswahl einer csv-Datei. Wenn die Daten in der Datei den Vorgaben entsprechen, wird diese importiert und dabei eine xml-Datei angelegt, die die Lastganginformationen enthält. Weiterhin wird ein entsprechender Knoten in Baumstruktur des Views erstellt und der Lastgang selbst als Graph angezeigt.
  - **Lastwert hinzufügen** `[AddLoadCurveValue]`
    Durch diese Action wird die Änderung dem Funktions-Editor gemeldet und ein entsprechender Wert in die Tabelle eingefügt.
  - **Lastwert entfernen** `[RemoveLoadCurveValue]`
    Durch diese Action wird die Änderung dem Funktions-Editor gemeldet und der entsprechende Wert aus der Tabelle entfernt.

  - **Lastwert verändern** `[ChangeLoadCurveValue]`
    Durch diese Action wird die Änderung dem Funktions-Editor gemeldet und der entsprechende Wert in der Tabelle verändert.

- Actions des Lastgangsammlung-Views
  - **Lastgang erstellen** `[CreateLoadCurve]`
    Diese Action erstellt eine Datei im Dateisystem, sowie einen entsprechenden Knoten in der Baumstruktur des Views und öffnet den Lastgang-Editor.
  - **Lastgang bearbeiten** `[EditLoadCurve]`
    Diese Action fokussiert den Lastgang-Editor, welcher wiederum die betreffende Datei öffnet.
  - **Lastgang kopieren** `[CopyLoadCurve]`
    Diese Action erstellt eine Datei im Dateisystem, mit den gleichen Inhalten, aber anderem Namen als die kopierte Datei, sowie einen entsprechenden Knoten in der Baumstruktur und sie öffnet den Lastgang-Editor.
  - **Lastgang entfernen** `[DeleteLoadCurve]`
    Diese Action entfernt die entsprechende Datei aus dem Dateisystem und der Baumstruktur.
  - **Lastgang anzeigen** `[ShowLoadCurve]`
    Diese Action öffnet den Lastgang-Editor, welcher wiederum die betreffende Datei öffnet.

19.5. diem.Profile

Autor: Rachid Lacheheb

19.5.1. Views

- Der Profil-View `[ProfileView]` (siehe Abbildung 96)

  Der Profil-View zeigt alle Profile, Geräte, Gerätekompositionen, Tarife und Lastgänge als Baumstruktur an. Der Anwender kann in dem Profil durch Auswahl des Editors Profile erstellen, löschen, kopieren und in dem Editor anzeigen lassen.

19.5.2. Actions

- Actions des Profil-Views
  - **Gerät hinzufügen** `[AddDevice]`
    Diese Action erweitert das betreffende Profil im Dateisystem und in der Baumstruktur um ein Gerät.

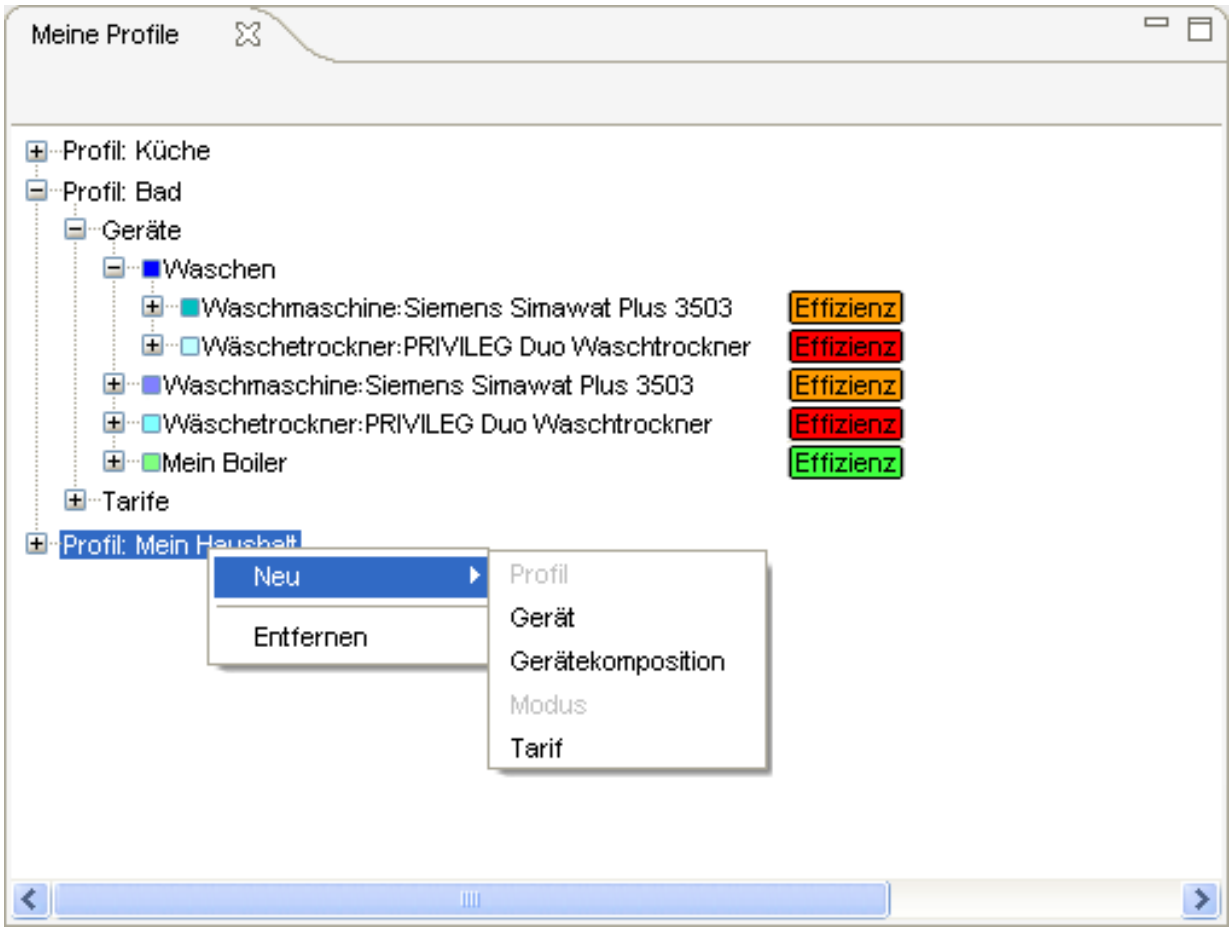

Abbildung 96: Der Profil-View

- **Gerät aus Profil entfernen** `[RemoveDevice]`
  Diese Action entfernt ein Gerät aus dem betreffende Profil im Dateisystem und in der Baumstruktur.
- **Gerätekomposition hinzufügen** `[AddDeviceComposition]`
  Diese Action erweitert das betreffende Profil im Dateisystem und in der Baumstruktur um eine Gerätekomposition.
- **Gerätekomposition aus Profil entfernen** `[RemoveDeviceComposition]`
  Diese Action entfernt eine Gerätekomposition aus dem Dateisystem und in der Baumstruktur.
- **Tarif hinzufügen** `[AddPayScale]`
  Diese Action erweitert das betreffende Profil im Dateisystem und in der Baumstruktur um einen Tarif.
- **Tarif aus Profil entfernen** `[RemovePayScale]`
  Diese Action entfernt einen Tarif aus dem Dateisystem und in der Baumstruktur.
- **Lastgang hinzufügen** `[AddLoadCurve]`
  Diese Action erweitert das betreffende Profil im Dateisystem und in der Baumstruktur um einen Lastgang.
- **Lastgang aus Profil entfernen** `[RemoveLoadCurve]`
  Diese Action entfernt einen Lastgang aus dem Dateisystem und in der Baumstruktur.
- **Profil erstellen** `[CreateProfile]`
  Diese Action erstellt eine Datei im Dateisystem, einen entsprechenden Knoten in der Baumstruktur des Views und öffnet das entsprechende Profil und dessen Einsatzplan.
- **Profil entfernen** `[DeleteProfile]`
  Diese Action entfernt die entsprechende Datei und dessen Einsatzplan aus dem Dateisystem und der Baumstruktur.
- **Profil kopieren** `[CopyProfile]`
  Diese Action erstellt eine Datei im Dateisystem, mit den gleichen Inhalten, aber anderem Namen als die kopierte Datei und einen entsprechenden Knoten in der Baumstruktur und öffnet die erstellt Kopie und dessen Einsatzplan.
- **Profil auswählen** `[ShowProfile]`
  Diese Action wählt ein Profil aus und lädt den entsprechenden Einsatzplan.

19.6. diem.Schedule

Autor: Christian Hinrichs

19.6.1. View

- Der Einsatzplan-View `[ScheduleView]` (siehe Abbildung 97)

  Dieser View dient dem Erstellen, Anzeigen und Bearbeiten von Einsatzplänen. Er stellt eine Matrix dar, welche auf der horizontalen Achse die Zeit und auf der vertikalen Achse die in diesem Plan enhaltenen Geräte und Gerätekompositionen anzeigt. Auf der "'Arbeitsfläche"' der Matrix können für die einzelnen Geräte Einsatzzeitpunkte und Einsatzdauern festgelegt werden. Den Einsätzen muss dabei ein Betriebmodus des Gerätes zugeordnet sein. Die Zeitpunkte müssen nicht explizit gesetzt werden, es können Bereiche festgelegt werden, in welchen der Geräteeinsatz von einer Planungsheuristik variabel verschoben werden darf. Zudem lassen sich zwischen den Einsätzen verschiedener Geräte mit der Maus Verbindungen ziehen, welche Constraints darstellen und damit Abhängigkeiten zwischen Geräteeinsätzen definieren. Die Art eines Constraints wird durch den genauen Verbindungspunkt an der grafischen Darstellung des Geräteeinsatzes festgelegt. So würde etwa eine Verbindung zwischen zwei Einsätzen, welche an der unteren Kante des weiter oben liegenden Einsatzes beginnt, und an der oberen Kante des weiter unten liegenden Einsatzes endet, implizieren, dass diese Einsätze parallel auszuführen sind und nicht separat verplant werden dürfen.
  Die Auswahl der Geräte und Kompositionen, deren Einsätze in diesem Einsatzplan geplant werden sollen, kann vom Benutzer per Drag & Drop aus seinem Profil erfolgen. Geräte werden dabei in der Grafik durch ihren Namen, sowie ihren Standard-Betriebsmodus dargestellt. Kompositionen werden ebenfalls durch ihren Namen identifiziert, stellen aber Mengen von Geräten und/oder weiteren Kompositionen dar und können daher "'aufgeklappt"' werden, um ihre Inhalte darzustellen. Die enthaltenen Elemente werden dann etwas eingerückt direkt unter der Komposition in den Plan eingefügt und können wie reguläre Geräte verplant werden.
  Die am unteren Rand der Arbeitsfläche gelegene Zeitachse ist frei skalierbar. Werden ihre Zeitintervalle verändert, so werden die in der Arbeitsfläche enthaltenen Elemente automatisch passend skaliert. Unterhalb der Zeitachse befindet sich ein `FunctionViewer`, in welchem verschiedene Graphen zu sehen sind (vgl. Abschnitt 19.7.1). Neben der grafischen Darstellung des gewählten Tarifes lassen sich Graphen in Abhängigkeit von den aktuell geplanten Einsätzen anzeigen. Dazu zählen zum Beispiel die über alle geplanten Geräteeinsätze kumulierte Lastkurve, oder die sich daraus ergebenden Kosten.

19.6.2. Actions

- Actions des Einsatzplans-Views
  - **Betriebsmodus in den Einsatzplan einfügen**
    `[AddOperationalModeToPlan]`

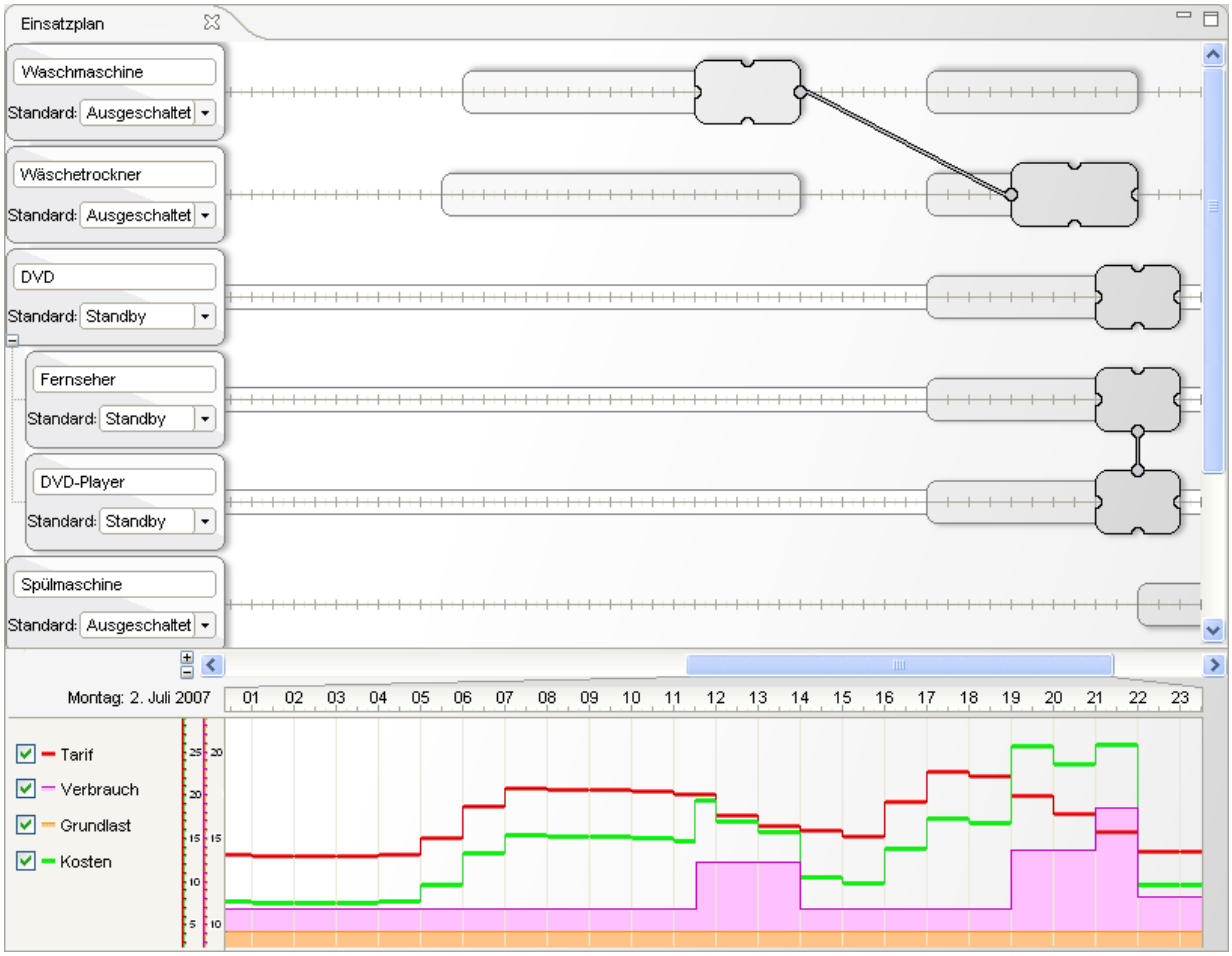

Abbildung 97: Der Einsatzplan-View

Fügt einen Betriebsmodus als Geräteeinsatz mit definiertem Zeitpunkt und Dauer in den Einsatzplan ein. Wird diese Action von dem Benutzer per Drag & Drop aufgerufen, so ergibt sich der Einsatzzeitpunkt aus der grafischen Position des *Drop* Ereignisses.

- **Betriebsmodus aus dem Einsatzplan entfernen**
  `[RemoveOperationalModeFromPlan]`
  Entfernt einen verplanten Betriebsmodus aus dem Einsatzplan.
- **Betriebsmodus im Einsatzplan kopieren**
  `[CopyOperationalModeInPlan]`
  Kopiert einen verplanten Betriebsmodus innerhalb des Einsatzplanes an eine andere Stelle.
- **Gerätekomposition in den Einsatzplan einfügen**
  `[AddDeviceCompositionToPlan]`
  Fügt eine Gerätekomposition mit definiertem Zeitpunkt und Dauer in den Einsatzplan ein. Wird diese Action von dem Benutzer per Drag & Drop aufgerufen, so ergibt sich der Einsatzzeitpunkt aus der grafischen Position des *Drop* Ereignisses.
- **Gerätekomposition aus dem Einsatzplan entfernen**
  `[RemoveDeviceCompositionFromPlan]`
  Entfernt eine verplante Gerätekomposition aus dem Einsatzplan.
- **Gerätekomposition im Einsatzplan kopieren**
  `[CopyDeviceCompositionInPlan]`
  Kopiert eine verplante Gerätekomposition innerhalb des Einsatzplanes an eine andere Stelle.
- **Einsatzzeitpunkt ändern** `[ChangeTaskDate]`
  Ändert den erlaubten Einsatzzeitpunkt eines verplanten Betriebsmodus oder einer Gerätekomposition.
- **Constraints setzen** `[CreateConstraint]`
  Fügt einen Constraint zwischen zwei verplanten Betriebsmodi oder Gerätekompositionen in den Einsatzplan ein.
- **Zyklen festlegen** `[CreateCycle]`
  Definiert eine zyklische Wiederholung eines verplanten Betriebsmodus.
- **Zeitachse ändern** `[ScaleTimeAxis]`
  Ändert die dargestellten Intervalle der Zeitachse und bewirkt die Skalierung der davon betroffenen grafischen Elemente.
- **Tarif zum Einsatzplan hinzufügen** `[AddPayScaleToPlan]`
  Fügt einen Tarif in den Informationsbereich unterhalb der Zeitachse ein.
- **Lastgang zum Einsatzplan hinzufügen** `[AddLoadCurveToPlan]`
  Fügt einen Lastgang in den Informationsbereich unterhalb der Zeitachse ein.

- **Tarif aus dem Einsatzplan entfernen** `[RemovePayScaleFromPlan]`
  Entfernt einen Tarif aus dem Informationsbereich unterhalb der Zeitachse.
- **Lastgang aus dem Einsatzplan entfernen** `[RemoveLoadCurveToPlan]`
  Entfernt einen Lastgang aus dem Informationsbereich unterhalb der Zeitachse.
- **Kurve anzeigen** `[ShowInfoChart]`
  Blendet einen Graphen im Informationsbereich unterhalb der Zeitachse ein.
- **Kurve entfernen** `[HideInfoChart]`
  Blendet einen Graphen im Informationsbereich unterhalb der Zeitachse aus.

19.7. diem.Shared

Autor: Mark Eilers

19.7.1. Grafische Editoren

- Die grafische Funktions-Anzeige `[FunctionViewer]`

  Die Funktions-Anzeige ist ein grafischer Editor des GEF zur Visualisierung von Treppenfunktionen. Je nach darzustellender Funktion (Tarif oder Lastgang) wird eine Treppenkurve in ein Koordinatensystem gezeichnet und eine entsprechende Ordinate generiert.

- Der grafische Funktions-Editor `[FunctionEditor]`

  Der Funktions-Editor ist ein grafischer Editor des GEF zum Bearbeiten und visualisieren von Treppenfunktionen. Je nach darzustellender Funktion (Tarif oder Lastgang) wird eine Treppenfunktion in ein Koordinatensystem gezeichnet und eine entsprechende Ordinate generiert. Der Funktions-Editor erlaubt es dem Anwender, durch verschiedene Werkzeuge die dargestellte Treppenfunktion zu bearbeiten.

19.7.2. View

- Der Statistik-View `[StatisticsView]` (siehe Abbildung 98)

  Der View "'Statistik"' enthält eine skalierbare Zeitachse und eine grafische Funktions-Anzeige (siehe Abschnitt 19.7.1). Per Drag&Drop können als Treppenfunktion darstellbare Objekte (z.B. Lastgänge oder Tarife) in die Funktions-Anzeige gezogen werden, die durch einzelne Checkboxes aus- und eingeblendet werden können.

19.7.3. Actions

- Actions des Statistik-Views
  - **Treppenfunktion hinzufügen** `[AddFunction]`
    Diese Action teilt der Funktions-Anzeige mit, eine Treppenfunktion zu visualisieren.
  - **Treppenfunktion entfernen** `[RemoveFunction]`
    Diese Action teilt der Funktions-Anzeige mit, die entsprechende Treppenfunktion nicht mehr zu visualisieren.
  - **Treppenfunktion aus-/einblenden** `[ToggleFunctionVisibility]`
    Diese Action teilt der Funktions-Anzeige mit, die entsprechende Treppenfunktion auszublenden, falls sie sichtbar, bzw. einzublenden, falls sie ausgeblendet ist.
  - **Zeitachse ändern** `[ScaleTimeAxis]`
    Ändert die dargestellten Intervalle der Zeitachse und meldet die Änderung an die Funktions-Anzeige.

19.8. diem.Model

Autor: Moritz Eysholdt

Das Plugin Model enthält das Diem-Datenmodell, welches im Kapitel 18 definiert wurde. Es zeichnet sich dadurch aus, dass es keine Abhängigkeiten auf andere Plugins besitzt und ausschließlich automatisch generierten Quellcode enthält. Es wird sowohl vom Client als auch vom Server verwendet.
Das Datenmodell baut auf EMF (Abschnitt 16.2) auf und besteht aus einer Menge von Java-Beans.

19.9. diem.ModelEdit

Autor: Moritz Eysholdt

Das Plugin ModelEdit wird ebenso wie das Plugin Model vollständig automatisch vom EMF generiert. Es enthält Helferklassen, welche den Entwicklern optional bei der Verknüpfung von Datenmodell und RCP-Klassen zur Verfügung stehen.
Diese Helferklassen sind `ItemProvider` für jeden Typen des Datenmodells, welche die Interfaces `IEditingDomainItemProvider`, `IStructuredItemContentProvider`, `ITreeItemContentProvider`, `IItemLabelProvider` und `IItemPropertySource` der RCP implementieren.

19.10. diem.Logic

Autor: Michael Erlmann

Die Heuristik zur automatischen Einsatzplan-Optimierung wird in `diem.Logic` enthalten sein. Dabei ist ein Interface enthalten, welches zur Kommunitation mit diem.Schedule dient. Weiter enthält es eine Klasse, welche das Auswahlkriterum für die Reihenfolge der Planung zum Erzeugen des Plans enthält, und eine Klasse, welche den Einsatzplan erstellt. Dadurch ist im Plugin intern eine hohe Modularität gegeben, so dass zum Beispiel das Auswahlkriterium leicht durch ein anderes ersetzt oder das Plugin durch weitere Funktionalitäten ergänzt werden kann.

20. Server

Autor: Christian Bley

Die Projektgruppe richtet einen, über das Internet erreichbaren Server ein, um das Update der Online-Gerätebibliothek und der Online-Tarifbibliothek, sowie das Hochladen von selbst erstellten Geräten in die Online-Gerätebibliothek zu ermöglichen, wie es in der Anforderungsdefinition im Abschnitt 4.6 definiert wurde.Die Kommunikationsschnittstelle zum Server wird in Abschnitt 20.4 beschrieben. Diese Funktionen sind ohne Zugriffsbeschränkung von der Anwendung oder einem Browser heraus aufrufbar. Des weiteren werden folgende Funktionen zur Verfügung gestellt, die nur vom Browser aus aufgerufen werden können oder zugriffsbeschränkt sind:

- Verifizieren eines Gerätes als vertrauenswürdig
- Einstellen eines Tarifes in die Online-Tarifbibliothek
- Löschen, Verschieben oder Verändern von Geräten und Tarifen in den Online-Bibliotheken.

Aufgrund der einfachen Handhabung hat sich die Projektgruppe für den Einsatz des Http-Protokolls entschieden. Die Anfragen werden über einen Webserver an Servlets weitergeleitet, die diese verarbeiten und die entsprechende Reaktion zurückgeben. Einen Überblick über die Serverstrukur gibt Abbildung 99.
Als Webserver wurde Tomcat 6.0 gewählt, da er frei verfügbar ist und sich leicht konfigurieren lässt. Der Server kann aber in jedem Servletcontainer betrieben werden, der die Anforderungen der Java^{TM} Servlet Specification Version 2.4 erfüllt.
Der Server arbeitet auf dem selben auf EMF aufbauenden Datenmodell wie der Client. Für die Datenhaltung kommt eine PostgreSQL-Datenbank zum Einsatz. Der Zugriff erfolgt durch ein Ressourcenobjekt, das über die Kombination Teneo/Hibernate mit der Datenbank verbunden ist.

20.1. Komponenten

Die Verarbeitung von Anfragen wird in den Servlets des Servers erfolgen. Der Server beinhaltet folgende Servlets und Klassen:

DeviceServlet Übernimmt die Eingabe der Gerätedateien und die Ausgabe der Onlinegeräte-Bibliothek oder einzelner Gerätedateien.

GET Gibt als Antwort das Gerät zurück, welches durch den übergebenem Pfad identifiziert wird. Wird kein Pfad als Parameter angegeben, beinhaltet die Antwort die gesamte Online-Gerätebibliothek.
Im Falle eines Fehlers beim Verarbeiten der Anfrage wird ein HTTP-Fehlercode 500 für einen internen Serverfehler zurückgegeben. Sollte das gewünschte Gerät nicht verfügbar sein, weil es nicht an dem Pfad gespeichert ist oder der Pfad nicht existiert, wird der HTTP-Fehlercode 404 zurückgesendet.

POST Nimmt eine Gerätedatei als Parameter `device` des POST-Request entgegen und speichert sie in der Online-Gerätebibliothek. Zusätzlich zur Datei muss als Parameter der Pfad des Gerätes innerhalb der Ordnerstrukur der Bibliothek angegeben werden.
Im Falle eines Fehlers beim Verarbeiten der Anfrage wird ein HTTP-Fehlercode 500 für einen internen Serverfehler zurückgegeben. Der HTTP-Code 415 wird zurückgegeben, falls die Datei nicht dem `Device`-Schema entspricht.

PayScaleServlet Übernimmt die Eingabe der Tarifdateien und die Ausgabe der Onlinetarif-Bibliothek oder einzelner Tarifdateien.

GET Gibt als Antwort den Tarif zurück, der am übergebenem Pfad gespeichert ist. Wird kein Pfad als Parameter angegeben, beinhaltet die Antwort die gesamte Onlinetarif-Bibliothek.
Im Falle eines Fehlers beim Verarbeiten der Anfrage wird ein HTTP-Fehlercode 500 für einen internen Serverfehler zurückgegeben. Sollte der Pfad keinen Tarif identifizieren, wird der HTTP-Fehlercode 404 zurückgesendet.

POST Nimmt eine Tarifdatei als Parameter `payscale` des POST-Request entgegen und speichert sie in der Online-Tarifbibliothek. Zusätzlich zur Datei muss als Parameter der Pfad des Tarifes innerhalb der Ordnerstrukur der Bibliothek angegeben werden.
Im Falle eines Fehlers beim Verarbeiten der Anfrage wird ein HTTP-Fehlercode 500 für einen internen Serverfehler zurückgegeben. Der HTTP-Code 415 wird zurückgegeben, falls die Datei nicht dem `PayScale`-Schema entspricht.
Diese Funktion wird vom Client nicht wahrgenommen, sie soll der Projektgruppe die Möglichkeit geben, auf einfachen Wege Tarife in die Onlinetarif-Bibliothek einzupflegen. Denkbar ist zu einem späteren Zeitpunkt auch eine Nutzung durch Stromanbieter, um ihre Tarife direkt in die Datenbank einzubringen.

ManagerServlet Das `ManagerServlet` arbeitet als Controller und verarbeitet die Anfragen zur Verwaltung der Online-Bibliotheken. Jede Aktion wird über POST oder GET entgegengenommen und entsprechend der Parameter `action` und `scope` verarbeitet, die die gewünschte Aktion referenzieren. Die möglichen Werte für `action` beschreibt Abschnitt 20.2, `scope` darf nur *device* oder *payscale* enthalten. Je nach Anordnung wird das Servlet die Eingaben verarbeiten und das Ergebnis über folgende JSPs visualisieren:

index.jsp Dies ist die Einstiegsseite des Servers. Es werden die Funktionen zur Auswahl gestellt. Wenn das `ManagerServlet` ohne Parameter aufgerufen wird, wird auf diese JSP umgeleitet.

devicedialog.jsp Diese JSP zeigt den Inhalt eines Ordners der Online-Gerätebibliothek an und stellt die Funktionen zum Löschen und Verschieben von Geräten zur Verfügung. Es werden Namen von Ordnern, sowie Namen und wichtige Attribute von Geräten angezeigt. Es wird insbesondere angezeigt, ob ein Gerät verifiziert wurde. Es existiert auch ein Formular um eine Gerätedatei in den aktuellen Ordner hoch zu laden.

showdevice.jsp Zeigt alle Eigenschaften eines gewählten Gerätes und ermöglicht das Verifizieren.

payscaledialog.jsp Diese JSP zeigt den Inhalt eines Ordners der Online-Tarifbibliothek an und ermöglicht das Löschen von Tarifen. Es werden Namen von Ordnern, Beschreibung und wichtige Attribute von Tarifen angezeigt. Es existiert auch eine Funktion um eine Tarifdatei in den aktuellen Ordner hoch zu laden.

showpayscale.jsp Zeigt alle Eigenschaften eines gewählten Tarifes.

error.jsp Visualisiert eventuelle Fehler und Exceptions des Servers. Sollte ein Fehler auftreten, der zu schwerwiegend ist oder nicht abgefangen werden kann, so wird hier eine Meldung mit genauer Fehlerbeschreibung erzeugt.

Die Funktionen und Auswirkungen der Kontrollparameter `action` und `scope` werden in Abschnitt 20.2 beschrieben.

ResourceHandler Für den zentralen Zugriff auf das Datenmodell sorgt die Klasse `ResourceHandler`. Bei der Initialisierung werden die Frameworks Teneo und Hibernate konfiguriert, sowie das Datenmodell registriert. Die Klasse dient dann als Factory für `Resource`-Objekte, über die Entitäten gespeichert und geladen werden können.

Util Die Klasse `Util` enthält sinnvolle Funktionen, die von allen Servlets verwendet werden, wie das Filtern von Steuerzeichen.

20.2. Management

Um die gewünschten Aktionen auszuführen nimmt das `ManagerServlet` über den Parameter `action` die Anordnungen entgegen, während der Parameter `scope` den Kontext entweder auf Tarife oder Geräte lenkt. Nachfolgend werden die einzelnen `action`-Varianten aufgelistet und deren Verhalten je nach `scope` beschrieben.

list Listet einen Bereich der Bibliotheken auf. Benötigt zusätzlich den Parameter `path` mit dem der anzuzeigende Ordner identifiziert wird.

- device: Zeigt einen Ordner der Online-Gerätebibliothek. Das Ergebnis wird mit `devicedialog.jsp` angezeigt.
- payscale: Zeigt einen gewählten Bereich der Online-Tarifbibliothek. Das Ergebnis wird mit `payscaledialog.jsp` angezeigt.

show Zeigt je nach `scope` ein Element detailiert an. Benötigt zusätzlich den Parameter `path` mit dem der anzuzeigende Ordner identifiziert wird, und den Parameter `id` mit der das gewünschte Element identifiziert wird.

- device: Das Gerät wird über `showdevice.jsp` angezeigt.
- payscale: Der gewählte Tarif wird mit `showpayscale.jsp` dargestellt.

verify Verifiziert ein Gerät. Ignoriert den Parameter `scope`. Benötigt zusätzlich den Parameter `path` mit dem der aktuelle Ordner identifiziert wird und den Parameter `id` mit der das gewünschte Gerät identifiziert wird. Als Ergebnis wird `devicedialog.jsp` mit dem aktuellen Ordner angezeigt.

delete Löscht ein Element. Benötigt zusätzlich den Parameter `path` mit dem der aktuelle Ordner identifiziert wird und den Parameter `id` mit der das gewünschte Element identifiziert wird.

- device: Löscht das gewählte Gerät aus der Onlinegeräte-Bibliothek. Als Ergebnis wird `devicedialog.jsp` mit dem aktuellen Ordner angezeigt.
- payscale: Löscht den gewählten Tarif aus der Onlinetarif-Bibliothek. Als Ergebnis wird `payscaledialog.jsp` mit dem aktuellen Ordner angezeigt.

move Verschiebt ein Elementund benötigt den Parameter `path` mit dem der aktuelle Ordner identifiziert wird, der auch der Zielordner ist. Der Parameter `movefrom` stellt den Pfad dar, an dem sich das Gerät vor dem Verschieben befindet und der Parameter `id` identifiziert das gewünschte Element.

- device: Ein Gerät wird innerhalb der Online-Gerätebibliothek verschoben. Als Ergebnis wird `devicedialog.jsp` mit dem aktuellen Ordner angezeigt.
- payscale: Ein Tarif wird innerhalb der Online-Tarifbibliothek verschoben. Als Ergebnis wird `payscaledialog.jsp` mit dem aktuellen Ordner angezeigt.

20.3. Sicherheit

Die Projektgruppe trifft entsprechende Vorkehrungen, um die Sicherheit und Stabilität des Servers zu gewährleisten.
Auf die zugriffsbeschränkten Funktionen des Servers erhalten nur authorisierte Personen Zugriff. Hierfür wird im Deploymentdeskriptor ein *authorization constraint* für den Context `/restricted/*` eingerichtet, zu dem nur Nutzer mit der Rolle *diem* Zugriff erhalten. Zusätzlich wird der POST-Zugriff auf das `PayScaleServlet` mit derselben Rolle eingeschränkt, damit nur berechtigte Stellen neue Tarife erstellen können. Die Zuordnung von Nutzernamen und Passwörtern zu den Nutzerrollen obliegt dem Server. Dieser gewährt nur Zugriff auf den geschützten Context, wenn die Anmeldedaten mit der Zugriffsdatenbank des Servers abgeglichen und authentifiziert wurden. Als Anmeldeverfahren wird die Basic-Authentification des HTTP-Protokolls verwendet.Für weitere Informationen Siehe Java^{TM} Servlet Specification Version 2.4, Kapitel 12 und 13.
Um die Typsicherheit von auf den Server zu übertragenden Dateien zu garantieren, werden diese auf die Einhaltung des `Device`- bzw. des `PayScale`-Schemas überprüft. Bei Verletzung des Schemas wird der Server die Datei ablehnen und einen entsprechenden Fehlercode als Antwort senden.
Um der Ausnutzung von Parametern zum Einfügen von schädlichem Code vorzubeugen, werden sämtliche Eingaben genau geprüft und kritische Zeichen herausgefiltert.

20.4. Client-Server-Kommunikation

Die Kommunikation zwischen Client und Server beschränkt sich auf drei Aufgaben:

Online-Gerätebibliothek vom Server beziehen Das `DeviceServlet` wird über HTTP-GET vom Client angesprochen.

Parameter Keine weiteren Parameter.

Antwort Die ganze Onlinegeräte-Bibliothek wird als `DeviceLib`-XML-Datei als Antwort mit dem HTTP-CODE 200 zurückgeliefert.

Fehlschlag Im Falle eines Serverfehlers wird ein HTTP-CODE 500 für einen internen Serverfehler zurückgegeben.

Gerätedatei zum Server schicken Das `DeviceServlet` wird über HTTP-POST vom Client angesprochen.

Parameter `path`: Muss folgendem Schema entsprechen:
`<Stammordner>/{<Unterordner>/}<Dateiname>.diem`
`device`: Enthält die Gerätedatei

Antwort Als Antwort wird der HTTP-CODE 204 für eine erfolgreiche Durchführung gesendet.

Fehlschlag Im Falle eines Serverfehlers wird ein HTTP-CODE 500 für einen internen Serverfehler zurückgegeben. Falls die gesendete Datei nicht dem `Device`-Schema entspricht, wird als antwort HTTP-CODE 415 (Unsupported Media Type) gesendet.

Online-Tarifbibliothek vom Server beziehen Das `PayScaleServlet` wird über HTTP-GET vom Client angesprochen.

Parameter Keine weiteren Parameter.

Antwort Die ganze Onlinetarif-Bibliothek wird als `PayScaleLib`-XML-Datei als Antwort mit dem HTTP-CODE 200 zurückgeliefert.

Fehlschlag Im Falle eines Serverfehlers wird ein HTTP-CODE 500 für einen internen Serverfehler zurückgegeben.

21. Programmablauf

Innerhalb des Entwurfes wurden 67 funktionale Anforderungen aufgestellt. Auf Grund der Komplexität und der Redundanz vieler Anforderungen wird auf eine Darstellung des gesamten Programmablaufes verzichtet. Stattdessen werden anhand von drei Beispielen die wichtigsten Programmabläufe beschrieben und in Form eines jeweiligen Sequenzdiagramms dargestellt. Zu diesen Abläufen werden die jeweils ähnlichen Anforderungen genannt. Die Abläufe werden jeweils auf einer unterschiedlichen Abstraktionsebene betrachtet. Folgende Abläufe werden im Einzelnen vorgestellt:

- ein neues Gerät zur Online-Bibliothek hinzufügen
- eine Gerätekomposition erstellen
- Tarif bearbeiten

21.1. Gerät zur Online-Bibliothek hinzufügen

Autor: Malte Hurrelmann

Um selbst erstellte Geräte dauerhaft zu speichern und von anderen Anwendern benutzbar zu machen ist es möglich das Gerät an einen zentralen Server zu Senden, welcher dieses dann in eine Online-Bibliothek aufnimmt. Der Vorgang der Übertragung vom Client zum Server soll im Folgenden näher beschrieben werden.

Folgende Voraussetzungen müssen erfüllt sein:

- Der Anwender hat das Programm gestartet, ein Profil ausgewählt und befindet sich im GerätesammlungsView.

Die Grafik Abbildung 100 zeigt als Sequenzdiagramm die einzelnen Abläufe innerhalb der beteiligten Plugins beim ausführen der entsprechenden Actions.

