Entwurf und Implementierung eines Simulators für zeitdiskrete Simulation in C++

Ein Werkzeug zur Entwicklung wiederverwendbarer Modelle

Diplomarbeit

Band 1

Fachbereich Informatik

Universität Hamburg

vorgelegt von

Thomas Schniewind

im

März 1998

Erstbetreuer

Prof. Dr. Bernd Page

Zweitbetreuer

Prof. Dr. Rainer Lang

Inhaltsverzeichnis

1 Einleitung 1

1.1 Einführung 1

1.2 Ziele 1
1.2.1 Vereinfachung 1
1.2.2 Modellkonzept 1
1.2.3 Harmonisierung der Weltbilder 2
1.2.4 Erweiterung der Einsatzmöglichkeiten 2

1.3 Übersicht 2

2 Grundlagen 5

2.1 Grundlegende Begriffe der objektorientierten Programmierung (OOP) 5
2.1.1 Klassen 5
2.1.2 Objekte 6
2.1.3 Beziehungen 7
2.1.4 Interaktion 9
2.1.5 Polymorphismus 10
2.1.6 Entwurfsmuster 11

2.2 Weltbilder der zeitdiskreten Simulation 12
2.2.1 Ereignisorientierter Ansatz 12
2.2.2 Prozeßorientierter Ansatz 14

2.3 DESMO 15
2.3.1 Umsetzung des ereignisorientierten Ansatzes 15
2.3.2 Umsetzung des prozeßorientierten Ansatzes 16
2.3.3 Gemeinsame Funktionen 17
2.3.3.1 Warteschlangen 17
2.3.3.2 Zufallszahlenströme 17
2.3.3.3 Statistische Datensammelobjekte 18
2.3.3.4 Report 18
2.3.4 Erweiterungen des prozeßorientierten Ansatzes 18

2.4 SiFrame 19
2.4.1 Ereignisorientierter Ansatz 21
2.4.2 Prozeßorientierter Ansatz 21
2.4.3 Gemeinsame Funktionen 21
2.4.3.1 Warteschlangen 21
2.4.3.2 Zufallszahlenströme 22
2.4.3.3 Statistische Datensammelobjekte 22
2.4.3.4 Report 23
2.4.4 Erweiterungen des prozeßorientierten Ansatzes 23

2.5 Beurteilung 25
2.5.1 Klarheit der Begriffe 25
2.5.2 Trennung der Weltbilder 26
2.5.3 Durchführung eines Experiments 26

3 Das neue Konzept: DESMO-C 27

3.1 Die Zusammenführung der Weltbilder 27
3.1.1 Warum die Kombinierbarkeit sinnvoll ist 27
3.1.2 Unterschiede und Überschneidungen der Weltbilder 28
3.1.3 Entities 28
3.1.4 Prozesse als Entities mit aktivem Verhalten 28
3.1.5 Ereignisse 30
3.1.6 Vormerken von Objekten 31
3.1.7 Konsequenzen 32

3.2 Modell und Experiment - das Baukastenprinzip 33
3.2.1 Modellkomponenten 34
3.2.2 Modellhierarchien 35
3.2.3 Modellintegrität 36
3.2.4 Experimente 36
3.2.5 Experimentreihen innerhalb eines Programms 38
3.2.6 Einbettung in andere Systeme 39

3.3 Warteschlangenbasierte Objekte 39
3.3.1 Implizite Warteschlangen 39
3.3.2 Warteschlangen für Entities und Prozesse 41
3.3.3 Suche in Warteschlangen 42

3.4 Höhere Synchronisationsmechanismen 45
3.4.1 Puffer (Bin) 46
3.4.2 Bedingtes Warten (CondQueue) 46
3.4.3 Ressourcenwettbewerb (Res) 47
3.4.4 Direkte Prozeßkooperation (WaitQueue) 47
3.4.5 Unterbrechung von Prozessen 49

3.5 Statistische Datensammelobjekte 50
3.5.1 Wertlieferanten für Beobachtungsgrößen 51
3.5.2 Statistik über Beobachtungsgrößen 51
3.5.3 Anwendung des Beobachtermusters 53

3.6 Reportfähige Objekte 56
3.6.1 Ausgliederung der Reportfunktionalität 57
3.6.2 Vom Reporter zum Report 58
3.6.3 Rücksetzung 59

4 Implementierung 61

4.1 Besondere Aspekte von C++ 61
4.1.1 Automatisch generierte Methoden 61
4.1.2 Downcast - ein Problem der strengen Typisierung 63
4.1.2.1 Einfacher Cast 63
4.1.2.2 RTTI (Runtime Type Information) 64
4.1.2.3 Virtuelle Cast-Funktionen 64
4.1.2.4 Kompromiß (Empfehlung für Systeme ohne RTTI) 65
4.1.3 Templates 66

4.2 Elemente der Simulationssteuerung 68
4.2.1 Ereignisliste 68
4.2.1.1 Die abstrakte Ereignisliste 68
4.2.1.2 Lineare Liste 70
4.2.2 Scheduler und Simulationsuhr 70
4.2.3 Das Nachrichtensystem 74
4.2.3.1 Nachrichten 74
4.2.3.2 Nachrichtenquellen 75
4.2.3.3 Nachrichtenverteiler 76
4.2.3.4 Nachrichtensenken 78
4.2.4 Fehlerbehandlung 79
4.2.4.1 Gültigkeit von Objekten 79
4.2.4.2 Fehlerarten 80
4.2.4.3 Behandlung von Fehlern 80

4.3 Experimentverwaltung 81
4.3.1 Experimentmanager 81
4.3.2 Speichermangel 84
4.3.3 Ende des Experiments 84
4.3.4 Dynamische Objekte und Freispeicherverwaltung 84
4.3.5 Automatische Numerierung von Namen 85

4.4 Hilfsklassen 86
4.4.1 Koroutinen 86
4.4.1.1 Die Klasse Coroutine aus [Weber96] 87
4.4.1.2 Die Klasse Coroutine in DESMO-C 87
4.4.1.3 Konsequenzen 88
4.4.2 Aus SiFrame übernommene Klassen 89
4.4.2.1 String 89
4.4.2.2 SimTime 89
4.4.2.3 Ring 89
4.4.2.4 QueueLink 90
4.4.2.5 Avl 90
4.4.2.6 ResourceDB 90

5 Anwendung und Beispiele 91

5.1 Schrittweise Entwicklung eines einfachen Modells 91
5.1.1 Das Ping-Pong-Modell 91
5.1.2 PingPong (prozeßorientiert) 91
5.1.2.1 Der Ping-Pong-Prozeß 91
5.1.2.2 Modell und Experiment 93
5.1.3 PingPong (ereignisorientiert) 96
5.1.3.1 Die Ereignistypen “Ping” und “Pong” 96
5.1.3.2 Modell und Experiment 98

5.2 Beispielmodelle mit DESMO-C 101
5.2.1 Das Hauptprogramm 101
5.2.2 Simulation eines Fertigungssystems 107
5.2.2.1 Gemeinsamkeiten der Versionen 107
5.2.2.2 Ereignisorientierte Version 112
5.2.2.3 Version mit Ressoucen (transaktionsorientiert) 120
5.2.3 Simulation einer Fähre 125
5.2.3.1 Version mit Puffern (Bin) 125
5.2.3.2 Version mit direkt kooperierenden Prozessen 129
5.2.4 Platzreservierungsmodell 134
5.2.4.1 Grundmodell 134
5.2.4.2 Variante mit Bürokunden 143
5.2.4.3 Variante mit Pausen 148
5.2.5 Simulation eines Steinbruchs 153
5.2.6 Hafenmodell 166
5.2.6.1 Grundmodell 166
5.2.6.2 Variante mit Gezeiten 171

6 Referenz der Schnittstellenobjekte 177

6.1 Übersicht 177
6.1.1 Experiment, Modell und Modellkomponenten 178
6.1.2 Reportfähige Objekte (Reportable) 179
6.1.2.1 Zufallszahlenströme (Distribution) 179
6.1.2.2 Warteschlangen und darauf basierende Objekte 180
6.1.2.3 Höhere Synchronisationsmechanismen 181
6.1.2.4 Statistische Datensammelobjekte (StatisticObject) 181
6.1.3 Dynamische Objekte (DynamicalObject) 182
6.1.4 Verdrängung mit Hilfe von 'NOW' 183
6.1.5 Objekte dynamisch anlegen 183
6.1.6 Von DESMO nach DESMO-C 184

6.2 Katalog ... 185
6.2.1 Accumulate ... 186
6.2.2 Bin ... 189
6.2.3 BoolDist ... 192
6.2.4 BoolDistBernoulli ... 194
6.2.5 BoolDistConst ... 196
6.2.6 Condition ... 198
6.2.7 CondQueue ... 199
6.2.8 Count ... 201
6.2.9 Distribution ... 203
6.2.10 DynamicalObject ... 205
6.2.11 Entity ... 207
6.2.12 Event ... 210
6.2.13 Experiment ... 212
6.2.14 ExperimentOpts ... 219
6.2.15 ExternalEvent ... 221
6.2.16 Histogram ... 223
6.2.17 IntDist ... 226
6.2.18 IntDistConst ... 228
6.2.19 IntDistEmpirical ... 230
6.2.20 IntDistPoisson ... 232
6.2.21 IntDistUniform ... 234
6.2.22 InterruptCode ... 236
6.2.23 Message ... 238
6.2.24 MessageReceiver ... 241
6.2.25 Model ... 242
6.2.26 ModelComponent ... 247
6.2.27 NamedObject ... 254
6.2.28 Observable ... 256
6.2.29 Observer ... 257
6.2.30 Output ... 259
6.2.31 Process ... 261
6.2.32 ProcessCooperation ... 265
6.2.33 ProcessQueue ... 268
6.2.34 Queue ... 272
6.2.35 QueueBased ... 276
6.2.36 RealDist ... 278
6.2.37 RealDistConst ... 280
6.2.38 RealDistEmpirical ... 282
6.2.39 RealDistErlang ... 284
6.2.40 RealDistExponential ... 286
6.2.41 RealDistNormal ... 288
6.2.42 RealDistUniform ... 290
6.2.43 Regression ... 292
6.2.44 Reportable ... 295
6.2.45 Res ... 297
6.2.46 Schedulable ... 300
6.2.47 SimTime ... 303
6.2.48 StatisticObject ... 305
6.2.49 StdDebug ... 307
6.2.50 StdError ... 308
6.2.51 StdOutput ... 309
6.2.52 StdReport ... 311
6.2.53 StdTrace ... 313
6.2.54 String ... 314
6.2.55 Tally ... 317
6.2.56 TimeSeries ... 319
6.2.57 ValueStatistics ... 322
6.2.58 ValueSupplier ... 324
6.2.59 WaitQueue ... 325

7 Schlußbemerkungen und Ausblick 331
7.1 Schlußbemerkungen 331
7.2 Laufzeitvergleich 331
7.3 Ausblick 332

8 Literatur 333

9 Verzeichnisse 335
9.1 Abkürzungen 335

1 Einleitung

1.1 Einführung

Das Simulationspaket DESMO (Discrete Event Simulation in Modula-2) wurde 1987-89 am Fachbereich Informatik der Universität Hamburg als eine Modulsammlung für die Sprache Modula-2 entwickelt. Es basiert auf den für prozeßorientierte Simulation nutzbaren Konzepten von DEMOS (Discrete Event Modelling On Simula), das 1979 von G. Birtwistle vorgestellt wurde. Zusätzlich erweitert DESMO sein Vorbild um die ereignisorientierte Simulation.

DESMO wurde seit seiner Einführung am Fachbereich Informatik intensiv in der Lehre eingesetzt, um den Studierenden die Konzepte der zeitdiskreten Simulation nahezubringen. Dabei bildete die Sprache Modula-2 eine gute Grundlage, da der überwiegende Teil der Studierenden die Kenntnis der Sprache aus dem Grundstudium mitbrachten. Mittlerweile hat das objektorientierte Paradigma Einzug gehalten, so daß der Bedarf nach einer Umsetzung in einer entsprechenden Sprache wächst.

SiFrame (Simulation Framework) ist ein im Rahmen einer Diplomarbeit am Fachbereich Informatik der Universität Hamburg entwickelter Ansatz, die Funktionalität von DESMO in der Sprache C++ zu realisieren. Leider verläuft die Arbeit in weiten Zügen so dicht am Original, daß objektorientierte Konzepte oft ungenutzt bleiben, obwohl sie enorm zur Vereinfachung beitragen könnten. Eine Überarbeitung ist erforderlich und soll hier geleistet werden. Die Ziele dieser Arbeit sind jedoch noch etwas weiter gesteckt, indem die Klassenbibliothek **DESMO-C** (Discrete Event Simulation and Modelling in C++) zur zeitdiskreten Simulation für die Sprache C++ entwickelt wird, deren Funktionalität der von DESMO entspricht und um neue Konzepte erweitert.

1.2 Ziele

1.2.1 Vereinfachung

SiFrame ist zwar eine Implementierung der Funktionalität von DESMO in einer objektorientierten Programmiersprache, aber es nutzt in einigen Fällen die Konzepte dieses Paradigmas nicht aus. Statt dessen übernimmt SiFrame Mechanismen aus DESMO, die seinerzeit in Ermangelung besserer Alternativen verwendet wurden. Durch die konsequente Nutzung objektorientierter Techniken soll die Anwendung des Simulationspakets vereinfacht werden. Dabei wird insgesamt auf eine starke Übereinstimmung von Begriffen und Beziehungen aus dem Anwendungsbereich, allgemein der Simulation, mit den Bezeichnern für Klassen und Methoden des Simulationspakets Wert gelegt.

1.2.2 Modellkonzept

Um die Modellbildung und das Experimentieren mit Modellen zu unterstützen, sollen Werkzeuge zur Verfügung gestellt werden, die es ermöglichen, Modelle als eigenständige Einheiten zu erstellen. Diese Modelle sollen sich, weniger durch pures Kopieren des Quelltextes sondern vielmehr durch Wiederverwendung ihrer selbst, zu komplexeren kombinieren lassen. Auf diese Weise kann eine ganze Bibliothek an Modellen entstehen. Ein Modell

selbst enthält noch keine Funktionalität zur Simulation, es beschreibt vielmehr das Verhalten des abzubildenden realen Systems. Um die Simulation in Gang zu bringen, muß ein Experiment mit dem Modell durchgeführt werden. Auch können in Abhängigkeit von Ergebnissen eines Experiments Folgeexperimente durchgeführt werden, ohne das Programm zu beenden.

1.2.3 Harmonisierung der Weltbilder

Die Einführung von Modellen und die Forderung nach deren uneingeschränkter Verwendbarkeit erfordert die Koexistenz von ereignis- und prozeßorientiertem Ansatz, denn sonst wäre bei der Kombination von Modellen stets zu berücksichtigen, daß alle im gleichen Ansatz entwickelt wurden. Ein Modellentwickler, der ein Modell als Baustein anbieten möchte, könnte nicht mehr frei entscheiden, welcher Ansatz ihm für die Beschreibung des Modells am geeignetsten erscheint, oder müßte zwei Versionen zur Verfügung stellen. Sind jedoch in beiden Ansätzen die Konzepte des jeweils anderen ebenso verfügbar, bleiben die Modelle frei kombinierbar, ohne daß die Art der Realisierung berücksichtigt werden muß. Abgesehen davon ist es von Vorteil, wenn man stets das angemessene Konzept verwenden kann. Z.B. würde ein einmaliger Störfall, der also eher Ereignischarakter hat, in einem prozeßorientierten Modell mit Hilfe eines dedizierten Prozesses modelliert werden müssen, der genau einmal während der Simulation aktiviert wird, um die Störung zu simulieren. Statt dessen kann ein regelrechtes Störereignis angesetzt werden. Manche Modelle lassen sich zudem in dem einen oder anderen Ansatz effizienter realisieren (vgl. [Bölck89] S. 44 f.). Für DESMO-C soll ein Simulationskern und darauf aufbauend ein neues Konzept entwickelt werden, das eine gemeinsame Nutzung aller Modellierungsstile zuläßt, aber nicht vorschreibt.

1.2.4 Erweiterung der Einsatzmöglichkeiten

Die entstehende Simulationsbibliothek soll mit anderen Bibliotheken und Frameworks harmonieren können. So könnte z.B. ein grafisches Modellierungswerkzeug entstehen, das auf der Simulationsbibliothek aufbaut. Dies erfordert, daß die Bibliothek nicht in einen irreversiblen Zustand gerät (außer durch schwerwiegende Fehler), der sich nur durch Beenden oder Abbruch des ganzen Programms zurücksetzen läßt. Hierfür soll die Menge an globalen statischen Strukturen auf ein Minimum reduziert werden, indem möglichst viele der Komponenten, die für einen Simulationslauf benötigt werden, dynamisch erzeugt werden. Das Löschen eines durchgeführten Experiments sollte alle angeforderten Ressourcen wieder freigeben, so daß eine beliebige Anzahl von Experimenten durchgeführt werden kann.

In SiFrame wurde die C-Funktion `main`, die den Hauptanweisungsblock des Programms darstellt, in die Bibliothek integriert. Dadurch wird dem Modellprogrammierer die Kontrolle über den Ablauf des Programms entzogen. Dies führt dazu, daß SiFrame nicht mit anderen Bibliotheken kombinierbar ist, die das gleiche Prinzip anwenden, da hierdurch die Funktion `main`, die im gesamten Programm nur einmal definiert sein darf, mehrfach auftreten würde. Daher soll `main` wieder in die Hände des Modellprogrammierers gelegt werden.

1.3 Übersicht

Kapitel 2 erläutert die wichtigsten Grundlagen, die zum Verständnis der in dieser Arbeit beschriebenen Konzepte erforderlich sind. Neben der Einführung einer grafischen Notation

für objektorientierte Konzepte und einer Darstellung der Weltbilder der zeitdiskreten Simulation stellt es die für die weiteren Betrachtungen wichtigsten Aspekte der Vorläufer DESMO und SiFrame vor. Das Kapitel endet mit einer kritischen Betrachtung und Verbesserungsvorschlägen.

Kapitel 3 stellt das in dieser Arbeit entwickelte Konzept vor und geht soweit auf deren Realisierung ein, wie dies zum Verständnis der für die Gestaltung der Schnittstelle getroffenen Entwurfsentscheidungen notwendig war. Mit der inneren Sicht auf das Simulationspaket und den damit verbundenen Entwurfsproblemen befaßt sich Kapitel 4. Eine kurze Einführung in die Anwendung von DESMO-C anhand der schrittweisen Entwicklung eines einfachen Beispielmodells ist Gegenstand von Kapitel 5. Dort sind auch größere Modelle zu finden.

Einen erheblichen Teil dieser Arbeit stellt die in Kapitel 6 zusammengestellte Referenz der Schnittstellenobjekte dar, die ich für eine gute Nutzbarkeit der Klassenbibliothek als außerordentlich wichtig erachte. Dabei wird nicht der Versuch unternommen, die einzelnen Klassen thematisch zu gliedern. Vielmehr handelt es sich um einen alphabetischen Katalog aller relevanten Klassen, deren Beschreibungen durch Bereitstellung umfangreicher Querverweise ein schnelles Auffinden der gewünschten Information ermöglichen soll. Eine kurze Einleitung zu Beginn des Kapitels gibt einen strukturierten Überblick über die wichtigsten Klassen im Zusammenhang mit der gesamten Klassenhierarchie.

2 Grundlagen

In diesem Kapitel werden die Grundlagen für diese Arbeit vorgestellt. Zunächst werden in Abschnitt 2.1 die zum Verständnis der folgenden Darstellungen wichtigsten Konzepte der objektorientierten Programmierung erläutert und eine entsprechende Notation eingeführt. Nach einer kurzen Einführung in die Weltbilder der zeitdiskreten Simulation[1] in Abschnitt 2.2 werden die für die weitere Betrachtung wesentlichen Aspekte der Vorgänger DESMO und SiFrame näher beleuchtet. Im Anschluß daran werden die verbesserungswürdigen Punkte, insbesondere von SiFrame, aufgezeigt.

2.1 Grundlegende Begriffe der objektorientierten Programmierung (OOP)

In diesem Abschnitt werden die für das Verständnis dieser Arbeit wichtigsten Konzepte der objektorientierten Programmierung kurz beleuchtet, und eine Notation eingeführt, um entsprechende Zusammenhänge grafisch darstellen zu können. Eine ausführliche Darstellung kann im Rahmen dieser Arbeit jedoch nicht geleistet werden, weshalb ich für das tiefere Verständnis auf die folgenden Quellen verweise:

OOP: [Booch95] , [Meyer90], [Gamma95]

C++: [Strou92], [Meyer96], [Meyer98], [Eckel96]

2.1.1 Klassen

Zum Begriff der Klasse findet sich in [Booch95] folgende Definition:

> Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen.[2]

Bestandteil der Struktur sind u.a. die Attribute der Objekte, auch Elemente genannt. Die Attribute repräsentieren einen Teil des Objektzustands. Das Verhalten wird mit Hilfe von Methoden realisiert, die auch Nachrichten genannt werden.

Als Beispiel soll hier eine Klasse von geometrischen Figuren (`Shape`) dienen. Eine solche geometrische Figur besitzt das Attribut Position (`position`), das angibt, wo sich diese Figur in einem Koordinatensystem, z.B. dem Bildschirm, befindet. Das Verhalten äußert sich in einer Methode `Draw` zum Zeichnen der Figur. Mit dieser Methode wird die Figur gewissermaßen aufgefordert, sich selbst an der entsprechenden Position zu zeichnen. Da das Attribut von außen nicht direkt zugänglich sein soll, gibt es eine Methode für den Zugriff auf die Position `GetPosition`. Die Methode `MoveTo` veranlaßt die Figur, sich an eine andere Stelle zu bewegen. Zusätzlich wird zur Repräsentation von Positionen die Klasse der Punkte (`Point`) benötigt. Punkte besitzen zwei Attribute für ihre x- bzw. y-Koordinate. Diese sind über die Methoden `Set`, `GetX` und `GetY` zugreifbar. Abbildung 2-1 zeigt die Klassen in der Notation nach Booch.

---

[1] Für eine ausführlichere Beschreibung verweise ich auf [Page91].

[2] [Booch95], S. 136

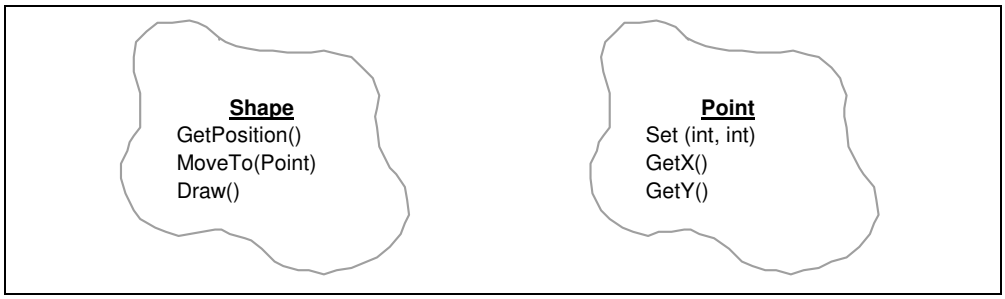

Abbildung 2-1: Klassendiagramm der Klassen `Shape` und `Point`

Listing 2-1 zeigt die selben Klassen in C++-Notation.

```
 1. class Point
 2. {
 3.     public:
 4.         void Set  (int X, int Y) { x = X; y = Y; }
 5.         int  GetX ()             { return x; }
 6.         int  GetY ()             { return y; }
 7. private:
 8.         int  x, y;
 9. };
10.
11. class Shape
12. {
13.     public:
14.         Point GetPosition ()    { return position; }
15.         void  Draw   ()         { ... }
16.         void  MoveTo (Point p) { position = p; Draw(); }
17.     private:
18.         Point position;
19. };
```

Listing 2-1: C++-Klassendeklarationen für `Shape` und `Point`

In der Notation nach Booch können Zugriffsbeschränkungen auf Methoden bzw. Attribute mit | für geschützt (nur die Klasse selbst und ihre Unterklassen haben Zugriff auf dieses Element) und || für privat (nur die Klasse selbst hat Zugriff) angegeben werden.

2.1.2 Objekte

Der Begriff des "Objekts" soll durch die folgenden vier Definitionen aus [Booch95] konkretisiert werden:

> Ein Objekt hat einen Status, ein Verhalten und eine Identität; die Struktur und das Verhalten ähnlicher Objekte sind in ihrer gemeinsamen Klasse definiert; die Begriffe Instanz und Objekt sind austauschbar.[3]
>
> Der Status eines Objekts umfaßt alle (normalerweise statischen) Eigenschaften des Objekts, zusammen mit den aktuellen (normalerweise dynamischen) Werten dieser Eigenschaften.[4]
>
> Verhalten ist die Art und Weise, wie ein Objekt agiert und reagiert, in Form von Statusänderungen und der Übergabe von Nachrichten.[5]

[3] [Booch95], S. 111
[4] [Booch95], S. 112
[5] [Booch95], S. 115

> Die Identität ist die Eigenschaft eines Objekts, die es von allen anderen Objekten unterscheidet. [6]

Anstelle von Instanz bzw. Objekt wird in [Gamma95] und [Meyer90] auch der Begriff "Exemplar" verwendet, was die Bedeutung des englischen Wortes "Instance" im Deutschen besser trifft als die häufig verwendete Übersetzung "Instanz".[7] Die Definition von Verhalten führt dazu, daß der Status eines Objekts auch als "die kumulierten Ergebnisse seines Verhaltens"[8] dargestellt werden kann.

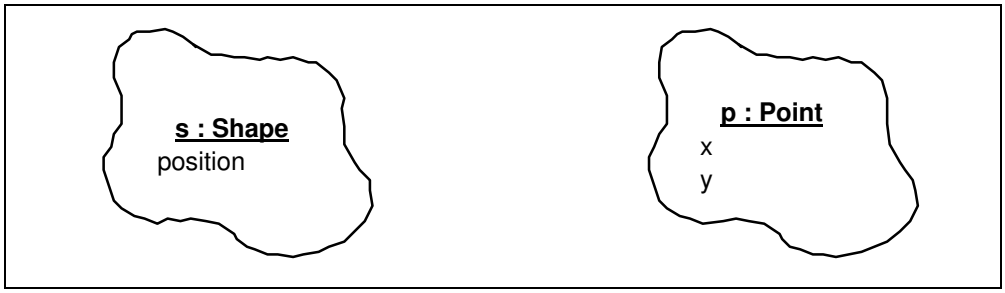

Abbildung 2-2: Objektdiagramm für je ein `Shape`- bzw. `Point`-Objekt

Als Beispiel für die Notation von Objekten ist in Abbildung 2-2 jeweils ein Objekt `p` der Klasse `Point` und ein Objekt `s` der Klasse `Shape` dargestellt. Dabei ist der Name des Objekts vom Namen der Klasse durch einen Doppelpunkt getrennt. Beide können jedoch auch allein stehen: z.B. "`s`" oder "`: Shape`". Unter dem Objektnamen sind die Attribute dargestellt. Listing 2-2 zeigt die Objekte in ihrer jeweiligen C++-Notation.

```
1. Point p;
2. Shape s;
```

Listing 2-2: Die Objekte `p` und `s` in C++-Notation

2.1.3 Beziehungen

Abbildung 2-3 zeigt die vier wichtigsten Beziehungen zwischen Klassen, wobei die Beschriftung die Beziehung von links nach rechts beschreibt. Mit der Assoziation kann eine zunächst unspezifizierte Beziehung notiert werden, die in späteren Entwicklungsschritten verfeinert werden kann. Sie wird aber auch verwendet, wenn die genaue Art der Beziehung im dargestellten Zusammenhang nicht von Bedeutung ist. Die Vererbung wird durch einen Pfeil notiert, der auf die Oberklasse zeigt. Vererbung bedeutet immer eine "ist ein"-Beziehung zwischen Unter- und Oberklasse[9].

Häufig werden objektorientierte Systeme ausschließlich anhand der Darstellung von Vererbungsbeziehungen beschrieben.[10] Doch ergeben sich viele wichtige Aspekte erst durch die Eigentumsbeziehung. Sie dient der Darstellung von Teil/Ganzes-Beziehungen, auch Aggregation genannt. Ihre Notation entsteht durch Hinzufügen eines ausgefüllten Punktes auf der Seite des Eigentümers oder "Ganzen". Die Eigentumsbeziehung ist in zwei Varianten notierbar. Physikalisches Enthaltensein eines Teils in einem Ganzen wird mit einem ausge-

---

6 [Booch95], S. 121. Booch übernimmt diese Definition aus [Khosh86], S. 406

7 Vgl. [Gamma95], S. 19

8 [Booch95], S. 116

9 Eine Ausnahme stellt die private Vererbung dar, die eine "ist implementiert mit"-Beziehung ausdrückt.

10 Z.B. in [Weber96] oder [Trush95]

füllten Quadrat auf der Seite des Teils angemerkt. Die Lebensdauer von Teil und Ganzem sind hier unmittelbar miteinander verknüpft. Enthält der Eigentümer jedoch nur einen Verweis auf das Teil, so wird dies mit einem nicht ausgefüllten Quadrat notiert. Das Teil kann hier auch über die Lebensdauer des Ganzen hinaus weiter existieren. Mit der Verwendungsbeziehung kann dargestellt werden, daß eine Klasse als Parameter einer Methode einer anderen Klasse verwendet wird. Sie wird mit einem nicht ausgefüllten Kreis auf der Seite der verwendenden Klasse notiert.

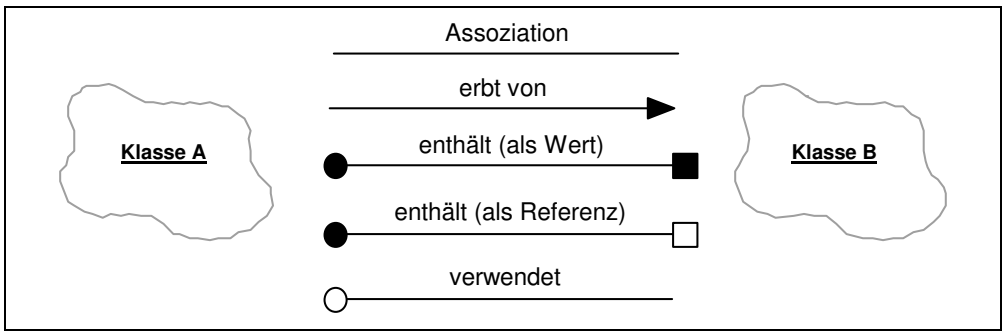

Abbildung 2-3: Beziehungen zwischen Klassen

Das Beispiel aus Abschnitt 2.1.1 beinhaltet bereits eine Eigentumsbeziehung, nämlich im Zusammenhang mit der Position der geometrischen Figur. Diese Beziehung wird in Abbildung 2-4 dargestellt. Zusätzlich wird für die Eigentumsbeziehung die Kardinalität 1 angegeben, was bedeutet, das ein `Shape`-Objekt genau ein `Point`-Objekt enthält. Darüber hinaus erbt die Klasse der Kreise `(Circle)` von der Klasse der geometrischen Figuren, denn ein Kreis “ist eine” geometrische Figur.

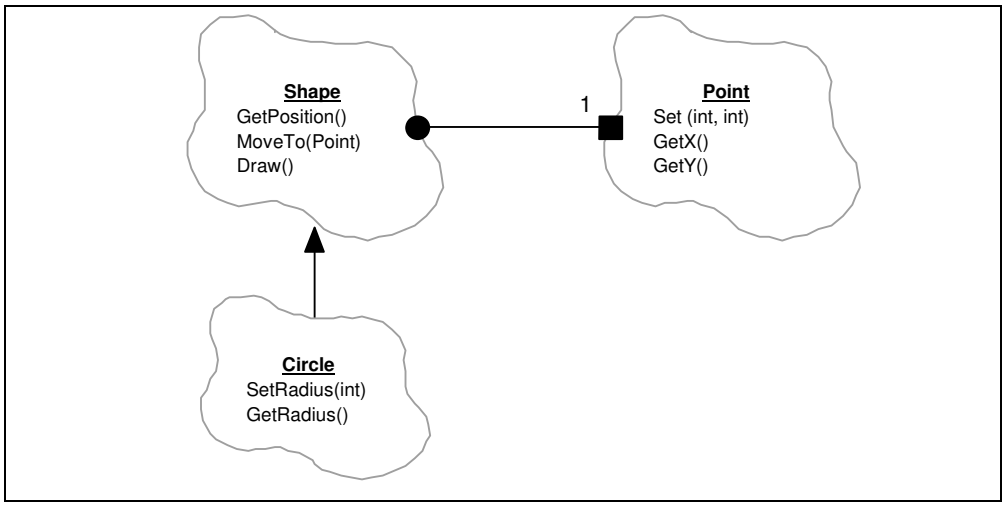

Abbildung 2-4: `Circle` erbt von `Shape`

Listing 2-3 gibt die C++-Notation für Vererbung am Beispiel der Klasse `Circle` wieder.

```
1. class Circle : public Shape
2. {
3.     public:
4.         int  GetRadius ()       { return radius; }
5.         void SetRadius (int r) { radius = r;    }
6.     private:
7.         int  radius;
8. };
```

Listing 2-3: Vererbung in C++ am Beispiel der Klasse `Circle`

Eine Beziehung kann auf einer Seite mit der Anmerkung F versehen werden (Abbildung 2-5), um zu kennzeichnen, daß die Klasse "Freund" der anderen Klasse ist. Die Freund-Klasse besitzt Zugriff auf alle internen Details der anderen Klasse. Die Freund-Eigenschaft ist weder vererbbar noch transitiv.

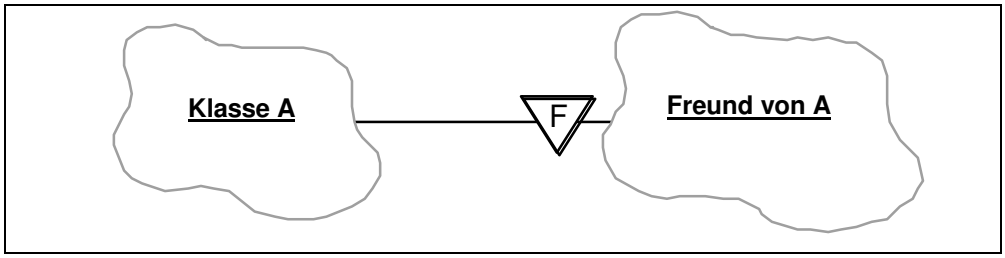

Abbildung 2-5: "Freundschaft" zwischen Klassen

2.1.4 Interaktion

In einigen Situationen ist es von großem Nutzen, die Interaktionen zwischen Objekten darzustellen. Mechanismen, die über mehrere Methodenaufrufe unterschiedlicher Objekte verteilt sind, können mit Hilfe von Interaktionsdiagrammen leichter nachvollzogen werden. Als Beispiel sollen die Interaktionen der Objekte `s` der Klasse `Shape` und `p` der Klasse `Point` betrachtet werden, wenn die Position des `Shape`-Objekts gesetzt wird.

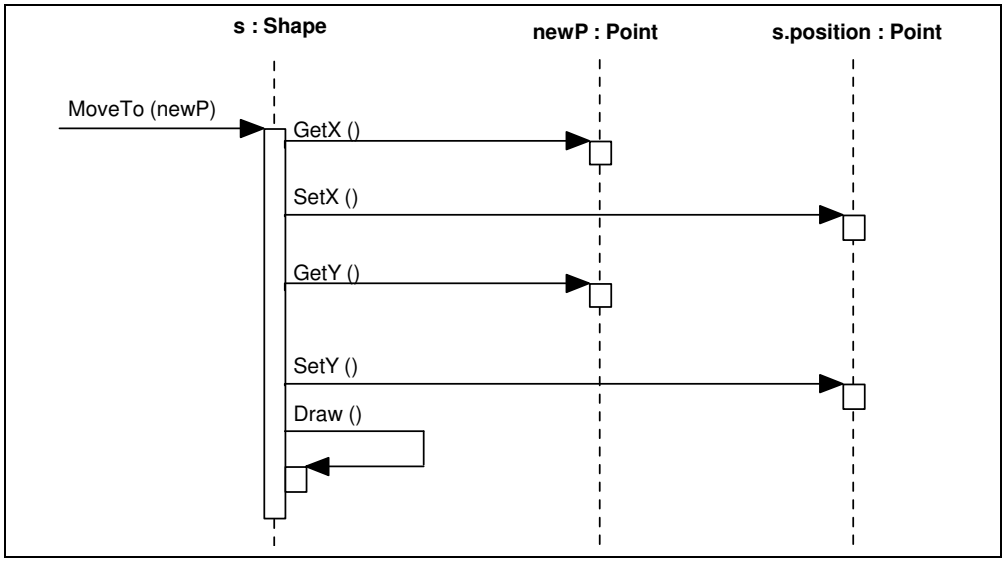

Abbildung 2-6: Interaktionsdiagramm für das Verschieben einer Figur

In Abbildung 2-6 kennzeichnen die senkrechten Balken die Zugehörigkeit von Aktionen zu einem Objekt. Die Objekte selbst sind durch die senkrechten gestrichelten Linien dargestellt. Die Methode `MoveTo` wird mit dem Parameter `newP`, einem `Point`-Objekt, aufgerufen. Die Figur `s` extrahiert zunächst die x-Koordinate des Parameters `newP` und ruft mit dem erhaltenen Wert die Methode `SetX` des eigenen Attributs `position` auf, das ebenfalls ein Objekt der Klasse `Point` ist. Gleiches erfolgt für die y-Koordinate. Zum Schluß ruft die Figur `s` ihre eigene Methode `Draw` auf, um sich neu zu zeichnen.

2.1.5 Polymorphismus

> Polymorphismus wird die Fähigkeit genannt, verschiedene Formen anzunehmen. In der objektorientierten Programmierung bezieht sich das auf die Fähigkeit einer Größe, zur Laufzeit auf Exemplare verschiedener Klassen zu verweisen.[11]

Dies sei wieder an dem Beispiel der geometrischen Figuren verdeutlicht, indem wir eine zweite spezielle Figur "Rechteck" einführen (Klasse `Rectangle`). Ein Rechteck ist eine geometrische Figur aber auf keinen Fall ein Kreis. `Rectangle` erbt also von der Klasse `Shape`, was in Abbildung 2-7 wiedergegeben ist.

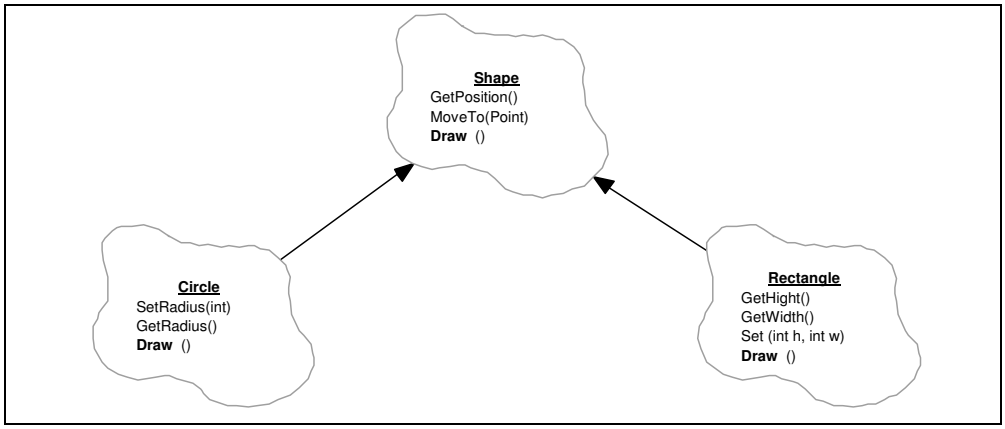

Abbildung 2-7: `Circle` und `Rectangle` erben von `Shape`

Das Entscheidende an diesem Beispiel ist, daß der Aufruf von `Draw` bei einem Kreis zwar das gleiche bedeutet wie bei einem Rechteck, nämlich daß die geometrische Figur gezeichnet wird. Aber wie dies zu geschehen hat, hängt von der speziellen Klasse ab. Schließlich muß ein Kreis anders gezeichnet werden als ein Rechteck. Nun ist es möglich sowohl Kreise als auch Rechtecke als geometrische Figur zu behandeln, d.h. in einem Zusammenhang, in dem nur die Eigenschaften der Klasse `Shape` eine Rolle spielen. Der statische Typ des behandelten Objekts auf dieser Ebene ist `Shape`, der dynamische Typ hängt jedoch davon ab, ob das Objekt konkret ein Kreis oder ein Rechteck ist. Wird auf dieser Ebene durch Aufruf von `Draw` die Figur angewiesen, sich selbst zu zeichnen, so ist das Ergebnis bei einem Kreis ein anderes als bei einem Rechteck. M.a.W. der Aufruf von `Draw` verhält sich polymorph.

Um in C++ dieses Verhalten zu erzielen, muß in der Klasse `Shape` die Methode `Draw` entsprechend deklariert werden, indem ihr das Schlüsselwort `virtual` vorangestellt wird. Andernfalls würde auf einer Ebene, auf der nur bekannt ist, daß es sich um ein `Shape`-Objekt handelt, die Implementierung von `Shape::Draw` aufgerufen. Somit wäre kein polymorphes Verhalten erreicht.

---

[11] [Meyer90], S. 241

In diesem Beispiel macht es allerdings keinen Sinn, ein allgemeines `Shape`-Objekt zu erzeugen. Wie sollte so ein Objekt sich zeichnen? Eine Implementierung der Methode `Shape::Draw` ist also nicht sinnvoll. In einem solchen Fall wird die Methode als "abstrakte Methode", oder in C++-Terminologie als "rein virtuelle Elementfunktion", deklariert, wodurch die Klasse zu einer abstrakten Klasse wird. Von einer abstrakten Klasse können keine Objekte erzeugt werden. Unterklassen müssen abstrakte Methoden definieren, andernfalls bleiben die Klassen abstrakt. In der Notation nach Booch wird eine abstrakte Klasse mit der Anmerkung A versehen. In C++ wird dies durch das Anhängen von "= 0" an den Methodenkopf erreicht, was soviel bedeutet, daß diese Methode keine Implementierung besitzt. Abbildung 2-8 und Listing 2-4 zeigen dies für die Klasse `Shape`.

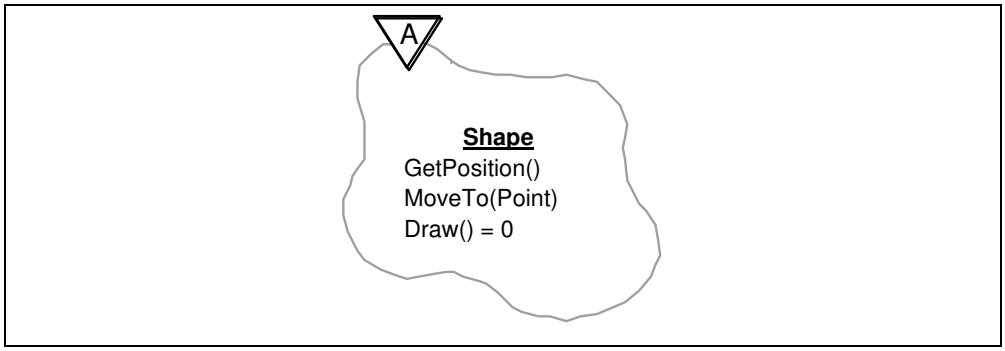

Abbildung 2-8: `Shape` als abstrakte Klasse

```
1. class shape
2. {
3. ...
4.          virtual void Draw() = 0;
5. ...
6. };
```

Listing 2-4: `Draw` als rein virtuelle Methode macht `Shape` abstrakt

Eine andere Form des Polymorphismus wird in der Literatur auch als "Ad hoc-Polymorphismus"[12] bezeichnet, der in C++ "Überladen von Funktionen" genannt wird. Dabei ist es möglich für einen Funktions- bzw. Methodennamen mehrere Implementierungen vorzusehen, die abhängig von den Funktionsparametern ausgewählt werden. In C++ wird der Rückgabewert für die Unterscheidung jedoch bis jetzt nicht mit einbezogen.[13] Sogar die Kombination beider Polymorphie-Arten ist in C++ möglich.

2.1.6 Entwurfsmuster

> Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.[14]

Gamma et al. erkennen diese Definition, die Alexander auf architektonische Begriffe bezieht, in [Gamma95] als auf objektorientierten Entwurf anwendbar. Sie beschreiben allgemein Mustername, Problemabschnitt, Lösungsabschnitt und Konsequenzabschnitt als

12 Vgl. [Booch95], S. 151.

13 Möglicherweise ist die Berücksichtigung des Rückgabewertes in den während dieser Arbeit laufenden Standardisierungsprozeß der Sprache C++ mit eingeflossen.

14 [Alexa77] zitiert in [Gamma95], S. 2

grundlegende Elemente eines Musters. Sie teilen Muster in die drei Kategorien ihrer Aufgabenbereiche ein: Erzeugungsmuster, Strukturmuster und Verhaltensmuster.

Das Studium der in [Gamma95] dargestellten Entwurfsmuster hat in dieser Arbeit in einigen Fällen eine gute Anregung für die getroffenen Entwurfsentscheidungen gegeben. Eine Darstellung der entsprechenden Muster sowie der Konzepte kann im Rahmen dieser Arbeit nicht geleistet werden, weshalb ich an dieser Stelle auf [Gamma95] verweise und all jenen, die sich mit objektorientiertem Entwurf befassen, als Lektüre wärmstens empfehlen möchte.

2.2 Weltbilder der zeitdiskreten Simulation

Simulation dient der Gewinnung von Erkenntnissen über ein reales System. Von diesem wird ein Modell angefertigt, an dem Experimente durchgeführt werden. Anhand der Ergebnisse der Experimente können Rückschlüsse auf das reale System gezogen werden.

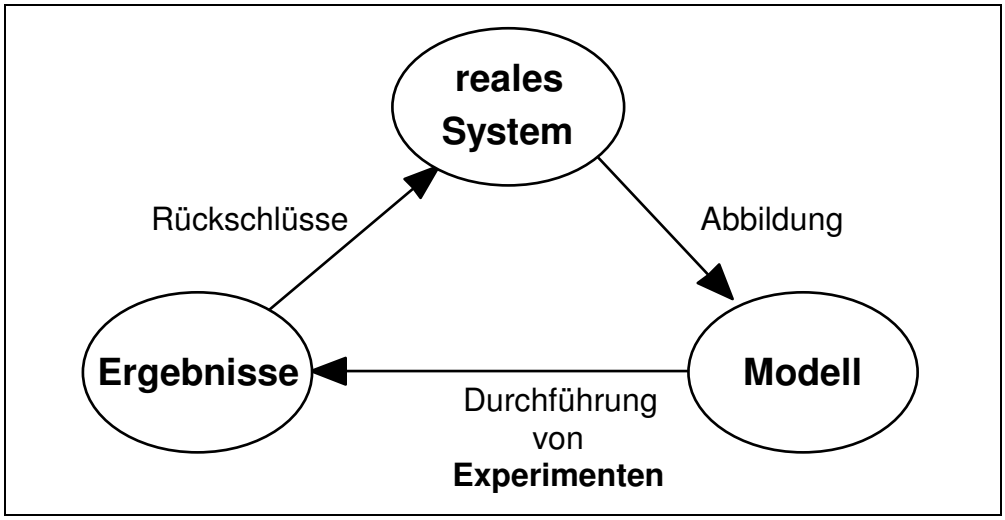

Abbildung 2-9: Beziehungen zwischen System, Modell und Experiment[15]

Diskrete Simulation für eine Klasse von Modellen, bei denen sich der Systemzustand zu bestimmten Zeitpunkten sprunghaft ändert. Für die Abbildung des realen Systems auf das Modell haben sich in bezug auf zeitdiskrete Simulation vier grundsätzliche Sichtweisen etabliert: ereignis-, prozeß-, transaktions- und aktivitätsorientierter Ansatz. Die beiden letztgenannten Ansätze lassen sich auch als Erweiterung der prozeßorientierten Sicht darstellen und sollen deshalb hier nicht näher betrachtet werden. Anstelle des Begriffes Sichtweise wird auch der Begriff Weltbild verwendet.

2.2.1 Ereignisorientierter Ansatz

Der ereignisorientierte Ansatz basiert auf einer materialorientierten Sicht des zu modellierenden Systems. Im Vordergrund steht die Betrachtung von sich durch das System bewegenden Komponenten (Material), die meist als temporäre Objekte (z.B. Werkstücke) von außen in das System gebracht werden, und es irgendwann wieder verlassen. Auf ihrem Weg durch das System werden sie in der Regel von Maschinen, oder allgemein in Bedienstationen, bearbeitet, d.h. ihr Zustand wird verändert. Dabei wird die Zustandsänderung in

[15] In Anlehnung an Abb. 1-5 und Abb. 1-7 aus [Page91].

der Regel dem Zeitpunkt des Bearbeitungsendes zugeschrieben. Solche Zustandsänderungen werden dargestellt durch Ereignisse, auf deren Eintreten das Modell reagiert. Die Behandlung eines Ereignisses bewirkt eine Änderung des Modellzustands. Damit ist der gesamte zeitliche Verlauf des Modellzustandes darstellbar durch eine Kette von Ereignissen und der Beschreibung, wie das Modell auf sie reagiert.

In zeitdiskreten Simulatoren werden Ereignisse intern in einer Datenstruktur verwaltet, die unabhängig von ihrer Realisierung, z.B. als Baumstruktur, oft Ereignisliste genannt wird. Sie ermöglicht den Zugriff auf das Ereignis mit dem kleinsten Ereigniszeitpunkt. Dies ist dann jeweils das nächste eintretende Ereignis. Im ereignisorientierten Ansatz sind die Ereignisse gewissermaßen der Motor der Simulation. Üblicherweise entstehen durch die Behandlung eines Ereignisses einzelne oder eine ganze Reihe von Folgeereignissen. Wäre dies nicht so, würde der Strom von Ereignissen abreißen und die Simulation zum Stillstand kommen.

Beim Eintreten eines Ereignisses wird zunächst die Simulationsuhr auf den Zeitpunkt des Ereignisses vorgestellt. In Abhängigkeit vom Typ des Ereignisses werden entsprechende Behandlungsmaßnahmen ergriffen, um die Zustandsänderungen vorzunehmen. Hierbei wird keine Simulationszeit verbraucht. Der Zeitverbrauch von Aktionen im Realsystem ergibt sich im Modell erst aus der Zeitdifferenz zweier Ereignisse.

Ein Beispiel soll die Zusammenhänge verdeutlichen:

> Ein Werkstück soll in einer Maschine bearbeitet werden. Bei seiner Ankunft ist es unbearbeitet. Nach der Bearbeitung in der Maschine verläßt es das System als bearbeitetes Werkstück. Der Zustand des Werkstücks ist charakterisiert durch seinen Bearbeitungszustand[16]. Zunächst kommt das Werkstück zum Zeitpunkt t_1 im System an, was durch das Ereignis E_1 “Ankunft des Werkstücks” modelliert wird. Als Reaktion auf das Eintreten des Ereignisses wird die Simulationsuhr auf den Zeitpunkt t_1 vorgestellt und das Ereignis behandelt, indem im Modellzustand vermerkt wird, daß die Maschine belegt ist. Außerdem wird ein neues Ereignis E_2 “Bearbeitungsende des Werkstücks” zum Zeitpunkt $t_2 = t_1 + \Delta t$ angesetzt (Δt ist hier die Bearbeitungsdauer). Damit ist E_1 vollständig behandelt. Während der Behandlung des Ereignisses ist keine Simulationszeit vergangen.
>
> Nachdem bis zum Zeitpunkt t_2 alle Ereignisse behandelt wurden, tritt das Ereignis E_2 “Bearbeitungsende des Werkstücks” ein. Die Simulationsuhr wird auf t_2 vorgestellt und nun spiegelt sich der Zeitverbrauch der Bearbeitung in der Simulationszeit wider. Während der Behandlung von E_2 wird die Maschine als frei markiert und der Bearbeitungszustand des Werkstücks von unbearbeitet auf bearbeitet gesetzt. Ferner wird das Werkstück aus dem System entfernt.

Im Beispiel ist ein Ereignis immer mit einem Werkstück, einem temporären Objekt, assoziiert (Ankunft des Werkstücks). Dies ist in ereignisorientierten Modellen meist der Fall, es muß aber nicht so sein. Z.B. wäre auch ein Ereignis “Stromausfall” denkbar, das alle Maschinen des Modells betreffen könnte. Im Beispiel ist der Ereignistyp also “Ankunft eines Werkstücks”. Durch die Assoziation eines Ereignisses von diesem Typ mit einem ganz

[16] Dies ist jedoch nicht der vollständige Zustand des Werkstücks, da die Information, wo es sich innerhalb des Systems befindet, und wie es zu anderen Objekten in Beziehung steht, ebenfalls eine Rolle spielt.

bestimmten Werkstück erhält das Ereignis seinen eigenen Charakter, wodurch es sich von anderen Ankunftsereignissen abhebt[17].

2.2.2 Prozeßorientierter Ansatz

Im prozeßorientierten Ansatz ist die Sichtweise eine ganz andere. Hier werden nicht Zustandsänderungszeitpunkte modelliert. Statt dessen werden ganze Handlungsstränge der unterschiedlichen aktiven Systemkomponenten betrachtet, die sich über einen ganzen Simulationszeitraum erstrecken, wenn nicht sogar über die gesamte Dauer des Experiments. Für den Modellentwickler bietet dieser Ansatz die Möglichkeit, sich in die Objekte des Systems hineinzuversetzen, um zu beschreiben, wie sie sich verhalten und auf den Systemzustand einwirken. Die aktiven Komponenten des Systems werden als Objekte mit Attributen und einer Handlungsbeschreibung modelliert. Der Zeitverbrauch der Objekthandlungen im Modell entspricht dem im realen System. Das Gesamtverhalten des Modells ergibt sich aus den Interaktionsbeziehungen der in ihm wirkenden Komponenten, welche sich jeweils in aktiven oder passiven Phasen befinden.[18] Ihre Aktivierungszeitpunkte sind vergleichbar mit den Ereigniszeitpunkten des ereignisorientierten Ansatzes, wodurch dem prozeßorientierten Ansatz ein ereignisorientierter Mechanismus zugrunde gelegt werden kann.

Auch hier soll das Beispiel aus dem vorangegangen Abschnitt dargestellt werden:

> Als aktive Komponenten werden sowohl das Werkstück als auch die Maschine modelliert. Es gilt nun, diese beiden Komponenten und ihr Zusammenwirken näher zu betrachten.
>
> Ein Werkstück hat wie zuvor das Attribut, das seinen Bearbeitungszustand widerspiegelt. Es wird “von außen” in das System gebracht und aktiviert. Damit beginnt es, seiner Handlungsbeschreibung zu folgen: es aktiviert die Maschine, um sie zur Bearbeitung aufzufordern, und wartet auf das Ende der Bearbeitung. Dieses Warten äußert sich im Verstreichen von Simulationszeit und stellt eine passive Phase des Werkstücks dar. Wenn es danach wieder aktiviert wird, bedeutet das, daß es fertig bearbeitet ist. Das Werkstück kann also sein Attribut entsprechend ändern und sich aus dem System entfernen.
>
> Die Maschine hingegen beginnt mit einer passiven Phase: sie wartet auf das Eintreffen eines Werkstücks. Wird sie aktiviert, bedeutet dies, daß ein eingetroffenes Werkstück bearbeitet werden soll. Die Maschine setzt ihr Attribut auf “belegt” und läßt die Bearbeitungszeit verstreichen, wodurch tatsächlich auch Simulationszeit verbraucht wird. Dies ist die aktive Phase der Maschine. Nach der Bearbeitung, setzt sie ihr Attribut auf “frei” zurück, aktiviert das gerade bearbeitete Werkstück und wartet auf das nächste. Die Maschine beginnt ihre Handlungen wieder von vorne, also mit ihrer passiven Phase.

Im Beispiel wird deutlich, daß es sowohl temporäre als auch statische Komponenten gibt. Das Werkstück durchläuft seine Handlungsbeschreibungen genau einmal und entfernt sich wieder aus dem System, hat also temporären Charakter. Die Maschine hingegen ist durch

---

17 Natürlich spielt der Ereigniszeitpunkt dabei auch eine Rolle.

18 Die Verwendung der Begriffe ‘aktiv’ und ‘passiv’ kann leicht zu Verwirrung führen, da ihre Bedeutung variiert, je nachdem in welchem Zusammenhang sie verwendet werden. Dieses Problem wird in Abschnitt 3.1.3 (S. 28) näher betrachtet.

einen Zyklus von Handlungen bestimmt, und verbleibt für die gesamte Dauer des Experiments im System.

2.3 DESMO

DESMO (Discrete Event Simulation in Modula-2) wurde 1987-89 am Fachbereich Informatik der Universität Hamburg als eine Modulsammlung für die Sprache Modula-2 entwickelt. Seit seiner Einführung am Fachbereich Informatik wurde das Paket intensiv in der Lehre eingesetzt, um den Studierenden die Konzepte der zeitdiskreten Simulation nahezubringen. Es basiert auf den für prozeßorientierte Simulation nutzbaren Konzepten von DEMOS (Discrete Event Modelling On Simula), das 1979 von G. Birtwistle vorgestellt wurde. Dadurch, daß bei DESMO die Wahl der Programmiersprache auf Modula-2 gefallen war, mußten die objektorientierten Konzepte des Vorbildes mit den Mitteln einer prozeduralen Sprache nachgebildet werden. Für die Schnittstelle entstand eine Reihe von Modulen, von denen jedes einen abstrakten Datentyp[19] anbietet, der die Funktionalität der entsprechenden DEMOS-Klasse realisiert. Zusätzlich wurde in DESMO der ereignisorientierte Ansatz implementiert. Die Funktionalität soll hier kurz skizziert werden, für eine ausführliche Beschreibung sei auf [Bölck89] und [Page91] verwiesen.