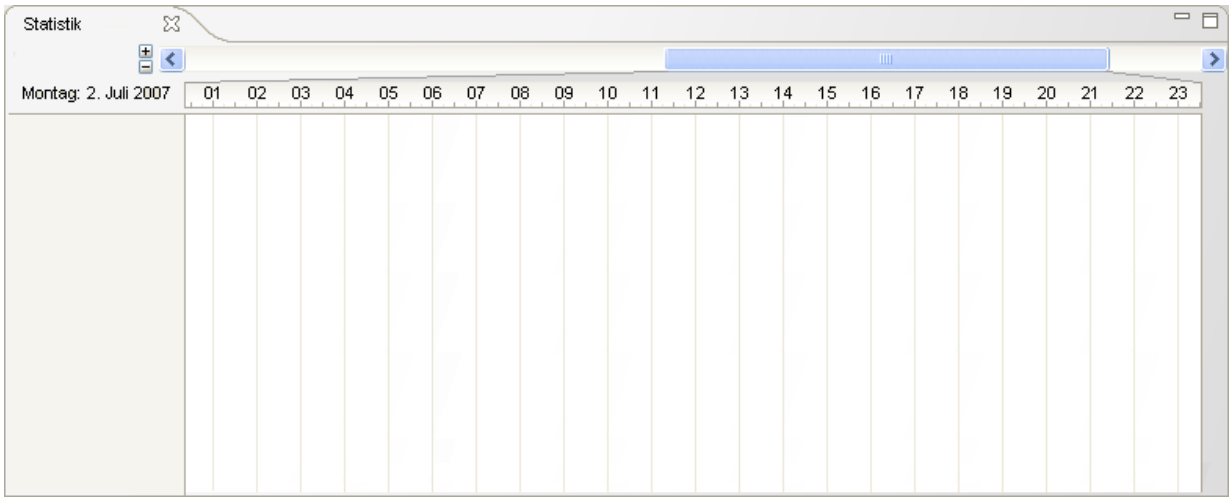

Abbildung 98: Der Statistik-View

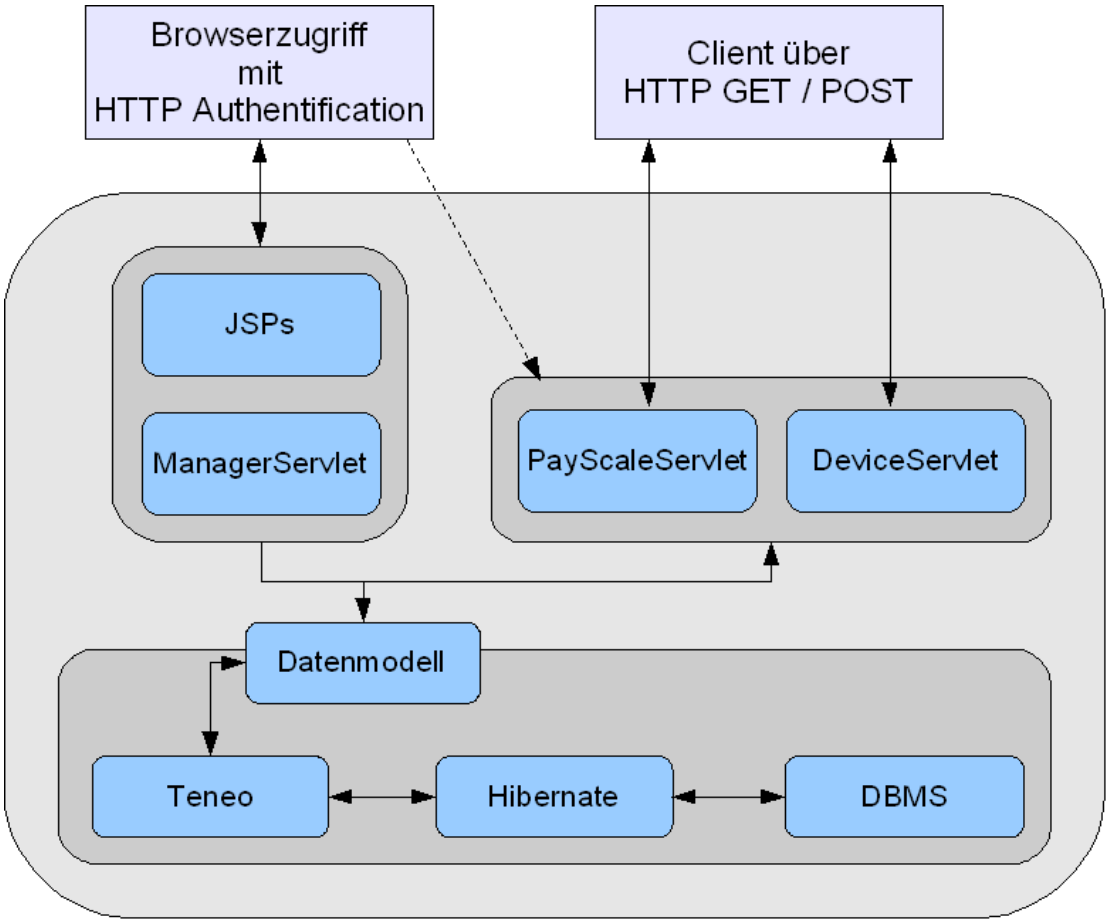

Abbildung 99: Serverstruktur

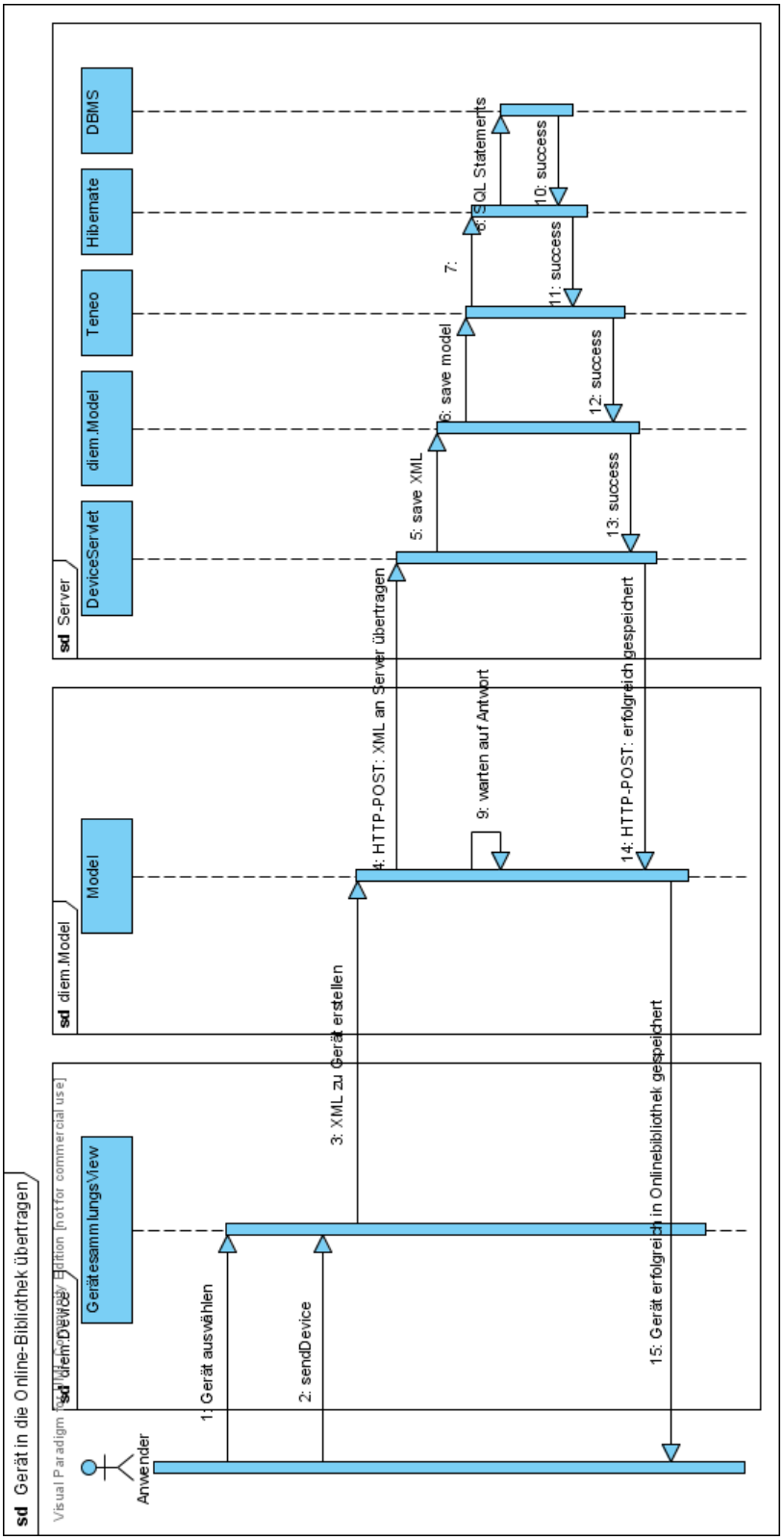

Abbildung 100: Sequenzdiagramm Gerät zur Online-Bibliothek hinzufügen

1. Der Anwender wählt das an den Server zu übermittelnde Geräte aus.
2. Der Anwender kann über die Action **Gerät an den Server senden** `[sendDevice]` das ausgewählte Gerät an den Server senden.
3. Nach start der Action `[sendDevice]` wird zu dem entsprechenden Gerät eine XML-Datei erzeugt.
4. Die erstellte XML-Datei wird per HTTP-POST an den Server übertragen. Weiterhin wird der Pfad mit übergeben, an dem das Gerät gespeichert in der Datenstruktur des Servers gespeichert werden soll.
5. In der Serveranwendung nimmt das DeviceServlet die per HTTP-POST vom Client geschickte XML Datei an und gibt es gibt es zur Speicherung an diem.Model weiter.
6. diem.Model liest die XML-Datei ein und erzeugt ein Geräteobjekt entsprechend dem Datenmodellschema.
7. Teneo liest das Geräteobjekt und übergibt es Hibernate.
8. Hibernate erzeugt aus den Geräteobjekt ein SQL-Statement und führt dieses auf einer relationalen Datenbank aus.
9. Die relationale Datenbank liefert eine Meldung über Erfolg oder Fehlschlag an Hibernate zurück.
10. Die Hibernate liefert eine Meldung über Erfolg oder Fehlschlag an Teneo zurück.
11. Die Teneo liefert eine Meldung über Erfolg oder Fehlschlag an diem.Model zurück.
12. diem.Model liefert eine Meldung über Erfolg oder Fehlschlag an das DeviceServlet zurück.
13. Das DeviceServlet liefert eine Meldung über Erfolg oder Fehlschlag an den noch wartenden Client zurück. Dieses geschieht über HTTP Fehlercodes.
14. diem.Model des Clients informiert den Anwender über Erfolg oder Fehlschlag des Speichervorgangs.

21.2. Gerätekomposition erstellen

Autor: Michael Belasus

Innerhalb des Ablaufes **Gerätekomposition erstellen** wird das Anlegen einer neuen Gerätekomposition bestehend aus drei Geräten beschrieben. Dabei werden die drei Fälle betrachtet, von wo der Anwender diese Geräte beziehen kann. Zum einen ist das Gerät in der Gerätesammlung und zum anderen ist es in der lokalen Gerätebibliothek vorhanden. Im dritten Fall muss das Gerät neu durch den Anwender angelegt werden.

Folgende Voraussetzungen müssen erfüllt sein:

- Der Anwender hat das Programm gestartet, ein Profil ausgewählt und befindet sich im GerätekomopsitionsView.
- Eins der Geräte befindet sich in der lokalen Gerätesammlung.
- Eins der Geräte befindet sich in der lokalen Gerätebibliothek.

Die Abbildung 101 zeigt als Sequenzdiagramm die einzelnen Abläufe innerhalb der beteiligten Plugins beim Ausführen der entsprechenden Actions.

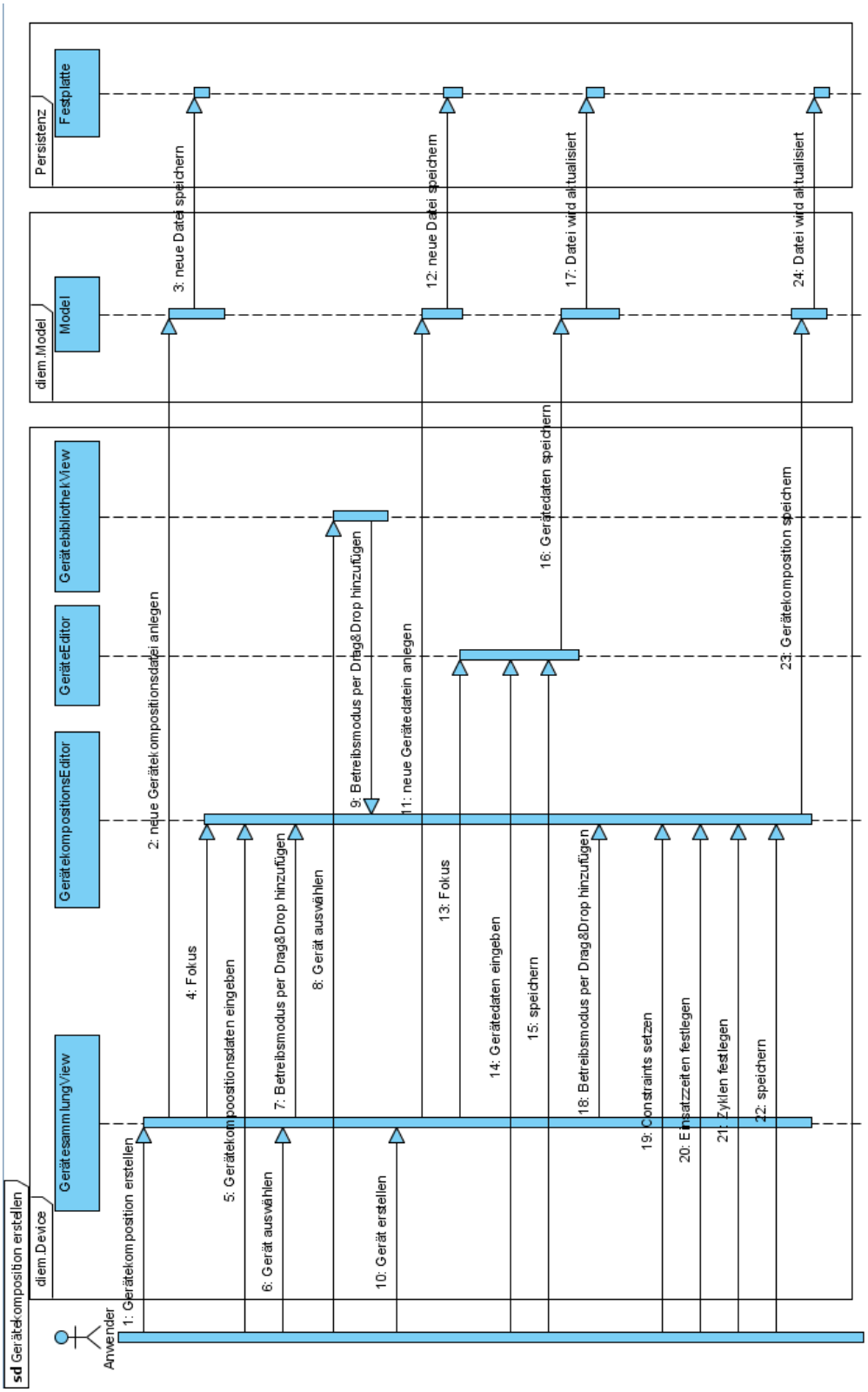

Abbildung 101: Sequenzdiagramm neue Gerätekomposition erstellen

1. Der Anwender kann über die Action **Gerätekomposition erstellen** `[CreateDeviceComposition]` eine neue Gerätekomposition innerhalb des GerätesammlungsView erstellen.
2. Die ausgeführte Action bewirkt, dass das Model eine neue Datei für die Gerätekomosition anlegt.
3. Diese neue Gerätekompositionssdatei wird auf der Festplatte gespeichert.
4. Der Fokus wechselt nach der erfolgten Action auf den GerätekompositionsEditor.
5. Der Anwender kann jetzt die Daten zu dieser Gerätekomposition wie Name und Bemerkungen eingeben.
6. Um ein vorhandenes Gerät aus der Gerätesammlung zur Komposition hinzufügen zu können, wechselt der Anwender in den GerätesammlungsView und wählt das gewünschte Gerät aus.
7. Den entsprechenden Betriebsmodus des Gerätes kann der Anwender nun per Drag&Drop in den GerätekompositionsEditor ziehen.
8. Um ein vorhandenes Gerät aus der Gerätebibliothek zur Komposition hinzufügen zu können, wechselt der Anwender in den GerätebibliotheksView und wählt das gewünschte Gerät aus.
9. Den entsprechenden Betriebsmodus des Gerätes kann der Anwender nun per Drag&Drop in den GerätekompositionsEditor ziehen.
10. Um ein neues Gerät zur Komposition hinzufügen zu können, wechselt der Anwender in den GerätesammlungsView und durch die Action **Gerät erstellen** `[CreateDevice]` wird ein neues Gerät angelegt.
11. Die ausgeführte Action bewirkt, dass das Model eine neue Datei für das Gerät anlegt.
12. Diese neue Gerätedatei wird auf der Festplatte gespeichert.
13. Der Fokus wechselt auf den GeräteEditor.
14. Der Anwender kann jetzt die Daten zu diesem neuen Gerät eingeben:
    - Name,
    - Bemerkungen,
    - Betriebsmodi anlegen,
    - den Verbrauch eintragen,
    - wenn vorhanden einen Lastgang importieren

15. Der Anwender kann mit der Action **Speichern** `[SaveChanges]` die eingegebenen Gerätedaten an das Model weitergegeben.
16. Die Gerätedaten werden aktualisiert.
17. Diese bereits vorhandene Gerätedatei wird auf der Festplatte gespeichert.
18. Der Anwender kann nun den gewünschten Betriebsmodus des Gerätes per Drag&Drop in den GerätekompositionsEditor ziehen.
19. im GerätekomositionsEditor können nun Constraints festgelegt werden
20. im GerätekomositionsEditor können nun Einsatzzeiten festgelegt werden
21. im GerätekomositionsEditor können nun Zyklen festgelegt werden
22. Mit der Action **Speichern** `[SaveChanges]` wird die erstellte Gerätekomposition an das Model weitergegeben
23. Die Gerätekompositionsdaten werden aktualisiert.
24. Diese bereits vorhandene Gerätekompositionsdatei wird auf der Festplatte gespeichert.

Dieser Ablauf beinhaltet bereits das Anlegen eines Gerätes als eine funktionale Anforderung aus der Anforderungsanalyse. Weiterhin ist der Ablauf vergleichbar mit dem Anlegen eines Profils und von Einsätzen. Die Punkte 6, 8 und 10 aus der Abbildung 101 können beliebig oft wiederholt werden, ebenfalls das Festlegen der Constraints, Zyklen und Einsatzzeiten.

21.3. Tarif bearbeiten

Autoren: Mark Eilers und Christian Hinrichs

Der folgende Programmablauf beschreibt eine Ausführung der Action "'Tarifwert verändern"' (`ChangePayScaleValue`). Es wird vorausgesetzt, dass der Tarif-Editor bereits geöffnet ist und den Fokus besitzt. Die einzelnen Schritte entsprechen den nummerierten Pfeilen in Abbildung 102.

1. Der Anwender klickt auf einen Preispunkt im Editor und beginnt, diesen bei gehaltener Maustaste zu verschieben. Für jeden Pixel, um den die Maus nun verschoben wird, wird über die Benutzungsoberfläche (konkret: SWT im Zusammenspiel mit Betriebssystem-Listenern) ein `DragEvent` im RCP ausgelöst.
2. Dieses `DragEvent` wird direkt zum `PayScaleEditor` geleitet, wo es an den enthaltenen `FunctionEditor` weitergegeben wird.

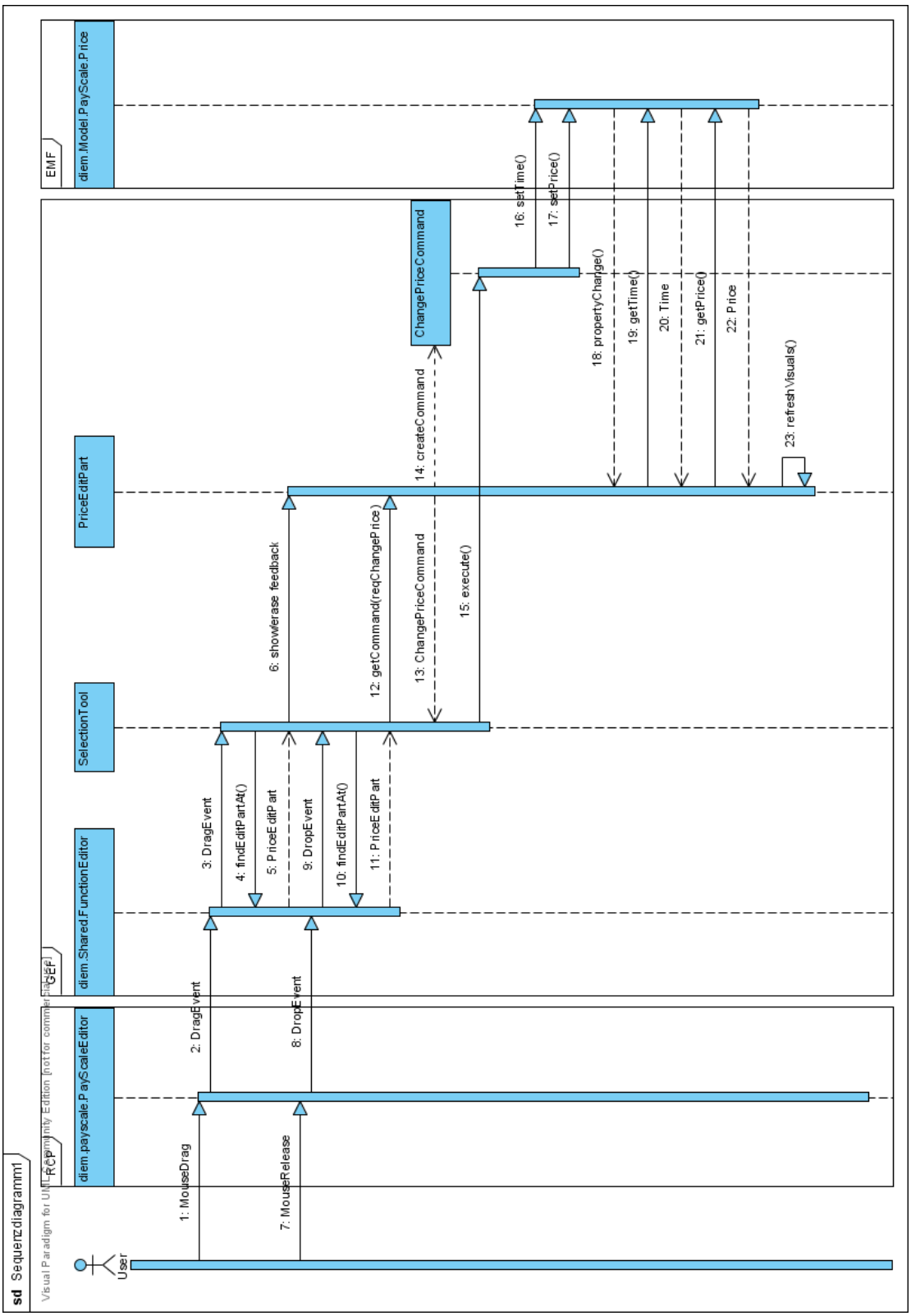

Abbildung 102: Sequenzdiagramm zu der Action "'Tarif bearbeiten"'

3. Der Funktions-Editor leitet das `DragEvent` nun an das aktive Werkzeug weiter. Im Funktions-Editor ist implizit das Selektions-Werkzeug ausgewählt, ein Werkzeug zum Selektieren und Verschieben von Draw2d `Figure` Objekten.
4. Das Selektions-Werkzeug ruft die Methode `findEditPartAt()` des Funktions-Editors auf und übergibt die Position des `DragEvent`, um an das dazugehörige `EditPart` zu gelangen.
5. Der Funktions-Editor gibt das EditPart an der entsprechenden Position zurück: im Falle eines zu bearbeitenden Tarifs ist es ein `PriceEditPart`.
6. Das Selektions-Werkzeug veranlasst nun, dass das `PriceEditPart` das `DragEvent` visualisiert, indem der selektierte Preispunkt analog zu den Koordinaten des `DragEvent` verschoben wird.
7. Die obigen Schritte werden so oft wiederholt, bis der Anwender die Maustaste loslässt und damit über die Benutzungsoberfläche ein `DropEvent` im RCP ausgelöst.
8. Das `DropEvent` landet wiederum im `PayScaleEditor`, wo es ebenso wie das vorherige `DragEvent` an den enthaltenen `FunctionEditor` weitergegeben wird.
9. Der Funktions-Editor leitet das `DropEvent` wieder an das Selektions-Werkzeug weiter.
10. Das Selektions-Werkzeug fordert das betreffende `EditPart` an und der Funktions-Editor gibt das `PriceEditPart` zurück.
11. Siehe Punkt 5.
12. Aus dem `DropEvent` erstellt das Selektions-Werkzeug ein `ChangePrice` Request und übergibt dieses über die Methode `getCommand()` an das `PriceEditPart`.
13. Das `PriceEditPart` erstellt aus dem Request ein `ChangePriceCommand` (14.) und gibt es dem aktiven Werkzeug zurück.
14. Siehe Punkt 13.
15. Dieses ruft nun die Methode `execute()` des erhaltenen `ChangePriceCommands` auf.
16. Die Methode `execute()` verändert nun durch die Methode `setTime()` das betreffende `Price` Objekt im Datenmodell.
17. Die Methode `execute()` verändert durch die Methode `setPrice()` das betreffende `Price` Objekt im Datenmodell.

18. Das `Price` Objekt meldet nun dem `PriceEditPart`, dass sich Werte verändert haben.
19. Das `PriceEditPart` liest den veränderten Wert über die Methode `getTime()` des `Price` Objektes.
20. Liefert den `Time`-Wert des des `Price` Objektes.
21. Das `PriceEditPart` liest den veränderten Wert über die Methode `getPrice()` des `Price` Objektes.
22. Liefert den `Price`-Wert des des `Price` Objektes.
23. Die aktuellen Daten werden nun über die Methode `refreshVisuals()` an die Draw2d `Figure` Objekte des `PriceEditParts` übergeben, welche daraufhin neu gezeichnet werden.

Eine Stärke vom GEF liegt in der Unabhängigkeit des Editors in Bezug auf das darzustellende Datenmodell. Sollte beispielsweise statt eines Tarifs ein Lastgang bearbeitet werden, so würde der Funktions-Editor das ausgewählte `PayScaleEditPart` an das Selektions-Werkzeug übergeben, das dann analog über entsprechende `ChangePayScaleCommands` den Lastgang verändern würde.

22. Qualitätssicherung

Autor: Michael Erlmann

Die Sicherstellung der Qualität ist eine wichtige Vorraussetzung für das Gelingen großer Softwareprojekte. Daher werden auch bei der Umsetzung dieser Software Methoden der Qualitätssicherung durchgeführt. Es werden folgende Vorgehensmethoden während der Entwicklung angewandt:

- Maximale konstruktive Qualitätssicherung
- Frühe Fehlererkennung und -behebung
- Unabhängige Qualitätssicherung
- Entwicklungsbegleitende Qualitätssicherung

Absichtlich verzichtet wird auf das Anwenden der Vorgehensmethode der quantitativen Qualitätssicherung, da der Mehraufwand zum Ermitteln und Überprüfen von quantifizierbaren Soll- und Ist-Werten bei der vorliegenden Software nicht zu lohnen scheint. Die verwendeten Vorgehensmethoden finden Anwendung bei der Entwicklung der einzelnen Plugins. Zur Betrachtung der Entwicklung des Gesamtsystems soll das so genannte V-Modell zum Einsatz kommen (Vergleiche Abbildung 103).

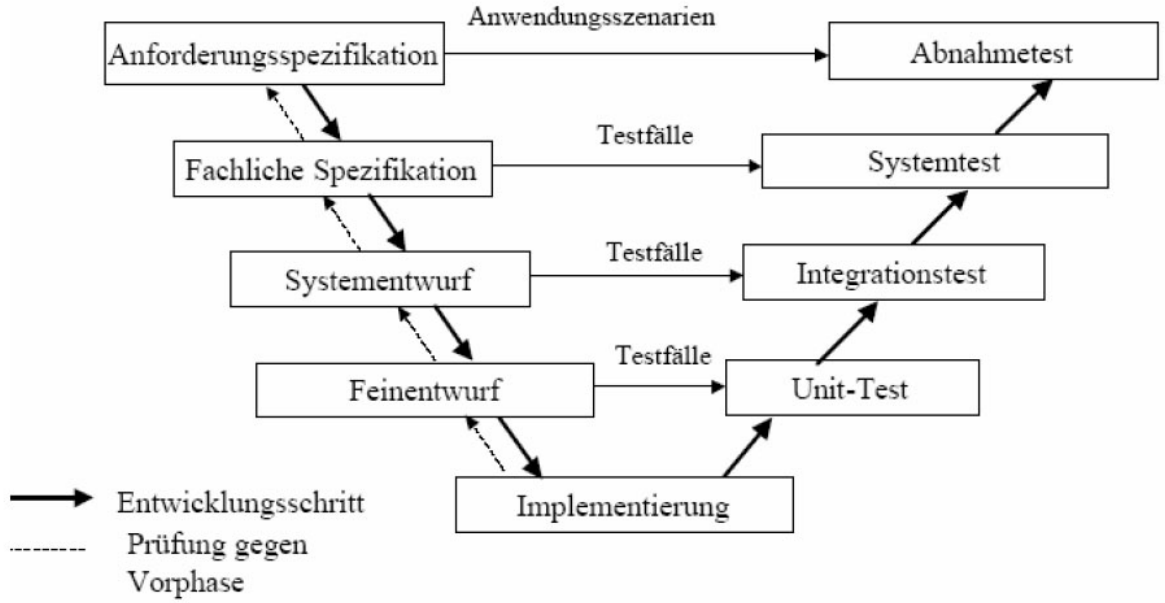

Abbildung 103: Das V-Modell. Quelle: Martin Wirsing: Methoden des Software-Engineering, Skript zur Vorlesung, LMU München, 2003

Für weitere Informationen über das V-Modell sowie über die Vorgehensmethoden sei hier auf die Seminararbeit Qualitätssicherung von Michael Erlmann aus der Projektgruppe Demonstrator InHouse Energie-Management verwiesen. Da es sich bei dem vorliegenden Dokument um den Systementwurf handelt, ist der zugehörige Teil der Qualitätssicherung damit der Integrationstest der Komponenten. Dabei werden in der vorliegenden Softwarearchitektur die Plugins als Komponenten angenommen.

22.1. Client

Autor: Michael Erlmann

Der durchzuführende Integrationstest betrachtet die einzelnen Plugins als Blackboxen. Daher müssen bereits in diesem Dokument die Interaktionen zwischen den einzelnen Plugins festgehalten sein, um entsprechende Treiber und Testfälle bestimmen zu können. Da der Integrationstest entsprechend der Implementationsreihenfolge als Bottom-Up durchgeführt werden soll, kann auf das Erstellen von Stümpfen verzichtet werden. Dabei soll der Integrationstest parallel zur Entwicklung immer dann durchgeführt werden, wenn ein weiteres Plugin fertig gestellt wurde. Für eine Übersicht über die Zusammenhänge zwischen den einzelnen Plugins sei hier nochmal auf Abbildung 82 verwiesen.
Die folgenden Abschnitte beschreiben kurz die einzelnen Plugins sowie ihre Position

in der Architektur. Dabei liegt das Augenmerk auf den Interaktionen zwischen den Plugins, um hieraus die nötigen Verbindungen zwischen ihnen für den Integrationstest zu erknennen und zum Ermitteln von Testdaten. Eine genaue Beschreibung der internen Funktionalitäten der einzelnen Plugins findet sich in Kapitel 19. Dabei enthalten die nachfolgenden Abschnitte absichtlich redundante Informationen, um jeden der Abschnitte unabhängig vom Dokument zum Testen des zugehörigen Plugins heran ziehen zu können, ohne andere Teile des Entwurfs durchsuchen zu müssen.

22.1.1. diem.App

Autor: Michael Erlmann

Das Plugin diem.App ist das zentrale Plugin, welches den Lebenszyklus der Anwendung kontrolliert. Es enthält die notwendigen Klassen für die Darstellung der grafischen Benutzeroberfläche und organisiert das Layout. Daher sind alle tiefer liegenden Plugins mit einer Oberfläche von diem.App abhängig. Initialisiert wird diem.App durch die Equinox-Umgebung.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit
- diem.Device
- diem.LoadCurve
- diem.PayScale
- diem.Logic
- diem.Profile
- diem.Schedule

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge (Aufrufe durch andere Plugins) :

- `diem.app.Init` // von Equinoxumgebung

Ausgänge (Aufrufe in ein anderes Plugin) :

- `diem.device.Init`
- `diem.payScale.Init`
- `diem.loadCurve.Init`
- `diem.schedule.Init`

- `diem.profile.Init`

Dieses Plugin kann in der Architektur als das am weitesten oben liegende betrachtet werden. Da es die Oberfläche bereit stellt, muss beim Integrationstest überprüft werden, ob die Funktionalitäten, welche die tiefer liegenden Plugins bieten, auch für den Anwender zugreifbar sind und ob sie an all den Stellen erscheinen, an welchen sie auftreten sollen, und nur dort.

22.1.2. diem.Device

Autor: Michael Erlmann

Das Plugin diem.Device enthält den Geräte-Editor und den Gerätekompositions-Editor sowie die Gerätesammlung und die Gerätebibliothek. Dabei bestehen Interaktionen mit diem.Profile und diem.Schedule, da diese Geräte aus der Gerätesammlung referenzieren oder Geräte aus der Bibliothek zu diesen hinzugefügt werden können. Ebenso hat es eine Verbindung zu diem.LoadCurve, da vorhandene Betriebsmodi enthaltene Lastgänge referenzieren können. Durch die Datenhaltung und Persistenz im Model ist Interaktion mit diem.Model und diem.ModelEdit vorhanden. Diese tiefer liegende Struktur muss beim Integrieren dieses Plugins berücksichtigt werden.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit
- diem.LoadCurve

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.device.ShowDevice` // von diem.Profile, diem.Schedule
- `diem.device.ShowOperationalMode` // von diem.Profile, diem.Schedule
- `diem.device.ShowDeviceComposition` // von diem.Profile, diem.Schedule

Ausgänge :

- `diem.loadCurve.ShowLoadCurve`
- `diem.profile.AddDevice`
- `diem.profile.AddDeviceComposition`
- `diem.schedule.AddOperationalModeToPlan`

Im Besonderen ergibt sich bei diem.Device der folgende Testfall: Falls ein Gerät aus der Gerätebibliothek in ein Profil oder einen Einsatzplan eingefügt wird, so muss überprüft werden, ob sich das Gerät in der Gerätesammlung befindet. Falls dies nicht der Fall ist, muss es hier noch hinzu gefügt werden.
Der Treiber für das Plugin muss somit den Aufruf des Anzeigens eines Gerätes, einer Gerätekomposition oder eines Betriebsmodus von extern abdecken. Weiter muss der Treiber Testfälle enthalten, bei denen Geräte aus der Gerätesammlung sowie aus der Gerätebibliothek heraus genommen werden. Dazu kommt ein Testfall für den Aufruf, den Lastgang eines Betriebsmodus anzeigen zu wollen.

22.1.3. diem.PayScale

Autor: Ulrich Denecke

Das Plugin diem.PayScale enthält den Tarif-View und den Tarif-Editor. Der Tarif-Editor nutzt den Funktions-Editor aus diem.Shared, wodurch eine Verbindung zu diesem Plugin besteht. Durch die Datenhaltung und Persistenz im Model ist Interaktion mit diem.Model und diem.ModelEdit vorhanden.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit
- diem.Shared

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.payScale.ShowPayScale` // von diem.Profile, diem.Schedule

Ausgänge :

- `diem.shared.FunctionEditor`
- `diem.shared.FunctionViewer`
- `diem.profile.AddPayScale`
- `diem.schedule.AddPayScaleToPlan`

Besonders bei der Integration zu beachten ist die Nutzung des grafischen Funktions-Editors aus diem.Shared. Weiterhin ist zu beachten, dass beim direkten Einfügen aus der Bibliothek die Konsistenz zwischen Bibliothek und Sammlung gewahrt bleibt.

22.1.4. diem.LoadCurve

Autor: Ulrich Denecke

Das Plugin diem.LoadCurve enthält den Lastgangs-View und den Lastgang-Editor. Der Lastgang-Editor nutzt den Funktions-Editor aus diem.Shared, wodurch eine Verbindung zu diesem Plugin besteht. Durch die Datenhaltung und Persistenz im Model ist Interaktion mit diem.Model und diem.ModelEdit vorhanden.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit
- diem.Shared

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.loadCurve.ShowLoadCurve` // von diem.Device, diem.Profile, diem.Schedule

Ausgänge :

- `diem.shared.FunctionEditor`
- `diem.shared.FunctionViewer`
- `diem.profile.AddLoadCurve`

Besonders bei der Integration zu beachten ist die Nutzung des grafischen Funktions-Editors aus diem.Shared.

22.1.5. diem.Shared

Autor: Michael Erlmann

In diem.Shared liegen die Funktions-Anzeige und der Funktions-Editor sowie der Statistik-View. Diese werden verwendet, um Tarife und Lastgänge an zu zeigen und zu bearbeiten. Daher besteht eine Verbindung zu diem.PayScale und diem.LoadCurve. Angezeigt und bearbeitet werden die Daten direkt auf dem Model wodurch eine Verbindung zu diem.Model und diem.ModelEdit besteht.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.shared.FunctionEditor` // von diem.PayScale, diem.LoadCurve
- `diem.shared.FunctionViewer` // von diem.PayScale, diem.LoadCurve

Ausgänge :

- Keine

Der Treiber für das Plugin muss sowohl Testfälle für das Anzeigen wie auch für das Bearbeiten von Tarifen und von Lastgängen bieten. Dabei muss geprüft werden, ob die angezeigten und veränderten Daten jederzeit mit denen im Model konform sind.

22.1.6. diem.Profile

Autor: Ulrich Denecke

Das Plugin diem.Profile enthält den Profil-View, über welchen ein Profil erstellt und bearbeitet werden kann. Der View zeigt Geräte und Gerätekompositionen sowie Tarife und Lastgänge. Daher bestehen Verbindungen zu diem.Device, diem.PayScale und diem.LoadCurve. Weiterhin sind Einsatzpläne in Profilen referenziert. Daher existiert eine Verbindung zu diem.Profile. Durch die Datenhaltung und Persistenz im Model ist Interaktion mit diem.Model und diem.ModelEdit vorhanden.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit
- diem.Device
- diem.PayScale
- diem.Logic

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.profile.AddDevice` // von diem.Device, diem.Schedule
- `diem.profile.AddDeviceComposition` // von diem.Device, diem.Schedule
- `diem.profile.AddLoadCurve` // von diem.LoadCurve, diem.Schedule
- `diem.profile.AddPayScale` // von diem.PayScale, diem.Schedule

Ausgänge :

- `diem.device.ShowDevice`
- `diem.device.ShowOperationalMode`
- `diem.device.ShowDeviceComposition`
- `diem.payScale.ShowPayScale`
- `diem.loadCurve.ShowLoadCurve`
- `diem.schedule.AddOperationalModeToPlan`
- `diem.schedule.AddPayScaleToPlan`

Bei der Integration ist bei diem.Profile Folgendes zu beachten: Werden Geräte oder Tarife direkt aus den entsprechenden Bibliotheken in das Profil eingefügt, so ist eine Überprüfung auf Konsistenz zwischen Bibliothek und Sammlung notwendig.
Der Treiber für dieses Plugin muss die Testfälle für das Einfügen von Geräten, Gerätekompositionen, Lastgängen und Tarifen in ein Profil sowie das Anzeigen der genannten Komponenten bieten.

22.1.7. diem.Schedule

Autor: Michael Erlmann

Das Plugin diem.Schedule enthält den Einsatzplan-View, über welchen ein Einsatzplan erstellt und bearbeitet werden kann. Da Geräte und Gerätekompositionen sowie Tarife direkt in einen Einsatzplan eingefügt werden können, bestehen Verbindungen zu diem.Device und diem.PayScale. Zusätzlich können diese Elemente aus einem Profil heraus eingefügt werden. Weiter sind Einsatzpläne in Profilen referenziert. Daher existiert eine Verbindung zu diem.Profile. Da es Möglich ist einen Einsatzplan automatisch generieren zu lassen besteht eine Verbindung zu diem.Logic. Durch die Datenhaltung und Persistenz im Model ist Interaktion mit diem.Model und diem.ModelEdit vorhanden.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model
- diem.ModelEdit
- diem.Device
- diem.PayScale
- diem.Logic
- diem.Profile

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.schedule.AddOperationalModeToPlan` // von diem.Profile, diem.Device
- `diem.schedule.AddPayScaleToPlan` // von diem.Profile, diem.PayScale

Ausgänge :

- `diem.device.ShowDevice`
- `diem.device.ShowOperationalMode`
- `diem.device.ShowDeviceComposition`
- `diem.payScale.ShowPayScale`
- `diem.loadCurve.ShowLoadCurve`
- `diem.profile.AddDevice`
- `diem.profile.AddDeviceComposition`
- `diem.profile.AddPayScale`
- `diem.logic.Optimize`

Bei diem.Schedule ergeben sich bei der Integration folgende besondere Fälle: Falls ein Gerät oder ein Tarif eingefügt wird, muss die Existenz einer Instanz in der jeweiligen Sammlung geprüft werden, da das Einfügen in den Einsatzplan aus der Bibliothek direkt möglich ist. Ebenso müssen Referenzen auf die einzelnen Instanzen in der Sammlung nach Abschluss der Einfügeaktion im Profil enthalten sein.
Somit muss beim Test des Plugins betrachtet werden, ob Geräte, Gerätekompositionen und Tarife in den Einsatzplan eingefügt werden können, und ob dabei die entsprechenden Instanzen in der Sammlung und die Referenzen auf diese im Profil erzeugt werden, falls sie noch nicht existieren. Ebenso werden Testfälle benötigt, um das automatische Generieren des Einsatzplanes zu überprüfen.

22.1.8. diem.Logic

Autor: Michael Erlmann

Im Plugin diem.Logic befindet sich die Heuristik zum automatischen Generieren eines Einsatzplanes. Daher ist eine Interaktion mit diem.Schedule notwendig. Da die Daten im Model gehalten werden, nutzt diem.Logic diese direkt für seine Berechnungen beim Erzeugen des Einsatzplanes.
Folgende Plugins müssen für den Bottom-Up-Test bereits implementiert sein:

- diem.Model

Im Folgenden sind die Interaktionen aufgelistet:

Eingänge :

- `diem.logic.Optimize` // von diem.Schedule

Ausgänge :

- Keine

Der Treiber für dieses Plugins übergibt zum Test Einsatzpläne und benötigt entsprechende Testfälle.

22.1.9. diem.ModelEdit

Autor: Michael Belasus

Das Plugin diem.ModelEdit ist vollständig aus dem Datenmodell generiert und enthält Adapter-Klassen, welche grafische Elemente mit dem Datenmodell verbinden können (Vergleich Kapitel 18 und Abschnitt 19.9). Für einen Bottom-Up-Test muss das Plugin diem.Model implementiert sein. Durch die automatische Generierung wird auf eine vollständige Prüfung der Klassen verzichtet. Der Test dieses Plugins bezieht sich nur auf die Funktionsfähigkeit der grafischen Anzeige.

22.1.10. diem.Model

Autor: Michael Belasus

Das Plugin diem.Model enthält das Datenmodell, auf Grund dessen die Daten persistiert und geladen werden können. Es müssen keine weiteren Plugins vorhanden sein (Vergleich Abbildung 82). Das Plugin wird, wie bereits in Kapitel 18 und Abschnitt 19.8 beschrieben, durch EMF Abschnitt 16.2 generiert, daher wird das Plugin daraufhin getestet, ob die Daten vollständig gespeichert bzw. geladen werden.

22.2. Server

Autor: Christian Bley

Analog zum Client wird beim Server ein Integrationstest durchgeführt, bei dem alle Servlets und Klassen als Blackboxen angesehen werden. Da sämtliche Servlets auf einer horizontalen Ebene liegen, kann hier eigentlich nicht von einem Bottom-Up-Test gesprochen werden. Dieser Ansatz ergibt sich erst, wenn die Datenhaltung mit dem Datenmodell und der Persistenzebene zunächst parallel zu den Servlets getestet werden. Daher benötigt man als Stumpf für die Servlets die Verbindung zur Datenhaltung. Die möglichen Anfragen an die Servlets werden über Treiberklassen getestet.

Für die Datenhaltung werden Treiber benötigt, die Testdaten an die Persistenzschicht senden. Die Persistenzebene selbst kann nur über Blackbox-Tests getestet werden, da die Projektgruppe die Frameworks Teneo und Hibernate im Verbund nutzt und hier keine kleinere Einheit zum Testen existiert. Die Daten werden dann in der Datenbank verifiziert. Weitere Treiber laden die Daten und überprüfen sie auf Konsistenz.
Wenn diese Tests abgeschlossen sind kann der Server als Ganzes getestet werden, indem die gleichen Treiber für die Servlets verwendet werden.
Das Datenmodell des Servers ist identisch mit dem des Clients und wird daher so getestet, wie in Abschnitt 22.1.10 beschrieben wird.

22.2.1. DeviceServlet

Das `DeviceServlet` ist die Schnittstelle für das Herunterladen von Gerätedaten vom Server und das Hochladen von Gerätedateien auf den Server. Auf die Daten in den Bibliotheken iwrd über den zentralen `ResourceHandler` zugegriffen.
Folgende Plugins müssen für den Test bereits implementiert sein:

- diem.Model

Eingänge (Aufrufe über HTTP von einem Browser oder dem Client):

- HTTP-POST mit Parameter `path` und `device`
- HTTP-GET ohne Parameter
- HTTP-GET mit Parameter `path` und `id`

Ausgänge :

- `diem.server.ResourceHandler`

22.2.2. PayScaleServlet

Das `PayScaleServlet` ist die Schnittstelle für das Herunterladen von Tarifen und der Onlinetarif-Bibliothek vom Server und das Hochladen von Tarifdateien auf den Server. Auf die Daten in den Bibliotheken iwrd über den zentralen `ResourceHandler` zugegriffen.
Folgende Plugins müssen für den Test bereits implementiert sein:

- diem.Model

Eingänge (Aufrufe über HTTP von einem Browser oder dem Client):

- HTTP-POST mit Parameter `path` und `payscale`
- HTTP-GET ohne Parameter
- HTTP-GET mit Parameter `path` und `id`

Ausgänge :

- `diem.server.ResourceHandler`

22.2.3. ManagerServlet

Über das `ManagerServlet` werden die Daten des Servers verwaltet. Es ermöglicht die Nutzung eines Browsers, um auf den Server zuzugreifen. Auf die Daten in den Bibliotheken iwrd über den zentralen `ResourceHandler` zugegriffen.
Folgende Elemente müssen für den Test bereits implementiert sein:

- diem.Model

Eingänge (Aufrufe über HTTP von einem Browser oder dem Client):

- HTTP-POST mit Parameter `action` und `scope`
- HTTP-POST mit Parameter `action` und `scope`, sowie `path` und `id`
- HTTP-POST mit Parameter `action` und `scope`, sowie `path`, `id` und `movefrom`

Ausgänge :

- `diem.server.ResourceHandler`

22.2.4. ResourceHandler

Über ein `ResourceHandler`-Objekt wird auf ein EMF-Resource zugegriffen, das die Verbindung zur Persistenzebene über Teneo und Hibernate herstellt.
Folgende Elemente müssen für den Test bereits implementiert sein:

- diem.Model

Eingänge :

- `DeviceServlet`
- `PayScaleServlet`
- `ManagerServlet`

Ausgänge :

- Keine

22.2.5. Sicherheit und Verfügbarkeit

Das Thema Sicherheit wurde bereits in Abschnitt 20.3 dargelegt. Im Rahmen der Qualitätssicherung müssen auch diese Punkte überprüft werden.

- Alle zugriffsbeschränkten Elemente müssen sich in einem gesichertem Kontext befinden. Dazu gehören auch die JSPs, die nur zur Darstellung genutzt werden.
- Die Authentifizierungsmethode muss korrekt funktionieren und an entsprechender Stelle vom Server aufgerufen werden.

- Direkt aufgerufene JSPs dürfen zu keinem unbehandeltem Fehlverhalten des Servers führen.
- Der Server darf nur korrekte XML-Dateien akzeptieren, die dem `Device`- bzw. dem `Payscale`-Schema entsprechen.
- Der Server muss eventuell schadhafte Zeichenketten aus Formulardaten herausfiltern.
- Alle korrekten Parameterkombinationen müssen akzeptiert und verarbeitet werden, sofern sie den Spezifikationen aus Abschnitt 20.1 entsprechen.

Teil V.
Prototypen

23. Aufteilung der Prototypen

Autor: Michael Belasus

Innerhalb der Anforderungsdefinition wurden die funktionalen und nicht funktionalen Anforderungen beschrieben. Die Umsetzung dieser Anforderungen erfolgt über die Entwicklung von drei Prototypen. Dabei werden mit dem ersten Prototypen die grundlegenden Anforderungen umgesetzt. Das Anlegen und Bearbeiten von Geräten, -kompositionen, Tarifen und Lastgängen. Des Weiteren werden alle notwendigen Editoren und Ansichten implementiert. Innerhalb des zweiten Prototyps kommen dann die Funktionen für die automatische Einsatzplanoptimierung hinzu, ebenso wie die Möglichkeit Verknüpfungen zwischen dem Ablauf mehrerer Geräte festzulegen. Gleichzeitig wird begonnen das Handbuch und notwendige Tutorials zu erstellen. Mit dem dritten Prototypen wird dann das endgültige Layout festgelegt. Für den Anwender werden unterstützende Wizards und die Möglichkeit Zyklen festzulegen implementiert.

23.1. Erster Prototyp

Wie bereits beschrieben werden im ersten Prototypen grundlegende Anforderungen implementiert. Dazu gehören das

- Erstellen, Bearbeiten, Kopieren und Löschen von
  - Geräten
  - Gerätekompositionen
  - Tarifen und
  - Lastgängen.

Diese umfassen bereits einen Großteil der gestellten funktionalen Anforderungen, die im folgenden Kapitel 24 detaillierter aufgezählt werden. Dazu gehören sämtliche Editoren und verschiedene Ansichten. Innerhalb dieses Prototyps soll bereits die Möglichkeit implementiert werden, dass ein einfacher Einsatzplan erstellt werden kann.

Wie in der Anforderungsdefinition beschrieben worden ist, werden Geräte und Tarife zusätzlich über eine Onlinebibliothek eingepflegt. Dies ermöglicht es auf einen umfangreichen Datenbestand zurückzugreifen. Für die Entwicklung der Software und deren Funktionen ist es hilfreich auf diese Funktion bereits früh zugreifen zu können, daher wird innerhalb des ersten Prototyps bereits der Server mit der Updatefunktion für Geräte und Tarif umgesetzt.

23.2. Zweiter Prototyp

Der Anwender der Software kann selbst erstellte Geräte in die Onlinebibliothek laden. Diese werden dann als nicht verifiziert aufgenommen. Zur Verifizierung dieser Daten und dem Einpflegen von Tarifen wird eine Administrationsumgebung für den Server benötigt. Dieser wird innerhalb der Entwicklung des zweiten Prototyps implementiert.

Mit dem zweiten Prototypen wird innerhalb des Clients die Funktion *Erstellen von Constraints* implementiert. Über die Eingabe von Constraints legt der Anwender fest, welche Geräte nicht oder zwingend gleichzeitig eingeschaltet sind. Auch Abhängigkeiten, dass der Trockner nach der Waschmaschine eingeschaltet wird, kann festgelegt werden. Darauf aufbauend kann dann die automatische Einsatzplanung implementiert werden, so dass bei einem variablen Tarif eine eventuelle Kostenersparnis ermittelt werden kann. Damit Änderungen in Abläufen für den Anwender besser dargestellt werden können, ist eine visuelle Darstellung dieser Informationen angedacht. Diese wird ebenfalls innerhalb des zweiten Prototyps umgesetzt. Zur Verbesserung der Darstellung ist eine skalierbare Zeitachse vorgesehen. Diese ermöglicht es dem Anwender die Ansicht auf ein prinzipiell beliebiges Zeitintervall zu skalieren.

Zu den gestellten nichtfunktionalen Anforderungen gehören ein Handbuch und anwenderunterstützende Tutorials. Damit wird innerhalb des zweiten Prototyps begonnen.

23.3. Dritter Prototyp

Mit dem dritten Prototypen wird das endgültige Design der Software implementiert. Dazu gehören Farben, Grafiken und die Position der Ansichten und Editoren. Wizards, die bei der Einrichtung und den Funktionen der Software helfen, werden implementiert. Aufgrund der Gerätebibliothek existiert eine umgangreiche Datensammlung mit deren Verbrauch. Daher können dem Anwender Gerätesubstitutionen vorgeschlagen werden. Ebenfalls wird innerhalb des dritten Prototypen die Eingabe von Zyklen implementiert. Mithilfe von Zyklen können wiederkehrende Abläufe einfach durch den Anwender im Einsatzplan angelegt werden. Der vom Anwender erstellte Einsatzplan soll laut Anforderungsdefinition ausgedruckt werden können, damit nicht notwendigerweise der Computer gestartet werden muss. Diese Funktion wird in diesem Prototyp verwirklicht.

24. Prototyp 1

Wie bereits beschrieben, werden mit dem ersten Prototyp grundlegende Anforderungen umgesetzt. Es folgen im Einzelnen die 47 funktionalen Anforderungen aus der Anforderungsdefinition, die bereits innerhalb des ersten Prototyps implementiert werden.

Server

4.6.1 Online-Gerätebibliothek
4.6.1 a) Gerätedaten zur Online-Gerätebibliothek hinzufügen
4.6.2 Lokale Gerätebibliothek
4.6.2 a) Gerätedaten aus Internet beziehen
4.6.4 Lokale Tarifbibliothek
4.6.4 a) Tarifdaten aus dem Internet beziehen

Erstellen, Bearbeiten, Kopieren und Löschen von Geräten

4.1.1 Geräte verwalten
4.1.1 a) Gerät erstellen 4.1.1 b) Gerät bearbeiten 4.1.1 c) Gerät kopieren 4.1.1 d) Gerät löschen 4.1.1 e) Gerätedaten anzeigen
4.1.2 Betriebsmodi verwalten
4.1.2 a) Betriebsmodus erstellen 4.1.2 b) Betriebsmodus bearbeiten 4.1.2 c) Betriebsmodus kopieren 4.1.2 d) Betriebsmodus löschen 4.1.2 e) Lastgang importieren 4.1.2 f) Durchschnittsverbrauch aus Gerätelastgang berechnen

Erstellen, Bearbeiten, Kopieren und Löschen von Gerätekompositionen

4.5.2 Gerätekompositionen verwalten
4.5.2 a) Gerätekomposition erstellen 4.5.2 b) Gerätekomposition kopieren 4.5.2 c) Gerätekomposition löschen 4.5.2 d) Betriebsmodus hinzufügen 4.5.2 e) Betriebsmodus löschen

Erstellen, Bearbeiten, Kopieren und Löschen von Tarifen

4.2.1 Tarife verwalten
4.2.1 a) Fixtarif erstellen
4.2.1 b) Fixtarif bearbeiten
4.2.1 c) Variablen Tarif erstellen
4.2.1 d) Variablen Tarif bearbeiten
4.2.1 e) Tarif anzeigen
4.2.1 f) Tarif kopieren
4.2.1 g) Tarif löschen

Erstellen, Bearbeiten, Kopieren und Löschen von Lastgängen

4.3.1 Lastgänge verwalten
4.3.1 a) Lastgang erstellen
4.3.1 b) Lastgang bearbeiten
4.3.1 c) Lastgang kopieren
4.3.1 d) Lastgang löschen
4.3.1 e) Lastgang anzeigen

alle Editoren

alle Views

4.4.1 Profile verwalten
4.4.1 a) Profil erstellen
4.4.1 b) Profil laden
4.4.1 c) Profil löschen
4.4.1 d) Profil kopieren
4.4.1 e) Profileigenschaften bearbeiten
4.4.2 Profil bearbeiten
4.4.2 a) Gerät hinzufügen
4.4.2 b) Gerät löschen
4.4.2 c) Tarif hinzufügen
4.4.2 d) Tarif löschen
4.4.2 e) Lastgang hinzufügen
4.4.2 f) Lastgang löschen

4.5.3 Einsatzpläne verwalten
4.5.3 a) Einsatzplan erstellen
4.5.3 b) Einsatz planen
4.5.3 c) Tarif zum Einsatzplan hinzufügen
4.5.3 d) Tarif aus Einsatzplan löschen

Von diesen genannten Anforderungen entfällt 4.4.1 b) Profil laden. Da für die Profile ein eigener View implementiert wurde. Die Anforderungen 4.5.2 a) Gerätekomposition erstellen und 4.5.3 a) Einsatzplan erstellen sind nur bedingt ausführbar. Das Erstellen ist jeweils möglich, aber nicht mit eigenen Daten. Dies ist bedingt durch die Änderung, dass nicht mehr GEF verwendet wird. Dadurch verschieben sich auch die Anforderungen

- 4.5.2 d) Betriebsmodus hinzufügen
- 4.5.2 e) Betriebsmodus löschen
- 4.5.3 b) Einsatz planen
- 4.5.3 c) Tarif zum Einsatzplan hinzufügen
- 4.5.3 d) Tarif aus Einsatzplan löschen

Gründe, die gegen die Verwendung von GEF sprechen, werden in Kapitel 25.6 näher erläutert. Die Anforderung 4.1.2 c) *Betriebsmodus kopieren* konnte bisher auch nicht verwirklicht werden.

25. Abweichungen des ersten Prototypen

Dieser Abschnitt enthält die Ergänzugen und Änderungen zu Prototyp 1 gegenüber dem Entwurf und der Einteilung der Prototypen.

25.1. Datenmodell

Autor: Moritz Eysholdt

Im Datenmodell wurden gegenüber dem Entwurf folgende Änderungen vorgenommen:

- Einige Multiplizitäten wurden neu gesetzt:
  - Die Beschreibungen (Description) aller Elemente sind jetzt optional.
  - Die Felder Verified und Planable eines Gerätes ist jetzt optional.
- Beim Gerät (Device) wurde der Typ des Herstellungsjahres (YearOfConstruction) von date auf gyear umgestellt.
- Sowohl die Gerätebibliothek (Devicelib) als auch die Tarifbibiothek (Payscalelib) haben ein weiteres Datenfeld (SourceURI) erhalten, welches die URI speichert, von welcher die Bibliothek heruntergeladen und gegebenenfalls später wieder geupdated werden kann.

- Die Gerätebibliothek hat ein weiteres Datenfeld (UploadURI) enthalten, welches die URI speichert, unter welcher neue Geräte in der Gerätebibliothek gespeichert (hochgeladen) werden können.
- Geräte haben jetzt ein Datenfeld (EnergyEfficiencyGrade) für ihre Energieeffizienzklasse, wie sie hier beschrieben ist: http://www.kunnig-elektro.de/energie.htm
- Zu einem Betriebsmodus kann jetzt optional seine Dauer (Duration) in Sekunden gespeichert werden.
- Im Profil können jetzt alle Elemente in eine hierarchische ordnerstruktur bestehend aus ProfileFolder-Objekten einsortiert werden.
- Bei einem TimePricePair eines Tarifs ist der preis jetzt vom Typ float.

25.2. GUI & Layout

Autor: Christian Hinrichs

Da die GUI einer Anwendung maßgeblich von den eigentlichen Funktionalitäten und dem Zusammenspiel der Komponenten des Programmes abhängt, ist es besonders schwierig, das endgültige Layout von vornherein festzulegen. Im Entwurfsdokument wurden bereits einige Elemente der Benutzungsoberfläche vorgestellt. Diese beschränkten sich jedoch meist auf einzelne Komponenten und wurden losgelöst von ihrer Einbettung in die GUI betrachtet.
In diesem Abschnitt sollen die bis zum Zeitpunkt der Fertigstellung des ersten Protypen gefällten größeren Designentscheidungen vorgestellt und erläutert werden. Es ist darauf hinzuweisen, dass dies nicht zwangsläufig die endgültige Version der GUI darstellt, da sich viele Layout-Probleme erst während der Entwicklung und des Testens zeigen.

25.2.1. Aufteilung in Perspektiven

Betrachtet man die in der Phase "'Anforderungsanalyse"' aufgestellten Anforderungen an die Software, so lassen sich diese zu einigen wesentlichen Kategorien zusammenfassen:

- **Profil bearbeiten**
  Der Anwender soll die Möglichkeit erhalten, Geräte, Tarife, Lastgänge und Kompositionen anzulegen, zu bearbeiten und zu verwalten. Es sollen Profile angelegt werden können, welche eine Teilmenge dieser Elemente zu einer Gruppe zusammenfassen. Außerdem ist eine Kommunikation mit einem Server erwünscht, um diese Elemente entweder von einer Datenbank zu beziehen, oder dort öffentlich abzulegen.

- **Einsatzplanung durchführen und optimieren**
  Nachdem der Anwender seinen Haushalt adäquat über sein(e) Profil(e) abgebildet hat, ist er in der Lage, eine Einsatzplanung seiner Geräte durchzuführen. Diese sollte grafisch ansprechbar und übersichtlich sein und diverse Funktionalitäten und Freiheitsgrade beinhalten.

- **Lastgänge, Tarife und Einsatzpläne evaluieren**
  Der Hauptaspekt der Software soll das Ziel der Kostenersparnis sein. Daher muss eine Möglichkeit gegeben sein, die vorhandenen Elemente möglichst einfach gegeneinander abzuwägen. Dazu ist ein Vergleich von o.g. Elementen auf grafischer Basis erforderlich.

- **Automatisierte Entscheidungshilfen**
  Zusätzlich zu der manuellen Optimierung des Haushaltes ist eine automatische Unterstützung gewünscht, welche dem Anwender etwa Änderungen des Einsatzplanes oder der Gerätekonfiguration vorschlägt.

In diesen Kategorien ist bereits eine rudimentäre zeitliche Abfolge zu erkennen: Zunächst muss der Haushalt des Anwenders konfiguriert werden, indem bestimmte Geräte, Lastgänge und Tarife zu einem oder mehreren Profilen zusammengefasst werden. Über diese Profile kann nun eine Einsatzplanung durchgeführt werden. In einem dritten Schritt sollen die erstellten Konfigurationen optimiert werden.

Aufgrund dieser sinnvollen Gruppierung der Funktionen und der Tatsache, dass ein "'roter Faden"' einem unerfahrenen Anwender sehr hilfreich sein kann, wurde entschieden, dass die Benutzungsoberfläche der Anwendung in drei Perspektiven eingeteilt wird. Eine Perspektive wird definiert als eine Untermenge der verfügbaren Anzeigeelemente, plus einer sinnvollen Anordnung dieser Elemente innerhalb des Anwendungsfensters. Perspektiven nehmen zudem Einfluss auf die dargestellten Menüpunkte und Buttons der Toolbar, um nicht benötigte Funktionalitäten auszublenden, sodass eine Perspektive immer genau ein Themengebiet wiederspiegelt. Daraus ergeben sich zu diesem Zeitpunkt drei Perspektiven, welche jeweils einer der oben aufgezählten Kategorien entsprechen. Die vierte Kategorie "'Automatisierte Entscheidungshilfe"' soll dabei nicht in einer eigenen Perspektive verankert, sondern über die anderen verteilt werden, da die beinhalteten Funktionen sowohl auf die Profilkonfiguration, als auch auf die Einsatzplanung einwirken.

In Abbildung 104 ist die bisher verwirklichte Profilperspektive dargestellt. Im oberen Bereich befindet sich auf der linken Seite eine Auflistung der vorhandenen Profile. Im oberen rechten Bereich sind die Bibliotheken und Sammlungen zu finden, welche alle zu verwendenden Geräte, Tarife etc. beinhalten. Der gesamte untere Bereich der Perspektive ist für die einzelnen Editoren reserviert. Dieser Bereich stellt einen Container dar, sodass mehrere Editoren (auch verschiedenen Typs) gleichzeitig geöffnet sein können. Die Anordnung erfolgt standardmäßig über Registerkarten, sodass immer nur ein

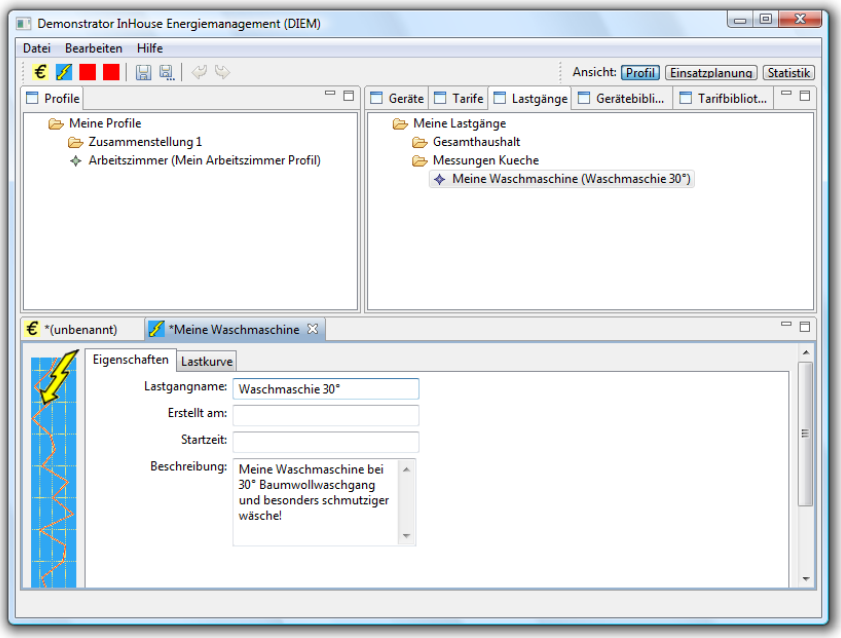

Abbildung 104: Die Profilperspektive

Editor sichbar ist. Bei Bedarf können einzelne Editoren jedoch mit der Maus innerhalb es Containers beliebig positioniert werden, sodass es z.B. möglich ist, zwei ähnliche Geräte zu öffnen und die Editoren so anzuordnen, dass beide Inhalte sichtbar und damit visuell vergleichbar sind.
In der Toolbar und im Menü befinden sich nur solche Elemente, welche dieser Perspektive zugeordnet sind.

Ganz oben rechts in der Grafik ist auf der Höhe der Toolbar die Perspektivenleiste angebracht. Diese erlaubt das bequeme Umschalten zwischen den Perspektiven, außerdem zeigt sie als Orientierungshilfe an, in welcher Perspektive sich die Anwendung derzeit befindet. Abbildung 105 zeigt eine Vergrößerung der Perspektivenleiste.

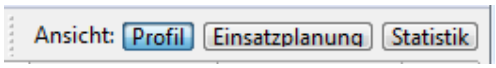

Abbildung 105: Die Perspektivenleiste

Möchte der Anwender zu einer anderen Perspektive wechseln, so muss er auf den entsprechenden Button der Perspektivenleiste klicken. Sind während dieser Aktion allerdings noch Editoren mit ungespeichertem Inhalt geöffnet, so erscheint ein Hinweis mit der Möglichkeit, diese Editoren zu speichern, die Änderungen zu verwerfen, oder doch in der aktuellen Perspektive zu bleiben. Dies ist erforderlich, da es dem Prinzip der

durch Themen getrennten Perspektiven widersprechen würde, wenn geöffnete Editoren in einer anderen Perspektive weiterhin geöffnet sein würden. Dieser Vorgang ist in Abbildung 106 dargestellt.

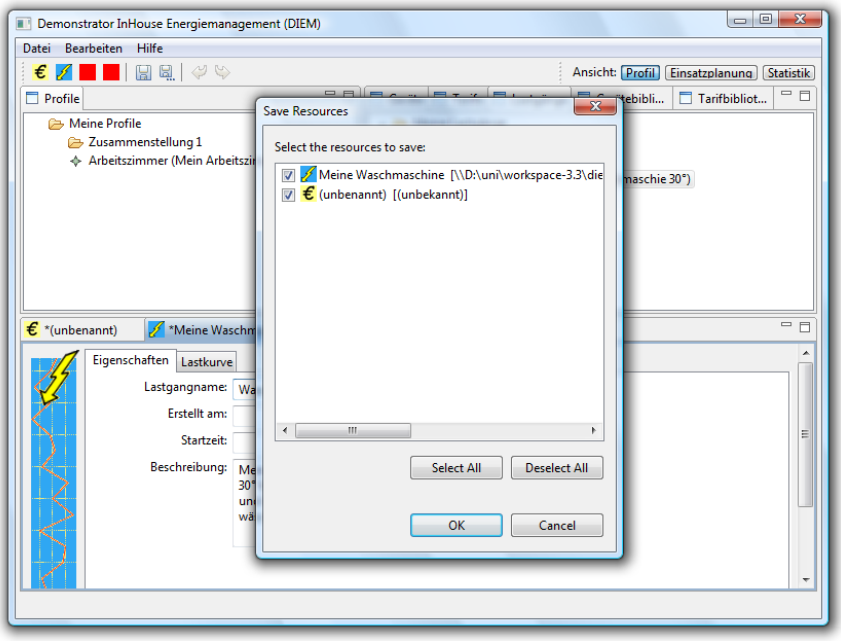

Abbildung 106: Speichern bei Perspektivenwechsel

Abbildung 107 zeigt den aktuellen Stand der Einsatzplanperspektive. Der Unterschied zur Profilperspektive ist deutlich zu erkennen. Statt der horizontalen Unterteilung in Sammlungen und Editorbereich ist in dieser Perspektive eine vertikale Trennung vorgenommen worden. Im linken Bereich befindet sich eine Übersicht der angelegten Profile. Die Sammlungen sind an dieser Stelle nicht mehr erforderlich, da eine Einsatzplanung immer nur mit Elementen aus einem Profil erfolgen kann. Der rechte Bereich des Fensters stellt wiederum einen Editorbereich mit den gleichen Eigenschaften wie in der Profilperspektive dar, nur dass er nun Einsatzplaneditoren enthält. Zu beachten sind auch die Veränderungen der Menüs und der Toolbar, welche jetzt nur noch die für diese Perspektive relevanten Elemente enthalten.

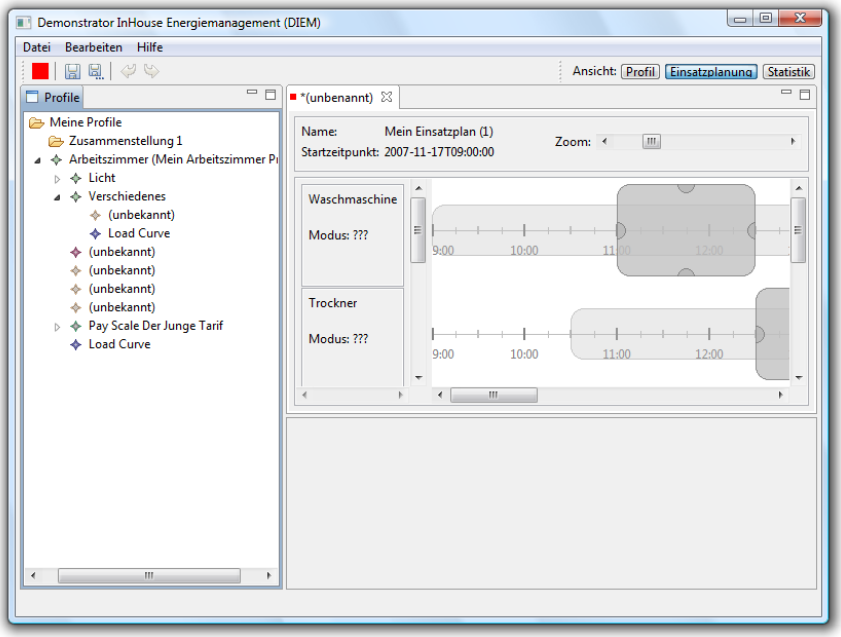

Abbildung 107: Die Einsatzplanperspektive

25.2.2. Änderungem der Editoren und Views

Im Vergleich mit den entsprechenden Abschnitten im Kapitel "'Plugins"' des Entwurfsdokumentes sind auf den obigen Screenshots weitere Änderungen in den einzelnen Editoren und Views zu sehen. Da diese Elemente sich aber entweder nur kaum verändert haben (zum Beispiel die Positionierung von Eingabefeldern) oder voraussichtlich noch weiter angepasst werden (wie etwa der Einsatzplaneditor) soll an dieser Stelle keine detaillierte Aufzählung erfolgen.

25.2.3. Implementierung des Layouts

Um die in obigem Abschnitt 25.2.1 vorgestellten Designentscheidungen durchzuführen mussten einige Anpassungen und Erweiterungen am Code vorgenommen werden.

Im Plugin `diem.app` (vergleiche Abschnitt "'diem.App"' des Kapitels "'Plugins"' im Entwurfsdokument) wurde eine Klasse `actions.OpenPerspectiveAction` erstellt, welche parametrisiert instanziiert wird und zur Laufzeit mehrfach (für je eine Perspektive einmal) im Menü "'Ansicht"' enthalten ist. Die Ausführung der Action bewirkt den Wechsel zur gewünschten Perspektive. Die Klasse achtet selbstständig auf Perspektivenwechsel, sodass am Status der Buttons bzw. Menüeinträge stets die aktuelle Perspektive abzulesen ist.
Außerdem wurde in diesem Plugin ein zusätzliches Package `perspectives` angelegt, das folgende Elemente enthält:

- `ProfilePerspective`, `SchedulePerspective`, `StatisticsPerspective`
  Diese Klassen implementieren das Interface `IPerspectiveFactory` und stellen die Profil-, Einsatzplan- und Statisitikperspektive dar, indem über die Methode `createInitialLayout(IPageLayout)` entsprechende Views geöffnet und positioniert sowie Platzhalter für die Editoren angelegt werden. Zusätzlich werden die für die jeweilige Perspektive benötigten `Actions` aktiviert, indem das korrespondierende `ActionSet` geladen wird, welches in der `plugin.xml` definiert ist und eine Gruppierung von `Actions` verkörpert.

- `DiemPerspectiveBarContribution`
  Diese Klasse erzeugt die Perspektivenleiste, mittels derer zwischen den verschiedenen Perspektiven umgeschaltet werden kann. Sie ist von der Klasse `WorkbenchWindowControlContribution` abgeleitet, ihre Einbindung erfolgt über die `plugin.xml` dieses Plugins. Der konkrete Perspektivenwechsel erfolgt über die Ausführung einer mit der gewünschten Perspektive instanziierten `OpenPerspectiveAction`.

25.3. EMF.edit

Autor: Moritz Eysholdt

Das EMF.edit-Framework ist ein auf dem EMF aufbauendes und es ergänzendes Framework. Ein Großteil seines Codes wird für dieses Projekt automatisch aus dem Datenmodell generiert. Im Entwurf wurde das EMF.edit Framework als etwas Verfügbares dargestellt, dessen Nützlichkeit sich erst herausstellen müsse. Inzwischen hat sich diese Nützlichkeit bestätigt.
Betrachtet man die Anwendung aus der Model-View-Controler Perspektive, so ist der View klar durch das SWT und das Model durch das EMF gegeben. Der an das SWT anbindende Teil des Controllers wird durch JFace gegeben und es stellt sich die Frage nach dem Bindeglied zwischen JFace und dem EMF. Hier kommt EMF.Edit ins Spiel.
Das EMF.Edit Framework bildet die Infrastruktur, welche ein EMF-Modell durch eine GUI bearbeitbar macht und stellt hierzu eine Vielzahl von Klassen bereit, die sich nahtlos in JFace und den Eclipse Workbench integrieren. Dazu Zählen vorgefertigte Actions für Copy'n'Paste und Löschen sowie die zugehörigen Commands, welche von den Actions erstellt werden und die tatsächlichen Änderungen am Modell durchführen. Die Commands sind auch in der Lage, ihre Änderungen wieder Rückgängig zu machen. Zusammen mit einem Commandstack, in welchem alle ausgeführten Commands abgelegt werden, ergibt sich so eine volle Unterstützung für Undo/Redo.
Der automatisch generierte Teil des EMF-Frameworks besteht aus sogenannten ItemProvidern, welche spezifisch für jeweils eine Klasse des Modells sind. Ein ItemProvider implementiert diverse von JFace benötigten Provider-Interfaces und ermöglicht so mit minimalem eigenem Programmieraufwand den Inhalt des Datenmodells in einem TreeViewer oder ListViewer anzuzeigen.
Das Konzent des EMF.edit-Frameworks ist es, möglichst viele Standardprobleme mit möglichst allgemein gehaltenem Code zu lösen. Es lässt sich auch als eine große Menge wiederverwendbaren Codes betrachten, der keine klare Schittstelle zum eigenen Code hat und dessen Verwendung optional ist.
Letzteres hat massiv die Erfahrungen in diesem Projekt mit dem EMF.edit-Framework geprägt: Dadurch, dass es keine klare Schnittstelle zum eigenen Code gibt funktioniert das Konzept des Information Hiding nicht mehr und der Entwickler muss sämtliche Implementierungsdetails des Frameworks kennen, um es nutzen zu können. Das hat den Einarbeitungsaufwand beträchtlich in die Höhe getrieben. Auf der anderen Seite war es sehr hilfreich, viel Standardfunktionalität wie Copy'n'Paste, Drag'n'Drop etc. sofort nutzen zu können. Des weiteren war die Strukturvorgabe des Frameworks (z.B. das Konzept von Actions, Commands, dem Commandstack und deren Zusammenspiel) eine große Erleicherung.
Tatsächlich zur Anwendung kommt das EMF.edit Framework in allen Baumansichten von Sammlungen, Bibliotheken und im baumartigen Profileditor.

25.4. JFace Databinding

Autor: Michael Erlmann

Eine maßgebliche Erweiterung gegenüber dem Entwurf stellt die Verwendung des Frameworks JFace Databinding dar.
Databinding kümmert sich um die automatische Validierung und Synchronisation von Werten zwischen Objekten. Das bedeutet, dass Änderungen in Eingabefeldern von Oberflächen wie dem SWT aus dem im Projekt verwendeten JFace-Toolkit direkt ins Model übertragen werden, ohne hierfür extra einen Listener schreiben und an das Eingabefeld anmelden zu müssen, welcher sich dann um Konvertierung in andere Datenformate und das Schreiben ins Model kümmern müsste. Ebenso ist es nicht notwendig, dass sich eine Observerklasse an den Notifier im Model anmeldet und dortige Änderungen dann ins Eingabefeld überträgt. Eben diese Aufgaben übernimmt das Databinding-Framework. Hierdurch wird dem Programmierer abgenommen, für jedes Eingabefeld ein eigenes Eventhandling anfertigen zu müssen. Auch um die Konvertierung der Eingaben, welche aus den Eingabefeldern zumeist als String- oder Booleanwert zurückgegeben werden, kümmert sich das Databinding.

Beim Databinding wird ein Kontext angelegt und innerhalb dieses Kontextes werden die einzelnen Eingabefelder mit den jeweiligen Objekten oder Variablen im Model verbunden. Die Validierung und Konvertierung erfolgt innerhalb eines Kontextes automatisch. Um hier möglichst flexibel und erweiterbar zu bleiben, kann der Programmierer eigene Klassen zur Validierung und Konvertierung anlegen und jeweils für eine Bindung anmelden. Dadurch ist es möglich aus der Eingabe, welche aus dem Eingabefeld gelesen wird, jede beliebige Ausgabe zu erzeugen. So wird zum Beispiel die Eingabe aus einem Textfeld, welches mit einer Integer-Variable im Model gebunden ist, aus einem String- in einen Intwert umgewandelt. Auch Umwandlungen in beliebige komplexe Objekte sind dabei möglich. Ebenso ist dies umgekehrt der Fall. So kann man einen Wert im Model, der z.B. als Date-Wert gespeichert ist, so in einen String umwandeln, dass er regionalen Darstellungsformen entspricht.
Falls eine Konvertierung wegen nicht eingehaltener Konventionen, wie zum Beispiel einem Buchstaben in einem Feld 'Alter', nicht möglich ist, so wird dies von der Validator-Klasse abgefangen. In ihr kann man je nach Art des Fehlers reagieren und entweder eine detailierte Fehlernachricht erzeugen oder die Eingabe abändern, so dass sie doch den Konventionen entspricht. Diese Validatorklasse kann an drei Stellen angemeldet werden (Vergleiche Abbildung 108):

1. nachdem eine Änderung festgestellt wurde, bevor sie an die Konverterklasse übergeben wird,
2. nachdem eine Änderung durch die Konverterklasse umgewandelt wurde oder
3. bevor eine Änderung in das jeweils andere Ende geschrieben wird.

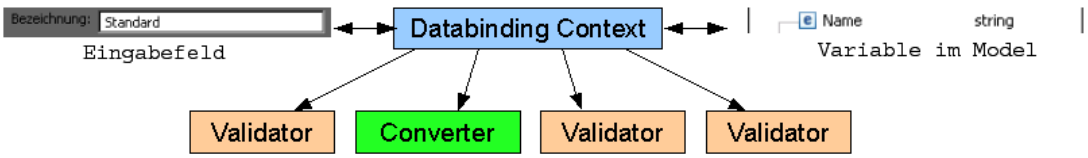

Abbildung 108: Databinding zwischen Eingabefeld und Model

Zudem ist es möglich, die Fehlerausgabe eines Kontextes an eine Variable im Model oder an ein Feld zu binden. So kann man die Ausgaben der Validatoren an ein Label binden. Auch diese Bindung läuft erneut durch eine Validatorklasse und eine Konverterklasse, wodurch diese Ausgaben entsprechend den Wünschen des Programmierers umgewandelt und aufbereitet werden können, wie einen einfachen Hinweis, einen Informationstext zur Fehlerbehebung oder gar eine Grafik.

Insgesamt vereinfacht Databinding den Umgang mit Eingabefeldern erheblich. Es spart Programmierarbeit ein und ist durch seine Flexibilität äußerst erweiterbar, wodurch auch das Produkt, welches Databinding verwendet, profitiert.

25.5. Server

Autor: Christian Bley

Der Server soll dem Anwender die Möglichkeit bieten, stets akutelle Geräte und Tarife von einer zentralen Quelle beziehen zu können. In dieser Hinsicht wurden im Entwurf die Kernfunktionen erläutert, die der Client verwenden soll.