2.3.1 Umsetzung des ereignisorientierten Ansatzes

Die Funktionalität für den ereignisorientierten Modellierungsstil wird in DESMO vom Modul `EventSimulation` bereitgestellt, mit dessen Import sich der Modellentwickler auf diesen Stil festlegt. Temporäre Simulationsobjekte werden als “Entity” bezeichnet. Hierfür wird ein abstrakter Datentyp angeboten. Die Attribute eines Entities müssen in einer vom Modellentwickler zu definierenden Datenstruktur angelegt und bei der Erzeugung eines Entities angegeben werden. Im weiteren Verlauf kann auf die Attribute über die Prozedur `Attributes` zugegriffen werden. Die verschiedenen Ereignistypen werden vom Modellentwickler als Konstanten eines Aufzählungstyps definiert. Mit Hilfe der bereitgestellten Datenstruktur `EventDescriptor` wird ein Ereignistyp mit den Aktionen zur Behandlung eines Ereignisses dieses Typs in Verbindung gebracht. Hierzu enthält der `EventDescriptor` u.a. eine Prozedurvariable, in der die vom Modellentwickler zu formulierenden Aktionen in Form einer Prozedur mit einem `Entity`-Parameter, bezeichnet als “Ereignisroutine”, abgelegt wird. D.h. über den `EventDescriptor` wird ein Ereignistyp mit genau einer Ereignisroutine assoziiert. Der Simulationssteuerung wird beim Start ein Array[20] von Event-Deskriptoren übergeben, womit das Modellverhalten spezifiziert ist.

Das Modul `EventSimulation` bietet Prozeduren an, um Ereignisse anzusetzen. Mittels `Schedule`-Prozeduren wird ein Ereignis zusammen mit einem Entity für einen bestimmten Simulationszeitpunkt vorgemerkt. Dabei müssen der Ereignistyp, das Entity und das Zeitintervall bis zu diesem Zeitpunkt als Parameter übergeben werden. Ist der Zeitpunkt erreicht, so wird über den Ereignistyp mit Hilfe des Event-Deskriptors die entsprechende Ereignisroutine ermittelt und aufgerufen. Ihr wird das assoziierte Entity als Parameter ü-

[19] Ein abstrakter Datentyp ist ein Mittel zur Strukturierung, bei dem ein Datentyp mit den auf ihm operierenden Prozeduren gekapselt wird (vgl. [DalCi89]).

[20] Das englische Wort ‘array’ wird meist mit ‘Feld’ übersetzt. Da die deutsche Bezeichnung jedoch sehr allgemein ist und in verschiedenen Zusammenhängen sehr unterschiedliche Bedeutungen haben kann, ziehe ich die Verwendung des englischen Wortes vor.

bergeben. Innerhalb der Ereignisroutine kann mittels `Attributes` auf die Attribute diees Entities zugegriffen werden.

Die Angabe des Ereignistyps wird nur zur Erzeugung neuer Ereignisse benötigt. Alle weiteren Prozeduren sind auf Entities bezogen. Neben den benutzerdefinierten Attributen besitzt ein Entity auch abfragbare Systemattribute: `Priority` liefert die Priorität eines Entity, mit `Idle` läßt sich abfragen, ob das Entity vorgemerkt ist, `EvTime` liefert den Zeitpunkt des nächsten Ereignisses[21], das in Bezug auf das Entity eintritt. `Current` liefert das Entity, das mit dem gerade behandelten Ereignis assoziiert ist. Es befindet sich nicht mehr auf der Ereignisliste.

Ein Simulationslauf wird mit Hilfe der Prozedur `StartSimulation` gestartet. Er läuft so lange bis `StopSimulation` aufgerufen wurde, die angegebene Simulationszeit verstrichen ist oder die Ereignisliste leer ist. Eine leere Ereignisliste wird dabei als Fehlersituation betrachtet. Durch den Aufruf von `Report` läßt sich eine Datei erzeugen, in der die Ergebnisse des Simulationslaufes präsentiert werden.

2.3.2 Umsetzung des prozeßorientierten Ansatzes

Der prozeßorientierte Modellierungsstil wird in DESMO vom Modul `ProcessSimulation` angeboten. Entities repräsentieren aktive Komponenten des zu modellierenden Systems. Dabei wird das Konzept des Entities aus dem ereignisorientierten Ansatz erweitert um eine lokale Ablaufkontrolle, die es ermöglicht, aus der Sicht des Entities zu beschreiben, wie es durch seine Aktivitäten auf das System und seine Bestandteile einwirkt. Die Aktivitäten stellen zusammen mit den Entity-Attributen einen Prozeß dar. Zur Erzeugung eines neuen Prozesses müssen seine Attribute erzeugt und zusammen mit einer Prozedur angegeben werden, die das Ablaufschema des Prozesses beschreibt. Ein Prozeßtyp ist also charakterisiert durch diese Prozedur, im folgenden Prozeßroutine genannt, und seine Attribute.

Der Prozeßroutine kann die Kontrolle übergeben werden, indem eine der `Schedule`-Prozeduren aufgerufen wird. Als Parameter müssen nur das Entity, dessen Ablaufkontrolle aktiviert werden soll, und der Zeitpunkt, zu dem dies geschehen soll, übergeben werden. Eine Prozeßroutine kann die Kontrolle auf drei verschiedene Arten wieder abgeben:

1. Sie ruft die Prozedur `Hold` auf, wodurch die Kontrolle abgegeben wird. Zusätzlich wird eine Reaktivierung der Prozeßroutine in einer anzugebenden Zeitspanne vorgemerkt. Mit `Hold` werden in der Regel aktive Phasen des Prozesses modelliert (z.B. Bearbeitung eines Werkstücks).

2. Sie ruft die Prozedur `Passivate` auf, wodurch die Kontrolle für unbestimmte Zeit abgegeben wird. Der Prozeß kann sich dann “aus eigener Kraft” nicht mehr aktivieren, sondern muß von anderer Stelle angestoßen werden, um weiter arbeiten zu können. Mit `Passivate` werden meist die passiven Phasen des Prozesses modelliert (z.B. Warten auf Bearbeitung).

3. Die Prozeßroutine erreicht ihr Ende, wodurch die Kontrolle implizit abgegeben wird. Der Prozeß gilt fortan als terminiert und kann nicht mehr aktiviert werden. Dieser Mechanismus wird für temporäre Simulationsobjekte (z.B. Werkstücke) benutzt, die sich

[21] Bei DESMO wird dies Aktivierungszeitpunkt genannt, da Entities hier als aktive Objekte betrachtet werden. (s. [Page91] S. 179)

im Vergleich zu stationären oder permanenten Komponenten (z.B. Maschinen) nur relativ kurz im System aufhalten.

Im Gegensatz zum ereignisorientierten Ansatz verbleibt das Entity, dessen Prozeßroutine gerade aktiv ist, auf der Ereignisliste und es wird als das "aktive Entity" bezeichnet. Die Prozeduren `Hold` und `Passivate` beziehen sich immer auf das aktive Entity, unabhängig davon, in welchem Kontext sie aufgerufen werden. Außerdem sind Prozesse unterbrechbar, indem die Prozedur `Interrupt` unter Angabe einer Unterbrechungsursache in Form eines Interrupt-Codes aufgerufen wird. Dieser wird solange in den System-Attributen des Prozesses gespeichert, bis der Prozeß erneut unterbrochen wird oder `ClearInterrupt` aufgerufen wird. Eine Unterbrechung wirkt ähnlich wie eine Aktivierung des Prozesses, jedoch kann über die Prozedur `Interrupted` der Grund für die Unterbrechung abgefragt werden.

Ein Simulationslauf wird gestartet, indem der Hauptprozeß, ein spezieller Prozeß, der das Hauptprogramm repräsentiert, deaktiviert wird (mittels `Hold` oder `Passivate`). Daraufhin wird dem ersten auf der Ereignisliste stehenden Prozeß die Kontrolle übergeben (oder genauer: dessen Prozeßroutine). Zum Beenden kann die Kontrolle wieder dem Hauptprozeß übergeben werden. Ein Leerlaufen der Ereignisliste führt zum Programmabbruch. Auch im prozeßorientierten Ansatz läßt sich durch den Aufruf von `Report` eine Datei erzeugen, in der die Ergebnisse des Simulationslaufes präsentiert werden.

2.3.3 Gemeinsame Funktione beider Ansätze

2.3.3.1 Warteschlangen

Für Warteschlangen gibt es in DESMO zwei Module (`eQueue` bzw. `Queue`), die beide dieselbe Funktionalität umsetzen, aber jeweils nur mit einem der beiden Ansätze (ereignis- bzw. prozeßorientiert) verwendbar sind. Sie unterscheiden sich im Typ der Entities, die in eine Warteschlange eingereiht werden können. Es werden Prozeduren zum Einfügen, Entfernen und Finden von Entities in Warteschlangen angeboten. Dabei muß die jeweilige Warteschlange stets als Parameter mit angegeben werden. Es wird eine automatische Statistik über die Benutzung der Warteschlange geführt.

Zu einem in einer Warteschlange wartenden Entity können über die Prozeduren `Next` bzw. `Succ` sowohl Nachfolger als auch Vorgänger ermittelt werden. Eine Suche nach bestimmten Entities in einer Warteschlange ist über die Prozeduren `Find` und `FindNext` möglich. `Find` wird eine Prozedur mit einem Entity-Parameter und booleschem Rückgabewert übergeben, die die Suchbedingung spezifiziert. Wird ein Entity gefunden, so wird es im VAR-Parameter zurückgegeben. Zusätzlich wird die Suchbedingung in der die Warteschlange repräsentierenden Datenstruktur gespeichert, um mit `FindNext` an der letzten Fundstelle mit der Suche fortfahren zu können.

2.3.3.2 Zufallszahlenströme

Zur Erzeugung von reproduzierbaren Pseudo-Zufallszahlen bietet DESMO drei Module an (`BoolDist`, `IntDist` und `RealDist`), die sich jeweils im Typ der generierten Zahl unterscheiden. Jedes von ihnen stellt einen abstrakten Datentyp zur Verfügung, der Zufallszahlenströme mit jeweils unterschiedlichen zugrundeliegenden Verteilungsformen repräsentieren kann. Für jeden dieser Verteilungsformen wird eine Prozedur angeboten, die einen entsprechenden Zufallszahlenstrom erzeugt, und in den späteren `Sample`-Aufrufen als Parameter übergeben werden muß, um eine neue Zufallszahl zu ziehen. Alle Zufallszah-

lenströme arbeiten intern mit demselben Generator, der Zahlen zwischen 0 und 1 erzeugt. Um größtmögliche Unabhängigkeit der Zufallszahlenströme untereinander zu gewährleisten, erhält jeder von ihnen automatisch einen Startwert, der gewährleistet, daß der interne Generator an einer jeweils anderen Stelle seiner Periode aufsetzt. Dadurch daß die Zufallszahlen reproduzierbar sind, lassen sich unterschiedliche Simulationsläufe direkt miteinander vergleichen. Außerdem wird die Erzeugung von antithetischen[22] Zufallszahlen angeboten.

2.3.3.3 Statistische Datensammelobjekte

Unterstützung zur Automatisierung von Auswertungen statistischer Daten bieten die Module `Accumulate`, `Tally`, `Histogram`, `Count` und `TimeSeries`. Jedes von ihnen stellt einen abstrakten Datentyp zur Verfügung, der mittels `Update` explizit aktualisiert werden kann. Mit Ausnahme von `Count`-Objekten muß bei der Erzeugung eine Prozedur angegeben werden, die den Wert einer statistisch auszuwertenden Beobachtungsgröße liefert. Bei `Accumulate`-Objekten ist aufgrund der zeitlichen Gewichtung die automatische Aktualisierung mit jeder Änderung der Simulationszeit wählbar. Ein `TimeSeries`-Objekt protokolliert die Beobachtungsgröße zusammen mit dem Zeitpunkt des `Update`-Aufrufs in eine Datei. `Count` bietet einfache Zählfunktionen an.

2.3.3.4 Report

Alle ansatzunabhängigen Komponenten sowie die im folgenden Abschnitt dargestellten Erweiterungen des prozeßorientierten Ansatzes werden durch den Aufruf der Prozedur `Report` automatisch in einer Datei aufgeführt. Dort werden die für die jeweiligen Objekte interessierenden Daten tabellarisch aufgelistet. Für diesen Zweck stellt jedes Modul Prozeduren zur Verfügung, die die Ausgabe für die in dem jeweiligen Modul definierten Datentypen vornehmen. Diese Prozeduren können vom Modellentwickler durch eigene ersetzt werden, um auf die Gestaltung der Reportausgabe Einfluß nehmen zu können.

2.3.4 Erweiterungen des prozeßorientierten Ansatzes

Für den prozeßorientierten Ansatz stellt DESMO höhere Modellierungskonstrukte zur Verfügung, mit denen sich Modelle auch im transaktions- und aktivitätsorientierten Stil entwerfen lassen. Jedes dieser Konstrukte ist in einem eigenen Modul gekapselt und wird mit Entities aus dem Modul `ProcessSimulation` verwendet:

- **Ressourcenwettbewerb** (Modul `Res`) kann in DESMO dargestellt werden, indem für einen Ressourcenpool ein neues Objekt erzeugt wird, das eine begrenzte Anzahl von einzelnen Ressourcen vorhält. Diese können mittels `Acquire` von Entities angefordert werden. Sind nicht genügend Ressorcen vorhanden, so wird das Entity in eine implizite Warteschlange eingereiht und solange blockiert, bis die Anforderung erfüllt werden kann. Nicht mehr benötigte Ressourcen können mittels `Release` wieder zurückgegeben werden. DESMO überwacht Anforderungen und Rückgaben und gibt bei möglichen Verklemmungssituationen Warnungen aus. Diese Überwachung kann auf verschiedene Stufen eingestellt werden: `Off` - es findet keine Überwachung statt, `Static` (Vorgabe) - prüft gemäß Deklarationsreihenfolge, `DynamicA` - Prüfung auf Zyklenfreiheit im Ressourcen-Allokationsgraph bei jedem Belegungsversuch, `DynamicB` - wie `DynamicA`, jedoch nur, wenn auf die Zuteilung gewartet werden muß.

---

[22] Zu Maßnahmen zur Varianzreduzierung durch die Verwendung von antithetischen Zufallszahlen s. [Page91].

- **Produzenten-/Konsumenten-Beziehungen** (Modul `Bin`) ermöglichen die Synchronisation von Entities über Puffer. Dabei wird nur die Anzahl der im Puffer vorhandenen Produkte betrachtet, nicht jedoch deren Art oder Identität. Mittels `Give` können Entities Produkte in einen Puffer geben. `Take` dient der Entnahme von Produkten, wobei Entities in eine implizite Warteschlange eingereiht und blockiert werden, sofern nicht genügend Produkte vorhanden sind.

- **Bedingtes Warten** (Modul `CondQ`) bietet die Möglichkeit, Entities solange zu blockieren, bis eine anzugebende Bedingung erfüllt ist. Um auf die Erfüllung einer Bedingung zu warten reiht sich ein Entity unter Angabe der Bedingung mittels `WaitUntil` in eine Warteschlange für bedingtes Warten ein. Etwaige Änderungen der Bedingung müssen der Warteschlange explizit mittels `Signal` mitgeteilt werden.

- **Direkte Prozeßkooperation** (Modul `WaitQ`) ermöglicht, gemeinsame Handlungen zweier Entities durch eine Master-Slave-Beziehung zu modellieren. Dabei behält der Master während der Kooperation die Kontrolle, der Slave verhält sich passiv bis zum Ende der Kooperation. Ein Entity kann einem `WaitQ`-Objekt den Wunsch, als Slave mit einem Master zu kooperieren, durch Aufruf der Prozedur `Wait` mitteilen. Es wird dann solange blockiert, bis sich ein kooperationswilliger Master einfindet. Im Gegensatz dazu signalisiert der Aufruf von `CoOpt`, daß das Entity bei der Kooperation die Rolle des Masters übernehmen will. Auch hier findet eine Blockierung statt, solange kein Slave verfügbar ist. Der Master muß bei der Äußerung seines Kooperationswunsches eine Prozedur mit angeben, die die gemeinsamen Handlungen beschreibt. Dieser Prozedur werden dann beim Zustandekommen der Kooperation Verweise auf Master- und Slave-Entity als Parameter übergeben, so daß deren Attribute innerhalb der gemeinsamen Handlungen manipuliert werden können. Zusätzlich kann ein Master eine Bedingung angeben, die für einen Slave erfüllt sein muß, damit die Kooperation zustande kommt. Dies ist über die Prozedur `Find` realisiert, die gegenüber `CoOpt` einen zusätzlichen Parameter für die Bedingung hat.

2.4 SiFrame

Mit SiFrame (Simulation Framework) wurde 1996 die Funktionalität von DESMO als ein Rahmenwerk für Simulationsprogramme in C++ implementiert. Es stellt sowohl die ereignis- als auch die prozeßorientierten Konstrukte zur Verfügung. Darüber hinaus wurde die in [Ritsc91] entwickelte Erweiterung zur kontinuierlichen Simulation integriert, die in dieser Arbeit jedoch nicht näher betrachtet wird. In diesem Abschnitt sollen die wesentlichen Aspekte von SiFrame in Kontrast zu DESMO gesetzt werden. Für detailliertere Informationen zu SiFrame sei auf [Weber96] verwiesen.

Auch in SiFrame sind Entities hauptsächlich durch ihre Attribute charakterisiert. Die Klasse `SystemEntity` faßt Eigenschaften zusammen, die in beiden Modellierungsstilen (ereignis- und prozeßorientiert) verwendet werden. Von ihr werden die Klassen `EventEntity` und `ProcessEntity` abgeleitet, die für den jeweiligen Ansatz benötigte zusätzliche Systemattribute hinzufügen, und dem Modellentwickler als Basisklasse für seine modellspezifischen Entity-Klassen dienen (s Abbildung 2-10).

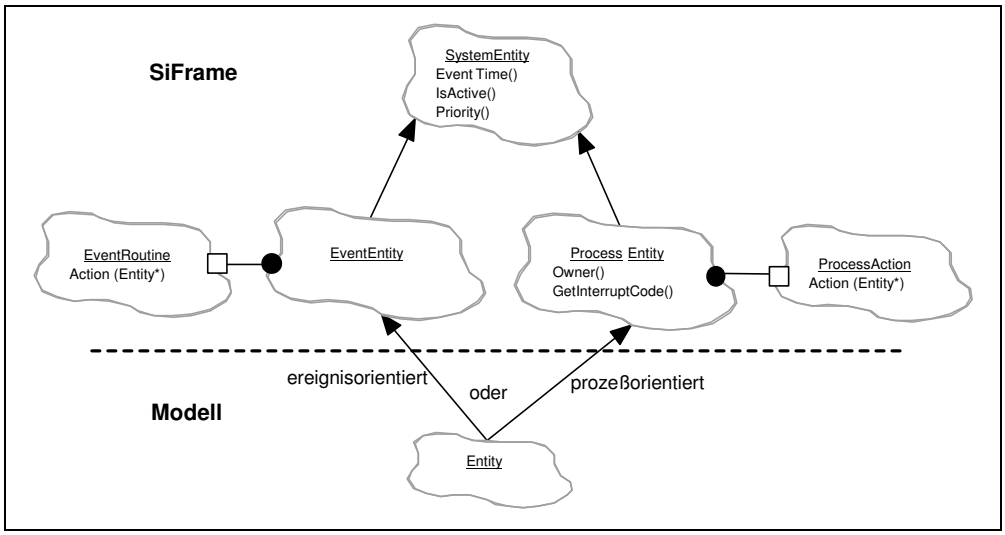

Abbildung 2-10: Entities in SiFrame

Einen anderen wichtigen Zweig in der Klassenhierarchie stellt die Klasse `Simulation-Object` dar (s. Abbildung 2-11), von der aktive Simulationsobjekte für Ereignisroutinen oder Prozeßaktionen abgeleitet sind. Außerdem sind hier die Klassen `MainEventSimulation` bzw. `MainProcessSimulation` für die das Hauptprogramm repräsentierenden Objekte zu finden. Dabei muß der Modellentwickler genau ein solches Objekt deklarieren, in dessen Methode `Init` die ersten Ereignislisteneinträge erfolgen und der Simulationslauf in Gang gesetzt werden muß. Da die C-Funktion `main`[23] in SiFrame integriert ist, reicht die Deklaration eines solchen Objektes aus. Die Methode `Init` wird automatisch aufgerufen.

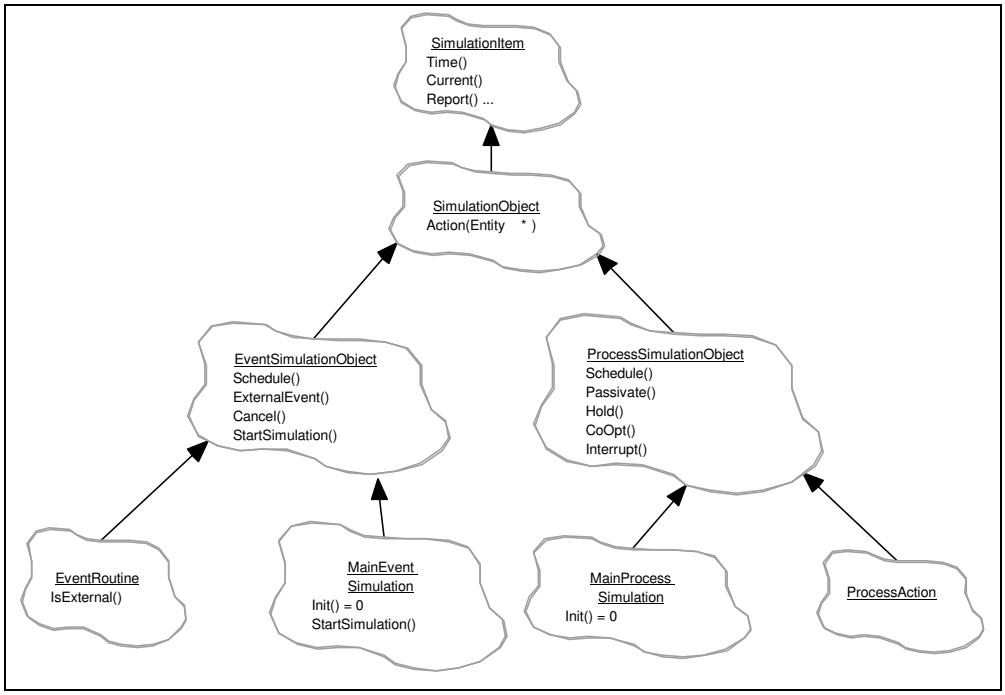

Abbildung 2-11: Aktive Simulationsobjekte in SiFrame

[23] In C bzw. C++ ist die Funktion `main` der Rumpf des Programms. Er wird ausgeführt, nachdem alle globalen Objekte initialisiert sind.

2.4.1 Ereignisorientierter Ansatz

Um eigene Entitytypen zu realisieren, muß der Modellentwickler die Klasse `Entity` definieren, die von der Klasse `EventEntity` abgeleitet sein muß (s. Abbildung 2-10) und zur Darstellung der Anwenderattribute des Entities dient. Ereignisse treten als solche gar nicht in Erscheinung. Es werden lediglich die Ereignisroutinen als Unterklassen von `EventRoutine` definiert, wobei die virtuelle Methode `Action` zu überschreiben ist, um die Behandlungsmaßnahmen zu beschreiben. Die Unterklassen von `EventRoutine` entsprechen hier also den Ereignistypen. Zum Vormerken von Ereignisroutinen werden von der Oberklasse `EventSimulationObject` Methoden geerbt, denen jeweils eine Referenz auf die Ereignisroutine und das mit dem Ereignis assoziierte Entity sowie ein Simulationszeitpunkt übergeben werden muß. Durch den Aufruf einer solchen Methode entsteht letztlich erst ein Ereignis. Dabei wird ein Verweis auf die Ereignisroutine im Entity gespeichert, so daß nur das Entity auf die Ereignisliste gesetzt werden muß.

Um ein Simulationsxperiment durchzuführen, muß eine Unterklasse von `MainEventSimulation` gebildet werden, von der genau ein Objekt zu erzeugen ist. In der Methode `Init` müssen die ersten Ereignisse angesetzt sowie die Methode `StartSimulation` aufgerufen werden. `Init` wird dann automatisch aufgerufen, sobald die in SiFrame integrierte C-Funktion `main` ausgeführt wird. Mit dem Ende des Simulationsxperiments endet auch das Programm.

2.4.2 Prozeßorientierter Ansatz

Wie im ereignisorientierten Ansatz wird die Klasse `Entity` definiert, jedoch muß sie für die prozeßorientierte Sicht von der Klasse `ProcessEntity` abgeleitet werden. Das Entity an sich dient nur der Aufname der Attribute. Bei der Erzeugung eines `ProcessEntity` muß jedoch ein Zeiger auf ein Objekt vom Typ `ProcessAction` übergeben werden. Dieses Objekt stellt die aktive Komponente, die Handlungen, des Entities dar. So muß also von `ProcessAction` ebenfalls eine Unterklasse gebildet werden, in der die Methode `Action` definiert wird, um die Aktivitäten des Entities zu beschreiben. Dabei erhält `Action` einen Zeiger auf das zugehörige Entity als Parameter, um auf die Entity-Attribute zugreifen zu können.

Zur Durchführung eines Experimentes muß analog zur ereignisorientierten Sicht eine Klasse gebildet werden, die von `MainProcessSimulation` abgeleitet wird. Hiervon wird genau ein Objekt erzeugt, das den Hauptprozeß darstellt. Die Methode `Init` muß definiert werden, um die ersten Prozesse zu aktivieren. Sie wird wie im ereignisorientierten Ansatz automatisch aufgerufen. Um die Simulation in Gang zu setzen, muß sich der Hauptprozeß mittels `Passivate` oder `Hold` für die zu simulierende Dauer zur Ruhe setzen.

2.4.3 Gemeinsame Funktionen beider Ansätze

2.4.3.1 Warteschlangen

Da der Modellentwickler die Klasse Entity, mit der das Simulationspaket arbeitet, selbst definieren muß, reicht in SiFrame eine Klasse für Warteschlangen. Die Funktionalität entspricht den Modulen `eQueue` bzw. `Queue` aus DESMO. Hinzugekommen ist die Möglichkeit, daß Entities in mehr als einer Warteschlange zur Zeit warten können. Dies kann global über die Klassenmethode `SetQueueOption` der Klasse `SystemEntity` eingestellt werden.

Für die Suche in Warteschlangen wird die Klasse `QueueCondition` eingeführt, in deren Unterklassen die Methode `Condition` zu definieren ist, um die Suchbedingung zu formulieren. Den Objekten muß bei der Erzeugung eine Referenz auf eine Warteschlange übergeben werden. Damit ist ein solches Objekt fest mit einer Warteschlange verbunden und speichert die Position der letzten Fundstelle. Zu Beginn ist dies das erste Element der Warteschlange. Mittels `Search` kann das jeweils nächste Entity gefunden werden, das die in der Methode `Condition` zu spezifizierende Bedingung erfüllt. Wurden keine Entities gefunden, so kann die Suchposition mittels `Reset` zurückgesetzt werden.

2.4.3.2 Zufallszahlenströme

Zufallszahlenströme finden in SiFrame ebenfalls ihre Entsprechung. So existiert analog zu den Modulen `BoolDist`, `IntDist` und `RealDist` je eine von `Distribution` abgeleitete Klasse `BooleanDist`, `IntDist` bzw. `RealDist`, von denen ihrerseits jeweils die einzelnen Zufallszahlenströme mit ihren speziellen zugrundeliegenden Verteilungen abgeleitet sind. Als Erweiterung gegenüber DESMO kommen sog. externe Zufallszahlenströme hinzu, die mit einer Datei verbunden werden und zur Verwendung von anderweitig gewonnenen Daten als Zufallszahlen dienen. Der Startwertgenerator ist in der gemeinsamen Oberklasse `Distribution` untergebracht und basiert auf Klassenattributen, d.h. es gibt einen Startwertgenerator für das gesamte Programm.

2.4.3.3 Statistische Datensammelobjekte

Auch für die statistischen Datensammelobjekte gilt, daß jedem Modul aus DESMO eine Klasse in SiFrame entspricht, die die jeweilige Funktionalität realisiert. Bei der Erzeugung eines Datensammelobjektes wird ein Objekt der Klasse `ReportObservation` mit angegeben, das zu den gegebenen Zeitpunkten den Wert der Beobachtungsgröße zu liefern vermag. Hierfür muß eine neue Klasse gebildet werden, die von `ReportObservation` abgeleitet ist. Dabei ist die rein virtuelle Methode `Observation` so zu definieren, daß sie den Wert der Beobachtungsgröße liefert. Eine Ausnahme bilden die Klassen `Count`, der kein solches Objekt zugeordnet wird, und `Regression`, das mit einem Objekt einer von `RegressionObservation` abgeleiteten Klasse verbunden werden muß. Hier ist analog zu `ReportObservation` die Methode `Observation` zu definieren, die jedoch zwei Referenzparameter besitzt, in denen die Werte der Beobachtungsgrößen ausgetauscht werden.

Die gemeinsamen Funktionen der Klassen `Accumulate` und `Tally` wurden in einer Oberklasse `StandardReport` zusammengefaßt (s Abbildung 2-12). Da Histogramme die Datensammelobjekte für nichtzeitgewichtete Statistik um Histogrammfunktionen erweitern, erbt die Klasse `Histogram` von `Tally`.

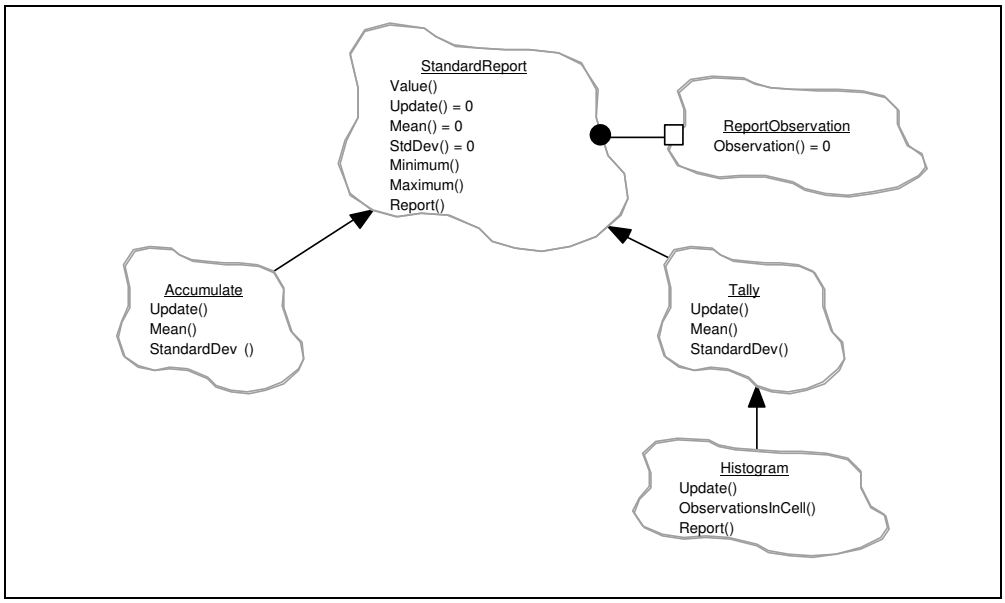

Abbildung 2-12: `Accumulate` und `Tally` in SiFrame

2.4.3.4 Report

Die Reportfunktionalität ist in SiFrame analog zu DESMO in die einzelnen Klassen direkt integriert, indem die Methoden `ReportId`, `Report`, `Header` und `Title` der allen reportfähigen Objekten gemeinsamen Oberklasse `ReportObject` überschrieben werden. Hier wird ein Aufzählungstyp definiert, der für jede innerhalb von SiFrame bekannte Gruppe von Reportobjekten eine eigene Konstante angibt, so daß die einzelnen Objekte nachher im Report richtig gruppiert und in jeweils eigenen Tabellen ausgegeben werden können. Hierfür wird in den Unterklassen die Methode `ReportId` so überschrieben, daß die entsprechende Konstante des Aufzählungstyps zurückgegeben wird. Reportausgaben können durch Unterklassenbildung an eigene Bedürfnisse angepaßt werden.

2.4.4 Erweiterungen des prozeßorientierten Ansatzes

Alle Erweiterungen des prozeßorientierten Ansatzes benötigen zur Verwaltung der blockierten Entities eine (oder zwei) implizite Warteschlangen. Bevor auf die einzelnen Klassen näher eingegangen wird, soll zunächst die Realisierung dieses Teilproblems geschildert werden.

Das Problem ergibt sich aus der Tatsache, daß die Funktionalität der Warteschlange geerbt werden könnte, damit jedoch auch das direkte Einfügen bzw. Entfernen von Entities in die bzw. aus der impliziten Warteschlange möglich wäre, was unbedingt zu vermeiden ist. In SiFrame gibt es für jede der vier Klassen `Bin`, `CondQ`, `Res` und `WaitQ` jeweils eine Unterklassen der Klasse `Queue` für die Implementierung der impliziten Warteschlange. Diese Unterklassen (`bQueue`, `cQueue`, `rQueue` und `wQueue`), die bis auf den Klassennamen exakt übereinstimmen, überschreiben lediglich die Methode `ReportId`, so daß ein Wert geliefert wird, der anzeigt, daß das Objekt nicht im Report erscheinen soll. Es wird eine neue Klasse `qBased` eingeführt, die direkt von `ReportObject` erbt, und mit einer Warteschlange verbunden ist. `qBased` bildet die Funktionen für die Warteschlangenstatistik auf die implizite Warteschlange ab.

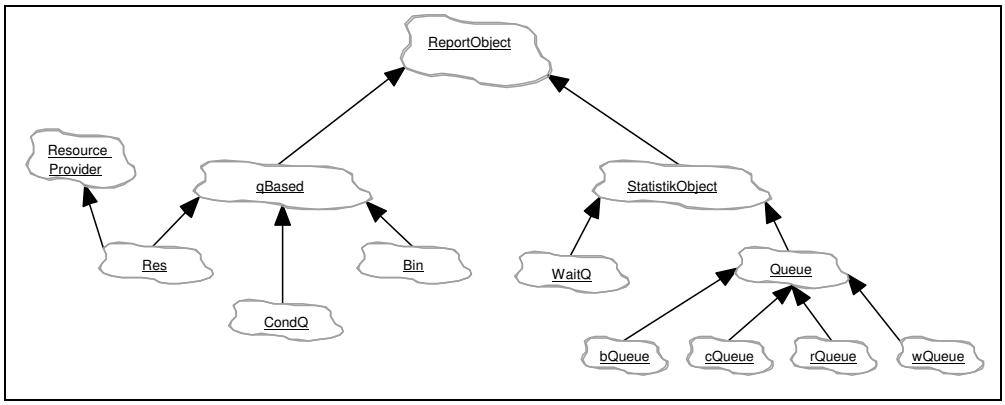

Abbildung 2-13: Warteschlangen und darauf basierte Objekte in SiFrame

`Bin`, `CondQ` und `Res` erben von `qBased` und generieren bei ihrer Erzeugung eine implizite Warteschlange (`bQueue`, `cQueue` bzw. `rQueue`), die sie an die Oberklasse weiterreichen. `WaitQ` hingegen erbt genau wie `Queue` von `StatistikObject`. Dabei enthält `WaitQ` zwei Objekte der Klasse `wQueue`, wobei die von `StatistikObject` geerbten Methoden für die Warteschlangenstatistik ungenutzt bleiben und keine sinnvollen Werte liefern.

- **Ressourcenwettbewerb** wird in SiFrame über die Klasse `Res` realisiert. `Res` erbt einerseits von `qBased` die Funktionen zur Warteschlangenstatistik und andererseits durch die Verwendung der Mehrfachvererbung von `ResourceProvider` die Schnittstelle, die für die Zusammenarbeit mit der internen Ressourcedatenbank benötigt wird. Diese Datenbank speichert Anforderungen, Belegungen und Freigaben von einzelnen Ressourcen und ist in der Lage, Deadlock-Situationen zu erkennen. Es ist jedoch nur die Prüfung auf der Stufe `DynamicA` implementiert. Um- oder Ausschalten der Überwachung ist nicht möglich.

- **Produzenten-/Konsumenten-Beziehungen** werden mit Hilfe der Klasse `Bin` modelliert, die dem gleichnamigen DESMO-Modul entspricht.

- **Bedingtes Warten** (Klasse `CondQ`) erfordert vom Modellentwickler beim Aufruf von `WaitUntil`, die Bedingung als Objekt einer Unterklasse von `CondQCondition` anzugeben. Dabei muß die Methode `Condition` entsprechend definiert werden, um die Bedingung auszudrücken. Von `QueueCondition` unterscheidet sich `CondQCondition` jedoch deutlich, da hier keine Position gespeichert wird, der Suchmechanismus folglich anders umgesetzt ist.

- **Direkte Prozeßkooperation** wird mit Hilfe der Klasse `WaitQ` modelliert. Der Modellentwickler muß für die Beschreibung der gemeinsamen Handlungen eine Unterklasse von `EntityCoOperation` bilden, in der er die Methode `CoOperation` definiert. `CoOperation` entspricht der in DESMO zu definierenden Prozedur für die Kooperation zweier Entities. Für die Auswahl eines bestimmten Kooperationspartners muß eine Bedingung in Form eines Objektes einer von `WaitQCondition` abgeleiteten Unterklasse angegeben werden. Dort ist die Methode `Condition` zu definieren, die analog zu DESMO zwei Entity-Zeiger als Parameter hat. Die von `StatistikObject` geerbte Funktionalität für Warteschlangenstatistik wird nicht benutzt und liefert unbrauchbare Werte. Die Statistik der impliziten Warteschlangen ist nicht zugänglich, sondern wird lediglich im Report präsentiert.

2.5 Beurteilung

DESMO ist ein gut strukturiertes und robustes Simulationspaket, das die objektorientierten Techniken des Vorgängers DEMOS mit Hilfe von abstrakten Datentypen sehr gut nachbildet und um den ereignisorientierten Ansatz erweitert. SiFrame implementiert die DESMO-Funktionalität mit Hilfe der objektorientierten Programmiersprache C++. Dabei wirkt es oft, daß die Portierung an einigen Stellen so eng am Vorbild verläuft, daß es sich eher um eine Übertragung auf rein syntaktischer Ebene handelt. Dies betrifft insbesondere die Umsetzung des Entities im prozeßorientierten Ansatz (s. Abschnitt 2.5.1).

2.5.1 Klarheit der Begriffe

Auf DESMO aufbauende Arbeiten aber auch die Erfahrung bei der Betreuung von Übungsgruppen haben gezeigt, daß die Bedeutung der Begriffe Entity, Ereignis und Prozeß oft nicht richtig erkannt wird, und mit Formulierungen äußerst sorgsam umgegangen werden muß. So finden sich in [Weber96] z.B. folgende Formulierungen:

> "Ein Ereignis ist eine Menge von Handlungen, …"
>
> "Innerhalb eines Simulationsprogramms werden unterschiedliche Ereignistypen unterschieden, welche von Ereignisroutinen ausgeführt werden."

Eine merkwürdige Entwurfsentscheidung wird in [Trush95] getroffen, einer Umsetzung von DESMO in Oberon-2. Dort erbt die Klasse `Event` von der Klasse `Entity`. Da Vererbung eine "ist ein"-Beziehung darstellt, entsteht eine falsche Bedeutung des Begriffes Ereignis. In SiFrame taucht der Begriff des Ereignisses wiederum gar nicht auf. Hier entsprechen die Ereignistypen den von `EventRoutine` abgeleiteten Klassen der Ereignisroutinen. Jedoch widerspricht es dem Sprachgebrauch, wenn Ereignisroutinen vorgemerkt werden.

Im prozeßorientierten Ansatz übernimmt SiFrame die in DESMO unumgängliche Trennung von Prozeßattributen und Beschreibung der Aktivitäten des Prozesses in zwei Klassen `Entity` und `ProcessAction`. Das Entity selbst degeneriert also wie im ereignisorientierten Ansatz wieder zu einem passiven Objekt, das von der Prozeßroutine manipuliert wird. Innerhalb von `ProcessAction` können die Prozeßattribute nur über das Entity manipuliert werden. Die Möglichkeit, Prozesse als aktive Simulationsobjekte zu beschreiben und als solche zu modellieren wird hier nicht genutzt. `ProcessAction` ist dem Konzept der Ereignisroutine sehr ähnlich (vgl. Abbildung 2-10). Es wäre von Vorteil, wenn die Begriffe Ereignis, Entity und Prozeß eine direkte Entsprechung im Simulationspaket hätten.

In Bezug auf die warteschlangenbasierten Objekte ist die Realisierung der SiFrame-Klassenhierarchie nicht ganz schlüssig. So entstehen bei solchen Objekten jeweils zwei Report-Objekte (bzw. drei bei `WaitQ`-Objekten), jeweils eines für die implizite Warteschlange und eines für das Objekt selbst. Im Falle der impliziten Warteschlangen wird deren Reportfunktionalität jedoch gar nicht genutzt. So definieren die beiden Klassen, sowohl `qBased` als auch `StatistikObject`, die gemeinsame Schnittstelle für die Warteschlangenstatistik, wobei `qBased` die Aufrufe nur an die implizite Warteschlange weiterreicht. Im Falle von `WaitQ` wird die Funktionalität der Oberklasse sogar überhaupt nicht mit einbezogen. Die Statistikfunktionen liefern daher unbrauchbare Werte.

Was bei SiFrame die Suche in Warteschlangen bzw. darauf basierten Objekten betrifft, so werden drei verschiedene Konzept angeboten (`QueueCondition`, `CondQCondition`

und `WaitQCondition`), die sich nicht sehr von einander unterscheiden. Letztere hebt sich lediglich dadurch ab, daß hier zwei Entities mit einander verglichen werden, um ein Entity auszuwählen. Für die Suche in Warteschlangen und zur Angabe einer Suchbedingung in `CondQ`-Objekten, könnte hingegen dasselbe Konzept verwendet werden.

Die Benennung einiger Klassen ist z.T. recht verwirrend, und trägt nicht dazu bei, sich in der Klassenhierarchie zurechtzufinden. So heben sich `SimulationItem` und `SimulationObject` in ihrer Bedeutung kaum voneinander ab. `StatistikObject` ist Oberklasse für Warteschlangen, nicht jedoch für statistische Datensammelobjekte. Der Name `StandardReport` deutet eher darauf hin, daß es sich um eine Ausgabeform des Reports handelt, anstatt um die Oberklasse für `Tally`- und `Accumulate`-Objekte.

2.5.2 Trennung der Weltbilder

Sowohl in DESMO als auch in SiFrame findet eine strikte Trennung von ereignis- und prozeßorientiertem Ansatz statt. In DESMO wurde diese Abgrenzung aus Gründen der Implementierung vorgenommen. Aus der Sicht des Modellentwicklers besteht dafür jedoch keine Notwendigkeit. So ist es durchaus denkbar, in einem prozeßorientierten Modell ein Ereignis “Ausfall einer Maschine” mit Hilfsmitteln des ereignisorientierten Ansatzes zu modellieren. Kapitel 3 beleuchtet diesen Aspekt nochmals in Bezug auf die Wiederverwendbarkeit von Modellen und stellt ein Konzept vor, in dem beide Ansätze für sich, aber auch gemeinsam nutzbar sind.

2.5.3 Durchführung eines Experiments

Hat man ein Modell entwickelt, kann man an ihm Experimente durchführen. In DESMO entspricht dem Konzept des Experiments der Rumpf des Hauptprogramms. Alle Datenstrukturen zur Simulationssteuerung sind global angelegt. So besteht ein Programmlauf aus der Initialisierung dieser Datenstrukturen (z.T. implizit), dem Starten der Simulation und der Ausgabe der Ergebnisse. Ein Programmlauf entspricht damit genau einem Experiment. Das wiederholte Experimentieren mit dem selben Modell unter Variation der Parameter ist zwar nicht ausgeschlossen, wird aber nicht unterstützt. So läßt sich die Simulationsuhr nach einem Simulationslauf nicht zurückstellen. Statische Modellkomponenten lassen sich nicht löschen, um mit einem anderen Modell experimentieren zu können.

In SiFrame verhält es sich ähnlich. Innerhalb des Programms muß genau ein Objekt einer Unterklasse von `MainEventSimulation` bzw. `MainProcessSimulation` erzeugt werden. Dieses wird von der in SiFrame integrierten C-Funktion `main` “angestoßen”, wodurch die Simulation in Gang kommt. Mehrere Objekte einer dieser Klassen sind nicht zugelassen, so daß innerhalb eines Programms genau ein Experiment durchgeführt werden kann. Die Datenstrukturen zur Simulationssteuerung sind meist als Klassenvariablen der entsprechenden Klassen realisiert (z.B. der Zustand des Startwertegenerators für Zufallszahlenströme).

Bei SiFrame ist die C-Funktion `main` in die Simulationsbibliothek integriert. Dieser Umstand schließt die Verwendung von SiFrame mit anderen Bibliotheken aus, bei denen der Zugriff auf `main` erforderlich ist, oder die `main` ebenfalls integrieren. In Kapitel 3 wird ein Konzept vorgestellt, welches es ermöglicht, innerhalb eines Programms Experimente zu erzeugen und wieder zu löschen, so daß zum einen mehrere Experimente an dem selben Modell unter Variation der Ausgangsbedingungen durchgeführt werden könne, aber auch mit ganz unterschiedlichen Modellen gearbeitet werden kann.

3 Das neue Konzept: DESMO-C

In Kapitel 2 wurden die Vorgänger DESMO und SiFrame vorgestellt und Vorschläge zu deren Verbesserung gemacht. In diesem Kapitel werden die Schnittstellenkonzepte der in dieser Arbeit entwickelten Klassenbibliothek DESMO-C (Discrete Event Simulation and Modelling in C++) vorgestellt. Dabei wird auch auf die für die Gestaltung der Schnittstelle relevanten Entwurfsentscheidungen eingegangen. Besonderheiten der Implementierung sowie die internen Strukturen zur Simulationssteuerung sind Inhalt von Kapitel 4.

Bei der Entwicklung der Schnittstelle von DESMO-C war es ein Ziel, möglichst streng nach dem Top-Down-Verfahren vorzugehen und den Bottom-Up-Effekt erst sehr spät wirken zu lassen. Dadurch konnte eine gute Ausrichtung an der Terminologie des Problembereichs, der zeitdiskreten Simulation, erzielt werden.

Fast alle Klassen der Schnittstelle von DESMO-C erben von der Klasse `NamedObject` die Fähigkeit einen Namen zu tragen. Der Name kann über den Konstruktor initialisiert oder später mittels `Rename` geändert werden. Um die folgenden Ausführungen nicht unnötig zu überfrachten, sei dort auf die Erwähnung dieser Klasse verzichtet.

3.1 Die Zusammenführung der Weltbilder

In Abschnitt 2.2 wurden die Weltbilder der zeitdiskreten Simulation eingeführt. Die strikte Trennung zwischen ereignis- und prozeßorientiertem Weltbild war in DESMO erforderlich, da kein Zusammenhang zwischen den Entity-Konzepten des ereignis- bzw. prozeßorientierten Ansatzes hergestellt wurde. Eine gemeinsame Betrachtung beider Konzepte wurde durch die Typstrenge der zugrundeliegenden Programmiersprache Modula-2 und fehlenden Mitteln zur Umsetzung von Vererbung und Polymorphie erschwert. Auf die Nützlichkeit zur Formulierung externer Ereignisse auch in prozeßorientierten Modellen wird in [Bölck89] nur hingewiesen, ohne jedoch eine Lösung zur Integration anzubieten.[24]

3.1.1 Warum die Kombinierbarkeit sinnvoll ist

Für die Modellierung würde die Nutzbarkeit beider Ansätze die adäquate Abbildung der Gegebenheiten des Realsystems auf die Konstrukte des Modells ermöglichen. In der Regel lassen sich die Objekte des Realsystems relativ leicht als aktive Objekte im Modell darstellen. Diese beeinflussen mit ihren zeitverbrauchenden Handlungen den Modellzustand und interagieren mit anderen Objekten des Modells. Hier ist der prozeßorientierte Ansatz das Werkzeug der Wahl. Die Aktivitäten der Objekte können als Prozesse modelliert werden. Darüber hinaus hat man es in Realsystemen häufig auch mit einmaligen Situationen zu tun, die sich “ereignen”, wie z.B. in ausfallanfälligen Systemen. Der Ausfall einer Maschine läßt sich durch ein Ereignis viel passender beschreiben, als ihn mit Hilfe eines Prozesses zu modellieren.

Ein weiteres Argument für die Forderung der Koexistenz von ereignis- und prozeßorientiertem Ansatz ist die Absicht, bereits entwickelte Modelle wiederzuverwenden, um sie zu komplexeren Modellen zu kombinieren. Liegen verschiedene Teilmodelle vor, sollte ihre Verwendung nicht von der Art ihrer Realisierung abhängen. Vielmehr dürfen die Einbettungsmöglichkeiten eines Modells allein von seiner Schnittstelle abhängen. So kann für

[24] [Bölck89], S. 58 ff.

jedes Teilmodell der geeignete Modellierungsstil verwendet werden, ohne die Nutzbarkeit des Modells einzuschränken.

3.1.2 Unterschiede und Überschneidungen der Weltbilder

Im ereignisorientierten Ansatz wird ein temporäres Simulationsobjekt als Entity bezeichnet. Es repräsentiert aktive oder passive Objekte des Realsystems. In der programmtechnischen Realisierung sind jedoch die Ereignisroutinen die einzig aktiven Komponenten. In ihnen werden Entities manipuliert, um so deren Aktivitäten abzubilden. Im prozeßorientierten Ansatz wird die Rolle der Entities von Prozessen eingenommen. Sie sind Objekte mit Attributen und aktivem Verhalten. Somit können Prozesse als eine Erweiterung der Entities des ereignisorientierten Ansatzes betrachtet werden. In beiden Ansätzen können solche Simulationsobjekte in Warteschlangen eingereiht bzw. daraus entfernt werden. Sowohl Zufallszahlenströhme als auch statistische Datensammelobjekte hängen nicht vom Weltbild ab und konnten daher schon in DESMO für beide Ansätze genutzt werden.

3.1.3 Entities

Das Problem der Vereinigung beschränkt sich somit auf die Konzepte von Entities und Prozessen. Beide sind ein zentrales Element ihres jeweiligen Ansatzes. Sie repräsentieren häufig sogar dieselben Objekte des Realsystems. Nicht umsonst tragen sie in DESMO auch denselben Namen “Entity”. Aber was verbirgt sich eigentlich genau hinter diesem Begriff? Eine Definition von “Entität” in [Brock86] liefert eine gute Vorstellung der wesentlichen Merkmale:

> **Entität** [lat.], die bestimmte Seinsverfassung eines Seienden, auch dieses selbst.

In Bezug auf Simulationsobjekte sind für ein Entity also sein Zustand (“Seinsverfassung”) sowie seine Identität (“eines Seienden”) charakteristisch. Der Zustand äußert sich in den Werten seiner Attribute einerseits und in den Beziehungen zu anderen Komponenten des Modells andererseits. Die Identität gewährleistet, daß zwischen zwei Entities unterschieden werden kann.

Auch wenn Entities aktive Objekte im zu modellierenden System repräsentieren, sind sie doch, was die Programmierung betrifft, passive Komponenten. Denn die eigentlichen Handlungen geschehen in den Ereignisroutinen. Der Versuch, eine Ereignisroutine als einen Teil der Aktionen eines Entities aufzufassen, kann die ereignisorientierte Modellierung sogar erschweren. Auch wenn ein Ereignis mit einem bestimmten Entity assoziiert ist, kann das eine Manipulation anderer Entities bedeuten. Wenn z.B. das Ereignis “Bearbeitungsende von Werkstück A” behandelt wird, bedeutet dies zunächst, daß Werkstück A eine Maschine verläßt und evtl. die Bearbeitung in einer anderen Maschine beginnt. Dies bedeutet aber nicht, daß sich das Ereignis ausschließlich auf Werkstück A auswirkt. Beispielsweise könnte ein Werkstück B, das vor der Maschine gewartet hat, nun bearbeitet werden. Werkstück B müßte also aus einer Warteschlange entfernt werden und es würde ein weiteres Ereignis “Bearbeitungsende von Werkstück B” angesetzt werden.

3.1.4 Prozesse als Entities mit aktivem Verhalten

In DESMO werden Prozesse wie schon beim Vorgänger DEMOS ebenfalls als “Entity” bezeichnet. Birtwistle stellt in [Birtw79] zur Einführung des Begriffes “Entity” eine Analogie zu den Rollen eines Theaterstücks her, in die sich die Schauspieler hineinversetzen. Er unterteilt jede Rolle in aktive Phasen, in denen gesprochen wird, und passive Phasen, in

denen der jeweilige Darsteller darauf wartet, daß er wieder an der Reihe ist. Weiter beschreibt er:

> Once we have sorted out how each object synchronises its actions with other objects (a major modelling problem), the description of the rest of an object's life history can be completed separately. We simply psyche ourselves into each role in turn and then write out its actions from its own viewpoint.[25]

Dies beschreibt genau das, was der Modellentwickler tut, wenn er die aktiven Objekte des Realsystems im prozeßorientierten Ansatz modelliert. Er versetzt sich in die Lage der Objekte und beschreibt ihre Aktivitäten aus der Sicht der Objekte selbst. Dies umfaßt zum einen die aktiven und passiven Phasen und zum anderen die Synchronisierung mit anderen Objekten.

Die Analogie zu einer Rolle, in die sich ein Schauspieler hineinversetzt, erlaubt einen direkten Vergleich mit dem Entity-Begriff des ereignisorientierten Ansatzes. Auch bei Birtwistle finden sich Identität (zwei Rollen (Schauspieler) sind unterscheidbar, sie können sich z.B. nicht an der selben Stelle auf der Bühne befinden) und Attribute (z.B. Kostüm und Geschlecht der Rolle) wieder. Somit lassen sich Entities des prozeßorientierten Ansatzes, im folgenden Prozesse genannt, als Erweiterung der Entities des ereignisorientierten Ansatzes auffassen. M.a.W. Prozesse sind Entities. Dies stellt eine "ist ein"-Beziehung dar, was den Einsatz der Vererbung nahelegt.

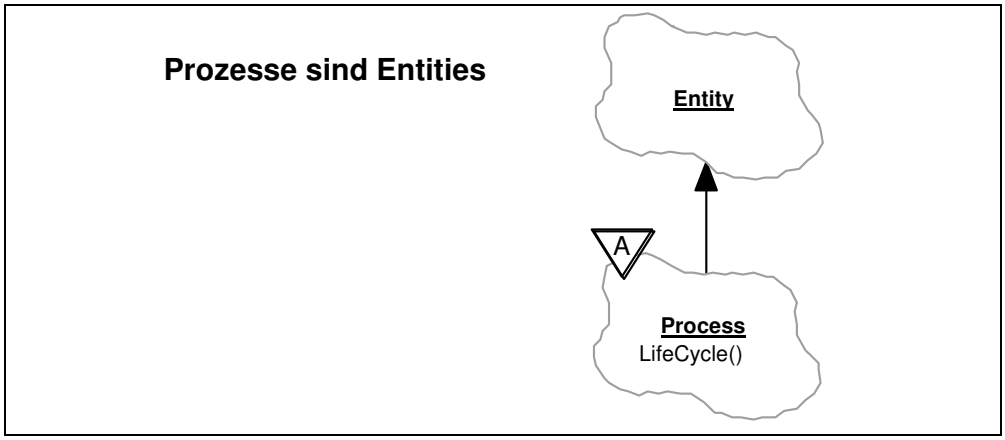

Abbildung 3-1: Prozesse als Entities mit aktivem Verhalten

Die Erweiterung besteht in der Hinzunahme der Aktivitäten. Entities können zwar sehr wohl Verhalten aufweisen, jedoch ist dies rein passiv. Sie sind in der Lage zu reagieren, aber sie wirken nicht aus "eigener Kraft" auf sich selbst oder ihre Umgebung ein. Prozesse hingegen nutzen ihre aktiven Phasen, um den Modellzustand zu verändern. Sie folgen während ihrer gesamten Lebensdauer der Beschreibung ihres Lebenszyklus. Aus dieser Betrachtung heraus wäre es sinnvoll Entities als passive Entities[26] und Prozesse als aktive Entities zu bezeichnen. Die Begriffe "aktiv" und "passiv" können jedoch andere Bedeutung haben, je nachdem ob man sie im Modell oder im Programm verwendet. Als Beispiel soll wieder die Maschine dienen, vor der Werkstücke auf ihre Bearbeitung warten. Die Maschine bearbeite gerade ein Werkstück. Im Modell ist sie aktiv. Im Programm dagegen muß dies nicht so sein. Kommt z.B. zwischen Bearbeitungsanfang und -ende ein neues Werk-

[25] [Birtw79], Seite 25

[26] In [Booch95] findet sich im Glossar auf S. 614 als Beschreibung für "passives Objekt": "Ein Objekt, das keinen eigenen Steuerfluß (thread) besitzt.

stück an, so ist dies der aktive Prozeß, der die (Programm-) Kontrolle hat. Im Modell sind beide aktiv. Die Bedeutung im Programmkontext weicht hier also von der im Modell ab. Aus diesem Grunde sollte mit den Begriffen eher sparsam umgegangen werden, da sonst sehr leicht Mißverständnisse entstehen können.

Für die ereignisorientierte Modellierung steht in DESMO-C eine Klasse `Entity` zur Verfügung, von der durch Bildung von Unterklassen neue Entitytypen abgeleitet werden können. Den Prozeßtypen eines prozeßorientierten Modells hingegen dient die Klasse `Process` als Oberklasse. Deren Methode `LifeCycle` muß implementiert werden, um die Aktivitäten des Prozesses zu beschreiben. Anders als in SiFrame, wo die Handlungen als ein separates Objekt der Klasse `ProcessAction` explizit mit dem zugehörigen Prozeß verbunden werden mußten, besteht diese Verbindung in DESMO-C implizit dadurch, daß die Handlungsbeschreibung eine Methode des Prozesses ist. Außerdem kann innerhalb der Methode `LifeCycle` direkt auf die Prozeßattribute zugegriffen werden.