- Ersetzen der lokalen Gerätebibliothek, durch die aktuelle vom Server
- Hochladen eines selbst erstellten Gerätes in die Gerätebibliothek des Servers
- Ersetzen der lokalen Tarifbibliothek, durch die aktuelle vom Server

Im Prototyp 1 wurden diese, sowie die Anforderungen an die Sicherheit des Servers umgesetzt, weitestgehend wie in den Abschnitten 6.3 und 6.4 des Entwurfes beschrieben.
Es wurde auf die Möglichkeit verzichtet, Geräte oder Tarife einzeln als Dateien vom Server zu laden. Da bei den übergebenen Pfadangaben so keine Referenz auf spezielle Geräte bzw. Tarife notwendig ist, wurde das Pfadschema dementsprechend angepasst: `/(<Ordnername>/)*`.

25.5.1. Zusätzliche Funktionen

Um die Funktionen des Servers testen zu können wurden bereits einige der zum Management gehörenden JSPs umgesetzt, die `index.jsp`, `devicedialog.jsp` und `payscaledialog.jsp`.

Im Entwurf wurde keine Möglichkeit vorgesehen, eine ganze Bibliothek als XML-Datei, wie sie lokal vom Client gespeichert wird, in der Datenbank des Servers zu hinterlegen. Da diese Option die Einrichtung bzw. den Neustart des Servers sehr erleichtert, wurde eine weitere Klasse `de.oldenburg.uni.informatik.diem.server.AdminServlet` erstellt, die diese Funktionen umsetzt. Diese wird von diemadmin.jsp aufgerufen, indem eine Bibliothek-Datei hochgeladen und zusätzlich ein Kennwort übergeben wird. Das Kennwort dient der Vorbeugung eines versehentlichen Überschreibens der bestehenden Bibliothek. Bei korrektem Kennwort wird die bestehende Bibliothek durch die neue ersetzt.
Alle Servlet-Klassen erben von der Klasse `de.oldenburg.uni.informatik.diem.server.DefaultServlet`, die Methoden und Funktionen enthält, die für alle Servlets des Servers sinnvoll sind, z.B. Loggingfunktionen.

25.5.2. Zusätzliche Komponenten

Für den Server wurden zusätzlich folgende Komponenten verwendet.

Apache Log4J ist eine Logginglösung für Java von der Apache Software Foundation. Sie wurde in der Version 1.2 verwendet, da hiermit auch die Loggingausgaben der benutzen Frameworks EMF, Teneo und Hibernate konfiguriert und umgeleitet werden können.
(`http://logging.apache.org/`)

JSTL 1.1 ist die Java Standard Tag Library Implementierung vom Jakarta Projekt der Apache Software Foundation. Diese ermöglicht eine übersichtlichere Arbeit in JSP-Dateien, da programmative Vorgänge in XML-Tags gekapselt werden.
(`http://jakarta.apache.org/taglibs/index.html`)

Apache Commons Fileupload ist eine Hilfsbibliothek des Apache Commons Projektes der Apache Software Foundation. Es ermöglicht einen einfachen Umgang mit Daten, die per HTTP-Post an ein Servlet gesendet werden. Diese werden dabei vom Browser in einen BASE64 kodierten Datenstrom umgewandelt. Im Servlet wird dann dieser Datenstrom von einen FileUploadobjekt gefiltert und die einzelnen Elemente dann als Liste zurückgegeben. Die Daten werden im Speicher oder in temporären Dateien gespeichert, die dann weiterverarbeitet werden können.
(`http://commons.apache.org/fileupload/`)

25.5.3. Persistenz

Im Entwurf wurde festgelegt, dass für die Persistenz die gleichen Mechanismen wie im Client verwendet würden, d.h. das Laden und Speichern über EMF-Resourcen erfolgt. Während der Entwicklung zeigte sich, dass das Verwenden der von Teneo ge-

steuerten EMF-Resourcen schlecht zu Überwachen bzw. zu Debuggen war. Aus diesem Grund erfolgt das Laden und Speichern von Objekten mit der bei Hibernate üblichen Methode, über eine Hibernate-Session. Dies entspricht der Vorgehensweise, wie sie in der Teneo-Dokumentation beschrieben wird. Sämtliche Vorzüge des EMF-Frameworks können weiterhin genutzt werden.
Für das Erzeugen bzw. Lesen von Objekten aus XML-Daten, werden wie beschrieben die EMF-Resourcen des Modells verwendet.

25.6. Graphical Editing Framework

Autor: Mark Eilers

Zum Ende des ersten Prototyps hat sich die Projektgruppe dazu entschieden, dass Graphical Editing Framework GEF nicht länger zu benutzen. Im Laufe der Entwicklung haben sich die Mechanismen des Frameworks als zu komplex und starr erwiesen um die an die grafischen Editoren gestellten Erwartungen bei vertretbarem Aufwand zu erfüllen. Einige Gründe für diese Entscheidung sollen im Folgenden kurz erläutert werden.

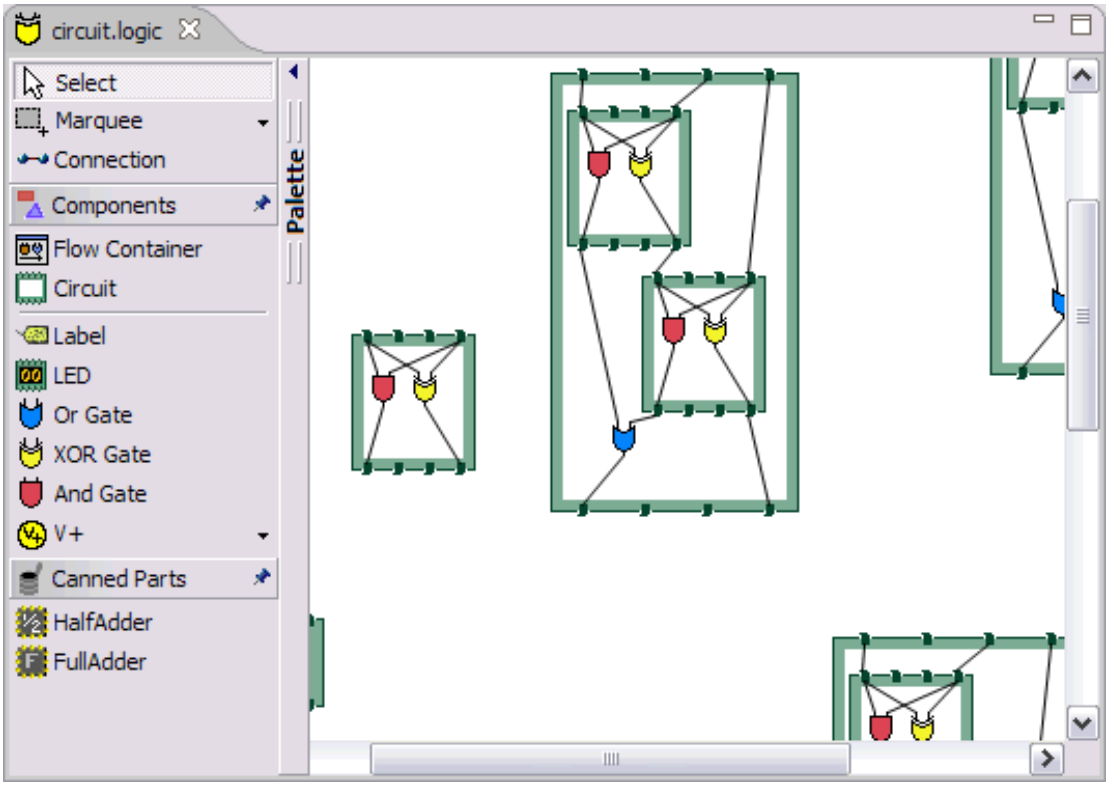

Abbildung 109: Ein GEF-Standard Editor

Die verfügbaren Beispiele und die vorhandene Dokumentation beziehen sich auf eine Art grafischen Standard-Editor (siehe Abb. 109), bestehend aus einer Werkzeugpalette

und der eigentlichen Arbeitsfläche, auf der alle zu bearbeitenden Daten, nach Wunsch des Entwicklers, angeordnet und dargestellt werden. Allen diesen Standard-Editoren ist gemein, dass das sichtbare Gerüst des Editors, also die Scrollbars und Begrenzungen, ein einzelnes Fenster auf die Arbeitsfläche darstellt. Dagegen bestehen die geplanten grafischen Editoren der Projektgruppe (vergl. Abb. 110) aus vielen teilweise voneinander abhängigen Teilen mit eigenen Scrollbars und ganz unterschiedlichen Funktionaltitäten, wodurch der ursprüngliche Plan, die grafischen Editoren ganz in GEF zu verwirklichen, hinsichtlich Funktionalität, Layout und Design, nicht im Sinne der Projektgruppe umsetzbar war. So sind die im Einsatzplan-Editor (siehe Abb. 110) am linken Rand angesiedelten Platzhalter für die Geräteeinsätze zwar unmittelbar mit diesen durch eine vertikale Scrollbar verknüpft, bleiben jedoch von der horizontalen Scrollbar unberührt. Dafür beeinflusst diese die Anzeige der Lastgänge und Tarif am unteren Editor-Rand, die wiederum von der vertikalen Scrollbar unabhängig sein muss. Ohne Dokumentation wie GEF an solch eine Situation angepasst werden könnte, blieben nur die Möglichkeiten, das Layout radikal zu ändern oder alle notwendigen Elemente wie im Standard-Editor auf einer großen Arbeitsfläche anzuzeigen. Da beide Möglichkeiten der Vorstellung einer intuitiven auf den späteren Nutzer zugeschnittenen Bedienung widersprachen, blieb als Ausweg nur, die Verwendung von GEF auf die tatsächlich bearbeitbaren Teile der gewünschten Editoren zu beschränken und die, ursprünglich als reine GEF-Editoren gedachten, Editoren in normale grafische Benutzeroberflächen mit integriertem grafischen Editor umzuwandeln.
Leider hat sich auch dieses Vorhaben im weiteren Verlauf der Implementierung als erfolglos erwiesen, da sich vor allem die großen Vorteile von GEF in diesem Projekt große Probleme verursachten. So bietet die Verwendung von GEF nach einiger Einarbeitungszeit zwar einen raschen Fortschritt hinsichtlich der Funktionalität, diese Funktionalität muss jedoch in einem mühseligen Prozess wieder auf das tatsächlich gewünschte Maß reduziert und angepasst werden. Als Beispiel soll hier das Verschieben eines Funktionspunktes auf der Arbeitsfläche betrachtet werden: Durch das Verwenden bestimmter RootEditParts bietet GEF die sofortige Fähigkeit, beim Verschieben eines Objektes über die Grenzen der Arbeitsfläche, diese, mit sofortiger Anpassung aller Scrollbars etc., entsprechend zu vergrößern. GEF bietet hierfür ein SimpleRootEditPart, das erlaubt die Arbeitsfläche nach rechts und nach unten, sowie ein FreeformRootEditPart, das erlaubt die Arbeitsfläche in alle vier Richtungen zu erweitern. Für unseren Funktionseditor muss die Arbeitsfläche jedoch nach oben und rechts erweiterbar sein. Da dieser Fall wohl so nicht vorgesehen war muss ein FreeformRootEditPart verwendet werden und die Erweiterbarkeit der Arbeitsfläche nach links und nach unten explizit verboten werden, ein Unterfangen, dass mangels ausreichender Dokumentation nicht ganz einfach und mangels genauer Kenntnis der tatsächlichen Arbeitsweise von GEF, auch sehr fehleranfällig ist.
Diese und viele andere Probleme im Layout, Design, Bedienbarkeit und Speichermanagement, die hier nicht weiter ausgeführt werden, haben dazu geführt, dass sich die Projektgruppe schon während der Entwicklung des ersten Prototypen Gedanken über Alternativen gemacht hat. Vermutlich ist es möglich, die grafischen Editoren durch GEF

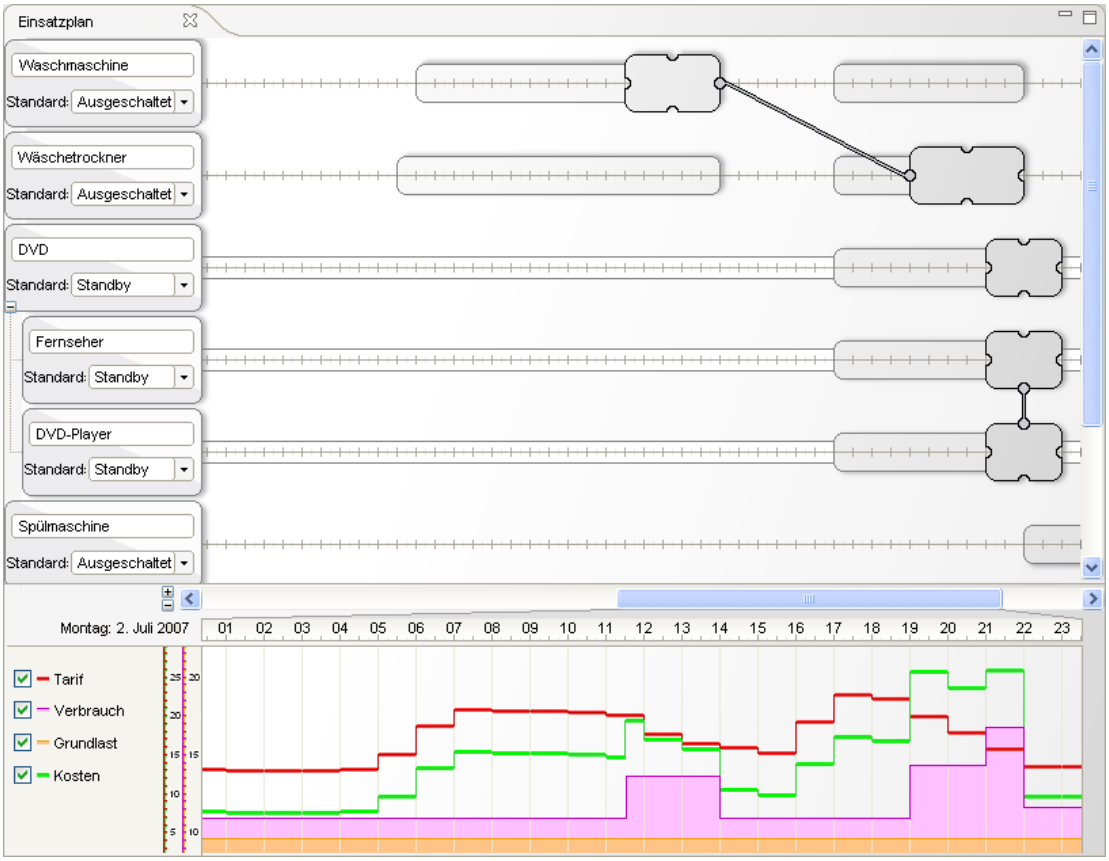

Abbildung 110: Der Einsatzplan-Editor

zu verwirklichen, doch die bisherigen Versuche die Vorgaben und Anwendungsregeln eines GEF-Editors an die Wünsche der Projektgruppe anzupassen, benötigten mehr Zeit als eine vergleichbare Umsetzung ohne GEF und drohten den verfügbaren Zeitrahmen für die Entwicklung der grafischen Editoren zu sprengen. GEF ist ein sehr mächtiges Framework zur Erstellung von hoch komplexen Editoren, für dieses Projekt aber leider ungeeignet. Die Projektgruppe hat sich daher entschlossen das Framework GEF für die weitere Entwicklung nicht länger zu verwenden sondern die grafischen Editoren durch eine Mischung aus SWT, Draw2d und den in Java verfügbaren Mouse-Listenern zu realisieren.

26. Prototyp 2

26.1. Einleitung

Autor: Michael Belasus

Im ersten Prototyp wurden grundlegende Funktionen wie das Anlegen und Bearbeiten von Geräten und Tarifen implementiert. Mit dem zweiten Prototyp wird innerhalb des Clients die Funktion *Erstellen von Constraints* implementiert. Über die Eingabe von Constraints legt der Anwender fest, welche Geräte nicht oder zwingend gleichzeitig eingeschaltet sind. Auch Abhängigkeiten, dass z.B. der Trockner nach der Waschmaschine eingeschaltet wird, kann festgelegt werden. Darauf aufbauend kann dann die automatische Einsatzplanung implementiert werden, so dass bei einem variablen Tarif eine eventuelle Kostenersparnis ermittelt werden kann. Damit Änderungen in Abläufen für den Anwender besser dargestellt werden können, ist eine visuelle Darstellung dieser Informationen angedacht. Diese wird ebenfalls innerhalb des zweiten Prototyps umgesetzt. Zur Verbesserung der Darstellung ist eine skalierbare Zeitachse vorgesehen. Diese ermöglicht es dem Anwender die Ansicht auf ein prinzipiell beliebiges Zeitintervall zu skalieren.

Zu den gestellten nichtfunktionalen Anforderungen gehören ein Handbuch und anwenderunterstützende Tutorials. Damit wird innerhalb des zweiten Prototyps begonnen. Der Anwender der Software kann selbst erstellte Geräte in die Onlinebibliothek laden. Diese werden dann als nicht verifiziert aufgenommen. Zur Verifizierung dieser Daten und dem Einpflegen von Tarifen wird eine Administrationsumgebung für den Server benötigt. Dieser wird innerhalb der Entwicklung des zweiten Prototyps implementiert.

26.2. Funktionale Anforderungen 2. Prototyp

Mit dem zweiten Prototyp werden folgende funktionale Anforderungen aus der Anforderungsdefinition implementiert.

- **automatische Einsatzplanoptimierung**

4.5.3 Einsatzpläne verwalten
4.5.3 e) Einsatzplan optimieren

- **visuelle Statistik**

4.5.4 Einsatzpläne vergleichen
4.5.4 a) Erwarteten mit gemessenem Stromverbrauch vergleichen
4.5.4 b) Zwei Einsatzpläne visuell vergleichen
4.5.4 c) Einsatzpläne preislich vergleichen

- **Constraints**

4.5.1 Constraints verwalten
4.5.1 a) Constraint hinzufügen
4.5.1 b) Constraint bearbeiten
4.5.1 c) Constraint löschen

- **Serveradministration**

4.7.1 Online-Gerätebibliothek verwalten
4.7.1 a) Geräte auflisten
4.7.1 b) Gerät verifizieren
4.7.1 c) Gerät löschen
4.7.1 d) Gerät als Datei vom Server herunterladen
4.7.1 e) Gerät als Datei auf den Server übertragen
4.7.2 Online-Tarifbibliothek verwalten
4.7.2 a) Tarife anzeigen
4.7.2 b) Tarif als Datei vom Server herunterladen
4.7.2 c) Tarif als Datei auf den Server übertragen
4.7.2 d) Tarif löschen

26.3. Ergänzende Funktionale Anforderungen aus dem 1. Prototyp

Innerhalb des ersten Prototyps konnten nicht alle gestellten funktionalen Anforderungen umgesetzt werden. Diese werden im zweiten Prototyp nachgeholt. Dabei handelt es sich um folgende funktionale Anforderungen:

- 4.5.2 a) Gerätekomposition erstellen
- 4.5.3 a) Einsatzplan erstellen
- 4.5.2 d) Betriebsmodus hinzufügen
- 4.5.2 e) Betriebsmodus löschen
- 4.5.3 b) Einsatz planen
- 4.5.3 c) Tarif zum Einsatzplan hinzufügen
- 4.5.3 d) Tarif aus Einsatzplan löschen

Die Gründe für die Verschiebung wurden im Dokument Prototyp 1 erläutert.

27. Änderungen und Erweiterungen zum Entwurf durch Prototyp 2

Im Folgenden werden die Bereiche, welche in der Umsetzung des zweiten Prototypen behandelt werden, und bei denen es eine Änderung oder Ergänzung zum Entwurf gibt, betrachtet.

27.1. Tarif

Autor: Michael Belasus

In der Anforderungsdefinition wurden drei Typen von Tarifen vorgestellt: Standardtarif, Time-Of-Use-Tarif und Day-Ahead-Tarif. Bei einem Standardtarif gibt es nur einen festen Preis. Der Time-Of-Use-Tarif hat jeweils unterschiedliche Preise. Für beide Tarife wird jeweils eine Dauer gespeichert, so dass längere Einsatzpläne erstellt werden können. Damit können die Einsatzpläne auch über einen längeren Zeitraum verglichen werden. Bei einem Day-Ahead-Tarif werden die Tarifinformationen für den kommenden Tag aus dem Internet bezogen. Damit auch bei einem solchen Tarif eine Vergleichsmöglichkeit besteht, werden die Informationen zu diesem flexiblen Day-Ahead-Tarif wie in Abbildung 111 dargestellt gespeichert.

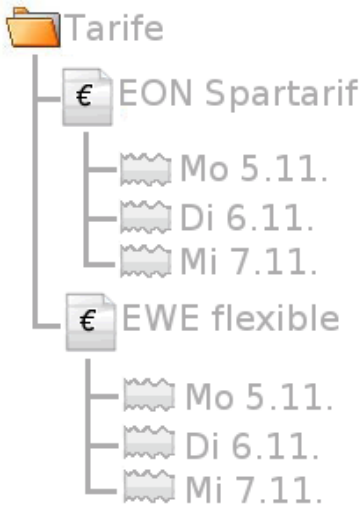

Abbildung 111: Tarife

Beispielsweise gibt es die fiktiven Day-Ahead-Tarife Eon Spartarif und EWE flexible. Die Preisinformationen für alle Tarife werden in einer XML-Datei gespeichert. Die täglich hinzukommenden neuen Preise bei einem Day-Ahead-Tarif werden für den jeweiligen Tarif innerhalb eines Monats in die bereits vorhandene XML-Datei dazugefügt. Dadurch lassen sich Preisverläufe über einen beliebig großen Zeitraum darstellen. Sofern ein Anwender mal eine zeitlang kein Update der Tarifinformationen durchgeführt

hat, kann er die fehlenden Informationen nachladen. Mögliche Vergleiche wie nur die Preise von Montagen lassen sich sehr einfach in der Statistik darstellen. Ob der gesamte Zeitraum immer weiter anwächst oder begrenzt wird, z. B. auf ein Jahr oder nur ein halbes Jahr, kann noch nicht festgelegt werden. Zurzeit können noch nicht die Größe der dadurch entstehenden Dateien und die Zeitspanne für den statistischen Vergleich abgeschätzt werden. Dies kann nach der Fertigstellung der Statistik und der Implementierung dieses Models besser abgeschätzt werden.

Für eine bessere Bedienbarkeit ist vorgesehen, dass nach erfolgtem Update der Tarife automatisch geprüft wird, welche Tarife der Anwender in die Tarifsammlung übernommen hat. Diese werden dann mit denen aus der Tarifbibliothek verglichen. Bei Unterscheidungen erscheint ein Dialog in dem der Anwender auf die neueren Tarifinformation in der Tarifbibliothek hingewiesen wird. Jetzt kann der Anwender entscheiden ob die vorhandenen Tarifinformationen mit den neuen überschrieben oder ergänzt werden sollen. Da der Anwender die Möglichkeit hat Tarife selbst zu verändern, muss die Möglichkeit eines merge berücksichtigt werden. Die neuen Tarifinformationen werden dann entsprechend in die Tarifsammlung übernommen. Da in den Einsatzplänen auf die Tarife in der Tarifsammlung referenziert wird, sind diese automatisch aktualisiert und um einen weiteren Tag bei einem Day-Ahead-Tarif erweitert.

27.2. Profilperspektive

Autor: Christian Hinrichs

Während der Anwendungstests der Software hat sich ergeben, dass die bestehende Profilperspektive recht unintuitiv ist, zusätzlich wurde die bisherige Mischung aus Sammlungen und Bibliotheken als zu umständlich erachtet. Daher hat die Projektgruppe entschieden, diese Konzepte zu ändern, indem einerseits die Profil-, Lastgang- und Tarifsammlung zu einer einzelnen Sammlung vereinigt werden. Diese gemeinsame Sammlung enthält die bisherigen Elemente jeweils als Unterordner einer Baumstruktur. Andererseits wird die Positionierung der Elemente innerhalb der Perspektive verändert, in Abbildung 112 ist das neue Layout dargestellt.
Diese Anordnung hat zwei Vorteile. Zunächst einmal befinden sich die Bibliotheken ganz links oben, über den Sammlungen des Benutzers, und links vom Editor-Arbeitsbereich. Dadurch wird der Arbeitsfluss deutlicher, da Elemente aus den Bibliotheken nicht direkt verwendet werden können, sondern erst in eine Sammlung eingefügt werden müssen, und es intuitiver ist, von oben nach unten und von links nach rechts zu arbeiten, als umgekehrt. Außerdem bekommt der Editorbereich auf diese Weise mehr Platz zur Verfügung, was die Übersichtlichkeit und Anwendbarkeit erhöht. Als dritter Punkt lässt sich anführen, dass die Perspektive sich so den anderen Perspektiven annähert, da sich die Sammlungen den Nutzers (und damit auch sein Profil) nun in jeder Perspektive auf der linken Seite des Fensters befindet, was wiederum der Übersichtlichkeit zugute kommt.

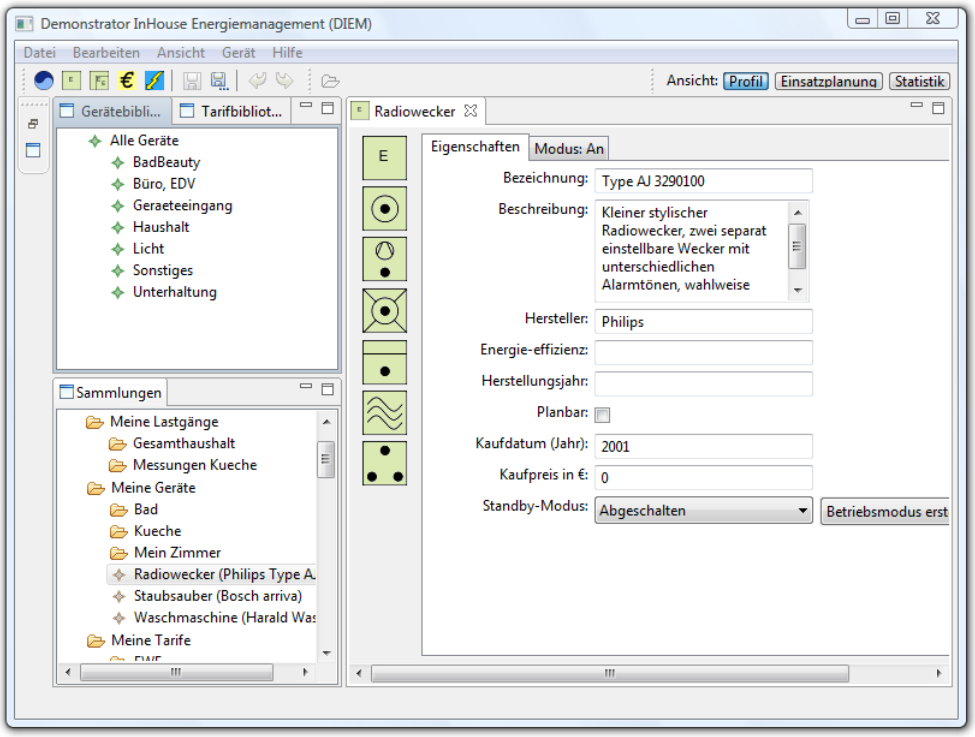

Abbildung 112: Die geänderte Profilperspektive

Die Änderungen im Code betreffen dabei einerseits die Klasse `ProfilePerspective` im Plugin `diem.app`, in welcher die Positionierung der Elemente angepasst wird. Andererseits werden die Klassen `DeviceCollectionView`, `LoadCurveCollectionView` und `PayscaleCollectionView` aus den Plugins `diem.device`, `diem.loadcurve` und `diem.payscale` (vgl. entsprechende Abschnitte des Kapitels "'Plugins"' im Entwurfsdokument) zu einer einzelnen Klasse vereinigt, welche dann die gemeinsame Sammlung repräsentiert.

27.3. Visuelle Statistik

Autor: Mark Eilers

Im Zuge der Entwicklung des zweiten Prototyps wird die Projektgruppe DIEM die visuelle Statistik implementieren. Da der Vergleich von Tarifen, Lastgängen und Einsatzplänen zu den Grundideen und Anforderungen des Demonstrator Inhouse-Energiemanagement gehört, bekommt die visuelle Statistik eine eigene Perspektive. Wie im Entwurf zum ersten Prototypen beschrieben, sind alle Komponenten der Statistik-Perspektive im Plugin *diem.Shared* angesiedelt. Die visuelle Statistik besteht, wie bereits in

der Anforderungsdefinition angedeutet, aus dem auch in beiden anderen Perspektiven verfügbarem Profil-View und zwei Statistik-Views (vergl. Abbildung 113).
Die Projektgruppe hat sich dagegen entschieden dem Nutzer vorgefertigte Vergleiche zu präsentieren. Stattdessen kann der Nutzer alle Funktionen, bzw. Einsatzpläne vom Profil, per Drag & Drop, in die Statistik-Views ziehen und so alle Vergleiche anstellen, die ihm interessant erscheinen. Als Funktionen werden in diesem Zusammenhang die grafischen Repräsentationen der Lastkurven bzw. Tarife in Form von Treppenfunktionen angesehen.
Die Aufteilung in zwei Statistik-Views gibt dem Nutzer die Möglichkeit, Lastgänge, Tarife und Einsatzpläne sowohl übereinandergelegt in einem View, als auch untereinander in seperaten Views zu vergleichen. Eine größere, bzw. variable Anzahl von Views ist aus Gründen der Übersichtlichkeit zur Zeit nicht geplant, allerdings könnte eine beliebige Anzahl von Statistik-Views über Reiter angeordnet werden, wie es mit anderen Views und Editoren bereits geschieht.

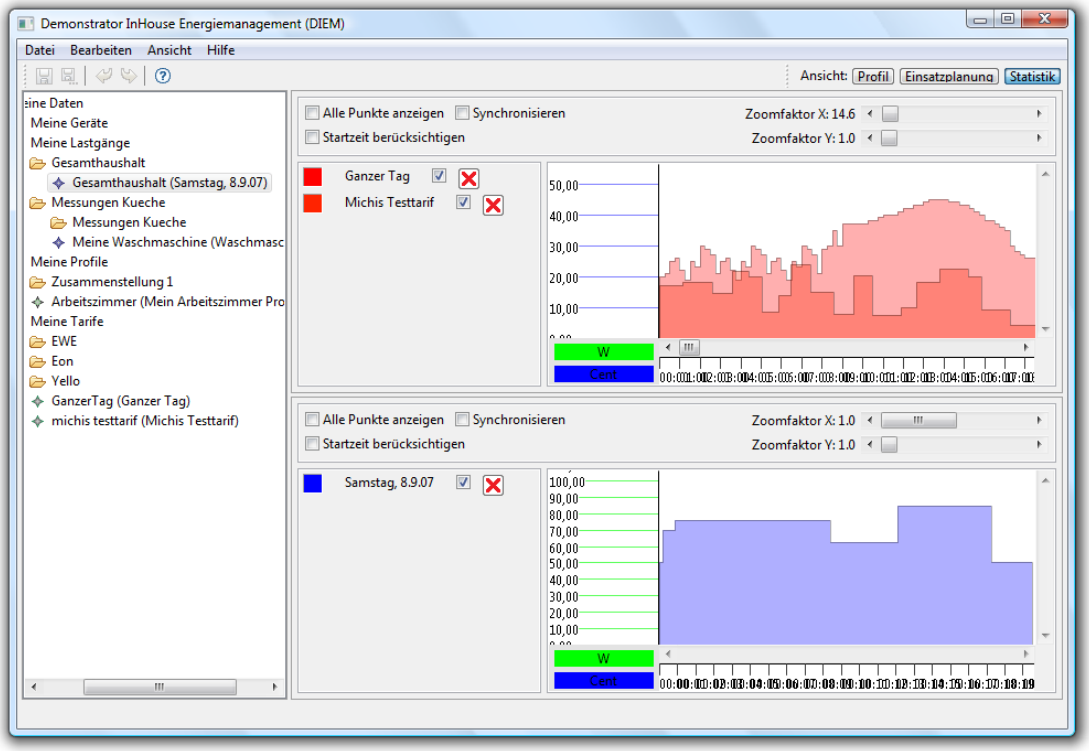

Abbildung 113: Die Statistik-Perspektive

27.3.1. View

- Der Statistik-View `[StatisticsView]`

Der View "'Statistik"' enthält als einzige Komponente einen sogenannten *FunctionViewer*. Ein FunctionViewer ist ein Composite, bestehend aus einer Optionsleiste, einer Legende, und einer Funktionsanzeige mit X-Achse und Y-Achse. Dem FunctionViewer können Lastgänge und Tarife übergeben werden, sie werden dann in der Funktionsanzeige in Form von Treppenfunktionen angezeigt. Der Nutzer kann dem View per Drag & Drop Tarife, Lastgänge, Betriebsmodi und auch Einsatzpläne übergeben, die dann von dem Funktions-Viewer angezeigt werden. Im Falle eines übergebenen Betriebsmodus aus einem Gerät wird, falls vorhanden, der entsprechende Lastgang bzw. Durchschnittsverbrauch angezeigt, im Falle eines ganzen Einsatzplans, alle Funktionen, die auch im Einsatzplan-Editor angezeigt werden. Geplant sind dabei bislang Tarife, Lastgänge, Stromkosten und Grundlast.

27.3.2. Aktions

- Actions des Statistik-Views
  - **Funktion hinzufügen** `[AddFunction]`
    Diese Action übergibt einen Lastgang oder Tarif an den *FunctionViewer*, der die betreffende Funktion visualisiert.
  - **Funktion entfernen** `[RemoveFunction]`
    Diese Action entfernt einen Lastgang oder Tarif aus dem *FunctionViewer*.
  - **Betriebsmodus hinzufügen** `[AddOperationalMode]`
    Diese Action übergibt den Lastgang eines Betriebsmodus an den *FunctionViewer*, der die betreffende Funktion visualisiert.
  - **Einsatzplan hinzufügen** `[AddSchedule]`
    Diese Action übergibt alle verfügbaren Funktionen eines Einsatzplans an den *FunctionViewer*, der die betreffenden Funktionen visualisiert.
  - **Einsatzplan entfernen** `[RemoveSchedule]`
    Diese Action entfernt alle betrefenden Funktionen eines Einsatzplans aus dem *FunctionViewer*.
  - **Funktion aus-/einblenden** `[ToggleFunctionVisibility]`
    Diese Action teilt dem *FunctionViewer* mit, die entsprechende Funktion auszublenden, falls sie sichtbar, bzw. einzublenden, falls sie ausgeblendet ist.
  - **Punkte aus-/einblenden** `[TogglePointVisibility]`
    Diese Action teilt dem *FunctionViewer* mit, alle Einzelpunkte der Funktionen auszublenden, falls sie sichtbar, bzw. einzublenden, falls sie ausgeblendet sind.
  - **Startzeiten beachten** `[ToggleStartTime]`
    Diese Action teilt dem *FunctionViewer* mit, alle Funktionen entsprechend ihrer Startzeiten anzuordnen, bzw. alle Funktionen auf den gleichen Startzeit zu legen.

- **Horizontaler Zoom** `[ZoomXAxis]`
  Ändert die dargestellten Intervalle der X-Achse und skaliert die Funktionen im *FunctionViewer*.
- **Vertikaler Zoom** `[ZoomYAxis]`
  Ändert die dargestellten Intervalle der Y-Achse und skaliert die Funktionen im *FunctionViewer*.

27.4. Automatische Einsatzplanoptimierung

Autor: Michael Erlmann

Im Folgenden wird die Herangehensweise an die Umsetzung der automatischen Einsatzplanung konkretisiert. Eine generelle Betrachtung der Problemstellung und erste Lösungsansätze finden sich bereit im Entwurf in Kapitel 2.6.
Entgegen der dort gestellten Analyse soll jedoch nicht eine Konstruktionsheuristik entwickelt werden, um den Einsatzplan zu optimieren. Statt dessen soll ein Constraint-Solver eingesetzt werden, da erwartet wird, dass der Aufwand für die Modellierung des Problems und den Einsatz eines solchen Solvers geringer ist, als eine Konstruktionsheuristik für das Problem zu entwerfen und zu optimieren.
Das Problem der automatischen Einsatzplanoptimierung lässt sich als so genanntes Constraint-Satisfaction-Problem (CSP) modellieren. Es beschreibt eine Struktur, auf der algorithmisch durch einen so genannten Constraint-Solver ein Einsatzplan erzeugt werden kann. Hierbei gibt es Variablen, Domänen und Constraints. Die Variablen beschreiben generell den Startzeitpunkt eines so genannten Jobs, und die Domänen sind die zeitlichen Intervalle, in welcher dieser Job starten kann. Constraints beschreiben Beziehungen zwischen einzelnen Variablen, welche beim Lösen des CSP nicht verletzt werden dürfen. Dabei übersetzt sich die gegebene Problemstellung durch den Einsatzplan in der Struktur wie er in DIEM vorliegt folgendermaßen:

- Jedem Einsatz wird eine Variable im CSP zugeordnet. Sie entspricht dem Startzeitpunkt dieses Einsatzes.
- Jeder dieser Variablen wird dann eine Domäne zugeordnet, welche dem zeitlichen Abschnitt entspricht, in dem der Einsatz gestartet werden darf, begrenzt durch seine Einsatzzeiten.
- Für alle vorhandenen Beziehungen zwischen den Einsätzen werden Constraints im CSP angelegt, welche diese Beziehungen darstellen.
- Zur Vereinfachung werden zusätzlich Variablen angelegt, um die Endzeitpunkte der Einsätze darzustellen. Sie erhalten den Constraint, dass sie genau soviel Zeit später als der Startzeitpunkt des Einsatzes liegen müssen, wie die Dauer des Einsatzes beträgt. Hierdurch wird die Realisierung der Constraints erheblich vereinfacht. Die Domäne dieser Variablen entspricht derjenigen der Variablen der Startzeiten eines Einsatzes, addiert mit dem Betrag seiner Dauer.

Auf die folgende Weise werden dabei die einzelnen Constraints abgebildet:
Seien x und y zwei Einsätze. Dann sind $V_s(x)$ und $V_e(x)$ die Variablen über Start- und Endzeitpunkt von x und $V_s(y)$ und $V_e(y)$ die Variablen über Start- und Endzeitpunkt von y. Die Constraints werden dann folgendermaßen modelliert:

SameStart $V_s(x) = V_s(y)$

SameEnd $V_e(x) = V_e(y)$

Prohibit $V_e(x) < V_s(y)\ or\ V_s(x) > V_e(y)$

AfterEachOther $V_e(x) + duration < V_s(y)$, wobei $duration$ die Dauer beschreibt, welche zwischen dem Ende von x und dem Start von y mindestens vergehen muss

Identical $V_s(x) \leq V_s(y)\ and\ V_e(x) \geq V_e(y)$, wobei x der längere Einsatz ist, falls nicht beide Einsätze gleich lang sind

Zu Beachten ist, dass der Identical-Constraint anders modelliert ist als im bisherigen Design. Auf diese Weise werden mehr Variationen zur Kombination zweier Einsätze zugelassen. Weiterhin ist die vorherige Variante, bei der zwei Einsätze gleich lang sind, immernoch möglich, da $\leq$ und $\geq$ als Operatoren gewählt wurden, statt $<$ und $>$.
Sofern im Einsatzplan Constraints gesetzt werden, welche sich gegenseitig ausschließen, wie zum Beispiel Prohibit und SameEnd zwischen zwei Aufträgen, so wird dies dem eingesetzen Constraint-Solver überlassen, diese fest zu stellen, da jeder Solver einen solchen Test bereits intern durchführt und ein separates Überprüfen nur die Laufzeit erhöhen würde.
Wie beschrieben ist die Umsetzung des vorliegenden Problems der automatischen Einsatzplanung in ein CSP ohne große Umstände erfolgt, da die beiden Ansätze bereits sehr nah beieinander liegen. Dadurch ergibt sich der Vorteil, dass nicht erst eine Konstruktionsheuristik für die Problemstellung erzeugt und optimiert werden muss. Statt dessen kann ein bereits existierender frei verfügbarer Constraint-Solver herangezogen und eingebunden werden. Dabei fiel die Wahl auf Choco. Die Projektseite zu Choco inklusive Dokumentation ist zu finden unter `http://choco-solver.net` [11].
Alternativ zu Choco wäre auch noch die Verwendung des Constraint-Solvers Cream denkbar gewesen, welcher unter `http://bach.istc.kobe-u.ac.jp/cream/` [12] zu finden ist. Im Gegensatz zu Choco wurden mit Cream im Vorfeld der Projektgruppe schon Erfahrungen gesammelt, mit Choco wurden wir erst während des Projektes bekannt gemacht. Dennoch wurde auf die Verwendung von Cream zugunsten von Choco aus folgenden Gründen verzichtet:

- Die Dokumentation zu Cream ist sehr knapp und größtenteils auf Japanisch, im Gegensatz zu Choco, wo eine wesentlich umfassendere Dokumentation existiert.

[11] Zugriff am 12.01.2008
[12] Zugriff am 12.01.2008

- Cream löst viele Modellierungsformen des CSP sehr unintuitiv und schwer nachvollziehbar. Choco ist in dieser Angelegenheit sehr geradlinig.
- Anke Lederer benutzte Choco in ihrer Diplomarbeit 'Integration von Constraints und Verfahren zum Online-Scheduling in Optimierungsheuristiken zur Lastadaption' und kann deshalb bei Schwierigkeiten unterstützen.

Eine genauere Betrachtung weiterer Constraint-Solver wurde nicht vorgenommen, da wir bei einem Treffen mit Anke Lederer einen kurzen Überblick über weitere Solver erhalten haben, welche sie jedoch als für unser Problem zu überdimensioniert empfand.
Im Folgenden wird die Umsetzung des vorliegenden Einsatzplanes in die CSP-Struktur von Choco betrachtet. Zentrales Objekt ist das Problem. An diesem werden alle Variablen, Domänen und Constraints angemeldet und anschließend der Lösungsalgorithmus gestartet. Dabei werden Variablen sofort mit der dazugehörigen Domäne im Problem angemeldet. Da wir bei der Modellierung entschieden haben, für jeden Einsatz zwei Variablen anzulegen, müssen wir bereits jetzt die entsprechenden Domänen errechnen. Für die Variable, welche den Startzeitpunkt des Einsatzes darstellt, gilt das Intervall zwischen frühester Startzeit und spätester Endzeit minus Dauer des Einsatzes. Für die Endvariable gilt das Intervall zwischen frühester Startzeit des Einsatzes plus seine Dauer und späteste Endzeit. Zusätzlich muss noch der Constraint erzeugt werden, dass die Endvariable genau der Dauer des Einsatzes später liegen muss als die Startvariable.
Zur Umsetzung der Constraints bietet Choco folgende grundlegende mathematische Funktionen an:

- `neq(IntExp v1, IntExp v2)  : v1 != v2`
- `eq(IntExp v1, IntExp v2)  : v1 = v2`
- `leq(IntExp v1, IntExp v2) : v1 <= v2`
- `lt(IntExp v1, IntExp v2) : v1 < v2`
- `minus(IntExp exp1, IntExp exp2) : exp1 - exp2`
- `plus(IntExp exp1, IntExp exp2) : exp1 + exp2`
- `and(Constraint c1, Constraint c2)  : c1 and c2`
- `or(Constraint c1, Constraint c2)  : c1 or c2`

Mit diesen ist es möglich alle Constraints zu übersetzen, bis auf den MaxLoad-Constraint. Dieser wird wegen der im folgenden betrachteten Schwierigkeiten aber extern betrachtet.
Zusätzlich dazu treten bei der Lösungsfindung durch Choco jedoch noch weitere Probleme auf. Zum einen kann Choco nur mit Integer-Werten arbeiten, die Tarife und die

Lastgänge der Einsätze liegen jedoch, da es sich bei ihnen um Geldbeträge und Leistungsaufnahmen mit Nachkommastellen handelt, als Float-Werte vor und können daher nicht direkt in Choco eingegeben werden. Mögliche Lösungen wären das Runden der Werte, was jedoch zu einer erheblichen Ungenauigkeit führen würde, sowie das Multiplizieren mit einer hohen Zehnerpotenz und das anschließende Dividieren durch diese, wodurch man die Nachkommastellen erhalten würde. Eine weitere Schwierigkeit bereitet die Modellierung der Bewertungsfunktion in Choco selbst, bei der intern durch Choco der Preis für eine generierte Lösung berechnet werden müsste. Hierbei muss in Choco für die Tarifkurve über die Dauer des Einsatzplanes für jede Minute eine Variable über den jeweiligen Preis zu dem Zeitpunkt angelegt werden, sowie eine Variable für den durchschnittlichen Verbrauch eines jeden Einsatzes. Weiter muss Choco beigebracht werden, diese Werte miteinander zu kombinieren und aus der resultierenden Kurve das Integral zu bilden.Dies kann bei längeren Einsatzplänen zu einer sehr großen Anzahl an Objekten und somit zu Speicherkomplexitätsproblemen führen würde. Statt dessen wird der Ansatz gewählt, welchen auch Anke Lederer in ihrer Diplomarbeit verwendet hat:

Schritt 1: *Erzeuge initiale Lösung S1 mittels Choco*
Setze S0 = S1
Ermittel G(S0) außerhalb von Choco

Schritt 2: *Erzeuge neue gültige Lösung Sc mittles Choco*
Ermittel G(Sc) außerhalb von Choco
Wenn G(Sc) < G(S0), dann setze S0 = Sc

Schritt 3: *Wenn die maximale Zeit oder die letzte mögliche*
Lösung noch nicht erreicht ist, gehe zu Schritt 2
Ansonsten gib beste gefundene Lösung S0 zurück

Der Vorteil dieses Lösungsansatzes ist, dass damit sowohl die Schwierigkeiten mit den Float-Werten umgangen werden, als auch der MaxLoad-Constraint behandelt werden kann, da sie außerhalb von Choco benutzt werden. Daher wird dieser Ansatz umgesetzt.

27.5. Dokumentation

Autor: Moritz Eysholdt

Für den Anwender wird es sowohl ein Handbuch als auch Tutorials geben. Zunächst wird die technische Umsetzung der Dokumentation beschrieben. Anforderung ist, diese sowohl im Programm zur Verfügung zu stellen als auch als PDF-Datei.

27.5.1. Technische Umsetzung

- Für die PDF-Erstellung soll Latex verwendet werden.
- Für die Dokumentation in der Anwendung selbst soll das Hilfesystem von Eclipse verwendet werden. Es besteht aus einem kleinen eingebetteten Webserver, welcher die Dokumentation in Form von HTML ausliefert. Beim Aufrufen der Hilfe wird der Webbrowser des Anwenders gestartet und er kann in der Hilfe suchen oder im hierarchisch strukturierten Inhaltsverzeichnis zum gesuchten Hilfethema navigieren.

Und die Dokumentation nicht an zwei verschiedenen Orten separat pflegen zu müssen, soll selbige im Docbook-Format (www.docbook.org) geschrieben werden. Docbook ist ein auf XML basierendes Dateiformat, welcher zum Verfassen von Dokumentationen verwendet wird. Es gibt eine Vielzahl an XSL-Transformationen, mit denen sich aus Docbook-Dateien PDFs, Latex, Manpages, HTML, Eclipse Hilfen etc. generieren lassen. In diesem Projekt soll aus dem Docbook Latex und die Eclipse Hilfe generiert werden.

27.5.2. Handbuch

Autor: Moritz Eysholdt und Michael Belasus

Das Handbuch soll die Funktionen der Anwendung kurz erläutern. Zusätzlich werden mithilfe von Screenshots alle Menueinträge, Views und Editoren abgebildet und erklärt. Wie die Funktionen sinnvoll kombiniert werden können veranschaulichen die Tutorials.

27.5.3. Tutorial

Autor: Michael Belasus

Das Tutorial zeigt anhand eines komplexen Beispiels wie der Anwender den Einsatz für den kommenden Tag planen kann. Dabei werden Daten, die tatsächlich in den beiden Bibliotheken vorkommen, verwendet.

Folgendes Szenario wird als Ablauf für den nächsten Tag geplant.

- Wäsche waschen und trocknen
- Geschirr spülen
- Staub saugen

- eine DVD anschauen

Innerhalb dieses Szenarios wird ein variabler Tarif verwendet. Derzeit existieren keine variablen Tarife, daher wird dieser Tarif ein ausgedachtes Beispiel sein.
Für das genannte Szenario wird von folgenden Voraussetzung ausgegangen. **Voraussetzung**

- Software installiert
- Geräte- und Tarifsammlung leer
- Geräte- und Tarifbibliothek von der Installation

Im Anschluss werden die folgenden Abläufe näher beschrieben. **Update- und Uploadfunktion Client**

- Gerätebibliothek herunterladen
- Gerät aus der Gerätebibliothek in die Sammlung übernehmen
- neues Gerät erstellen
- dieses Gerät auf den Server laden
- Tarifbibliothek herunterladen
- Tarif aus der Tarifbibliothek in die Tarifsammlung übernehmen

Geräte-, Tarif- und Lastgangeigenschaften anzeigen und bearbeiten

- Gerät laden, ändern, löschen
- Betriebsmodus öffnen, hinzufügen, löschen
- Lastgang anzeigen, importieren, hinzufügen
- neue Gerätekomposition anlegen
- variablen Tarif laden, ändern, löschen
- aus variablem Tarif festen Tarif erstellen
- festen Tarif laden, ändern, löschen
- aus festem Tarif variablen Tarif erstellen
- neuen Tarif erstellen

Einsatzplanung

- Einsatz planen

- Constraints setzen
  - nacheinander
  - gleichzeitig
  - maximale Last

Statistik

- Statistiken erstellen
  - Tarife
  - Lastgänge

27.6. Server

Autor: Christian Bley

Im zweiten Prototypen werden alle Anforderungen an die Administration der Online-Bibliotheken im Server umgesetzt. Während der Entwicklung und Tests zeigte sich, dass weitere Funktionen die Nutzbarkeit des Servers erhöhen bzw. vereinfachen. Hieraus ergeben sich die folgenden, zusätzlich zu den in der Anforderungsdefinition definierten Anwendungsfälle:

- Geräte innerhalb der Gerätebibliothek verschieben.
- Tarife innerhalb der Tarifbibliothek verschieben.
- Ordner erstellen
- Ordner umbenennen
- Ordner löschen
- Ordner verschieben

Der Administrator kann unter Verwendung des Clients Geräte und Tarife erstellen. Bis auf diese zwei Funktionen sind alle administrativen Aufgaben nur über den zugriffsbeschränkten Server per Browser durchführbar. Während die Geräte wie auf herkömmlichen Weg über den Client in die Bibliothek geladen werden können, besteht über den Browser die Möglichkeit, eine Geräte- oder Tarifdatei sofort in einem gewünschten Ordner zu speichern.
Die Ordner können beliebig erstellt und umbenannt werden, vorausgesetzt der Name enthält nur erlaubte Zeichen, wie in Tabelle 6 dargestellt. Es können auch ganze Ordner inklusive Inhalt verschoben werden.

Whitespace Zeichen	- `[\t\n\x0B\f\r]`
Ziffern	- `[0-9]`
Wortzeichen	- `[a-Z_A-Z]`
Umlaute	- `[äüöÄÜÖ]`
Pluszeichen	- `[+]`
Bindestrich / Minus	- `[-]`
Komma / Semikolon	- `[,;]`

Tabelle 6: Erlaubte Zeichen in Ordnernamen

Von anderen Benutzern hochgeladene Gerätedaten können auf ihren korrekten Inhalt geprüft und verifiziert werden. Der Administrator kann dann diese in die entsprechenden Ordner verschieben, um eine sinnvolle Anordnung innerhalb der Bibliothek zu erreichen. Zu diesem Zweck können auch alle Elemente der Bibliotheken gelöscht werden. Da dieser Vorgang endgültig ist, wird vorher eine Sicherheitsabfrage gestellt. Bestätigt der Administrator diese, dann wird das gewünschte Element, bei Ordnern auch der Inhalt endgültig gelöscht.

27.6.1. Management

Dieser Abschnitt ersetzt den gleichnamigen Abschnitt im Entwurfsdokument.
Die Managementfunktionen des Servers werden über das `ManagerServlet` angestoßen. Gesteuert werden diese Vorgänge über die Parameter `action` und `scope`, wobei `action` die gewünschte Aktion repräsentiert und `scope` den gewünschten Kontext.
Der Parameter `scope` darf folgende Werte annehmen, die den gewünschten Kontext festlegen:

DEVICE Geräte innerhalb der Gerätebibliothek.

DEVICEFOLDER Ordner innerhalb der Gerätebibliothek.

PAYSCALE Tarife innerhalb der Tarifbibliohtek.

PAYSCALEFOLDER Ordner innerhalb der Tarifbibliohtek.

Je nach Kontext sind nur bestimmte `action`-Werte erlaubt und je nach `action` sind zusätzliche Parameter notwendig.
Es folgt die Beschreibung der möglichen Aktionen durch die `action`-Parameter. Desweiteren wird definiert, für welchen Kontext eine Aktion gültig ist und welche weiteren Parameter hierfür notwendig sind. Die Visualisierung geschieht über JSPs.

LIST Es wird der Inhalt des Ordners mit der als `ID` angegebenen Id angezeigt. Ist keine Id angegeben, wird der Inhalt des Wurzelordners der jeweiligen Bibliothek aufgelistet.

Gültiger Kontext: `DEVICE, DEVICEFOLDER, PAYSCALE, PAYSCALEFOLDER`.
Optionaler Parameter: `ID`.
Das Ergebnis wird über `devicedialog.jsp`, bzw. `payscaledialog.jsp` dargestellt.

SHOW Es werden die Schlüsselattribute mit der als `ID` angegebenen Id angezeigt. Ist keine Id angegeben, wird der Inhalt des Wurzelordners der jeweiligen Bibliothek aufgelistet.
Gültiger Kontext: `DEVICE, PAYSCALE`.
Benötigter Parameter: `ID`.
Das Ergebnis wird über `showdevice.jsp`, bzw. `showpayscale.jsp` dargestellt.

CUT Das mit `ID` referenzierte Element wird in einer Liste an der Browsersitzung gespeichert. Pro Kontext gibt es eine eigene Liste. Je nach Kontext werden beim Auflisten diese Elemente angezeigt, so dass der Administrator diese Elemente an anderen Stellen einfügen kann.
Gültiger Kontext: `DEVICE, DEVICEFOLDER, PAYSCALE, PAYSCALEFOLDER`.
Benötigter Parameter: `ID`.
Das Ergebnis wird über `devicedialog.jsp` bzw. `payscaledialog.jsp` dargestellt.

PASTE Das mit `ID` referenzierte Element wird in den Ordner mit der in `TO` übergebenen Id verschoben. Es ist nicht möglich einen Ordner in seinen eigenen Unterordner zu kopieren. Sollte dieser Fall eintreten, wird kein Fehler ausgegeben, sondern der Administrator per Mitteilung im Auflistungsdialog darauf aufmerksam gemacht.
Gültiger Kontext: `DEVICE, DEVICEFOLDER, PAYSCALE, PAYSCALEFOLDER`.
Benötigte Parameter: `ID, TO`.
Das Ergebnis wird über `devicedialog.jsp` bzw. `payscaledialog.jsp` dargestellt.

DELETE Das mit `ID` referenzierte Element wird aus der Bibliothek gelöscht. Handelt es sich um einen Ordner werden alle Inhalte dieses Ordners rekursiv mitgelöscht.
Gültiger Kontext: `DEVICE, DEVICEFOLDER, PAYSCALE, PAYSCALEFOLDER`.
Benötigter Parameter: `ID`.
Das Ergebnis wird über `devicedialog.jsp` bzw. `payscaledialog.jsp` dargestellt.

VERIFY Das Feld `verified`, des mit `ID` referenzierten Gerätes wird invertiert. Wenn das Gerät vor der Aktion verifiziert war, ist es nach der Aktion nicht verifiziert und umgekehrt.
Gültiger Kontext: `DEVICE`.
Benötigter Parameter: `ID`.
Das Ergebnis wird über `showdevice.jsp` dargestellt.

RENAME Der Name des mit `ID` referenzierten Ordners wird mit dem Parameter `NAME` überschrieben, falls dieser Name nicht gegen die Konventionen der Ordnerbenennung verstößt.
Gültiger Kontext: `DEVICEFOLDER, PAYSCALEFOLDER`.
Benötigte Parameter: `ID,NAME`.
Das Ergebnis wird über `devicedialog.jsp` bzw. `payscaledialog.jsp` dargestellt.

CREATE Erzeugt einen neuen Ordner. Der Parameter `TO` referenziert den Überordner des neuen Ordners. Wird er nicht angegeben wird der Ordner im Wurzelordner der entsprechenden Bibliothek erzeugt. Der Parameter `NAME` stellt den Namen des neuen Ordners dar. Wird dieser nicht angegeben, so bekommt der neue Ordner einen Standardnamen.
Gültiger Kontext: `DEVICEFOLDER, PAYSCALEFOLDER`.
Optionale Parameter: `TO, NAME`.
Das Ergebnis wird über `devicedialog.jsp`, bzw. `payscaledialog.jsp` dargestellt.

Sollten bei diesen Aktionen Fehler auftreten, wie fehlende Parameter oder Exceptions aus anderen Gründen, so wird eine vom Container generierte Fehlerseite mit einer Fehlerbeschreibung angezeigt. Für dieses Fehlermanagement wird eine spezielle `OperationException` verwendet, die einen Fehlercode und eine Fehlernachricht enthält. Diese Exception kapselt alle anderen Exceptions, die während des Servletaufrufs auftreten. Wenn diese `OperationException` abgefangen wird, sendet das Servlet die Fehlerseite mit der Fehlernachricht als Fehlerbeschreibung.

28. Prototyp 3

28.1. Änderungen und Erweiterungen zum Entwurf durch Prototyp 3

Im Folgenden werden alle die Bereiche, die in der Umsetzung des dritten Prototypen behandelt werden, oder bei denen es eine Änderung bzw. Ergänzung zum Entwurf gibt, vorgestellt.

28.2. Numerische Statistik

Autor: Mark Eilers

Zusätzlich zu der im zweiten Prototyp implementierten visuellen Statistik, wurde der Wunsch nach einer "'numerischen"' Statistik geäußert, also der Möglichkeit Funktionen (als Funktionen gelten dabei die grafische Darstellung von Tarifen und Lastgängen in Form von Treppenfunktionen) nicht allein visuell, sondern auch nach ihren mathematischen Eigenschaften zu vergleichen. Dem wird im Zuge der Entwicklung des dritten Prototyps Genüge getan.
Die bisher rein visuelle Statistik wird zu diesem Zweck in drei Teile unterteilt, nämlich "'Visueller Vergleich"', "'Direkter Vergleich"' sowie "'Daten und Fakten"', die im Statistik-View über Reiter verfügbar sind(siehe Abb. 114). Der Anwender kann, wie schon im zweiten Prototyp implementiert, Tarife, Lastgänge, Betriebsmodi, Geräte mit Betriebsmodi und Einsatzpläne per Drag&Drop in die Funktionsanzeige ziehen und über die Reiter wählen, ob er die Funktionen grafisch oder über ihre Eigenschaften vergleichen möchte. Innerhalb der neuen Ansicht werden alle Funktionen zeilenweise in tabellarischer Weise dargestellt.
Die neue Seite "'Direkter Vergleich"' ermöglicht dem Anwender über Dropdown-Boxen jeweils einen Tarif und einen Lastgang auszuwählen, falls diese in den Statistik-View gezogen wurden. Ist ein Tarif und ein Lastgang ausgewählt, werden daraus die Gesamtkosten berechnet. Dies ermöglicht dem Anwender seine Stromkosten bei verschiedene Tarifen zu vergleichen.
Innerhalb der neuen Seite "'Daten und Fakten"' werden alle Funktionen zeilenweise in tabellarischer Form dargestellt. Dabei werden dem Nutzer folgende Eigenschaften angezeigt:

- Name
- Typ ("'Tarif"' oder "'Lastgang"')
- Maximum
- Minimum

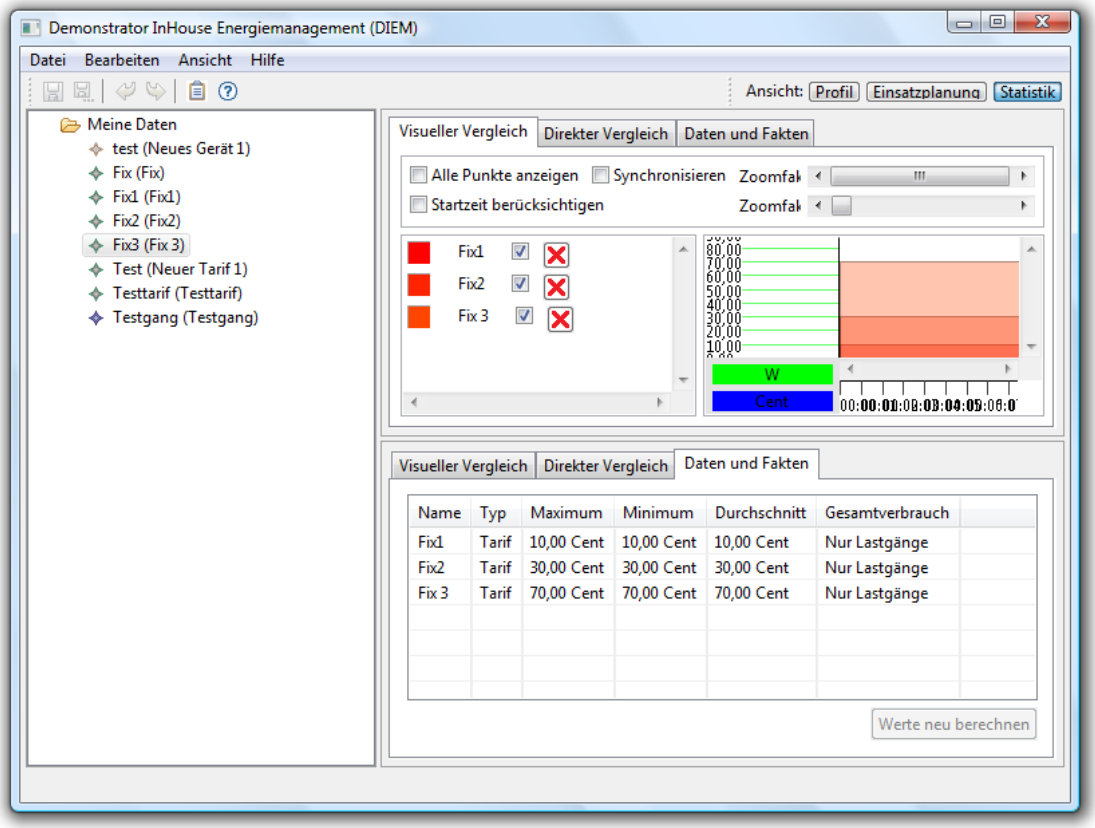

Abbildung 114: Die erweiterte Statistik-Perspektive

- Durchschnitt
- Gesamtverbrauch (nur für Lastgänge verfügbar)

Unterhalb der Tabelle wird ein Button implementiert, der aktiviert wird, falls sich Änderungen im Model ereignen. Dies hat zwar den Nachteil, dass der Anwender die Berechnungen durch Knopfdruck aktualisieren muss, dem gegenüber steht jedoch eine erhebliche bringt Steigerungen der Performance, gegenüber einer vollautomatischen Aktualisierung. Namensänderungen werden jedoch weiterhin vollautomatisch aktualisiert.
Die, für die Berechnungen der Eigenschaften Maximum, Minimum, Durchschnitt, Gesamtverbrauch und Gesamtkosten benötigten Algorithmen werden im Plugin *diem.Logic*, innerhalb der Klasse *StatistcFunctions*, implementiert und sind dadurch über die *LogicFacade* verfügbar. Die ermöglicht eine einfache Erweiterbarkeit der Statistik um weitere Berechnungen.

28.3. Serien

Autor: Ulrich Denecke

Um sich wiederholende Geräteeinsätze komfortabel verwalten zu können, soll im dritten Prototypen eine Zyklusfunktionalität implementiert werden. Die Gruppe hat sich entschieden, sich vorerst von dem Begriff "'Zyklus"' zu entfernen und anfangs eine erweiterbare Serienfunktionalität zur Verfügung zu stellen. Die Serienfunktionalität soll folgende Eigenschaften erreichen:

- Anlegen einer Serie
- Erkennen der Tasks einer Serie
- Möglichkeiten zum Bearbeiten einzelner Tasks
- Möglichkeiten zum Bearbeiten der kompletten Serie

Das Anlegen einer Serie soll entweder über mehrfaches Hinzufügen eines Tasks zum Einsatzplan oder durch Multiplizieren eines bereits im Einsatzplan vorhandenen Tasks umgesetzt werden. Wird der gleiche Betriebsmodus eines Gerätes mehrfach zu einem Einsatzplan hinzugefügt, so werden diese nicht mehr untereinander angeordnet wie bisher, sondern in einer Zeile. Alle Tasks in einer Zeile, also identische Betriebsmodi eines Gerätes, werden als eine Serie diese Betriebsmodus betrachtet. Die Datenstruktur muss also nicht geändert werden. Die Bearbeitung einzelner Tasks innerhalb einer Serie soll wie gewohnt weiter möglich sein, allerdings soll zusätzlich dazu per Button die ganze Serie zur Bearbeitung ausgewählt werden können, um Daten wie zum Beispiel Abstand zwischen den einzelnen Tasks, Länge der Tasks und ähnliches auch für alle Tasks der Serie gleichzeitig ändern zu können.
Je nach verbleibender Zeit sind weitere Annährungen an die vorherig angedachte Zyklusdefinition denkbar. Die Entscheidungsgrundlage hierfür bestand darin, dass die so implementierten Serien erweiterbar bleiben und für den Benutzer einfach zu nutzen sind. Die Auswahl einer kompletten Serie per Button stellt eine intuitive Möglichkeit dar, die einfach zu verstehen ist. Die Anforderung, dass sich ein Einsatz zur Laufzeit des Einsatzplanes wiederholt entspricht dem Entwurf. Jedoch haben sich durch die oben genannten Entwurfsentscheidungen einige Aspekte der Verwaltung der Serien geändert.Constraints können nun entgegen des Enturfs auch einzelnen Einsätzen einer Serie zugeordnet werden. Weiterhin werden die vorgesehenen Attribute "'Betriebsdauer"' und "'Wartezeit"' nicht mehr benötigt, da jeder Einsatz einer Serie seine eigene Startzeit und Dauer enthält.

28.4. Gerätesubstitution

Autor: Moritz Eysholdt

Die zentrale Aufgabe dieser Software ist es, die Stromkosten des Anwenders so gering wie möglich zu halten. Dazu kann der Anwender seinen Stromverbrauch auf Zeitpunkte legen, wo dieser besonders kostengünstig ist. Die nachhaltigere Lösung ist jedoch, Verbraucher einzusetzen, welche von sich aus einen geringeren Stromverbrauch haben. Die Gerätesubstitution ist als Hilfe für den Anwender gedacht, seine aktuellen Verbraucher durch welche mit geringerm Stromverbrauch aber dem gleichen Leistungsvermögen zu ersetzen.
Hat der Anwender einen Einsatzplan geöffnet, soll die Funktion der Gerätesubstitution verfügbar sein. Diese soll ihm alle Geräte aus dem offenen Einsatzplan auflisten und für jedes eine Liste mit möglichen Alternativen einblenden. Zu jeder auswählbaren Alternative soll das Einsparpotential bzgl. des Verbrauches angezeigt werden. Der Anwender soll beliebige Geräte als Substitution auswählen können, die vorgeschlagenen jedoch sollen möglichst die gleiche Funktionalität wie das zu substituierende Gerät besitzen.
Damit die Software beurteilen kann, weche Geräte die gleiche Funktionalität besitzen, sollen diese Geräteklassen zugeordnet werden. Beispiele: Eine Mikrowelle die auch eine Grillfunktion hat, wäre damit in den Geräteklassen “Mikrowelle” und “Grill”. Eine üblicher Kühlschrank mit Eisfach gehörte den Klassen “Kühllösung” und “Gefrierlösung” an. Die vorgeschlagene Liste der Substitutionsmöglichkeiten zu einem Geräte soll mit den Geräten beginnen, welche den selben Geräteklassen wie das zu substituierende Geräte zugeordnet sind. Des weiteren sollen Geräte mit mindestens einer übereinstimmenden Geräteklasse vorgeschlagen werden.
Geräteklassen sind wie ein Tag aus der Web 2.0 Welt zu verstehen. Es gibt keine zentrale Liste von gültigen Geräteklassen um Konsistenzprobleme zu vermeiden. Eine Geräteklasse selbst ist ein String-Wert, welcher einem Gerät zugeordnet ist. Ein Gerät wiederum kann beliebig vielen Geräteklassen zugeordnet sein. Um alle Geräteklassen einer Gerätebibliothek zu identifizieren, muss über alle Geräte iteriert und eine Liste über alle verwendeten Geräteklassen zusammengetragen werden. Die Geräteklasse ist im Datenmodell bisher nicht vorhanden, es muss darum erweitert werden.
Hat der Anwender alle zu substituierenden Geräte und ihre neuen Betriebsmodi ausgewählt, werden die alten Geräte und Betriebsmodi im Einsatzplan durch diese ersetzt.

28.5. Einsatzplan drucken

Autor: Michael Belasus

Damit ein Anwender nicht immer das Programm starten muss, um geplante Einsätze durchführen zu können, kann jeder Einsatzplan gedruckt werden. Nachdem der gewünschte Plan ausgewählt wurde, öffnet sich vor dem Ausdruck ein Dialogfenster. Innerhalb dieses Dialoges werden die geplanten Einsätze tabellarisch aufgelistet. Der Anwender kann nun auswählen, welche Angaben zu den Einsätzen gedruckt werden sollen. Folgende Angaben können dabei ausgewählt werden.

- Startzeit des Einsatzes (vorausgewählt)

- Endzeit des Einsatzes (vorausgewählt)
- Dauer des Einsatzes
- Gerätename (vorausgewählt)
- Betriebsmodus (vorausgewählt)
- Gesamtkosten des Einsatzen
- Gesamtverbrauch des Einsatzes

Die ausgewählten Angaben werden in der Tabelle farblich gekennzeichnet. Im Vorfeld sind vier bereits ausgewählt. Die Zeilen innerhalb der Tabelle müssen ebenfalls für den Ausdruck markiert werden. Dazu stehen unterhalb der Tabelle weiter Markierungsoptionen zur Verfügung. Folgende Optionen können genutzt werden: alle Zeilen, innerhalb einer beliebigen Startzeiten, definierte Start- und Endzeiten, Geräte und Betriebsmodi. Der Anwender ist aber in keiner Weise in seiner Auswahl beschränkt. Es muss lediglich eine Spalte und eine Zeile markiert sein, sonst kann der Druckvorgang nicht gestartet werden. Vor dem eigentlichen Druckvorgang öffnet sich die Druckerauswahl. Hier kann der Anwender bestimmen welcher der installierten Drucker verwendet werden soll. Dabei kann dann der Ausdruck zu einem Drucker geleitet oder in eine Datei umgeleitet werden. Die Projektgruppe hat entschieden keinen PDF Druck zu integrieren, da spätere Anwender, wenn sie drucken, tatsächlich einen Ausdruck wollen. Sofern der Einsatzplan ein PDF Dokument ist, muss dies entweder erneut gedruckt oder der PC muss eingeschaltet werden, um das Dokument lesen zu können. Dann könnte aber auch gleich das Programm gestartet werden. Durch die Option den Drucker selbst wählen zu können, besteht jederzeit die Möglichkeit ein PDF Dokument zu generieren, sofern ein entsprechendes Programm installiert wurde.

28.6. Wizards

Autor: Malte Hurrelmann

Um dem Benutzer weitere Unterstützung bei der Modellierung seines Haushaltes zu geben, sollen im dritten Prototypen Wizards implementiert werden.
Wizards werden an den Stellen der Applikation eingesetzt, an denen es für den Benutzer evtl. leicht unübersichtlich werden könnte. So z.B. in der Gerätesammlung und dem Einsatzplan. Werden hier vom Benutzer sehr viele Geräte eingefügt, so können selbst Verbraucher mit sehr hohem Verbrauch (wie z.B. einer Spülmaschine) oder eine ganze Gruppe von Verbrauchern (wie z.B. Beleuchtung) leicht in der Planung vergessen werden. Der Wizard vergleicht eine vorher definierte Liste von Haushaltsgeräten, die in einem durchschnittlichen Haushalt vorhanden sind, mit der Auswahl an Geräten

die der Benutzer aus der Gerätebibliothek in seine Gerätesammlung gezogen hat. Danach zeigt er checklistenartig an, welche Geräte der Benutzer gewählt hat und welche er im Vergleich mit dem durchschnittlichen Haushalt nicht in seiner Gerätesammlung verfügbar hat. Diese Geräte werden ihm dann als Zusatz zu seiner Gerätesammlung vorgeschlagen.
Technisch funktioniert ein solcher Abgleich mit dem, ebenfalls in Prototyp drei eingeführten "'Geräteklassen"' vergl. . Somit werden nicht die eigentlichen Geräte aus den beiden beschriebenen Listen, sondern nur deren Tags miteinander verglichen und somit eine Auswahl an fehlenden Tags und somit Geräteklassen ermittelt. Wählt der Benutzer eines der ihm vorgeschlagenen Geräte als Zusatz für seine Gerätesammlung oder Einsatzplan aus, so wird das entsprechende Gerät automatisch hizugefügt. Der Wizard läßt sich von überall starten und sich bei Bedarf auch in der Ausführung Beenden.

28.7. Server

Autor: Christian Bley

Eine zentrale Anforderung des Projektes ist die Unterstützung von variablen Tarifen, in erster Linie Day-Ahead-Tarife. Im dritten Prototypen werden Funktionen zum Server hinzugefügt, die das Konfigurieren und Importieren von diesen variablen Tarifen erlauben. Eine Beschreibung der neuen Funktionen und deren Nutzung enthält Abschnitt 28.7.2.
Da es keine offenen, klaren Standards für die Verteilung von Tarifen gibt, wird die Projektgruppe eine offene Schnittstelle für die Importfunktionen erstellen. Diese kann dann beliebig implementiert werden, um verschiedene Formate und Quellen in das von der Projektgruppe genutzte Modell zu konvertieren. Gesteuert werden diese Vorgänge über `ImportJobs`, die quasi als Konfiguration eines `Importers` dienen. Genauer wird dies unter Punkt 28.7.1 beschrieben.
Diese Funktionen führen zu neuen funktionalen Anforderungen an den Server:

- Einen `ImportJob` konfigurieren
- Einen `ImportJob` löschen
- Einen `ImportJob` bearbeiten

Es fehlte bisher eine Funktion für die Protokollierung und das Monitoring des Servers, die über den Webbrowser erreichbar ist. Um einen Überblick über neu eingestellte Geräte oder importierte Tarife zu erhalten, wird unter 28.7.3 ein Protokollmechanismus eingeführt, mit dem wichtige Ereignisse als `Log`-Objekte in der Datenbank gespeichert werden. Ein Administrator kann diese dann, nach Dringlichkeit gefiltert anzeigen lassen.
Das Package `de.oldenburg.uni.informatik.diem.server.persistence` enthält alle Klassen, die für den Zugriff auf persistente Daten benötigt werden. Dazu dient

Name	- Ein beschreibender Bezeichner oder eine Kurzbeschreibung
LastRun	- Der Letzte Zeitpunkt zu dem ein Import durchgeführt wurde
Interval	- Zeitintervall der Aktualisierung in Stunden
Type	- Der Importertyp als vollqualifizierten Klassenbezeichner
Source	- Die externe Quelle der Tarifdaten
Target	- Das interne Ziel innerhalb der Bibliothek
Merge	- true: Alte Werte bleiben erhalten / false: Alte Werte werden ersetzt
Param	- Parameter im Properties-stil zur Konfiguration des Importers

Tabelle 7: Eigenschaften eines `ImportJob`

die generische `ModelAccess`-Schnittstelle, die einen einfacheren Zugriff auf die Datenhaltung mit Hibernate ermöglichen soll. Hierzu werden Accessklassen erstellt, die von einer Hilfsklasse für Hibernatenutzung erben und über die man transparent an die Objekte aus der Persistenzschicht des Servers gelangen kann. Dies soll vor allem bei der Implementierung der Importer hilfreich sein.

28.7.1. Importer Schnittstelle

Um eine möglichst freie und offene Möglichkeit des Datenimports zu ermöglichen, enthält das Package `de.oldenburg.uni.informatik.diem.server.importer` das Interface `Importer`. Ein `Importer` stellt mit Hilfe eines `ImportJobs` eine Verbindung zu einer externen Quelle her, verarbeitet die Fremddaten und portiert sie in das genutzte Datenmodell.
Ein `ImportJob` dient als Konfiguration eines Importers und zeigt auch dem `Scheduler`, welche Implementierung des Importers er instanzieren muss. Über das Feld `Type` (Siehe auch Tabelle 28.7.1) wird der `Importer` eindeutig indentifiziert und geladen. Somit ist es auch möglich, nachträglich neue `Importer`-Implementierungen einzufügen und zu verwenden. Das Feld `Source` beschreibt eine URI oder einen Pfad an der die externen Daten bezogen werden können. Die Interpretation obliegt hierbei dem `Importer`. `Target` bezeichnet einen Pfad innerhalb der Bibliothek und kann so einen Odner oder einen einzelnen Tarif referenzieren. Um weitere Parameter für die gewünschte `Importer`-Implementierung zu übergeben gibt es den `String Param`, der in der Properties-Schreibweise weitere Parameter enthalten kann. Die Interpretation übernimmt auch hierbei der `Importer`.
Ein `Importer` erhält die benötigten Informationen aus dem zugehörigen `ImportJob`-Objekt, das ihm direkt nach der Initialisierung übergeben werden muss. `Importer` erweitert das Interface `java.util.Runnable` und kann so von einem `ThreadPool` oder einem anderem `Executor` ausgeführt werden. Dies ist die Vorgehensweise des `Schedulers`, der überprüft welche `ImportJobs` abgearbeitet werden müssen, die `Importer` instanziiert und die entsprechenden `ImportJobs` übergibt. Dann wird der Importer über die `run()`-Methode angestoßen. Das Ergebnis des Imports kann dann

auch über `getTarget()` bezogen werden.
Je nach Implementierung wird ein `Importer` die Daten in ein `PayScale`-Objekt aus der Bibliothek übertragen und dieses dann speichern. Danach ist der Tarif über das Update der Bibliothek für den Client verfügbar. Denkbar ist auch ein Aktualisieren eines ganzen Tarif-Ordners durch einen einzelnen Importer und das Veröffentlichen von Week- oder Hour-Ahead-Tarifen.
Die Schnittstelle ist offen genug um auch das Importieren von anderen Elementen des Modells zu ermöglichen. Die Projektgruppe wird davon aber keinen Gebrauch machen. Geplant sind drei Implementierungen. Eine, die aus Daten der Energiebörse EEX einen Day-Ahead-Tarif für den nächsten Tag erstellt. Eine weitere soll aus externen CSV-Dateien eine Tarifkurve erstellen können. Eine dritte kopiert einfach das Projektgruppeneigene XML-Datenformat für Tarife von einem Fileserver in die Bibliothek.

28.7.2. Day-Ahead Tarife

Das genutzte Modell enthält bereits alle Datentypen, die für die Verwendung von variablen Tarifen benötigt werden. Der Typ `PayScale` ist in Hinsicht auf solche Tarife modelliert worden und die bereits im ersten Prototyp implementierte Updatefunktion der Tarifbibliothek genügt zum regelmäßigen Aktualisieren der Day-Ahead-Tarife. Eine Modifikation ist hierbei nur auf Serverseite notwendig, da man eine Quelle und eine Typkonvertierung oder ähnliches festlegen können muss. Hierzu wird der Typ `ImportJob` erstellt. Er dient als Konfiguration für den `Importer`. Eine genaue Beschreibung der Felder gibt Tabelle 28.7.1.
Um einen Day-Ahead-Tarif zu konfigurieren legt der Administrator einen Tarif innerhalb der Bibliothek an, der das zukünftige Ziel des Imports sein soll. Je nach konfiguration können die alten Tarifdaten dieses Tarifes beim Import gelöscht oder beibehalten werden. In letzterem Fall werden bestehende Preise überschrieben, falls für diesen Zeitpunkt neue vorgesehen Dann wird ein `ImportJob` erstellt, der den Pfad des Tarifes als Ziel erhält. Der gewünschte `Importer` und die Quelle wird festgelegt. Wenn der `ImportJob` vollständig konfiguriert ist, wird er gespeichert und in die Datenbank übertragen. Der Server enthält eine aktuelle Liste aller `ImportJobs` die regelmäßig vom `Scheduler` auf durchzuführende `ImportJobs` überprüft wird. Der `Scheduler` instanziert dann den im `ImportJob` referenzierten `Importer`, übergibt den `ImportJob` und startet den Import.
Die aktualisierten Tarife werden dann in der Tarifbibliothek aktualisiert. Für den Client wird der aktualisierte variable Tarif durch ein Update der Tarifbibliothek dann verfügbar. Da aus Sicht des Clients keine Änderung am Modell oder deren Nutzung notwendig ist, bleibt der eigentliche Importvorgang der Tarife für den Nutzer transparent. Es ist nur die regelmäßige Nutzung der Updatefunktion notwendig, wobei das bei der Natur solcher Tarife nicht anders erwartet werden kann.
Verwaltet werden diese Funktionen über das `JobServlet` mit dem Paramter `action`. Die verfügbaren Parameter für `action` zeigt Tabelle 28.7.2. Die Darstellung geschieht über `jobdialog.jsp`.