3.1.5 Ereignisse

Die Erfahrung mit der Betreuung von Studierenden bei der Arbeit mit DESMO hat gezeigt, daß die Begriffe Ereignis und Entity leicht durcheinandergebracht werden und deren ganaue Rollen, die sie im ereignisorientierten Ansatz einehemen, schwer zu fassen sind. In DESMO tauchen Ereignisse als solche letztlich auch gar nicht auf. Hingegen sollen sie in DESMO-C als Objekte existieren. Sie müssen erzeugt und gelöscht werden wie andere Objekte auch, und sollen dadurch greifbarer werden. Folglich bietet DESMO-C eine Klasse `Event` an, von der Unterklassen zur Modellierung neuer Ereignistypen abgeleitet werden können. Ein Ereignistyp entspricht also genau einer (direkten oder indirekten) Unterklasse von `Event`.

Eine andere Frage ist, wie Ereignisroutinen repräsentiert und mit Ereignistypen in Verbindung gebracht werden sollen. Ein Modell reagiert auf ein Ereignis eines bestimmten Typs mit dem Aufruf der dem Ereignistyp entsprechenden Ereignisroutine. Zwischen Ereignistyp und Ereignisroutine besteht also eine 1:1-Beziehung. Wird die Ereignisroutine als Methode der Ereignisklasse realisiert, wird diese Beziehung ohne zusätzlichen Verknüpfungsaufwand auf natürliche Weise hergestellt. In DESMO-C heißt diese Methode `EventRoutine`, so daß sowohl der Begriff des Ereignisses als auch der der Ereignisroutine konkret in Erscheinung treten. Die Methode `EventRoutine` besitzt einen Parameter, in dem das Entity übergeben wird, für das das Ereignis eingetreten ist.

Die Ereignisroutine ist die aktive Programmkomponente. Diesem Umstand steht in DESMO die Sicht entgegen, daß das assoziierte Entity als 'aktiv' betrachtet wird. Im Trace äußert sich das durch entsprechende Meldungen. Wird das Entity 'Job 1' z.B. bei seiner Ankunft (Ereignis 'Arrival') in die Warteschlange 'JobQueue' eingereiht, lautet die entsprechende Zeile im Trace:

```
Arrival     Job 1       inserts itself into 'JobQueue'
```

Diese Sicht erschwert das Verständnis dafür, daß in der Ereignisroutine nicht nur das übergebene Entity manipuliert wird, sondern auch andere beliebige Zustandsänderungen vorgenommen werden können. Ausgehend von der Vorstellung, daß die Ereignisroutine eigentlich die aktive Komponente ist, die das Entity manipuliert, sollte die Trace-Ausgabe folgendermaßen lauten:

```
Arrival     Job 1       inserts it into 'JobQueue'
```

Denn das Eintreten des Ereignisses “Ankunft von Job 1” bewirkt, daß ‘Job 1’ in die Warteschlange eingereiht wird. Dem wird in DESMO-C Rechnung getragen.

Eine Unterklasse von `Event` ist die Klasse `ExternalEvent`. Externe Ereignisse unterscheiden sich von herkömmlichen Ereignissen dadurch, daß sie nicht mit einem ganz bestimmten Entity assoziiert werden, sondern eher global auf den Modellzustand wirken. `ExternalEvent` weist folglich dieselben Methoden wie `Event` auf, jedoch fehlen jeweils die `Entity`-Parameter. Im Gegensatz zur Oberklasse `Event`, muß hier die parameterlose Methode `ExternalEventRoutine` definiert werden.

3.1.6 Vormerken von Objekten

Das in DESMO-C verfolgte Prinzip zum Vormerken von Objekten (Ereignissen, Entities oder Prozessen) besteht darin, daß man die Objekte selbst dazu auffordert, sich auf die Ereignisliste zu setzen. D.h. die Klassen `Event`, `Entity` und `Process` bieten Methoden an, mit deren Hilfe sich die Objekte selbst vormerken. So versteht ein Entity z.B. die Nachricht `Schedule`, der zwei Parameter übergeben werden: das Ereignis, das mit diesem Entity vorgemerkt werden soll, und der Zeitpunkt, an dem das Ereignis eintreten soll. Äquivalent dazu könnte man das Ereignis mit Hilfe seiner `Schedule`-Methode vormerken. Als Parameter werden hier entsprechend das Entity und der Zeitpunkt angegeben. Beide Aufrufe haben das selbe Resultat, nämlich daß Ereignis und Entity zusammen auf die Ereignisliste gesetzt werden. Ein Beispiel soll dies verdeutlichen:

```
 1. class Arrival : public Event  {...};
 2. class Job     : public Entity {   };
 3.
 4. Arrival* arrival  = new Arrival (...);
 5. Job*     job      = new Job     (   );
 6.
 7. // folgende Aufrufe können alternativ verwendet werden:
 8. arrival->Schedule (1.0, job);
 9. job    ->Schedule (1.0, arrival);
10.
```

Listing 3-1: Ansetzen eines Ereignisses

In Listing 3-1 definiert Zeile 1 einen neuen Ereignistyp “Arrival”, Zeile 2 einen neuen Entitytyp “Job”. In Zeile 4 wird nun ein konkretes Ankunftsereignis erzeugt. Zeile 5 gibt die Erzeugung eines neuen Auftrags wieder. Um nun die “Ankunft des Auftrags” vorzumerken gibt es zwei Möglichkeiten:

1. Zeile 8 teilt dem Ereignis `arrival` mit, “es möge sich zum Zeitpunkt 1.0 zusammen mit dem Entity `job` vormerken”.

2. Zeile 9 teilt dem Entity `job` mit, “es möge sich zum Zeitpunkt 1.0 zusammen mit dem Ereignis `arrival` vormerken”.

Beide Aufrufe führen dazu, daß in einer Zeiteinheit das Ereignis “Ankunft des Auftrags” (`arrival`, `job`) eintritt. Neben der einfachen `Schedule`-Methode, stehen sowohl bei Entities als auch bei Ereignissen die Methoden `ScheduleBefore` und `ScheduleAfter` zum gezielten Vormerken vor bzw. hinter einem anderen bereits vorgemerkten Entity oder Ereignis zur Verfügung.

Zum Vormerken von Prozessen gibt es in DESMO-C mehr Möglichkeiten, als in DESMO. Dadurch, daß `Process` von `Entity` erbt, ist es möglich, einen Prozeß als Entity zu behandeln. Ein Prozeß läßt sich auf diese Weise mit einem Ereignis vormerken. Dies bewirkt, daß zum entsprechenden Zeitpunkt die Ereignisroutine aufgerufen wird und einen Verweis auf den Prozeß als Parameter übergeben bekommt. Der Prozeß wird dadurch nicht aktiviert. Vielmehr wird er in der Ereignisroutine manipuliert, er nimmt somit eine passive Rolle ein. Zum Aktivieren eines Prozesses können daher die `Schedule`-Methoden nicht verwendet werden. Zu diesem Zweck bietet die Klasse `Process` die Methoden `Activate`, `ActivateBefore` und `ActivateAfter` an, die die Aktivierung eines Prozesses zu einem gegebenen Zeitpunkt bzw. vor oder nach der Aktivierung eines anderen Prozesses vormerken.

Für alle drei Klassen `Event`, `Entity` und `Process` sind verschiedene Methoden verfügbar, die von der Oberklasse `Schedulable` bereitgestellt werden, da sie unabhängig von der jeweiligen Unterklasse sind. Abbildung 3-2 gibt einen Überblick über die wichtigsten Methoden.

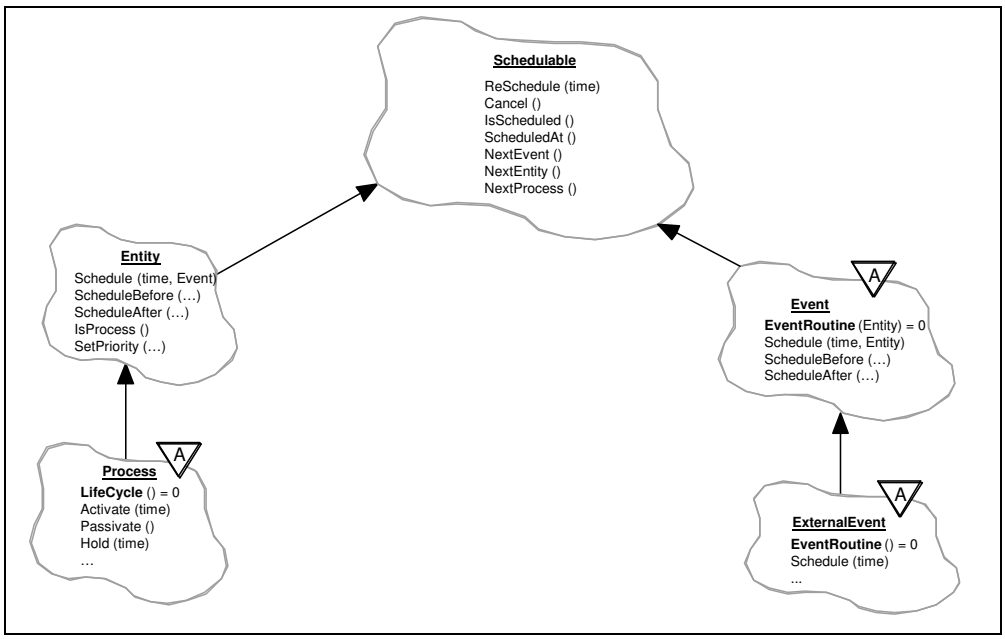

Abbildung 3-2: Vormerkbare Objekte (`Schedulable`)

`ReSchedule` und `Cancel` entsprechen den gleichnamigen Prozeduren aus DESMO. `ScheduledAt` entspricht `EvTime` aus DESMO, und liefert den Zeitpunkt des Ereignislisteneintrages. `IsScheduled` liefert `true`, wenn das Objekt vorgemerkt ist, und entspricht der Negation der Prozedur `Idle`. Dabei ist zu beachten, daß das aktuelle Objekt nicht mehr auf der Ereignisliste steht. Im Gegensatz zu DESMO gilt dies für beide Ansätze gleichermaßen. Eine detaillierte Beschreibung der übrigen Methoden findet sich in Kapitel 6.

3.1.7 Konsequenzen

Um in DESMO-C im ereignisorientierten Stil zu modellieren, werden die Klassen `Event` und `Entity` benutzt. Im Gegensatz dazu findet die Klasse `Process` im prozeßorientierten Ansatz Verwendung. Es können so rein ereignisorientierte bzw. rein prozeßorientierte Modelle entwickelt werden. Eine strikte Trennung wird von der Klassenbibliothek jedoch

nicht erzwungen. So ist es z.B. möglich, den Ausfall einer als Prozeß realisierten Maschine mit Hilfe eines Ereignisses zu simulieren.

Die Tatsache, daß ereignis- und prozeßorientierte Konstrukte innerhalb eines Programms nebeneinander existieren können, hat jedoch weitreichende Konsequenzen auf die Fähigkeit, wiederverwendbare Modelle mit DESMO-C zu entwickeln. Ein Modell kann nun unabhängig vom verwendeten Stil in einen komplexeren Zusammenhang eingebettet werden. So können für die Implementierung eines Modells die Konstrukte ausgewählt werden, die für die Modellierung angemessen erscheinen. Der einbettende Kontext bleibt von dieser Entscheidung unberührt.

Besondere Beachtung in diesem Zusammenhang erfordert die Behandlung der Pseudo-Simulationszeit `NOW`, die in beiden Ansätzen unterschiedlich behandelt wird. Während eine Vormerkung mit dieser Konstanten als Parameter im prozeßorientierten Teil zu einer Verdrängung des aktuellen Prozesses führt, darf dies im ereignisorientierten Ansatz nicht geschehen. Eine Ereignisroutine muß stets bis zu ihrem Ende ausgeführt werden. Um dies zu gewährleisten, wirkt `NOW` in Abhängigkeit davon, ob gerade ein Ereignis behandelt wird oder ein Prozeß aktiv ist. Nur im letzteren Falle findet eine Verdrängung statt. Im Trace kann die Verwendung von `NOW` eindeutig über die neue Textkonstante `'NOW'` gegenüber von `'now'` für 0.0 identifiziert werden.

Für die Ablaufverfolgung gibt es in DESMO zwei unterschiedliche Trace-Formate. Dabei stellt die ereignisorientierte Version eine um eine Ereignisspalte erweiterte Variante der prozeßorientierten Version dar. In DESMO-C existiert hingegen immer auch eine Spalte zur Angabe von Ereignissen, die jedoch leer verbleibt, wenn das in der entsprechenden Zeile behandelte Objekt ein Prozeß ist. Bei der ersten Trace-Ausgabe des ersten Prozesses steht in der Ereignisspalte “---”, um anzudeuten, daß es kein Ereignis gibt.

3.2 Modell und Experiment - das Baukastenprinzip

In einem DESMO-Modell werden alle statischen Modellkomponenten meist als globale Variablen angelegt. Das Paket wird einmal initialisiert, die Simulation gestartet und schließlich ein Report erzeugt. Soll ein weiteres Experiment mit dem gleichen Modell erfolgen, muß das Programm beendet werden, da sich der Simulationskern nicht zurücksetzen läßt. Dadurch wird die Durchführung einer ganzen Experimentserie zu einer Aufgabe, die allein mit DESMO nicht mehr vernünftig zu handhaben ist. Wenn Benutzerinteraktionen zwischen den Programmläufen vermieden werden sollen, muß auf einen externen Mechanismus, z.B. ein Shell-Script unter UNIX, zurückgegriffen werden, um das jeweilige Experiment mit den entsprechenden Parametern zu versorgen. Diese Beschränkung wurde auch in SiFrame zugunsten einer leichteren Implementierung aufrecht erhalten.

Ein Ziel der vorliegenden Arbeit ist es, einen Mechanismus zu entwickeln, der es erlaubt, beliebig viele Experimente durchzuführen, ohne dafür das Programm beenden zu müssen. Nicht zuletzt hierdurch wurde eine vollständige Neuimplementierung des Simulationskerns erforderlich.

Das Prinzip besteht nun darin, daß der Modellentwickler sein Modell entwickelt, unabhängig davon wie und wann er damit experimentiert. Alle zum Modell gehörenden statischen Komponenten sind im Modell gekapselt und werden mit ihm erzeugt. Um mit seinem Modell nun ein Experiment durchzuführen, legt der Modellentwickler ein neues Experiment an, das das Modell initialisiert und mit Parametern versorgt. Er läßt das Experiment über

einen vorgegebenen Zeitraum laufen oder wählt ein anderes Abbruchkriterium. Nach Ausgabe eines Reports ist das Experiment beendet. Es kann nun gelöscht werden und mit ihm alle zur Simulationssteuerung erforderlichen Strukturen. Diese Schritte zur Erzeugung, Durchführung und Vernichtung eines Experiments können nun beliebig oft wiederholt werden, ohne das Programm zwischendurch beenden zu müssen.

3.2.1 Modellkomponenten

Ein Modell besteht aus Komponenten, den Modellkomponenten (Klasse `ModelComponent`). Um in DESMO-C diese Beziehung auszudrücken, wird eine Komponente bei ihrer Erzeugung mit dem Modell verbunden, zu dem sie gehören soll. Abbildung 3-3 zeigt die wichtigsten Unterklassen von `ModelComponent`.

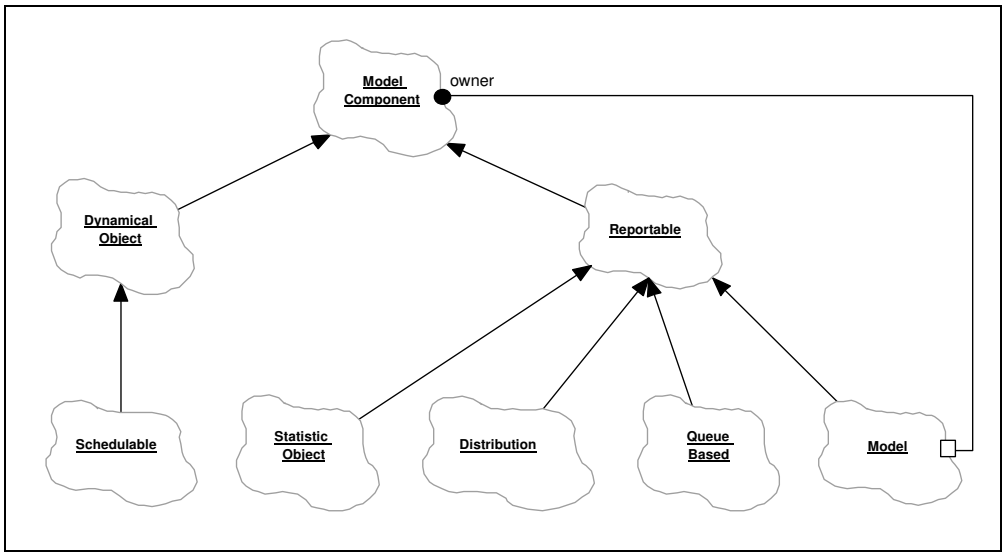

Abbildung 3-3: Klassenhierarchie der wichtigsten Modellkomponenten

Auf der einen Seite gibt es dynamische Objekte, die in der Regel erst während des Simulationslaufes erzeugt und noch vor dem Ende des Experiments gelöscht werden können. Die wichtigsten Unterklassen in diesem Zweig sind die Klassen der vormerkbaren Objekte (`Schedulable`), zu denen `Event`, `Entity` und `Process` gehören. Auf der anderen Seite befindet sich der große Teilbaum der reportfähigen Objekte (`Reportable`). Diese Objekte verfügen über einen Mechanismus, der es erlaubt, Informationen über sie im Report auszugeben.

Die Klasse `ModelComponent` hat die Aufgabe, die Verbindung zu genau einem Modell herzustellen. Diese Verbindung wird bei der Erzeugung der Komponente eingerichtet und ist für seine gesamte Lebensdauer unveränderlich. Die Lebensdauer eines Modells ist eng verknüpft mit der seiner Komponenten. Das Modell stellt daher den idealen Aufbewahrungsort für sie dar. Komponenten können als Attribute (oder Elemente) des Modells implementiert werden.

Über die Verbindung zum Modell können Objekte der Klasse `ModelComponent` Informationen in Bezug auf das laufende Experiment zugänglich machen. Hierfür bietet sie die folgenden Methoden an:

- `CurrentTime`, `NOW` und `Epsilon` zum Zugriff auf die aktuelle Simulationszeit, die Pseudo-Simulationszeit für Verdrängung bzw. die kleinste Zeiteinheit, die beim Stellen der Simulationsuhr berücksichtigt wird

- `CurrentEvent`, `CurrentEntity` und `CurrentProcess` zur Ermittlung der gerade aktuellen Objekte

- `TraceOn`, `TraceOff`, `DebugOn` und `DebugOff` um Trace- bzw. Debug-Modus ein- und ausschalten zu können

- `Warning`, `Error`, `FatalError`, `TraceNote` und `SendMessage` zur Signalisierung eines Fehlers, ausgeben einer Notiz in die Trace-Ausgabe bzw. versenden einer beliebigen Nachricht, die von der Simulationssteuerung an entsprechende Kanäle weitergeleitet wird

3.2.2 Modellhierarchien

Blickt man Abbildung 3-3 zurück, so wird deutlich, daß Modelle selbst Modellkomponenten sind. Andererseits können Modelle Modellkomponenten enthalten. Dieser Zusammenhang ist dem Kompositionsmuster[27] nachempfunden. Es ermöglicht, ein Modell als Komponente eines anderen Modells zu betrachten. Auf diese Weise läßt sich eine ganze Hierarchie von Modellen erzeugen. Modelle können so verwendet werden, um als Untermodelle Teile eines komplexeren Modells darzustellen.

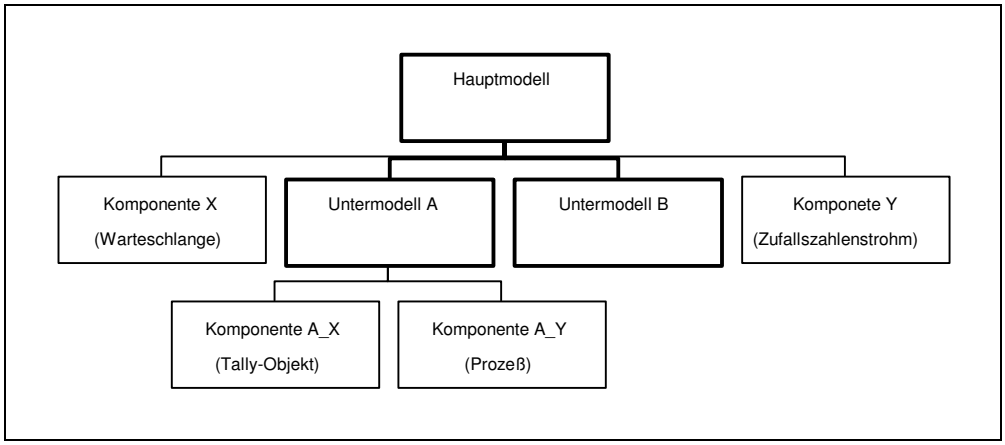

Abbildung 3-4: Beispiel einer Modellhierarchie

In einer Modellhierarchie gibt es genau ein Modell, das nicht Komponente eines anderen Modells ist. Es bildet somit die Wurzel des Modellbaumes und ist die einzige Modellkomponente, der bei ihrer Erzeugung kein Modell zugeordnet werden muß. Dieses Modell wird im Folgenden das Hauptmodell genannt. Als Komponenten verwendete Modelle werden dagegen als Sub- oder Untermodelle bezeichnet. Abbildung 3-4 zeigt eine Modellhierarchie aus einem Hauptmodell mit vier Komponenten. Zwei davon sind ihrerseits wieder Modelle.

[27] Vgl. [Gamma95], S. 213. Im englischen wird dieses Muster auch “Composite” genannt.

Jedes Modell muß eine Schnittstelle anbieten, über die es initialisiert werden kann. Sie ist vom Modellentwickler zu gestalten. Die Methode `DoInitialSchedules` kann überschrieben werden, um erste Ereignisse oder erste Prozesse vorzumerken. Sie wird von der Simulationssteuerung beim Start des Experiments für jedes Modell automatisch aufgerufen.

3.2.3 Modellintegrität

Die Möglichkeit, mehrere Modelle kombinieren zu können, bringt das Problem mit sich, daß ein Objekt, dessen Klasse in einem Modell bekannt ist, in ein anderes Modell gelangen kann, in dem keine Kenntnis über die Klasse des Objekts besteht. Denn um innerhalb eines Modells auf spezielle Eigenschaften z.B. eines aus einer Warteschlange entnommenen Entities zugegreifen zu können, muß sein Typ von Entity in der Klassenhierarchie abwärts in seinen speziellen Modell bekannten Typ, z.B. Job, konvertiert werden.[28] Dafür muß jedoch gewährleistet sein, daß das Entity nicht aus einem anderen Modell stammt, und somit die Umwandlung in einen falschen Typ unvorhersehbare Folgen hätte.

In DESMO-C wird daher in den Fällen, wo Modellkomponenten miteinander kombiniert werden, eine Prüfung auf "Modellkompatibilität" durch geführt. Als Vorgabe ist dabei die Prüfung auf die Identität der zugehörigen Modelle eingerichtet, und in der Methode `IsModelCompatible` der Klasse `ModelComponent` umgesetzt. Z.B. wird der Versuch, ein fremdes Entity in eine Warteschlange einzureihen, dadurch mit einer Warnung quittiert und ansonsten ignoriert. So kann es später auch nicht entnommen werden im Glauben, ein "kompatibles" Entity zu erhalten, dessen spezieller Typ bekannt ist.

Wäre diese Prüfung unveränderbar, würde dies eine starke Beschränkung für die Verwendbarkeit von Modellen bedeuten. Modelle wären dann in einigen Fällen, insbes. im Bezug auf dynamische Objekte und Warteschlangen, nicht in der Lage, Komponenten gegenseitig zu nutzen. Um Abhilfe zu schaffen, bietet die Klasse `Model` eine virtuelle Methode `CheckCompatibility` an, die den Kompatibilitätstest durchführt. Ihr werden zwei Zeiger auf Modellkomponenten übergeben, wovon der erste stets auf die eigene Komponente zeigt. Die zweite Komponente ist die evtl. fremde. Die Modellkomponentenmethode `IsModelCompatible` delegiert die Prüfung nun einfach an die Methode `CheckCompatibility` ihres Modells. Diese kann in der speziellen Modellklasse definiert werden, um den Austausch von Komponenten zwischen zwei Modellen zuzulassen. Dabei können die in Abschnitt 4.1.2 vorgeschlagenen Techniken zum Einsatz kommen.

3.2.4 Experimente

Um mit einem Modell ein Experiment durchzuführen, muß ein Objekt der Klasse `Experiment` erzeugt werden. Damit werden alle für die Simulationssteuerung erforderlichen Datenstrukturen angelegt. Dem Konstruktor kann ein Objekt der Klasse `ExperimentOpts` übergeben werden. `ExperimentOpts` faßt verschiedene Werte in einer Datenstruktur zusammen. Dies sind Anzahl der Stellen und die Genauigkeit für die Ausgabe von Simulationszeitpunkten, Anzahl der Stellen für die Ausgabe von Objektnamen und für die Zählung von vormerkbaren Objekten (`Event`, `Entity` und `Process`) sowie die Konstante `Epsilon`, die angibt, ab welcher Zeitdifferenz die Simulationsuhr von einem Zeitpunkt auf den nächsten gestellt werden soll. Diese Werte können in DESMO-C für jedes Experiment individuell eingestellt werden.

---

[28] Dieses Problem wird im Hinblick auf die Sprache C++ in Abschnitt 4.1.2 eingehend diskutiert.

Um ein Experiment mit einem Modell zu verbinden, bestehen zwei Möglichkeiten. Die erste basiert darauf, das Modell als Element eines Experiments zu betrachten. In diesem Fall wird eine Unterklasse von der Klasse `Experiment` gebildet, die das entsprechende Modell als Element physikalisch enthält. Für die zweite Möglichkeit genügt es, direkt ein Objekt der Klasse `Experiment` zu erzeugen, ohne daß erst eine neue Experimentklasse gebildet werden muß. Ein Problem hierbei ist die Tatsache, daß die meisten Modellkomponenten für ihre Erzeugung Zugriff auf Datenstrukturen der Simulationssteuerung benötigen. Damit scheidet die Möglichkeit aus, das Modell dem Experiment-Konstruktor zu übergeben, denn dann müßte das Modell vor dem Experiment existieren.

Die andere Alternative ist, dem Modell-Konstruktor das Experiment zu übergeben. Sie wurde jedoch verworfen, da sonst speziell zur Einrichtung des Hauptmodells entweder ein zweiter Modell-Konstruktor mit einem Experiment-Parameter existieren muß, der vom Modellentwickler in seinem Modell bedient werden muß, oder der Modell-Konstruktor stets mit einem Experiment-Parameter überfrachtet ist. Für DESMO-C wurde daher ein Mechanismus gewählt, der ein Hauptmodell mit dem aktuellen Experiment verbindet, z.B. einem gerade neu angelegten. Die Mißachtung dieses Schemas wird von DESMO-C abgefangen. Dabei können folgende Situationen auftreten:

- das Modell wird vor dem Experiment erzeugt
- es wird ein zweites Hauptmodell erzeugt

Beide Situationen werden mit einer Fehlermeldung quittiert und führen unmittelbar zum Programmabbruch. Ebenso gibt es eine Warnung, wenn versucht wird, ein Experiment zu starten, das noch nicht mit einem Modell verbunden ist.

An der Schnittstelle bietet die Klasse `Experiment` Methoden zum Starten, Unterbrechen und Fortfahren des Experiments sowie zur Manipulation der Ausgabe eines Simulationslaufes an. Der Methode `Start` kann eine Simulationszeit übergeben werden, die angibt, bei welcher Zeit die Simulation beginnen soll. Die Simulationsuhr wird auf diese Zeit gestellt, bevor die Modelle Gelegenheit bekommen, ihre ersten Objekte vorzumerken. Mit `Stop` kann ein laufendes Experiment angehalten werden. Eine optional zu übergebende Simulationszeit hält das Experiment erst nach dieser Zeit an. Mit der Pseudo-Simulationszeit `NOW` läßt sich Experiment sogar unmittelbar anhalten, wenn gerade ein Prozeß aktiv ist. Eine in Ausführung befindliche Ereignisroutine hingegen wird in jedem Fall erst vollständig abgearbeitet. Die Programmkontrolle kehrt an die Stelle zurück, an der `Start` (bzw. `Continue`) aufgerufen wurde. Ein angehaltenes Experiment kann mittels `Continue` fortgesetzt werden.

Bei der Erzeugung muß dem Experiment ein Name übergeben werden. Dieser Name wird für die Ausgabedateien verwendet und um eine entsprechende Endung ergänzt. Experimente können nachträglich nicht umbenannt werden, d.h. für ein Experiment gibt es genau einen Satz an Standard-Ausgabedateien für Report, Trace, Debug- und Fehlermeldungen.

Ein Experiment enthält vier Ausgabekanäle für Report, Trace, Debug- und Fehlermeldungen. An jeden dieser Kanäle ist eine Standardausgabe angeschlossen, um die Meldungen entsprechend formatiert in eine Datei auszugeben. Die Trace-Ausgabe ist zusätzlich an den Fehlerkanal angeschlossen, damit Warnungen und Fehlermeldungen auch im Trace erscheinen und so das Aufspüren der Fehlerursache erleichtert wird. In der Debug-Ausgabe erscheinen zusätzlich zu den eigentlichen Debug-Informationen Trace- und Fehlermeldungen. Darüber hinaus lassen sich über die Methoden `AddDebugOutput`, `AddErrorOut-`

put, AddReportOutput und AddTraceOutput eigene Ausgaben an die entsprechenden Kanäle hängen, um die Aufbereitung der Informationen in einer anderen Form zu ermöglichen.

Desweiteren verfügt DESMO-C über drei stream-Objekte für die Bildschirmein-, -aus- und -fehlerausgabe. Als Vorgabe sind hier die in C++ gängigen Objekte cin, cout und cerr eingestellt. Diese werden von den Methoden In, Out und Err geliefert. Wird die Ein- bzw. Ausgabe über eigene stream-Objekte gewünscht, so können diese mittels SetDesmoIn, SetDesmoOut und SetDesmoErr bestimmt werden. Mittels ResetDesmoIO kann wieder auf die Vorgabe zurückgeschaltet werden. Ist in einem Modell eine Bildschirmein- bzw. ausgabe erforderlich, so sollte dies stets über die Methoden In bzw. Out erfolgen anstatt über die Objekte cin bzw. cout. Die Methoden zur Bildschirmein- und ausgabe wirken für die gesamte Klassenbibliothek. Sie sind als Klassenmethoden deklariert und können daher auch genutzt werden, ohne daß ein Experiment existiert.

3.2.5 Experimentreihen innerhalb eines Programms

Experimente können nach belieben erzeugt und wieder gelöscht werden. Dadurch besteht die Möglichkeit, innerhalb eines Programms eine ganze Reihe von Experimenten durchzuführen. Z.B. könnten n verschiedene Experimente mit demselben Modell, jedoch mit unterschiedlichen Startwerten für den Zufallszahlengenerator gestartet werden, um n unterschiedliche Stichproben zu gewinnen. Eine weitere Anwendungsmöglichkeit stellen Aufgabenstellungen dar, bei denen ein Modellparameter von einem Experiment zum anderen verändert werden soll, bis eine statistische Kenngröße des Modells einen bestimmten Wert über- oder unterschreitet.

DESMO-C ist so konzipiert, daß es möglich ist, mehrere Experimente quasi parallel durchzuführen. Obwohl in der vorliegenden Version von DESMO-C an der Schnittstelle noch keine Methoden für die parallele Durchführung mehrerer Experimente existieren, ist dies trotzdem schon realisierbar. Hierfür müssen zunächst mehrere Experimente mit ihren Modellen erzeugt werden. Danach kann man eine Schleife bilden, in der die Experimente jeweils nur für eine kurze Zeit laufen. Dadurch ist es möglich, mehrere unterschiedliche Experimente während ihrer Durchführung vergleichen zu können, und jene zu beenden, deren Entwicklung für die betrachtete Fragestellung nicht relevant erscheinen. Ein Beispiel soll dies verdeutlichen:

```
 1. Experiment *e1, *e2, *e3;
 2. ...    // Erzeugung der Experimente und Modelle
 3. e1->Stop (1);
 4. e1->Start();
 5. ...    // dto. fuer e2 und e3
 6. for (int i = 1; i < 100; ++i)
 7. {
 8.     e1->Stop (1);
 9.     e1->Continue();
10.     // dto. fuer e2 und e3
11. }
12. // alle drei Experimente enden hier zur Zeit t = 100
```

Listing 3-2: Parallele Durchführung mehrere Experimente

In Listing 3-2 wird jedes Experiment, bevor es mittels Start bzw. Continue gestartet bzw. fortgesetzt wird (Zeile 4 bzw. 9), mittels Stop angewiesen, nach einer Zeiteinheit wieder anzuhalten (Zeile 3 und 8). Dadurch werden nacheinander alle drei Experimente

jeweils für eine Zeiteinheit durchgeführt. Dies wird wiederum in der Schleife (Zeile 6 bis 11) 99 mal wiederholt.

3.2.6 Einbettung in andere Systeme

Das Koroutinen-Konzept in [Weber96] führte dazu, daß die C-Funktion `main` in SiFrame integriert werden mußte, um dort einige Initialisierungen vornehmen zu können. Unter dieser Voraussetzung wäre die Durchführung mehrerer Experimente nicht möglich gewesen. Außerdem ist die Verwendbarkeit der Bibliothek dadurch stark eingeschränkt, da sich erhebliche Schwierigkeiten in Kombination mit anderen Klassenbibliotheken oder Frameworks ergeben können.

Für DESMO-C wurde das Koroutinen-Konzept daher überarbeitet, so daß eine explizite Initialisierung nicht mehr notwendig ist. DESMO-C läßt sich folglich in jedes Framework einbetten. Z.B. ließe sich eine grafische Oberfläche entwickeln, mit der ein Anwender sich ein Modell mit Hilfe von grafischen Symbolen zusammenbaut. Beim Design eines solchen Editors könnte sich der Entwickler ganz nach den Richtlinien des verwendeten Application-Frameworks[29] richten, da von DESMO-C keine Einschränkungen auferlegt werden.

3.3 Warteschlangenbasierte Objekte

Wie in DESMO können auch in DESMO-C Entities in Warteschlangen warten. Die Klasse `Queue` bietet Methoden zum Einreihen bzw. Entfernen von Entities in bzw. aus Warteschlangen. Die in Abschnitt 3.4 näher beschriebenen Klassen `Bin`, `CondQueue`, `Res` und `WaitQueue` zur Modellierung höherer Synchronisationsmechanismen basieren auf impliziten Warteschlangen. In Objekten dieser Klassen werden Prozesse blockiert und für die Dauer der Blockierung in einer impliziten Warteschlangen verwaltet.

3.3.1 Implizite Warteschlangen

Ein erster intuitiver Ansatz zur Realisierung der Synchronisationsklassen `Bin`, `CondQueue`, `Res` und `WaitQueue` besteht in der Verwendung der Vererbungsbeziehung. Am Beispiel der Klasse `Bin` (Puffer), was analog für `CondQueue`, `Res` und `WaitQueue` gilt, würde dies bedeuten, daß `Bin` von `Queue` erbt. Damit stünden in `Bin` jedoch auch die Methoden `Insert` bzw. `Remove` zum Einreihen bzw. Entfernen von Entities zur Verfügung. Dies würde zwar auf bequeme Art die erforderliche Statistik über die durch Blockierung wartenden Entities abdecken, jedoch würde die Kapselung des internen Warteschlangen-Mechanisums dadurch aufgebrochen. Außerdem ist die Vererbung hier nicht die geeignete Beziehung zwischen `Bin` und `Queue`, denn Puffer sind keine Warteschlangen. Vielmehr sind Puffer mit Hilfe von Warteschlangen implementiert.

Dieses Problem wird schon in [Weber96] erwähnt, weshalb in SiFrame die Klasse `qBased` eingeführt wurde (s. Abbildung 2-13, S. 24). `qBased` wird von ihren Unterklassen mit der Warteschlange verbunden, die für die Implementierung verwendet wird. Die Klasse `Bin` erzeugt z.B. ein Objekt der Klasse `bQueue`, eine Unterklasse von `Queue`, und reicht dies an die Oberklasse `qBased` weiter. Dadurch kann `qBased` alle Statistikfunktionen an das `bQueue`-Objekt delegieren. Tatsächlich entstehen hier jedoch zwei reportfähige Objekte, `bQueue` und `Bin`, von denen aber nur eines für den Report genutzt wird. Bei der Klasse `WaitQ` wird ein ganz anderer Weg eingeschlagen. `WaitQ` erbt von `StatistikObject`

[29] Application-Frameworks sind Rahmenwerke zur Anwendungsentwicklung.

die Schnittstelle und Funktionalität für eine Warteschlangenstatistik, die von `WaitQ` jedoch nicht aktualisiert wird. Statt dessen führt `WaitQ` für jede der beiden impliziten Warteschlangen eine neue Schnittstelle für die Statistik ein. Die Vererbungsbeziehung zwischen `StatistikObject` und `WaitQ` macht hier also überhaupt keinen Sinn.

In DESMO-C wurde daher eine andere Lösung gewählt. Es entstand eine Klasse für die Implementierung von Warteschlangen, die Klasse `QueueImpl`. Jede Klasse die auf diesem Mechanismus basiert, kann ein Objekt von `QueueImpl` erzeugen und nutzen. Die Klasse `QueueBased` ist eine Unterklasse von `Reportable`, und ist für die Funktionen der Warteschlangenstatistik verantwortlich. Ein `QueueImpl`-Objekt wird mit einem `QueueBased`-Objekt verbunden, so daß das `QueueImpl`-Objekt das `QueueBased`-Objekt über die Methoden `addItem` und `delItem` anweisen kann, die Statistik entsprechend zu aktualisieren. Diese Methoden sind nur für die Klasse `QueueImpl` zugänglich, da sie in `QueueBased` als `friend` deklariert ist.

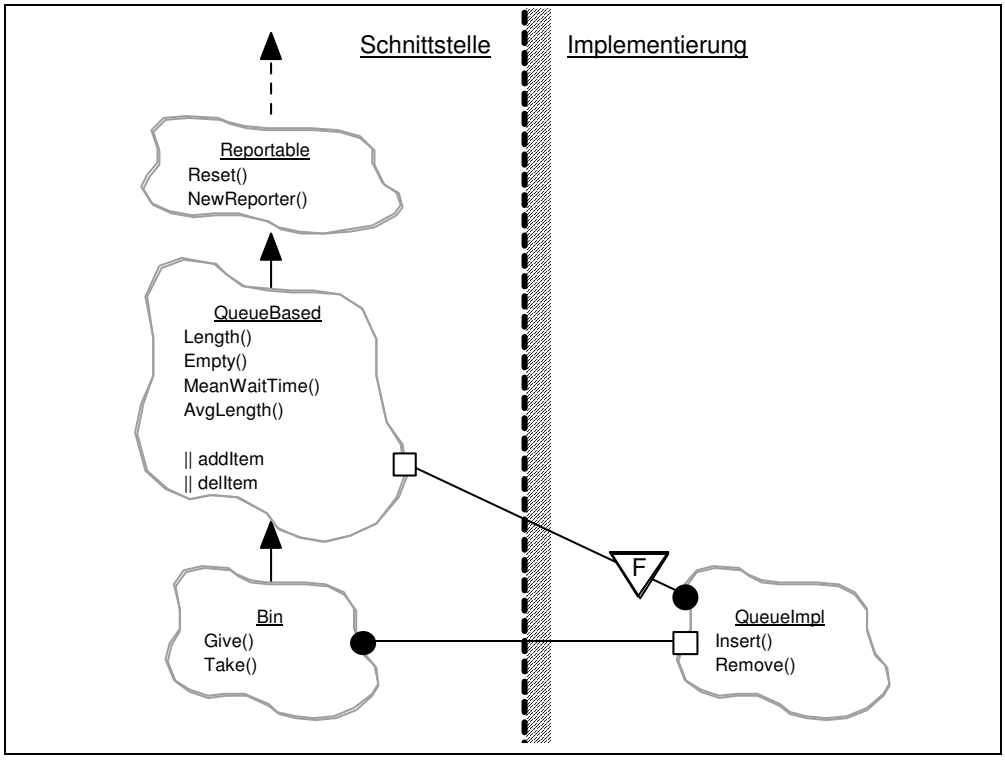

Abbildung 3-5: Implizite Warteschlange am Beispiel `Bin`

Die Klasse `QueueImpl` sorgt weder für Reportausgaben noch gibt sie Trace-Meldungen aus. Die für den Report relevanten Daten können über die von `QueueBased` geerbte Schnittstelle abgerufen werden. Für die Trace-Ausgaben sind die jeweiligen Unterklassen zuständig. Durch diese Entwurfsentscheidung können alle auf Warteschlangen basierenden Klassen direkt von `QueueBased` erben, ohne sich um die Führung der Warteschlangenstatistik kümmern zu müssen. Abbildung 3-6 gibt einen Überblick über alle Unterklassen von `QueueBased`.

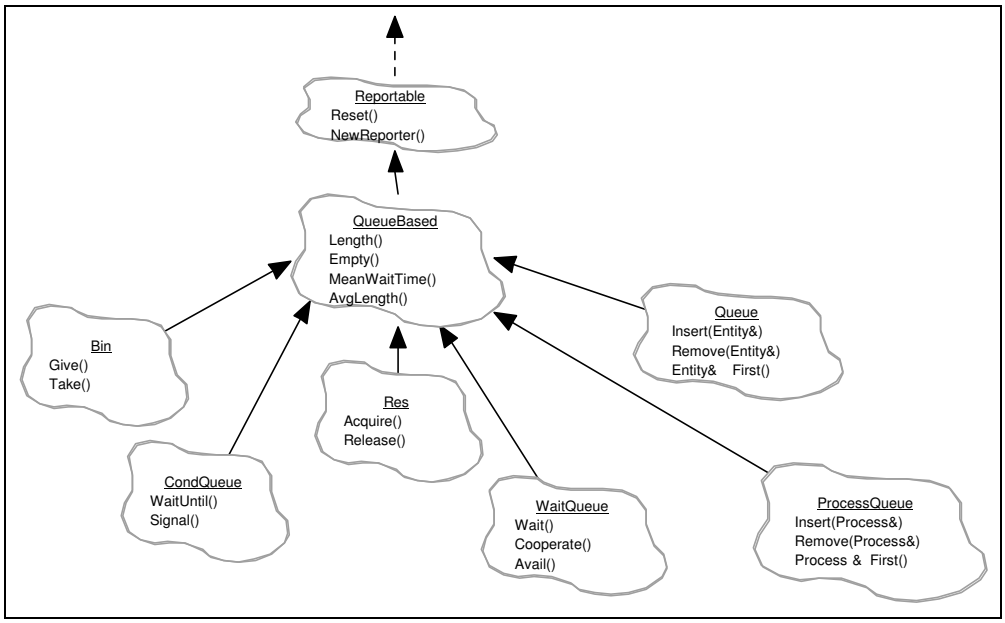

Abbildung 3-6: Auf Warteschlangen basierende Klassen

3.3.2 Warteschlangen für Entities und Prozesse

Durch die Zusammenführung der Weltbilder wurde zusätzlich zur Klasse `Queue`, die zur Aufnahme von Entities dient, die Klasse `ProcessQueue` eingeführt. Sie findet Verwendung für Warteschlangen in prozeßorientierten Modellen, in denen ausschließlich Prozesse warten können. Denn durch die Vererbungsbeziehung zwischen `Entity` und `Process` können in eine normale Warteschlange (`Queue`) sowohl Entities als auch Prozesse eingereiht werden. Solange diese bei der Entnahme nur als Entities behandelt werden sollen, stellt dies kein Problem dar. Ist es in einem Modell aber erforderlich, das ein entnommenes Entity ein Prozeß ist, so sollte eine `ProcessQueue` verwendet werden. In sie können ausschließlich Prozesse eingereiht werden.

Die Umsetzung erfolgt wie im Beispiel für Puffer (Abbildung 3-5). Dabei übernimmt die Oberklasse `QueueBased` die Funktionen für die Warteschlangenstatistik. Die Methoden zum Einreihen und Entfernen von Entities werden nach Ausgabe einer Trace-Meldung einfach an die implizite Warteschlange weitergereicht. Im Falle der Prozeßwarteschlangen müssen diese Methoden zusätzlich eine Typumwandlung von `Entity` nach `Process` vornehmen, da die implizite Warteschlange auf dem Typ `Entity` arbeitet.

DESMO-C übernimmt die Neuerung von SiFrame, den Warteschlangenmechanismus so einstellen zu können, daß ein Entity in mehreren Warteschlangen gleichzeitig stehen kann. Anders als bei SiFrame ist diese Warteschlangeneinstellung eine Eigenschaft des Entities. D.h. über die Entity-Methode `SetQueueOption` kann für ein bestimmtes Entity eingestellt werden, ob es in einer oder in mehreren Warteschlangen zur Zeit warten darf. Diese Einstellung kann ebenfalls für ein ganzes Modell vorgenommen werden. Die Option wird von allen Entities bei ihrer Erzeugung vom zugehörigen Modell übernommen. Ein Problem ergibt sich nur, wenn die Einstellung bei einem Entity von `MultipleQueue` auf `OnlyOneQueue` gesetzt wird, das sich gerade in mehreren Warteschlangen befindet. In diesem Fall wird eine Warnung ausgegeben und das Entity bei der nächsten Einfüge-Operation aus den übrigen Warteschlangen entfernt.

3.3.3 Suche in Warteschlangen

Sowohl in DESMO als auch in SiFrame gibt es die Möglichkeit, in Warteschlangen nach Entities zu suchen, die bestimmte Bedingungen erfüllen. In SiFrame muß von der Klasse `QueueCondition` abgeleitet werden, wobei die virtuelle Methode `Condition(Entity*)` implementiert werden muß, um das Kriterium zu beschreiben, das ein Entity erfüllen muß, um bei einer Suche gefunden zu werden. Ein Objekt dieser neu definierten Klasse muß bei seiner Erzeugung mit einer Warteschlange fest verbunden werden. Ist diese Verbindung hergestellt, kann mit der Suchbedingung eine Suche gestartet werden, indem an der Bedingung, nicht an der Warteschlange, die Methode `Search` aufgerufen wird, um das erste Entity zu finden. Listing 3-3 zeigt ein Beispiel für eine Suche in SiFrame:

```
 1. Queue myQueue(...);
 2.
 3. // Einfügen von Entities in myQueue:
 4. ...
 5.
 6. class PriorityOne : public QueueCondition {
 7.     public:
 8.         bool Condition (Entity* e)
 9.                   {    // Nullzeiger abfangen:
10.                        if (!e) return false;
11.                        return e->Priority() == 1;
12.                   }
13.     ...
14. } priorityIsOne (myQueue);
15.
16. // das erste Entity:
17. Entity* e1 = myQueue.First();
18.
19. // das erste Entity mit Priorität 1:
20. Entity* e2 = priorityIsOne.Search();
21.
22. // alle Entity-Namen mit Priorität 1 ausgeben:
23. Entity* e;
24. priorityIsOne.Reset()
25. while ((e = priorityIsOne.Search()) != 0)
26.     cout << e->Name() << endl;
```

Listing 3-3: Durchsuchen einer Warteschlange in SiFrame

In Listing 3-3 wird in Zeile 1 die Warteschlange `myQueue` angelegt. In den Zeilen 3 und 4 wird angenommen, daß in `myQueue` einige Entities eingereiht werden. In den Zeilen 6 bis 14 wird eine neue Bedingungsklasse definiert, deren Objekte dazu benutzt werden können, um nach Entities mit Priorität 1 zu suchen. In Zeile 14 wird das Bedingungsobjekt `priorityIsOne` angelegt und mit der Warteschlange `myQueue` verbunden. In Zeile 17 liefert der Aufruf von `First` das erste Entity, das in `myQueue` wartet. In Zeile 20 dagegen wird die Suchbedingung `priorityIsOne` benutzt, um das erste Entity aus `myQueue` zu finden, das Priorität 1 hat.

In den Zeilen 25 und 26 ist eine Schleife programmiert, die die Namen aller in `myQueue` wartenden Entities mit Priorität 1 ausgibt. Die Suchbedingung speichert die Position, ab der bei weiteren Aufrufen von `Search` die Suche fortgesetzt werden soll. Mit `Reset` wird diese interne Position wieder auf den Anfang der Warteschlange gesetzt. In Zeile 24 muß daher zunächst die Suchbedingung `priorityIsOne` zurückgesetzt werden. In Zeile 25 wird im Kopf der `while`-Schleife der Zeiger `e` auf das jeweils nächste Entity mit Priorität 1 gesetzt. Befindet sich kein solches Entity in `myQueue`, so wird ein Null-Zeiger geliefert, der die Schleife abbrechen läßt.

Bei der Speicherung der letzten Suchposition in dem Bedingungsobjekt muß das System gewährleisten, daß die Position immer einen gültigen Wert aufweist. Wird z.B. das Entity an dieser Position aus der Warteschlange entfernt, muß die Bedingung ihre Position auf das nächste Entity setzen. Sind zu diesem Zeitpunkt mehrere Bedingungen mit der Warteschlange verbunden, so müssen sie alle auf ihre korrekte Position hin überprüft werden. Außerdem erscheint es verwirrend, daß eine Bedingung angewiesen wird, etwas zu suchen.

Für DESMO-C wird daher ein anderer Ansatz gewählt. Eine Bedingung (Klasse `Condition`) zeichnet sich dadurch aus, daß mit ihrer Hilfe für ein gegebenes Entity ermitteln werden kann, ob es bestimmten Kriterien entspricht oder nicht (Methode `Check(Entity&)` analog zu SiFrame). Diese Bedingung ist ganz allgemein verwendbar und nicht z.B. an eine Warteschlange gebunden. Der Suchmechanismus wird durch Überladen der Methoden `First`, `Last`, `Pred` und `Succ` der Klasse `Queue` realisiert, die jeweils als zusätzlichen Parameter die Suchbedingung erhalten. Damit lassen sich Formulierungen wie “das erste Entity, daß die Bedingung erfüllt” oder “das nächste Entity nach Entity X, das die Bedingung erfüllt” leicht realisieren. Zusätzlich besteht durch die Methoden `Last` und `Pred` die Möglichkeit die Warteschlange in umgekehrter Richtung zu durchsuchen. Das oben beschriebene Problem beim Entfernen eines Entities aus der Warteschlange und einer nachfolgenden Folgesuche, muß jedoch vom Modellprogrammierer berücksichtigt werden. Da dieses Problem jedoch direkt im anwendenden Kontext ersichtlich ist, wird diese Einschränkung in Kauf genommen werden. Ein Beispiel soll dies verdeutlichen:

```
 1. Queue myQueue (...);
 2.
 3. // Einfügen von Entities:
 4. ...
 5.
 6. class PriorityOne : public Condition {
 7.     public:
 8.         bool Check (Entity& e)
 9.                 { return e.Priority() == 1; }
10.     ...
11. } priorityIsOne;
12.
13. // das erste Entity:
14. Entity& e1 = myQueue.First ();
15.
16. // das erste Entity mit Priorität 1:
17. Entity& e2 = myQueue.First (priorityIsOne);
18.
19. // alle Entity-Namen mit Priorität 1 ausgeben:
20. Entity* e = &myQueue.First (priorityIsOne);
21. while (!e->IsNullEntity())
22. {
23.     cout << e->Name() << endl;
24.     e = &myQueue.Succ (*e, priorityIsOne);
25. }
```

Listing 3-4: Durchsuchen einer Warteschlange in DESMO-C

In Listing 3-4 wird in Zeile 1 die Warteschlange `myQueue` angelegt. In den Zeilen 3 und 4 wird angenommen, daß in `myQueue` einige Entities eingereiht werden. In den Zeilen 6 bis 11 wird eine neue Bedingungsklasse definiert, deren Objekte dazu benutzt werden können, um nach Entities mit Priorität 1 zu suchen. In Zeile 11 wird das Bedingungsobjekt `priorityIsOne` angelegt. In Zeile 14 liefert der Aufruf von `First` das erste Entity, das in `myQueue` wartet. In Zeile 17 dagegen wird durch die zusätzliche Angabe der Bedingung `priorityIsOne` das erste Entity aus `myQueue` geliefert, das Priorität 1 hat.

In den Zeilen 20 bis 25 ist eine Schleife programmiert, die die Namen aller in `myQueue` wartenden Entities mit Priorität 1 ausgibt. In Zeile 20 wird der Zeiger `e` auf das erste Entity mit Priorität 1 gesetzt. Befindet sich kein solches Entity in `myQueue`, so wird das Pseudo-Entity `NullEntity` geliefert. Dies ist auch das Abbruchkriterium in Zeile 21. In Zeile 24 wird der erste Nachfolger von `e` mit Priorität 1 ermittelt.

Ein weiterer Unterschied zu SiFrame ergibt sich aus der in DESMO-C verfolgten Konvention, Referenzen statt Zeiger zurückzugeben, wenn garantiert ein gültiges Objekt geliefert werden kann. Dadurch wird das Risiko, versehentlich einen Null-Zeiger zu dereferenzieren, erheblich abgesenkt. Im Falle einer fehlgeschlagenen Suche ist dies das `NullEntity`. Soll eine Schleife programmieren, die nach mehreren Entities sucht, so wird man dort einen Zeiger verwenden müssen. Das hat zur Folge, daß die von der Suche gelieferte Referenz über den Adress-Operator '&' in einen Zeiger gewandelt werden muß, und andererseits der Zeiger bei Übergabe als Parameter mittels Dereferenz-Operator '*' in eine Referenz. Dadurch wirkt der Programmtext etwas verwirrend.

Die Gegenüberstellung zweier Beispiele in SiFrame und DESMO-C soll die Unterschiede verdeutlichen:

```
 1.  // SiFrame:
 2.  Queue             myQueue("Warteschlange");
 3.  MyCondition       myCondition(myQueue);
 4.  Entity* e;
 5.
 6.  while ((e = myCondition.Search()) != 0) {
 7.        cout << e->Name() << endl;
 8.        myQueue.Remove (e);
 9.  }
10.
11.  // DESMO-C:
12.  Queue             myQueue("Warteschlange");
13.  MyCondition       myCondition();
14.  Entity* e = &myQueue.First (myCondition);
15.
16.  while (!e->IsNullEntity()) {
17.        cout << e->Name() << endl;
18.        Entity& tmp = *e;
19.        // erst Nachfolger bestimmen ...
20.        e = myQueue.Succ (*e, MyCondition);
21.        // ... dann aus der Queue entfernen
22.        myQueue.Remove (tmp);
23.  }
```

Listing 3-5: Durchsuchen einer Warteschlange mit Entfernen von Entities

In DESMO-C existiert also mit der Klasse `Condition` ein einheitliches Schema mit dessen Hilfe Entities in Warteschlangen gesucht werden können. Beim Einsatz von Suchbedingungen bei einer `WaitQueue`, wird der Methode `Check` grundsätzlich der Slave übergeben. Soll für die Kooperationsbedingung jedoch der Slave mit dem Master verglichen werden, so kann der Master einfach als Attribut der Unterklasse von `Condition` eingerichtet werden. In der Methode `Check` besteht somit Zugriff auf den Master. Damit entfällt auch die Doppeldeutigkeit bei der Übergabe von Master und Slave als Entity-Parameter der Bedingungsmethode bzw. -prozedur in SiFrame bzw. DESMO. Dort ergibt sich die Rolle, Master oder Slave, aus der Position in der Parameterliste, was eine häufige Fehlerquelle darstellt. Abgesehen davon ist es in vielen Fällen ausreichend, die Attribute des Slaves heranzuziehen, um die Bedingung auszuwerten.

3.4 Höhere Synchronisationsmechanismen

Wie auch DESMO verfügt DESMO-C über Konstrukte zur Synchronisation von Prozessen. Jedes der Konstrukte ist mit Hilfe einer Klasse realisiert. Ein Objekt einer solchen Klasse stellt einen Synchronisationspunkt dar, über den sich verschiedene Prozesse synchronisieren können. Prozesse werden dort u.U. solange blockiert, bis eine Bedingung erfüllt ist, die für die weiteren Aktivitäten des Prozesses erforderlich ist. In DESMO und SiFrame ist es möglich, einen blockierten Prozeß explizit zu aktivieren. Die Reaktion darauf hängt allerdings vom jeweiligen Synchronisationspunkt ab, in dem der Prozeß blockiert ist. Bei einer `CondQ` z.B. ist es dadurch möglich, einen Prozeß durch explizite Aktivierung aus der Blockade zu befreien, und ihn somit gegenüber anderen in derselben `CondQ` wartenden Prozessen vorzuziehen.

Bei als Slaves in einer `WaitQ` wartenden Prozessen kann eine explizite Aktivierung sogar ohne weiteres vorgenommen werden. Der Slave "denkt" dann, er hätte bereits mit einem anderen Prozeß kooperiert, obwohl eine Kooperation nie stattgefunden hat. Im Falle von Unterbrechungen wird auf jeden Fall der Code zur Kennzeichnung der Unterbrechungsursache gesetzt, auch wenn letztlich gar keine Aktivierung erfolgt.

Der Versuch, einen blockierten Prozeß explizit zu aktivieren oder zu unterbrechen, führt im Gegensatz zu DESMO und SiFrame in DESMO-C zu einer Warnmeldung, wird aber ansonsten ignoriert. Der Blockadezustand eines Prozesses kann mit Hilfe der Prozeßmethode `Blocked` abgefragt werden. Aufheben läßt sich dieser Zustand jedoch nicht explizit, da sonst das Konzept des jeweiligen Synchronisationskonstrukts nicht aufgebrochen würde.