Wert	Aktion	Parameter
LIST	Es werden die im Server vorhanden `ImportJobs` angezeigt	Keine
DELETE	Der mit `ID` referenzierte `ImportJob` wird gelöscht	`ID`
CREATE	Erzeugt einen neuen `ImportJob` im Server	Entsprechen den Feldern in Tabelle 28.7.1 bis auf `LastRun`
MODIFY	Ändert die Parameter eines `ImportJobs`, der mit `ID` referenziert wird	Entsprechen den Feldern in Tabelle 28.7.1 bis auf `LastRun`

Tabelle 8: Mögliche Aktionen des `JobServlets` über `action`-Parameter

28.7.3. Protokollierung

Für die Protokollierung verwendet die Projektgruppe eine eigene Lösung in Form der neu modellierten `Log`-Objekte. Wichtige Ereignisse werden über diese `Log`-Objekte im Protokoll des Servers gespeichert. Es wird hierfür eine eigene Lösung verwendet, da sich diese besser im Server und den Servlets integrieren lässt und die Einbettung im Modell die gleiche einfache Verwendung von Hibernate wie bei den Geräte und Tarifbibliotheken ermöglicht. Außerdem ist dafür nicht der Funktionsumfang des bisher verwendeten Log4J notwendig. Log4J wird weiterhin für die Ausgabe von Debug- und Fehlerinformationen in Form von Serverlogfiles verwendet, da es frei konfiguriert werden und auch Meldungen aus den genutzten Frameworks ausgeben kann.
Hierbei gibt es drei Stufen der Wichtigkeit.

1. Error - Schwerwiegendes Ereignis. Zeigt Fehler an, die einen Abbruch einer Operation bewirkt haben könnten.
2. Warn - Ein wichtiges Ereignis. Zeigt einen aufgefangenen Fehler oder den Upload eines neuen Gerätes bzw. Tarifes an.
3. Info - Generelle Information, wie das umbenennen von Ordnern, Aktionen des Schedulers oder das Verifizieren von Geräten.

Der Administrator kann sich diese über einen Webdialog anzeigen lassen. Hierfür wird eine neue JSP `logs.jsp` erstellt. Der Aufruf des Dialoges erfolgt über das `LogServlet` ohne notwendigen Parameter. Optionale Parameter sind `count` als Angabe wieviele Logs angezeigt werden sollen und `severity` mit den kombinierbaren Werten `ERROR`, `WARN` und `INFO`.

Teil VI.
Feinentwürfe

29. Feinentwurf grafische Editoren

Autor: Mark Eilers

29.1. Einleitung

Dieser Feinentwurf beschreibt den Aufbau der für die Anwendung DIEM entwickelten grafischen Editoren. Der Begriff des grafischen Editors stammt noch aus der ursprünglich geplanten Verwendung des Frameworks GEF. Da im Laufe der Projektgruppe entschieden wurde GEF nicht mehr zu verwenden, ist dieser Begriff eigentlich nicht ganz richtig, wird aber zur Namensgebung weiterhin verwendet. Es gilt jedoch zu beacheten, dass ein grafischer Editor im eigentlichem Sinne nicht mehr unbedingt als Editor, sondern eher als SWT-Composite zur Visualisierung von Tarifen und Lastgängen in Form von Treppenfunktionen zu verstehen ist.
Für die Anwendung DIEM wurden drei verschiedene grafische Editoren entwickelt: Der `FunctionEditor`, der `FunctionViewer` und der `ScheduleFunctionViewer`. Der `FunctionEditor` bietet die Möglichkeit einen einzelnen Tarif oder Lastgang zu bearbeiten, der `FunctionViewer` und der `ScheduleFunctionViewer` visualisieren mehrere Tarife und Lastgänge zur selben Zeit.

29.2. Paketübersicht

Die grafischen Editoren werden an mehreren Stellen der Anwendung DIEM verwendet. Aus diesem Grund wurde bereits im Entwurf festgelegt, dass sie im Plugin `shared` implementiert werden sollen. Für die Klassen der grafischen Editoren sind folgende Pakete erstellt worden:

- `shared.graphicaleditors`
  Das Paket enthält neben dem `FunctionComposite` als Elternklasse der grafischen Editoren den `FunctionEditor`, den `FunctionViewer` und den `ScheduleFunctionViewer`.

- `shared.graphicaleditors.figures`
  Das Paket enthält fünf Klassen zur Visualisierung von Treppenfunktion und Achsen: das `XAxisFigure`, das `YAxisFigure`, das `FunctionFigure`, das `FunctionCurveFigure` und das `ViewerPairFigure`.

- `shared.graphicaleditors.helpers`
  Dieses Paket enthält nur eine Klasse: Das `ZoomHandle`, in dem Werte für Zoom und Skalierung gespeichert werden.

29.3. Klassenübersicht

Im Folgenden soll eine Übersicht aller von den grafischen Editoren verwendeter Klassen erfolgen. Abbildung 115 zeigt zum besseren Verständnis ein Klassendiagramm dieser Klassen mitsamt den wichtigsten Methoden.

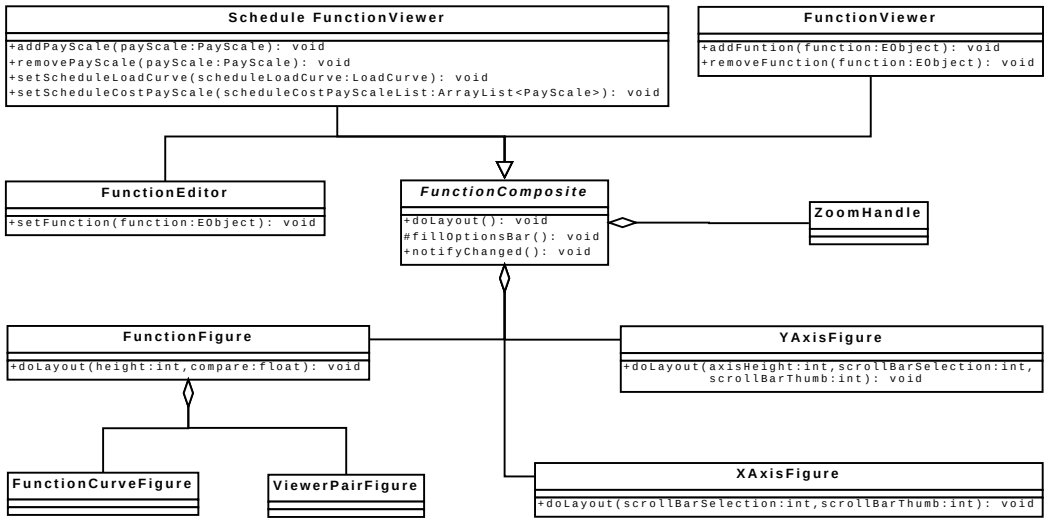

Abbildung 115: Klassendiagramm der grafischen Editoren

- `shared.graphicaleditors`
  - **`FunctionComposite`**
    Das `FunctionComposite` ist die abstrakte Elternklasse aller grafischen Editoren. Als abstrakte Klasse wird sie nicht instanziiert aber sie definiert einige Methoden, die von allen Erweiterungen implementiert werden müssen. Als Grundlage aller grafischen Editoren kapselt es die Elemente, die allen grafischen Editoren gemein sind. Das `FunctionComposite` erweitert die Klasse `Composite` des SWT und gewährleistet dadurch, dass ein grafischer Editor wie eine normale SWT-Komponente verwendet werden kann. Weiterhin implementiert es das Interface `Adapter`, damit sich ein grafischer Editor als Listener von EMF-Elementen registrieren kann.

    Das `FunctionComposite` definiert den grundlegenden Aufbau der grafischen Editoren. Es besteht aus einer Optionsleiste, einer Legende, der X- und Y-Achse sowie der Zeichenfläche. Dieser Aufbau ist schematisch in Abbildung 116 dargestellt.

Die Optionsleiste ist ein eigenes `Composite` und enthält neben `Slidern` zum Einstellen des Zooms, je nach Art des grafischen Editors verschiedene SWT-Komponenten, die die Verwendung aller speziellen Features des grafischen Editors ermöglichen. Die Höhe der Optionsleiste wird durch einen Klassenparameter `optionsHeight` bestimmt. Zum Füllen der Optionsleiste definiert das `FunctionComposite` die abstrakte Methode `fillOptionsBar()`.

Die Legende besteht aus einem `ScrolledComposite`, sie stellt also eigene Scrollbalken zur Verfügung. Für jede im grafischen Editor visualisierte Treppenfunktion wird ein aus verschiedenen `Labels` bestehender Eintrag zur Legende hinzugefügt. Die Breite der Legende wird durch den Klassenparameter `legendWidth` bestimmt, das Hinzufügen von Elementen zur Legende wird von den Erweiterungen des `FunctionComposite` beim Hinzufügen von Tarifen oder Lastgängen selbst gesteuert.

X-Achse, Y-Achse und die Zeichenfläche bestehen jeweils aus einem `FigureCanvas`, die Zeichenfläche befindet sich zusätzlich in einem `ScrolledComposite`. X- und Y-Achse enthalten gemäß ihres Names eine `X-` bzw. eine `YAxisFigure`. Der Zeichenfläche wird für jeden zu visualierenden Tarif oder Lastgang eine `FunctionFigure` als Treppenfunktion hinzugefügt.

Neben der Methode `fillOptionBar()` definiert das `FunctionComposite` noch zwei weitere abstrakte Methoden: `doLayout()` und `notifyChanged()`. Die Methode `doLayout()` wird aufgerufen wenn ein grafischer Editor neu gezeichnet werden soll. Innerhalb der Methode müssen dafür zuerst die aktuellen Werte des Zooms und der Skalierung berechnet werden und danach die `doLayout()`-Methoden der `X-` und `YAxisFigure` sowie aller `FunctionFigures` aufgerufen werden. Die Werte des Zooms für X- und Y-Ebene können in einem grafischen Editor durch die Zoomslider verändert werden. Die Werte der Skalierung basieren auf der aktuellen Größe der Zeichenfläche und werden bei jeder Größenänderung derselben vom grafischen Editor neu berechnet und an ein `ZoomHandle` übergeben. Sie sind dabei so bemessen, dass unabhängig von der Größe der Zeichenfläche immer alle Treppenfunktionen auf niedrigster Zoomstufe als Ganzes sichtbar sind. Da sich die genauen Berechnungen je nach Art des grafischen Editors unterscheiden, wird die Methode `doLayout()` nur als abstrakte Methode definiert, den Inhalt der Methode müssen die Erweiterungen des `FunctionComposites` realisieren.

Die Methode `notifyChanged()` muss für das Interface `Adapter` implementiert werden. Die grafischen Editoren registrieren sich für jeden zu visualisierenden `PayScale` oder `LoadCurve` als Adapter. Als Folge werden sie durch einen Aufruf der Methode `notifyChanged()` informiert, sobald sich

ein überwachtes Objekt ändert. Innerhalb der Methode wird dann überprüft inwieweit ein Neuzeichnen des grafischen Editors notwendig geworden ist. Da sich die genaue Vorgehensweise je nach Art des grafischen Editors unterscheidet, muss auch diese Methode von jeder Erweiterung des `FunctionComposite` selbst implementiert werden.

- **`FunctionEditor`**
  Der `FunctionEditor` erweitert die Klasse `FunctionComposite` und bietet die Möglichkeit die Wertepaare innerhalb von Objekten des Typs `PayScale` oder `LoadCurve` zu bearbeiten. Er wird vom `LoadCurveEditor`, `PayScaleEditor` und dem `DeviceEditor` verwendet.

  Der `FunctionEditor` implementiert eine Methode `setFunction()`, über die ein Objekt des Typs `PayScale` oder `LoadCurve` übergeben werden kann. Innerhalb der Methode registriert sich der `FunctionEditor` als Adapter für das übergebene Objekt, dann wird ein `FunctionFigure` erstellt und der Zeichenfläche hinzugefügt. Der `FunctionEditor` kann immer nur ein einzelnes Objekt bearbeiten. Da daher auch immer nur eine Treppenfunktion visualisiert wird und eine Verwechslungsgefahr mit anderen Treppenfunktionen ausgeschlossen ist, verzichtet der `FunctionEditor` auf eine explizite Legende indem der Wert für `legendWidth` auf 0 gesetzt wird.

  Als Erweiterung des `FunctionComposites` muss der `FunctionEditor` die Methoden `doLayout()`, `fillOptionsBar()` und `notifyChanged()` implementieren. In der Methode `doLayout()` werden, wie bereits für das `FunctionComposite` beschrieben, die aktuellen Werte für Zoom und Skalierung berechnet und danach die `doLayout()`-Methoden des `FunctionFigures` sowie des `X`- und `YAxisFigures` aufgerufen.

  In der Methode `notifyChanged()` wird überprüft inwieweit sich das überwachte Objekt verändert hat. Hat sich die Länge geändert, dann muss auch die Länge des `XAxisFigures` geändert werden, hat sich der maximale Wert des Objekts geändert, muss auch die Höhe des `YAxisFigures` angepasst werden. Hat sich der Name des Objekts geändert muss ein entsprechendes `Label` in der Optionsleiste geändert werden.

  In der Methode `fillOptionsBar()` wird die Optionsleiste des `FunctionEditors` um verschiedene SWT-Komponenten erweitert. Dazu gehören neben den Slidern für den Zoom vor allem zwei SWT-Komponenten des Typs `DateTime`, zum Einstellen von Datum und Uhrzeit, sowie ein `Spinner` zum Einstellen des Wertes von Cent oder Watt. Zusätzlich werden drei `Buttons` implementiert die dem Erstellen, Verschieben und Entfernen von Wertepaaren in einem `PayScale` oder einer `LoadCurve` dienen.

  Das visualisierte Objekt kann durch das Bearbeiten der repräsentierenden

Treppenfunktion mittels Maus oder durch die Elemente in der Optionsleiste auch mittels Tastatur editiert werden. Dafür implementiert der `FunctionEditor` verschiedene `Listener` als innere Klassen:

* **`MyMouseListener`**
  Der `MyMouseListener` ist die Verbindung eines `MouseListeners` zum Überwachen der Maustasten und eines `MouseMotionListeners` zum Überwachen von Mausbewegungen. Der `MyMouseListener` implementiert die Möglichkeit alle Punkte der Treppenkurve zu verschieben, neue Punkte zu erstellen oder bestehende zu entfernen. Diese Änderungen werden dann innerhalb des `Listeners` direkt auf das Modell übertragen und die Treppenkurve auf Basis des Modells neu gezeichnet. Der `MyMouseListener` ist mit einer Länge von über 1000 Zeilen sehr komplex geraten. Die Begründung für diese hohe Komplexität ist der Wunsch nach einer möglichst guten Performance. Um diese zu gewährleisten wird vom `MyMouseListener` bei jeder Aktion geprüft, ob ein komplettes Neuzeichnen aller Komponenten durch einen Aufruf der Methode `doLayout()` des `FunctionEditors` notwendig ist, oder ob ein teilweises Layout einzelner Komponenten genügt. Werden beispielsweise Punkte mit der Maus verschoben, so genügt im Großteil der Fälle ein einfaches Neuzeichnen der Treppenfunktion, während X- und Y-Achse unverändert bleiben können.

* **`MyNewSelectionListener`**
  Die Klasse `MyNewSelectionListener` implementiert das Interface `SelectionListener` und wird benachrichtigt, wenn der Anwender den entsprechenden `Button` in der Optionsleiste betätigt um einen neuen Punkt zu erstellen. Im Gegensatz zu dem `MyMouseListener` muss bei dem Bearbeiten über die Tastatur kein gesteigerter Wert auf die Perfomance gelegt werden, daher werden die Werte aus der Optionsleiste eingelesen, ein neuer Punkt im `PayScale` oder `LoadCurve` erstellt und die Methode `doLayout()` aufgerufen.

* **`MyChangeSelectionListener`**
  Die Klasse `MyChangeSelectionListener` implementiert das Interface `SelectionListener` und wird benachrichtigt, wenn der Anwender den entsprechenden `Button` in der Optionsleiste betätigt um einen Punkt zu verschieben.

* **`MyDeleteSelectionListener`**
  Die Klasse `MyDeleteSelectionListener` implementiert das Interface `SelectionListener` und wird benachrichtigt, wenn der Anwender den entsprechenden `Button` in der Optionsleiste betätigt um einen Punkt zu entfernen.

- **FunctionViewer**
  Der FunctionViewer ist eine Erweiterung des FunctionComposites und wird in der Statistik verwendet um mehrere Tarife und Lastgänge gleichzeitig zu visualisieren. Dafür implementiert der FunctionViewer zwei Methoden: addFunction() und removeFunction(). Mit der Methode addFunction() kann dem FunctionViewer ein Objekt des Typs PayScale oder LoadCurve übergeben werden. Innerhalb der Methode wird dann ein Eintrag für die Legende und ein neues FunctionFigure für das übergebene Objekt gebildet. Durch die Methode removeFunction() kann ein Objekt angegeben werden, das aus dem FunctionViewer entfernt werden soll.

  Als Erweiterung des FunctionComposite muss der FunctionViewer die Methoden doLayout(), fillOptionsBar() und notifyChanged() implementieren. In der Methode fillOptionsBar() wird die Optionsleiste des FunctionViewers um verschiedene SWT-Komponenten erweitert. Die Methode doLayout() unterscheidet sich nur minimal von der des FunctionEditors. Da der FunctionViewer im Gegensatz zum FunctionEditor mehrere Treppenfunktionen enthalten kann, müssen auch die doLayout()-Methoden von mehreren FunctionFigures aufgerufen werden. Gleiches gilt für die Implementierung der Methode notifyChanged().

  Der FunctionViewer bietet verschiedene Möglichkeiten die Darstellung der Treppenkurven zu verändern. Alle werden dem Anwender über SWT-Komponenten in der Optionsleiste zur Verfügung gestellt. Die eigentliche Funktionlität befindet sich jedoch nicht im FunctionViewer sondern in den Figures selbst. Der FunctionViewer setzt nur benötigte Werte innerhalb der einzelnen FunctionFigures oder des X- und YAxisFigures, die sich dann selber um eine neue Darstellung kümmern. Beispielsweise ist es möglich den Startzeitpunkt aller Treppenfunktionen auf einen Nullpunkt zu setzen oder die Treppenkurven gemäß des tatsächlichen Startzeitpunkts des repräsentierten Objekts anzuordnen. Dafür stellen sowohl das X- und YAxisFigure als auch ein FunctionFigure notwendige Methoden bereit um einen Startzeitpunkt festzulegen.

- **ScheduleFunctionViewer**
  Der ScheduleFunctionViewer ist eine Erweiterung des FunctionComposite und wird innerhalb des Einsatzplaneditors verwendet. Er kann als eine in ihrer Funktionalität reduzierte und an die Bedürfnisse des Einsatzplaneditors angepasste Form des FunctionViewers verstanden werden. Wie der FunctionViewer ist er in der Lage, mehrere Treppenfunktionen gleichzeitig zu visualisieren.

  Um dem Einsatzplaneditor zu ermöglichen, verschiedene Treppenfunktionen eines Einsatzplans im ScheduleFunctionViewer zu visualisieren, im-

plementiert der `ScheduleFunctionViewer` die Methoden `addPayScale()`, `removePayScale()`,`setScheduleLoadCurve()` und `setScheduleCostPayScale()`.

Über die Methoden `addPayScale()` kann dem `ScheduleFunctionViewer` ein `PayScale` hinzugefügt werden, während über die Methode `removePayScale` ein `PayScale` aus dem `ScheduleFunctionViewer` entfernt werden kann.

Über die Methode `setScheduleLoadCurve()` kann dem `ScheduleFunctionViewer` ein `LoadCurve`-Objekt übergeben werden. Die Methode wird bei Öffnen des Einsatzplans einmalig verwendet um dem `ScheduleFunctionViewer` den aus den Einsätzen resultierenden Lastgang des Einsatzplans zu übergeben. Da sich der `ScheduleFunctionViewer` als Adapter des `LoadCurves` registriert, wird er bei allen Änderungen des `LoadCurves` durch Aktivitäten im Einsatzplaneditors in Form eines Aufrufs der Methode `notifyChanged()` automatisch benachrichtigt.

Über die Methode `setScheduleCostPayScale()` bekommt der `ScheduleFunctionViewer` eine `ArrayList` von `PayScales` übergeben. Diese Methode wird vom Einsatzplaneditor nach jeder Änderung neu aufgerufen um die Kostenkurven des Einsatzplans zu übergeben.

Als Erweiterung des `FunctionComposite` muss auch der `ScheduleFunctionViewer` die Methoden `doLayout()`, `fillOptionsBar()` und `notifyChanged()` implementieren. Für diese Methoden gilt das Gleiche wie bereits beim `FunctionViewer` erwähnt.

- `shared.graphicaleditors.figure`
  - **`XAxisFigure`**
    Das `XAxisFigure` ist eine Erweiterung der Draw2d-Klasse `Figure` und visualisiert die X-Achse innerhalb eines grafischen Editors. Das `XAxisFigure` besteht aus einer horizontalen `Polyline` als X-Achse sowie einer Anzahl von maximal 51 `Polylines` und dazugehörigen `Labels` als Beschriftungen der Achse.

    Das `XAxisFigure` kümmert sich selbst um das richtige Layout. Es besitzt verschiedene Klassenparameter, welche die Länge und den Startzeitpunkt des `XAxisFigures` bestimmen und auf die über `Get-` und `Set`-Methoden zugegriffen werden kann. Ein Aufruf der Methode `doLayout()` führt zu einem neuen Layout des `XAxisFigures`. Dabei bekommt das `XAxisFigure` den derzeitigen Sichtbereich der Zeichenfläche eines grafischen Editors übergeben. Aus diesen Werten wird dann berechnet wieviele und wo Unterteilungen auf der X-Achse dargestellt werden müssen. Aus der Startzeit der X-

Achse und der Position der Unterteilungen wird dann für jede Unterteilung ein konkreter Zeitpunkt in Form eines `GregorianCalendar` berechnet und mit dessen Hilfe die eigentliche Beschriftungen für eine Unterteilung ermittelt. Die Umwandlung in ein zeitliches Datum bietet den Vorteil einer korrekten Zeitdarstellung trotz unterschiedlicher Monatslängen oder Schaltjahren. Das `XAxisFigure` erlaubt dadurch ein stufenloses und genaues Verschieben und Zoomen der X-Achse in einem Zeitbereich von Sekunden bis zu Jahrzehnten.

- **`YAxisFigure`**
  Das `YAxisFigure` ist eine Erweiterung der Draw2d-Klasse `Figure` und visualisiert die Y-Achse innerhalb eines grafischen Editors. Die Funktionsweise des `YAxisFigure`s ist mit der des `XAxisFigure`s vergleichbar.

  Das `YAxisFigure` ist in der Lage, zwei unterschiedliche Unterteilungen für Tarife und Lastgänge zeitgleich auf einer Achse darzustellen. Daher verfügt es über eine vertikale `Polyline` als Achse und jeweils 21 `Polylines` für die Unterteilungen. Ein Aufruf der Methode `doLayout()` führt wiederum zu einem Neuzeichnen des `YAxisFigures` Dabei bekommt es zusätzlich zu dem Sichtbereich der Zeichenfläche die Gesamthöhe übergeben. Dies ist notwendig, da der Ursprung eines `Figures` in der linken, oberen Ecke liegt, während die Y-Achse ihren Ursprung in der linken, unteren Ecke besitzt. Die Positionen der einzelnen Unterteilungen müssen daher mit Hilfe der Gesamthöhe umberechnet werden.

- **`FunctionFigure`**
  Das `FunctionFigure` ist eine Erweiterung der Draw2d-Klasse `Figure` und kann sowohl einen Tarif als auch einen Lastgang visualisieren. Bei der Initialisierung bekommt ein `FunctionFigure` das zu visualisierende `PayScale`- oder `LoadCurve`-Objekt und über eine weitere Methode ein `ZoomHandle` von einem grafischen Editor übergeben.

  Ein `FunctionFigure` besteht aus einer `FunctionCurveFigure`, einem Polygon zur Visualisierung der Treppenfunktion, und je einem `ViewerPairFigure` pro Wertepaar des bei der Initialisierung übergebenen `PayScale`- bzw. `LoadCurve`-Objekts.

  Wie das `X`- und das `YAxisFigure` kümmert sich auch ein `FunctionFigure` bei einem Aufruf der Methode `doLayout()` selbstständig um das richtige Layout. Dafür wird die Position jedes Punktes der `FunctionCurveFigure` und die Position aller `ViewerPairFigures` aus den Werten der Wertepaare des repräsentierten Objekts und aus den im `ZoomHandle` gespeicherten Werten für Zoom und Skalierung berechnet.

  - **`FunctionCurveFigure`**
    Die Klasse `FunctionCurveFigure` erweitert die Draw2d-Klasse `Polygon` um die Möglichkeit über verschiedene Funktionen zwischen einer transparenten oder undurchlässigen sowie zwischen einer gefüllten oder ungefüllten Darstellung zu wechseln.

  - **`ViewerPairFigure`**
    Die Klasse `ViewerPairFigure` erweitert die Draw2d-Klasse `Figure` und visualisiert ein einzelnes Wertepaar eines `PayScales` oder einer `LoadCurve`. Über Methoden kann gesteuert werden, ob ein `ViewerPairFigure` bei einem Aufruf der Methode `doLayout` der dazugehörigen `FunctionFigure` gezeichnet wird oder nicht.

- `shared.graphicaleditors.helpers`

  - **`ZoomHandle`**
    Das `ZoomHandle` ist eine Helferklasse zur Kapselung der aktuellen Werte von Zoom und Skalierung in Form von Fließkommazahlen. Ein `ZoomHandle` ist Teil jedes grafischen Editors und wird per Referenz an alle `Function-Figures` wie auch an das `X-` und das `YAxisFigure` übergeben.

30. Feinentwurf Logikpaket

30.1. Einleitung

Das Logik-Paket umfasst entsprechend dem Entwurf die automatische Einsatzplanoptimierung. Daneben enthält es Funktionen für statistische Berechnungen und einen Algorithmus zur Überprüfung der Constraintzustände in Einsatzplan und Gerätekomposition. Durch diese Kapselung wird erreicht, dass diese Funktionalitäten von allen Komponenten innerhalb der Anwendung verwendet werden können. Dadurch sind alle Möglichkeiten zur Modifikation von Daten, die nicht vom Anwender durch einen der Editoren direkt auf dem Modell durchgeführt werden, sondern durch mathematische Berechnung geschehen, an dieser Stelle zentralisiert.

30.2. Schnittstelle

Die Schnittstelle des Logik-Paketes besteht aus zwei Teilen.
Der erste Teil wird durch das Interface `LogicFacade` definiert, das von der Klasse `Logic` implementiert wird. Die gegebenen Methoden können den drei Funktionalitäten des Logik-Pakets zugeordnet werden.
Die folgenden Methoden dienen dabei der Verbindung zur automatischen Optimierung: Die beiden Methoden `executeAlgorithm()` dienen zum Anstoßen der `plan()`-Methoden innerhalb der automatischen Einsatzplanoptimierung, welche in Abschnitt 30.3 genauer beschrieben werden. Über die Methode `getAvailableAlgorithms()` erhält man die Liste der Descriptoren aller implementierten Optimierungsalgorithmen in der Form `List<AlgorithmDescriptor>`.
Die Methode `calcConstraintStatus()` bietet die Constraintzustandsüberprüfung an.
Für die mathematische Statistik gibt es zwei Gruppen von Funktionen. Für die graphische Statistik gibt es eine Klasse, welche den Gesamtlastgang errechnet. Sie ist über die Methode `deriveLoadCurveFromSchedule()` erreichbar. Weiter gibt es eine Klasse, um aus einem Einsatzplan die entsprechende Kostenkurve zu ermitteln, angeboten über die Methode `deriveCostcurve()`.
Die numerische Statistik besteht aus Methoden, welche generelle Eigenschaften zurück geben, zumeist als `float`-Wert (vergleiche Abbildung 117). Auf die einzelnen mathematischen Berechnung kann durch die Methoden `calcTaskPrice()`, `getMostExpensiveTask()`, `getMaxOfLoad()`, `getMaxPayScaleValue()`, `getMaxLoadCurveValue()`, `getMinPayScaleValue()`, `getMinLoadCurveValue()`, `getMiddlePayScaleValue()`, `getTotalLoadOfLoadCurve()`, `getMiddleLoadCuvreValue()`, `getCostOfLoadCurveByPayScale()` und `getMiddleCostOfPayScaleByLoadCurve()` zugegriffen werden.

Der zweite Teil der Schnittstelle des Logik-Paketes wird über vier so genannte Actions

aus dem EMF-Konzept geboten. Sie bieten Menüpunkte zum Einbinden in die Oberfläche und dadurch Möglichkeiten die Funktionalitäten des Paketes direkt auf Anwenderwunsch anzustoßen. Es handelt sich dabei um die folgenden vier Actions:

`PerformScheduleOptimizationAction` Diese Action startet den Auswahldialog, der von der Klasse `AlgorithmConfigDialog` geboten wird (vergleiche Abschnitt 30.3). Nach Auswahl eines Algorithmus und Erstellung der Konfiguration wird die Optimierung über die entsprechende `plan()`-Methode angestoßen.

`PerformDeviceCompositionOptimizationAction` Diese Action startet ebenfalls einen Auswahldialog, jedoch bietet dieser nur das Fenster zur Auswahl eines Algorithmus, da die Konfiguration für die Constraintherstellung nicht notwendig erscheint. Nach der Auswahl des Algorithmus wird die entsprechende `plan()`-Methode angestoßen.

`ConstraintCheckAction` Diese Action bietet den Algorithmus zur Überprüfung der Constraints an.

`CalculateNumericalStatisticsAction` Einige der statistischen Funktionen werden direkt im Einsatzplan-Editor angezeigt. Um diese anstoßen zu können wurde die Action `CalculateNumericalStatisticsAction` angelegt.

30.3. Automatische Optimierung

Im Logikpaket wird eine Gruppe von Interfaces bereit gestellt, so dass die Implementierung von Lösungsverfahren erleichtert wird. Zusätzlich existiert die Klasse `AlgorithmConfigDialog`, welche vorgeschaltet vor den Optimierungsvorgang ausgeführt wird. Im Folgenden werden die einzelnen Klassen, welche im Diagramm in Abbildung 118 in ihrer Struktur zu sehen sind, einzeln in ihrer Funktionsweise erläutert.

`Algorithm` Vom implementierten Algorithmus müssen zwei Funktionalitäten bereit gestellt werden: Einerseits die Optimierung des Einsatzplanes anhand der Tarifkurve, andererseits die Erstellung der Constraints in Gerätekompositionen. Hierfür existiert die Klasse `Algorithm`. Durch diese Schnittstelle erhält man die Verbindung zwischen `Logic` und dem umgesetzten Algorithmus. Die Klasse bietet zwei `plan()`-Methoden an, welche jeweils eine der beiden geforderten Anforderungen umsetzen müssen. Die Rückgabe beider `plan()`-Methoden ist ein `String`. Dieser wird nach Ausführung des Algorithmus in einem einfachen Dialogfenster an den Anwender ausgegeben. Somit erhält der Entwickler die Möglichkeit eine Nachricht oder ein Protokoll über den Optimierungsvorgang an den Anwender auszugeben. Die Algorithmen innerhalb der `plan()`-Methoden haben Zugriff auf die jeweilige Implementierung der `AlgorithmConfig`, welche die Konfigurationsparameter für den Algorithmus bereit hält.

`AlgorithmDescriptor` Jeder implementierte Algorithmus muss einen Descriptor anlegen. In diesem werden grundlegende Informationen wie der Name (`getName()`) und eine kurze Beschreibung (`getDescription()`) bereitgehtalten, so dass der Algorithmus im Auswahlfenster innerhalb von `AlgorithmConfigDialog` beschrieben werden kann. Weiter kennt diese Klasse die Implementierung des Algorithmus, welche über `getAlgorithm()` erreicht werden kann, sowie die Konfiguration (`|`AlgorithmConfig`|`) und das dazugehörige Composite (`AlgorithmConfigComposite`) zum Anzeigen im Dialog (jeweils über `getConfig()` und `getConfigComposite()`).

`AlgorithmConfig` Die Parameter für die Konfiguration eines Algorithmus werden in der `AlgorithmConfig` bereit gehalten. Da diese sehr Algorithmus-spezifisch sind, hält dieses Interface keine Methoden vor.

`AlgorithmConfigComposite` Die `AlgorithmConfigComposite` stellt das Konfigurationsfenster für den Algorithmus dar. Sie erbt von `org.eclipse.swt.widgets.Composite`, um innerhalb des `AlgorithmConfigDialog` angezeigt werden zu können. Dazu kümmert sie sich um die Erstellung und das Füllen der jeweiligen `AlgorithmConfig`, da hier zu jedem einstellbaren Parameter ein Eingabefeld vorhanden ist, welches die Eingabe vom Anwender entgegen nimmt.

`AlgorithmConfigDialog` Die Klasse `AlgorithmConfigDialog` erbt von `org.eclipse.jface.dialogs.Dialog` und stellt einen Dialog dar, welcher vor der Optimierung angestoßen wird. Innerhalb des Dialoges werden zwei Composites angezeigt. Im Ersten wird eine Liste aller implementierten Algorithmen zur Auswahl geboten. Dazu muss dem Dialog beim Instanziieren die Liste aller im Logik-Paket umgesetzten Algorithmus-Descriptoren (in der Form `List<AlgorithmDescriptor>`) übergeben werden. Hieraus werden mittels der `getName()`- und `getDescription()`-Methoden Informationen für den Anwender erzeugt. Nach Auswahl des jeweiligen Algorithmus und dem Druck auf den Weiter-Button wird vom jeweiligen Descriptor durch die `getAlgorithmConfigComposite()`-Methode das Konfigurationsfenster erzeugt, in welchem der Anwender die Parameter für den Algorithmus einstellen kann. Innerhalb der `AlgorithmConfigComposite`-Klasse wird dann die zum Algorithmus gehörige Konfigurationsklasse `AlgorithmConfig` erstellt und der Algorithmus angestoßen.

30.3.1. Implementierung des Choco-Solvers

Eine Implementierung eines Optimierungsalgorithmus ist in Form des Open-Source Constraint-Satisfaction-Problem-Solvers "'Choco"' (im Folgenden Solver genannt) in der Software schon vorhanden. Im Folgenden wird anhand der dazu verwendeten Klassen beschrieben, wie die Implementierung nach der in Abschnitt 30.3 gegebenen Struktur erfolgt.

`ChocoSolver` Die Klasse `ChocoSolver` implementiert das Interface `Algorithm`, das heißt diese Klasse kümmert sich um die Ausführung des Algorithmus. Daher hat sie eine Abhängigkeit auf die "'Choco"'-Bibliothek, welche zum Optimieren verwendet wird. Weiter hat sie eine Abhängigkeit auf `ChocoConfig`, woraus sie die Informationen zur Konfiguration des Solvers entnimmt. Weiter gibt es eine Abhängigkeit auf die Klasse `Solution`, welche sich aus dem Verwendungsansatz für den Solver ergibt, nachzulesen im Dokument zu Prototyp 2 im Kapitel "'Änderungen und Erweiterungen zum Entwurf durch Prototyp 2"' Unterkapitel 2.4 "'Automatische Einsatzplanoptimierung"'. Als letztes hat die Klasse noch eine Abhängigkeit zum Modell. Innerhalb der beiden `plan()`-Methoden wird der Solver eingesetzt, um die jeweils geforderte Änderung am Modell durchzuführen.

`Solution` Innerhalb der Klasse `Solution` wird je Instanz jeweils eine gültige Lösung für das Optimierungsproblem vorgehalten, sowie die dazugehörige Bewertung. Durch die Klasse `ChocoSolver` werden die erzeugten Lösungen jeweils in einer Instanz der Klasse `Solution` abgelegt und miteinander verglichen. Da es sich bereits um fertig formatierte Lösungen handelt, ist eine Abhängigkeit von der "'Choco"'-Bibliothek nicht erforderlich.

`ChocoDescriptor` Der geforderte Descriptor für die Implementierung des Solvers erfolgt durch die Klasse `ChocoDescriptor`.

`ChocoConfig` Die Konfiguration für den Solver wird in `ChocoConfig` gehalten. Hierbei handelt es sich lediglich um so genannte Getter- und Setter-Methoden für die Dauer, die der Algorithmus maximal zur Optimierung einsetzen darf, sowie die Schrittweite in Minuten, in welcher der Algorithmus versuchen soll, die einzelnen Einsätze einzuplanen.

`ChocoConfigComposite` Das zur Konfiguration benötigte Composite wird in `ChocoConfigComposite` implementiert.

Zusätzlich zu diesen expliziten Klassen gibt es noch ein paar weitere Helferklassen. Diese bieten jedoch keine weitere Funktionalität bieten sondern lediglich zur Kapselung einzelner Teilaufgaben dienen. Es handelt sich dabei um die Klassen `ConstAfterEachOtherConverter`, `ConstOverlapConverter`, `TaskConverter` und `VariablesHolder`.

30.4. Zustandsüberprüfung für Constraints

Um die Überprüfung des Zustands der Constraints kümmert sich die Klasse `ConstraintStatusCalculator`, zu sehen in Abbildung 120.
Wegen der Kompaktheit der Anforderung an die Zustandsüberprüfung wird auf eine komplexe Vererbungsstruktur verzichtet. Die Klasse besteht intern aus der statischen Methode `calculate()`, welche als Rückgabe ein Mapping zwischen den einzelnen

Constraints mit ihrem Zustand in der Form `Map<Constraint, ConstraintStatus>` erzeugt. Hierbei sind die vier Zustände `VALID`, `INVALID`, `UNSATISFIED` und `UNNECESSARY` als Möglichkeiten vorgegeben. Aufgrund der hohen Komplexität, welche hinter der Berechnung des `UNNECESSARY`-Constraints stehen würde, wurde dieser bei der Implementierung jedoch bisher nicht berücksichtigt, ist für zukünftige Erweiterungen aber schon vorhanden. Somit enthält die Klasse die Berechungen, ob ein Constraint erfüllt, unerfüllt oder unerfüllbar ist.

30.5. Statistische Funktionen

Die statistischen Funktionen teilen sich auf in die numerische Statistik, welche konkrete Zahlenwerte erzeugt, während die graphische Statistik Kurven berechnet, die dem Anwender in entsprechenden Viewern angezeigt werden.

30.5.1. Numerische Statistik

Die numerische Statistik wird zum größten Teil in der Klasse `StatisticFunctions` umgesetzt. Hierbei wird jeder von der Klasse `Logic` gebotenen Methode eine statische Methode gegenüber gestellt, welche die jeweilige Funktion berechnet. Eine Ausnahme hiervon bildet die Berechnung des Preises eines Einsatzes, da dieser anders als die anderen Berechnungen nicht nur von Lastgang oder Tarif abhängig ist, sondern sich in einem Einsatzplan befinden muss. Diese Funktionalität wird ebenfalls durch eine statische Methode in der Klasse `TaskPriceCalculator` geboten.

30.5.2. Graphische Statistik

Für die graphische Statistik sind zwei Funktionalitäten implementiert, welche sich auf die beiden Klassen `CostcurveCalculator` und `LoadcurveCalculator` aufteilen, zu sehen in Abbildung 122. Diese beiden Klassen werden im Folgenden vorgestellt:

`CostcurveCalculator` Innerhalb der Klasse `CostcurveCalculator` wird aus den übergebenen Lastgang- und Tarifwerten eine Kostenkurve errechnet. Dazu bietet sie eine statische `calculate()`-Methode an. Durch die Kapselung der Funktion in dieser Klasse wird keine Vererbungsstruktur verwendet, sondern die Funktionalität direkt durch die Methode `deriveCostcurve()` aus der Klasse `Logic` angesprochen.