Ein diskussionswürdiger Punkt ist die Frage nach dem Aktivierungszeitpunkt von blockierten Prozessen. In DESMO wird ein wartender Prozeß zum Zeitpunkt 0.0 aktiviert, um die Bedingung für seine Blockade zu überprüfen. Der Grund für diese Entscheidung war, daß etwaige Zustandsänderungen zum gleichen Zeitpunkt noch berücksichtigt werden können[30]. Bei Puffern ist z.B. folgende Situation möglich:

> Vor einem leeren Puffer wartet ein Konsument k_1 auf genau ein Produkt. Für den aktuellen Zeitpunkt (0.0) ist ein anderer Konsument k_2 mit einer höheren Priorität als k_1 vorgemerkt, um zwei Produkte zu konsumieren. Der gerade laufende Prozeß gebe nun ein Produkt in den Puffer. Dies würde in DESMO dazu führen, daß k_1 hinter k_2 implizit vorgemerkt würde. D.h. zunächst würde k_2 aktiviert, der nun zwei Produkte anfordert. Da nur ein Produkt vorhanden ist, muß er blockiert werden, und wartet aufgrund seiner höheren Priorität vor k_1 in der Warteschlange, so daß dieser mindestens solange blockiert ist wie k_2.

Allerdings sind solche Abläufe leichter nachzuvollziehen, wenn die implizite Aktivierung eines blockierten Prozesses durch Änderung einer Synchronisationsbedingung grundsätzlich hinter dem laufenden Prozeß vorgenommen wird. Im Beispiel würden die Produktion und deren Konsequenzen dann mehr zu einer Einheit zusammengefaßt werden. Aus diesem Grund werden alle impliziten Aktivierungen in DESMO-C einheitlich nach dem laufenden Prozeß vorgenommen. Dazu zählt auch die Aktivierung der Nachfolger von Prozessen, die aus ihrer Blockierung befreit werden.

[30] Vgl. [Bölck89], S. 151 - 153

3.4.1 Puffer (Bin)

Puffer in DESMO-C entsprechen weitestgehend denen in DESMO und SiFrame, daher sollen hier nur die Unterschiede behandelt werden. Zusätzlich zu der Methode `Users`, die die Anzahl der produzierenden Zugriffe auf den Puffer liefert, werden die Methoden `Consumers` und `Producers` angeboten. Dabei ist `Producers` synonym zu `Users` zu verwenden. Sie wurde lediglich eingeführt, da der Name `Users` allein nicht hinreichend darüber informiert, ob es sich um produzierende oder konsumierende Nutzungen handelt. Zusätzlich liefert `Consumers` die Zahl der konsumierenden Zugriffe.

Ein Puffer kann bei seiner Erzeugung mit einem initialen Inhalt versehen werden. Die Zahl dieser Produkte liefert die Methode `Initial`. Die Warteschlangenstatistik wird über die Klasse `QueueBased` abgewickelt (s. Abschnitt 3.3).

Eine besondere Situation ergibt sich in DESMO bei dem Versuch, einen Puffer zu entleeren, indem alle verfügbaren Produkte entnommen werden `(Take (Avail()))`. Sind in diesem Moment keine Produkte vorhanden, der wird der Aufruf zu `Take (0)` und führt zur Blockierung des Konsumenten, wenn vor dem Puffer bereits ein anderer Prozeß wartet. In DESMO-C wird dieser Fall abgefangen, so daß eine Anforderung von 0, in keinem Fall zu einer Blockierung führt. Die Statistik wird dadurch nicht aktualisiert. So kann ein Prozeß einen Puffer vollständig entleeren, ohne diese Anweisung durch die Abfrage schützen zu müssen, ob überhaupt Produkte vorhanden sind:

```
1. Bin myBin (...);
2. ...
3. // if (myBin.Avail() > 0)     // kann entfallen
4. myBin.Take (myBin.Avail());  // alle entnehmen
```

Listing 3-6: Entleeren eines Puffers

Die Trace-Ausgabe für die Methode `Take` lautet in DESMO-C `'takes ...'` anstatt `'seizes ...'`, um eine leichtere Unterscheidung von Ressourcen zu ermöglichen.

3.4.2 Bedingtes Warten (CondQueue)

Warteschlangen für bedingtes Warten (`CondQueue`) realisieren eine Klasse von Synchronisationspunkten, an denen Prozesse solange blockiert werden, bis eine anzugebende Bedingung erfüllt ist. Mittels `WaitUntil` können sich Prozesse unter Angabe einer Bedingung (`Condition`) in ein `CondQueue`-Objekt einreihen. Im Gegensatz zu SiFrame sind hier keine speziellen Bedingungen erforderlich. Es kann mit den selben Bedingungen wie beim Durchsuchen von Warteschlangen gearbeitet werden.

Der Aufruf von `Signal` führt in jedem Fall dazu, daß der erste wartende Prozeß hinter dem aktuellen Prozeß vorgemerkt wird, auch wenn ersterer (durch einen früheren Aufruf von `Signal`) bereits vorgemerkt war.

Da sich Prozesse in einer `CondQueue` mit unterschiedlichen Bedingungen eintragen können, wird im Gegensatz zu DESMO und SiFrame im Trace der Name der Bedingung sowohl beim Blockieren als auch beim Verlassen mit ausgegeben. Diese zusätzliche Ausgabe wird jedoch unterdrückt, wenn die Bedingung nicht mit einem Namen versehen wurde.

Beispiel DESMO und SiFrame:

```
Ship 1    waits until in 'Reede'
Ship 1    leaves 'Reede'
```

Beispiel DESMO-C:

```
Ship 1    waits in 'Reede' until 'it can dock'
Ship 1    leaves 'Reede', cause 'it can dock'
```

3.4.3 Ressourcenwettbewerb (Res)

Ein Pool von Ressourcen entspricht in DESMO-C einem Objekt der Klasse `Res`. Mit den Methoden `Acquire` und `Release` können einzelne Ressourcen des Pools belegt bzw. wieder freigegeben werden.

Bei der Modellierung des Wettbewerbs von Prozessen um knappe Ressourcen, kann es typischer Weise zu Deadlock-Situationen kommen, in denen ein Prozeß u.U. nie wieder aus seiner Blockade befreit wird. In DESMO gibt es verschiedene Strategien, um solche Situationen aufzuspüren. Neben der statischen Überwachung, bei der geprüft wird, ob die Belegungsreihenfolge mit der Deklarationsreihenfolge von Ressourcen übereinstimmt, wird eine dynamische Überwachung in zwei Varianten angeboten, `DynamicA` und `DynamicB`. Für die dynamische Überwachung wird intern anhand der Belegungen ein Allokationsgraf erstellt, der auf Zyklenfreiheit überprüft wird.[31] `DynamicB` prüft erst, wenn feststeht, daß der Prozeß tatsächlich blockiert werden muß. `DynamicA` ist hingegen eine strengere Form, bei der die Überprüfung schon bei dem Versuch erfolgt, eine Ressource zu belegen. Dadurch wird allein die Möglichkeit einer Verklemmung entdeckt, ohne daß es je zu einem tatsächlichen Deadlock kommen muß. Um das Laufzeitverhalten bei ausgetesteten Programmen verbessern zu können, kann die Deadlock-Überwachung abgeschaltet werden.

In SiFrame ist nur die Strategie `DynamicA` implementiert. Es gibt keine Möglichkeit, den Überwachungsmodus aus- bzw. umzuschalten. In DESMO-C konnte die in SiFrame zur Darstellung des Allokationsgrafen verwendete Ressourcedatenbank übernommen und derart erweitert werden, daß nun die Stufen `DynamicA` und `DynamicB` realisiert sind und sich die Überwachung auch abschalten läßt. Dabei gibt es eine Ressourcedatenbank pro Experiment. Die unterschiedlichen Überwachungsstufen werden über die `Experiment`-Methode `DeadlockLevel` eingestellt. Wurden einmal Ressourcen belegt, kann die Überwachung nur noch abgeschaltet werden (`Off`).

3.4.4 Direkte Prozeßkooperation (WaitQueue)

`WaitQueue`-Objekte realisieren das Prinzip der Rendevouz-Synchronisation. Über ein `WaitQueue`-Objekt können Prozesse aufeinander warten, um miteinander zu kooperieren. Dabei nehmen die beteiligten Prozesse unterschiedliche Rollen ein, so daß eine Master-Slave-Beziehung entsteht. Der Master übernimmt die Durchführung der gemeinsamen Handlungen, während der Slave passiv bleibt.

In SiFrame ist die Prozeßkooperation als eigene Klasse realisiert. In der von ihr abgeleiteten Unterklasse wird die Methode `CoOperation (Entity*, Entity*)` definiert. Dieser Methode wird beim Zustandekommen der Kooperation im ersten Entity-Parameter der Master und im zweiten der Slave übergeben. So besteht innerhalb der Kooperationsmethode Zugriff auf Attribute und Methoden von Master und Slave. Dies bedeutet, daß um an die Master-Attribute zu gelangen, eine Typumwandlung des ersten Parameters durchge-

[31] Vgl. [Bölck89], S. 138 ff. und[Page91], S. 198 ff.

führt werden muß, um anschließend über die so gewonnene Referenz (oder Zeiger) Master-Elemente ansprechen zu können. Gleiches gilt für den zweiten Parameter bzw. den Slave. Wie auch schon in DESMO ergibt sich die Rolle aus der Position innerhalb der Parameterliste, was eine fehlerträchtige Angelegenheit ist.

Eine gute Alternative scheint zu sein, die Kooperation als Methode des Masters zu formulieren. Ein Zeiger auf diese Methode kann der Master beim Äußern seines Kooperationswunsches mit angeben. Beim Zustandekommen der Kooperation kann dann über den Funktionszeiger die Kooperationsmethode aufgerufen werden. Da diese Teil des Masters ist, stehen alle Elemente des Masters direkt zur Verfügung (`this` zeigt auf den Master).

Jedoch ist dieser Ansatz nicht nicht realisierbar. Denn bei Übergabe eines Zeigers auf eine Elementfunktion muß deren Klasse, eine Unterklasse von `Process`, bekannt sein. Da diese Klasse aber erst vom Modellentwickler definiert wird, kann die Simulationsbibliothek diese Klasse nicht kennen. Es ließe sich allenfalls ein Zeiger auf eine Elementfunktion von `Process` deklarieren, aber dort sind die Handlungen der Kooperation nicht bekannt, da sie erst in der konkreten Unterklasse formuliert werden. Die Möglichkeit, in `Process` eine rein virtuelle Methode für die gemeinsamen Handlungen vorzusehen, läßt pro Prozeß maximal eine Kooperationsvariante zu. Aus diesem Grunde wird diese Möglichkeit nicht weiter verfolgt.

Der Ansatz in DESMO erlaubt auch die Kooperation zwischen einem Master und mehreren Slaves. Dies kann erreicht werden, indem in der Kooperation erneut ein Kooperationswunsch geäußert wird, so daß es zu kaskadierenden Kooperationen kommt. Durch die rekursive Natur dieser Verfahrensweise lassen sich eigentlich nur Kooperationen nach dem LIFO-Prinzip durchführen, da die Kooperation mit dem letzten Slave auch als erste wieder endet. Um trotzdem eine FIFO-Strategie zu ermöglichen, ist es in DESMO erlaubt, einen Slave noch vor Ende der Kooperation explizit zu aktivieren. Problematisch dabei ist jedoch, daß die Kooperation, die ja eigentlich die gemeinsamen Handlungen der beiden Prozesse repräsentiert, vom Master allein zu Ende geführt wird, während der Slave seine Rolle als socher bereits abgelegt hat.

Dieses Problem ist lösbar, wenn die Kooperation selbst als Prozeß modelliert werden kann. Jeder dieser Kooperationsprozesse repräsentiert dabei die gemeinsamen Handlungen von genau zwei Prozessen. Damit ist es möglich mehrere Kooperationen eines Masters mit mehreren Slaves tatsächlich auch als nebenläufig zu formulieren. Allerdings steigt die Komplexität der Verwaltungsaufgaben hierbei drastisch an und es müssen Fragen geklärt werden, deren Bearbeitung den Umfang dieser Arbeit jedoch übersteigt. Z.B. muß gewährleistet sein, daß ein Prozeß seine seine Rolle als Master erst dann wieder ablegt, wenn alle Kooperationsstränge ordnungsgemäß beendet sind. Das Problem des Zugriffs auf die Attribute des Masters bleibt außerdem bestehen.

In DESMO-C wird also die SiFrame-Umsetzung übernommen, jedoch mit folgenden Änderungen bzw. Erweiterungen:

- Die Klasse `ProcessCooperation` stellt alle in `Process` verfügbaren Methoden bereit, die den Aufruf an die entsprechenden Methoden des Masters durchreichen. Dadurch kann die Formulierung der gemeinsamen Handlungen größtenteils aus der Sicht des Masters erfolgen.

- Die Methode `WaitQueue::Cooperate` wird in Analogie zur Suche in Warteschlangen überladen. Eine Version besitzt einen Parameter für die Kooperation wie in

SiFrame. Die andere Version erhält einen zusätzlichen Parameter für eine Bedingung (`Condition`). Diese zweite Version ersetzt die frühere Methode `Find`, mit der ein Master eine Bedingung angeben kann, die ein Slave erfüllen muß, damit eine Kooperation zustande kommt. Die Technik des Überladens entspricht hier genau der Funktionsweise der Methode. Denn abgesehen von der Kooperationsbedingung bewirken beide Methoden das gleiche. Die Angabe der Bedingung stellt tatsächlich eine Erweiterung der Funktionalität von `Cooperate` dar. Dies kommt durch den zusätzlichen Parameter bei gleichem Methodennamen gut zum Ausdruck.

- Die Methoden zur Statistik der beiden impliziten Warteschlangen können über die aus `Queue` bekannten Methoden (`Length`, `AvgWaitTime` etc.) abgerufen werden, hier jedoch mit dem Präfix `'m'` für die Master- und `'s'` für die Slave-Warteschlange. Die Statistik der Master-Warteschlange kann synonym über die entsprechenden Methoden der Oberklasse `QueueBased` abgefragt werden.

- Die Prozeßmethode `Owner`, zum Ermitteln des Masters eines in Kooperation befindlichen Slaves, wurde in DESMO-C umbenannt in `Master`, da dieser Bezeichner leichter mit seiner Funktion in Verbindung gebracht werden kann. Aus dem selben Grund wurde die Methode `Avail`, die feststellt, ob ein Prozeß durch den Aufruf von `WaitQueue::Wait` als Slave auf eine Kooperation wartet, umbenannt in `CanCooperate`.

Auf eine wichtige Besonderheit im Zusammenhang mit der Unterbrechung von Prozessen werde ich im folgenden Abschnitt gesondert eingehen.

Im Trace werden die Namen von Kooperation und evtl. Bedingung mit ausgegeben, falls für sie Namen vergeben wurden und die Trace-Fähigkeit für sie nicht ausgeschaltet wurde (mittels `ModelComponent::ShowInTrace`). Hierzu eine Beispiel:

```
myWaitQueue.Cooperate (mating, onlyLargeTrucks);
```

führt bei Blockierung zu folgender Trace-Ausgabe:

```
'D 1' waits in 'WaitQ M' for 'large truck'
```

und beim Zustandekommen der Kooperation:

```
'D 1' finds 'large truck' 'Truck 1' in 'WaitQ S' for 'Mating'
```

3.4.5 Unterbrechung von Prozessen

Prozesse können jederzeit unterbrochen werden. Sie bekommen den Grund für die Unterbrechung als Code überliefert. Der Code wird in den Systemattributen des Prozesses solange gespeichert, bis er entweder erneut unterbrochen wird oder die Methode `ClearInterruptCode` aufgerufen wird. In DESMO führt der Versuch, einen blockierten Prozeß zu unterbrechen, dazu, daß die Bedingung für die Blockade erneut überprüft wird und es ggf. zu einer Aktivierung kommt. Die Unterbrechungsursache wird auf jeden Fall gespeichert. Im schlimmsten Fall wird der Prozeß aus seiner Blockade befreit, was den Synchronisationsmechanismus aufbricht. Bei einem in einer `CondQueue` wartenden Prozeß z.B. führt eine Unterbrechung dazu, daß die Bedingung ausgewertet wird, und der Prozeß ggf. unabhängig von seiner Position in der Warteschlange aktiviert wird. Bei einem als Slave blockierten Prozess führt eine Unterbrechung sogar auf jeden Fall zur Aktivierung. Für den Prozeß hat es dann den Anschein, als wäre die Kooperation ordnungsgemäß vollendet worden, obwohl es tatsächlich nie zu einer Kooperation gekommen ist.

In DESMO-C ist es daher grundsätzlich nicht möglich, blockierte Prozesse zu unterbrechen. Der Versuch wird mit einer Warnung quittiert, ansonsten bleibt der Unterbrechungsversuch wirkungslos. Im Zusammenhang mit direkter Prozeßkooperation kommt in DESMO-C eine Erweiterung ins Spiel. In DESMO ist es nicht möglich einen kooperierenden Slave ordnungsgemäß zu unterbrechen, denn eine Unterbrechung wirkt nur auf den Slave, während der Master davon unberührt die Kooperation fortführt. In DESMO-C wird daher die Unterbrechung an den Master weitergeleitet, so daß letztlich die Kooperation von Master und Slave unterbrochen wird. Zunächst wird der Unterbrechungsgrund nur beim Master eingetragen. Wurde jedoch vor dem Ende der Kooperation `ClearInterruptCode` nicht aufgerufen, wird der `InterruptCode` vom Master auf den Slave übertragen, so daß er von beiden Prozessen abgefragt werden kann. Ist die Unterbrechung in der Kooperation behandelt worden, gehen beide Prozesse ohne gesetzten Unterbrechungsgrund aus der Kooperation hervor. Damit ist es ausreichend einen beliebigen Kooperationspartner zu unterbrechen, wenn die Kooperation unterbrochen werden soll.

Im Trace wird der Name des Unterbrechungsgrundes, sofern vorhanden, angegeben. Der Code, der implizit vergeben wird, wird in jedem Fall in eckigen Klammern nachgestellt.

Beispiel:

```
L-Dig 2    interrupts 'S-Dig 1', cause: 'L-Dig free' [2]
```

3.5 Statistische Datensammelobjekte

Zu den Statistikobjekten zählen in DESMO `Count`, `Tally`, `Histogram`, `Accumulate`, `TimeSeries` und `Regression`. Allen gemeinsam ist eine Prozedur `Update`, um die Statistik fortzuschreiben. Jedes Objekt wird mit einer Funktion verbunden, die den entsprechenden Beobachtungswert liefert. Ausnahmen hiervon sind `Count`, hier wird `Update` mit einem ganzzahligen Parameter aufgerufen, und `Regression`, dem zwei Beobachtungsfunktionen zugeordnet werden müssen.

Im objektorientierten Ansatz bieten sich drei Implementierungsalternativen an:

1. Bei der Erzeugung eines Statistikobjektes wird wie in DESMO ein Funktionszeiger auf die Beobachtungsfunktion übergeben.

2. Die Beobachtungsfunktion ist eine virtuelle Methode des Statistikobjekts, die von Unterklassen definiert wird.

3. Es gibt eine Klasse für die Beobachtungsfunktionen, von der abgeleitet werden muß, um die den Beobachtungswert liefernde Methode zu überschreiben. Ein Objekt einer solchen Klasse wird dann mit einem Statistikobjekt verbunden.

Da in C bzw. C++ Funktionszeiger etwas “unbequem” sind und die Lesbarkeit des Programms beeinträchtigen, wird Variante 1 nicht in Betracht gezogen.

Alternative 2 hat den Vorteil, daß Statistikobjekt und Beobachtungsfunktion auf natürliche Art und Weise miteinander verbunden sind. Allerdings erfordert sie, daß von den Klassen der Statistikobjekte erst abgeleitet wird. Außerdem können zwei verschiedene Statistikobjekte nicht dieselbe Beobachtungsfunktion verwenden, wenn z.B. eine Größe zusätzlich über `TimeSeries` in einer Datei protokolliert werden soll. Die Beobachtungsfunktion

müßte dann zweimal implementiert werden, einmal beim eigentlichen Statistikobjekt und ein zweites mal bei dem `TimeSeries`-Objekt.

Variante 3 trennt die Berechnung der Beobachtungsgröße vom Statistikobjekt. Dies erhöht einerseits die Verwendbarkeit der Beobachtungsgröße und verlagert die Notwendigkeit, eigene Unterklassen zu bilden, auf den Klassenbaum für Beobachtungsfunktionen. Es ergeben sich die folgenden Vorteile, weswegen diese Variante für DESMO-C gewählt wurde:

- mehrere Statistikobjekte können dieselbe Beobachtungsfunktion verwenden
- die Möglichkeit, die Beobachtungsfunktion zu wechseln, kann leicht implementiert werden

3.5.1 Wertlieferanten für Beobachtungsgrößen

DESMO-C bietet eine Klasse `ValueSupplier` an, deren Objekte die Aufgabe haben, den Wert einer statistisch auszuwertenden Beobachtungsgröße zu liefern. Von ihr müssen Unterklassen gebildet werden, die die rein virtuelle Methode `Value` definieren, um den entsprechenden Wert zu ermitteln. Ein Objekt solch einer Unterklasse kann einem statistischen Datensammelobjekt übergeben werden. Wird es aktualisiert, greift es auf den Wertlieferanten zurück, um sich den aktuellen Wert der Beobachtungsgröße zu beschaffen.

Ein statistisches Datensammelobjekt ist während seiner gesamten Lebensdauer mit genau einem Wertlieferanten verbunden. Letzterer kann jedoch mit beliebig vielen Datensammelobjekten verbunden werden. Dadurch ist es z.B. möglich, das selbe `ValueSupplier`-Objekt sowohl mit einem `TimeSeries`-Objekt als auch mit einem `Tally`-Objekt zu verbinden.

`ValueSupplier` erbt von `ModelComponent`. Somit stehen innerhalb der Methode `Value` die Methoden zur Verfügung, auf die alle Modellkomponenten Zugriff haben, z.B. `CurrentTime`, `CurrentProcess` etc. Als nützlich kann sich erweisen, mit Hilfe der Methode `TraceNote` eine Ausgabe in den Trace zu schreiben, um die korrekte Ermittlung der Beobachtungsgröße zu prüfen, bzw. mittels `Warning` bei zweifelhaften Situationen eine entsprechende Mitteilung zu machen.

3.5.2 Statistik über Beobachtungsgrößen

Die Oberklasse aller statistischen Datensammelobjekte ist in DESMO-C die Klasse `StatisticObject`. Sie führt als neues Schnittstellenelement die rein virtuelle Methode `Update` ein, die in Unterklassen definiert wird, um die spezielle Statistik zu aktualisieren. Außerdem ist hier die Funktionalität vorgesehen, die über die nur für Unterklassen zugängliche Methode `traceUpdate` eine Aktualisierung im Trace protokolliert, falls dies gewünscht ist. Als Unterklassen von `StatisticObject` ergeben sich `Count`, `TimeSeries`, `Regression`, `Tally`, `Histogram` und `Accumulate`. Eine Übersicht zeigt Abbildung 3-7:

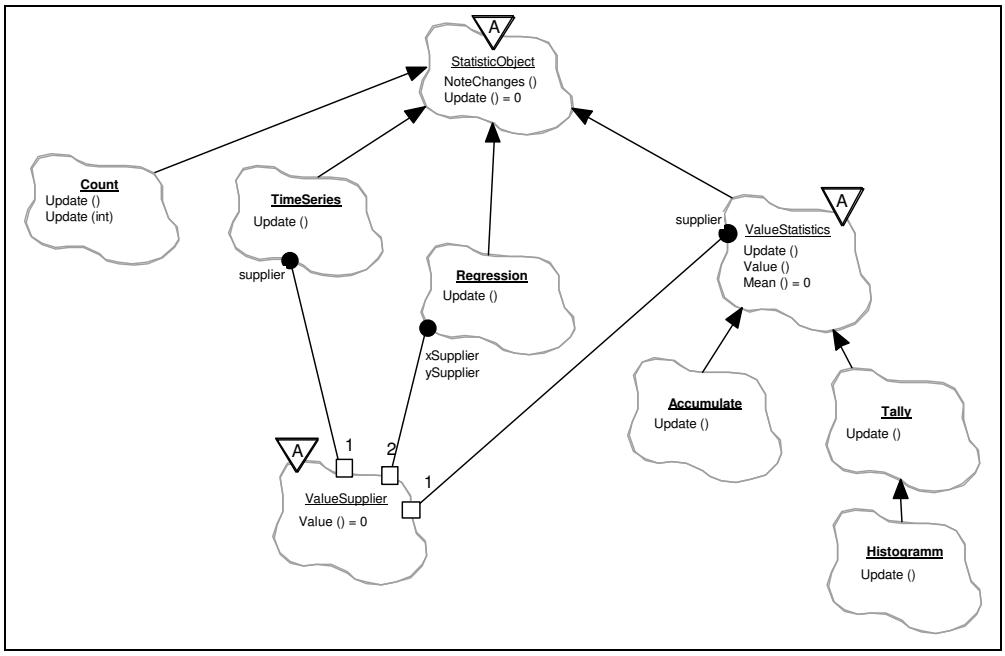

Abbildung 3-7: Klassenhierarchie der statistischen Datensammelobjekte

Die Klasse `Count` realisiert einfache Zähler. Eine Beobachtungsgröße wird nicht benötigt, da der Zähler bei jedem `Update`-Aufruf um Eins bzw. einen anzugebenden Wert erhöht wird. Als Zähler wird einfach die Anzahl der Beobachtungen genutzt, die in der Oberklasse `Reportable` bereits geführt wird. Somit muß die Methode `Update` lediglich `Reportable::incObservations` aufrufen. Damit Zählerobjekte im Report innerhalb einer Gruppe mit eigenem Titel aufgeführt werden, muß `Count` einen entsprechenden Reporter liefern können, der sich von einem allgemeinen Reporter jedoch nur marginal unterscheidet.

`TimeSeries`-Objekte für die Protokollierung eines Wertes in eine Datei werden mit einem `ValueSupplier`-Objekt verbunden, das den bei jedem `Update` in die Datei zu schreibenden Wert liefert. Bei der Erzeugung werden einem `TimeSeries`-Objekt ein Dateiname, sowie zwei Parameter vom Typ `SimTime` übergeben, die das Intervall angeben, innerhalb dessen die Werte protokolliert werden sollen. Da Zeitreihen nicht im Report erscheinen, wird kein Reporter benötigt. `NewReporter` liefert daher einen Null-Zeiger. Die Rücksetzung einer Zeitreihe führt zum Schließen der Datei und anschließendem erneuten Öffnen, so daß der bisherige Inhalt gelöscht wird.

Als Gemeinsamkeiten von `Tally` und `Accumulate` konnten folgende Eigenschaften in der Oberklasse `ValueStatistics` zusammengefaßt werden[32]:

- Beide sind mit einem `ValueSupplier`-Objekt verbunden, über den eine Statistik geführt wird.

- Die Methoden `Value`, für den zuletzt ermittelten Wert, `Minimum` und `Maximum`, für den seit der letzten Rücksetzung kleinsten bzw. größten beobachteten Wert, können unabhängig von den Unterklassen direkt in `ValueStatistics` implementiert werden. Die erforderlichen Werte werden bei einem Aufruf von `Update` aktualisiert.

[32] Prinzipiell hätte `TimeSeries` auch von `ValueStatistics` erben können, aber da Zeitreihen die Werte einfach nur protokollieren sollen, wurde auf den Mehraufwand zur Führung der Statistik wie in `ValueStatistics` verzichtet.

- Die Methoden `Mean` und `StdDev` sind als rein virtuelle Methoden eingerichtet, und werden erst in den Unterklassen von `Tally` bzw. `Accumulate` definiert, da die erforderlichen Berechnungsverfahren von der Art der Statistik in Hinblick auf die zeitliche Gewichtung abhängen.

`Accumulate` implementiert die Methoden `Mean` und `StdDev`, indem ein zeitlich gewichteter Mittelwert bzw. die Standardabweichung davon geliefert wird. Hierfür wird `Update` so erweitert, daß der Zeitpunkt der letzten Aktualisierung gespeichert wird. Beim Aufruf von `Mean` bzw. `StdDev` wird die seit diesem Zeitpunkt vergangene Zeit stets mit berücksichtigt. Durch die zeitliche Gewichtung ist es möglich, eine automatische Aktualisierung zu jedem Ereigniszeitpunkt zu wählen. Auf diesen Aspekt geht Abschnitt 3.5.3 näher ein.

`Tally` hingegen mittelt die beobachteten Werte über die Anzahl der Beobachtungen. `Mean` und `StdDev` sind entsprechend implementiert. Da über die Anzahl der Beobachtungen gemittelt wird, ist eine automatische Aktualisierung zu jedem Ereigniszeitpunkt nicht möglich. Wie dennoch eine Automatik realisiert werden kann, beschreibt Abschnitt 3.5.3

Da Histogramme zur Berechnung von Mittelwert und Standardabweichung die selben Verfahren benutzen wie `Tally`-Objekte, können Histogramme als eine Erweiterung von `Tally`-Objekten betrachtet werden. Es liegt daher nahe, daß `Histogram` von `Tally` erbt. Die geerbten Eigenschaften werden um die Fähigkeit erweitert, beobachtete Werte zu klassifizieren und die Anzahl der Beobachtungen für jede Werteklasse getrennt festhalten zu können. Dem Konstruktor wird ein Intervall übergeben und eine Anzahl von Zellen, in die das Intervall eingeteilt werden soll. Diese Daten lassen sich nachträglich über die Methode `ChangeParameter` ändern, jedoch nur, wenn zu dem Zeitpunkt keine Beobachtungen vorliegen, was auch nach einem `Reset` der Fall ist. Für jede Zelle wird im Report eine Zeile ausgegeben, die einem Balken eines Balkendiagramms entspricht und die Häufigkeit, daß ein Wert dieser Zelle zuzuordnen war, was eine Zelle mit anderen Zellen vergleichbar macht. Zusätzlich wird je eine Zelle für Unter- und Überläufe angelegt. Alle Informationen, die im Report erscheinen, lassen sich auch über entsprechende Methoden abrufen.

Die Klasse `Regression` stellt einen Sonderfall dar. Zur Berechnung der Statistik werden zwei Werte benötigt. In DESMO bzw. SiFrame wird hierfür eine eigene Prozedur bzw. Klasse eingeführt, die in der Lage ist, zwei Werte auf einmal zu liefern. In DESMO-C wird dies vermieden, indem ein `Regression`-Objekt mit zwei `ValueSupplier`-Objekten verbunden wird. Der Aufruf von `Update` führt folglich dazu, daß beide `ValueSupplier`-Objekte nacheinander aufgefordert werden, ihren Wert zu liefern. Auf die zuletzt ermittelten Werte kann über die Methoden `xValue` bzw. `yValue` zugegriffen werden. Zusätzlich zu DESMO und SiFrame werden die Methoden `xConstant` und `yConstant` angeboten, die einen booleschen Wert liefern, der anzeigt, ob die jeweiligen Werte seit dem letzten Rücksetzen konstant waren, oder so geringen Schwankungen unterlagen, daß sie als konstant betrachtet werden können.

3.5.3 Anwendung des Beobachtermusters

Als Erweiterung gegenüber DESMO kann das `ValueSupplier`-Objekt als Auslöser für die Aktualisierung des Datensammelobjekts konstruiert werden, so daß ein vollautomatischer Mechanismus entsteht. Hierzu wird eine vereinfachte Version des Beobachtermus-

ters[33] angewendet. Der `ValueSupplier` kann über Mehrfachvererbung zusätzlich von der Klasse der beobachtbaren Objekte erben (Klasse `Observable`), und kann so die Rolle eines Auslösers für die Aktualisierung der Statistik übernehmen. Die Statistikobjekte erben zusätzlich von `Observer`. Sie können sich bei einem `Observable` als Beobachter anmelden, der über Änderungen des beobachteten Objekts informiert werden möchte. Ändert sich ein `Observable`, so benachrichtigt es alle bei ihm angemeldeten `Observer`.

Dieser Mechanismus findet z.B. Anwendung bei der Realisierung der aus DESMO bekannten vollautomatischen Aktualisierung von Datensammelobjekten für zeitgewichtete Statistik. In diesem Fall wird die Aktualisierung zu jedem Simulationszeitpunkt vorgenommen. Um dies zu realisieren, meldet sich das Statistikobjekt bei der Simulationsuhr an, einem Objekt der Klasse `SimClock`. `SimClock` erbt ebenfalls von `Observable` und informiert ihre Beobachter vor dem Stellen der Uhrzeit. Der gleiche Mechanismus wird auch angewendet, um bei einem globalem Rücksetzen oder globalem Report die in DESMO realisierte Verzögerung bis zum Ende des aktuellen Simulationszeitpunktes zu erreichen[34]. Abbildung 3-8 zeigt die Beziehungen zwischen den Klassen, die diesen Mechanismus realisieren.

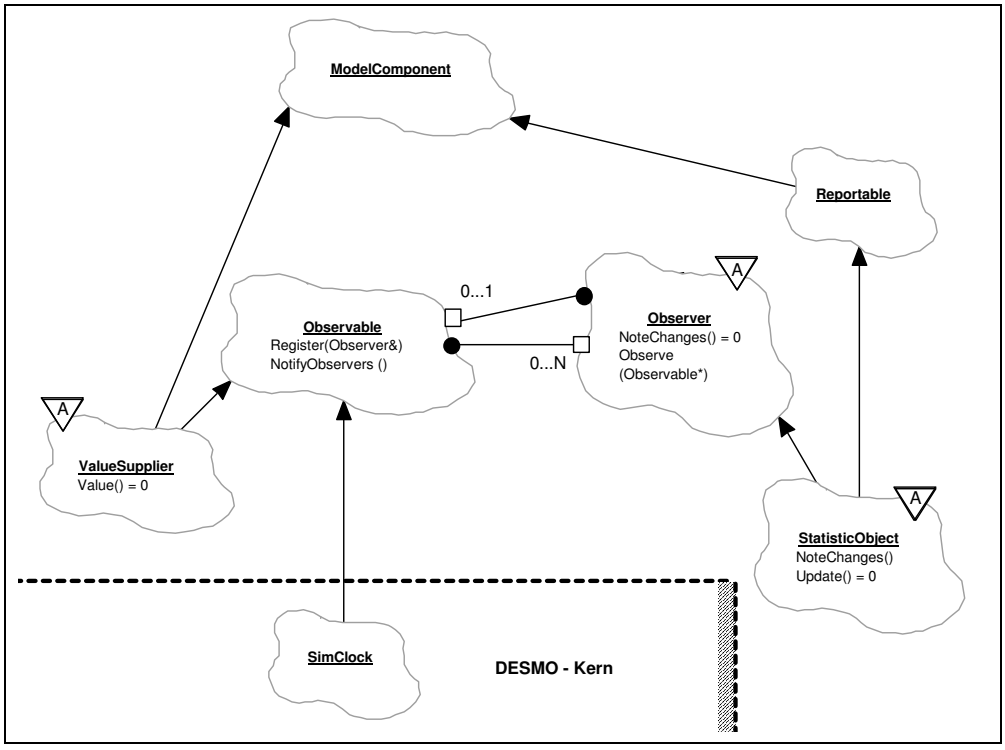

Abbildung 3-8: Beobachtermuster bei Statistikobjekten

Bei Objekten der Klassen `Accumulate` und `TimeSeries` kann die Aktualisierung nach jedem Simulationsschritt erfolgen, da die Statistik zeitgewichtet ist bzw. der Wert nur in eine Datei geschrieben wird (`TimeSeries`). Es entsteht dadurch jedoch ein erhöhter Rechenaufwand, da auch Aktualisierungen stattfinden, wenn sich die Beobachtungsgröße nicht geändert hat. Durch die Anwendung des Beobachtermusters läßt sich dieser Aufwand auf ein Minimum reduzieren, da die Aktualisierung nur noch vorgenommen werden muß, wenn sich der Wert tatsächlich geändert hat.

---

33 Zum Beobachtermuster vgl. [Gamma95].

34 Vgl. [Bölck89], S. 132 f. Auf die Rücksetzung von Objekten und den globalen Report wird in Abschnitt 3.6.3 näher eingegangen.

Darüber hinaus läßt sich durch diese Technik auch eine automatische Aktualisierung der nicht zeitgewichteten Statistiken erzielen. Als Auslöser für die Aktualisierung kann grundsätzlich jedes beliebige Objekt dienen, dessen Klasse von `Observable` erbt und gewährleistet, daß nach jeder Wertänderung der zu beobachtenden Größe die Methode `NotifyObservers` aufgerufen wird. Da `Observable` keine Oberklasse hat und eine minimale Schnittstelle aufweist, eignet sie sich als Mixin-Klasse für Mehrfachvererbung. Das folgende Beispiel soll einen einfachen Integer-Wert realisieren, der bei jeder Änderung seine Beobachter informiert:

```
 1. class ObservableIntSupplier : public ValueSupplier,
 2.                               public Observable
 3. {
 4.     public:
 5.         ObservableIntSupplier (Model& owner)
 6.             :    ValueSupplier (owner, "obs. Int"),
 7.                  Observable    (),
 8.                  i             (0)
 9.             {}
10.
11.         void Set (int newI) { i = newI;
12.                               NotifyObservers();
13.                             }
14.
15.         double Value () const { return i; }
16.
17.     private:
18.         int i;
19. } myInt (myModel);
20.
21. Tally myTally (myModel, "Auto-Tally", myInt);
22.
23. // myTally als Beobachter von myInt anmelden:
24. myTally.Observe (&myInt);
25.
26. // durch folgenden Aufruf wird myTally aktualisiert
27. myInt.Set (5);
```

Listing 3-7: Beispiel für ein automatisches Tally-Objekt

`ObservableIntSupplier` erbt von `ValueSupplier`, weshalb in Zeile 21 `myInt` dem `Tally`-Objekt `myTally` als Wertlieferant übergeben werden kann. Zusätzlich wird von `Observable` die Beobachtbarkeit geerbt, so daß in Zeile 24 `myTally` als Beobachter von `myInt` angemeldet werden kann. Wird nun mit der Methode `Set` der Wert von `myInt` geändert, kommt der Stein ins Rollen: In Zeile 11 wird zunächst der neue Wert von `myInt` gesetzt. In Zeile 12 werden daraufhin mit Hilfe der von `Observable` geerbten Methode `NotifyObservers` alle angemeldeten Beobachter, also auch `myTally`, benachrichtigt. D.h. bei `myTally` wird die von `Observer` geerbte Methode `NoteChange` aufgerufen, welche dafür sorgt, daß `Update`, die eigentliche Methode zum Aktualisieren der Statistik, aufgerufen wird. In `Update` wird schließlich die Methode `Value` des `ValueSupplier`-Objekts aufgerufen, welches genau das Objekt `myInt` ist.– M.a.W. bei jeder Änderung von `myInt` wird `myTally` automatisch aktualisiert. Das folgende Interaktionsdiagramm veranschaulicht die Aufrufkette:

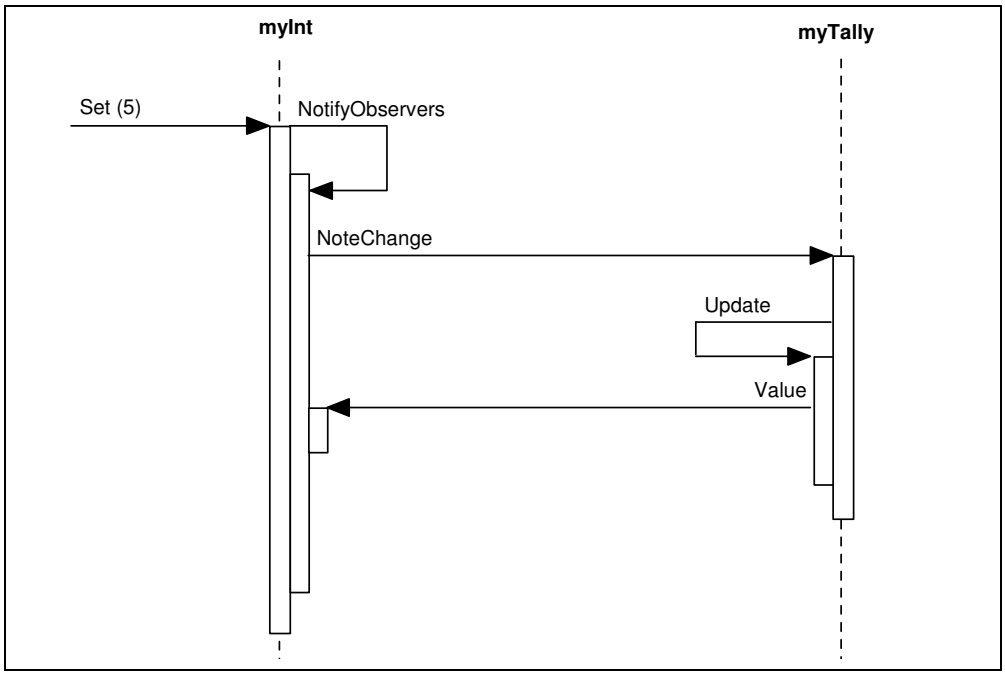

Abbildung 3-9: Interaktionsdiagramm zu Listing 3-7

3.6 Reportfähige Objekte

Einige der Objekte in DESMO beinhalten eine automatische Ausgabe aller relevanten Informationen in den Report. Zu dieser Art von Objekten gehören die statistischen Datensammelobjekte, Zufallszahlenströme und die auf Warteschlangen basierenden Objekte. In DESMO sind für diese Funktionalität in den Modulen dieser Objekte entsprechende Prozeduren vorgesehen, um die Informationen über ein Objekt dieses Moduls entsprechend aufbereitet in die Report-Datei zu schreiben. Anstelle der angebotenen Prozeduren können auch eigene installiert werden. Dieser Ansatz, der in SiFrame übernommen wurde, führt dazu, daß die Schnittstellen der Module bzw. Klassen mit den Methoden zur Reportgenerierung überfrachtet werden, da diese in jeder Modul- bzw. Klassenschnittstelle auftreten.

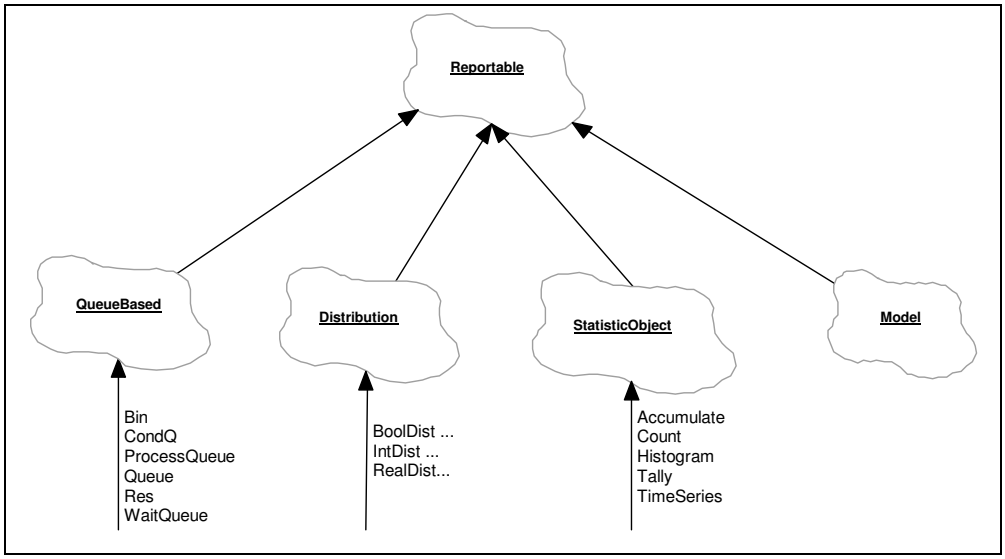

Abbildung 3-10: Unterklassen von `Reportable`

3.6.1 Ausgliederung der Reportfunktionalität

In DESMO-C wurde daher ein anderer Weg gewählt, der die Aufbereitung der Daten eines Objektes aus diesem ausgliedert. Dadurch konzentriert sich die Schnittstelle der jeweilige Klassen auf die eigentliche Funktionalität. Außerdem läßt sich die Darstellung der Informationen getrennt von den informationstragenden Klassen weiterentwickeln und pflegen.

Zu diesem Zweck wird in DESMO-C die Klasse `Reporter` eingeführt. Ein Reporter muß in der Lage sein, auf Kommando die Informationen zu einem bestimmten Objekt in einen gegebenes `ostream`-Objekt[35] zu schreiben. Ein Reporter muß das Objekt, über das er berichten soll, also insoweit kennen, daß er alle relevanten Informationen aus ihm herausziehen kann. In der Regel ist dies durch die Schnittstelle der entsprechenden Klasse gewährleistet. So lassen sich z.B. alle Informationen, die über eine Warteschlange im Report erscheinen sollen, über entsprechende Methoden der Klasse `QueueBased` abfragen. Alles, was ein reportfähiges Objekt tun muß, ist, auf Verlangen einen Reporter zu liefern, der über das Objekt berichten kann. Bei der Erzeugung wird dem Reporter ein Verweis auf das reportfähige Objekt übergeben, damit er weiß, über welches Objekt er berichten soll.

Auf der anderen Seite müssen Reporter ein Protokoll unterstützen, das es dem reportgenerierenden Mechanismus erlaubt, den Report in geeigneter Weise zusammenzustellen. Dazu gehört:

- Ein Reporter muß über eine Gruppen-ID einer Gruppe gleichartiger Objekte zugeordnet werden können. Z.B. liefern alle Reporter, die über eine Warteschlange berichten, über die Methode `GetGroupID` dieselbe Gruppen-ID. Um für eindeutige ID's zu sorgen, verwaltet die Klasse `Reporter` eine Menge von Nummern, von der sich jede neue Reporterklasse mit Hilfe der Methode `NewGroupID` eine neue noch nicht vergebene Nummer geben lassen kann. Die Operatoren <, <=, > und >= sind überladen, so daß mit ihrer Hilfe mehrere Reporter nach Gruppen-ID sortiert werden können.

- Reporter müssen den Klassennamen ausgeben und ihn unterstreichen können. Hierzu dienen die Methoden `WriteTitle` bzw. `UnderscoreTitle`.

- Für die tabellarische Ausgabe muß ein Tabellenkopf ausgegeben werden können, der die Spaltenüberschriften für die in den einzelnen Zeilen auszugebenden Informationen enthält. Dies geschieht über die Methoden `WriteHeader` bzw. `UnderscoreHeader`.

- Schließlich muß eine zeilenweise Ausgabe der Objektdaten mit Hilfe der Methode `WriteAsLine` erfolgen können. Als alternative Ausgabeform für ein einzelnes Objekt, z.B. in einem Informationsfenster einer grafischen Umgebung, kann `WriteAsBlock` verwendet werden.

[35] `ostream` ist eine Klasse aus der C++-Standardbibliothek. Sie repräsentiert einen Kanal, in den Zeichen geschrieben werden können. Dies kann eine Datei aber auch z.B. der Bildschirm sein. cout ist ein Beispiel für einen `ostream`-Objekt.

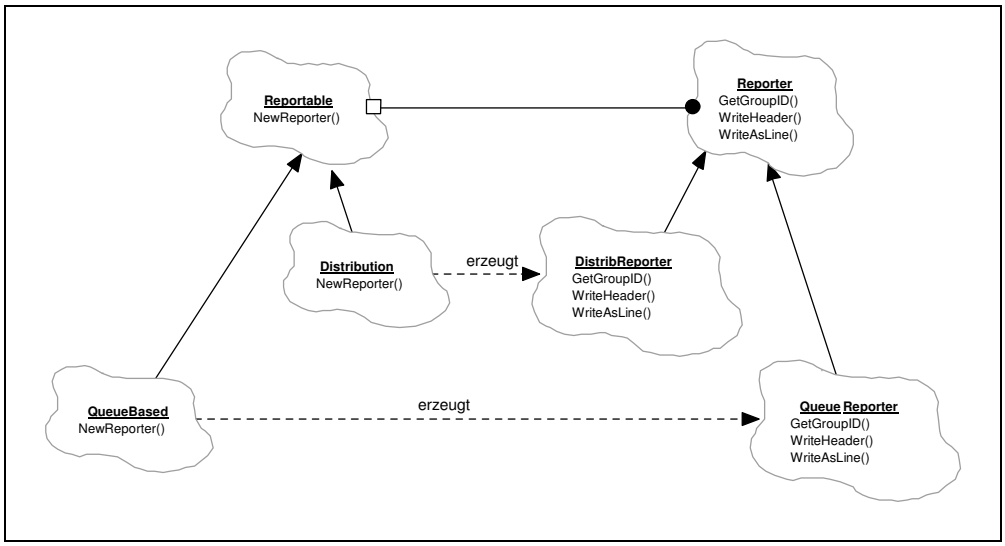

Abbildung 3-11: Zusammenhang zwischen `Reporter` und `Reportable`

Die Klasse Reporter implementiert die angeführten Methoden bereits soweit, daß die allen reportfähigen Objekten gemeinsamen Informationen (Name, Zeitpunkt der letzten Rücksetzung und Anzahl der Beobachtungen) ausgegeben werden können. Von `Reporter` können Unterklassen gebildet werden, die das Verhalten der Ausgabemethoden erweitern oder ganz neu gestalten. Damit entwickelt sich parallel zu der Hierarchie der reportfähigen Objekte eine Reporterhierarchie. Abbildung 3-11 veranschaulicht den Zusammenhang zwischen `Reporter` und `Reportable`.

3.6.2 Vom Reporter zum Report

In Abbildung 3-10 ist erkennbar, daß die Klasse `Model` ebenfalls von `Reportable` erbt. Ein Modell ist somit ebenfalls in der Lage, einen Reporter zu liefern, einen Modell-Reporter. Wird die Reportgenerierung über die Methode `Report` des Experiments initiiert, wird das Hauptmodell aufgefordert, einen Reporter zu liefern. Dieser Reporter wird einfach an den Reportkanal weitergeleitet, und landet somit bei allen Objekten, die einen Reporter empfangen wollen, um z.B. eine Report-Datei zu erzeugen. Von diesen Objekten wird der Reporter aufgefordert, über sein Modell zu berichten.

Ein Modell führt eine Liste mit allen zu ihm gehörenden reportfähigen Objekten, die sich bei ihrer Erzeugung automatisch bei ihrem Modell anmelden. Wenn das Modell einen Reporter erzeugt, übergibt es ihm diese Liste. Wird er nun aufgefordert, über das Modell zu berichten, gibt er zunächst die relevanten Daten des Modells aus und fordert anschließend jedes reportfähige Objekt in der Liste auf, einen Reporter zu liefern. Diese Reporter sortiert er nach ihrer Gruppenzugehörigkeit und sammelt sie in einer weiteren Liste, der Reporterliste. Jetzt braucht er nur noch die Reporter der Reihe nach an den Reportkanal weiterzuleiten. Von dort werden die einzelnen Reporter aufgefordert, über ihr jeweiliges Objekt zu berichten. Dabei kann unter ihnen natürlich auch wieder ein Modell-Reporter sein, der über ein Untermodell berichten soll. Auf diese Weise ergibt sich eine Gliederung nach Modellen, falls mehrere verwendet werden.

Um zu verhindern, daß zu einem Objekt Informationen im Report erscheinen, kann die Reportfähigkeit für dieses Objekt mittels `ShowInReport`, eine Methode der Klasse `Reportable`, ausgeschaltet werden. Eine andere Möglichkeit besteht darin, daß das Objekt keinen Reporter liefert. In diesem Fall erfolgt ebenfalls keine Reportausgabe für dieses

Objekt. Schaltet man die Reportfähigkeit eines Modells aus, so werden auch für alle seine Komponenten und Untermodelle keine Informationen in den Report gegeben.

3.6.3 Rücksetzung

Alle reportfähigen Objekte können über die virtuelle Methode `Reset` zurückgesetzt werden. Die von `Reportable` vordefinierte Reaktion besteht darin, den Zeitpunkt der letzten Rücksetzung auf die aktuelle Zeit und die Anzahl der Beobachtungen auf Null zu setzen. Unterklassen werden dieses Verhalten in der Regel erweitern, insbes. die statistischen Datensammelobjekte. Bei einem Modell bewirkt der Aufruf von `Reset` die Rücksetzung aller seiner reportfähigen Komponenten.

Im Zusammenhang mit `Reset` und `Report` gibt es eine Besonderheit, auf die an dieser Stelle eingegangen werden soll. Über die Methoden des Experiments, können Rücksetzung bzw. Report global (d.h. experimentweit) vorgenommen werden. In diesem Fall, soll der Report alle zum selben Zeitpunkt noch austehenden Ereignisse und Aktionen berücksichtigen. Beim `Reset` sollen die Auswirkungen dieser Aktionen noch mit zurückgesetzt werden.[36] Um dies zu realisieren, wird jeweils ein Ereignis erzeugt, das sich als Beobachter[37] bei der Simulationsuhr anmeldet. Diese benachrichtigt vor dem Stellen der Uhrzeit alle interessierten Beobachter. Durch diesen Mechanismus kann die Rücksetzung bzw. der Report auf einfache Weise zum Ende eines Simulationszeitpunktes erfolgen.

Ist diese zeitliche Verzögerung nicht erwünscht, so können die Methoden unter Angabe von `NOW` aufgerufen werden. Die Rücksetzung bzw. der Report erfolgt dann unmittelbar. Gleiches gilt für das Anhalten eines Experiments. Wird der Methode Stop ein von `NOW` verschiedener Wert übergeben, so wird das Experiment erst angehalten, wenn für diesen Zeitpunkt keine Objekte mehr vorgemerkt sind. Die Klasse `Model` bietet zusätzlich eine Methode `Report` an, die einen unmittelbaren Report nur für das entsprechende Modell einschließlich aller Untermodelle generiert.

[36] Vgl. [Bölck89], S. 132 f.

[37] Zum Beobachtermuster vgl. [Gamma95].

4 Implementierung

4.1 Besondere Aspekte von C++

Dieser Abschnitt beleuchtet einige Aspekte der Sprache C++, die mir im Zusammenhang mit den getroffenen Entwurfsentscheidungen besonders wichtig erscheinen. Die Kenntnis der Sprache C++ wird vorausgesetzt. Eine Einführung in die Sprache selbst kann in dieser Arbeit jedoch nicht geleistet werden. Hierfür sei auf [Strou92], [Meyer96], [Meyer98], [Eckel96] und [Ellis94] verwiesen.

4.1.1 Automatisch generierte Methoden

Es gibt Methoden, die der C++-Compiler automatisch erzeugt, wenn sie nicht explizit eingerichtet werden.[38] Dies sind Standard- und Kopierkonstruktor, Zuweisungs- und Adress-Operator. Damit wird, ohne daß man dies zu Gesicht bekommt, eine scheinbar leere Klassendeklaration `class Empty {};` zu:[39]

```
 1. class Empty
 2. {
 3.     public:
 4.         Empty();                  // Standardkonstruktor
 5.         Empty (const Empty rhs); // Kopierkonstruktor
 6.
 7.         Empty&                    // Zuweisungsoperator
 8.         operator=(const Empty& rhs);
 9.
10.         Empty* operator&();       // Adressoperator
11.         const Empty* operator&() const;
12. };
```

Listing 4-1: Automatisch generierte Methoden

Der Standardkonstruktor dient der Konstruktion eines Objekts ohne die Angabe jeglicher Parameter. Er ruft für jedes Element seinerseits den Standardkonstruktor auf und wird nur automatisch generiert, falls kein Konstruktor explizit eingerichtet wird. Er kann auch nicht automatisch erzeugt werden, wenn die Klasse ein Element besitzt, das keinen Standardkonstruktor aufweist, da dort mindestens ein Parameter zur Konstruktion erforderlich ist.

Der Kopierkonstruktor benötigt ein anderes Objekt der selben Klasse als Vorlage und ruft für jedes Element seinerseits den entsprechenden Kopierkonstruktor auf. Er wird immer generiert, wenn er nicht explizit angelegt wird. Das ist solange unproblematisch, solange die Klasse keine dynamischen Datenstrukturen enthält und bei der Konstruktion keine zusätzlichen Registriervorgänge erforderlich sind, wie das Inkrementieren eines Objektzählers. Hierzu ein Beispiel: Angenommen, eine Klasse deklariert einen Zeiger auf ein anderes Objekt, das im Konstruktor erzeugt und im Destruktor gelöscht wird.

---

[38] Vgl. [Meyer96], S. 210 ff.

[39] Dieses Beispiel ist [Meyer96], S. 210 entnommen

```
 1. class Any {};
 2.
 3. class Client
 4. {
 5.     public:
 6.         Client() { any = new Any; }
 7.        ~Client() { delete any;    }
 8.
 9.         Any* any;
10. };
11.
12. // ruft Standardkonstruktor auf:
13. Client* c1 = new Client;
14.
15. // ruft automatischen Kopierkonstruktor auf,
16. // wonach c2.any auf c1.any zeigt:
17. Client* c2 = new Client (*c1);
18.
19. // ruft Destruktor auf:
20. delete c1;
21.
22. // Zugriff auf bereits geloeschtes Any-Objekt:
23. c2.any;    // ???
```

Listing 4-2: Probleme mit dem automatischen Kopierkonstruktor

In Zeile 6 wird beim Aufruf des Standardkonstruktors ein neues `Any`-Objekt erzeugt, das beim Löschen im Destruktor ebenfalls gelöscht wird. In Zeile 17 wird ein zweites `Client`-Objekt als Kopie des ersten `Client`-Objekts erzeugt. Hier wird der automatisch generierte Kopierkonstruktor der Klasse `Client` aufgerufen. Dadurch wird kein neues `Any`-Objekt für `c2` erzeugt, sondern der Zeiger von `c1` kopiert, so daß `c2.any` auf das selbe Objekt zeigt wie `c1.any`. In Zeile 20 wird `c1` gelöscht, so auch durch Aufruf des Destruktors das `Any`-Objekt. `c2.any` zeigt nun auf ein bereits gelöschtes `Any`-Objekt!