`LoadcurveCalculator` Der tatsächliche Lastgang eines Einsatzplanes wird in der Klasse `LoadcurveCalculator` berechnet. Hierbei werden auch die Standby-Betriebslasten der eingesetzten Geräte beachtet. Ebenso wie bei `CostcurveCalculator` wird die Funktionalität über eine statische `calculate()`-Methode angeboten, welche aus der Klasse `Logic` durch die Methode `deriveLoadCurveFromSchedule()` angestoßen wird.

31. Strukturbeschreibung des Einsatzplaneditors

Autor: Christian Hinrichs

31.1. Einleitung

Der Gerätekompositions- bzw. Einsatzplaneditor ist eine der komplexeren Komponenten der Anwendung DIEM und besteht aus ungefähr 5000 Zeilen Code in über 30 Klassen (gemessen mit dem Eclipse Metrics Plugin[13]). Dazu zählen einerseits die tatsächlich dargestellten grafischen Elemente und andererseits die notwendigen algorithmischen Helferklassen, die das Layout und die Interaktivität steuern. Im Folgenden soll diese Klassenstruktur und das Zusammenspiel der einzelnen Teile näher beschrieben werden.

31.2. Paketübersicht

Bevor auf einzelne Klassen und deren Abhängigkeiten eingegangen wird, soll ein prinzipieller Überblick über die Paketstruktur der Komponente gegeben werden. Es wurde bei der Entwicklung auf eine strukturierte Organisation der Klassen in Bezug auf ihr Einsatzgebiet geachtet, sodass der Aufbau der Pakete zum grundlegenden Verständnis der Komponente hilfreich ist.
Die zu beschreibende Komponente realisiert sowohl den Gerätekompositions- als auch Einsatzplaneditor. Da der Datentyp `Einsatzplan` vom Datentyp `Gerätekomposition` erbt, bot es sich an, diese Vererbungshierarchie auch bei der Implementierung der Editoren anzuwenden. Aus diesem Grunde finden sich fast alle Klassen im Plugin `device`, während das Plugin `schedule` nur die notwendigen Erweiterungen enthält, die der Einsatzplaneditor gegenüber dem Gerätekompositionseditor bietet.

Für den Gerätekompositionseditor sind in dem Plugin `device` die folgenden Pakete enthalten:

- `device.actions`
  Hier finden sich diejenigen Action-Klassen, die nur bei geöffnetem Gerätekompositionseditor verfügbar sind. Zusätzlich befindet sich hier die Klasse `DeviceCompositionEditorActionBarContributor`, die eben diese Verfügbarkeit steuert.

- `device.editors`
  Dieses Paket enthält die konkrete Editor-Klasse `DeviceCompositionEditor`, welche als Ausgangsbasis für alle Teile dieser Komponente betrachtet werden kann und die Schnittstelle zum RCP und zum EMF darstellt.

[13] http://metrics.sf.net

- `device.editors.compositionEditorComposites`
  Hier finden sich alle SWT-Komponenten des grafischen Teiles sowie die algorithmischen Helferklassen des Editors.

- `device.editors.compositionEditorComposites.figures`
  In diesem Paket befinden sich ausschließlich Klassen für die grafische Präsentation. Wie der Paketname andeutet erben sie fast alle von der Draw2D-Klasse `Figure`. Jedoch sind auch hier algorithmische Teile zu finden, da die komplexeren dieser Klassen jeweils ihren eigenen Layouter enthalten, der die einzelnen grafischen Elemente korrekt darstellt.

- `device.editors.compositionEditorComposites.inputValidators`
  Dieses Paket enthält einige Validatoren, die zur Fehlerkorrektur und -anzeige bei der direkten Bearbeitung eines Einsatzes verwendet werden.

Das Plugin `schedule` enthält für die Erweiterung zum Einsatzplaneditor lediglich die Pakete `schedule.actions` und `schedule.editors`. Die Klassen dieser Pakete entsprechen in ihrer Funktion den Klassen in den entsprechenden Paketen des Plugins `device`.
Abbildung 123 zeigt die Paketstruktur als Diagramm.

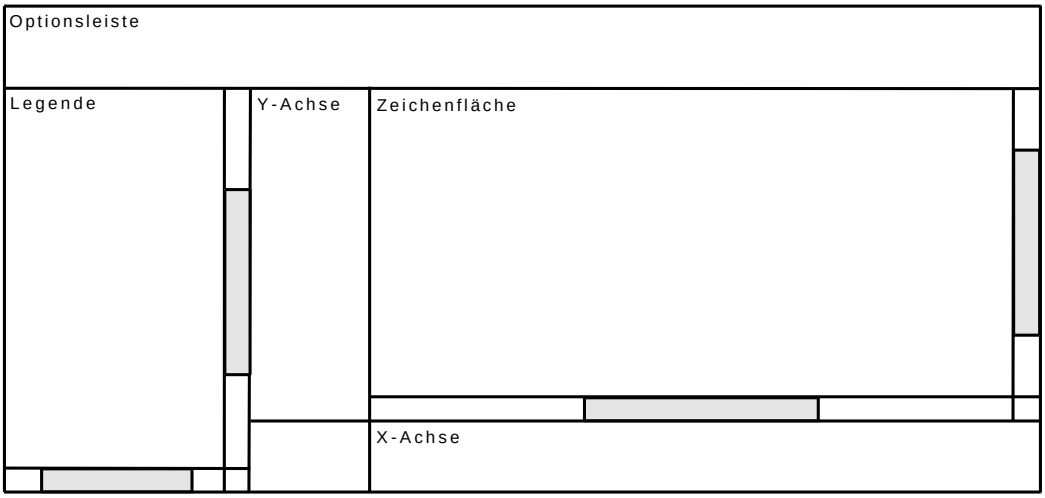

Abbildung 116: Schematischer Aufbau der grafischen Editoren

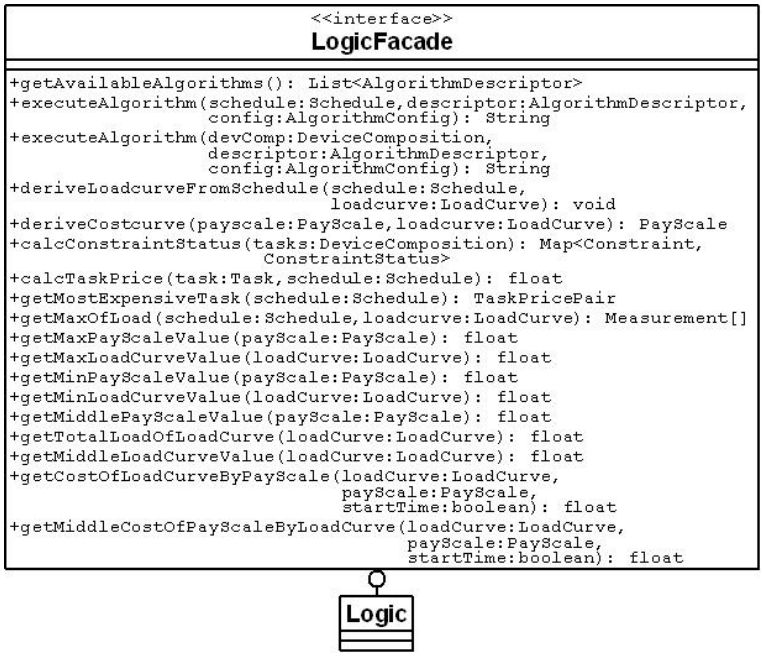

Abbildung 117: Die Schnittstelle "'LogicFacade"'

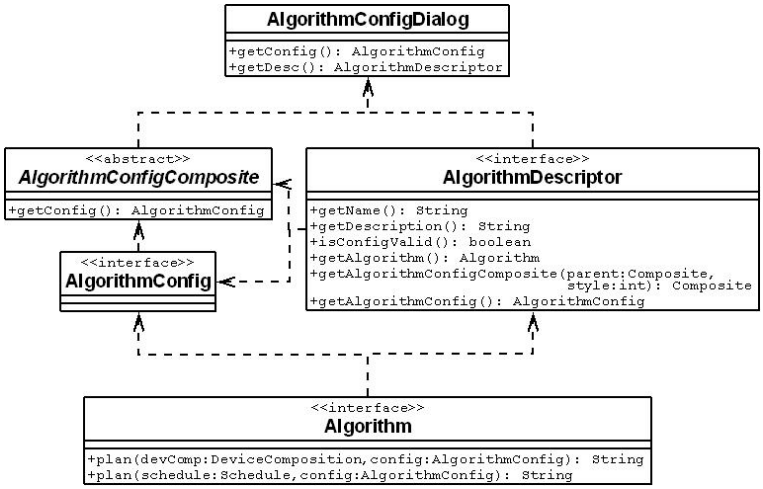

Abbildung 118: Interfaces und abstrakte Klassen als Grundgerüst für die Implementierung von Lösungsalgorithmen für das Optimierungsproblem

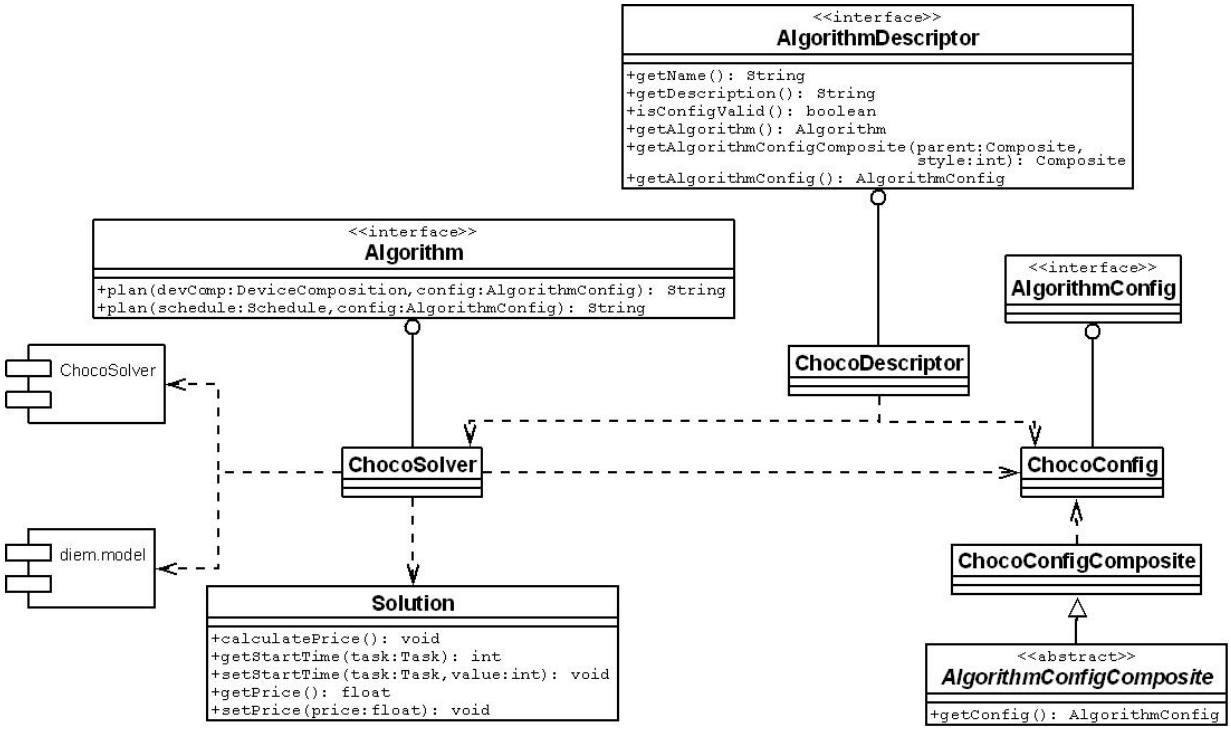

Abbildung 119: Klassen-Diagramm zur Implementierung des Choco-Solvers

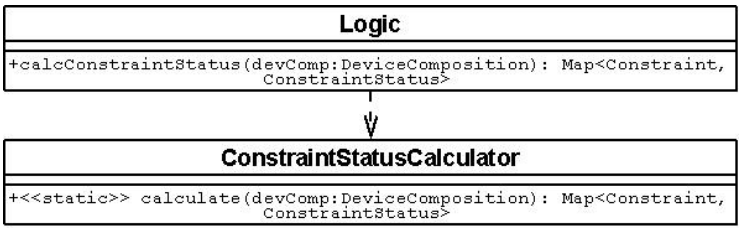

Abbildung 120: Klassen-Diagramm zur Implementierung des Algorithmus zur Zustandsüberprüfung der Constraints

Abbildung 121: Klassen-Diagramm zur Implementierung der numerischen Statistik

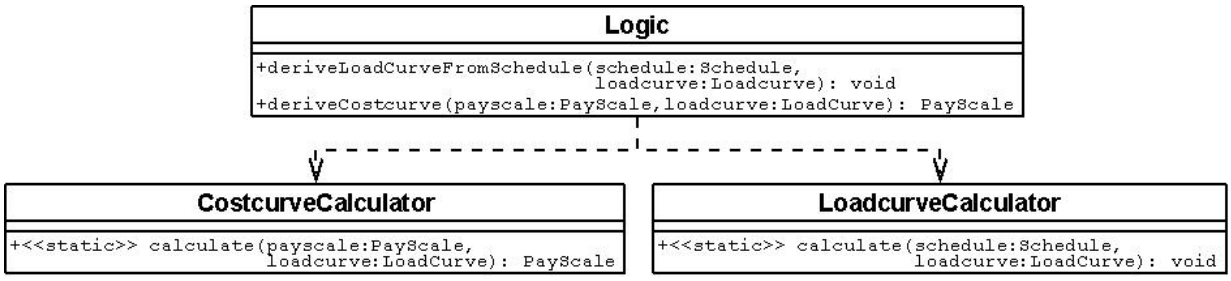

Abbildung 122: Klassen-Diagramm zur Implementierung der grapischen Statistik

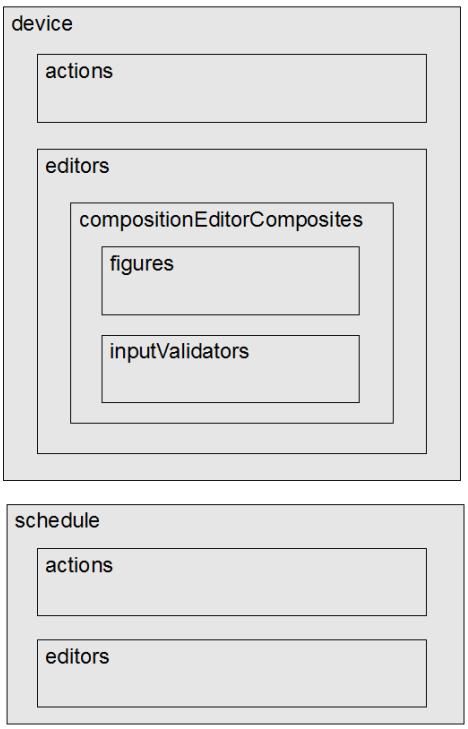

Abbildung 123: Die Paketstruktur des Einsatzplaneditors

31.3. Klassenübersicht

Nachdem die Paketstruktur erläutert wurde kann nun eine Übersicht über die Klassen in diesen Paketen erfolgen:

- `device.actions`
  - `DeviceCompositionEditorActionBarContributor`
    Wie bereits in Abschnitt 31.2 angedeutet regelt diese Klasse die Verfügbarkeit der editorspezifischen Actions. Dies erfolgt durch ein Listener-Konzept: Ändert sich der Fokus innerhalb der Anwendung auf einen Gerätekompositionseditor, so wird diese Klasse benachrichtigt und erhält die Möglichkeit, die Menüs und Toolbars der Anwendung um eigene Elemente zu erweitern.
  - `DeviceSubstitutionAction`
    Diese Klasse implementiert die Action, mit der in einem Gerätekompositionseditor der Dialog zur Gerätesubstitution gestartet wird.
- `device.editors`
  - `DeviceCompositionEditor`
    Diese Klasse implementiert den eigentlichen RCP Editor für den Gerätekompositionseditor, der die Ausgangsbasis dieser Komponente darstellt und neben der RCP und EMF spezifischen Datenverwaltung auch den grafischen Teil des Editors initialisiert.
- `device.editors.compositionEditorComposites`
  - `DeviceCompositionEditorComposite`
    Diese Klasse stellt den obersten SWT Container des Gerätekompositionseditors dar. Enthalten sind ein Bereich mit Zoom-Controls (wird innerhalb dieser Klasse erstellt und befindet sich am oberen Rand), und ein Bereich mit dem grafischen Teil des Editors (ist als eigene Klasse `DeviceCompositionEditorBodyComposite` realisiert).
  - `DeviceCompositionEditorBodyComposite`
    Diese Klasse stellt den Container für die grafischen Elemente des Gerätekompositionseditors dar. Enthalten sind zwei vertikal nebeneinander positionierte `ScrolledComposites` (Klasse aus dem SWT, welche ihren Inhalt scrollen kann, wenn er die darstellbare Größe übersteigt), von denen das linke ein `LabelColumnComposite` zur Anzeige der einzelnen Einsatzinformationen, und das rechte ein `WorkbenchComposite` für die grafische Darstellung der einzelnen Einsätze enthält. Die beiden ScrolledComposites sind durch mehrere spezifische `Listener` miteinander verbunden, sodass sie einerseits immer synchron scrollen, und andererseits, um zu gewährleisten, dass eine Benachrichtigung erfolgt, sobald über die Buttons des linken Teils eine grafische Aktualisierung des rechten Teils erforderlich wird.

Unter den `ScrolledComposites` befindet sich der `FunctionViewer`, der die Lastkurve der Gerätekomposition (bzw. Last-, Tarif- und Kostenkurve im Falle eines Einsatzplanes) visualisiert. Dieser wird allerdings separat in einem eigenen Feinentwurf beschrieben und wird deshalb hier nicht näher erläutert.

- `LabelColumnComposite`
  Diese Klasse stellt den linken Teil des Gerätekompositionseditors dar. Enthalten sind eine Menge von `LabelComposites`, welche je einen Einsatz repräsentieren.
- `LabelComposite`
  Diese Klasse stellt den Informationsteil eines Einsatzes dar. Enthalten sind der Name des Einsatzes und abhängig vom Typ des Einsatzes sein Betriebsmodus oder ein Button zum Aufklappen, sowie ein Button zum (De)Aktivieren des Serienmodus.
- `WorkbenchComposite`
  Diese Klasse stellt den Container für die rein grafischen Elemente des Gerätekompositionseditors dar. Enthalten sind eine Menge von `TaskRowFigures` für je einen verwendeten Betriebsmodus eines in der Komposition enthaltenen Gerätes. Das `WorkbenchComposite` ist die komplexeste Klasse des Gerätekompositionseditors und enthält zusätzlich drei stark integrierte interne Klassen (*nested classes*):
  * `RootLayouter`
    Dies ist ein `LayoutListener`, der bei jedem erforderlichen Layout des Editors die enthaltenen grafischen Elemente korrekt positioniert, und diese Elemente gegebenenfalls zu einem eigenen Layout ihrer internen Komponenten veranlasst.
  * `FigureMouseListener`
    Dies ist ein Listener, der auf Benutzeraktionen durch die Maus lauscht und die Interaktivität des Editors regelt. Dies reicht von der Modifikation des Mauszeigers je nach seiner Position (über einem Einsatz deutet ein Doppelkreuz beispielsweise an, dass der Einsatz verschoben werden kann) bis hin zum Ändern des Datenmodells (etwa wenn der Benutzer einen Einsatz verschiebt).
  * `MyMenuListener`
    Dieser Listener wird über den `FigureMouseListener` aktiviert, sobald der Benutzer mit der rechten Maustaste auf einen Einsatz klickt. Die Klasse ist dann für die Anzeige des Kontextmenüs zuständig, sowie für die Ausführung der vom Benutzer dort ausgewählten Aktion.

  Es fällt auf, dass diese drei internen Klassen alle nach dem Listener Konzept aufgebaut sind. Dies hat den enormen Vorteil, dass sie nicht direkt miteinander kommunizieren müssen, sondern implizit benachrichtigt werden, sobald

eine für sie relevante Aktion erfolgt ist. Prinzipiell könnten die Klassen auch eigenständig und unabhängig vom `WorkbenchComposite` implementiert werden, jedoch wäre dann etwas mehr Kommunikationsaufwand erforderlich, da die Listener auf einige Klassenvariablen des `WorkbenchComposite` angewiesen sind, welche dann als Parameter an sie übergeben werden müssten.

– `Mapper` Klassen
In diesem Paket existieren vier Mapper Klassen, die intensiv im `WorkbenchComposite` und anderen Klassen verwendet werden:

  * `EObjectMapper`
  Diese abstrakte Klasse enthält keine tatsächlichen abstrakten Vorschriften oder konkreten Methoden. Sie existiert nur aus Polymorphie-Gründen als Superklasse für die folgenden drei Klassen.

  * `OpModeMapper`
  Enthält einen `OperationalMode` sowie einen `EContainer`, der den Container desjenigen Einsatzes darstellt, in welchem der `OperationalMode` verwendet wird. Über die weiterhin implementierten Methoden `hashCode()` und `equals()` lassen sich so gleiche `OperationalModes` anhand ihres Einsatzortes innerhalb einer Gerätekomposition bzw. eines Einsatzplanes eindeutig identifizieren.
  Dieser Mapper wirft eine `IllegalArgumentException`, falls eines der beiden Objekte `null` ist, um immer eine eindeutige Identifikation zu gewährleisten.

  * `DevCompMapper`
  Enthält eine `DeviceComposition` sowie einen `EContainer`, der den Container desjenigen Einsatzes darstellt, in welchem die `DeviceComposition` verwendet wird. Über die weiterhin implementierten Methoden `hashCode()` und `equals()` lassen sich so gleiche `DeviceCompositions` anhand ihres Einsatzortes innerhalb einer Gerätekomposition bzw. eines Einsatzplanes eindeutig identifizieren.
  Dieser Mapper wirft eine `IllegalArgumentException`, falls eines der beiden Objekte `null` ist, um immer eine eindeutige Identifikation zu gewährleisten.

  * `TaskMapper`
  Die Einsätze in einer Gerätekomposition haben Zeitpunkte relativ zu dieser Gerätekomposition. Falls diese Gerätekomposition nun mehrfach in einer anderen Gerätekomposition bzw. einem Einsatzplan enthalten ist, kann bei den Einsätzen nicht mehr unterschieden werden, zu welcher Gerätekomposition sie gehören (da sie auf Codeebene gleiche Objekte mit gleichen Containern sind), und demzufolge auch nicht, wann sie beginnen sollen. Diese Klasse speichert daher neben einem Einsatz (`Task`)

aus einer Gerätekomposition zusätzlich den Einsatz (`Task`) der Gerätekomposition selbst. Somit identifiziert dieser Mapper einen `Task` über den `Task` einer Gerätekomposition, in welcher ersterer enthalten ist. Hier sind ebenfalls die Methoden `hashCode()` und `equals()` implementiert.
Dieser Mapper wirft nur dann eine `IllegalArgumentException`, falls das erste Argument `null` ist. Denn falls der zweite, identifizierende Task `null` ist, bedeutet das nur, dass der erste, zu identifizierende `Task` nicht durch eine Gerätekomposition definiert wird, sondern eigenständig verplant werden soll.

Die beschriebenen Mapper Klassen sind primär für den Einsatz als Schlüssel in `HashMaps` konzipiert. Die implementierten `equals()` Methoden verwenden zur Überprüfung der Gleichheit der enthaltenen Objekte die Methode `EcoreUtil.equals()` aus dem EMF, da diese exaktere Verfahren als die von den einzelnen Objekten implementierte `equals()` nutzt.

- `IMouseButtonNotifier`
  Dieses Interface fungiert als Brücke zwischen den Plugins `device` und `schedule`, da die Abhängigkeit zwischen diesen Plugins bereits in der Richtung "'`schedule` ist abhängig von `device`"' gesetzt ist, und ein Setzen der gleichzeitigen umgekehrten Beziehung zu einem Zyklus führen würde, mit dem die Anwendung nicht mehr startbar wäre. Die Nutzung des Interfaces beschränkt sich auf die Benachrichtigung des aktuellen `MouseButton` Zustandes, damit die implementierende Klasse darauf reagieren kann. Zur Zeit wird diese Klasse nur von `editors.ScheduleEditorComposite` aus dem Plugin `schedule` implementiert, da diese Klasse unter anderem für die Neuberechnung des Lastganges eines Einsatzplanes (`ScheduleLoadcurve`) verantwortlich ist, dies aber nur bei einem `MouseRelease` tun soll. Dieses `MouseRelease` lässt sich aber aufgrund von Listener-Einschränkungen nicht direkt in der Klasse abfragen, weshalb dies über dieses Interface von der Klasse `WorkbenchComposite` aus erfolgen muss.
- `TaskEditDialog`
  Diese Klasse repräsentiert einen Dialog, in welchem der Benutzer die Eigenschaften eines Einsatzes (wie Startzeit, Dauer etc.) direkt manipulieren kann.
- `TaskMultiplyDialog`
  Diese Klasse repräsentiert einen Dialog, durch welchen der Benutzer einen Einsatz nach seinen Vorgaben multiplizieren und so mehrfach in einer Gerätekomposition bzw. einem Einsatzplan einplanen kann.

- `device.editors.compositionEditorComposites.figures`
  - `ITaskOwner`
    Dieses Interface definiert, dass implementierende Klassen einen Task anbie-

ten müssen, den sie repräsentieren. Wird zur Zeit von `TimeFrameFigure` und `TimeOperationFigure` implementiert.

- `TaskRowFigure`
  Diese Figure repräsentiert eine Zeile im grafischen Bereich des Editors. Neben einer Zeitleiste (`TimeLineFigure`) enthält sie alle Einsätze des repräsentierten Betriebsmodus bzw. der repräsentierten Gerätekomposition in Form von jeweils einem erlaubten Ausführungszeitraum des Einsatzes (`TimeFrameFigure`) und dem konkreten Ausführungszeitraum des Einsatzes (`TimeOperationFigure`). Zudem kann sie, bei aktiviertem Serienmodus für die repräsentierte Zeile, eine Menge von Abstandshaltern enthalten (`TaskSpacerFigure`), welche für das simultane Verändern der Abstände zwischen den Einsätzen in dieser Zeile verwendet werden. Über Klassenparameter können Zoomfaktoren in X- und Y-Richtung sowie die Transparenz und Farbe des Zeilenhintergrundes konfiguriert werden.
  Wie beim `WorkbenchComposite` enthält diese Klasse ebenfalls eine interne Klasse, die für das Layout der enthaltenen Figures zuständig ist (`TaskRowLayouter`).
- `TimeLineFigure`
  Diese Klasse stellt die Zeitleiste in einer Zeile des Editors dar. Sie enthält eine horizontale Linie, auf der vertikale Balken inklusive Beschriftungen aufgetragen sind. Über Klassenparameter lässt sich einstellen, zu welchen Zeitpunkten Balken gezeichnet werden sollen (als Standard sind hier vier Balken pro Stunde definiert, wobei die Stundenbalken größer als die Halbstundenbalken, und diese größer als die Viertelstundenbalken gezeichnet werden), welche Transparenz und Farbe sie besitzen und ähnliches. Diese Klasse enthält ebenfalls einen internen Layouter für die enthaltenen Figures (`LineLayouter`).
- `TimeFrameFigure`
  Diese Klasse ist von `RoundedRectangle` aus dem Draw2D abgeleitet und repräsentiert den erlaubten Ausführungszeitraum eines Einsatzes in Form eines Rechteckes mit abgerundeten Ecken. Über Klassenparameter lassen sich die Farbe der Füllung sowie der Umrandung der Figure und ihr Transparenzwert einstellen.
- `TimeOperationFigure`
  Diese Klasse ist ebenfalls von `RoundedRectangle` aus dem Draw2D abgeleitet. Sie beschreibt den konkreten Ausführungszeitraum eines Einsatzes in Form eines abgerundeten Rechteckes. Das Rechteck enthält zudem an jeder Kante einen Konnektor (`ConnectorFigure`), an welche im Editor durch den Benutzer mit der Maus grafisch Constraints angefügt werden können. Außerdem ist die Klasse in der Lage, einen Tooltip mit Informationen zum repräsentierten Einsatz darzustellen (`TooltipFigure`). Über Klassenparameter können die Farbwerte der Füllung und der Umrandung der Figure

sowie der Konnektoren und die Transparenz der Figure eingestellt werden. Auch diese Klasse enthält einen internen Layouter für die enthaltenen Figures (`ConnectorLayouter`).

- `TooltipFigure`
  Diese Klasse ist von `Label` aus dem Draw2D abgeleitet und visualisiert den textuellen Inhalt eines Tooltips für einen Einsatz oder einen Constraint.
- `ConnectorFigure`
  Diese Klasse ist von `Ellipse` aus dem Draw2D abgeleitet und stellt einen Konnektor für Constraints in der `TimeOperationFigure` dar. Über einen Konstruktorparameter wird definiert, welche Orientierung der Konnektor haben soll (also an welcher Kante der `TimeOperationFigure` er angeheftet wird). Wird an diesen Konnektor ein Contraint angefügt, so wird diese Klasse benachrichtigt und erhält das Objekt des angehefteten Constraints sowie einen seiner beiden Anker (`ConstraintAnchor`), über welche der Constraint die Endpunkte seiner grafischen Repräsentation (`ConstraintFigure`) ermittelt. Abhängig vom Typ des angehefteten Constraints verändert sich das Aussehen dieses Konnektors wie in Abbildung 124 dargestellt (v.l.n.r.: gleicher Start, gleiches Ende, überlappend, ausschließend, nacheinander links, nacheinander rechts). Über Klassenparameter lassen sich weiterhin die Farbwerte und Transparenz der Figure konfigurieren.

Abbildung 124: Die verschiedenen Konnektorentypen

- `ConstraintAnchor`
  Diese Klasse ist von `AbstractConnectionAnchor` aus dem Draw2D abgeleitet. Sie definiert je einen Endpunkt für eine `ConstraintFigure`. Dazu wird ihr die `ConstraintFigure` übergeben, über die sie die Koordinaten des Ankers (hier: Mittelpunkt der Figure) ermittelt.
- `ConstraintFigure`
  Diese Klasse ist von `PolylineConnection` aus dem Draw2D abgeleitet. Sie ist die grafische Repräsentation eines Constraints und stellt eine Linie zwischen zwei Ankerpunkten dar. Diese Punkte werden über ihr zugewiesene `ConstraintAnchor` ermittelt. Die Farbe der Linie ergibt sich aus dem Status (erfüllt / nicht erfüllt / nicht erfüllbar) des repräsentierten Constraints. Sie enthält ebenfalls einen Tooltip durch eine `TooltipFigure`. Über Klassenvariablen lassen sich die Linienstärke sowie die Transparenz der Linie einstellen.

  - `TaskSpacerFigure`
    Diese Klasse stellt Abstandshalter zwischen zwei Einsätzen dar, welche für das simultane Verschieben von Einsätzen im Serienmodus verwendet werden. Die Klasse enthält die beiden Einsätze (`Tasks`), die sie verbindet, sowie einen Layouter für die enthaltenen Linien (`LineLayouter`). Über Klassenparameter lassen sich die Linienstärke sowie die Farbe und Transparenz regeln.

- `device.editors.compositionEditorComposites.inputValidators`
  - `StartTimeFitValidator`
    Diese Helferklasse enthält eine Methode, die eine übergebene *Startzeit* für einen Einsatz auf Gültigkeit überprüft.
  - `DurationFitValidator`
    Diese Helferklasse enthält eine Methode, die eine übergebene *Dauer* für einen Einsatz auf Gültigkeit überprüft.
  - `EarliestStartTimeFitValidator`
    Diese Helferklasse enthält eine Methode, die eine übergebene *früheste Startzeit* für einen Einsatz auf Gültigkeit überprüft.
  - `LatestRelEndTimeFitValidator`
    Diese Helferklasse enthält eine Methode, die eine übergebene *späteste relative Endzeit* für einen Einsatz auf Gültigkeit überprüft.

- `schedule.actions`
  - `ScheduleEditorActionBarContributor`
    Wie bereits in Abschnitt 31.2 angedeutet regelt diese Klasse die Verfügbarkeit der editorspezifischen Actions. Dies erfolgt durch ein Listener-Konzept: Ändert sich der Fokus innerhalb der Anwendung auf einen Einsatzplaneditor, so wird diese Klasse benachrichtigt und erhält die Möglichkeit, die Menüs und Toolbars der Anwendung um eigene Elemente zu erweitern.

- `schedule.editors`
  - `ScheduleEditor`
    Diese Klasse implementiert den eigentlichen RCP Editor für den Einsatzplaneditor, der die Ausgangsbasis dieser Komponente darstellt und neben der RCP und EMF spezifischen Datenverwaltung auch den grafischen Teil des Editors initialisiert.
  - `ScheduleEditorComposite`
    Diese Klasse stellt den obersten SWT Container des Einsatzplaneditors dar. Enthalten sind ein Bereich mit Zoom-Controls (wird innerhalb dieser Klasse erstellt und befindet sich am oberen Rand), und ein Bereich mit dem grafischen Teil des Editors (ist als eigene Klasse

`DeviceCompositionEditorBodyComposite` realisiert, jedoch überschreibt das `ScheduleEditorComposite` ein paar Methoden dieser Klasse in Form einer *anonymous nested class* aus Gründen der Erweiterung).

Da diese Fülle an Klassen und Abhängigkeiten auf textuelle Weise schwer fassbar ist, zeigt Abbildung 125 die Zusammenhänge in einer Gesamtansicht des Einsatzplaneditors anhand eines exemplarischen Screenshots der Anwendung. In der vertikalen Ebene auf der linken Seite sind die einzelnen Container aus dem SWT aufgetragen, während der eingerahmte Teil am oberen Rand alle Komponenten aus dem Draw2D beschreibt. Diese werden von dem `WorkbenchComposite` verwaltet, das seinerseits im `ScheduleEditorBodyComposite` liegt.

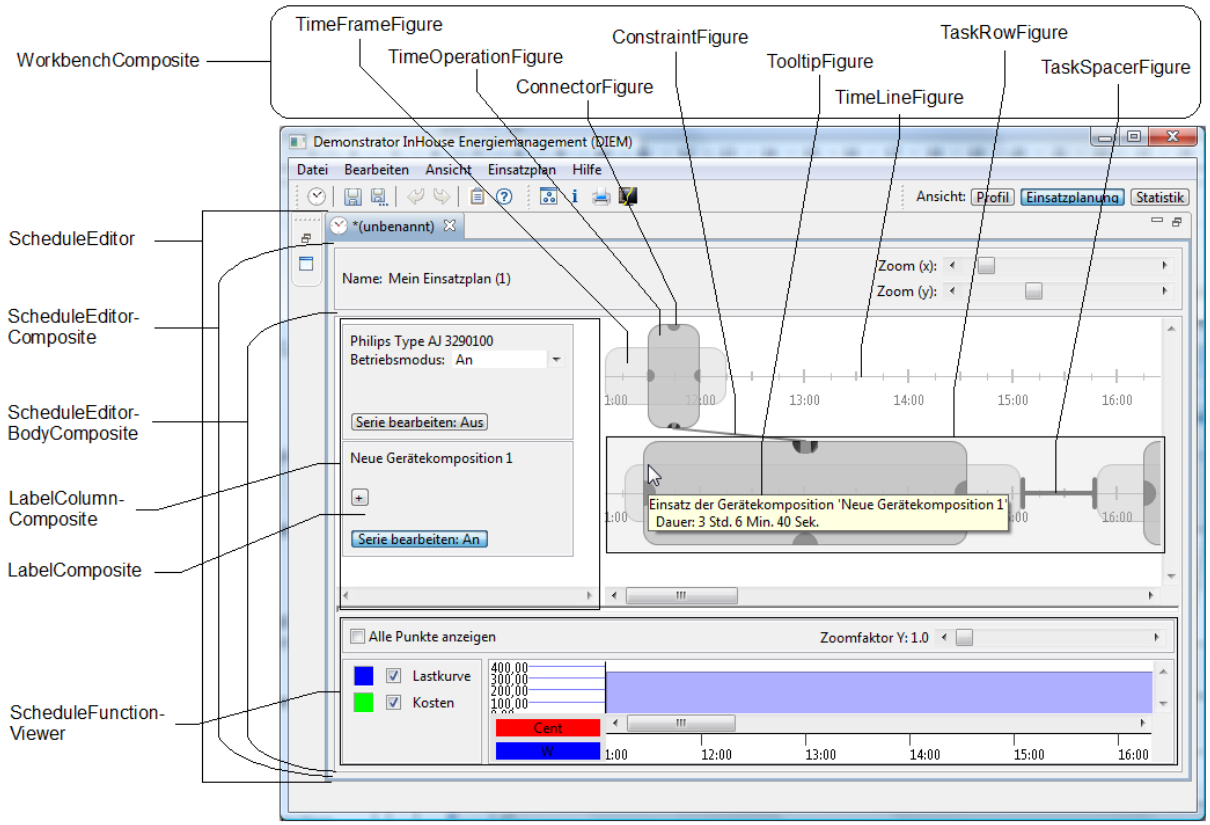

Abbildung 125: Gesamtansicht der Klassen

31.4. Kommunikation mit dem Datenmodell

Der Gerätekompositions- bzw. Einsatzplaneditor arbeitet direkt auf dem Datenmodell des EMF. Das heißt wenn der Anwender durch eine Interaktion mit dem Editor Werte modifiziert (etwa durch das Verschieben eines Einsatzes mit der Maus), so werden diese Werte direkt im Datenmodell aktualisiert. Gleichzeitig implementieren die zentralen und wichtigen Klassen des Editors das Interface `Adapter` aus dem EMF und registrieren sich im Datenmodell als Listener, sodass sie bei jeder Änderung des Models benachrichtigt werden und sich entsprechend aktualisieren. In Abbildung 126 ist die Methode `notifyChanged(Notification not)` der Klasse `TaskRowFigure` dargestellt, welche bei einer Änderung des Datenmodells aufgerufen wird.

```
public void notifyChanged(Notification not) {
  // Hack: führe GUI-Operationen Thread-sicher aus
  final Notification notification = not;
  Runnable run = new Runnable() {
    public void run() {
      switch(notification.getEventType()) {
      case Notification.SET: {
        if (notification.getNotifier() instanceof Task) {
          doLayout();
        }
        break;
      }
      }
    }
  };
  if(isEnableGlobalModelAdapter()) {
    if (Display.getDefault().getThread() != Thread.currentThread()) {
      Display.getDefault().syncExec(run);
    } else {
      run.run();
    }
  }
}
```

Abbildung 126: Quelltext des `Adapter` Interfaces

Das übergebene `Notification` Objekt enthält Informationen zur erfolgten Änderung, anhand derer die Anwendung entscheiden kann, welche Schritte einzuleiten sind. Im dargestellten Beispiel wird zunächst der Typ der Änderung überprüft. Für den Fall, dass ein Attribut eines Objektes verändert wurde (`Notification.SET`), und dieses Objekt von Typ `Task` ist, soll ein Layout durchgeführt werden.
Bemerkenswert ist an dem dargestellten Quelltext, dass die beschriebenen Aktionen

nicht unmittelbar durchgeführt werden, sondern in einem `Runnable` Objekt gekapselt sind, welches im Anschluss ausgeführt wird. Diese Vorgehensweise wurde durch den *Job* Mechanismus des RCP Frameworks notwendig: Wird eine automatische Optimierung eines Einsatzplanes durchgeführt, so läuft diese durch den *Job* Mechanismus in einem eigenen *Thread*. Während der Optimierung werden häufig die Datenwerte der Einsätze des Einsatzplanes verändert. Bei jeder Änderung werden somit die Listener des Datenmodells benachrichtigt, welche dann die erforderlichen Aktualisierungen durchführen. Jedoch dürfen Änderungen an den grafischen Komponenten nur vom sogenannten *Display Thread* durchgeführt werden, ansonsten kommt es zu einer Exception. Daher muss in jeder `notifyChanged(Notification not)` geprüft werden, ob der aktive *Thread* berechtigt ist, ein Layout der grafischen Komponenten durchzuführen, um im Zweifelsfall den korrekten *Thread* anzustoßen.

31.5. Die Klasse `TimeOperationFigure`

In diesem Abschnitt wird die Klasse `TimeOperationFigure` in ihren Einzelheiten erläutert. Abbildung 127 zeigt das zugehörige Klassendiagramm.

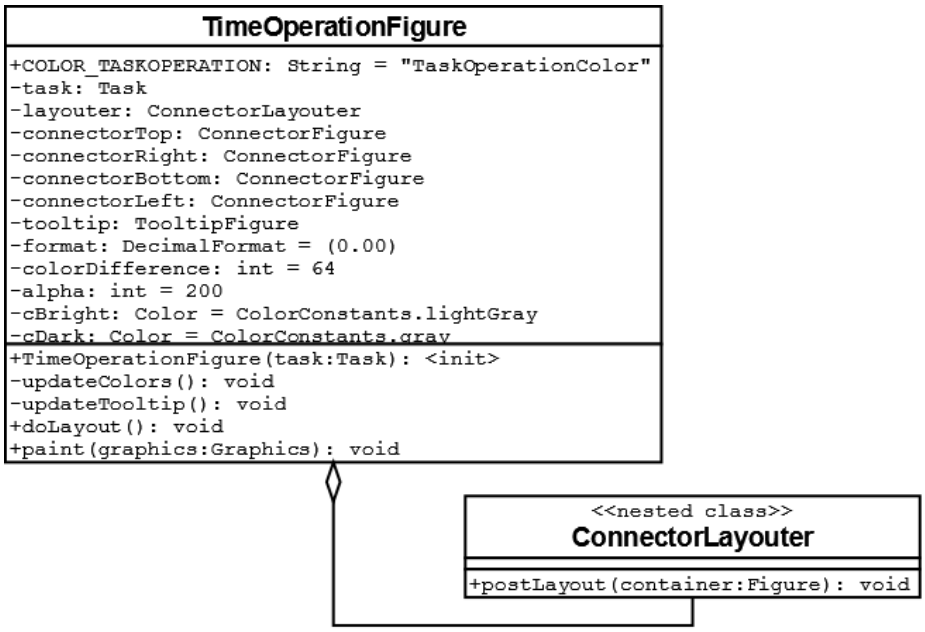

Abbildung 127: Klassendiagramm zur TimeOperationFigure

31.5.1. Attribute

In der Klasse `TimeOperationFigure` sind 13 Attribute enthalten, die im Folgenden erläutert werden:

- `String COLOR_TASKOPERATION`
  Dies ist ein statischer String-Wert, der den Schlüssel zum Auslesen der vom Benutzer definierten Farbwerte für diese Klasse enthält. Das Auslesen geschieht mittels des *Preference* Mechanismus aus dem Eclipse RCP Framework.

- `Task task`
  Eine `TimeOperationFigure` repräsentiert den konkreten Ausführungszeitraum eines Einsatzes. Dies ist das Objekt des Einsatzes, auf das sich die Instanz dieser Klasse bezieht.

- `ConnectorLayouter layouter`
  Wie bereits in Abschnitt 31.3 erwähnt, ist der `ConnectorLayouter` eine innere Klasse (*nested class*), die für das Layout der enthaltenen `ConnectorFigures` zuständig ist.

- `ConnectorFigure connectorTop`
  Dies ist der Konnektor am oberen Rand der `Figure`, an welche Constraints angeheftet werden können.

- `ConnectorFigure connectorRight`
  Dies ist der Konnektor am rechten Rand der `Figure`, an welche Constraints angeheftet werden können.

- `ConnectorFigure connectorBottom`
  Dies ist der Konnektor am unteren Rand der `Figure`, an welche Constraints angeheftet werden können.

- `ConnectorFigure connectorLeft`
  Dies ist der Konnektor am linken Rand der `Figure`, an welche Constraints angeheftet werden können.

- `TooltipFigure tooltip`
  Referenziert eine Instanz der `TooltipFigure` zur Darstellung eines Tooltips, wenn der Benutzer mit der Maus über dieser `Figure` schwebt.

- `DecimalFormat format`
  Formatiert den Zahlenwert der aktuellen Kosten dieses Einsatzes in einen sinnvollen String zur Anzeige im Tooltip.

- `int colorDifference`
  Den Konnektoren kann vom Benutzer keine eigene Farbe zugeordnet werden. Die Farbe ergibt sich aus der Hintergrundfarbe der `TimeOperationFigure`, indem von jedem Farbkanal der Hintergrundfarbe ein Wert abgezogen wird, sodass die Konnektoren in der gleichen Farbe, jedoch etwas dunkler gezeichnet werden. Dieses Attribut enthält den abzuziehenden Wert und damit die Helligkeitsdifferenz zwischen den Konnektoren und der Hintergrundfarbe.

- `int alpha`
  Dieser Wert bezeichnet die Transparenz dieser `Figure`. Dabei bedeutet ein Wert von 255 gar keine Differenz, während der Wert 0 vollständige Transparenz bewirkt.

- `Color cBright`
  Farbe der Füllung der `Figure` (Hintergrundfarbe).

- `Color cDark`
  Farbe der Umrandung der `Figure` (Vordergrundfarbe).

31.5.2. Methoden

In der Klasse `TimeOperationFigure` sind die folgenden nicht-trivialen Methoden enthalten:

- `public TimeOperationFigure(Task task)`
  Dies ist der Konstruktor der Klasse. Er arbeitet folgendermaßen:
  1. Weise übergebenen Task an das Klassenattribut zu.
  2. Setze einige Draw2D spezifischen Einstellungen, wie etwa das Aktivieren des Zeichnens der Hintergrundfläche.
  3. Initialisiere die `TooltipFigure` sowie die vier `ConnectorFigures`.
  4. Melde einen *PropertyChangeListener* für die vom Benutzer definierten Farben dieser Klasse mittels des im Attribut `COLOR_TASKOPERATION` hinterlegten Schlüssels im RCP Kontext an, um benachrichtigt zu werden, falls der Benutzer die Farbwerte zur Laufzeit ändert.
  5. Rufe die Methode `updateColors()` auf, um die aktuell in den RCP *Preferences* gespeicherten Farbwerte auszulesen und in den Attributen `cBright` und `cDark` zu speichern.
  6. Initialisiere `ConnectorLayouter` und melde als `LayoutListener` an, damit er bei jedem von der übergeordneten `Figure` initiierten Layout benachrichtigt wird und die Konnektoren aktualisiert.

- `private void updateColors()`
  Diese Methode liest die aktuell in den RCP *Preferences* gespeicherten Farbwerte aus, speichert sie in den Attributen `cBright` und `cDark`, weist sie der `Figure` sowie den Konnektoren zu und veranlasst eine Neuzeichnung.

- `private void updateTooltip()`
  Aktualisiert den Text der `TooltipFigure` mit den Werten des Klassenattributes `task`.

- `public void doLayout()`
  Veranlasst den `ConnectorLayouter`, ein Layout durchzuführen. Diese Methode ist für manuelle Aktualisierungsaufrufe gedacht.

- `public void paint(Graphics graphics)`
  Diese Methode existiert bereits in der Superklasse `Figure` und wurde an dieser Stelle lediglich zur Anwendung des Transparenz-Attributes im übergebenen `Graphics` Objekt überschrieben, da die `Figure` Klassen des Draw2D keine Methode zum Setzen des Transparenzwertes besitzen.

31.5.3. Innere Klasse `ConnectorLayouter`

Die private Klasse `ConnectorLayouter` implementiert das Interface `LayoutListener` aus dem Draw2D, verwendet jedoch nur die Methode `postLayout(IFigure container)`. Diese Methode sorgt für die Aktualisierung der Konnektoren und arbeitet folgendermaßen:

1. Initialisiere Arbeitsvariablen.
2. Überprüfe Werte der Arbeitsvariablen auf Fehler (z.B. negative Zahlen) und korrigiere sie falls notwendig.
3. Berechne den maximalen Durchmesser für die Konnektoren anhand der Breite und Höhe der `TimeOperationFigure` mit der Formel $min(width, height)/4$.
4. Berechne und setze die Position und Größe der einzelnen Konnektoren.
5. Aktualisiere den Tooltip durch Aufrufen der Methode `updateTooltip`.

31.6. Abschlussbetrachtung

Die Klasse `TimeOperationFigure` besteht aus knapp 300 Zeilen und ist damit eine der kleineren Klassen des Einsatzplaneditors. Sie wurde in dieser Beschreibung *exemplarisch* vorgestellt, da sich die enthaltenen Konzepte in Form von Attributen und Methoden nahezu direkt auf die anderen Klassen aus dem Draw2D Kontext (im Paket `figures`) abbilden lassen.
Die Klassen aus dem SWT Kontext werden hier nicht weiter vorgestellt, da sie entweder zu trivial (wie etwa `DeviceCompositionEditorComposite` – sie enthält nur standardisierte Elemente, wie sie in jeder SWT Anwendung vorkommen) oder zu komplex sind und damit den Rahmen dieses Dokumentes sprengen würden (die Klasse `WorkbenchComposite` etwa besteht aus knapp 2000 Zeilen). Diese sind jedoch mit JavaDoc- sowie Inline-Kommentaren versehen, um ihre Arbeitsweise direkt am Code zu erläutern.

Abschließend ist anzumerken, dass die Entwicklung des Gerätekompositions- bzw. Einsatzplaneditors trotz des Fehlschlags bezüglich des GEF Frameworks sehr strukturiert durchgeführt worden ist, wenn auch insbesondere die Implementierung der Interaktivität in der privaten Klasse `FigureMouseListener` im `WorkbenchComposite` dadurch viel mehr Handarbeit erforderte. Das Zusammenspiel des Editors mit anderen Komponenten der Anwendung konnte durch die regelmäßige Kommunikation der Projektgruppenmitglieder einfach und schnell implementiert werden. Aufgetretene Fehler wurden durch das modulare Konzept sofort eingegrenzt und konnten zügig behoben werden. Ebenso profitiert die Erweiterbarkeit von der starken Modularität, was sich schon während der Entwicklung bemerkbar machte, da neue Ideen und Wünsche der Mitglieder sowie Betreuer ohne viel Aufwand eingebaut werden konnten.

Teil VII.
Testen

Autor: Mark Eilers

Um sicherzustellen, dass die entwickelte Anwendung fehlerfrei und korrekt arbeitet, wurde sie im Laufe der Implementierung verschiedenen Tests unterzogen. Diese lassen sich dabei grundsätzlich in Systemtests der gesamten Anwendung und Unit-Tests augewählter Klassen und Methoden unterteilen.

32. Systemtests

Während der gesamten Entwicklung der Anwendung waren umfangreiche Systemtests ein fester Bestandteil der Implementation. Jede neu entwickelte Funktionalität wurde schon während der Implementation auf ihre Korrektheit überprüft. Da die Anwendung außerdem schon frühzeitig verwendet wurde, beispielsweise für die Erstellung der Bibliotheken, konnten viele Fehler in einem frühen Stadium der Entwicklung gefunden werden. Für jeden aufgetretenen Fehler wurde dann durch das Ticket-System in Trac ein Fehlerticket erstellt und der Fehler möglichst aussagekräftig protokolliert. Da über das Trac jeder noch nicht behobene Fehler jederzeit für alle Beteiligten der Projektgruppe sichtbar und damit allen bekannt war, konnte sichergestellt werden, dass jeder gemeldete Fehler auch behoben wurde.
Für jeden abgeschlossenen Prototypen wurde die Anwendung in vollem Umfang hinsichtlich ihrer Funktionstüchtigkeit geprüft. Für die jeweiligen Präsentationen wurden zusätzlich Szenarien entworfen, die jeden Aspekt der Anwendung aus Sicht des Anwenders simulieren. Die letzten vier Wochen des Zeitplans waren dann ausschließlich dem Testen und Bugfixing vorbehalten. Zusätzliche Systemtests wurden während der Anfertigung des Benutzerhandbuchs und des Tutorials durchgeführt. Alle dort erwähnten Beschreibungen sind zuvor hinsichtlich ihrer Funktionalität getestet worden.

33. Unit-Tests

Einige der Klassen und Methoden, die nicht direkt über die grafische Benuteroberfläche der Anwendung überprüfbar waren, wurden im Anschluss an die Systemtests durch verschiedene Unit-Tests bezüglich einer korrekten und fehlerfreien Funktionalität getestet. Alle Unit-Tests wurden durch das Framework *JUnit* in der Version 4.0 implementiert.

33.1. Tests des Pakets `diem.logic`

In dem Paket `diem.logic` befinden sich viele Algorithmen, die Werte aus Tarifen, Lastgängen oder Einsatzplänen berechnen sollen. Ein alleiniges Testen dieser Algorithmen innerhalb von Systemtests ist leider nicht immer ausreichend um ihre Korrektheit zu gewährleisten. Zum einen können beim manuellen Testen nur wenige der theoretisch möglichen Situationen abgedeckt werden, zum anderen ist die Korrektheit des Ergebnisses nur für sehr einfache Fälle nachvollziehbar. Um dennoch eine fehlerfreie Funktionsweise sicherzustellen wurden für jede der im Folgenden angegebenen Methoden im Verlauf des Testens mehrere JUnit-Tests erstellt. Jeder JUnit-Test besteht dabei aus einer einzelnen Methode, die einen möglichen Testfall repräsentiert. Diese sind so gewählt, dass sie insbesondere die möglichen Grenzfälle und Ausnahmen abdecken, die innerhalb der Systemtests nicht abgedeckt werden konnten.

33.1.1. `deriveLoadcurveFromSchedule(Schedule, LoadCurve)`

Die Methode `deriveLoadcurveFromSchedule(Schedule, LoadCurve)` berechnet einen Lastgang aus einem Einsatzplan. In den dazugehörigen JUnit-Tests wurde die Methode auf verschiedene Testfälle hin überprüft:

- `emptyScheduleTest()`
  In dem Testfall `emptyScheduleTest()` wird überprüft ob die zu überprüfende Methode bei einem Einsatzplan ohne Einsätze einen Lastgang ohne Verbrauch zurückgibt.

- `normalScheduleTest()`
  In dem Testfall `normalScheduleTest()` wird exemplarisch überprüft ob bei regulären Einsatzplänen der richtige Lastgang zurückgegeben wird.

- `taskBeyondDurationScheduleTest()`
  In dem Testfall `taskBeyondDurationScheduleTest()` wird überprüft ob die Methode mit Einsätzen über der regulären Länge des Einsatzplans zurechtkommt.

- `unsetDurationScheduleTest()`
  Der Testfall `unsetDurationScheduleTest()` überprüft die Methode auf den Umgang von nichtregulären Einsatzplänen mit undefinierter Länge.

33.1.2. `getCostOfLoadCurveByPayScale(LoadCurve, PayScale, boolean)`

Die Methode `getCostOfLoadCurveByPayScale(LoadCurve, PayScale, boolean)` berechnet die Kosten, die ein Lastgang für einen übergebenen Tarif verursacht. Durch Angabe eines boolschen Werts kann definiert werden, ob die Gesamtkosten anhand der realen oder einer standardisierten Startzeit berechnet werden. In den dazugehörigen JUnit-Tests wurde die Methode auf verschiedene Testfälle hin überprüft:

- `emptyLoadCurveEmptyPayScale()`
  In dem Testfall `emptyLoadCurveEmptyPayScale()` wird überprüft ob bei leerem Lastgang und leerem Tarif der richtige Fehlercode zurückgegeben wird.

- `emptyLoadCurveNormalPayScale()`
  In dem Testfall `emptyLoadCurveNormalPayScale()` wird überprüft ob bei leerem Lastgang und gültigem Tarif der richtige Fehlercode zurückgegeben wird.

- `noMatchOfLoadCurveAndPayScale()`
  In dem Testfall `noMatchOfLoadCurveAndPayScale()` wird überprüft ob der richtige Fehlercode zurückgegeben wird, wenn sich ein Lastgang und ein Tarif zeitlich nicht überschneiden.

- `normalLoadCurveAndFixPayScaleNoStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei standardisierter Startzeit, gültigem Lastgang und einem Fixtarif der richtige Wert berechnet wird.

- `normalLoadCurveAndFixPayScaleStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei berücksichtigter Startzeit, gültigem Lastgang und einem Fixtarif der richtige Wert berechnet wird.

- `normalLoadCurveAndPayScaleNoStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei standardisierter Startzeit, güligem Lastgang und einem variablen Tarif der richtige Wert berechnet wird.

- `normalLoadCurveAndPayScaleStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei berücksichtigter Startzeit, güligem Lastgang und einem variablen Tarif der richtige Wert berechnet wird.

- `normalLoadCurveEmptyPayScale()`
  In dem Testfall `normalLoadCurveEmptyPayScale()` wird überprüft ob bei gültigem Lastgang und leerem Tarif der richtige Fehlercode zurückgegeben wird.

- `payScaleTooShort()`
  In dem Testfall `payScaleTooShort()` wird überprüft ob der richtige Fehlercode zurückgegeben wird, wenn der Tarif nicht lang genug für den übergebenen Lastgang ist.

33.1.3. `getMaxOfLoad(Schedule, LoadCurve)`

Die Methode `getMaxOfLoad(Schedule, LoadCurve)` berechnet die Zeitdauer und die maximale Last eines Einsatzplans. In den dazugehörigen JUnit-Tests wurde die Methode auf verschiedene Testfälle hin überprüft:

- `emptyScheduleTest()`
  In dem Testfall `emptyScheduleTest()` wird der Umgang der Methode mit einem leeren Einsatzplans getestet.
- `normalTaskTest()` In dem Testfall `normalTaskTest()` wird exemplarisch überprüft ob für reguläre Einsatzpläne die richtigen Ergebnisse berechnet werden.
- `twoEqualTasksTest()` In dem Testfall `twoEqualTasksTest()` wird überprüft ob die richtigen Werte berechnet werden, wenn der Einsatzplan zwei Tasks mit gleichgroßer Last enthält.

33.1.4. `getMiddleCostOfPayScaleByLoadCurve(LoadCurve, PayScale, boolean)`

Die Methode `getMiddleCostOfPayScaleByLoadCurve(LoadCurve, PayScale, boolean)` berechnet die durchschnittlichen Kosten eines Tarifs für die Zeitdauer die sich mit dem Lastgang überschneidet. Durch Angabe eines boolschen Werts kann definiert werden, ob die durchschnittlichen Kosten anhand der realen oder einer standardisierten Startzeit berechnet werden. In den dazugehörigen JUnit-Tests wurde die Methode auf verschiedene Testfälle hin überprüft:

- `emptyLoadCurveEmptyPayScale()`
  In dem Testfall `emptyLoadCurveEmptyPayScale()` wird überprüft ob bei leerem Lastgang und leerem Tarif der richtige Fehlercode zurückgegeben wird.
- `emptyLoadCurveNormalPayScale()`
  In dem Testfall `emptyLoadCurveNormalPayScale()` wird überprüft ob bei leerem Lastgang und gültigem Tarif der richtige Fehlercode zurückgegeben wird.
- `noMatchOfLoadCurveAndPayScale()`
  In dem Testfall `noMatchOfLoadCurveAndPayScale()` wird überprüft ob der richtige Fehlercode zurückgegeben wird, wenn sich ein Lastgang und ein Tarif zeitlich nicht überschneiden.
- `normalLoadCurveAndFixPayScaleNoStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei standardisierter Startzeit, gültigem Lastgang und einem Fixtarif der richtige Wert berechnet wird.

- `normalLoadCurveAndFixPayScaleStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei berücksichtigter Startzeit, gültigem Lastgang und einem Fixtarif der richtige Wert berechnet wird.

- `normalLoadCurveAndPayScaleNoStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei standardisierter Startzeit, güligem Lastgang und einem variablen Tarif der richtige Wert berechnet wird.

- `normalLoadCurveAndPayScaleStartTime()`
  In dem Testfall `normalLoadCurveAndFixPayScaleNoStartTime()` wird überprüft ob bei berücksichtigter Startzeit, güligem Lastgang und einem variablen Tarif der richtige Wert berechnet wird.

- `normalLoadCurveEmptyPayScale()`
  In dem Testfall `normalLoadCurveEmptyPayScale()` wird überprüft ob bei gültigem Lastgang und leerem Tarif der richtige Fehlercode zurückgegeben wird.

- `payScaleTooShort()`
  In dem Testfall `payScaleTooShort()` wird überprüft ob der richtige Fehlercode zurückgegeben wird, wenn der Tarif nicht lang genug für den übergebenen Lastgang ist.

33.1.5. `getMiddleLoadCurveValue(LoadCurve)`

Die Methode `getMiddleLoadCurveValue(LoadCurve)` berechnet den Durchschnitt eines Lastgangs. In den dazugehörigen JUnit-Tests wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyLoadCurveTest()`
  Mit dem Testfall `emptyLoadCurveTest()` wird überprüft, ob die Methode auch einen leeren Lastgang fehlerfrei bearbeiten kann.

- `normalLoadCurveTest()`
  In dem Testfall `normalLoadCurveTest()` wird exemplarisch überprüft ob für reguläre Lastgänge die richtigen Ergebnisse berechnet werden.

- `oneValueTimeNullTest()`
  Der Testfall `oneValueTimeNullTest()` überprüft wie die Methode auf einen Lastgang reagiert, der aus nur einem Wert mit Zeitpunkt 0 besteht.

- `twoValuesSameTimeTest()`
  In dem Testfall `twoValuesSameTimeTest()` wird exemplarisch überprüft ob für Lastgänge mit zeitglich gleichen Wertepaaren die richtigen Ergebnisse berechnet werden.

33.1.6. `getMiddlePayScaleValue(PayScale)`

Die Methode `getMiddlePayScaleValue(PayScale)` berechnet den Durchschnitt eines Tarifs. In den dazugehörigen JUnit-Tests wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyPayScaleTest()`
  Mit dem Testfall `emptyPayScaleTest()` wird überprüft, ob die Methode auch einen Tarif ohne Wertepaare fehlerfrei bearbeiten kann.
- `normalPayScaleTest()`
  In dem Testfall `normalPayScaleTest()` wird exemplarisch überprüft ob für reguläre Tarife die richtigen Ergebnisse berechnet werden.
- `twoValuesSameTimeTest()`
  In dem Testfall `twoValuesSameTimeTest()` wird exemplarisch überprüft ob für Tarife mit zeitlich gleichen Wertepaaren die richtigen Ergebnisse berechnet werden.
- `unsetDurationTest()`
  Der Testfall `unsetDurationTest()` überprüft wie die Methode auf einen Tarif mit undefinierter Länge reagiert.

33.1.7. `getMaxPayScaleValue(PayScale)`

Die Methode `getMaxPayScaleValue(PayScale)` berechnet den Maximalwert eines Tarifs. In dem dazugehörigen JUnit-Test wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyTest()`
  Mit dem Testfall `emptyTest()` wird überprüft, ob die Methode auch einen Tarif ohne Wertepaare fehlerfrei bearbeiten kann.
- `maxTest()`
  In dem Testfall `maxTest()` wird exemplarisch überprüft ob für reguläre Tarife die richtigen Ergebnisse berechnet werden.
- `equivalentMaxTest()`
  In dem Testfall `equivalentMaxTest()` wird exemplarisch überprüft ob für Tarife mit zwei gleichen Maximalwerten die richtigen Ergebnisse berechnet werden.
- `nullTest()`
  Der Testfall `nullTest()` überprüft ob die Methode mit einem Tarif nur mit 0 als Maximalwert richtig funktioniert.

33.1.8. `getMaxLoadCurveValue(LoadCurve)`

Die Methode `getMaxLoadCurveValue(LoadCurve)` berechnet den Maximalwert eines Tarifs. In dem dazugehörigen JUnit-Test wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyTest()`
  Mit dem Testfall `emptyTest()` wird überprüft, ob die Methode auch einen Lastgang ohne Wertepaare fehlerfrei bearbeiten kann.
- `maxTest()`
  In dem Testfall `maxTest()` wird exemplarisch überprüft ob für reguläre Lastgänge die richtigen Ergebnisse berechnet werden.
- `equivalentMaxTest()`
  In dem Testfall `equivalentMaxTest()` wird exemplarisch überprüft ob für Lastgänge mit zwei gleichen Maximalwerten die richtigen Ergebnisse berechnet werden.
- `nullTest()`
  Der Testfall `nullTest()` überprüft ob die Methode mit einem Lastgang nur mit 0 als Maximalwert richtig funktioniert.

33.1.9. `getMinPayScaleValue(PayScale)`

Die Methode `getMinPayScaleValue(PayScale)` berechnet den Minimalwert eines Tarifs. In dem dazugehörigen JUnit-Test wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyTest()`
  Mit dem Testfall `emptyTest()` wird überprüft, ob die Methode auch einen Tarif ohne Wertepaare fehlerfrei bearbeiten kann.
- `minTest()`
  In dem Testfall `minTest()` wird exemplarisch überprüft ob für reguläre Tarife die richtigen Ergebnisse berechnet werden.
- `equivalentMinTest()`
  In dem Testfall `equivalentMinTest()` wird exemplarisch überprüft ob für Tarife mit zwei gleichen Minimalwerten die richtigen Ergebnisse berechnet werden.

33.1.10. `getMinLoadCurveValue(LoadCurve)`

Die Methode `getMinLoadCurveValue(LoadCurve)` berechnet den Minimalwert eines Tarifs. In dem dazugehörigen JUnit-Test wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyTest()`
  Mit dem Testfall `emptyTest()` wird überprüft, ob die Methode auch einen Lastgang ohne Wertepaare fehlerfrei bearbeiten kann.

- `minTest()`
  In dem Testfall `minTest()` wird exemplarisch überprüft ob für reguläre Lastgänge die richtigen Ergebnisse berechnet werden.

- `equivalentMinTest()`
  In dem Testfall `equivalentMinTest()` wird exemplarisch überprüft ob für Lastgänge mit zwei gleichen Minimalwerten die richtigen Ergebnisse berechnet werden.

33.1.11. `getMostExpensiveTask(Schedule)`

Die Methode `getMostExpensiveTask(Schedule)` ermittelt den teuersten Einsatz im übergebenen Einsatzplan. Damit die Kosten eines Einsatzes ermittelt werden können ist es notwendig, dass sie von einem Tarif abgedeckt sind. In den dazugehörigen JUnit-Tests wurde die Methode bezüglich der folgenden Testfälle getestet:

- `testGetMostExpensiveTaskBothInvalid()`
  Mit dem Testfall `testGetMostExpensiveTaskBothInvalid()` wird überprüft ob die Methode den richtigen Wert zurückgibt, wenn alle Einsätze außerhalb von Tarifen liegen.

- `testGetMostExpensiveTaskEqual()`
  Mit dem Testfall `testGetMostExpensiveTaskEqual()` wird überprüft ob die Methode den richtigen Wert zurückgibt, wenn zwei Einsätze die gleichen Kosten verursachen.

- `testGetMostExpensiveTaskFirst()`
  Mit dem Testfall `testGetMostExpensiveTaskFirst()` wird getestet ob die Methode tatsächlich den Einsatz mit den höchsten Kosten zurückgibt.

- `testGetMostExpensiveTaskFirstInvalid()`
  Mit dem Testfall `testGetMostExpensiveTaskFirstInvalid()` wird getestet ob die Methode tatsächlich den Einsatz mit den höchsten Kosten zurückgibt, wenn der Einsatz mit dem höchsten Verbrauch nicht von einem Tarif abgedeckt ist.

- `testGetMostExpensiveTaskSecond()`
  Mit dem Testfall `testGetMostExpensiveTaskSecond()` wird getestet ob die Methode tatsächlich den Einsatz mit den höchsten Kosten zurückgibt.

- `testGetMostExpensiveTaskSecondInvalid()`
  Mit dem Testfall `testGetMostExpensiveTaskSecondInvalid()` wird getestet ob die Methode tatsächlich den Einsatz mit den höchsten Kosten zurückgibt, wenn der Einsatz mit dem höchsten Verbrauch nicht von einem Tarif abgedeckt ist.

33.1.12. `getTotalLoadOfLoadCurve(LoadCurve)`

Die Methode `getTotalLoadOfLoadCurve(LoadCurve)` berechnet den Gesamtverbrauch eines Lastgangs. In den dazugehörigen JUnit-Tests wurde die Methode bezüglich der folgenden Testfälle getestet:

- `emptyLoadCurveTest()`
  Mit dem Testfall `emptyLoadCurveTest()` wird überprüft ob die Methode auch einen leeren Lastgang fehlerfrei bearbeiten kann.

- `normalLoadCurveTest()`
  In dem Testfall `normalLoadCurveTest()` wird exemplarisch überprüft ob für reguläre Lastgänge die richtigen Ergebnisse berechnet werden.

- `oneValueTimeNotNullTest()` Der Testfall `oneValueTimeNotNullTest()` überprüft die Methode ob sie für einen Lastgang mit einem einzelnen Wert das richtige Ergebniss zurückgibt.

- `oneValueTimeNullTest()` Der Testfall `oneValueTimeNotNullTest()` überprüft die Methode ob sie für einen Lastgang mit einem einzelnen Wert das richtige Ergebniss zurückgibt, wenn der Zeitpunkt dieses Wertes 0 ist.

- `twoValuesSameTimeTest()`
  Der Testfall `oneValueTimeNotNullTest()` überprüft die Methode ob sie das richtige Ergebniss zurückgibt, wenn zwei Wertepaare zum gleichen Zeitpunkt exitieren.

33.1.13. `calcConstraintStatus(DeviceComposition)`

Die Methode `calcConstraintStatus(DeviceComposition)` überprüft die Status aller Constraints in einem Einsatzplan oder einer Gerätekomposition. In den dazugehörigen JUnit-Tests wurde die Methode bezüglich der folgenden Testfälle getestet:

- `testCalcConstraintStatusAfterEachOther()`
  In dem Testfall `testCalcConstraintStatusAfterEachOther()` wird überprüft ob die Methode bei einem gegebenen *AfterEachOther*-Constraint die möglichen Status *Valid*, *Unsatisfied* und *Invalid* richtig erkennt.

- `testCalcConstraintStatusExclude()`
  In dem Testfall `testCalcConstraintStatusExclude()` wird überprüft ob die Methode bei einem gegebenen *Exclude*-Constraint die möglichen Status *Valid*, *Unsatisfied* und *Invalid* richtig erkennt.

- `testCalcConstraintStatusIdentical()`
  In dem Testfall `testCalcConstraintStatusIdentical()` wird überprüft ob die Methode bei einem gegebenen *Identical*-Constraint die möglichen Status *Valid*, *Unsatisfied* und *Invalid* richtig erkennt.

- `testCalcConstraintStatusSameEnd()`
  In dem Testfall `testCalcConstraintStatusSameEnd()` wird überprüft ob die Methode bei einem gegebenen *SameEnd*-Constraint die möglichen Status *Valid*, *Unsatisfied* und *Invalid* richtig erkennt.

- `testCalcConstraintStatusSameStart()`
  In dem Testfall `testCalcConstraintStatusSameStart()` wird überprüft ob die Methode bei einem gegebenen *SameStart*-Constraint die möglichen Status *Valid*, *Unsatisfied* und *Invalid* richtig erkennt.

33.1.14. `calcTaskPrice(Task, Schedule)`

Die Methode `calcTaskPrice(Task, Schedule)` ermittelt den Preis des übergebenen Einsatzes im übergebenen Einsatzplan. Damit die Kosten eines Einsatzes ermittelt werden können ist es notwendig, dass sie von einem Tarif abgedeckt sind. In den dazugehörigen JUnit-Tests wurde die Methode bezüglich der folgenden Testfälle getestet:

- `testCalcTaskPriceInvalid1()`
  Der Testfall `testCalcTaskPriceInvalid1()` überprüft die Methode ob sie den richtigen Wert für einen Einsatz zurückgibt, der teilweise nicht von einem Tarif abgedeckt ist.

- `testCalcTaskPriceInvalid2()`
  Der Testfall `testCalcTaskPriceInvalid2()` überprüft die Methode ob sie den richtigen Wert für einen Einsatz zurückgibt, der nicht von einem Tarif abgedeckt ist.

- `testCalcTaskPriceValid1()`
  In dem Testfall `testCalcTaskPriceValid1()` wird überprüft ob die Methode den richtigen Wert zurückgibt, wenn der Einsatz von einem Tarif abgedeckt wird.

- `testCalcTaskPriceValid2()`
  In dem Testfall `testCalcTaskPriceValid1()` wird überprüft ob die Methode den richtigen Wert zurückgibt, wenn der Einsatz von zwei Tarifen abgedeckt wird.

Teil VIII. Gerätedaten

34. Quellen für Gerätedaten

Autor: Ulrich Denecke

Um die entwickelte Software sinnvoll nutzen zu können, ist es notwendig, Gerätedaten zu besitzen. Damit die Anwender nicht alle Gerätedaten selber eingeben müssen, wurde die Gerätebibliothek von der Projektgruppe mit einem Grundstock von Geräten gefüllt. In diesem Abschnitt wird erläutert, welche Quellen dafür verwendet wurden, welche Geräte problematisch sind und welche Arten von Geräten zum Auslieferungszeitpunkt in der Bibliothek hauptsächlich vorhanden sind.

Zur Beschaffung der Gerätedaten bieten sich viele Quellen an. Zum einen sind dies Handbücher und Typenschilder der Geräte. Die Typenschilder sind für die ersten Sammlungen von der Gruppe verwendet wurden. Zur Beschaffung von Daten im großen Stil sind sie jedoch nicht geeignet, da man die Geräte hierfür besitzen muss. Jedoch sind sie sehr gut für den späteren Anwender geeignet, da es ja für ihn interessant ist, die Geräte, die er auch besitzt - sofern sie in der Bibliothek noch nicht vorhanden sind - einzugeben. Handbücher sind ähnlich zu betrachten, jedoch sind sie teilweise auch im Internet vorhanden, so dass sie auch ohne das Gerät zu besitzen benutzt werden können.
Eine weitere Quelle für Produktdaten war Produktwerbung aus Tageszeitungen. Hierbei stellte ein Hersteller seine Produkte aus dem Bereich Kühlgeräte vor. Der größte Teil der Gerätedaten stammt jedoch aus dem Internet. Es wurden verschiedene Herstellerhomepages aufgesucht und Gerätedaten von dort übernommen. Weiterhin wurden auch Gerätedaten von den EcoTopTen entnommen. Vorteil von EcoTopTen ist, dass dies eine Quelle für Daten sparsamer Geräte ist. Dadurch enthält die Bibliothek auch sehr verbrauchsarme Geräte. Da bei den im Internet angegebenen Gerätedaten, beispielsweise für Waschmaschinen, nur der Verbrauch für ein Standardprogramm angegeben ist, wurden einige Hersteller per Email angeschrieben und um die Angabe von Lastgängen für verschiedene Geräte aus den Bereichen Waschmaschinen, Spülmaschinen, Wäschetrockner und Kochfelder gebeten. Die Antworten, falls welche von den Herstellern kamen, waren ausnahmslos ablehnend. Hierbei wurde darauf verwiesen, dass dies interne Daten seien.

Für stufenlos einstellbare Geräte oder Geräte mit kombinierbaren Programmen ist nicht einfach ein Betriebsmodus mit Laufzeit und Leistungsaufnahme anzugeben, da die aufgenommene Leistung nicht ohne weiteres bestimmt werden kann. Beispiele für solche Geräte mit kombinierbaren Programmen sind Waschmaschinen oder Kochfelder. Bei-

spiele für stufenlos einstellbare Geräte sind beispielsweise dimmbare Lampen, Staubsauger oder Ähnliches. Das gleiche Problem tritt auch bei Geräten auf, deren Leistungsaufnahme von Umgebungsbedingungen oder Auslastung abhängt. Wird eine Waschmaschine nicht an den Kaltwasser- sondern den Warmwasseranschluss angeschlossen, so sinkt ihre Leistungsaufnahme in den Heizphasen erheblich. Auch bei einem Computer ist die Leistungsaufnahme nicht konstant, sondern hängt von der Auslastung ab. Im Falle der eben genannten Geräte ist es sinnvoll für den Nutzer, sie selber anzulegen, einen Lastgang für die jeweils genutzen Programme aufzuzeichnen und diesen zu importieren.

35. Gerätegruppierungen

Autor: Ulrich Denecke

Für die Gerätesubstitution ist es notwendig, Gerätegruppierungen für die einzelnen Geräte zu vergeben. Diese Gruppen sollen es ermöglichen, verbrauchsärmere mit gleichen oder vergleichbaren Eigenschaften zu finden. Die Geräte erhalten falls notwendig mehrere Gruppen, die ihre Eigenschaften möglichst gut beschreiben. Hierbei sollten die Geräte Tags erhalten, die das Gerät zum einen möglichst genau beschreiben, beispielsweise bei Computerbidschirmen Art und Diagonale (TFT-Display, 17”), und auch Tags, die eine weitläufigere Gruppierung zulassen, bezogen auf das eben angeführte Beispiel der Computerbidschirme also die Gruppenbeschreibung (Bildschirm). Durch die genauere Beschreibung werden zuerst weitere 17”-TFT-Displays bei der Substitution als Auswahlmöglichkeit angeboten. durch die Gruppenbeschreibung Bildschirm werden jedoch auch andere Displays zur Auswahl angeboten, dass der Anwender auch auf ein verbrauchsärmeres 19”-Display zurückgreifen kann. Das gleiche Schema für die Gruppierung gilt auch für alle anderen Geräteklassen.

Teil IX. Ausblick

Autor: Malte Hurrelmann

Schon während der Entwicklung der DIEM Software ist aufgefallen, dass es durch die erreichte Modularität der auf Plugins basierenden Applikation an vielen Stellen ein Leichtes wäre, die Software für andere Aufgabenbereiche zu modifizieren oder zu erweitern. Einige dieser Möglichkeiten sollen im Folgenden vorgestellt werden.
Zuerst einmal ist festzustellen, dass die Software generell nicht nur für den Einsatz im Privathaushalt - wofür sie ja konzipiert und entwickelt worden ist - einsetzbar ist, sondern durch einfache Ersetzung einiger Komponenten auch für andere Einsatzgebiete benutzbar wird.
So ist es z.B. denkbar, durch einen Austausch der Gerätebibliothek, welche auf dem DIEM Server für den Client bereitgehalten wird, die Software für die Planung des Energiebedarfs eines Betriebes umzurüsten. Insbesondere für größere Industrieunternehmen, welche Maschinen mit hohem Strombedarf benutzen und somit auf die maximal erzeugte Last achten müssen, kann die Einsatzplanung der Software mit ihrer Maximallastbeschränkung hilfreich sein.
Im momentanen Zustand ist der DIEM Client in deutscher Sprache. Jedoch ließe sich durch einen Austausch der Sprachpakete leicht eine Anpassung an andere Sprachen erreichen.
Würde man weiterhin ein wenig Programmierarbeit aufbringen, so wäre es relativ einfach den DIEM Client durch zusätzliche Dateneingänge und -ausgänge zu erweitern. Somit wäre es denkbar Lastgänge, welche momentan über CSV-Dateien importiert werden können, direkt von einem Stromverbrauchsmessgerät zu beziehen. Hierfür müsste ein Java Modul implementiert werden, welches direkt den Ausgabedatenstrom eines solchen Messgerätes interpretiert und in ein DIEM Lastkurvenobjekt umwandelt. Dieses Gerät könnte beispielsweise via USB an den Rechner angeschlossen sein.
Auch die angesprochene Ausgabe des DIEM Client ließe sich durch die Modularität komfortabel erweitern. Somit ist es vorstellbar den Client durch ein Modul zu erweitern, welches die direkte Gerätesteuerung übernimmt. So könnte ein erstellter Einsatzplan direkt mit Hilfe von steuerbaren Steckdosen(-leisten) umgesetzt werden und insbesondere Elektrogrossgeräte wie Waschmaschinen oder Spülmaschinen gezielt zum kostengünstigsten Zeitpunkt betrieben werden.
Neben der Gerätebibliothek ließe sich prinzipiell auch die Zielfunktion, nach der optimiert werden soll, ersetzen. Somit wäre es denkbar einen Einsatzplan, nicht wie im Moment mit Blick auf die geringsten entstehenden Kosten, sondern eine möglichst gleichmäßige Lastverteilung hin zu optimieren. So könnte ein Energieerzeuger mit dem DIEM Client theoretisch auch positive wie negative Regelenergie planen um somit sein Netz auf der benötigten Frequenz zu halten.