Ähnliche Probleme ergeben sich, wenn sich ein Objekt bei seiner Konstruktion registrieren lassen soll, wie z.B. im Falle der Klasse `Reportable` in DESMO-C. Dies würde im Konstruktor Rumpf geschehen. Da der Rumpf beim automatischen Kopierkonstruktor jedoch stets leer ist, würde eine Registrierung des Objekts ausbleiben. Beim Zuweisungsoperator kommt zudem der Aspekt der Identität von Objekten hinzu. Im Falle von Entities würde eine Zuweisung gar keinen Sinn machen. Was sollte es bedeuten, ein Entity einem anderen zuzuweisen?

Um zu verhindern, daß Methoden automatisch generiert werden, werden sie in DESMO-C in den entsprechenden Fällen als privat deklariert, jedoch nicht definiert. Dadurch wird es in der Regel beim Versuch, sie implizit oder explizit aufzurufen eine Compiler-Meldung geben. Falls jedoch Zugriff auf private Elemente besteht, wird spätestens der Linker melden, daß für diese Methoden kein Code existiert, was dann als ein Hinweis auf eine versehentlich falsche Benutzung der Klasse verstanden werden sollte.

Durch das gezielte Verbieten von Kopierkonstruktor und Zuweisungsoperator lassen sich auch folgende Fehlerquellen ausschließen:

- In einer Schleife, in der über mehrere Objekte iteriert werden soll, muß ein Zeiger verwendet werden. Bei dem versehentlichen Einsatz einer Referenzvariablen würde nur deren Initialisierung zum gewünschten Ziel führen, die darauffolgenden Zuweisungen würden das bei der Initialisierung angegebene Objekt verändern, nicht die Referenz!

```
1. Entity& e = queue.First(); // Initialisierung von e
2. while (!e.IsNullEntity())
3.     e = queue.Next();      // Zuweisung!!!
```

- Bei der Übergabe eines Objekts mit Identität, z.B. `Entity`, als Parameter einer Funktion muß der formale Parameter als Referenz deklariert sein. Würde statt dessen direkt ein Objekt erwartet werden, würde implizit der Kopierkonstruktor aufgerufen, um ein neues temporäres Objekt zu erzeugen, das innerhalb der Funktion verwendet würde.

```
1. void f (Entity& e);     // OK
2. void g (Entity  e);     // ruft implizit
3.                         // Kopierkonstruktor auf
```

4.1.2 Downcast - ein Problem der strengen Typisierung

C++ ist eine streng typisierte Sprache, d.h. zur Übersetzungszeit werden Typüberprüfungen durchgeführt. Dabei ist eine Typumwandlung eines Zeigers auf ein Objekt in einen Zeiger auf ein Objekt einer seiner Oberklassen (Upcast) zulässig. Dieser Mechanismus ist schließlich Voraussetzung für die Nutzung des Polymorphismus. Ein Downcast ist hingegen die Umwandlung eines Zeigers auf ein Objekt vom Typ der Basisklasse in einen Zeiger auf ein Objekt vom Typ einer von der Basisklasse abgeleiteten Klasse. In der Regel sollte man solche Konstrukte vermeiden, da sie meist unübersichtlich und fehleranfällig sind. In manchen Situationen läßt sich ein Downcast jedoch nur sehr schwer vermeiden. Die besondere Schwierigkeit liegt hier z.B. darin, daß beim Entwurf eines Frameworks oder einer Klassenbibliothek, die von den angebotenen Klassen abgeleiteten Klassen nicht bekannt sind, da sie erst vom Nutzer deklariert werden.

In DESMO-C z.B. muß eine Unterklasse von der Klasse `Entity` vom Modellentwickler definiert werden, um einen neuen Entity einzuführen, z.B. die Klasse `Job`. Eine Ereignisroutine muß in der Regel auf die Attribute eines solchen Objekts zugreifen können, bekommt aber vom Simulationspaket nur ein allgemeines Objekt vom Typ `Entity` übergeben. Um innerhalb der Ereignisroutine auf die speziellen Attribute des Entities zugreifen zu können, muß dort eine Typumwandlung (Cast) von `Entity` (Oberklasse) nach `Job` (Unterklasse). Da diese Typumwandlung in der Klassenhierarchie abwärts verläuft (von Ober- nach Unterklasse), wird dies Downcast genannt. Eine Umwandlung in entgegengesetzter Richtung ist dagegen unproblematisch, da ein Objekt stets[40] als ein Objekt seiner Oberklasse behandelt werden kann (“ist ein”-Beziehung).

4.1.2.1 Einfacher Cast

Die einfachste Lösung dieses Problems bietet die Anwendung des Cast-Operators. Sei die Klasse `Special` von der Klasse `Entity` abgeleitet, dann könnte der Cast-Operator folgendermaßen angewendet werden:

```
1. class Special : public Entity { ... };
2.
3. Entity*  ptEntity  = ...;
4. Special* ptSpecial = (Special*) ptEntity;  // Downcast
5. Entity*  ptEntity2 = ptSpecial;            // Upcast
```

Listing 4-3: Einfacher Cast

[40] Eine Ausnahme ist die private Vererbung.

Dieser Cast ist jedoch leider nicht sicher, d.h. wenn `ptEntity` zur Laufzeit auf ein Objekt einer anderen von `Entity` abgeleiteten Klasse zeigt, kann die Benutzung des Zeigers nach dem Downcast unvorhersehbare Folgen haben.

4.1.2.2 RTTI (Runtime Type Information) [41]

Eine weitere Möglichkeit bietet RTTI, eine Erweiterung, auf die sich das ANSI/ISO-Komitee für die Standardisierung von C++ im Mai 1993 einigte[42]. Der Downcast würde dann mit Hilfe des `dynamic_cast`-Operators vorgenommen werden:

```
1. if (ptSpecial = <dynamic_cast> (Special*) ptEntity)
2.     // OK, ptEntity ist vom erwarteten Typ
3.   ...
4. else
5.     // kein Downcast möglich, ptSpecial ist gleich 0
```

Listing 4-4: Sicherer Downcast mit RTTI

`dynamic_cast` liefert einen Zeiger auf das erwartete Objekt, wenn eine Umwandlung möglich ist, andernfalls 0. Leider wird RTTI noch nicht von allen Compilern unterstützt.

4.1.2.3 Virtuelle Cast-Funktionen

Der `dynamic_cast`-Operator läßt sich nachahmen, indem der Basisklasse für jede ihrer Unterklassen je eine virtuelle Funktion hinzugefügt wird, die einen Zeiger auf eben diese Unterklasse liefert. Als Standardimplementierung wird 0 geliefert. Die Unterklassen überschreiben dann die ihnen entsprechende Funktion, so daß sie einen Zeiger auf sich selbst (`this`) zurückliefern:

---

41 Vgl. [Ellis94]

42 Vgl. [Meyer96]

```
 1. class Special;     // forward
 2. class Another;     // forward
 3.
 4. class Entity
 5. {
 6.     public:
 7.         virtual Special* CastToSpecial () { return 0; }
 8.         virtual Another* CastToAnother () { return 0; }
 9.     ...
10. };
11.
12. class Special : public Entity
13. {
14.     public:
15.         Special* CastToSpecial () { return this; }
16.         ...
17. };
18.
19. class Another : public Entity
20. {
21.     public:
22.         Another* CastToAnother () { return this; }
23.         ...
24. };
25.
26. ...
27.
28. if (ptSpecial = ptEntity->CastToSpecial ())
29.     // OK, ptEntity ist vom erwarteten Typ
30.     ...
31. else
32.     // kein Downcast möglich, ptSpecial ist gleich 0
```

Listing 4-5: Nachahmung von `dynamic_cast` über virtuelle Cast-Funktion

Diese Lösung erfordert jedoch, daß beim Entwurf der Basisklasse alle von ihr abgeleiteten Klassen bekannt sind, damit für alle Unterklassen die entsprechende Cast-Operation deklariert werden kann. Dies ist der Weg, der bei der Umsetzung der Beispielmodelle in SiFrame eingeschlagen wurde. würde jedoch dem Modellkonzept entgegenwirken. Die Einführung eines neuen Entitytyps würde jedoch immer eine Änderung der im System verwendeten Klasse `Entity` bedingen, was sich nachteilig auf die Verwendbarkeit von Modellen auswirkt. Gekapselte Modelle, die keine Informationen über die intern verwendeten Entitytypen preisgeben, könnten somit nicht wiederverwendet werden.

4.1.2.4 Kompromiß (Empfehlung für Systeme ohne RTTI)

Wenn man vom Modellkonzept ausgeht, kann man für jedes Modell eine eigene Klasse von `Entity` bzw. `Process` ableiten, die als Basisklasse für alle innerhalb dieses Modells verwendeten Entities dient. Diese Basisklasse wird mit den oben beschriebenen virtuellen Cast-Funktionen ausgestattet, die den `dynamic_cast`-Operator ersetzen. Da innerhalb des Modells nur Entitytypen auftreten können, die im Modell bekannt sind[43], kann ein Entity über einen einfachen Cast in das modellspezifische Basis-Entity umgewandelt werden. Einmal in ein Basis-Entity umgewandelt, können nun die virtuellen Cast-Funktionen aufgerufen werden, um “sicher” in den speziellen Entitytyp umzuwandeln.

Problematisch wird dieser Ansatz jedoch bei gemischten Modellen, in denen sowohl Entities (ereignisorientiert) als auch Prozesse (prozeßorientiert) existieren, da die Unterklassen von `Entity` und `Process` eine gemeinsame Oberklasse, die der Modell-Entities, benö-

[43] Dies wird in DESMO-C durch das Konzept der Modellintegrität gewähleistet (s. Abschnitt 3.2.3).

tigten. Eine Lösung dieses Problems ist die Realisierung der virtuellen Cast-Funktionen als Mixin-Klasse. Jede Klasse, die einen neuen Entity- bzw. Prozeßtyp für das Modell einführt, erbt zusätzlich von dieser Mixin-Klasse, so daß ein Entity dieses Modells stets über einen einfachen Cast in ein Objekt der Mixin-Klasse konvertiert werden kann und so die Cast-Funktionen zur Verfügung stehen. Abbildung 4-1 zeigt dieses Schema.

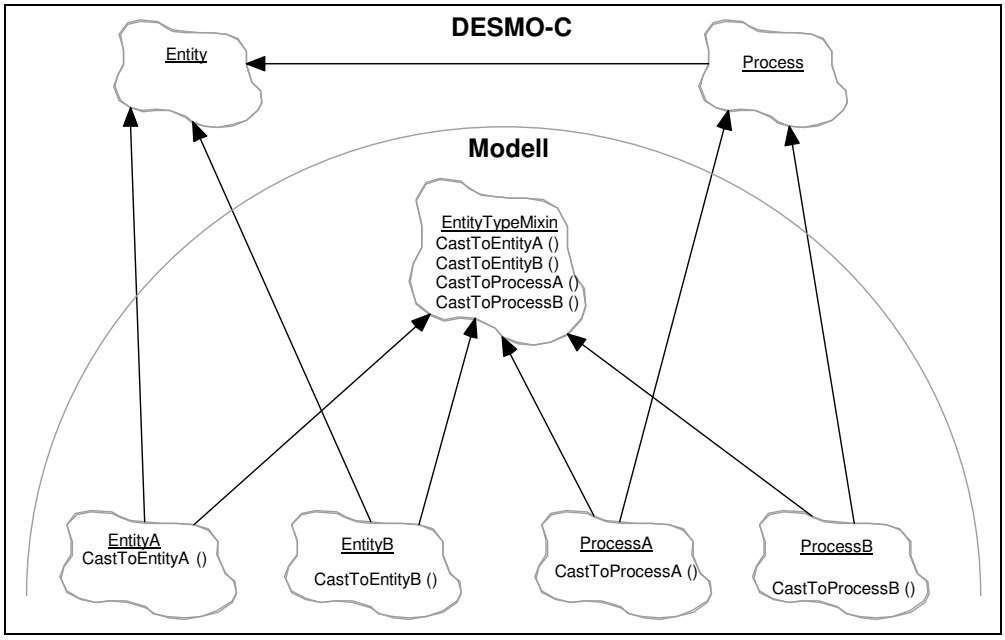

Abbildung 4-1: Typumwandlung über Mixin-Klasse

4.1.3 Templates

Bei der Verwendung von Templates kann es Probleme beim Linken geben, wenn Implementierung und Deklaration auf zwei Dateien verteilt werden, wie dies sonst üblich ist. Bindet der Klient der Template-Klasse lediglich die Header-Datei ein, so findet der Linker die Implementierung nicht. Die beste Lösung scheint darin zu bestehen, die Implementierung in die Header-Datei einzubinden. Dies führt zu einem Include-Zyklus zwischen Header- und Implementierungsdatei, der durch Abfrage und Definition von Symbolen verhindert werden muß:

```
 1. // Datei: x.h:
 2. #ifndef X_H
 3. #define X_H
 4.
 5. template <class T> class X {...};
 6.
 7. #include "x.cc"    // Implementierung
 8.
 9. #endif    // X_H
10.
11.
12. // Datei: x.cc:
13. #ifndef X_CC
14. #define X_CC
15.
16. #include "x.h"
17.
18. ...    // Implementierung
19.
20. #endif // X_CC
```

Listing 4-6: .h- und .cc-Datei einer Template-Klasse

Dabei ist zu beachten, daß eine Änderung der Implementierung schon dazu führt, daß alle Klienten neu übersetzt werden müssen. Außerdem wird die Kapselung der Implementierung aufgebrochen. Dies läßt sich jedoch nicht vermeiden, weshalb in DESMO-C Templates nur intern verwendet werden. Anstatt die Template-Klasse direkt zu verwenden, wird ein Zeiger auf eine von der Template-Klasse abgeleiteten Klasse deklariert. Die genaue Deklaration dieser Klasse muß dafür nicht bekannt sein. Eine einfache "Forward"-Deklaration genügt. Ihre Definition findet dann in der Implementierungsdatei des Klienten statt:

```
 1. // Datei: xclient.h:
 2. class Xint;    // forward für Unterklasse von X<int>
 3.
 4. class XClient
 5. {
 6.     public:
 7.         ...
 8.     private:
 9.         Xint* xInt;
10. };
11.
12.
13. // Datei: xclient.cc:
14. #include "xclient.h"
15. #include "x.h"
16.
17. class Xint : public X<int> { ... };
18.
19. ...
20.
```

Listing 4-7: Aufschieben der Template-Klassendefinition

Damit ist nur noch `xclient.cc` von `x.cc` abhängig, nicht jedoch Dateien, die nur `xclient.h` einbinden. Ein Nachteil mag sein, daß in `xclient.h` nur Zeiger verwendet werden können, da andernfalls die Definition der Zwischenklasse, also auch die des Templates bekannt sein müßten. Ein Beispiel dafür ist die einzige in DESMO-C verwendete Template-Klasse `Ring` (vgl. Abschnitt 4.4.2). Diese Technik wird in DESMO-C außerdem für Nicht-Template-Klassen verwendet, um die Schnittstelle rein interner Strukturen nicht preisgeben zu müssen.

```
 1. #include "entity.h"
 2. // #include "coroutin.h" kann entfallen
 3.
 4. class Coroutine;    // forward
 5.
 6. class Process : public Entity
 7. {
 8.     ...
 9.     private:
10.         Coroutine* coroutine;
11.         ...
12. );
```

Listing 4-8: Aufschieben einer internen Klassendefinition

4.2 Elemente der Simulationssteuerung

4.2.1 Ereignisliste

Die Ereignisliste ist die zentrale Datenstruktur in einem ereignisgesteuerten Simulator wie DESMO-C. In ihr werden Ereignisnotizen verwaltet, in denen festgehalten wird, welche Objekte wann und wie aktiviert werden. Diese Objekte werden in DESMO auf der Ebene des Simulationskerns als `Item` bezeichnet und entsprechen den gemeinsamen Teilen von ereignis- und prozeßorientierten Entities.

4.2.1.1 Die abstrakte Ereignisliste

Bei der Implementierung der Ereignisliste muß es sich keinesfalls um eine Liste handeln. Es können z.B. auch Baumstrukturen zum Einsatz kommen, wie bei DESMO. Dort sind alle die Ereignisnotiz betreffenden Informationen im `Item`[44] selbst gespeichert. Dies sind der Ereigniszeitpunkt und die für die Verkettung notwendigen Zeiger, um die Ereignislistenelemente miteinander zu einer Baumstruktur zu verknüpfen. Es werden somit drei Zeiger `Left`, `Right` und `Back` zur Vorwärts- und Rückwärtsverkettung benötigt.

Durch die Speicherung der Verkettungszeiger im `Item` selbst betrifft eine Änderung der zugrundeliegenden Ereignislistenimplementierung immer auch die Datenstruktur `Item`. In DESMO-C wird diese enge Kopplung vermieden, damit die Ereignislistenstruktur unabhängig weiterentwickelt werden kann. So ist in DESMO-C die Ereignisliste auf einer abstrakten Ebene mit Hilfe von zwei Klassen realisiert: `EventNote`, die Klasse der Ereignisnotizen, und `EventList`, die Klasse für die Ereignisliste, die Ereignisnotizen verwaltet. Dabei dienen Objekte der Klasse `EventNote` zur Aufnahme des Aktivierungszeitpunktes und der vorgemerkten Objekte. Dies sind maximal ein Ereignis und ein Entity. Daraus ergeben sich die in Tabelle 4-1 wiedergegebenen Kombinationen.

Bedeutung	Ereignis	Entity
kommt nicht vor	-	-
externes Ereignis	X	-
"normales" Ereignis	X	X
Prozeßaktivierung	-	X

Tabelle 4-1: Mögliche Kombinationen in einer Ereignisnotiz

[44] Das `Item` stellt in DESMO die interne Sicht auf ein Entity dar.

Die Klasse `EventNote` definiert zwei Konstruktoren und alle Vergleichsoperatoren, um eine sortierte Verwaltung zu ermöglichen. Für den Vergleich wird ausschließlich der Aktivierungszeitpunkt herangezogen. Die Klasse `EventList` ist eine rein abstrakte Klasse. Alle Methoden sind virtuell und müssen in den Unterklassen implementiert werden. Eine Ereignisliste muß auf Verlangen eine neue Ereignisnotiz erzeugen können, da nur sie selbst den zu ihr passenden konkreten Typ der Ereignisnotizen kennt. Hierfür müssen die Unterklassen von `EventList` die zweifach überladene Fabrikmethode[45] `NewEventNote` definieren, die entweder aus einer bestehenden Ereignisnotiz oder aus Zeitpunkt, Ereignis und Entity eine neue Ereignisnotiz erzeugt. Der Typ des Rückgabewertes ist `EventNote`, denn die Simulationssteuerung von DESMO-C arbeitet ausschließlich auf der abstrakten Ebene von `EventList` und `EventNote`, durch deren Schnittstelle die benötigte Funktionalität gegeben ist.

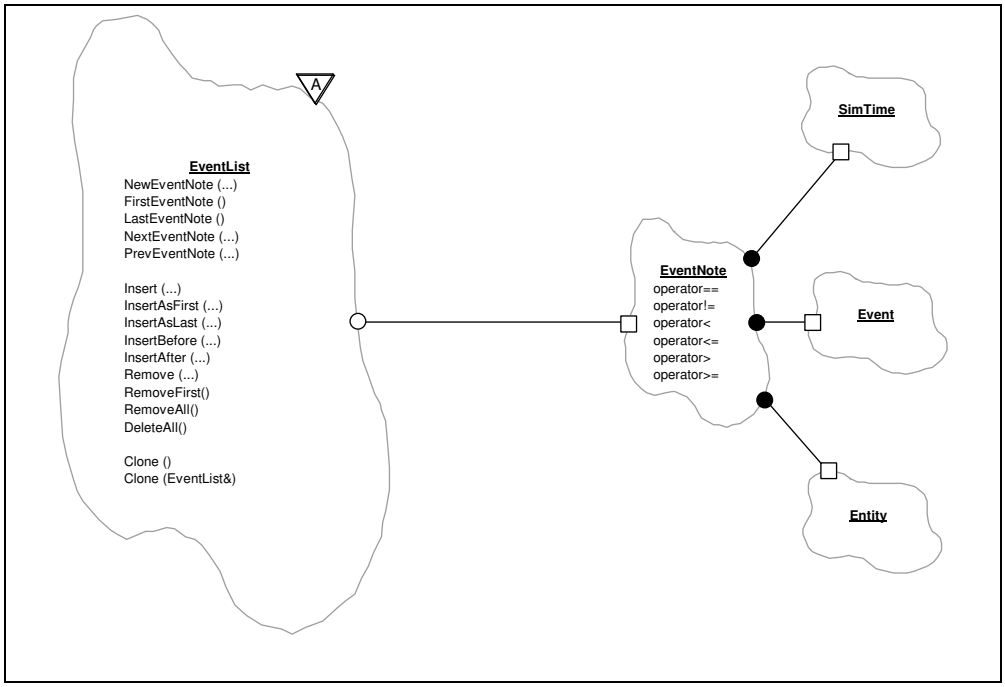

Abbildung 4-2: Die Klassen `EventList` und `EventNote`

Die Klasse `EventList` bietet Zugriff auf Ereignisnotizen über die Methoden `FirstEventNote`, `LastEventNote` für die erste bzw. letzte Ereignisnotiz, `NextEventNote` und `PrevEventNote` für den Nachfolger bzw. Vorgänger zu einer gegebene Ereignisnotiz. Alle vier Methoden liefern einen Zeiger auf die entsprechende Ereignisnotiz bzw. Null, wenn keines geliefert werden kann. Zum Einfügen von Ereignisnotizen stehen die Methoden `Insert`, `InsertAsFirst`, `InsertAsLast`, `InsertAfter` und `InsertBefore` zur Verfügung. Entfernt werden können sie mit `Remove`, `RemoveFirst` zum Entfernen einzelner und `DeleteAll` zum Löschen aller Ereignisnotizen.

Die gewählte Lösung zur Realisierung der Ereignisliste bietet darüber hinaus sogar die Möglichkeit, während eines Simulationslaufes die Datenstruktur der Ereignisliste und die damit verbundenen Algorithmen austauschen zu können. Damit ist ein Verfahren denkbar, das die Zugriffe auf die Ereignisliste während der Simulation analysiert, um den für das jeweilige Zugriffsverhalten günstigsten Algorithmus auszuwählen.

[45] Zum Entwurfsmuster der Fabrikmethode vgl. [Gamma95], S. 115 ff.

Mit Hilfe der Methode `Clone`, mit der das Prototypmuster[46] zur Anwendung kommt, ist es Klienten ermöglich, anhand eines Prototyps eine neue Ereignisliste der selben Art wie der Prototyp zu erzeugen. Dabei wird der Prototyp aufgefordert, sich selbst zu klonen. Ohne Parameter kann `Clone` aufgerufen werden, um eine neue leere Ereignisliste zu erzeugen. Als Parameter kann `Clone` auch eine andere Ereignisliste übergeben werden, was dazu führt, daß alle Ereignisnotizen kopiert und in der selben Reihenfolge in die neue Ereignisliste einsortiert werden. Über die Methode `Clone` können Klienten eine Ereignisliste auf einfache Weise durch eine andere ersetzen (s. Abschnitt 4.2.2).

4.2.1.2 Lineare Liste

In DESMO-C ist die Ereignisliste prototypisch als doppelt verkettete lineare Liste implementiert. Hierfür wird von `EventNote` die Unterklasse `LinearEventNote` gebildet, die zwei Zeiger `prev` und `next` auf Vorgänger bzw. Nachfolger in der Liste enthält. Objekte der Klasse `LinearEventNote` werden in einem Objekt der von `EventList` abgeleiteten Unterklasse `LinearEventList` verwaltet. Um Zugriff auf die Verkettungszeiger zu haben, ist sie als Freundklasse von `LinearEventNote` deklariert. Abbildung 4-3 zeigt die neuen Klassen samt ihrer Oberklassen.

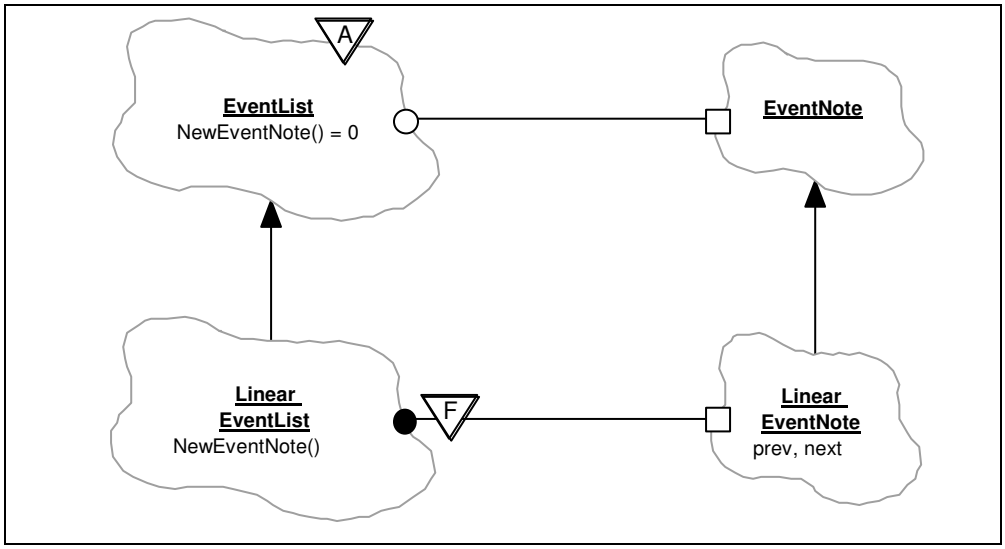

Abbildung 4-3: Die lineare Ereignisliste

Hat ein Klient erst einmal eine Ereignisliste, so ist es für ihn nicht von Bedeutung, von welchem Typ sie ist. Da die Methode `NewEventNote` virtuell ist und in den Unterklassen definiert wird, liefert sie genau eine solche Ereignisnotiz, die mit der Ereignisliste verträglich ist.

4.2.2 Scheduler und Simulationsuhr

In Abschnitt 4.2.1 ist die Rede vom “Klienten” der Ereignisliste. In DESMO-C ist dies der Scheduler (Klasse `Scheduler`). Er enthält eine Ereignisliste, in der er die vorgemerkten Objekte (`Schedulable`) verwaltet. Über ihn laufen alle Aktionen zur Manipulation der Ereignisliste. Außerdem kann er Auskunft über aktuelle Zeit, aktuelles Ereignis, aktuelles Entity bzw. Prozeß sowie aktuelles Modell geben. Zur Erzeugung eines Schedulers kann ein Prototyp einer Ereignisliste übergeben werden, anhand derer sich der Scheduler seine

[46] Vgl. [Gamma95], S. 127 ff.

eigene Ereignisliste klont, auf die nur er Zugriff hat. Wird kein Prototyp übergeben, so legt er eine lineare Ereignisliste an. Die Beziehung zwischen Scheduler und Ereignisliste ist in Abbildung 4-4 wiedergegeben.

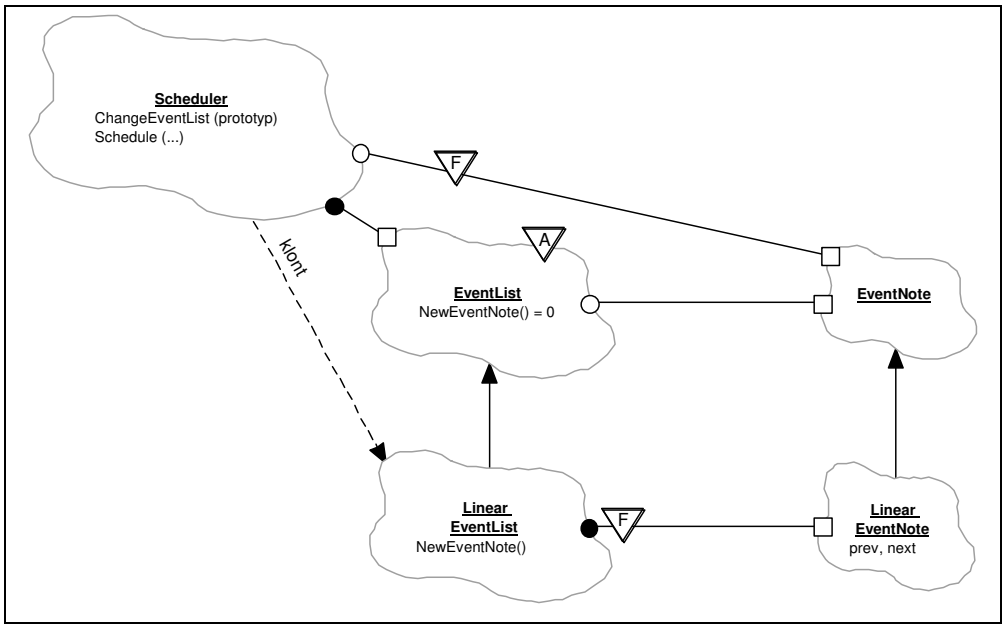

Abbildung 4-4: Beziehung zwischen `Scheduler` und `EventList`

Um die Ereignisliste auszutauschen, kann die Methode `ChangeEventList` des Schedulers aufgerufen werden. Dies kann zu jeder Zeit erfolgen. Als Parameter muß ein Prototyp einer Ereignisliste übergeben werden. Von diesem Prototyp klont sich der Scheduler eine neue Ereignisliste, die dadurch den selben Typ erhält, wie der Prototyp. Durch diese Technik ist der Scheduler in der Lage, sich selbst eine Ereignisliste zu erzeugen, auf die er allein Zugriff hat. Der Methode `Clone` übergibt er die alte Ereignisliste, so daß die geklonte Liste inhaltlich mit der alten Liste übereinstimmt, die nun gelöscht werden kann. Nach dem Aufruf von `ChangeEventList` wird der Prototyp nicht mehr benötigt.

Um ein Objekt (`Schedulable`) vorzumerken, berechnet der Scheduler zunächst den absoluten Simulationszeitpunkt und läßt sich von der Ereignisliste eine neue Ereignisnotiz erzeugen, die mit den vorzumerkenden Objekten und dem Simulationszeitpunkt initialisiert wird. Die so neu erzeugte Ereignisnotiz fügt er über die Methoden der Ereignisliste in diese ein. Diese verwendet die auf Ereignisnotizen definierten Vergleichsoperatoren, um die Notiz an der korrekten Stelle einzusortieren. Bei einer plazierten Einfügung über `InsertBefore, InsertAfter` oder `InsertAsFirst` muß der Scheduler gewährleisten, daß der Zeitpunkt der einzufügenden Ereignisnotiz konsistent ist.

Die Methoden des Schedulers zur Ereignislistenmanipulation `Schedule`, `ScheduleBefore`, `ScheduleAfter`, `ReSchedule, Passivate` und `Cancel` erwarten stets eine Reihe von Vorbedingungen, deren Nichtbeachtung auf höheren Ebenen abzufangen ist (z.B. in der Klasse `Process`). Im einzelnen sind dies:

- Objekte, die vorgemerkt werden sollen, dürfen nicht auf der Ereignisliste stehen. Die aktuellen Objekte stehen nicht auf der Ereignisliste.

- Das Zeitintervall `dt` darf nicht negativ sein.

- Die bei `ScheduleBefore` und `ScheduleAfter` zur Plazierung angegebenen Objekte müssen auf der Ereignisliste stehen. Bei `ScheduleBefore` darf ein aktuelles Objekt nur eingesetzt werden, wenn es sich um einen Prozeß handelt, da nur in diesem Fall eine Verdrängung erlaubt ist.

- Ein `ReSchedule` bzw. `Cancel` übergebenes Objekt muß auf der Ereignisliste stehen.

- Bei den Methoden `Schedule`, `ScheduleBefore` und `ScheduleAfter` werden je ein Ereignis und ein Entity übergeben. Wird als Ereignis `NullEvent` übergeben, so muß das Entity ein Prozeß sein. Andernfalls werden Ereignis und Entity im ereignisorientierten Sinn vorgemerkt. `NullEvent` und `NullEntity` dürfen auch nicht zusammen übergeben werden (Zeile 1 in Tabelle 4-1).

- `Passivate` verlangt, daß der übergebene Prozeß der aktuelle ist.

Zum Zugriff auf den Nachfolger bzw. Vorgänger stehen die Methoden `Next`, `NextEvent`, `NextEntity` und `NextProcess` bzw. `Prev`, `PrevEvent`, `PrevEntity` und `PrevProcess` zur Verfügung, wobei `Next` und `Prev` ein Objekt vom Typ `Schedulable` liefern. Dabei wird zunächst versucht, ein Entity und, falls dies nicht möglich, ist ein Ereignis zurückzugeben. Ist beides nicht möglich, wird `NullEntity` geliefert. Die aktuellen Objekte werden jeweils mit einbezogen.

Um zu einem vorgemerkten Objekt die entsprechende Ereignisnotiz schnell auffinden zu können, enthalten Objekte der Klasse `Schedulable` einen Verweis auf die entsprechende Ereignisnotiz. Dadurch können die Operationen zum Entfernen von der Ereignisliste, zum plazierten Einfügen und zum Ermitteln des Vorgängers oder Nachfolgers sehr effizient implementiert werden. Allein der Scheduler ist für diese Verbindung zuständig, weshalb er Freund der Klasse `Schedulable` ist (vgl. Abbildung 4-5). Durch diese Verbindung ergibt sich die schon in DESMO bestehende Einschränkung, daß ein Objekt nicht mehrmals auf der Ereignisliste stehen darf. Diese Einschränkung gilt ebenso für Ereignisse, wodurch es möglich ist, zu einem eingetragenen Ereignis das Folgeereingis zu ermitteln.

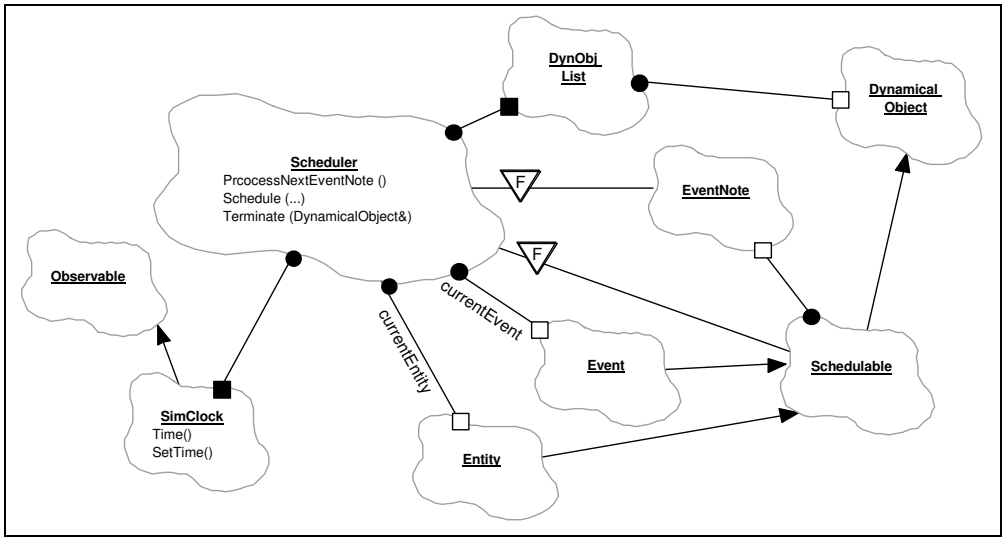

Abbildung 4-5: Der Scheduler und seine Beziehungen zu anderen Klassen

Für die Zeitführung enthält der Scheduler eine Simulationsuhr (Klasse `SimClock`). Wann immer er die aktuelle Zeit stellen möchte, ruft er die Methode `SetTime` der Simulations-

uhr auf. Diese überprüft zunächst, ob die neue Zeit zulässig ist und einen Zeitfortschritt von mehr als Epsilon[47] bedeuten würde. Ist dies der Fall, werden über den Mechanismus des Beobachtermusters[48] alle angemeldeten Beobachter benachrichtigt. Erst dann wird die Uhr tatsächlich gestellt. Andernfalls wird der Aufruf ignoriert. Alle Anfragen nach der aktuellen Simulationszeit reicht der Scheduler weiter an die Simulationsuhr.

Wann immer ein dynamisches Objekt (Klasse `DynamicalObject`) terminiert, wird es der Methode `Terminate` des Schedulers übergeben, sofern das Löschen bei Terminierung für dieses Objekt eingestellt ist (`DeleteOnTermination`). `Terminate` trägt das Objekt in eine Liste ein (Klasse `DynObjList`), um das Löschen auf einen Zeitpunkt zu verschieben, zu dem das Objekt garantiert nicht eines der aktuellen Objekte ist (`currentEvent` bzw. `currentEntity`). Sind die neuen aktuellen Objekte bestimmt, kann die Liste abgearbeitet werden, um alle in ihr enthaltenen Objekte zu löschen.

Das Herzstück des Schedulers ist die Methode `ProcessNextEventNote`, die genau eine Ereignisnotiz aus der Ereignisliste entnimmt und verarbeitet. Dies gewährleistet, daß die Ereignisliste stets in einem konsistenten Zustand ist[49]. Die Objekte der Ereignisnotiz werden zu den aktuellen gemacht und die Notiz von der Ereignisliste entfernt. Hier ergibt sich nun eine der drei letzten Situationen aus Tabelle 4-1, die auf zwei Fälle reduziert werden können:

1. Ist eine Ereignisbehandlung angezeigt, wird die Ereignisroutine `EventRoutine` des Ereignisses mit dem aktuellen Entity aufgerufen. Bei einem externen Ereignis wird der Aufruf automatisch an die Methode `ExternalEventRoutine` weitergeleitet. Nach Ende der Ereignisroutine wird das Ereignis ggf. mittels `Terminate` für seine Vernichtung vorgesehen.

2. Falls die Ereignisnotiz kein Ereignis enthielt, muß das aktuelle Entity ein Prozeß sein, dessen Koroutine die Kontrolle sodann übertragen wird. Der Scheduler ist so konzipiert, daß die Methode `ProcessNextEventNote` stets unter Kontrolle der Hauptkoroutine abläuft, also auch jegliche Ereignisroutinen. Zur Aktivierung eines Prozesses wird dessen Koroutine die Kontrolle transferiert. Wird dieser passiviert, so geht die Kontrolle in jedem Fall zunächst an die Hauptkoroutine über, um wieder im Scheduler zu landen.

Um einen kompletten Simulationslauf in Gang zu halten, muß die Methode `ProcessNextEventNote` nur in eine Schleife eingebettet werden, die zu einem geeigneten Zeitpunkt abbricht, spätestens jedoch wenn `ProcessNextEventNote false` liefert, was bedeutet, daß die Ereignisliste leer ist. Durch diese Art der Implementierung läßt sich der Scheduler auch im Einzelschrittmodus betreiben, d.h. daß nach jeder Verarbeitung einer Ereignisnotiz die Möglichkeit besteht, andere Aufgaben zu erledigen. Dies können auf der einen Seite bestimmte Aufrufe zur Erfüllung von Anforderungen an Programme in auf kooperativem Multitasking basierenden Systemen sein[50]. Auf der anderen Seite kann diese

47 Epsilon ist die kleinste Zeiteinheit, die ein Stellen der Simulationsuhr bewirkt.

48 SimClock erbt von Observable. Dadurch können sich Beobachter-Objekte (Unterklassen von Observer) bei der Simualtionsuhr anmelden, um über eine Änderung der Zeit informiert zu werden. Zum Beobachtermuster vgl. [Gamma95], S. 257 ff.

49 Dies ist bei DESMO ein Problem, wenn z.B. versucht wird, innerhalb einer Zeitreihenfunktion auf die Ereignisliste zuzugreifen.

50 In [Schni96] wurde DESMO für Apple MacOS so angepaßt, daß es sich in die Umgebung des kooperativen Multitaskingsystems einfügte, und somit z.B. auch Simulationsläufe im Hintergrund möglich sind. Dies geschah jedoch mit erheblichem Aufwand im Vergleich zu den Möglichkeiten von DESMO-C.

Fähigkeit aber auch dazu genutzt werden, um mehrere Experimente parallel durchzuführen, indem eine Reihe von Schedulern nacheinander die Gelegenheit bekommen, ihre jeweils nächste Ereignisnotiz zu verarbeiten.

Der Scheduler selbst ist nur ein Element der Simulationssteuerung. Was fehlt, ist eine Instanz, die den Scheduler steuert. Diese Instanz ist in DESMO-C der Experimentmanager, der Gegenstand von Abschnitt 4.3.1 ist.

4.2.3 Das Nachrichtensystem

DESMO organisiert die Daten, die ein Simulationslauf produziert, in vier Dateien: Debug-, Error-, Report- und Fehlerdatei. Zur Verteilung dieser Daten müssen Informationen zwischen den Komponenten des Simulationspakets ausgetauscht werden. Um den dabei entstehenden Informationsfluß zu organisieren, ist in DESMO-C ein Nachrichtensystem eingerichtet, das darauf ausgelegt ist, Nachrichten unterschiedlicher Typen zu verarbeiten. Ganz allgemein unterliegt diesem System das in Abbildung 4-6 dargestellte Schema.

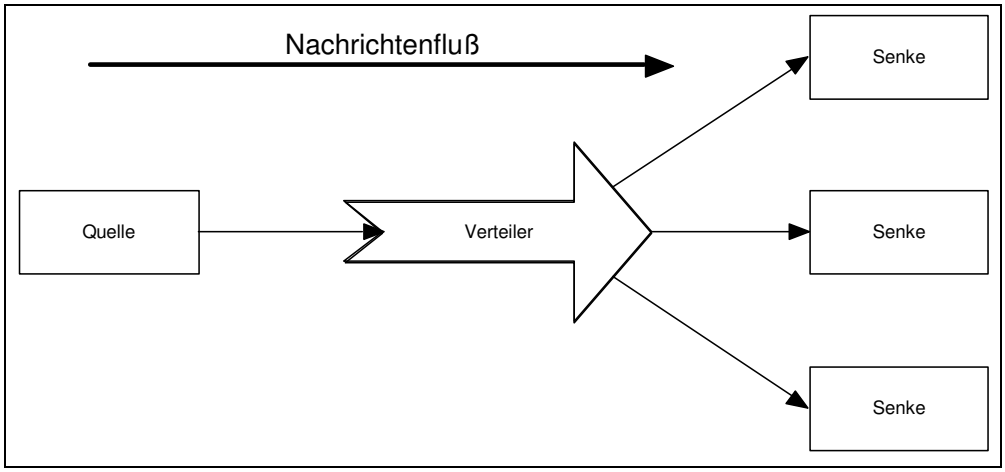

Abbildung 4-6: Schema des Nachrichtensystems in DESMO-C

Eine Nachricht wird an der Stelle an der sie entsteht (Quelle) an eine zentrale Instanz (Verteiler) geschickt, die die Nachricht an entsprechende Empfänger (Senke) weiterleitet. So kann die Nachricht an mehrere Stellen gelangen, wo sie verarbeitet wird. Dabei braucht der Erzeuger sich nur um die Übermittlung zum Verteiler zu kümmern.

4.2.3.1 Nachrichten

In DESMO-C ist eine Nachricht eine Unterklasse der Klasse `Message`. Ein Objekt dieser Klasse ist in der Lage, aktuelle Zeit, Ereignis, Entity, Modell, und Experiment zu ermitteln und liefert über die Methode `Description` eine textuelle Beschreibung der Nachricht. Über die Methode Type kann der Nachrichtentyp ermittelt werden, der einem aus der folgenden Aufzählung entspricht:

1. `trace`: Meldungen für die Ablaufverfolgung, die in der Regel in der Datei mit der Endung ‘.trc’ erscheinen, sofern der Trace-Modus eingeschaltet ist.

2. `report`: Meldungen für den Report, die in der Regel in der Datei mit der Endung ‘.rpt’ erscheinen.

3. `debug`: Meldungen für die Debug-Hilfe, die in der Regel in der Datei mit der Endung '.dbg' erscheinen, sofern die Debug-Funktion eingeschaltet ist.

4. `error`: Warnungen und Fehlermeldungen, die in der Regel in der Datei mit der Endung '.err' erscheinen.

5. `globalError`: Warnungen und Fehlermeldungen, die nicht einem bestimmten Experiment zuzuordnen sondern globaler übergreifender Natur sind. Sie erscheinen in der Regel auf dem Bildschirm.

Falls eine textuelle Beschreibung nicht ausreicht, kann eine Nachricht zusätzlich mit einem Code versehen werden, auf den in einer Senke über die Methode `Code` zugegriffen werden kann, um auf bestimmte Nachrichten in besonderer Weise reagieren zu können. Z.B. sind für die Nachrichten des Typs `error` bzw. `globalError` folgende Codes vorgesehen:

- `warning`: für Warnungen, deren Ursache keinen Experimentabbruch erfordern

- `error`: für Fehlermeldungen, bei denen ein Fortsetzen des Experiments nicht sinnvoll ist

- `fatalError`: Fehlermeldungen, die so gravierend sind, daß das Programm beendet werden sollte.

Für Fehlermeldungen sind auch die folgenden virtuellen Methoden vorgesehen, die in Unterklassen definiert werden müssen, um einen entsprechenden Text zu liefern:

- `Location`: sollte die Stelle im Programm liefern, an der der Fehler entdeckt wurde, z.B. `"Process::Hold"`

- `Consequences`: sollte die Reaktion auf den Fehler beschreiben

- `Hint`: kann einen Hinweis geben, wie der Fehler zu vermeiden ist

Nachrichten werden grundsätzlich als temporäre Objekte betrachtet. D.h. eine Nachricht ist nur für die Dauer des Aufrufs der Methode gültig, der sie übergeben wurde. Weder Verteiler noch Senken sollten eine Nachricht speichern. Sollte dies erforderlich sein, so muß eine Kopie angelegt werden, da das Original in der Regel nach der Verarbeitung wieder gelöscht wird. Einen weiteren Typ von Nachricht stellen die in Abschnitt 3.6 beschriebenen Reporter dar. Diese können ebenfalls an den Nachrichtenverteiler gesendet werden.

4.2.3.2 Nachrichtenquellen

Die meisten Nachrichten werden innerhalb von Modellkomponenten erzeugt. Die Klasse `ModelComponent` bietet zum Versenden von Nachrichten die Methode `SendMessage` an, die die Nachricht an den entsprechenden Nachrichtenverteiler weiterleitet. Die Methode `TraceNote` bietet eine Abkürzung, eine Meldung in den Trace zu schreiben, ohne eine neue Nachrichtenklasse definieren zu müssen. `TraceNote` konstruiert erst eine Trace-Nachricht, die dann per `SendMessage` verschickt wird. Allerdings wird der Versand unterbunden, falls entweder der Trace-Modus ausgeschaltet ist oder für die Modellkomponente eingestellt ist, daß sie nicht im Trace erscheinen soll.

Eine Abkürzung wie für Trace-Nachrichten ist auch für die drei Fehlertypen vorgesehen. Die Methoden lauten `Warning`, `Error` und `FatalError`. Sie erhalten jeweils vier

`String`-Parameter für Beschreibung, Methode, in der der Fehler aufgetreten ist, Reaktion auf den Fehler und Hinweis zu dessen Vermeidung. Damit ist es auch vom Modellentwickler erstellten Modellkomponenten möglich, auf einfache Weise sowohl Trace- als auch Fehler- bzw. Warnmeldungen auszugeben.

4.2.3.3 Nachrichtenverteiler

Für den Empfang von Nachrichten ist in DESMO-C zunächst die Klasse `MessageReceiver` eingerichtet. Sie definiert die Schnittstelle, die jeder Nachrichtenempfänger aufweisen muß. Die beiden Methoden `Note` und `TakeReporter` zum Empfangen einer Nachricht bzw. eines Reporters sind virtuell und können in Unterklassen definiert werden, um eine Reaktion auf Nachricht und/oder Reporter zu implementieren. Als Vorgabe ist ein leerer Methodenrumpf vorgesehen.

Eine Unterklasse von `MessageReceiver` ist die Klasse der Nachrichtenverteiler (`MessageDistributor`). Sie führt für jeden Nachrichtentyp eine Liste von Empfängern und stellt Methoden zur Verfügung, über die sich Nachrichtenempfänger registrieren lassen können, um in eine Verteilerliste aufgenommen zu werden (s. Abbildung 4-7). Bei der Registrierung ist der Nachrichtentyp mit anzugeben. Auf diese Weise können sich Empfänger auch für mehr als einen Nachrichtentyp registrieren lassen. Über `DeRegister` können sich Nachrichtenempfänger gezielt in Bezug auf einen bestimmten Nachrichtentyp oder für alle Nachrichtentypen auf einmal abmelden.

Die Methoden `Note` und `TakeReporter` sind in `MessageDistributor` so implementiert, daß die entsprechende `Distribute`-Methode für die Verteilung aufgerufen wird. Diese fragt über die `Message`-Methode `Type` den Typ der Nachricht ab, um die entsprechende Verteilerliste auszuwählen.

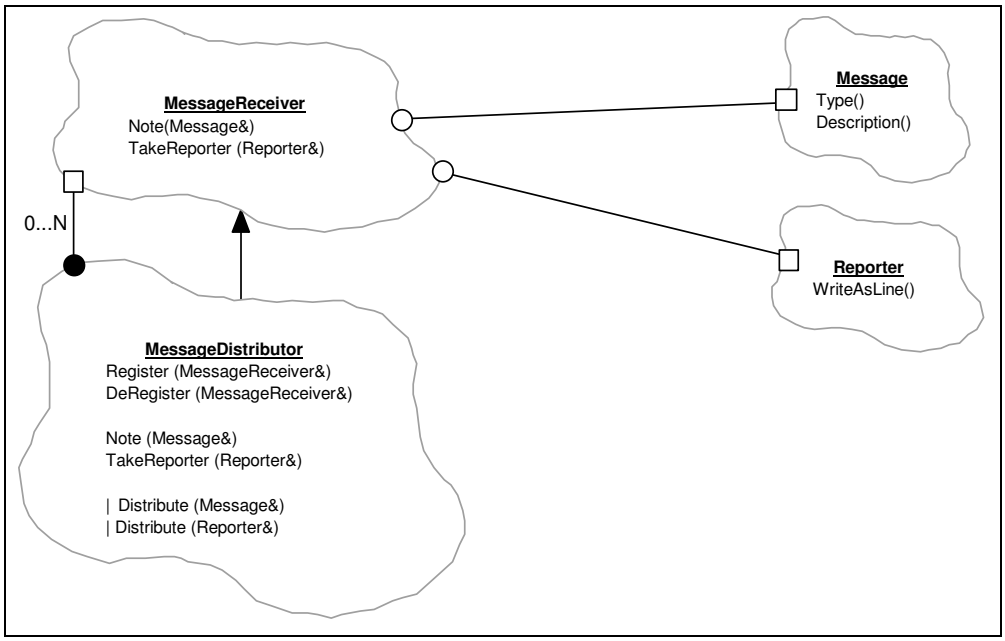

Abbildung 4-7: Die Klassen `MessageReceiver` und `MessageDistributor`

Die “zentrale Instanz”, an die jegliche Nachrichten von einer Quelle zunächst gesendet werden, ist der `MessageManager`, eine Unterklasse von `MessageDistributor`. Jedes Experiment verfügt über genau einen solchen “obersten” Nachrichtenverteiler. Er verwaltet für jeden Nachrichtentyp einen Kanal der über ihn zentral an- und abschaltbar ist

(Methoden `SwitchOn` und `SwitchOff`). Nur in angeschaltete Kanäle werden Nachrichten weitergeleitet. Nach jedem An- und vor jedem Ausschalten sendet der `MessageManager` eine Nachricht mit dem Code `switchOn` bzw. `switchOff` in den entsprechenden Kanal, um den Empfängern eine entsprechende Reaktion zu ermöglichen. Zu jedem Kanal zählt er die weitergeleiteten Nachrichten. Dieser Zähler ist über `GetCount` abfragbar. Außerdem kann mittels `Skip` die Weitergabe einer bestimmten Anzahl von Nachrichten unterbunden werden. Die Kanäle für Fehler- und Report-Nachrichten sind zu Beginn angeschaltet, alle anderen sind abgeschaltet.

Für die Kanäle `debug`, `error`, `report` und `trace` sind weitere Verteiler eingerichtet, die `OutputManager`, die als Bindeglied zwischen `MessageManager` und den entsprechenden Nachrichtensenken (Klasse `Output`) fungieren. Bei der Erzeugung wird einem `OutputManager` der Nachrichtentyp übergeben, für den er zuständig ist und durch den er sich von den anderen `OutputManagern` unterscheidet. Der Vorteil in der Einführung dieser Zwischenschicht besteht darin, daß sich ein `OutputManager` für mehrere Nachrichtentypen beim `MessageManager` anmelden kann. Dadurch ist es möglich, daß z.B. Fehlermeldungen grundsätzlich auch im Trace ausgegeben werden können. Empfängt ein `OutputManager` eine Nachricht mit dem Code `switchOn` bzw. `switchOff`, so kann er darauf reagieren, indem er sich für zusätzliche Nachrichtentypen beim `MessageManager` an- bzw. abmeldet. Für seinen eigenen Nachrichtentyp bleibt er stets angemeldet. Bei ausgeschaltetem Kanal verhindert der `MessageManager` bereits eine Verbreitung der Nachrichten. Durch diese zusätzlichen Abmeldungen wird verhindert, daß z.B. bei ausgeschaltetem Trace-Modus Fehlermeldungen trotzdem in den Trace gelangen, weil der für Trace-Meldungen zuständige `OutputManager` noch für Fehlernachrichten angemeldet ist. Abbildung 4-8 zeigt die Beziehung zwischen `OutputManager` und `MessageManager`.

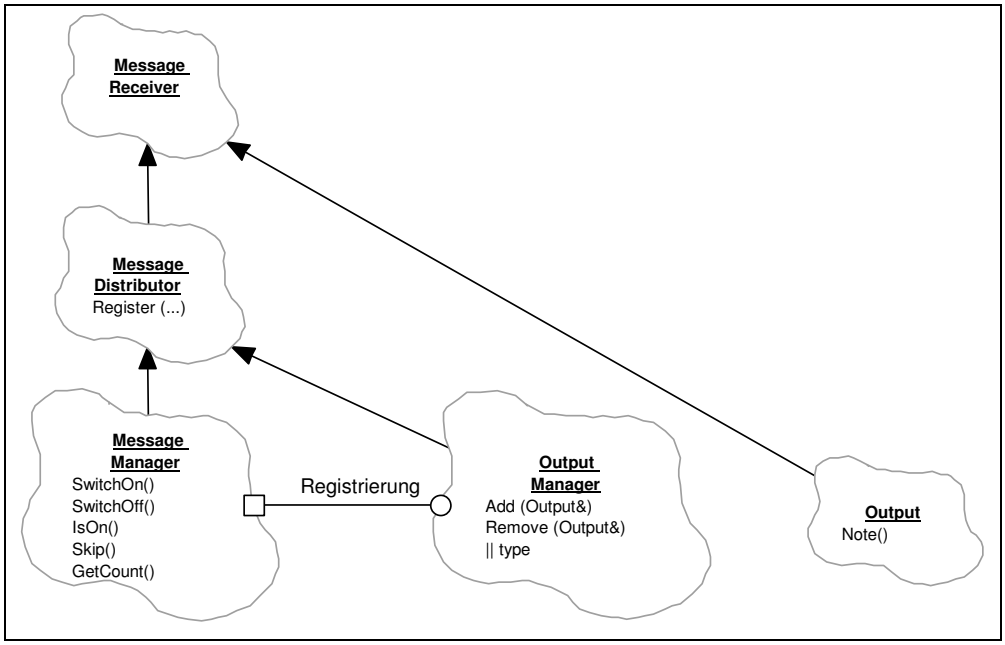

Abbildung 4-8: `OutputManager` und `MessageManager`

Über die Methoden `Add` und `Remove` können Nachrichtensenken bei einem `OutputManager` an- bzw. abgemeldet werden. Die Senken erhalten nur die Nachrichten, für die der `OutputManager` beim `MessageManager` angemeldet ist, und natürlich nur dann, wenn dort der entsprechende Kanal eingeschaltet ist.

4.2.3.4 Nachrichtensenken

Eine Nachrichtensenke ist in DESMO-C eine Unterklasse der Klasse `Output`, die wiederum von `MessageReceiver` erbt (s. Abbildung 4-8). Nachrichtensenken sind die Objekte, die Nachrichten verarbeiten, ohne sie weiter zu verteilen. Ein `Output`-Objekt ist mit einem Objekt der C++-Bibliotheksklasse `ostream` verbunden, in das die Ausgabe der Nachrichten erfolgt. Auf Wunsch kann ein `Output`-Objekt direkt eine Datei anlegen. Das `ostream`-Objekt wird dann mit einem `ofstream`-Objekt[51] initialisiert. Hierfür muß dem `Output`-Konstruktor anstelle eines `ostream`-Objekts ein Dateiname übergeben werden.

Von `Output` lassen sich beliebige Unterklassen bilden, die ein eigenes `ostream`-Objekt erzeugen, und an die Oberklasse weitergeben. Dadurch kann DESMO-C um eigene Nachrichtensenken erweitert werden, die die Nachrichten in einer vollkommen anderen Weise präsentieren können, als dies von DESMO-C im Standardfall angeboten wird. Außerdem enthält die Klasse `Output` einen Integer-Wert, der angibt, in welcher Breite die Ausgabe erfolgen soll.

Als Vorgabe stellt DESMO-C vier Nachrichtensenken zur Verfügung, die den vier Ausgabedateien in DESMO entsprechen. Hierfür faßt zunächst die Klasse `StdOutput` als Unterklasse von `Output` alle Gemeinsamkeiten der vier Standardnachrichtensenken zusammen. Dies sind die Methoden

- `Box` und `Box2` zur Ausgabe von einer bzw. zwei Zeilen in einer Box, die durch '*'-Zeichen in der entsprechenden Ausgabebreite umrahmt ist
- `Line` zur Ausgabe einer Zeile aus dem zu übergebenden Zeichen in der entsprechenden Ausgabebreite
- `ClockTime` zur zentrierten Ausgabe einer zu übergebenden Simulationszeit
- `wrap` zur Ausgabe von Fließtext beginnend bei einer bestimmten Position (`offset`) und zusätzlicher Einrückung aller weiteren Zeilen um `indent` gegenüber `offset` (`indent` und `offset` werden als Parameter übergeben

[51] `ofstream` ist eine dateibasierte Unterklasse von `ostream` zur Augabe in eine Datei.

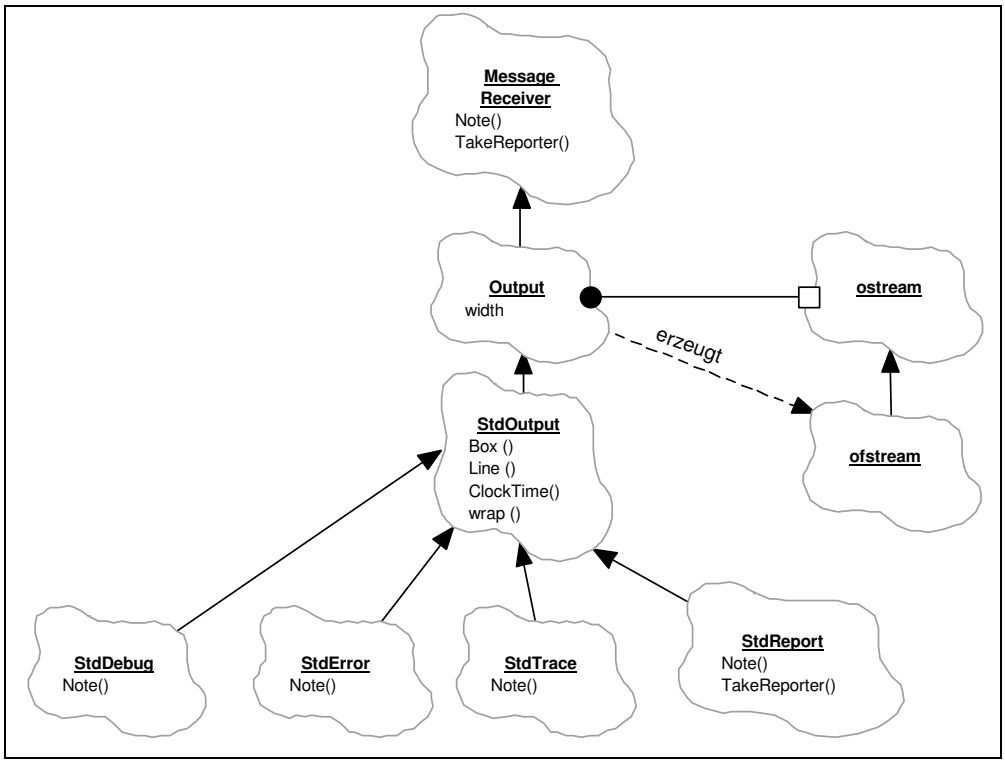

Abbildung 4-9: Standardnachrichtensenken, Unterklassen von `StdOutput`

Abbildung 4-9 zeigt auch die Unterklassen von `StdOutput` für die Standardnachrichtensenken für Debug-, Error-, Report- und Trace-Kanal. Diese Unterklassen sehen die von DESMO gewohnte Formatierung der entsprechenden Nachrichten in einer Datei vor.

Ein Beispiel für eine eigene Senke wäre eine Vorrichtung, die es gestattet, in einer grafischen Umgebung den Trace während des Experiments am Bildschirm in einem Fenster zu betrachten. Hieran wird der Vorteil des gewählten Entwurfes deutlich. Es kann ein Objekt der Klasse `StdTrace` erzeugt werden. Dessen Konstruktor wird ein `ostream`-Objekt übergeben, das die Zeichen im gewünschten Fenster darstellt. Auf diese Weise würde ein Teil der Fensterumgebung, das `ostream`-Objekt, mit der von DESMO-C angebotenen Standardformatierung (`StdTrace`) kombiniert. Das so konstruierte `Output`-Objekt kann nun über die Experiment-Methode `AddTraceOutput` mit dem Trace-Kanal verbunden werden, wodurch zu der Ausgabe in der Standard-Trace-Datei eine Ausgabe in einem Fenster erfolgt.

4.2.4 Fehlerbehandlung

4.2.4.1 Gültigkeit von Objekten

In der Klasse `NamedObject` wird der aus [Bölck89][52] übernommene Selbstreferenzmechanismus für die Gültigkeitsprüfung von Objekten angewendet, jedoch in DESMO-C nicht zum Schutz vor uninitialisierten Objekten, da in dieser Hinsicht die Sprache C++ mit Konstruktoren selbst genug bietet. Vielmehr sollen “übrig gebliebene Verweise” erkannt werden, die entstehen, wenn ein Objekt gelöscht wird, auf das aber an anderer stelle noch ein Verweis existiert. Hierzu enthalten Objekte der Klasse `NamedObject` einen Zeiger, der im Konstruktor mit `this` initialisiert wird. Im Destruktor hingegen wird der Zeiger auf

[52] Vgl. [Bölck89], S. 85.

Null gesetzt. Die Methode `Valid` liefert `true`, wenn dieser Zeiger mit `this` übereinstimmt, andernfalls `false`.

DESMO-C folgt der Konvention, Referenzen zu benutzen, wo Objekte garantiert werden können. Um dies bei Methoden einsetzen zu können, wie `Queue::First`, die u.U. kein Objekt liefern können, werden in DESMO-C die Pseudo-Objekte `NullEvent` und `NullProcess` benutzt. Letzteres wird auch als `NullEntity` verwendet. Diese Objekte dürfen jedoch nicht in Warteschlangen eingereiht werden, oder auf die Ereignisliste gesetzt werden. Um dies zu verhindern, werden diese Objekte stets als "ungültig" betrachtet. Dies wird dadurch erreicht, daß die Methode `Valid` überschrieben wird, so daß sie bei diesen Objekten stets `false` liefert. Aus diesem Grund ist `Valid` eine virtuelle Methode.

Im gesamten Simulationspaket gibt es genau ein `NullEvent` und einen `NullProcess`. D.h. diese Objekte müssen zu Beginn des Programms initialisiert werden. Hierfür ist jedoch die Angabe eines Modells erforderlich, da die Pseudo-Objekte Modellkomponenten sind, welche zu ihrer Erzeugung stets ein gültiges Modell benötigen. Aus diesem Grunde wird vorher ein Pseudo-Modell (`DefaultModel`) angelegt, das `NullEvent` und `NullProcess` übergeben werden kann. Für die Erzeugung eines Modells ist wiederum die Existenz eines Experiments notwendig. Also wird vor der Erzeugung des Modells ein Pseudo-Experiment angelegt, das erst zum Programmende wieder gelöscht wird.

4.2.4.2 Fehlerarten

DESMO-C unterscheidet drei Arten von Fehlern:

1. Warnung – zur Kennzeichnung von Situationen, die zwar einen Ausnahmefall darstellen, aber eine sinnvolle Fortführung des Experiments erlauben

2. Fehler – in Situationen, die eine sinnvolle Fortführung des Experiments nicht mehr gestatten

3. Fatale Fehler – in schweren Ausnahmezuständen, wenn ein Programmabsturz zu befürchten ist

Darüber hinaus lassen sich Fehler auch in lokale und globale Fehler einteilen. Lokale Fehler stehen im Zusammenhang mit einem bestimmten Experiment, während globale Fehler übergreifender Natur sind, und deswegen nicht an den Nachrichtenverteiler eines bestimmten Experiments gesendet werden können.

4.2.4.3 Behandlung von Fehlern

Tritt ein lokaler Fehler auf, so wird eine entsprechende Nachricht an den `MessageManager` des aktuellen Experiments geschickt. Nach der Weiterleitung in den Fehlerkanal, wird untersucht, um welche Fehlerart es sich handelt. Bei Warnungen muß nichts unternommen werden, Maßnahmen zur Fehlerbeseitigung müssen an der Stelle des Auftretens ergriffen werden. Handelt es sich um einen Fehler, so wird das aktuelle Experiment abgebrochen, nachdem der Fehler in den Fehlerkanal geleitet wurde. Es gerät damit in den Zustand `aborted`, in dem es nicht mehr fortgeführt werden kann. Die Generierung eines Reports ist hingegen immer noch möglich. Ein solches Experiment sollte gelöscht werden. Bei Auftreten eines fatalen Fehlers, wird nach Ausgabe der Meldung das Programm abgebrochen. Dabei wird jeweils vorher geprüft, ob es sich um den ersten Fehler handelt, und in diesem Fall eine Meldung auf den Bildschirm ausgegeben.

Globale Fehler werden genauso behandelt wie lokale Fehler, jedoch werden sie in einen anderen Kanal geleitet. Der Empfänger für diesen Kanal ist der `GlobalErrorManager`. Im Unterschied zu den anderen `OutputManagern` existiert er einmal im gesamten Programm, und zwar schon vor dem ersten Experiment. Dadurch ist er in der Lage, Meldungen zu verarbeiten, die vor der Erzeugung des ersten Experiments entstehen.

4.3 Experimentverwaltung

In DESMO ist die gesamte Simulationssteuerung global und existiert bis zum Programmende. Ein Ziel von DESMO-C ist die weitgehende Vermeidung globaler statischer Strukturen. Deshalb werden mit der Erzeugung eines Experiments alle zur Simulation benötigten Strukturen angelegt und mit dem Löschen des Experiments wieder entfernt. Diese Vorgehensweise ermöglicht erst die Durchführung mehrerer Experimente innerhalb eines Programms.

4.3.1 Experimentmanager

DESMO-C benötigt eine zentrale Instanz, die für die Verwaltung und Überwachung von Experimenten zuständig ist. In DESMO-C ist dies der Experimentmanager. Da es von der Klasse `ExperimentManager` innerhalb des gesamten Programms genau ein Exemplar geben soll, ist sie als Singleton[53] konzipiert. Die Konstruktoren sind geschützt, so daß auf herkömmliche Weise kein Objekt erzeugt werden kann. Der einzige Weg, an den Experimentmanager heranzukommen ist über die Klassenmethode `Instance`. Sie überprüft, ob die Klassenvariable[54] `theSingleton` bereits auf ein `ExperimentManager`-Objekt zeigt und liefert dieses ggf. zurück. Handelt es sich jedoch um den ersten Zugriff auf die Methode `Instance`, so wird `theSingleton` mit einem neuen `ExperimentManager`-Objekt initialisiert.

Der Konstruktor des Experimentmanagers erzeugt ein Objekt der Klasse `GlobalErrorManager` und legt das Pseudo-Experiment an, das seinerseits die anderen Pseudo-Objekte `DefaultModel`, `NullEvent` und `NullProcess` generiert. Das Objekt `noInterrupt` der Klasse `InterruptCode`, das den keinem Modell zugeordnet werden muß, wird direkt innerhalb des Experimentmanagers angelegt, und ist somit während des gesamten Programmlaufs beständig.

Wird ein neues Experiment angelegt, so legt diese zunächst ein neues "Experiment-Accessory" (Klasse `ExperimentAccessory`) an, das alle für die Simulationssteuerung notwendigen Datenstrukturen enthält (`Scheduler`, `SimClock`, `MessageManager`, die verschiedenen `OutputManager`, `DistribManager` und eine `ResourceDB`). Dabei übernehmen die jeweiligen `OutputManager` den Namen des Experiments, um daraus die Dateinamen der Standardausgabedateien zu bilden. Danach registriert es sich beim Experimentmanager mit Hilfe der Methode `Register`. Das bewirkt, daß das Experiment in eine Liste des Experimentmanagers eingetragen und der `GlobalErrorManager` beim `MessageManager` des Experiments angemeldet wird. Die Klasse `ExperimentAccessory` gestattet den Zugriff auf die enthaltenen Objekte nur den Klassen `Experiment` und `ExperimentManager` (s. Abbildung 4-10).

---

53 Zum Singlton-Muster s. [Gamma95], S. 139 ff.

54 Klassenvariablen sind in C++ durch das Schlüsselwort static gekennzeichnet. Eine solche Veraiable (oder Element) existiert einmal pro Klasse und ist für jedes Objekt dieser Klasse zugreifbar. Ähnliches gilt für Klassenmethoden. Auf beide kann zugegriffen werden, ohne das ein Objekt für diese Klasse existiert.

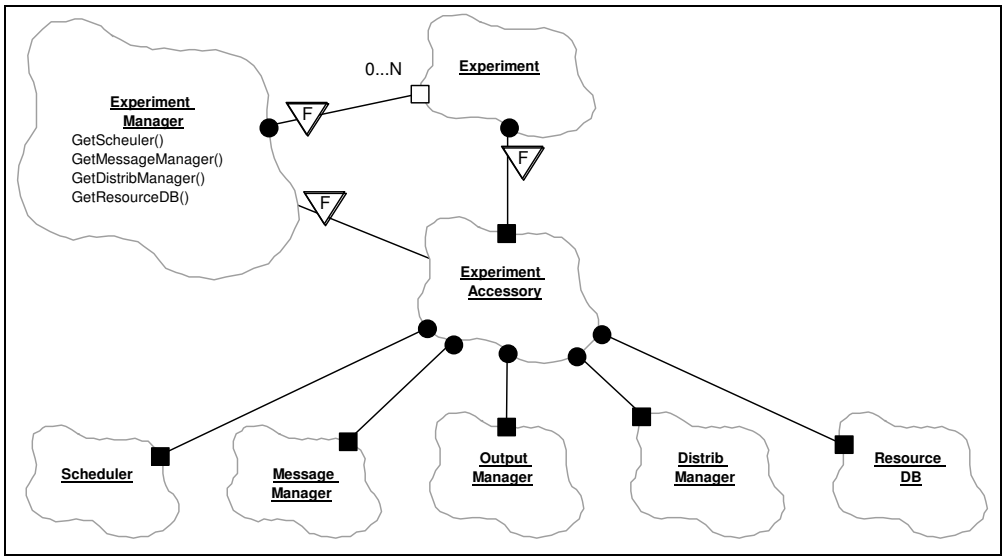

Abbildung 4-10: Experiment, -Accessory und -manager

Alle anderen Klassen können jedoch eine Verbindung über entsprechende Zugriffsmethoden des Experimentmanagers herstellen (s.u). Dabei wird in der Regel entweder ein Experiment oder eine Modellkomponente übergeben, zu dem das entsprechende Objekt der Simulationssteuerung geliefert werden soll. Z.B. lassen sich die Zufallszahlenströme bei ihrer Erzeugung über die Methode `GetDistribManager` den Manager für die Zufallszahlenströme liefern, damit sie sich bei diesem registrieren können und einen neuen Startwert erhalten. Dabei übergeben sie einen Verweis auf sich selbst, als Modellkomponente, damit der Experimentmanager über das zugehörige Modell das entsprechende Experiment und schließlich dessen Experiment-Accessory ermitteln kann, worin sich der gewünschte `DistribManager`-Objekt befindet.

Die Methode `StartExperiment` dient dem erstmaligen starten eines Experiments und erwartet als Parameter das zu startende Experiment sowie eine Simulationszeit, mit der das Experiment beginnen soll. Der Scheduler wird angewiesen, seine Simulationsuhr auf diese Zeit zu stellen, und das Hauptmodell zum aktuellen Modell zu machen. Anschließend wird das Hauptmodell zurückgesetzt, was eine Rücksetzung aller reportfähigen Objekte dieses Modells bewirkt, also auch der Untermodelle und deren Komponenten. Nach der Rücksetzung erhält jedes Modell der Hierarchie die Gelegenheit, seine ersten Vormerkungen zu tätigen, indem die Methode `DoInitialSchedules` des Hauptmodells und gleich darauf `DoInitialSchedulesOfSubModels` aufgerufen wird. Um das Experiment in Gang zu setzen, wird abschließend die Methode `ContinueExperiment` aufgerufen.

Mit `ContinueExperiment` kann ein angehaltenes Experiment fortgeführt werden. Die Methode besteht im Wesentlichen aus einer Schleife, in der der Scheduler mittels `ProcessNextEventNote` angewiesen wird, die nächste Ereignisnotiz zu verarbeiten. Diese Schleife wird solange durchlaufen, bis entweder das Experiment angehalten bzw. abgebrochen wurde (Status des Experiments: `stopped` bzw. `aborted`) oder die Ereignisliste leer ist. Im zweiten Fall wird eine entsprechende Warnung ausgegeben. Die hier beschriebene Schleife kann dazu genutzt werden, um DESMO-C so zu erweitern, daß nach der Verarbeitung einer Ereignisnotiz Aufgaben erledigt werden können, die nicht unbedingt mit der Simulation zu tun haben müssen. Dies könnte z.B. die Aktualisierung einer Darstellung bestimmter Experimentgrößen in einem Fenster einer grafischen Benutzungsoberfläche sein.

Mit den Methoden `StopExperiment` und `AbortExperiment` wird lediglich die `state`-Variable des Experiments entsprechend gesetzt, so daß das Abbruchkriterium der Schleife in `ContinueExperiment` erfüllt ist. Durch die klare Abgrenzung eines Verarbeitungsschrittes des Schedulers kann gewährleistet werden, daß nach jedem Aufruf von `ProcessNextEventNote` die Ereignisliste in einem konsistenten Zustand ist. Dies stellt einen Vorteil gegenüber DESMO dar, bei dem sich ein inkonsistenter Zustand der Ereignisliste über mehrere Anweisungen erstrecken konnte, innerhalb derer z.B. die Zeitreihen aktualisiert wurden.

Über die beiden `Report`-Methoden, die eine mit einem `Experiment`- die anderem mit `Model`-Parameter, läßt sich ein über ein Experiment bzw. Modell ein Report generieren. Die Experimentvariante gibt zunächst eine Beschreibung des Experiments aus und ruft anschließend die Modellvariante mit dem Hauptmodell als Parameter auf. Diese läßt sich vom übergeben Modell einen Reporter erzeugen, der an den Reportkanal geschickt und anschließend wieder gelöscht wird. Über die Modellvariante von Report läßt sich zu jedem Zeitpunkt gezielt ein Report für ein bestimmtes Modell generieren.

`ConnectToCurExp` wird vom Modell-Konstruktor im Falle des Hauptmodells aufgerufen. Hier wird geprüft, ob es bereits ein Experiment gibt und diese nicht schon mit einem anderen Modell verbunden ist. In Fehlerfällen wird eine Meldung über einen entsprechenden Fehler gemacht, die zum Abbruch des Programms führt, da das momentan in Konstruktion befindliche Modell nicht sinnvoll zu Ende konstruiert werden kann.

Die übrigen vom Experimentmanager angebotenen Methoden dienen lediglich dem Zugriff auf andere Objekte der Simulationssteuerung und sind hier nur aufgelistet.

Zugriff anhand eines Experiments:

```
Scheduler&       GetScheduler         (Experiment&)
Model&           GetModel             (Experiment&)
SimClock&        GetSimClock          (Experiment&)
MessageManager&  GetMessageManager    (Experiment&)
DistribManager&  GetDistribManager    (Experiment&)
ResourceDB&      GetResourceDB        (Experiment&)

Scheduler&       GetScheduler         (const ModelComponent&)
Model&           GetModel             (const ModelComponent&)
SimClock&        GetSimClock          (const ModelComponent&)
MessageManager&  GetMessageManager    (const ModelComponent&)
DistribManager&  GetDistribManager    (const ModelComponent&)
NameCatalog&     GetNameCatalog       (const ModelComponent&)
ResourceDB&      GetResourceDB        (const ModelComponent&)
```

Zugriff anhand einer Modellkomponente:

```
SimTime          CurrentTime          (const ModelComponent&)
Model&           CurrentModel         (const ModelComponent&)
Schedulable&     Current              (const ModelComponent&)
Event&           CurrentEvent         (const ModelComponent&)
Entity&          CurrentEntity        (const ModelComponent&)
Process&         CurrentProcess       (const ModelComponent&)
```

Zugriff jederzeit:

```
Experiment&          CurrentExperiment ()
Scheduler&           CurrentScheduler ()
Model&               CurrentModel ()
SimClock&            CurrentSimClock ()
MessageManager&      CurrentMessageManager ()

GlobalErrorManager& GetGlobalErrorManager ()
Experiment&          GetDefaultExperiment ()
Model*               GetNullModel ()
Event*               GetNullEvent ()
Entity*              GetNullEntity ()
Process*             GetNullProcess ()
```

4.3.2 Speichermangel

Der Konstruktor des Experimentmanagers reserviert 50000 Bytes für den Fall, daß der Arbeitsspeicher knapp wird. Zusätzlich installiert er einen sog. `new`-Handler. Dies ist eine Funktion, die der C++-Standardbibliothek übergeben werden kann, damit sie aufgerufen wird, wenn `new` keinen Speicher allozieren kann. In der Funktion kann dann der Versuch unternommen werden, nicht unbedingt benötigte Speicherbereiche freizugeben, so daß `new` letztlich doch noch erfolgreich ausgeführt werden kann.

Der `new`-Handler des Experimentmanagers ist die globale Funktion `outOfMemory`. Sie gibt zunächst den reservierten Speicherbereich frei und ruft die gleichnamige Methode des Experimentmanagers auf. Dort wird eine entsprechende Meldung auf dem Bildschirm ausgegeben und das aktuelle Experiment abgebrochen. Die aktuelle Ereignisnotiz wird erst zu Ende verarbeitet. Danach wird dem Nutzer die Möglichkeit gegeben, für das aktuelle Experiment noch einen Report zu generieren, bevor das Programm abgebrochen wird.

4.3.3 Ende des Experiments

Beim Löschen eines Experiments meldet es sich zunächst beim Experimentmanager, mit Hilfe der Methode `DeRegister` ab, worauf es aus der Experimentliste entfernt wird. Im Anschluß daran wird das Experiment-Accessory gelöscht, so daß alle Objekte zur Simulationssteuerung ebenfalls gelöscht werden. Das Löschen der `OutputManager`, die im Experiment-Accessory enthalten sind, bewirkt, das die Standardausgaben geschlossen werden. Sind die zugehörigen Dateien leer, so werden sie gelöscht. So bleiben nach dem Ende des Experiments nur diejenigen Dateien bestehen, die auch Informationen enthalten. Das Hauptmodell sollte zu diesem Zeitpunkt bereits gelöscht sein. Für seine Zerstörung ist der Modellentwickler bzw. der Modellbenutzer zuständig.

4.3.4 Dynamische Objekte und Freispeicherverwaltung

Um die Freispeicherverwaltung für dynamische Objekte (Klasse `DynamicalObject`) umzusetzen, enthält jedes Modell einen Katalog (Klasse `DynObjectCatalog`) für dynamische Objekte. Der Konstruktor von `DynamicalObject` nimmt mittels `Register` eine Registrierung beim Experimentmanager vor, was dazu führt, daß das dynamische Objekt im Katalog seines Modells eingetragen wird. Das Löschen eines dynamischen Objekts führt zur Entfernung des Katalogeintrages.

Wird ein Modell gelöscht, so wird mit ihm auch der Katalog gelöscht. Dieser ist als einfach verkettete Liste implementiert, in der neue Einträge am Ende eingefügt werden. Bei der Entfernung eines Eintrages wird die Liste vom Anfang an durchsucht. Dies entspricht der

erwarteten Reihenfolge der Entstehung und Vernichtung von dynamischen Objekten. Beim Löschen des gesamten Kataloges werden zunächst alle noch in der Liste befindlichen Objekte als "Müll" markiert (Flag: `isGarbage`), damit diverse Fehlerprüfungen beim Löschen keine Meldung ausgeben. Anschließend werden die Objekte beginnend beim ersten gelöscht. In diesem Fall darf nicht der Konvention gefolgt werden, Objekte in der umgekehrten Reihenfolge ihrer Erzeugung zu löschen, da folgende Situation möglich ist:

```
1. class D1 : public DynamicalObject {...}
2.
3. class D2 : public DynamicalObject
4. {
5.     ...
6.     private:
7.         D1    innerObject;
8. };
```

Listing 4-9: Beispiel geschachtelter dynamischer Objekte

In Listing 4-9 führt die Erzeugung eines Objekts der Klasse `D2` zunächst zur Registrierung des `D2`-Objekts. Anschließend wird das Element `innerObject` erzeugt, das ebenfalls ein dynamisches Objekt ist und nun, also nach dem `D2`-Objekt registriert wird. D.h. im Katalog steht `innerObject` hinter dem `D2`-Objekt. Würde der Katalog die Liste von hinten traversieren, würde also zuerst `innerObject` gelöscht werden und anschließend das `D2`-Objekt. Das wiederum würde dazu führen, daß `innerObject` ein zweitesmal gelöscht werden würde, was unvorhersehbare Folgen haben kann. Beginnt der Katalog jedoch, die Liste von vorne abzuarbeiten, tritt dieses Problem nicht auf. Das `D2`-Objekt wird zuerst gelöscht und mit ihm `innerObject`, wodurch beide auch automatisch aus dem Katalog entfernt werden. Die Löschung von `innerObject` muß vom Katalog also gar nicht mehr in die Wege geleitet werden.

Ein ähnliches Problem könnte sich ergeben, wenn wie in Listing 4-9 ein Modell ein dynamisches Objekt enthält, z.B. einen statischen Prozeß. Da jedoch der Destruktor der speziellen Modellklasse vor dem Destruktor der Klasse `Model` aufgerufen wird, ist der Prozeß bereits gelöscht, bevor der Katalog gelöscht wird.

4.3.5 Automatische Numerierung von Namen

Vormerkbaren Objekten (Klasse `Schedulable`) wird bei ihrer Erzeugung eine Seriennummer an den Namen gehängt. Diese Nummer muß in Bezug auf den vollständigen Namen und das zugehörige Modell eindeutig sein. Zur Realisierung enthält jedes Modell ein Namensverzeichnis der Klasse `NameCatalog`, die eine Unterklasse von `Avl`[55], also ein Avl-Baum ist. Das Verzeichnis wird mit der Breite, in der Namen ausgegeben werden sollen, und der Anzahl der Stellen für die Seriennummer initialisiert und bietet die Methode `AddNumberTo` an. Ihr wird ein `String`-Objekt übergeben, das mit der entsprechenden Seriennummer versehen wird und als neues `String`-Objekt zurückgeliefert wird. Hierzu wird zunächst mit Hilfe der Methode `GetNumberOf` geprüft, ob der Name bereits im Verzeichnis vorhanden ist und ggf. die nächste zu vergebende Nummer für diesen Namen zurückgeliefert. Falls der Name noch nicht verzeichnet war, wird er angelegt. Jeder Name wird so zusammen mit der nächsten zu vergebenden Nummer im Verzeichnis verwaltet.

[55] Vgl. Abschnitt 4.4.2.5

4.4 Hilfsklassen

4.4.1 Koroutinen

In DESMO wurde das von Modula-2 angebotene Koroutinen-Konzept[56] verwendet, für das es in C oder C++ keine standardisierte Lösung gibt. Mit der Klasse `Coroutine` ist in [Weber96] eine portable Realisierung des Koroutinen-Konzepts[57] aus Modula-2 in C++ gelungen. Hierfür konnten die Funktionen `setjmp` und `longjmp` sowie die Datenstruktur `jmp_buf` genutzt werden, die allesamt Teil der C-Standardbibliothek sind.[58] Dabei wird mit `setjmp` der aktuelle Ausführungskontext gesichert, jedoch mit Ausnahme des Stacks, der die Prozeduraufrufkette und lokale Variablen enthält. `longjmp` stellt einen mit `setjmp` gesicherten Kontext wieder her. `setjmp` liefert einen Wert zurück, an dem erkannt werden kann, ob `setjmp` gerade einen Kontext gesichert hat, oder über den Aufruf von `longjmp` ein Kontext restauriert wurde.

```
 1. jmp_buf context1, anotherContext;
 2.
 3. // Kontext 1
 4. ...
 5. if (setjmp (context1) == 0)
 6.     // Kontext 1
 7.     ...
 8. else
 9.     // Kontext stammt aus dem letzten longjmp-Aufruf
10.     ...
11. ...
12. longjmp (anotherContext, 1);
```

Listing 4-10: Beispiel für `setjmp` und `longjmp`

In Listing 4-10 wird beim ersten Aufruf von `setjmp` mit dem Parameter `context1` der aktuelle Kontext in die Variable `context1` gesichert. `setjmp` liefert 0 zurück, so daß die Ausführung in dem `if`-Zweig fortgesetzt wird. Der Aufruf von `longjmp` mit dem Parameter `anotherContext` bewirkt, daß der (irgendwo anders gesicherte) Kontext `anotherContext` restauriert wird. Die Ausführung wird an der Stelle fortgesetzt, an der `setjmp` mit `anotherContext` zuletzt aufgerufen wurde. Und zwar liefert `setjmp` als Ergebnis den Wert, der `longjmp` an zweiter Stelle übergeben wurde (in Zeile 12 ist dies der Wert 1). Wurde also `anotherContext` in einem anderen Durchlauf des Beispiels aus Listing 4-10 gesichert, so würde die Ausführung nach dem `longjmp` im `else`-Zweig fortgesetzt werden.

Um den Zustand einer Koroutine zu sichern, müssen der "relevante" Stackbereich, der die Prozeduraufrufkette, lokale Variablen und Prozedurparameter enthält, sowie die meisten Register gespeichert werden, die u.a. auch den Stackpointer enthalten. Schematisch sieht ein `Transfer`[59] etwa folgendermaßen aus:

Sichere Stack und Register der gerade aktiven Koroutine
Restauriere Stack und Register der zu aktivierenden Koroutine

---

56 Vgl. [DalCi89], S. 223 ff.

57 Vgl. [DalCi89], S. 223 ff.

58 Vgl. [Page88]

59 Auf Koroutinen definierte Prozedur, um die Kontrolle von einer zur anderen Koroutine zu übertragen. Vgl. [DalCi89], S. 223 ff.

Um die Registerwerte kümmern sich die Funktionen `setjmp` und `longjmp`. Der Stack kann mittels `memcpy`, einer Funktion aus der C-Standardbibliothek, kopiert werden.

4.4.1.1 Die Klasse `Coroutine` aus [Weber96]

Die Klasse `Coroutine` benötigt eine explizite Initialisierung, und zwar aus der Funktion `main` heraus. Diese Initialisierung liefert eine Stelle auf dem Stack, von der aus neue Koroutinen gestartet werden. Während der Benutzung der Koroutinen darf diese Stelle niemals unterschritten werden, d.h. der Stackpointer darf keinen kleineren Wert annehmen (bei aufsteigendem Stack[60]). Es ist somit erforderlich, daß Klienten der Klasse `Coroutine` in `main` einen bestimmten Aufruf ausführen. Solange also die Funktion `main` noch nicht ausgeführt wird, d.h. während der Initialisierungsphase des Programms, dürfen keine Koroutinen erzeugt werden. Das bedeutet, daß Objekte, die nicht mittel- oder unmittelbar durch `main` mittels `new` erzeugt werden, keine Erzeugung von Koroutinen bewirken dürfen.

Um dem Anwender die Initialisierung des Koroutinenmechanismus nicht zumuten zu müssen, das in [Weber96] ohnehin nur in prozeßorientierten Modellen benötigt wird, wird die Funktion `main` in das Framework verlagert. Der Modellprogrammierer muß sich für einen von fünf Modellierungsstilen entscheiden und dementsprechend eine Anwendungsklasse aus dem Framework beerben. Die Funktion `main` steckt in diesen Anwendungsklassen, so daß im Falle eines prozeßorientierten Ansatzes der Koroutinenmechanismus initialisiert werden kann. D.h. das Framework läßt sich nicht in einen übergeordneten Kontext betten, in der die Entscheidung über den Modellierungsstil erst zur Laufzeit des Programms getroffen wird.

Die Klasse `Coroutine` lehnt sich sehr eng an das Modula-2 Konzept in [DalCi89] an. So wird in [Weber96] zur Initialisierung einer Koroutine ein Zeiger auf eine globale Funktion mit einem typlosen Zeigerparameter erwartet. Diese Funktion repräsentiert den Handlungsstrang der Koroutine, der mit Hilfe der Methode `Transfer` die Kontrolle übergeben werden kann.

4.4.1.2 Die Klasse `Coroutine` in DESMO-C

In DESMO-C wurde die Klasse `Coroutine` aus [Weber96] so umgestaltet, daß in Unterklassen nunmehr eine virtuelle Methode `Body` zu definieren ist, die den Handlungsstrang der Koroutine repräsentiert. Ferner kann eine explizite Initialisierung entfallen, da alle Klassenvariablen bei Erzeugung der ersten Koroutine in deren Konstruktor über den Aufruf der Klassenmethode `InitMainCoroutine` initialisiert werden. In dieser Methode wird mittels `setjmp` der Kontext für den Start von zukünftigen Koroutinen gesetzt, in dem einfach die Methode `Body` aufgerufen wird, um die Koroutine zu starten. Dieser Kontext kann bei neuen Koroutinen bei deren Erzeugung direkt in die Attribute eingetragen werden, so daß die erste Kontrollübergabe keine Sonderbehandlung erfordert, weil ein gültiger Kontext bereits zur Verfügung steht. Der Konstruktor benötigt keine Parameter. Mit `Transfer` wird einer Koroutine die Kontrolle übertragen. Abbildung 4-11 zeigt eine Anwendung der Klasse `Coroutine` am Beispiel der Klasse `ProcessImplementation`.

[60] In Betriebssystemen ist der Stack entweder als nach oben oder als nach unten wachsende Struktur implementiert.

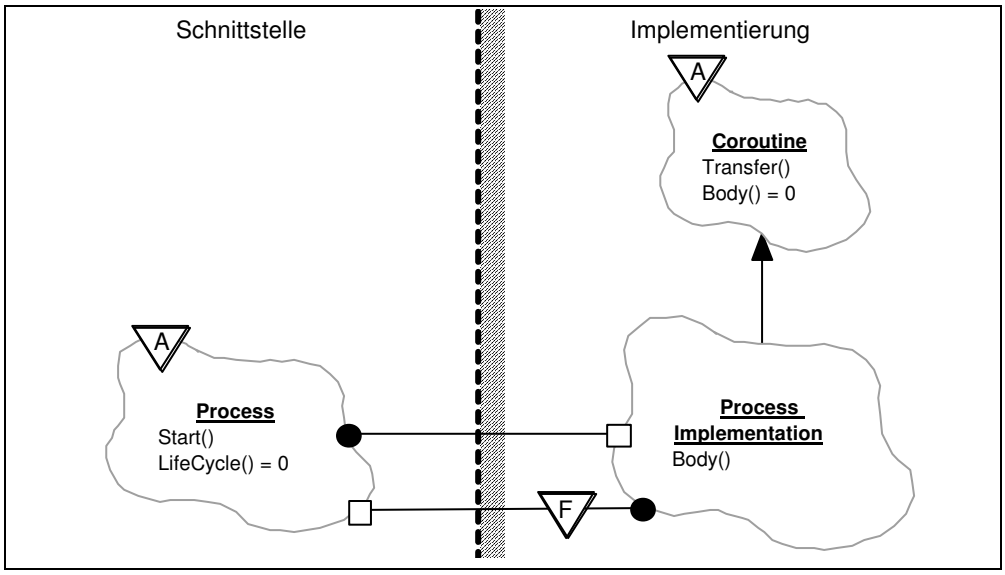

Abbildung 4-11: Die Koroutine `ProcessImplementation`

In Abbildung 4-11 erbt `ProcessImplementation` von `Coroutine` und definiert die Methode `Body` so, daß die Methode `Start` des zugehörigen Prozesses aufgerufen wird. `Start` wiederum ruft `LifeCycle` auf, und transferiert nach der Rückkehr aus `LifeCycle` die Kontrolle an das Hauptprogramm (`MainCoroutine`), nachdem der Prozeß als "terminiert" markiert wurde.

Bei der für einen Kontextwechsel notwendigen Restauration des Stacks ergibt sich ein Problem, wenn der zu restaurierende Stack größer ist als der gerade gesicherte. Denn der Aufruf von `memcpy` legt neue Daten auf den Stack, während in `memcpy` dieser Bereich durch das Zurückkopieren des neuen Stacks überschrieben wird[61]. Wenn `memcpy` jemals trotz des manipulierten Stacks korrekt zurückkehren kann, wird es aber durch die Rückkehr aus dem Funktionsaufruf den gerade restaurierten Stack der neuen Koroutine zerstören.

Dieses Problem beim Restaurieren des Stacks wird in der von [Weber96] übernommenen Methode `resume_execution` behandelt, die für DESMO-C in `RestoreStack` umbenannt ist. Dabei wird zunächst die aktuelle Stack-Adresse ermittelt. Würde ein Zurücksichern des Stacks die Stelle des Stackpointers überschreiben, so wird `resume_execution` solange rekursiv aufgerufen, bis der Stackpointer aus der Gefahrenzone bewegt ist. In DESMO-C wird in einem solchen Fall die so ermittelte Stelle mit einem zusätzlichen `setjmp` gespeichert, so daß sie bei folgenden Aufrufen von `RestoreStack` mittels `longjmp` direkt angesprungen werden kann. Sollte sich dann der Stackpointer immer noch in der Gefahrenzone befinden, so wird er über weitere rekursive Aufrufe von `RestoreStack` wieder in einen sicheren Bereich gebracht, und diese Stelle mit `setjmp` erneut gespeichert. Dadurch ist der Algorithmus zum Restaurieren des Stacks gegenüber [Weber96] um einiges effizienter.

4.4.1.3 Konsequenzen

Durch die automatische Initialisierung der Klasse `Coroutine`, muß dem Anwender kein zusätzlicher Aufruf aufgebürdet werden. Die Funktion `main` kann also vom Anwender definiert werden, der dadurch volle Kontrolle über sein Programm erhält. Bei diversen Tests mit DESMO-C ergab sich jedoch eine Einschränkung:

[61] Vgl. [Schni96], S. 14

Objekte, die global zugreifbar sein sollen, müssen dynamisch erzeugt, d.h. mittels `new`, oder in einem dynamisch erzeugten Objekt enthalten sein, sich also nicht auf dem Stack befindet.

Wird dies nicht befolgt, kann es passieren, daß bei einem Kontextwechsel das Objekt mit dem Stack an eine andere Stelle gesichert wird. Wird nun versucht, dieses Objekt über einen Verweis anzusprechen, befindet sich an der Stelle, auf die der Verweis zeigt, wahrscheinlich bereits etwas anderes, nur nicht das gewünschte Objekt. Dies kann natürlich unvorhersehbare Folgen haben. Leider gibt es kein Verfahren, um ein dynamisches Anlegen von Objekten zu erzwingen, ohne neue Einschränkungen hervorzurufen. Dieses Thema wird ausführlich in [Meyer98] auf den Seiten 160 bis 171 behandelt.

4.4.2 Aus SiFrame übernommene Klassen

Da DESMO-C gegenüber SiFrame eine gänzlich neue Architektur aufweist, war eine Neuimplementierung erforderlich. In einigen Fällen konnte auf Codefragmente aus SiFrame zurückgegriffen werden, die in die Implementierung der neuen Klassen eingesetzt wurden, wie z.B. bei den Berechnungsverfahren der statistischen Datensammelobjekte. Die Wiederverwendung ganzer Klassen war nur in wenigen Fällen möglich. Selbst hier waren teilweise geringere Änderungen oder Erweiterungen erforderlich. Die in ihrem Wesen belassenen und übernommenen Klassen sind in diesem Abschnitt kurz beschrieben. Für eine ausführlichere Darstellung sei auf [Weber96] verwiesen.

4.4.2.1 String

Die Klasse `String` dient der Repräsentation und Verarbeitung von Zeichenketten. Sie weist eine Vielzahl von Konstruktoren und Operatoren auf, so daß eine intuitive Nutzung möglich ist. Z.B. ist die Konkatenation mit Hilfe des `operator+` implementiert, so daß z.B. folgendes möglich ist:

```
String s1 = "Hello ";
String s2 = "world!";
String s3 = s1 + s2;     // s3 == "Hello world!"
```

Für DESMO-C wurde die Klasse um die Methoden `Left` und `Right` erweitert, die den Anfang bzw. das Ende der Zeichenkette in einer zu übergebenden Länge liefern.

4.4.2.2 SimTime

`SimTime` ist die Klasse der Simulationszeitpunkte. Auf ihr sind alle sinnvollen arithmetischen Operationen definiert sowie für ein Eingabe- und Ausgabe. Die Pseudo-Simulationszeitkonstante `NOW` ist ebenfalls hier gekapselt und über die Methode `Now` abrufbar.

4.4.2.3 Ring

`Ring` ist eine Klasse für eine zu einem Ring doppelt verkettete Liste mit einer Einfügeposition. Für DESMO-C wurde diese Klasse als Template-Klasse ausgebaut, da sie sehr häufig eingesetzt wird. Durch Methoden `Push` und `Pop` bzw. `Enqueue` und `Dequeue` eignet sie sich auch zur Implementierung von Stapelspeichern bzw. Warteschlangen.

4.4.2.4 QueueLink

Die Klasse `QueueLink` ist das Bindeglied zwischen Warteschlangen und Entities. Da in DESMO-C, wie auch schon in SiFrame, Entities in mehr als einer Warteschlange warten können, kann nicht einfach der Verweis auf die Warteschlange im Entity selbst gespeichert werden. `QueueLink`-Objekte lassen sich in zwei Dimensionen verketten. Eine für alle Entities einer Warteschlange, die andere für alle Warteschlangen, in denen ein Entity eingetragen ist. Allerdings mußte diese zweite Verkettung korrigiert werden, da sie fehlerhaft arbeitete.

4.4.2.5 Avl

`Avl` ist die Klasse für ausgeglichene Baumstrukturen, und konnte ohne Änderungen übernommen werden. Sie kommt zur Anwendung beim Namensverzeichnis für vormerkbare Objekte und zur Implementierung der Ressourcedatenbank.

4.4.2.6 ResourceDB

Für jedes Experiment wird ein Objekt der Klasse `ResourceDB` eingesetzt, um über Ressourcenbelegungen und -Freigaben buchzuführen. Dabei wird ein Avl-Baum zum Zugriff über ein `Res`-Objekt und ein AVL-Baum zum Zugriff über einen Prozeß verwaltet. Zusammen realisieren sie den Ressourcenallokationsgrafen, in dem über die Methode `DeadlockCheck` die Suche nach Zyklen durchgeführt werden kann. In SiFrame gab es keine Möglichkeit die Überwachung auszuschalten. Die Prüfung fand stets auf der Stufe `DynamicA` statt. Für DESMO-C wurden die Methoden `DynamicCheckA` und `DynamicCheckB` und eine Möglichkeit hinzugefügt, den Überwachungslevel auf eine der Stufen `DynamicA` oder `DynamicB` einzustellen oder ganz abzuschalten.

5 Anwendung und Beispiele

5.1 Schrittweise Entwicklung eines einfachen Modells

In diesem Abschnitt wird ein einfaches kleines Modell vorgestellt, das einmal im prozeßorientierten und ein zweitesmal im ereignisorientierten Modellierungsstil schrittweise entwickelt und mit Hilfe von DESMO-C implementiert wird.

5.1.1 Das Ping-Pong-Modell

Als einfaches Beispiel dient ein Spiel, das dem Tischtennis ähnelt. Ein Ball wird von zwei Seiten über ein Netz gespielt. Beim Kontakt mit den Schlägern entsteht ein “Ping”- bzw. “Pong”-Geräusch (Abbildung 5-1).

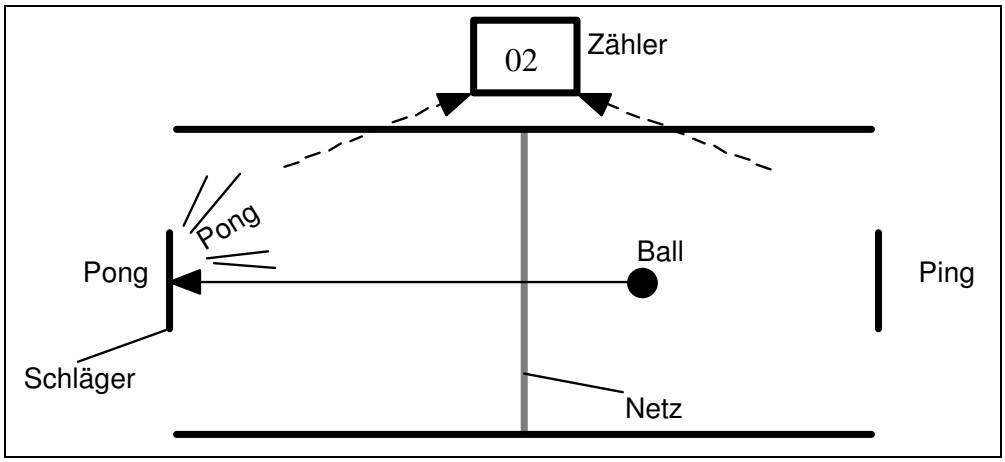

Abbildung 5-1: Das Ping-Pong-Modell

Der Ball wird zunächst von dem Schläger auf der Ping-Seite über das Netz geschlagen. Für die Netzüberquerung benötigt der Ball eine Zeiteinheit. Auf der anderen Seite angekommen wird er vom Pong-Schläger über das Netz geschlagen, wofür er wiederum eine Zeiteinheit benötigt. Um die Ballkontakte zu zählen, soll ein Zähler bei jedem Kontakt des Balls mit einem Schläger um eins erhöht werden.

5.1.2 PingPong (prozeßorientiert)

In der prozeßorientierten Version des Modells wird der Ball als einziger Prozeß modelliert (Klasse `PingPongBall`). Die Schläger werden nicht mitmodelliert. Für die Zählung der Ballkontakte wird ein `Count`-Objekt verwendet.

5.1.2.1 Der Ping-Pong-Prozeß

Der Ball kann durch einen Prozeß beschrieben werden, der, beginnend auf der Ping-Seite, endlos zwischen den beiden Schlägern hin- und herspringt. Vor jeder Netzüberquerung, muß er den Zähler aktualisieren. Listing 5-1 zeigt den Lebenszyklus des Balls.

```
 1. void PingPongBall::LifeCycle ()
 2.     {
 3.         while (true)    // endlos
 4.         {
 5.             TraceNote ("Ping");
 6.             counter.Update();
 7.             Hold (1);
 8.             TraceNote ("Pong");
 9.             counter.Update();
10.             Hold (1);
11.         }
12.     }
```

Listing 5-1: Lebenszyklus des Ping-Pong-Balls

Der Lebenszyklus des Balls besteht aus einer Endlosschleife (Zeile 3 bis 11). Zunächst wird eine Ausgabe in den Trace geschrieben, um die Seite zu kennzeichnen, auf der sich der Ball befindet (Zeile 5). Bevor das Netz überquert wird, muß der Zähler aktualisiert werden (Zeile 6), d.h. der Ball muß Zugriff auf den zu aktualisierenden Zähler haben. Der Zeitverbrauch für die Netzüberquerung wird durch das `Hold` erzielt (Zeile 7). Nach dem `Hold` ist der Ball auf der anderen Seite angekommen, und hat Kontakt mit dem Schläger, was er durch eine Ausgabe in den Trace wieder kennzeichnet (Zeile 8). In den Zeilen 9 und 10 wird analog zur ersten Seite verfahren. Nach einem Schleifendurchlauf hat der Ball das Netz zweimal überquert. Dabei sind insgesamt zwei Zeiteinheiten vergangen. Eine Trace-Ausgabe sieht dann folgendermaßen aus:

```
  Time Event        Entity       Action(s)
------------------------------------------------------------
0.000 ---           ---          activates 'Ball 1' now
                    Ball 1       Ping
                                 holds for 1.000, until 1.000
1.000                            Pong
                                 holds for 1.000, until 2.000
2.000                            Ping
                                 holds for 1.000, until 3.000
```

Listing 5-2: Trace-Ausgabe des Ping-Pong-Balls

Der erste Eintrag im Trace zum Zeitpunkt 0.000 mit dem Zeichen “---” für Ereignis und Entity bedeutet, daß dies die erste Aktivierung durch das Modell ist. In der zweiten Zeile hat dann der Ball die Kontrolle. Dort bedeutet “---” in der Ereignisspalte, daß das angegebene Entity der jetzt aktive Prozeß ist. “Ping” und “Pong” sind Resultat der `TraceNote`-Aufrufe aus den Zeilen 5 bzw. 8 in Listing 5-1.

Das einzige Attribut, das der Ball für dieses Modell benötigt, ist ein Verweis auf den zu aktualisierenden Zähler. Dies führt zu der Klassendeklaration in Listing 5-3.

```
 1. class PingPongBall : public Process
 2. {
 3.     public:
 4.         PingPongBall (Model& owner, Count& c)
 5.             :   Process (owner, "Ball"),
 6.                 counter (c)
 7.             {}
 8.
 9.     protected:
10.         virtual void    LifeCycle ();
11.
12.     private:
13.         Count&  counter;
14. };
```

Listing 5-3: Die Klasse `PingPongBall`

Da der Ball als Prozeß modelliert wird, erbt die Klasse `PingPongBall` von `Process`. Dem Konstruktor muß sowohl das Modell, zu dem der Ball gehören soll, als auch eine Referenz auf einen Zähler (Klasse `Count`) übergeben werden (Zeile 4), der irgendwo außerhalb des Ballprozesses definiert ist. An den Konstruktor der Oberklasse `Process` wird das Modell und der Name “Ball” weitergegeben (Zeile 5). D.h. alle Ping-Pong-Bälle in diesem Modell heißen “Ball”. Das einzige Attribut des Balls, die Zählerreferenz (Zeile 13), wird in Zeile 6 mit dem übergebenen Zähler `c` initialisiert. Weitere Initialisierungen sind nicht erforderlich. Ein solcher Ball kann unmittelbar nach seiner Erzeugung aktiviert werden.

5.1.2.2 Modell und Experiment

Das prozeßorientierte Modell selbst ist sehr übersichtlich, da die Hauptaktivitäten bereits im Ping-Pong-Prozeß enthalten sind. Die Aufgabe des Modells besteht hier nur in der Bereitstellung des Zählers und der Erzeugung und ersten Aktivierung des Ping-Pong-Balls.

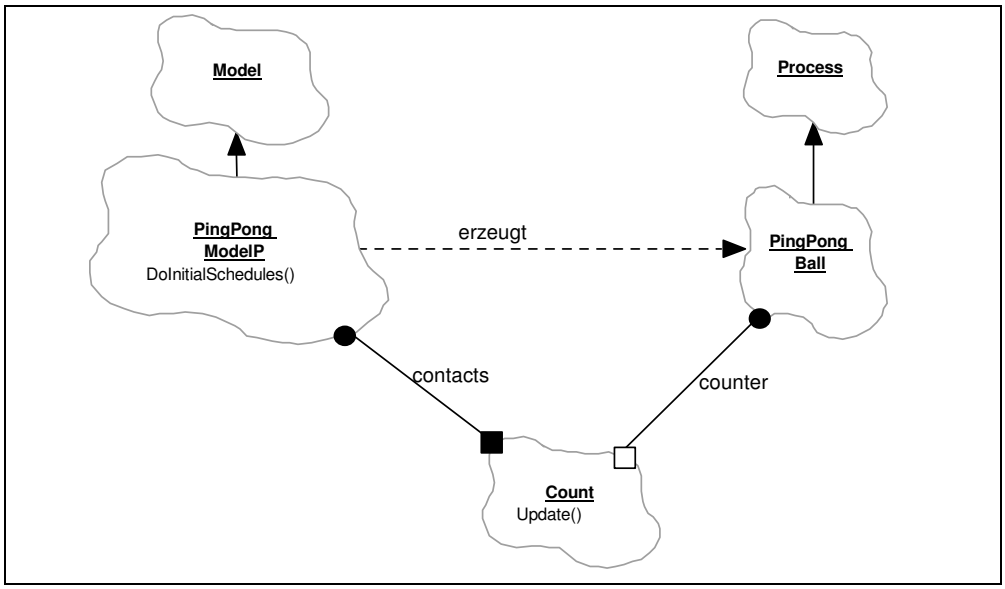

Abbildung 5-2: Das prozeßorientierte Ping-Pong-Modell

Die Beziehungen zwischen den Klassen des prozeßorientierten Ping-Pong-Modells sind in Abbildung 5-2 dargestellt. Im Klassendiagramm ist ersichtlich, daß der Zähler physikalisch im Modell enthalten ist, während der Prozeß nur einen Verweis auf ihn besitzt. Aus diesem Grund muß das Modell bei der Erzeugung des Prozesses diesem einen Verweis auf den Zähler übergeben. Deklaration und Implementierung der Methode `DoInitialSchedules` sind in Listing 5-4 bzw. Listing 5-5 wiedergegeben.

```
 1. class PingPongModelP : public Model
 2. {
 3.     public:
 4.         PingPongModelP (Model* owner)
 5.             :   Model       (owner, "PingPong P-Model"),
 6.                 contacts    (*this, "Contacts")
 7.             {}
 8.
 9.     protected:
10.         virtual void    DoInitialSchedules();
11.
12.         Count   contacts;   // Zaehler fuer Ballkontakte
13. };
```

Listing 5-4: Die Modellklasse `PingPongModelP`

Auf eine Besonderheit in Listing 5-4 soll an dieser Stelle eingegangen werden. Modellen wird grundsätzlich ein Zeiger auf das übergeordnete Modell in der Hierarchie übergeben, während alle anderen Modellkomponenten hier eine Referenz erwarten. Der Grund dafür ist, daß mit einem Nullzeiger bei Modellen ausgedrückt wird, daß es kein übergeordnetes Modell gibt, es sich somit um das Hauptmodell handelt. So erwartet auch das Ping-Pong-Modell einen Zeiger (Zeile 4), der in Zeile 5 direkt an die Oberklasse `Model` weitergeleitet werden kann. Zur Initialisierung der Modellkomponente `contacts`, ein `Count`-Objekt, darf auf keinen Fall der `owner`-Parameter übergeben werden, da es sich hierbei um einen Zeiger handelt, der u.U. auch ein Null-Zeiger sein könnte. Der Zähler muß in jedem Fall ein gültiges Modell erhalten, dem er zuzuordnen ist. Dieses ist ja gerade das Ping-Pong-Modell, in dessen Konstruktor wir uns befinden. Also wird der `this`-Zeiger dereferenziert und an den Zähler `contacts` übergeben (Zeile 6).

```
1. void PingPongModelP::DoInitialSchedules ()
2. {
3.     Process* ball = new PingPongBall (*this, contacts);
4.     ball->Activate (0);
5. }
```

Listing 5-5: Die erste Aktivierung: `DoInitialSchedules`

In Listing 5-5 ist die Methode `DoInitialSchedules` des Ping-Pong-Modells wiedergegeben, die automatisch beim Start des Experiments aufgerufen wird. In Zeile 3 wird ein neuer Ping-Pong-Ball erzeugt. Neben dem Verweis auf das Modell selbst (`this`) wird ihm der Zähler (`contacts`) übergeben. In Zeile 4 wird die Aktivierung des Balls für den Zeitpunkt 0.0 vorgemerkt, so daß die Ereignisliste durch den Scheduler abgearbeitet werden kann. Die erste Aktivierung führt dazu, daß der Prozeß seine Handlungen aufnimmt (Listing 5-1).

Bevor das Modell erzeugt werden kann, muß ein neues Experiment angelegt werden. Listing 5-6 zeigt das noch fehlende Hauptprogramm.

```
 1. void main()
 2. {
 3.     Experiment* pe = new Experiment("PingPong process");
 4.     Model*      pm = new PingPongModelP (0);
 5.
 6.     pe->TraceOn ();
 7.     pe->Stop    (10);
 8.     pe->Start   ();
 9.     pe->Report  ();
10.
11.     delete pm;
12.     delete pe;
13. }
```

Listing 5-6: Das Hauptprogramm für das prozeßorientierte Ping-Pong-Modell

Zeile 3 in Listing 5-6 legt ein neues Experiment an und nennt es “PingPong process”. Dadurch erhalten die Standardausgabedateien den selben Namen zuzüglich der jeweiligen Endung. Zeile 4 legt ein neues Ping-Pong-Modell an. Die Null als Parameter kennzeichnet das Modell als Hauptmodell, so daß es mit dem zuvor erzeugten Experiment verknüpft wird. Zeile 6 schaltet die Ablaufverfolgung ein. In Zeile 7 wird das Experiment angewiesen, die Simulation nach 10 Zeiteinheiten anzuhalten. Zeile 8 startet das Experiment, wodurch die Methode `DoInitialSchedules` des Modells aufgerufen wird. Nach den in Zeile 7 eingestellten 10 Zeiteinheiten kehrt die Kontrolle aus dem Aufruf von `Start` zurück, so daß in Zeile 9 der Report generiert werden kann. In den Zeilen 11 und 12 werden schließlich Modell und Experiment gelöscht. Listing 5-7 und Listing 5-8 zeigen Trace und Report in ihrer vollständigen Darstellung.

```
                                  Clock Time = 0.000
********************************************************************************
*                                                                              *
*                            Experiment: PingPong process                      *
*                                                                              *
*                                      Trace                                   *
*                                                                              *
********************************************************************************


Model              Time Event        Entity       Action(s)
--------------------------------------------------------------------------------
PingPong P-M      0.000 ---          ---          activates 'Ball 1' now
                                     Ball 1       Ping
                                                  holds for 1.000, until 1.000
                  1.000                           Pong
                                                  holds for 1.000, until 2.000
                  2.000                           Ping
                                                  holds for 1.000, until 3.000
                  3.000                           Pong
                                                  holds for 1.000, until 4.000
                  4.000                           Ping
                                                  holds for 1.000, until 5.000
                  5.000                           Pong
                                                  holds for 1.000, until 6.000
                  6.000                           Ping
                                                  holds for 1.000, until 7.000
                  7.000                           Pong
                                                  holds for 1.000, until 8.000
                  8.000                           Ping
                                                  holds for 1.000, until 9.000
                  9.000                           Pong
                                                  holds for 1.000, until 10.000
```

Listing 5-7: Die Trace-Datei “PingPong process.trc”

```
                              Clock Time = 10.000
*******************************************************************************
*                                                                             *
*                         Experiment: PingPong process                        *
*                                                                             *
*                                    Report                                   *
*                                                                             *
*******************************************************************************


                              Clock Time = 10.000
*******************************************************************************
*                                                                             *
*                           Model: PingPong P-Model                           *
*                                                                             *
*******************************************************************************

PingPong P-Model reset at: 0.000


                                    Counts
                                    ------

Title           (Re)set    Obs
------------------------------
Contacts          0.000     10

                              Clock Time = 10.000
*******************************************************************************
*                                                                             *
*                       End of Model: PingPong P-Model                        *
*                                                                             *
*******************************************************************************
```

Listing 5-8: Die Report-Datei “PingPong process.rpt”

5.1.3 PingPong (ereignisorientiert)

Die ereignisorientierte Sicht auf das Ping-Pong-Modell betrachtet den Ball als Entity (Klasse `Ball`). Listing 5-9 zeigt die Definition der Klasse `Ball`. In diesem einfachen Beispiel sind keinerlei benutzerdefinierte Attribute für das Entity erforderlich. Das Hin- und Herspielen des Balls wird mit Hilfe von zwei Ereignistypen modelliert (Klassen `Ping` und `Pong`). Dabei entspricht der Ballkontakt mit jedem der beiden Schläger je einem Ereignis.

```
 1. class Ball : public Entity
 2. {
 3.     public:
 4.         Ball (Model& owner)
 5.             :   Entity (owner, "Ball")
 6.             {}
 7. };
```

Listing 5-9: Die Klasse `Ball` als Unterklasse von `Entity`

5.1.3.1 Die Ereignistypen “Ping” und “Pong”

Der Ball wird zunächst auf der Ping-Seite ins Spiel gebracht, d.h. es erfolgt der erste Ballkontakt. Dies wird durch ein Ereignis der Klasse `Ping` abgebildet. Tritt ein Ping-Ereignis ein, bedeutet das, daß der Ball vom Ping-Schläger geschlagen wird. Die Reaktion darauf ist, neben der Aktualisierung des Zählers, der Kontakt mit dem Pong-Schläger auf der anderen Seite, aber erst nach der Netzüberquerung, die eine Zeiteinheit dauert. D.h. es muß ein Ereignis der Klasse Pong erzeugt und für eine Zeiteinheit später vorgemerkt werden. Die Reaktion auf solch ein Pong-Ereignis ist gewissermaßen gespiegelt: Nach der Aktualisierung des Zählers wird ein neues Ping-Ereignis erzeugt und vorgemerkt.

```
 1. class Ping : public Event
 2. {
 3.     public:
 4.         Ping (Model& owner, Count& c)
 5.             :   Event   (owner, "Ping"),
 6.                 counter (c)
 7.             {}
 8.
 9.     protected:
10.         virtual void    EventRoutine (Entity& ball);
11.
12.     private:
13.         Count&  counter;
14. };
15.
16.
17. class Pong : public Event
18. {
19.     public:
20.         Pong (Model& owner, Count& c)
21.             :   Event   (owner, "Pong"),
22.                 counter (c)
23.             {}
24.
25.     protected:
26.         virtual void    EventRoutine (Entity& ball);
27.
28.     private:
29.         Count&  counter;
30. };
```

Listing 5-10: Die Ereignisklasse `Ping` und `Pong`

In Listing 5-10 sind die beiden Ereignisklassen `Ping` und `Pong` als Unterklassen der Klasse `Event` deklariert. In den Zeilen 10 und 26 sind die beiden Ereignisroutinen als Methoden der Ereignisklassen deklariert. Da in den Ereignisroutinen der Zähler aktualisiert werden muß, benötigen die Ereignisklassen einen Verweis darauf (`counter` in den Zeilen 13 und 29). Dieser Verweis wird im jeweiligen Konstruktor initialisiert (Zeilen 6 und 22), der dafür einen Parameter vom Typ `Count&` (Verweis auf ein Zählerobjekt) benötigt. Ferner werden in den Zeilen 5 und 21 die Ereignisse mit “Ping” bzw. “Pong” benannt.

```
 1. void Ping::EventRoutine (Entity& ball)
 2. {
 3.     counter.Update();
 4.     ball.Schedule (1, *new Pong (GetModel(), counter));
 5.     DeleteOnTermination();
 6. }
 7.
 8.
 9. void Pong::EventRoutine (Entity& ball)
10. {
11.     counter.Update();
12.     ball.Schedule (1, *new Ping (GetModel(), counter));
13.     DeleteOnTermination();
14. }
```

Listing 5-11: Die Ereignisroutinen zu Ping- und Pong-Ereignissen

Listing 5-11 gibt die beiden Ereignisroutinen wieder. Beide unterscheiden sich inhaltlich nur in einer Zeile (4 und 12), in der ein Ereignis des jeweils anderen Typs erzeugt und vorgemerkt wird. Daher wird hier nur eine der beiden Ereignisroutinen beschrieben. In Zeile 3 wird zunächst der Zähler aktualisiert. Zeile 4 merkt für den Ball in einer Zeiteinheit ein neues Pong-Ereignis vor, das mittels `new` erzeugt wird. Diesem muß das zugehörige Modell übergeben werden, das mit Hilfe der Modellkomponentenmethode des Ping-

Ereignisses `GetModel` ermittelt wird. Außerdem wird der Verweis auf den Zähler (`counter`) weitergereicht. In Zeile 5 wird mittels `DeleteOnTermination` signalisiert, daß das Ereignis nach Ende der Ereignisroutine von der Simulationssteuerung gelöscht werden soll, da es nicht mehr benötigt wird, da bei Eintreten des nächsten Pong-Ereignisses ein neues Ping-Ereignis erzeugt wird.

5.1.3.2 Modell und Experiment

Das Modell unterscheidet sich von der prozeßorientierten Variante nur durch den Klassennamen, den Namen des Modells und die Implementierung der Methode `DoInitialSchedules` (Listing 5-12).

```
 1. class PingPongModelE : public Model
 2. {
 3.     public:
 4.         PingPongModelE (Model* owner)
 5.             :   Model       (owner, "PingPong E-Model"),
 6.                 contacts    (*this, "Contacts")
 7.             {}
 8.
 9.     protected:
10.         virtual void    DoInitialSchedules();
11.
12.         Count   contacts;
13. };
14.
15.
16. void PingPongModelE::DoInitialSchedules ()
17.     {
18.         Event* firstPing = new Ping (*this, contacts);
19.         firstPing->Schedule (0, *new Ball (*this));
20.     }
```

Listing 5-12: Die Klasse der ereignisorientierten Ping-Pong-Modelle

In Zeile 18 in Listing 5-12 wird zunächst das erste Ping-Ereignis (`firstPing`) erzeugt. Dabei wird ihm der Zähler übergeben (`contacts`). Zeile 19 merkt dieses Ereignis für den Zeitpunkt 0 zusammen mit einem neuen Ball vor. Da der Ball keine Attribute hat, benötigt er für seine Initialisierung lediglich das Modell (`*this`). Im Gegensatz zu den Ereignisroutinen in Listing 5-11 wird hier die Methode `Schedule` des Ereignisses aufgerufen und ein Entity (der Ball) übergeben. Beide Verfahren führen zum selben Ergebnis: Ereignis und Entity werden zusammen auf die Ereignisliste gesetzt (“Ping-Kontakt von Ball 1”).

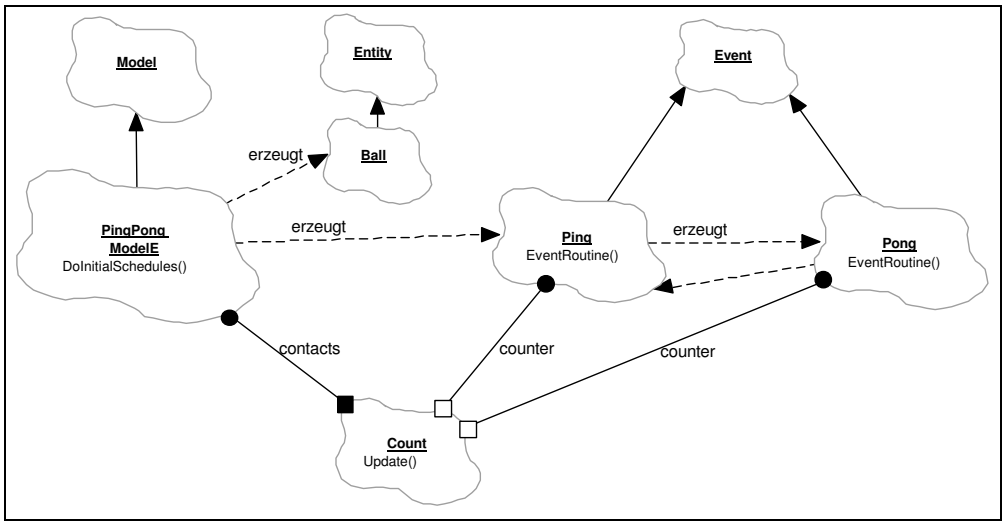

Abbildung 5-3: Das ereignisorientierte Ping-Pong-Modell

Die Beziehungen zwischen den Klassen verdeutlicht Abbildung 5-3. Durch die gestrichelten Pfeile ist sehr gut zu sehen, daß das Modell sowohl ein `Ball`- als auch ein `Ping`-Objekt erzeugt. Die zyklische Erzeugungsbeziehung zwischen den beiden Ereignisklassen `Ping` und `Pong` spiegelt das Bild vom Motor für die ereignisorientierte Simulation wieder[62]. Beide Ereignisklassen sind außerdem mit dem `Count`-Objekt verbunden, das physikalisch im Modell enthalten ist.

Wie für das Prozeßorientierte Modell sei auch hier das Hauptprogramm wiedergegeben (Listing 5-13), das sich jedoch nur in den Zeilen 3 und 4 von dem in Listing 5-6 unterscheidet.

```
 1. void PingPongE()
 2. {
 3.     Experiment* pe = new Experiment ("PingPong event");
 4.     Model*      pm = new PingPongModelE (0);
 5.
 6.     pe->TraceOn ();
 7.     pe->Stop    (10);
 8.     pe->Start   ();
 9.     pe->Report  ();
10.
11.     delete pm;
12.     delete pe;
13. }
```

Listing 5-13: Das Hauptprogramm für das ereignisorientierte Ping-Pong-Modell

Listing 5-14 zeigt den Trace in seiner vollständigen Darstellung. Der Report gleicht bis auf die Namen von Modell und Experiment dem in Listing 5-8.

62 Vgl. Abschnitt 2.2.1

```
                                  Clock Time = 0.000
*******************************************************************************************
*                                                                                         *
*                               Experiment: PingPong event                                *
*                                                                                         *
*                                        Trace                                            *
*                                                                                         *
*******************************************************************************************


Model              Time Event        Entity       Action(s)
-------------------------------------------------------------------------------------------
PingPong Ev       0.000 ---          ---          schedules 'Ping 1' of 'Ball 1' now
                        Ping 1       Ball 1       schedules 'Pong 1' of it at 1.000
                  1.000 Pong 1                    schedules 'Ping 2' of it at 2.000
                  2.000 Ping 2                    schedules 'Pong 2' of it at 3.000
                  3.000 Pong 2                    schedules 'Ping 3' of it at 4.000
                  4.000 Ping 3                    schedules 'Pong 3' of it at 5.000
                  5.000 Pong 3                    schedules 'Ping 4' of it at 6.000
                  6.000 Ping 4                    schedules 'Pong 4' of it at 7.000
                  7.000 Pong 4                    schedules 'Ping 5' of it at 8.000
                  8.000 Ping 5                    schedules 'Pong 5' of it at 9.000
                  9.000 Pong 5                    schedules 'Ping 6' of it at 10.000
```

Listing 5-14: Die Trace-Datei “PingPong event.trc”

5.2 Beispielmodelle mit DESMO-C

Das Modell eines Fertigungssystems aus [Page91] liegt in dieser Arbeit in der ereignis- und der transaktionsorientierten Version vor. Im Gegensatz zur Modula-Version wurde hier zunächst die allgemeine Jobshop-Modellklasse `JobShop` eingerichtet, die Gemeinsamkeiten der unterschiedlichen Versionen zusammenfaßt. Dadurch muß z.B. der Mechanismus zur Darstellung der Routen durch den Maschinenpark nicht in jeder Modellversion erneut implementiert werden. Die Repräsentation von Aufträgen und Maschinen muß jedoch in den Unterklassen der Klasse `JobShop` erfolgen (`E_JobShop` und `T_JobShop`).

Zum Jobshop-Modell sei an dieser Stelle angemerkt, daß eine exakte Vergleichbarkeit der Ergebnisse zu [Page91] nicht gegeben ist, da die Zufallszahlenströme in einer anderen Reihenfolge angelegt sind. Dies liegt daran, daß in [Page91] Zufallszahlenströme für die Bedienzeiten nach Auftragsarten sortiert angelegt werden, während dies in der C++-Version maschinenorientiert abläuft, da die Zufallszahlenströme hier als Elemente der Maschinengruppe modelliert werden. Zum direkten Vergleich muß also in der Modula-Version nur die FOR-Schleife für die Initialisierung andersherum geschachtelt werden.

Bei Platzreservierungs- und Hafenmodell konnte ebenfalls die Vererbung eingesetzt werden, um ausgehend von einem Grundmodell ein oder mehrere Varianten zu modellieren. Dabei stellt jede Variante eine Erweiterung seiner Oberklasse dar. Beim Platzreservierungsmodell ist in der ersten Variante ein zusätzlicher Kundentyp “Bürokunde” eingerichtet. Die zweite Variante erweitert die erste Variante um den Aspekt der Pausen von Bedienkräften. Das Grundmodell des Hafens wird in einer Unterklasse um die Berücksichtigung der Gezeiten erweitert.

5.2.1 Das Hauptprogramm

In den folgenden Abschnitten werden die Beispielmodelle vorgestellt. Den Modellen selbst fehlt jedoch der Teil zur Durchführung eines Experiments. In diesem Abschnitt wird ein Programm vorgestellt, daß auf die in den folgenden Abschnitten dargestellten Modelle zurückgreift. Im Programm werden alle implementierten Modelle in einem Menü aufgelistet, so daß der Anwender ein Modell auswählen kann, mit dem er experimentieren möchte.

Nach dem Ende eines Experiments zeigt das Programm wieder das Modellmenü. Ein zusätzlicher Menüpunkt ist die Laufzeitmessung für alle Modelle. Dabei wird zu jedem Experiment ein Name und die zur Durchführung benötigte Zeit durch Tabulator getrennt in die Datei “benchmark.txt” geschrieben, so daß eine anschließende Weiterverarbeitung der Ergebnisse in anderen Programmen möglich ist.

Um ein Experiment mit einem Modell durchzuführen, wird jeweils ein neues Objekte der Klasse `Experiment` mit einem entsprechenden Namen erzeugt. Eine Ausnahme von dieser Technik stellt das Fährenmodell dar, das ein Beispiel für ein alternatives Verfahren ist. Anstatt direkt ein Objekt der Klasse `Experiment` zu erzeugen, wird hier eine Unterklasse gebildet, deren Konstruktor das Experiment sofort durchführt. Daher reicht es in diesen Fällen aus, das jeweilige Experiment zu erzeugen. Es kann sofort danach wieder gelöscht werden.

main.cc

```
// Dateiname   : main.cc
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// ---------------------------------------------------------------------------
// Haupt-Programm:
//      - Experimentieren mit den Beispielmodellen
//      - Laufzeitmessung
// ---------------------------------------------------------------------------

#include "experime.h"   // Klasse Experiment

// ---------------------------------------------------------------------------
// Funktionen, die Experimente mit den Modellen durchfuehren

enum jobShopVersion { event, transaction };

bool chooseModel ();
void jobshop     (jobShopVersion version,   bool readParam = true);
void ferry       (unsigned version,         bool readParam = true);
void rail        (unsigned version,         bool readParam = true);
void harbour     (unsigned version,         bool readParam = true);
void quarry      (bool readParam = true);

// ---------------------------------------------------------------------------
// Zeitmessung

#include <time.h>

time_t t1, t2, t3;

void StartTime      ();
void StopTime       (ostream& os);
void DetermineSpeed ();

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

void main ()
{
    while (chooseModel())
    {}
}

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

bool chooseModel ()
{
    cout << "\nBitte wählen Sie ein Modell: \n"
                "\n"
            "  1 - JobShop-Modell, ereignisorientiert\n"
            "  2 - JobShop-Modell mit Ressourcen (transaktionsorientiert)\n"
                "\n"
            "  3 - Fährenmodell\n"
            "  4 - Fährenmodell mit direkt kooperierenden Prozessen\n"
                "\n"
            "  5 - Platzreservierung\n"
            "  6 - Platzreservierung mit Bürokunden\n"
            "  7 - Platzreservierung mit Bürokunden und Pausen\n"
                "\n"
            "  8 - Hafenmodell\n"
            "  9 - Hafenmodell mit Gezeiten\n"
                "\n"
            " 10 - Steinbruchmodell\n"
                "\n"
            " 11 - Zeitmessung über alle Modelle\n"
                "\n"
            "  0 - Ende\n"
                "\n"
            ">";

    unsigned choice;
    cin >> choice;

    switch (choice)
    {
        case  1: jobshop (event);       break;
        case  2: jobshop (transaction); break;
        case  3: ferry   (1);           break;
        case  4: ferry   (2);           break;
        case  5: rail    (1);           break;
        case  6: rail    (2);           break;
        case  7: rail    (3);           break;
```

```
        case  8: harbour (1);           break;
        case  9: harbour (2);           break;
        case 10: quarry  ();            break;
        case 11: DetermineSpeed  ();    break;
        default:
            return false;
    }

    return true;
}

// ------------------------------------------------------------------------
// Jobshop-Modelle

#include "jobevent.h"
#include "jobtrans.h"

void jobshop (jobShopVersion version, bool readParam)
{
    Experiment* experiment;
    JobShop*    model;

    switch (version)
    {
        case event:         experiment = new Experiment ("jobevent");
                            model      = new E_JobShop  ();
                            break;
        case transaction:   experiment = new Experiment ("jobtrans");
                            model      = new T_JobShop  ();
                            break;
        default:
            return;
    }

    SimTime     simPeriod   = 2920; // 365 Tage a 8 Stunden
    SimTime     tracePeriod =    8; // 1 Tag

    experiment->Out()   << "Experiment " << experiment->QuotedName() << endl;
    experiment->Out()   << "Laenge des Simulationslaufes  : " << simPeriod
                        << endl << endl;

    if (readParam && model->ReadParameters())
    {
        experiment->Out()   << "Laenge des Simulationslaufes   ("
                            << simPeriod << "): ";
        experiment->In()    >> simPeriod;
    }

    experiment->TraceOn  ();
    experiment->TraceOff (tracePeriod);
    experiment->Stop     (simPeriod);
    experiment->Start    ();
    experiment->Report   ();

    delete model;
    delete experiment;
}

// ------------------------------------------------------------------------
// Fährenmodelle

#include "ferry.h"
#include "ferry2.h"

void ferry (unsigned version, bool readParam)
{
    switch (version)
    {
        case 1: delete new FerryExperiment;     break;
        case 2: delete new Ferry2Experiment;    break;
        default:
            return;
    }
}

// ------------------------------------------------------------------------
// Platzreservierungsmodelle

#include "rail_proc1.h"
#include "rail_proc2.h"
#include "rail_proc3.h"

void rail (unsigned version, bool readParam)
{
    Experiment* experiment;
    RailModel*  model;

    if (version > 0 && version <= 3)
        experiment = new Experiment ("rail_proc" + String(version));
```

```
    switch (version)
    {
        case 1:     model = new RailModel  (); break;
        case 2:     model = new RailModel1a(); break;
        case 3:     model = new RailModel1b(); break;
        default:
            return;
    }

    SimTime     simPeriod   = 360000;
    SimTime     tracePeriod =    500;

    experiment->Out()   << "Experiment " << experiment->QuotedName() << endl;
    experiment->Out()   << "Laenge des Simulationslaufes  : " << simPeriod
                        << endl << endl;

    if (readParam && model->ReadParameters())
    {
        experiment->Out()   << "Laenge des Simulationslaufes   ("
                            << simPeriod << "): ";
        experiment->In()    >> simPeriod;
    }

    experiment->TraceOn  ();
    experiment->TraceOff (tracePeriod);
    experiment->Stop     (simPeriod);
    experiment->Start    ();
    experiment->Report   ();

    delete model;
    delete experiment;
}

// ----------------------------------------------------------------------------
// Hafenmodelle

#include "harbour1.h"
#include "harbour2.h"

void harbour (unsigned version, bool readParam)
{
    Experiment*     experiment;
    HarbourModel*   model;

    if (version > 0 && version <= 2)
        experiment = new Experiment ("harbour" + String(version));

    switch (version)
    {
        case 1: model = new HarbourModel            (); break;
        case 2: model = new HarbourWithTideModel    (); break;
        default:
            return;
    }

    SimTime     simPeriod   = 8760; // 365 Tage a 24 Stunden
    SimTime     tracePeriod =  200;

    experiment->Out()   << "Experiment " << experiment->QuotedName() << endl;
    experiment->Out()   << "Laenge des Simulationslaufes  : " << simPeriod
                        << endl << endl;

    if (readParam && model->ReadParameters())
    {
        experiment->Out()   << "Laenge des Simulationslaufes   ("
                            << simPeriod << "): ";
        experiment->In()    >> simPeriod;
    }

    experiment->TraceOn  ();
    experiment->TraceOff (tracePeriod);
    experiment->Stop     (simPeriod);
    experiment->Start    ();
    experiment->Report   ();

    delete model;
    delete experiment;
}

// ----------------------------------------------------------------------------
// Steinbruchmodell

#include "quarry.h"

void quarry (bool readParam)
{
    Experiment*     experiment  = new Experiment ("quarry");
    QuarryModel*    model       = new QuarryModel();
```

```
    SimTime     simPeriod       = 240;  // 30 Tage a 8 Stunden
    SimTime     tracePeriod     =   5;

    experiment->Out()   << "Experiment " << experiment->QuotedName() << endl;
    experiment->Out()   << "Laenge des Simulationslaufes  : " << simPeriod
                        << " h" << endl << endl;

    if (readParam && model->ReadParameters())
    {
        experiment->Out()   << "Laenge des Simulationslaufes   ("
                            << simPeriod << "h ): ";
        experiment->In()    >> simPeriod;
    }

    experiment->TraceOn  ();
    experiment->TraceOff (tracePeriod);
    experiment->Stop     (simPeriod);
    experiment->Start    ();
    experiment->Report   ();

    delete model;
    delete experiment;
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Zeitmessung

void StartTime ()
{
    t1 = time (NULL);
}

// ------------------------------------------------------------------------

#include <iomanip>

void StopTime (ostream& os)
{
    t2 = time (NULL);
    t3 = t2 - t1;

    tm*  dt     = localtime (&t3);
    char ofill  = os.fill  ('0');
    long oflags = os.flags (ios::fixed | ios::right);

    os << dt->tm_min << ":" << setw(2)
       << dt->tm_sec;

    os.flags (oflags);
    os.fill  (ofill);
}

// ------------------------------------------------------------------------

void DetermineSpeed ()
{
    const n = 10;
    char name [n][255] = {  "jobevent",
                            "jobtrans",
                            "ferry",
                            "ferry2",
                            "rail_proc1",
                            "rail_proc2",
                            "rail_proc3",
                            "harbour1",
                            "harbour2",
                            "quarry"};

    ofstream os ("benchmark.txt");  // Ausgabedatei, 2 Spalten

    for (unsigned choice = 1; choice <= n; ++choice)
    {
        os << name [choice -1] << '\t';
        StartTime();
        switch (choice)
        {
            case  1: jobshop (event,        false); break;
            case  2: jobshop (transaction,  false); break;
            case  3: ferry   (1,            false); break;
            case  4: ferry   (2,            false); break;
            case  5: rail    (1,            false); break;
            case  6: rail    (2,            false); break;
            case  7: rail    (3,            false); break;
            case  8: harbour (1,            false); break;
            case  9: harbour (2,            false); break;
            case 10: quarry  (              false); break;
            default:
                return;
```

```
        }
        StopTime (os);
        os << endl;
    }
}

// ----------------------------------------------------------------------------
```

5.2.2 Simulation eines Fertigungssystems

5.2.2.1 Gemeinsamkeiten der Versionen

jobshop.h

```
// Dateiname   : jobshop.h
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************)
(*                                                                           *)
(*      Dateiname:        JobEvent.MOD                                       *)
(*                                                                           *)
(*      Projektleiter:    Prof. Dr.-Ing. B. Page                             *)
(*                        Fachbereich Informatik der Universität Hamburg     *)
(*                                                                           *)
(*      Autoren:          R. Boelckow, A. Heymann, H. Liebert                *)
(*                                                                           *)
(*      Programminhalt:   "Jobshop-Modell"                                   *)
(*                        Ereignisorientiert                                 *)
(*                                                                           *)
(*****************************************************************************)
(*      Erstellt in TopSpeed Modula-2/87 Vers. 1.17 unter MS-DOS             *)
(*****************************************************************************)
(* Das Modell dient zur Engpassanalyse einer Fertigungsanlage, die aus       *)
(* mehreren Maschinengruppen mit einer unterschiedlichen Anzahl jeweils      *)
(* identischer Maschinen besteht.                                            *)
(* Die Maschinengruppen bearbeiten verschiedene Produktarten, die unter-     *)
(* schiedliche Verarbeitungsreihenfolgen durchlaufen. Die Teilauftraege      *)
(* werden in FIFO-Warteschlangen der einzelnen Maschinengruppen eingereiht,  *)
(* um dort auf Zuteilung einer freien Maschine zu warten.                    *)
(* Die Zwischenankunftszeit der Auftraege ist neg.-exponentiell verteilt,    *)
(* die Bedienzeiten sind k-Erlang (k=2) verteilt.                            *)
(* (Details s. Initialisierungsteil).                                        *)
(*                                                                           *)
(* Die Eingabedaten sind:                                                    *)
(*    - Simulationsdauer (in Tagen),                                         *)
(*    - Maschinenzahl pro Gruppe.                                            *)
(* Die geforderten Ausgabedaten sind:                                        *)
(*    - mittl. Verweildauer pro Auftragsart,                                 *)
(*    - mittl. Verweildauer ueber alle Auftraege,                            *)
(*    - mittl. Warteschlangenlaenge pro Maschinengruppe,                     *)
(*    - mittl. Auslastung -""-                                               *)
(*    - mittl. Wartezeiten in den Warteschlangen.                            *)
(*****************************************************************************/

#ifndef JOBSHOP_H
#define JOBSHOP_H

// ---------------------------------------------------------------------------

#include "model.h"

#include "intdist.h"
#include "realdist.h"

#include "queue.h"
#include "res.h"

#include "accumula.h"
#include "tally.h"

// ---------------------------------------------------------------------------

const NJobTypes         = 3; // Anz. der versch. Auftragsarten
const NMachineGroups    = 5; // Anz. der Maschinengruppen
const MaxRoutLength     = 9; // max. Laenge einer Vorgabe-Route

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Ein Weg durch den Maschinenpark

class Route
{   // zur Konstruktion einer Route durch den Maschinenpark
    // enthaelt die Nummer einer Maschine [0...n-1] sowie auf
    // den weiteren Verlauf der Route ('next')
    // Extend() erweitert die Route um eine Maschinengruppennr. am Ende
```

```
    public:
                Route (unsigned MGroupIndex)
                    :   mGroupIndex (MGroupIndex),
                        next        (0)
                    {}
                ~Route () { if (next) delete next; }

        void    Extend (unsigned groupIndex)
                    {
                        if (next)
                            next->Extend (groupIndex);
                        else
                            next = new Route (groupIndex);
                    }

        unsigned    GetMGroupIndex ()   { return mGroupIndex; }
        Route*      Next()              { return next; }

    private:
        unsigned    mGroupIndex;
        Route*      next;
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Das Job-Shop-Modell

class JobShop : public Model
{   // 'JobShop' dient als Oberklasse fuer die verschiedenen Implementierungen
    // des Fertigungssystem-Modells. So ist hier der Routing-Mechanismus
    // bereits implementiert, der in allen Varianten benoetigt wird.
    // Jobs, Maschinengruppen und deren Beziehungen werden in den Unterklassen
    // definiert. Modellparameter, ZZ-Stroeme und Statistik sind als protected
    // deklariert, so dass sie in Unterklassen zugreifbar sind.
    public:

        virtual ~JobShop ();

        virtual String      Description() const;
                                // Beschreibung des Modells fuer den Report
        virtual bool        ReadParameters();
                                // liest Parameter und liefert true, oder
                                // false, wenn Vorgabe benutzt werden sollen

                Route&      GetRoute (unsigned jobType) const;
                                /*  wird von den Jobs benoetigt und liefert
                                    zu einer Auftragsart eine Route.
                                    'jobType' von 0 bis NJobTypes - 1 */
                void        SetRoute (unsigned jobType, int* route);
                                /*  richtet fuer jobType eine neue Route ein:
                                    route ist ein Array von Maschinengruppen-
                                    Indices. Die Numerierung beginnt bei 0.
                                    Negative Werte terminieren die Route.
                                    z.B.: {0,4,3,2,-1}
                                    'jobType' von 0 bis NJobTypes - 1 */

                void        UpdateJobDelay (unsigned jobType);
                                /*  aktualisiert zu der Auftragsart 'jobType'
                                    die Statistikobjekte.
                                    'jobType' von 0 bis NJobTypes - 1 */

    protected:
                JobShop (   ValueSupplier*  jobDelaySupplier,
                                // muß Verzoegerung eines Jobs
                                // liefern koennen
                                // wird im Destruktor geloescht!
                            const   String& name = "JobShop",
                            Model*  owner = 0);
                                // falls als Submodell verwendet

        // Modellparameter
        const   SimTime             meanArrivalTime;
                                        // Zwischenankunftszeit der Auftraege
        const   SimTime             hoursPerDay;
                                        // taegl. Arbeitszeit der Maschinen
                unsigned            machinesPerGroup [NMachineGroups];
                                        // Anzahl Maschinen pro Gruppe

        // Statistik
                ValueSupplier*      jobDelaySupplier;
                                        // Beobachtungsgroesse
                Tally               overAllJobDelay;
                                        // Verweilzeit aller Auftraege
                Tally*              jobDelay [NJobTypes];
                                        // Verweilzeit pro Auftragsart
        // ZZ-Stroeme
                IntDistEmpirical    jobTypeStream;
                RealDistExponential arrivalStream;
```

```
                bool        canChangeParam;

        static  SimTime     defaultServiceTime  [NMachineGroups] [NJobTypes];

    private:
        // die Auftragsrouten fuer jede Auftragsart
                Route*      route               [NJobTypes];

        // Vorgabewerte
        static  unsigned    defaultMachPerGroup [NMachineGroups];
                                                              // = {3,2,4,3,1}
        static  int         defaultRoute        [NJobTypes][MaxRoutLength];
};

// ---------------------------------------------------------------------------

#endif  // JOBSHOP_H
```

jobshop.cc

```
// Dateiname   : jobevent.cc
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// ---------------------------------------------------------------------------

#include "jobshop.h"

#include <iomanip.h>

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// static-Komponenten von JobShopTrans:

unsigned    JobShop::defaultMachPerGroup [NMachineGroups] =
                {3,2,4,3,1};
SimTime     JobShop::defaultServiceTime [NMachineGroups][NJobTypes] =
                {   {0.6,   0.8,    0.7},
                    {0.85,  0.0,    1.2},
                    {0.5,   0.75,   1.0},
                    {0.0,   1.1,    0.9},
                    {0.5,   0.0,    0.25}   };
int         JobShop::defaultRoute [NJobTypes][MaxRoutLength] =
                {   {2,0,1,4,-1},
                    {3,0,2,-1},
                    {1,4,0,3,2,-1}  };

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

JobShop::JobShop (  ValueSupplier*  JobDelaySupplier,
            const   String&         name,
                    Model*          owner)
    :   Model               (owner, name),
        meanArrivalTime     (0.25),
        hoursPerDay         (8.0),
        jobDelaySupplier    (JobDelaySupplier),
        overAllJobDelay     (*this, "Ttl Job Dly", *JobDelaySupplier),
        jobTypeStream       (*this, "Job Type"),
        arrivalStream       (*this, "InterArrival", meanArrivalTime.Time()),
        canChangeParam      (true)
    {
        // Zahl der Maschinen pro Gruppe aus Vorgabe uebernehmen
        for (unsigned i = 0; i < NMachineGroups; i++)
            machinesPerGroup [i] = defaultMachPerGroup [i];

        // Empirische Verteilung mit Daten versorgen
        jobTypeStream.AddEntry (0, 0.3);
        jobTypeStream.AddEntry (1, 0.8);
        jobTypeStream.AddEntry (2, 1.0);

        // Tallies erzeugen
        for (unsigned i = 0; i < NJobTypes; i++)
            jobDelay [i] = new Tally (*this,
                                      String("Job Delay ") + (i+1),
                                      *jobDelaySupplier);

        // Routen einrichten
        for (unsigned i = 0; i < NJobTypes; i++)
        {
            route [i] = 0;
            SetRoute (i, defaultRoute [i]);
        }
    }
```

```
// ------------------------------------------------------------------------

JobShop::~JobShop ()
    {
        // Routen loeschen
        for (unsigned i = 0; i < NJobTypes; i++)
        {
            delete route [i];
            delete jobDelay [i];
        }
        delete jobDelaySupplier;
    }

// ------------------------------------------------------------------------

String JobShop::Description () const
    {
        strstream os;
        os  << "    MGroup  MsPerGrp\n"
            << "--------------------\n"
            << setiosflags(ios::right);
        for (unsigned i = 0; i < NMachineGroups; i++)
            os  << setw(10) << i+1
                << setw(10) << machinesPerGroup [i]
                << endl;

        os << ends;
        return os;
    }

// ------------------------------------------------------------------------

bool JobShop::ReadParameters ()
{
    if (!canChangeParam)
        return false;

    Out() << Description() << endl << endl;
    Out() << "Standardparameter benutzen (j/n)? ";
    char c;
    In() >> c;
    if (c == 'j' || c == 'J')
        return false;   // Nicht einlesen, sondern Standard verwenden
    else
    {
        Out()   << "\n*** Modell eines Fertigungssystems ***\n\n";

        for (unsigned i = 0; i < NMachineGroups; i++)
        {
            Out() << "Anzahl Maschinen in Gruppe " << i+1 << ": ";
            In()  >> machinesPerGroup [i];
        }
    }
    return true;
}

// ------------------------------------------------------------------------

Route& JobShop::GetRoute (unsigned jobType) const
    {
        return *route [jobType];
    }

// ------------------------------------------------------------------------

void JobShop::SetRoute (unsigned jobType, int* mGroups)
    {   // Numerierung bei mGroups beginnt bei 0, negative Werte terminieren!
        if (mGroups == 0 || mGroups[0] < 0)
            return;
        if (route [jobType])
            // alte Route loeschen
            delete route [jobType];

        route [jobType] = new Route (mGroups[0]);

        unsigned i = 1;
        while (mGroups[i] >= 0)
        {
            route [jobType]->Extend (mGroups[i]);
            ++i;
        }
    }

// ------------------------------------------------------------------------

void JobShop::UpdateJobDelay (unsigned jobType)
    {
        jobDelay [jobType]->Update();
```

```
        overAllJobDelay.Update();
    }
// ------------------------------------------------------------------------
```

5.2.2.2 Ereignisorientierte Version

jobevent.h

```
// Dateiname   : jobevent.h
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// Inhalt      : ereignisorientiertes Modell des Fertigungssystems
//
// Beschreibung siehe Datei: jobshop.h
//
// ---------------------------------------------------------------------------

#ifndef JOBEVENT_H
#define JOBEVENT_H

// ---------------------------------------------------------------------------

#include "jobshop.h"

#include "queue.h"

#include "accumula.h"
#include "tally.h"

// ---------------------------------------------------------------------------

class JobDelay;

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Das Modell

class E_JobShop : public JobShop
{
    public:
    // -----------------------------------------------------------------------
    // Maschinengruppen im ereignisorientierten Ansatz

    class MachineGroup : public ModelComponent
    {   // Maschinengruppe stellt Statistik (Accumulate) und ZZ-Stroeme fuer
        // die Bedienzeiten zur Verfuegung sowie Methoden zum Belegen und
        // Freigeben von einzelen Maschinen der Gruppe

        // -------------------------------------------------------------------
        // Beobachtungsgroesse fuer Auslastung der Maschinengruppen

        class UtilSuppl : public ValueSupplier
        {   // Wertlieferant fuer das Accumulate-Objekt
            public:
                        UtilSuppl (Model& owner, const MachineGroup& mg)
                            :   ValueSupplier   (owner),
                                mGroup          (mg)
                            {}

                virtual double  Value() const;
            private:
                const MachineGroup& mGroup;
        };
        // Ende Beobachtungsgroesse fuer Auslastung der Maschinengruppen
        // -------------------------------------------------------------------

        public:
                    MachineGroup (  Model& owner,
                                    unsigned group,     // Gruppennummer
                                    unsigned NMachines, // An. der Maschinen
                                    SimTime meanServiceTime [NJobTypes]);
                                        // mittl. Bedienzeit je Auftragsart

                    ~MachineGroup ();

                    unsigned    FreeMachines () const;
                                    // liefert die Anzahl freier Maschinen

                    unsigned    NumOfMachines () const;
                                    // liefert die Gesamtzahl der Maschinen

                    void        UseMachine ();
                                    // belegt eine Maschine und
                                    // aktualisiert die Statistik
                                    // Vorbedingung:
```

```
                                    //   FreeMachines() > 0

                    void        ReleaseMachine ();
                                    // gibt eine Maschine frei und
                                    // aktualisiert die Statistik
                                    // Vorbedingung:
                                    //   FreeMachines() < NumOfMachines()

            SimTime             NewServiceTime (unsigned jobType);
                                    // Zeit fuer Jobbearbeitung sampeln

            Queue       waitingJobs; // wartende Jobs

        private:
            unsigned    nOfMachines; // Gesamtanzahl der Maschinen
            unsigned    freeMachines;// Anzahl verfuegbarer Maschinen
            UtilSuppl   utilSuppl;   // Beobachtungsgroesse
            Accumulate  utilization; // Statistikobjekt dafuer
            RealDist*   serviceStream [NJobTypes]; // Bedienzeiten
    };
    // Ende Maschinengruppen im ereignisorientierten Ansatz
    // ------------------------------------------------------------------------
                E_JobShop ( Model*  owner = 0,
                                    // falls als Submodell verwendet
                    const   String& name = "JobShop",
                            SimTime meanArrivalTime = 0.25,
                                    // Zwischenankunftszeit der Auftraege
                            SimTime hoursPerDay     = 8.0);
                                    // taegl. Arbeitszeit der Maschinen
        virtual ~E_JobShop ();

                Entity&     NewJob();
                                    // erzeugt einen neuen Auftrag

                SimTime     NewArrivalTime();
                                    // erzeugt eine neue Zwischenankunftszeit

                MachineGroup&   GetMGroup (unsigned index) const;
                                    // liefert die entspr. Maschinengruppe

        virtual bool        ReadParameters();
                                    // liest Parameter und liefert true, oder
                                    // false, wenn Vorgabe benutzt werden sollen

    protected:
        virtual void        DoInitialSchedules();
                                    // merkt den ersten Job fuer t = 0.0 vor

    private:
        // die Maschinengruppen
        MachineGroup*   machineGroup [NMachineGroups];
};

// Ende des Modells
// ------------------------------------------------------------------------

#endif  // JOBEVENT_H
```

jobevent.cc

```
// Dateiname   : jobevent.cc
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// ------------------------------------------------------------------------

#include "jobevent.h"

#include "entity.h"
#include "event.h"
#include <assert.h>
#include <iomanip.h>

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Maschinengruppe

E_JobShop::MachineGroup::MachineGroup (
            Model&   owner,
            unsigned group,     // interne Gruppennummer
            unsigned NMachines, // Anzahl der Maschinen
            SimTime  meanServiceTime [NJobTypes])
                // mittlere Bedienzeit fuer jede Auftragsart
    :   ModelComponent (owner, String("Mach Group ") + (group +1)),
```

```
        waitingJobs    (owner, String("Jobs Wait ")  + (group +1)),
        nOfMachines    (NMachines),
        freeMachines   (NMachines),
        utilSuppl      (owner, *this),
        utilization    (owner,      String("Util Grp ") + (group +1),
                        utilSuppl,  false)
            // automatische Statistik zu jedem Simulationszeipunkt
    {
        // dynamische Erzeugung der ZZ-Stroeme
        for (int jType = 0; jType < NJobTypes; ++jType)
            serviceStream [jType] = new RealDistErlang
                (owner,
                 String("Service ") + (jType +1) + ' ' + (group +1),
                 2,
                 meanServiceTime [jType].Time());
    }

// ----------------------------------------------------------------------------

E_JobShop::MachineGroup::~MachineGroup ()
    {
        // loeschen der dynamischen Komponenten (ZZ-Stroeme)
        for (int i = 0; i < NJobTypes; i++)
            delete serviceStream [i];
    }

// ----------------------------------------------------------------------------

unsigned E_JobShop::MachineGroup::FreeMachines () const
    {
        return freeMachines;
    }

// ----------------------------------------------------------------------------

unsigned E_JobShop::MachineGroup::NumOfMachines () const
    {
        return nOfMachines;
    }

// ----------------------------------------------------------------------------

void E_JobShop::MachineGroup::UseMachine ()
    {
        const char* where = "E_JobShop::MachineGroup::UseMachine";

        if (freeMachines <= 0)
            Warning ("Attempt to use a busy machine of group " + QuotedName(),
                     where);
        else
        {
            --freeMachines;
            utilization.Update();
        }
    }

// ----------------------------------------------------------------------------

void E_JobShop::MachineGroup::ReleaseMachine ()
    {
        const char* where = "E_JobShop::MachineGroup::ReleaseMachine";

        if (freeMachines >= nOfMachines)
            Warning ("Attempt to release a busy machine to group " +
                     QuotedName() + ", where all machines are idle",
                     where);
        else
        {
            ++freeMachines;
            utilization.Update();
        }
    }

// ----------------------------------------------------------------------------

SimTime E_JobShop::MachineGroup::NewServiceTime (unsigned jobType)
    {
        if (jobType >= NJobTypes)
        {   // Ungueltige Auftragsart!
            Warning ("illegal jobtype '" + String(jobType) + "'",
                     "MachineGroup::SampleServiceTime() called ",
                     String(NJobTypes -1) + " will be used instead",
                     "jobType must be between 0 and " + String(NJobTypes -1));
            jobType = NJobTypes -1;
        }
        return serviceStream [jobType]->Sample();
    }

// ----------------------------------------------------------------------------
```

```
// ------------------------------------------------------------------------

class E_Job : public Entity
{   // Auftraege als Entities im ereignisorientierten Modell
    public:
                E_Job (E_JobShop&   owner, unsigned jobType)
                    :   Entity          (owner, "Job"),
                        jobShop         (owner),
                        type            (jobType),
                        route           (&owner.GetRoute (type)),
                        arrivalTime     (CurrentTime())
                    {}

                SimTime Delay () const { return CurrentTime() - arrivalTime;}

                bool    Ready () const { return route == 0; }

                E_JobShop::MachineGroup&    GetMGroup () const;
                                            // liefert die aktuelle M.-Gruppe

                void                        NextMGroup ();
                                            // setzt den Job auf die naechste
                                            // M.-Gruppe an

                unsigned Type() const   { return type; }    // Job-Typ
    private:
        E_JobShop&      jobShop;
        unsigned        type;
        Route*          route;
        SimTime         arrivalTime;
};

// ------------------------------------------------------------------------

E_JobShop::MachineGroup& E_Job::GetMGroup() const
    {
        const char* where = "E_Job::GetMGroup";

        if (Ready())    // route == 0
        {
            FatalError ("a ready job has no machine to go to", where);
            assert (false);
        }

        return jobShop.GetMGroup (route->GetMGroupIndex());
    }

// ------------------------------------------------------------------------

void E_Job::NextMGroup()
    {
        const char* where = "Job::NextMGroup";

        if (Ready())    // route == 0
        {
            FatalError ("a ready job has no next machine", where);
            assert (false);
        }

        route = route->Next();
    }

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Beobachtungsgroesse fuer Maschinenauslastung

double E_JobShop::MachineGroup::UtilSuppl::Value () const
{
    return      double(mGroup.NumOfMachines() - mGroup.FreeMachines())
            /   mGroup.NumOfMachines() * 100;
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Ereignis-Typen

class JobShopEvent : public Event
{   // allgemeiner im JobShop verwendeter Ereignistyp
    public:
        JobShopEvent (E_JobShop& owner, const String name)
            :   Event   (owner, name),
                jobShop (owner)
            {}

    protected:
        E_JobShop&  jobShop;    // damit Unterklassen Zugriff haben
};

// ------------------------------------------------------------------------
```

```
class JobCreation : public ExternalEvent
{   // Erzeugen eines neuen Jobs
    public:
        JobCreation (E_JobShop& owner)
            :   ExternalEvent   (owner, "Creation"),
                jobShop         (owner)
            {}

        virtual void    ExternalEventRoutine ();

    protected:
        E_JobShop&  jobShop;    // damit Unterklassen Zugriff haben
};

// ----------------------------------------------------------------------------

class JobArrival : public JobShopEvent
{   // Ankunft eines Auftrages an einer Maschinengruppe
    public:
        JobArrival (E_JobShop& owner)
            :   JobShopEvent    (owner, "Arrival")
            {}

        virtual void    EventRoutine (Entity& e);
};

// ----------------------------------------------------------------------------

class JobDeparture : public JobShopEvent
{   // Ende der Bedienung in einer Maschinengruppe
    public:
        JobDeparture (E_JobShop& owner)
            :   JobShopEvent    (owner, "Departure")
            {}

        virtual void    EventRoutine (Entity& e);
};

// ----------------------------------------------------------------------------
// Ereignis-Routinen

void JobCreation::ExternalEventRoutine ()
{
    // Das Ereignis (this) fuer den Nachfolger-Job erneut vormerken
    Schedule (jobShop.NewArrivalTime());

    // Neuen Jobs erzeugen:
    Entity& job = jobShop.NewJob();

    // Ereignis fuer Ankunft an der ersten Maschinengruppe erzeugen:
    Event* arrival = new JobArrival (jobShop);

    // Ankunftsereignis zusammen mit job vormerken;
    arrival->Schedule (0.0, job);
}

// ----------------------------------------------------------------------------

void JobArrival::EventRoutine (Entity& entity)
{
    E_Job& job = (E_Job&) entity;   // DownCast
    assert (!job.Ready());

    E_JobShop::MachineGroup& mGroup = job.GetMGroup();

    mGroup.waitingJobs.Insert (job);

    if (mGroup.FreeMachines() > 0)
    {
        mGroup.waitingJobs.Remove (job);
        mGroup.UseMachine();
        SimTime serviceTime = mGroup.NewServiceTime (job.Type());
        Event* departure = new JobDeparture (jobShop);
        departure->Schedule (serviceTime, job);
    }
    // Ereignis kann geloescht werden:
    DeleteOnTermination();
}

// ----------------------------------------------------------------------------

void JobDeparture::EventRoutine (Entity& entity)
{
    E_Job& job = (E_Job&) entity;   // DownCast
    assert (!job.Ready());  // wenn, dann wird er hier erst fertig

    // alte Maschine verlassen:
    E_JobShop::MachineGroup& mGroup = job.GetMGroup();
```

```
    if (mGroup.waitingJobs.Empty())
    {
        // Maschine freigeben:
        mGroup.ReleaseMachine ();
        // Ereignis kann nach Ende der Routine geloescht werden:
        DeleteOnTermination();
    }
    else
    {   // Bearbeitungsende des naechsten wartenden Auftrages vormerken:
        E_Job& nextJob = (E_Job&) mGroup.waitingJobs.First();
        mGroup.waitingJobs.Remove (nextJob);
        SimTime serviceTime = mGroup.NewServiceTime (nextJob.Type());
        // JobDeparture (this) mit nextJob vormerken:
        Schedule (serviceTime, nextJob);
    }

    // Job zur naechsten Maschine:
    job.NextMGroup();

    if (job.Ready())
    {   // es gibt keine naechste Maschine
        jobShop.UpdateJobDelay (job.Type());
        job.Delete();
    }
    else
    {
        //E_JobShop::MachineGroup& mGroup = job.GetMGroup();
        Event* arrival = new JobArrival (jobShop);
        arrival->Schedule (0.0, job);
    }
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Beobachtungsgroesse fuer Accumulates aus JobShop-Oberklasse

class JobDelay : public ValueSupplier
{
    public:
                JobDelay (Model& owner) : ValueSupplier (owner) {}

        virtual double  Value() const;
};

// ------------------------------------------------------------------------

double JobDelay::Value() const
    {
        E_Job* job = dynamic_cast<E_Job*> (&CurrentEntity());
        assert (job);
        return job->Delay().Time();
    }


// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

E_JobShop::E_JobShop (  Model*  owner,
                const   String& name,
                        SimTime MeanArrivalTime,
                        SimTime HoursPerDay)
    :   JobShop (new JobDelay (*this), name, owner)
    {
        // Maschinengruppen erzeugen
        for (unsigned mGroup = 0; mGroup < NMachineGroups; ++mGroup)
            machineGroup [mGroup] =
                new MachineGroup (
                    *this,
                    mGroup,
                    machinesPerGroup [mGroup],
                    defaultServiceTime [mGroup]);

    }

// ------------------------------------------------------------------------

E_JobShop::~E_JobShop ()
    {
        for (unsigned mGroup = 0; mGroup < NMachineGroups; ++mGroup)
            delete machineGroup [mGroup];
    }

// ------------------------------------------------------------------------

void E_JobShop::DoInitialSchedules ()
    {
        canChangeParam = false;
```

```
        // neues Anmkunftsereignis erzeugen:
        ExternalEvent* arrival = new JobCreation (*this);

        // und Ereigis vormerken:
        arrival->Schedule (0.0);
    }

// ------------------------------------------------------------------------

Entity& E_JobShop::NewJob ()
    {   // neuen Job erzeugen:
        return *new E_Job (*this, jobTypeStream.Sample());
    }

// ------------------------------------------------------------------------

SimTime E_JobShop::NewArrivalTime ()
    {   // neuen Zwischenankunftszeit erzeugen:
        return arrivalStream.Sample();
    }

// ------------------------------------------------------------------------

E_JobShop::MachineGroup& E_JobShop::GetMGroup (unsigned index) const
    {
        const char* where = "E_JobShop::GetMGroup";

        if (index >= NMachineGroups)
        {
            Warning ("invalid machine group number", where,
                     "the highest group will be returned");
            index = NMachineGroups - 1;
        }
        return *machineGroup [index];
    }

// ------------------------------------------------------------------------

bool E_JobShop::ReadParameters ()
{
    if (!JobShop::ReadParameters())
        return false;

    for (unsigned i = 0; i < NMachineGroups; i++)
    {
        delete machineGroup [i];
        machineGroup [i] = new MachineGroup (*this, i,
                                             machinesPerGroup [i],
                                             defaultServiceTime [i]);
    }
    return true;
}

// ------------------------------------------------------------------------
```

jobevent.rpt

```
                            Clock Time = 2920.000
*******************************************************************************
*                                                                             *
*                            Experiment: jobevent                             *
*                                                                             *
*                                  Report                                     *
*                                                                             *
*******************************************************************************


                            Clock Time = 2920.000
*******************************************************************************
*                                                                             *
*                               Model: JobShop                                *
*                                                                             *
*******************************************************************************

JobShop reset at: 0.000

    MGroup  MsPerGrp
--------------------
         1         3
         2         2
         3         4
         4         3
         5         1

                                Distributions
```

```
                                    -------------

Title           (Re)set    Obs Type              P a r a m e t e r s       Seed
-------------------------------------------------------------------------------
Job Type          0.000  11630 I-Empirical       0            0.3000  33427485
                                                 1            0.8000
                                                 2            1.0000
InterArrival      0.000  11630 Neg-Expon.        0.2500                22276755
Service 1 1       0.000   3513 k-Erlang          0.6000       2        46847980
Service 2 1       0.000   5794 k-Erlang          0.8000       2        43859043
Service 3 1       0.000   2284 k-Erlang          0.7000       2        64042082
Service 1 2       0.000   3483 k-Erlang          0.8500       2        44366385
Service 2 2       0.000      0 k-Erlang          0.0000       2        41357879
Service 3 2       0.000   2287 k-Erlang          1.2000       2        11320893
Service 1 3       0.000   3516 k-Erlang          0.5000       2         6528269
Service 2 3       0.000   5792 k-Erlang          0.7500       2        47478000
Service 3 3       0.000   2283 k-Erlang          1.0000       2        46802881
Service 1 4       0.000      0 k-Erlang          0.0000       2        59224073
Service 2 4       0.000   5799 k-Erlang          1.1000       2        22046052
Service 3 4       0.000   2283 k-Erlang          0.9000       2        65931384
Service 1 5       0.000   3482 k-Erlang          0.5000       2        25261809
Service 2 5       0.000      0 k-Erlang          0.0000       2        17756070
Service 3 5       0.000   2286 k-Erlang          0.2500       2        45177506

                                      Queues
                                      ------

Title           (Re)set    Obs    Qmax    Qnow      Qavg.   Zeros   avg.Wait
-------------------------------------------------------------------------------
Jobs Wait 1       0.000  11591      39       6      8.952    1161      2.255
Jobs Wait 2       0.000   5770      67      55     21.906     221     10.973
Jobs Wait 3       0.000  11591      13       0      0.704    6753      0.177
Jobs Wait 4       0.000   8082      62       3     15.505     614      5.602
Jobs Wait 5       0.000   5768      16       0      1.785    1421      0.904

                                   Accumulates
                                   -----------

Title           (Re)set    Obs       Mean    Std.Dev        Min         Max
-------------------------------------------------------------------------------
Util Grp 1        0.000   2319     94.843     17.052      0.000     100.000
Util Grp 2        0.000    440     97.551     13.212      0.000     100.000
Util Grp 3        0.000  13504     71.752     30.318      0.000     100.000
Util Grp 4        0.000   1225     96.234     14.687      0.000     100.000
Util Grp 5        0.000   2841     79.884     40.086      0.000     100.000

                                     Tallies
                                     -------

Title           (Re)set    Obs       Mean    Std.Dev        Min         Max
-------------------------------------------------------------------------------
Ttl Job Dly       0.000  11555     15.134      8.837      0.913      48.314
Job Delay 1       0.000   3481     16.742      8.338      1.326      38.827
Job Delay 2       0.000   5792     10.691      5.543      0.913      28.347
Job Delay 3       0.000   2282     23.960      8.915      3.511      48.314

                              Clock Time = 2920.000
*******************************************************************************
*                                                                             *
*                              End of Model: JobShop                          *
*                                                                             *
*******************************************************************************
```

5.2.2.3 Version mit Ressoucen (transaktionsorientiert)

jobtrans.h

```
// Dateiname   : jobtrans.h
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// Beschreibung siehe Datei: jobshop.h
//
// ---------------------------------------------------------------------------

#ifndef JOBTRANS_H
#define JOBTRANS_H

// ---------------------------------------------------------------------------

#include "jobshop.h"

#include "res.h"

// ---------------------------------------------------------------------------

class T_JobShop : public JobShop
{
    public:
    // -----------------------------------------------------------------------
    // Maschinengruppen im ereignisorientierten Ansatz
    class MachineGroup : public Res
    {   // Maschinengruppe erbt von Ressource, damit lassen sich die Methoden
        // Acquire() und Release() direkt anwenden
        // zusaetzlich stehen Statistik (Accumulate) und ZZ-Stroeme fuer
        // die Bedienzeiten zur Verfuegung

        // -------------------------------------------------------------------
        // Beobachtungsgroesse fuer mittlere Warteschlangenlaenge vor
        // der Maschinengruppen

        class QLength : public ValueSupplier
        {   // Wertlieferant fuer das Accumulate-Objekt
            public:
                        QLength (Model& owner, Res& r)
                            :   ValueSupplier   (owner),
                                res             (r)
                            {}

                virtual double  Value() const { return res.Length(); }
            private:
                Res&    res;    // Res, desseb WS-Laenge beobachtet wird
        };
        // Ende Beobachtungsgroesse
        // -------------------------------------------------------------------

        public:
                    MachineGroup (  Model& owner,
                                    unsigned group,     // Gruppennummer
                                    unsigned NMachines, // Anz. der Maschinen
                                    SimTime meanServiceTime [NJobTypes]);
                                        // mittl. Bedienzeit je Auftragsart

                    ~MachineGroup ();

            SimTime             NewServiceTime (unsigned jobType);
                                    // Zeit fuer Jobbearbeitung sampeln

        private:
            QLength     qLength;    // Warteschlangenlänge vor der M.-gruppe
            Accumulate  avgQlen;    // Statistikobjekt dafuer
            RealDist*   serviceStream [NJobTypes]; // Bedienzeiten
    };
    // Ende Maschinengruppen im ereignisorientierten Ansatz
    // -----------------------------------------------------------------------

                T_JobShop ( Model*  owner = 0,
                                // falls als Submodell verwendet
                            unsigned machPerGroup[NMachineGroups] = 0,
                                // Anzahl Maschinen pro Gruppe (Array)
                    const   String& name = "JobShop T");

        virtual ~T_JobShop ();

                void        ActivateNewJob ();
```

```
                                // erzeugt einen neuen Auftrag und merkt ihn
                                // vor wird von den Auftraegen aufgerufen

                MachineGroup&   GetMGroup (unsigned index) const;
                                    // liefert die entspr. Maschinengruppe

        virtual bool        ReadParameters();
                                // liest Parameter und liefert true, oder
                                // false, wenn Vorgabe benutzt werden sollen

    protected:
        virtual void        DoInitialSchedules();
                                // merkt den ersten Job fuer t = 0.0 vor
    private:
        // die Maschinengruppen
                MachineGroup*       machineGroup [NMachineGroups];
};

// ------------------------------------------------------------------------

#endif  // JOBTRANS_H
```

jobtrans.cc

```
// Dateiname   : jobtrans.cc
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// ------------------------------------------------------------------------

#include "jobtrans.h"

#include "process.h"

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

T_JobShop::MachineGroup::MachineGroup
        (   Model& owner,
            unsigned group,     // interne Gruppennummer
            unsigned NMachines, // Anzahl der Maschinen
            SimTime meanServiceTime [NJobTypes])
                                // mittlere Bedienzeit fuer jede Auftragsart
    :   Res     (owner, String("Mach Group ") + (group +1), NMachines),
        qLength (owner, *this),
        avgQlen (owner, String("Queue Len ") + (group +1), qLength, true)
            // automatische Statistik zu jedem Simulationszeipunkt
    {
        // dynamische Erzeugung des ZZ-Strom-Arrays
        for (int jType = 0; jType < NJobTypes; jType++)
            serviceStream [jType] =
                new RealDistErlang  (owner,
                                     String("Service ") + (jType +1) + ' '
                                                        + (group +1),
                                     2,
                                     meanServiceTime [jType].Time());
    }

// ------------------------------------------------------------------------

T_JobShop::MachineGroup::~MachineGroup ()
    {
        // loeschen der dynamischen Komponenten (ZZ-Stroeme)
        for (int i = 0; i < NJobTypes; i++)
            delete serviceStream [i];
    }

// ------------------------------------------------------------------------

SimTime T_JobShop::MachineGroup::NewServiceTime (unsigned jobType)
    {
        if (jobType >= NJobTypes)
        {   // Ungueltige Auftragsart!
            Warning ("illegal jobtype '" + String(jobType) + "'",
                     "MachineGroup::SampleServiceTime() called ",
                     String(NJobTypes -1) + " will be used instead",
                     "jobType must be between 0 and " + String(NJobTypes -1));
            jobType = NJobTypes -1;
        }
        return serviceStream [jobType]->Sample();
    }

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
```

```
class T_Job : public Process
{   // Auftraege als aktive Modell-Entities
    public:
                T_Job (T_JobShop&   owner, unsigned jobType)
                    :   Process         (owner, "Job"),
                        type            (jobType),
                        route           (&owner.GetRoute (type)),
                        arrivalTime     (0.0)
                    {};

                SimTime ArrivalTime() const { return arrivalTime; }

        virtual void    LifeCycle();

    private:
        unsigned    type;
        Route*      route;
        SimTime     arrivalTime;
};

// ----------------------------------------------------------------------------

void T_Job::LifeCycle()
    {
        T_JobShop&  jobShop = (T_JobShop&)GetModel();
        // Nachfolger erzeugen:
        jobShop.ActivateNewJob();

        arrivalTime = CurrentTime();

        Route*  r = route;
        while (r)
        {
            T_JobShop::MachineGroup&
                mGroup = jobShop.GetMGroup(r->GetMGroupIndex());

            // Maschine exklusiv belegen:
            mGroup.Acquire (1);

            // Bearbeitung:
            Hold (mGroup.NewServiceTime (type));

            // Maschine freigeben:
            mGroup.Release (1);

            r = r->Next();
        }
        jobShop.UpdateJobDelay (type);
        DeleteOnTermination();
    }

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------

class T_JobDelay : public ValueSupplier
{   // Verweilzeit eines Jobs
    public:
                T_JobDelay (Model& owner) : ValueSupplier (owner) {}

        virtual double  Value() const
            {
                T_Job& job = (T_Job&)CurrentProcess();
                return (CurrentTime() - job.ArrivalTime()).Time();
            }
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------

T_JobShop::T_JobShop (Model*    owner,
                      unsigned MachinesPerGroup [NMachineGroups],
                const   String& name)
    :   JobShop (new T_JobDelay(*this), name, owner)
    {
        // Maschinengruppen erzeugen
        for (unsigned mGroup = 0; mGroup < NMachineGroups; ++mGroup)
            machineGroup [mGroup] =
                new MachineGroup (
                    *this,
                    mGroup,
                    machinesPerGroup [mGroup],
                    defaultServiceTime [mGroup]);
    }

// ----------------------------------------------------------------------------

T_JobShop::~T_JobShop ()
    {
```

```
        for (unsigned i = 0; i < NMachineGroups; i++)
            delete machineGroup [i];
    }

// ------------------------------------------------------------------------

void T_JobShop::DoInitialSchedules ()
    {
        canChangeParam = false;
        (new T_Job (*this, jobTypeStream.Sample()))->Activate (0.0);
    }

// ------------------------------------------------------------------------

T_JobShop::MachineGroup& T_JobShop::GetMGroup (unsigned index) const
    {
        const char* where = "T_JobShop::GetMGroup";

        if (index >= NMachineGroups)
        {
            Warning ("invalid machine group number", where,
                     "the highest group will be returned");
            index = NMachineGroups - 1;
        }
        return *machineGroup [index];
    }

// ------------------------------------------------------------------------

void T_JobShop::ActivateNewJob ()
    {
        // neuen Job erzeugen:
        T_Job* newJob = new T_Job (*this, jobTypeStream.Sample());
        // und Aktivierung vormerken:
        newJob->Activate (arrivalStream.Sample());
    }

// ------------------------------------------------------------------------

bool T_JobShop::ReadParameters ()
{
    if (!JobShop::ReadParameters())
        return false;

    for (unsigned i = 0; i < NMachineGroups; i++)
        machineGroup [i] ->ChangeLimit (machinesPerGroup [i]);

    return true;
}

// ------------------------------------------------------------------------
```

jobtrans.rpt

```
                              Clock Time = 2920.000
********************************************************************************
*                                                                              *
*                              Experiment: jobtrans                            *
*                                                                              *
*                                    Report                                    *
*                                                                              *
********************************************************************************


                              Clock Time = 2920.000
********************************************************************************
*                                                                              *
*                                Model: JobShop T                              *
*                                                                              *
********************************************************************************

JobShop T reset at: 0.000

    MGroup  MsPerGrp
--------------------
         1         3
         2         2
         3         4
         4         3
         5         1


                                Distributions
                                -------------

Title           (Re)set    Obs Type             P a r a m e t e r s      Seed
```

Job Type	0.000	11631	I-Empirical	0	0.3000	33427485
				1	0.8000	
				2	1.0000	
InterArrival	0.000	11630	Neg-Expon.	0.2500		22276755
Service 1 1	0.000	3513	k-Erlang	0.6000	2	46847980
Service 2 1	0.000	5794	k-Erlang	0.8000	2	43859043
Service 3 1	0.000	2284	k-Erlang	0.7000	2	64042082
Service 1 2	0.000	3483	k-Erlang	0.8500	2	44366385
Service 2 2	0.000	0	k-Erlang	0.0000	2	41357879
Service 3 2	0.000	2287	k-Erlang	1.2000	2	11320893
Service 1 3	0.000	3516	k-Erlang	0.5000	2	6528269
Service 2 3	0.000	5792	k-Erlang	0.7500	2	47478000
Service 3 3	0.000	2283	k-Erlang	1.0000	2	46802881
Service 1 4	0.000	0	k-Erlang	0.0000	2	59224073
Service 2 4	0.000	5799	k-Erlang	1.1000	2	22046052
Service 3 4	0.000	2283	k-Erlang	0.9000	2	65931384
Service 1 5	0.000	3482	k-Erlang	0.5000	2	25261809
Service 2 5	0.000	0	k-Erlang	0.0000	2	17756070
Service 3 5	0.000	2286	k-Erlang	0.2500	2	45177506

Resources

Title	(Re)set	Users	Limit	Min	Now	Usage [%]	avg.Wait	QMaxL
Mach Group 1	0.000	11588	3	0	0	94.843	2.255	39
Mach Group 2	0.000	5768	2	0	0	97.551	10.973	67
Mach Group 3	0.000	11589	4	0	2	71.752	0.177	13
Mach Group 4	0.000	8079	3	0	0	96.234	5.602	62
Mach Group 5	0.000	5767	1	0	0	79.884	0.904	16

Accumulates

Title	(Re)set	Obs	Mean	Std.Dev	Min	Max
Queue Len 1	0.000	54410	8.952	8.247	0.000	39.000
Queue Len 2	0.000	54410	21.906	16.604	0.000	67.000
Queue Len 3	0.000	54410	0.704	1.471	0.000	13.000
Queue Len 4	0.000	54410	15.505	14.191	0.000	62.000
Queue Len 5	0.000	54410	1.785	2.288	0.000	16.000

Tallies

Title	(Re)set	Obs	Mean	Std.Dev	Min	Max
Ttl Job Dly	0.000	11555	15.134	8.837	0.913	48.314
Job Delay 1	0.000	3481	16.742	8.338	1.326	38.827
Job Delay 2	0.000	5792	10.691	5.543	0.913	28.347
Job Delay 3	0.000	2282	23.960	8.915	3.511	48.314

Clock Time = 2920.000

```
*****************************************************************************
*                                                                           *
*                          End of Model: JobShop T                          *
*                                                                           *
*****************************************************************************
```

5.2.3 Simulation einer Fähre

5.2.3.1 Version mit Puffern (Bin)

ferry.h

```
// Dateiname   : ferry.h
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************)
(*                                                                           *)
(*      Dateiname:        Ferry.MOD                                          *)
(*                                                                           *)
(*      Projektleiter:    Prof. Dr.-Ing. B. Page                             *)
(*                        Fachbereich Informatik der Universität Hamburg     *)
(*                                                                           *)
(*      Autoren:          R. Boelckow, A. Heymann, H. Liebert                *)
(*                                                                           *)
(*      Programminhalt:   Simulation einer Fähre                             *)
(*                        vgl. [Bir79a], S. 69                               *)
(*                                                                           *)
(*                                                                           *)
(*****************************************************************************)
(*      Erstellt in TopSpeed Modula-2/87 Vers. 1.17 unter MS-DOS             *)
(*****************************************************************************)
(* Die einzige Verbindung zwischen einer kleinen Insel und dem Festland be- *)
(* steht aus einer Autofaehre. Die Faehre liegt ueber Nacht am Festland,    *)
(* Betriebsbeginn ist um 7.00 Uhr, Betriebsschluss etwa um 21.45 Uhr. Die   *)
(* Faehre kann maximal 6 Pkw aufnehmen. Bei Ankungt an einem Kai werden     *)
(* zunaechst alle Autos von der Faehre gefahren, um dann wartende Pkw fuer  *)
(* die naechste Überfahrt aufzunehmen. Die Faehre startet, wenn die Ladeka- *)
(* pazitaet erreicht ist oder keine weiteren Fahrzeuge warten. Die Zwischen-*)
(* ankunftszeit der Pkw ist neg.-expon. verteilt, die Faehrzeit normalver-  *)
(* teilt, die Be- und Entladezeit pro Auto konstant.                        *)
(*****************************************************************************/

#ifndef FERRY_H
#define FERRY_H

// ---------------------------------------------------------------------------

#include "model.h"

class Bin;

// ---------------------------------------------------------------------------

class FerryModel : public Model
{
    public:
        enum    SideT { mainland = 0, island, stations};

                FerryModel (    Model*  owner = 0,
                                    // falls als Submodell verwendet
                        const   String& name = "Ferry");

        virtual ~FerryModel ();

    protected:
        virtual void        DoInitialSchedules();
                                // merkt den ersten Job fuer t = 0.0 vor
    private:
        Bin*            arrivedCars [stations];
};

// ---------------------------------------------------------------------------

#include "experime.h"

class FerryExperiment : public Experiment
{
    public:
        FerryExperiment ()
            :   Experiment ("Ferry"),
                model ()
            {
                TraceOn();
```

```
                Start (7 * 60);
                TraceOff ();
                Report();
            }
    private:
        FerryModel  model;
};

// ---------------------------------------------------------------------------

#endif  // FERRY_H
```

ferry.cc

```
// Dateiname   : ferry.cc
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// ---------------------------------------------------------------------------

#include "ferry.h"

#include "process.h"

#include "realdist.h"

#include "bin.h"

#include "count.h"
#include "tally.h"

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

class Arrival : public Process
{
    public:
                Arrival (Model& owner, const String& side, Bin& bin)
                    :   Process         (owner, "Arrival"),
                        arrivedCars     (bin),
                        arrivalStream   (owner, side, 1.0 / 0.15)
                    {}

        virtual void    LifeCycle()
                            {
                                for (;;)
                                {
                                    Hold (arrivalStream.Sample ());
                                    arrivedCars.Give (1);
                                }
                            }
    private:
        Bin&                arrivedCars;
        RealDistExponential arrivalStream;
};

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

class Ferry : public Process
{
    // -----------------------------------------------------------------------

    class FerryLoad : public ValueSupplier
    {
        public:
                    FerryLoad (Ferry& f)
                        :   ValueSupplier   (f.GetModel()),
                            ferry           (f)
                        {}
            virtual double  Value() const { return ferry.CurrentLoad(); }
        private:
            Ferry&  ferry;
    };

    // -----------------------------------------------------------------------

    public:
                Ferry (FerryModel& owner, Bin* arrivals [FerryModel::stations])
                    :   Process     (owner, "Ferry"),
                        cars        (0),
                        crossing    (owner, "Crossing", 8.0, 0.5),
                        trips       (owner, "Trips"),
                        empties     (owner, "Empties"),
```

```
                        ferryLoad   (*this),
                        load        (owner, "Avg. Load", ferryLoad)
                    {
                        arrivedCars [0] = arrivals [0];
                        arrivedCars [1] = arrivals [1];
                    }

        virtual void    LifeCycle();

                double  CurrentLoad() const { return cars; }
    private:
        unsigned        cars;

        RealDistNormal  crossing;
        Bin*            arrivedCars [FerryModel::stations];

        // Statistik
        Count           trips,
                        empties;
        FerryLoad       ferryLoad;
        Tally           load;
};

// ---------------------------------------------------------------------------

void Ferry::LifeCycle()
    {
        while (CurrentTime() < double (21 * 60 + 45))   // bis 21:45 h
        {
            for (unsigned side = FerryModel::mainland;
                 side < FerryModel::stations; side++)
            {
                cars = 0;
                while (cars < 6 && arrivedCars [side]->Avail() > 0)
                {
                    arrivedCars [side]->Take (1);
                    Hold (0.5);
                    cars++;
                }
                load.Update();
                if (cars == 0)
                    empties.Update();
                TraceNote ("Crossing with " + String(cars) + " cars loaded");

                Hold (crossing.Sample());
                Hold (cars * 0.5);
            }
            trips.Update();
        }
        DeleteOnTermination();
        GetModel().GetExperiment().Stop();
    }

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------

FerryModel::FerryModel (Model* owner, const String& name)
    :   Model (owner, name)
    {
        arrivedCars [mainland]  = new Bin (*this, "Mainland", 3);
        arrivedCars [island  ]  = new Bin (*this, "Island",   1);
    }

// ---------------------------------------------------------------------------

FerryModel::~FerryModel()
    {
        for (unsigned side = mainland; side < stations; side++)
            delete arrivedCars [side];
    }

// ---------------------------------------------------------------------------

void FerryModel::DoInitialSchedules()
    {
        String sideString [stations] = {"Mainland", "Island"};
        const SimTime start = CurrentTime();

        for (unsigned side = mainland; side < stations; side++)
            (new Arrival (*this, sideString [side], *arrivedCars [side]))
                ->Activate (0.0);

        Ferry* f = new Ferry (*this, arrivedCars);
        f->Activate (0.0);
    }

// ---------------------------------------------------------------------------
```

Ferry.rpt

```
                              Clock Time = 1315.296
*******************************************************************************
*                                                                             *
*                               Experiment: Ferry                             *
*                                                                             *
*                                    Report                                   *
*                                                                             *
*******************************************************************************


                              Clock Time = 1315.296
*******************************************************************************
*                                                                             *
*                                 Model: Ferry                                *
*                                                                             *
*******************************************************************************

Ferry reset at: 420.000

Ferry-Model

                                 Distributions
                                 -------------

Title           (Re)set    Obs Type              P a r a m e t e r s      Seed
-------------------------------------------------------------------------------
Mainland        420.000    133 Neg-Expon.        6.6667                33427485
Island          420.000    146 Neg-Expon.        6.6667                22276755
Crossing        420.000     78 Normal            8.0000       0.5000   46847980

                                    Bins
                                    ----

Title           (Re)set  Users   Init    Max    Now    Average   avg.Wait QMaxL
-------------------------------------------------------------------------------
Mainland        420.000    132      3      6      2      1.678      0.000     1
Island          420.000    145      1      7      7      1.705      0.000     1

                                   Counts
                                   ------

Title           (Re)set    Obs
-------------------------------
Trips           420.000     39
Empties         420.000      2

                                   Tallies
                                   -------

Title           (Re)set    Obs       Mean    Std.Dev        Min        Max
-------------------------------------------------------------------------
Avg. Load       420.000     78      3.487      1.634      0.000      6.000

                              Clock Time = 1315.296
*******************************************************************************
*                                                                             *
*                              End of Model: Ferry                            *
*                                                                             *
*******************************************************************************
```

5.2.3.2 Version mit direkt kooperierenden Prozessen

ferry2.h

```
// Dateiname   : ferry2.h
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************)
(*                                                                           *)
(*      Dateiname:        Ferry2.MOD                                         *)
(*                                                                           *)
(*      Projektleiter:    Prof. Dr.-Ing. B. Page                             *)
(*                        Fachbereich Informatik der Universität Hamburg     *)
(*                                                                           *)
(*      Autoren:          R. Boelckow, A. Heymann, H. Liebert                *)
(*                                                                           *)
(*      Programminhalt:   Simulation einer Fähre                             *)
(*                        vgl. [Bir79a], S. 80f.                             *)
(*                                                                           *)
(*                                                                           *)
(*****************************************************************************)
(*      Erstellt in TopSpeed Modula-2/87 Vers. 1.17 unter MS-DOS             *)
(*****************************************************************************)
(* Das Modell unterscheidet sich vom einfachen Faehrmodell (FERRY.MOD) da-   *)
(* durch, dass die Faehre und die Pkw jeweils als direkt kooperierende Pro-  *)
(* zesse formuliert sind. Ausserdem fahren die Autos nach einer normalver-   *)
(* teilten Aufenthaltszeit von der Insel zurueck zum Festland.               *)
(* Das Modell enthaelt zudem zwei Versionen des Entladevorganges:            *)
(* Pkw koennen das Schiff in FIFO- oder LIFO-Reihenfolge verlassen.          *)
(* LIFO: Entladung der Fahrzeuge aus der Fähre nach LIFO-Strategie           *)
(* FIFO: Entladung der Fahrzeuge aus der Fähre nach FIFO-Strategie           *)
(*****************************************************************************/

#ifndef FERRY2_H
#define FERRY2_H

// ---------------------------------------------------------------------------

#include "model.h"

#include "realdist.h"

#include "waitq.h"

#include "count.h"
#include "tally.h"

// ---------------------------------------------------------------------------

class Ferry2Model : public Model
{
    public:
        enum    SideT { mainland = 0, island, stations};

                Ferry2Model (   Model*  owner = 0,
                                    // falls als Submodell verwendet
                        const   String& name = "Ferry2",
                                SimTime timeToWork = 14.75 * 60,
                                    // Zeit für einen Arbeitstag
                                    // nach dieser Zeit wird das Experiment
                                    // vom Modell angehalten
                                bool    fifo = true);
                                    // Entlade-Strategie

        virtual ~Ferry2Model ();

    protected:
        virtual void        DoInitialSchedules();
                                // merkt den ersten Job in dt = 0.0 vor
    private:
        bool                fifo;
        SimTime             timeToWork;
        WaitQueue*          quay [stations];
        RealDistExponential arrivalStream;
        RealDistNormal      crossingStream;
        RealDistNormal      stayStream;
};

// ---------------------------------------------------------------------------
```

```
#include "experime.h"

// ----------------------------------------------------------------------------

class Ferry2Experiment : public Experiment
{
    public:
        Ferry2Experiment ()
            :   Experiment ("Ferry2"),
                model ()
            {
                TraceOn();
                Start (420);
                TraceOff ();
                Report();
            }
    private:
        Ferry2Model model;
};

// ----------------------------------------------------------------------------

#endif  // FERRY2_H
```

ferry2.cc

```
// Dateiname   : ferry2.cc
//
// Datum       : 08.03.1998
//
// Autor       : Thomas Schniewind
//
// ----------------------------------------------------------------------------

#include "ferry2.h"

#include "process.h"
#include "waitq.h"

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------

class Car : public Process
{
    public:
                Car (   Model&      owner,
                        RealDist&   arrival,
                        RealDist&   stay,
                        WaitQueue*  waitQ [Ferry2Model::stations]
                    )
                    :   Process         (owner, "Car"),
                        arrivalStream   (arrival),
                        stayStream      (stay)
                    {
                        for (   unsigned side = Ferry2Model::mainland;
                                side < Ferry2Model::stations; side++)
                            quay [side] = waitQ [side];
                    }

        virtual void    LifeCycle();
    private:
        RealDist&           arrivalStream;
        RealDist&           stayStream;
        WaitQueue*          quay [Ferry2Model::stations];
};

// ----------------------------------------------------------------------------

void Car::LifeCycle()
    {
        Car* nextCar = new Car (GetModel(), arrivalStream, stayStream, quay);
        nextCar->Activate (arrivalStream.Sample());

        quay [Ferry2Model::mainland]->Wait();
        for (unsigned side = Ferry2Model::mainland +1;
             side < Ferry2Model::stations; side++)
        {
            Hold (stayStream.Sample());
            quay [side]->Wait();
        }
        DeleteOnTermination();
    }

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
```

```
class Ferry2 : public Process
{
    // ------------------------------------------------------------------------

    class FerryLoad : public ValueSupplier
    {
        public:
                    FerryLoad (Ferry2& f)
                        :   ValueSupplier   (f.GetModel()),
                            ferry           (f)
                        {}
            virtual double  Value() const { return ferry.CurrentLoad(); }
        private:
            Ferry2& ferry;
    };

    // ------------------------------------------------------------------------

    class Transfer : public ProcessCooperation
    {
        public:
                    Transfer (Ferry2& f)
                        :   ProcessCooperation  (f.GetModel(), "Transfer")
                        {}
            virtual void    Cooperation (Process& master, Process& slave);
    };
    friend class Transfer;

    // ------------------------------------------------------------------------

    public:
                Ferry2 (Ferry2Model&    owner,
                        WaitQueue*      waitQ [Ferry2Model::stations],
                        RealDist&       crossingStream,
                        SimTime&        TimeToWork,
                        bool            FIFO
                    )
                    :   Process     (owner, "Ferry"),
                        fifo        (FIFO),
                        timeToWork  (TimeToWork),
                        cargo       (owner, "Cargo"),
                        crossing    (crossingStream),
                        transfer    (*this),
                        trips       (owner, "Trips"),
                        empties     (owner, "Empties"),
                        ferryLoad   (*this),
                        load        (owner, "Avg. Load", ferryLoad)
                    {
                        for (   side = Ferry2Model::mainland;
                                side < Ferry2Model::stations; side++)
                            quay [side] = waitQ [side];
                        side = Ferry2Model::mainland;
                    }

        virtual void    LifeCycle();

                double  CurrentLoad() const { return cargo.Length(); }
    private:
        unsigned        side;
        bool            fifo;
        SimTime         timeToWork;

        ProcessQueue    cargo;

        RealDist&       crossing;
        WaitQueue*      quay [Ferry2Model::stations];
        Transfer        transfer;

        // Statistik
        Count           trips,
                        empties;
        FerryLoad       ferryLoad;
        Tally           load;
};

// ----------------------------------------------------------------------------

void Ferry2::LifeCycle()
    {
        SimTime finish = CurrentTime() + timeToWork;

        while (CurrentTime() < finish)  // bis 21:45 h
        {
            for (side = Ferry2Model::mainland;
                 side < Ferry2Model::stations; side++)
            {
                if (quay [side]->sEmpty())
                {
```

```
                    load.Update();
                    empties.Update();
                    Hold (crossing.Sample());
                }
                else
                    quay [side]->Cooperate (transfer);
            }
            trips.Update();
        }
        DeleteOnTermination();
        GetModel().GetExperiment().Stop();
    }

// ------------------------------------------------------------------------

void Ferry2::Transfer::Cooperation (Process& master, Process& slave)
    {
        Ferry2&         ferry   = (Ferry2&) master;
        Process&        car     = slave;
        ProcessQueue&   cargo   = ferry.cargo;
        WaitQueue&      quay    = *ferry.quay [ferry.side];

        // Auto in den Laderaum aufnehmen
        cargo.Insert (car);
        Hold (0.5);
        if (cargo.Length() == 6 || quay.sEmpty())
        {   // wenn Laderaum voll oder keine weiteren wartenden Autos:
            // abfahren
            ferry.load.Update();
            Hold (ferry.crossing.Sample());
        }
        else
            quay.Cooperate (ferry.transfer);

        if (ferry.fifo)
        {   // Autos in FIFO-Reihenfolge entladen:
            while (!cargo.Empty())
            {
                Process& car = cargo.First();
                Hold (0.5);
                cargo.Remove (car);
                car.Activate (0.0);
            }
        }
        else
        {   // Autos in LIFO-Reihenfolge entladen:
            Hold (0.5);
            cargo.Remove (car);
        }
    }

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

Ferry2Model::Ferry2Model (Model*  owner, const String& name,
                          SimTime TimeToWork,  bool    FIFO)
    :   Model           (owner, name),
        fifo            (FIFO),
        timeToWork      (TimeToWork),
        arrivalStream   (*this, "Arrival", 1.0 / 0.15),
        crossingStream  (*this, "Crossing", 8.0, 0.5),
        stayStream      (*this, "Stay", 60.0, 30.0)
    {
        quay [mainland] = new WaitQueue (*this, "Mainland");
        for (unsigned side = mainland +1; side < stations; side++)
            quay [side] = new WaitQueue (*this, "Island " + String(side));
    }

// ------------------------------------------------------------------------

Ferry2Model::~Ferry2Model()
    {
        for (unsigned side = mainland; side < stations; side++)
            delete quay [side];
    }

// ------------------------------------------------------------------------

void Ferry2Model::DoInitialSchedules()
    {
        Process* p;

        p = new Car (*this, arrivalStream, stayStream, quay);
        p ->Activate (0.0);

        p = new Ferry2 (*this, quay, crossingStream, timeToWork, fifo);
        p ->Activate (0.0);
    }
```

```
// ------------------------------------------------------------------------
```

Ferry2.rpt

```
                              Clock Time = 1319.250
********************************************************************************
*                                                                              *
*                              Experiment: Ferry2                              *
*                                                                              *
*                                   Report                                     *
*                                                                              *
********************************************************************************


                              Clock Time = 1319.250
********************************************************************************
*                                                                              *
*                                 Model: Ferry2                                *
*                                                                              *
********************************************************************************

Ferry2 reset at: 420.000

                                  Distributions
                                  -------------

Title           (Re)set    Obs Type              P a r a m e t e r s      Seed
------------------------------------------------------------------------------
Arrival         420.000    133 Neg-Expon.         6.6667                33427485
Crossing        420.000     80 Normal             8.0000        0.5000  22276755
Stay            420.000    132 Normal            60.0000       30.0000  46847980

                                    Queues
                                    ------

Title           (Re)set    Obs    Qmax    Qnow       Qavg.  Zeros   avg.Wait
------------------------------------------------------------------------------
Cargo           420.000    255       6       0       3.018      0     10.644

                                    Counts
                                    ------

Title           (Re)set    Obs
------------------------------
Trips           420.000     40
Empties         420.000      8

                                    Tallies
                                    -------

Title           (Re)set    Obs       Mean    Std.Dev        Min        Max
------------------------------------------------------------------------------
Avg. Load       420.000     80      3.188      1.801      0.000      6.000

                                  Wait-Queues
                                  -----------

Title           (Re)set    Obs    Qmax    Qnow       Qavg.  Zeros   avg.Wait
------------------------------------------------------------------------------
Mainland M      420.000    132       1       0       0.000    132      0.000
Mainland S      420.000    132       8       1       1.851      1     12.549

Island 1 M      420.000    123       1       0       0.000    123      0.000
Island 1 S      420.000    123       8       0       1.649      0     12.058


                              Clock Time = 1319.250
********************************************************************************
*                                                                              *
*                             End of Model: Ferry2                             *
*                                                                              *
********************************************************************************
```

5.2.4 Platzreservierungsmodell

5.2.4.1 Grundmodell

rail_proc1.h

```
// Dateiname    : rail_proc1.h
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************)
(*                                                                           *)
(*      Dateiname:        Rail_A1.MOD                                        *)
(*                                                                           *)
(*      Datum:            01.05.1992                                         *)
(*                                                                           *)
(*      Autoren:          Rolf Boelckow, Andreas Heymann,                    *)
(*                        Ralf Kadler,   Hansjoerg Liebert                   *)
(*      Modifiziert von:  Dirk Martinssen fuer                               *)
(*                        Uebungen zu Simulation II, WS 1997/98              *)
(*                        Aufgabe 1                                          *)
(*                                                                           *)
(*      Programminhalt:   Telefonische Eisenbahnreservierung                 *)
(*                        (prozessorientiert)                                *)
(*                                                                           *)
(*****************************************************************************)
(* Das Modell dient zur Engpassanalyse einer Eisenbahnreservierungsstelle.  *)
(* Buchungen koennen sowohl telephonisch als auch durch Kunden im Buero     *)
(* erfolgen. Wartende Buerokunden haben Prioritaet gegenueber Telefonkunden.*)
(* Es stehen 18 Telephonleitungen und 5 Reservierungskraefte zur Verfuegung.*)
(* Kunden werden entweder sofort bedient, warten eine bestimmte Zeit oder   *)
(* oder erhalten keine freie Leitung (Verlustsystem).                        *)
(* Die Zwischenankunftszeiten der Anrufer und die Bedienzeiten sind negativ-*)
(* exponential-, die "Geduld" normalverteilt.                                *)
(* Die Wahrscheinlichkeit fuer die Buchung einer Hin- und Rueckfahrt be-     *)
(* traegt 0.75. In diesem Fall verdoppelt sich die Bedienzeit.               *)
(* Zeitangaben erfolgen auf Sekundenbasis.                                   *)
(*                                                                           *)
(* Eingabedaten:                                                             *)
(*    - Simulationsdauer,                                                    *)
(*    - mittlere Zwischenankunftszeiten                                      *)
(*    - mittlere Bedienzeiten                                                *)
(*    - mittlere "Geduld" und deren Standardabweichung                       *)
(*                                                                           *)
(*    Mittlwert der Zwischenankunftszeiten   :  20 Sekunden                  *)
(*    Mittlwert fuer eine einfache Fahrt     :   1 Minute                    *)
(*    Mittlwert fuer eine Hin- und Rueckfahrt:   2 Minuten                   *)
(*    mittlere Wartezeit eines Kunden        :   4 Minuten                   *)
(*    Abweichung von der Wartezeit           :   2 Minuten                   *)
(*                                                                           *)
(* Ausgabedaten (Mindestanforderung):                                        *)
(*    - mittlere Wartezeit und Warteschlangenlaenge,                         *)
(*    - Anrufe ohne freie Leitung                                            *)
(*    - Anrufe ohne Bedienung nach Ablauf der "Geduldsphase"                 *)
(*                                                                           *)
(*****************************************************************************/

#ifndef RAIL_H
#define RAIL_H

#include "model.h"      // fuer Klasse Model

#include "realdist.h"   // fuer Klasse RealDistExponential & -Normal
#include "booldist.h"   // fuer Klasse BoolDistBernoulli
#include "pqueue.h"     // fuer Klasse ProcessQueue
#include "tally.h"      // fuer Klasse Tally
#include "count.h"      // fuer Klasse Count
#include "process.h"    // fuer Process

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Kunden

class RailModel;    // forward

class Customer : public Process
{   // Klasse fuer allgemeine Kunden
    // Unterklassen koennen das Verhalten durch Ueberschreiben von
```

```
    // 'EnterSystem' und 'LeaveSystem' anpassen
    public:
        Customer    (RailModel&         owner,
                     const String&      name,
                     BoolDist&          TripStream,
                     RealDist&          ServiceStream,
                     RealDist&          PatienceStream,
                     ProcessQueue&      CustQ,
                     ProcessQueue&      ServiceQ,
                     Count&             CustCounter,
                     Count&             ServedNoWait,
                     Count&             AftWaitUnserved,
                     Tally&             NonServedCust,
                     Tally&             NonZWaits);

                SimTime Serve ();
                            // wird vom Bediener beim Beginn
                            // der Bedienung aufgerufen und
                            // liefert die erforderliche Bedienzeit
                SimTime WaitingTime() const { return waitingTime;   }
    protected:
        virtual void    LifeCycle();

        virtual bool    EnterSystem() { return true; }
                            // muß true liefern, wenn der Kunde
                            // in das System gelangen kann
        virtual void    LeaveSystem() {};

        RailModel&          railModel;
    private:
        BoolDist&           tripStream;
        RealDist&           serviceStream;
        RealDist&           patienceStream;
        ProcessQueue&       custQ;
        ProcessQueue&       serviceQ;
        Count&              custCounter;
        Count&              servedNoWait;
        Count&              aftWaitUnserved;
        Tally&              nonServedCust;
        Tally&              nonZWaits;
        bool                roundTrip,
                            iveBeenWaiting,
                            iveBeenServed;
        SimTime             arrivalTime,
                            waitingTime;
};

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Bediener

class Server : public Process
{   // Bediener, der Telefonanrufe bearbeiten kann
    public:
        Server  (RailModel&     owner,
                 ProcessQueue&  ServiceQ,   // fuer untaetige Bediener
                 ProcessQueue&  CallQ);     // fuer wartende Anrufer

                void        WakeUpForService();
                            // wird von Kunden aufgerufen, um untaetige
                            // Bedienkraft zu aktivieren
    protected:
        virtual void        LifeCycle();

        virtual bool        CustomersAreWaiting();
                            // liefert true, wenn Kunden warten
        virtual Customer&   NextCustomer();
                            // liefert den naechsten zu bedinenden Kunden
                            // Fehler, wenn keiner wartet!!!
        virtual void        CleanUpService() {}
                            // wird nach Bearbeitung eines Kunden
                            // aufgerufen
        virtual void        Wait() { Passivate(); }
                            // hiermit setzt sich die Bedienkraft zur
                            // Ruhe, wenn kein Kunde wartet
    private:
        RailModel&      railModel;
        ProcessQueue&   serviceQ;   // fuer untaetige Bediener
        ProcessQueue&   callQ;      // fuer wartende Anrufer
};

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Beobachtungsgroessen

class CustWaitingTime : public ValueSupplier
{
    public:
        CustWaitingTime (RailModel& owner);
```

```
        // Diese Methode wird von den Tally-Objekten aufgerufen
        double  Value () const
            {
                Customer& customer = (Customer&) CurrentProcess();
                return customer.WaitingTime().Time();
            }
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell

class RailModel : public Model
{   // einfaches Platzreservierungsmodell mit Telefon-Bedienkraeften
    public:
            // Zeitbasis: Sekunden
        RailModel (Model* owner =  0, const String& name = "Rail");

        virtual bool        ReadParameters ();
                                // Liest die Modellparameter ein und liefert
                                // true, oder false, falls die Standard-
                                // Parameter benutzt werden sollen

        virtual String      Description() const;
                                // Liefert die Modellparameter

        virtual void        ActivateNewCustomer ();
                                // erzeugt einen neuen Kunden mit Hilfe von
                                // 'NewCustomer' und aktiviert ihn in etwa mit
                                // t = 'meanArrivalTime', fuer first = false
                                // t = 0.0,               fuer first = true
                bool        GetLine();
                                // Anrufer versuchen, eine freie Leitung zu
                                // bekommen. Bei Erfolg wird true geliefert
                                // Ausserdem wird die Statistik aktualisiert
                void        HangUp ();
                                // gibt die Leitung wieder frei und
                                // aktualisiert die Statistik fuer Anrufer
                                // ohne freie Leitung

    protected:
        virtual Customer&   NewCustomer ();
                                // erzeugt einen neuen Kunden
        virtual Server&     NewServer ();
                                // erzeugt eine neue Bedienkraft
        virtual void        DoInitialSchedules ();
                                // Generiert und aktiviert
                                // - die Bedienkraefte
                                // - den ersten Kunden mit Hilfe von
                                //   'ActivateNewCustomer'


        bool        canChangeParam; // koennen Parameter noch geaendert werden
        // Modellparameter
        unsigned    servers;        // Anzahl der Bedienkraefte
        unsigned    lines;          // Anzahl der Telefonleitungen
        double      roundTripProb;  // Wahrsch. fuer Hin-/Rueckfahrt

        SimTime     meanArrivalTime,// mittl. Zwischenankunftszeiten d. Kunden
                    meanServiceTime,// mittl. Bedienzeit fuer einfache Fahrt
                    meanPatience,   // mittlere Geduld
                    stdDevPatience; // mit Standardabweichung

        // Modellvariablen
        unsigned        freeLines;      // Anzahl der freien Leitungen

        // Zufallszahlenstroeme
        RealDistExponential arrivalStream;  // Zwischenankunftszeit der Kunden
        RealDistExponential serviceStream;  // Bedienzeiten
        RealDistNormal      patienceStream; // Geduld der Kunden
        BoolDistBernoulli   tripStream;     // fuer Hin-/Rueckbuchungen

        // Warteschlangen
        ProcessQueue    serviceQ,       // wartende Bediener
                        callQ;          // wartende Anrufer

        // Statistiken
        Count           callCounter,    // alle Anrufer
                        unserved,       // Anrufer ohne freie Leitung
                        servedNoWait,   // Anrufer mit sofortiger Bedienung
                        aftWaitUnserved;// Anrufer ohne Bedienung

        CustWaitingTime waitingTime;    // Beobachtungsgroesse fuer
                                        // die Wartezeiten der Anrufer
                                        // wird fuer die Tallies benoetigt

                                        // Verweilzeiten im System von
        Tally           nonServedCall,  // nicht bedienten Anrufern
```

```
                        nonZWaitsCall;  // bedienten Anrufern, die
                                        // warten mußten
};

// ------------------------------------------------------------------------

#endif  // RAIL_H
```

rail_proc1.cc

```
// Dateiname    : rail_proc1.cc
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ------------------------------------------------------------------------

#include "rail_proc1.h"

#include "process.h"    // fuer Klasse Process

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Kunden

Customer::Customer  (RailModel&         owner,
                     const String&      name,
                     BoolDist&          TripStream,
                     RealDist&          ServiceStream,
                     RealDist&          PatienceStream,
                     ProcessQueue&      CustQ,
                     ProcessQueue&      ServiceQ,
                     Count&             CustCounter,
                     Count&             ServedNoWait,
                     Count&             AftWaitUnserved,
                     Tally&             NonServedCust,
                     Tally&             NonZWaits
                    )
    :   Process         (owner, name),
        railModel       (owner),
        tripStream      (TripStream),
        serviceStream   (ServiceStream),
        patienceStream  (PatienceStream),
        custQ           (CustQ),
        serviceQ        (ServiceQ),
        custCounter     (CustCounter),
        servedNoWait    (ServedNoWait),
        aftWaitUnserved (AftWaitUnserved),
        nonServedCust   (NonServedCust),
        nonZWaits       (NonZWaits),
        roundTrip       (false),
        iveBeenWaiting  (false),
        iveBeenServed   (false),
        arrivalTime     (0.0),
        waitingTime     (0.0)
{}

// ------------------------------------------------------------------------

void Customer::LifeCycle()
{
    // naechsten Kunden generieren:
    railModel.ActivateNewCustomer();

    custCounter.Update();
    if (EnterSystem())
    {   // Kunde kann in das System
        // Attribute setzen:
        roundTrip   = tripStream.Sample();
        arrivalTime = CurrentTime();
        // die maximale Wartezeit entspricht der Geduld
        // erfolgt eine Bedienung, so wird waitingTime in Serve() korrigiert
        waitingTime = patienceStream.Sample();

        // Sollte es, bedingt durch die Normalverteilung, zu negativen Werten
        // bei der Geduld kommen, werden diese Werte durch 0.0 ersetzt.
        // Achtung: Damit wird die Verteilung verfaelscht !
        // In DESMO wird ein negatives dt wie 0.0 behandelt
        if (waitingTime < 0.0) waitingTime = 0.0;

        custQ.Insert (*this);
        if (serviceQ.Empty())
        {   // kein Bediener frei, Kunde muss warten
            iveBeenWaiting = true;
            Hold (waitingTime);
```

```
        }
        else
        {   // ein Bediener ist frei, Kunde kann sofort bedient werden
            // ersten untaetigen Bediener aktivieren:
            Server& firstServer = (Server&)serviceQ.First();
            firstServer.WakeUpForService();
            // auf Ende der Bedienung warten, die die Bedienkraft
            // durch Aufruf von Customer::Serve() durchfuehrt:
            Passivate();
        }

        if (!iveBeenServed)         // ich wurde nicht bedient
        {   // Geduldsende erreicht
            // dann bin ich noch in der Warteschlange
            custQ.Remove (*this);
            aftWaitUnserved.Update();
            nonServedCust.Update();
        }
        else if (iveBeenWaiting)
            nonZWaits.Update();
        else
            servedNoWait.Update();

        LeaveSystem ();
    }
    DeleteOnTermination();
}

// ------------------------------------------------------------------------

SimTime Customer::Serve()
{   // wird von der Bedienkraft aufgerufen, um die Bedienung einzuleiten
    // und liefert die erforderliche Bedienzeit
    if (iveBeenWaiting)
        Cancel();       // Geduldsende wird nicht erreicht

    waitingTime     = CurrentTime() - arrivalTime;
    iveBeenServed   = true;
    custQ.Remove (*this);

    // Bedienzeit berechnen und zurueckgeben
    if (roundTrip)
        return 2.0 * serviceStream.Sample();
    else
        return       serviceStream.Sample();
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Telefon-Kunden

class Caller : public Customer
{   // Telfonkunden erweitern das Verhalten allgemeiner Kunden dadurch, daß
    // sie zunaechst versuchen, eine freie Leitung zu bekommen ('EnterSystem')
    // und diese ggf. wieder freigeben ('LeaveSystem')
    public:
        Caller  (RailModel&         owner,
                 BoolDist&          TripStream,
                 RealDist&          ServiceStream,
                 RealDist&          PatienceStream,
                 ProcessQueue&      CallQ,
                 ProcessQueue&      ServiceQ,
                 Count&             CallCounter,
                 Count&             ServedNoWait,
                 Count&             AftWaitUnserved,
                 Tally&             NonServedCall,
                 Tally&             NonZWaits
                )
            :   Customer    (owner, "Caller",   // neuer Name
                             TripStream,
                             ServiceStream,
                             PatienceStream,
                             CallQ,
                             ServiceQ,
                             CallCounter,
                             ServedNoWait,
                             AftWaitUnserved,
                             NonServedCall,
                             NonZWaits
                            )
            {}

        virtual bool    EnterSystem();
        virtual void    LeaveSystem();
};

// ------------------------------------------------------------------------

bool Caller::EnterSystem()
```

```
{   // versucht, eine freie Leitung zu belegen; liefert im Erfolgsfall true
    return railModel.GetLine ();
}

// ------------------------------------------------------------------------

void Caller::LeaveSystem()
{   // gibt die Leitung wieder frei
    railModel.HangUp ();
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Bediener

Server::Server  (RailModel&     owner,
                 ProcessQueue&  ServiceQ,
                 ProcessQueue&  CallQ
                )
    :   Process         (owner, "Server"),
        railModel       (owner),
        serviceQ        (ServiceQ),
        callQ           (CallQ)
    {}

// ------------------------------------------------------------------------

void Server::LifeCycle()
{
    while (true)
    {
        if (!CustomersAreWaiting())
        {   // auf neue Kunden warten
            serviceQ.Insert (*this);
            Wait();
            serviceQ.Remove (*this);
        }
        else
        {   // naechsten Kunden bedienen
            Customer& customer = NextCustomer();
            // Bedienung einleiten und Bedienzeit ermitteln
            SimTime serviceTime = customer.Serve();
            // Bedienung:
            Hold (serviceTime);
            // Kunden aktivieren
            customer.ActivateAfter (*this);
        }
        // evtl. Vorbereitung fuer naechste Bearbeitung
        CleanUpService();
    }
}

// ------------------------------------------------------------------------

bool Server::CustomersAreWaiting()
{   // Warten gerade Kunden?
    return !callQ.Empty();
}

// ------------------------------------------------------------------------

Customer& Server::NextCustomer()
{   // den naechsten Kunden auswaehlen, vorrausgesetzt es warten welche
    const char* where = "Server::NextCustomer";

    if (callQ.Empty())
        Error ( "Attempt to get first Customer of empty Queue", where);
    return (Customer&) callQ.First();
}

// ------------------------------------------------------------------------

void Server::WakeUpForService()
{   // wird vom Kunden aufgerufen, um anzuzeigen, daß bedient werden will
    // es sollte gelten: Kunde == CurrentProcess()
    if (IsScheduled())
        Cancel();
    ActivateAfter (CurrentProcess());
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Beobachtungsgroessen

CustWaitingTime::CustWaitingTime (RailModel& owner)
    :   ValueSupplier   (owner, "WaitingTime")
    {}

// ------------------------------------------------------------------------
```

```
// --------------------------------------------------------------------------
// Modell

RailModel::RailModel (Model* owner, const String& name)
    :   Model           (owner, name),
        canChangeParam  (true), // wird in DoInitialSchedules zu false

        // Modellparameter Standardwerte
        servers         (     5),
        lines           (    18),
        roundTripProb   (     0.75),
        meanArrivalTime (    20),
        meanServiceTime (    60),
        meanPatience    (   240),
        stdDevPatience  (   120),

        // Modellvariablen
        freeLines       ( lines),

        // Zufallszahlenstroeme
        arrivalStream   (*this, "Arrival", meanArrivalTime.Time()),
        serviceStream   (*this, "Service", meanServiceTime.Time()),
        patienceStream  (*this, "Patience", meanPatience.Time(),
                                            stdDevPatience.Time()),
        tripStream      (*this, "RoundTrip", roundTripProb),

        // Warteschlangen
        serviceQ        (*this, "Servers"),
        callQ           (*this, "WaitingCalls"),

        // Zaehler
        callCounter     (*this, "No.Callers"),
        unserved        (*this, "UnservedCalls"),
        servedNoWait    (*this, "ServNoWaitCall"),
        aftWaitUnserved (*this, "AftWaitUnsCall"),

        // Beobachtungsgroesse fuer Tallies
        waitingTime     (*this),

        // Tallies
        nonServedCall   (*this, "NonServedCall", waitingTime),
        nonZWaitsCall   (*this, "NonZWaitsCall", waitingTime)
{}

// --------------------------------------------------------------------------

bool RailModel::ReadParameters ()
{
    if (!canChangeParam)
        return false;

    Out() << Description() << endl << endl;
    Out() << "Standardparameter benutzen (j/n)? ";
    char c;
    In() >> c;
    if (c == 'j' || c == 'J')
        return false;   // Nicht einlesen, sondern Standard verwenden
    else
    {
        Out()   << "*** Telephonische Eisenbahn-Reservierung ***\n\n"
                << "Foldende Angaben bitte auf Sekundenbasis:\n"
                << "(in Klammern die Referenzparameter)\n";
        Out()   << "Anzahl der Telefonleitungen    (    18): ";
        In()    >> lines;
        Out()   << "Anzahl der Bedienkraefte       (     5): ";
        In()    >> servers;
        Out()   << "Zwischenankunftszeit           (    20): ";
        In()    >> meanArrivalTime;
            arrivalStream.ChangeParameter (meanArrivalTime.Time());
        Out()   << "Mittlere Bedienzeit            (    60): ";
        In()    >> meanServiceTime;
            serviceStream.ChangeParameter (meanServiceTime.Time());
        Out()   << "Mittlere 'Geduld' der Kunden   (   240): ";
        In()    >> meanPatience;
        Out()   << "mit Standardabweichung von     (   120): ";
        In()    >> stdDevPatience;
            patienceStream.ChangeParameter (meanPatience.Time(),
                                            stdDevPatience.Time());
        Out()   << "Wahrsch. fuer Hin-/Rueckfahrt  (  0.75): ";
        In()    >> roundTripProb;
            tripStream.ChangeParameter (roundTripProb);
    }
    return true;
}

// --------------------------------------------------------------------------

#include "strstr.h"     // fuer strstream
#include <iomanip.h>    // fuer setw(), setprecision()
```

```
String RailModel::Description () const
{
    strstream ss;
    ss.flags(ios::showpoint | ios::fixed | ios::right);
    ss
    << "*** Telephonische Eisenbahn-Reservierung ***\n\n"
    << "Eingabedaten :         (auf Sekundenbasis)\n"
    << "-------------------------------------\n\n"
    << "Anzahl der Leitungen          : " << setw( 6) << lines          << endl
    << "Anzahl der Bedienkraefte      : " << setw( 6) << servers        << endl
    << "Wahrsch. fuer Hin-Rueckfahrt  : " << setw(10) << setprecision(3)
                                                    << roundTripProb  << endl
    << "Zwischenankunftszeit          : " << setw(10) << meanArrivalTime<< endl
    << "mittlere Bedienzeit           : " << setw(10) << meanServiceTime<< endl
    << "mittlere Wartezeit            : " << setw(10) << meanPatience   << endl
    << "mit Standardabweichung von    : " << setw(10) << stdDevPatience
    << ends;
    return ss;
}

// ----------------------------------------------------------------------------

Customer& RailModel::NewCustomer ()
{   // erzeugt einen neuen Telefonkunden und uebergibt ihm:
    // Zufallszahlenstroeme, Warteschlangen, Datensammelobjekte
    return *new Caller  (*this,
                         tripStream,    serviceStream,  patienceStream,
                         callQ,         serviceQ,
                         callCounter,   servedNoWait, aftWaitUnserved,
                         nonServedCall, nonZWaitsCall);
}

// ----------------------------------------------------------------------------

Server& RailModel::NewServer ()
{   // erzeugt eine neue Telefon-Bedienkraft und uebergibt:
    // die beiden Warteschlangen
    return *new Server  (*this, serviceQ, callQ);
}

// ----------------------------------------------------------------------------

void RailModel::ActivateNewCustomer ()
{   // erzeugt mit Hilfe von 'NewCustomer' einen neuen Kunden und
    // aktiviert ihn zum zufaelligen Zeitpunkt
    NewCustomer().Activate (arrivalStream.Sample());
}

// ----------------------------------------------------------------------------

void RailModel::DoInitialSchedules ()
{
    canChangeParam = false;
    for (int i = 1; i <= servers; ++i)
    {   // Bedienkraefte generieren und aktivieren
        NewServer().Activate (0.0);
    }

    // ersten Kunden generieren und aktivieren
    NewCustomer().Activate (0.0);
}

// ----------------------------------------------------------------------------

bool RailModel::GetLine ()
{   // ein Kunde versucht, eine Telefonleitung zu belegen
    if (freeLines > 0)
    {   // Anrufer kann in das System
        --freeLines;
        return true;
    }
    else
    {   // es war keine Leitung frei, => Zaehler aktualisieren
        unserved.Update();
        return false;
    }
}

// ----------------------------------------------------------------------------

void RailModel::HangUp ()
{   // Ein Kunde legt auf
    ++freeLines;
}

// ----------------------------------------------------------------------------
```

rail_proc1.rpt

```
                            Clock Time = 360000.000
*******************************************************************************
*                                                                             *
*                           Experiment: rail_proc1                            *
*                                                                             *
*                                   Report                                    *
*                                                                             *
*******************************************************************************


                            Clock Time = 360000.000
*******************************************************************************
*                                                                             *
*                                 Model: Rail                                 *
*                                                                             *
*******************************************************************************

Rail reset at: 0.000

*** Telephonische Eisenbahn-Reservierung ***

Eingabedaten :          (auf Sekundenbasis)
-----------------------------------------

Anzahl der Leitungen          :     18
Anzahl der Bedienkraefte      :      5
Wahrsch. fuer Hin-Rueckfahrt  :      0.750
Zwischenankunftszeit          :     20.000
mittlere Bedienzeit           :     60.000
mittlere Wartezeit            :    240.000
mit Standardabweichung von    :    120.000

                                Distributions
                                -------------

Title           (Re)set    Obs Type              P a r a m e t e r s      Seed
-------------------------------------------------------------------------------
Arrival           0.000  18033 Neg-Expon.       20.0000                33427485
Service           0.000  15636 Neg-Expon.       60.0000                22276755
Patience          0.000  17882 Normal          240.0000     120.0000  46847980
RoundTrip         0.000  17882 Bernoulli         0.7500                43859043

                                Queues
                                ------

Title           (Re)set    Obs    Qmax    Qnow      Qavg.  Zeros   avg.Wait
-------------------------------------------------------------------------------
Servers           0.000   3857       5       0      0.423      1     39.440
WaitingCalls      0.000  17876      13       6      3.572   4199     71.901

                                Counts
                                ------

Title           (Re)set    Obs
-------------------------------
No.Callers        0.000  18033
UnservedCall      0.000    151
ServNoWaitCa      0.000   3857
AftWaitUnsCa      0.000   2240

                                Tallies
                                -------

Title           (Re)set    Obs       Mean    Std.Dev        Min        Max
-------------------------------------------------------------------------------
NonServedCal      0.000   2240     91.334     70.458      0.000    367.654
NonZWaitsCal      0.000  11774     91.756     63.410      0.007    346.554

                            Clock Time = 360000.000
*******************************************************************************
*                                                                             *
*                             End of Model: Rail                              *
*                                                                             *
*******************************************************************************
```

5.2.4.2 Variante mit Bürokunden

rail_proc2.h

```
// Dateiname    : rail_proc2.h
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************)
(*                                                                           *)
(*      Dateiname:        Rail_A1.MOD                                        *)
(*                                                                           *)
(*      Datum:            01.05.1992                                         *)
(*                                                                           *)
(*      Autoren:          Rolf Boelckow, Andreas Heymann,                    *)
(*                        Ralf Kadler,   Hansjoerg Liebert                   *)
(*      Modifiziert von:  Dirk Martinssen fuer                               *)
(*                        Uebungen zu Simulation II, WS 1997/98              *)
(*                        Aufgabe 1                                          *)
(*                                                                           *)
(*      Programminhalt:   Telefonische Eisenbahnreservierung                 *)
(*                        mit zusaetzlichen Buerokunden                      *)
(*                        (prozessorientiert)                                *)
(*                                                                           *)
(*****************************************************************************)
(* Das Modell dient zur Engpassanalyse einer Eisenbahnreservierungsstelle.  *)
(* Buchungen koennen sowohl telephonisch als auch durch Kunden im Buero     *)
(* erfolgen. Wartende Buerokunden haben Prioritaet gegenueber Telefonkunden.*)
(* Es stehen 18 Telephonleitungen und 5 Reservierungskraefte zur Verfuegung.*)
(* Kunden werden entweder sofort bedient, warten eine bestimmte Zeit oder   *)
(* oder erhalten keine freie Leitung (Verlustsystem).                       *)
(* Die Zwischenankunftszeiten der Anrufer und die Bedienzeiten sind negativ-*)
(* exponential-, die "Geduld" normalverteilt.                               *)
(* Die Wahrscheinlichkeit fuer die Buchung einer Hin- und Rueckfahrt be-    *)
(* traegt 0.75. In diesem Fall verdoppelt sich die Bedienzeit.              *)
(* Von den ankommenden Kunden sind 40% Telefonkunden.                       *)
(* Ferner macht jede Bedienkraft alle 30 Minuten eine Pause zwischen 5 und  *)
(* 15 Minuten, in der sie fuer neue Reservierungen nicht zur Verfuegung     *)
(* steht. Eine laufende Bedienung wird dabei noch vor der Pause beendet.    *)
(* Der Zeitpunkt fuer die neue Pause wird erst bestimmt, wenn die Bedien-   *)
(* kraft wieder ihre Arbeit aufnimmt.                                       *)
(* Zeitangaben erfolgen auf Sekundenbasis.                                  *)
(*                                                                          *)
(* Eingabedaten:                                                            *)
(*    - Simulationsdauer,                                                   *)
(*    - mittlere Zwischenankunftszeiten                                     *)
(*    - mittlere Bedienzeiten                                               *)
(*    - mittlere "Geduld" und deren Standardabweichung                      *)
(*                                                                          *)
(*    Mittlwert der Zwischenankunftszeiten   :  20 Sekunden                 *)
(*    Mittlwert fuer eine einfache Fahrt     :   1 Minute                   *)
(*    Mittlwert fuer eine Hin- und Rueckfahrt:   2 Minuten                  *)
(*    mittlere Wartezeit eines Kunden        :   4 Minuten                  *)
(*    Abweichung von der Wartezeit           :   2 Minuten                  *)
(*                                                                          *)
(* Ausgabedaten (Mindestanforderung):                                       *)
(*    - mittlere Wartezeit und Warteschlangenlaenge,                        *)
(*    - Anrufe ohne freie Leitung                                           *)
(*    - Anrufe ohne Bedienung nach Ablauf der "Geduldsphase"                *)
(*                                                                          *)
(* Zusaetzlich fuer die Aufgabe 1, Teil a:                                  *)
(*    - die Anzahl aller Buerokunden                                        *)
(*    - die mittlere Warteschlangenlaenge der Buerokunden                   *)
(*    - die mittlere Wartezeit der Buerokunden (inkl. der sofort Bedienten  *)
(*    - die noch wartenden, nicht bedienten Buerokunden                     *)
(*                                                                          *)
(****************************************************************************/

#ifndef RAIL1A_H
#define RAIL1A_H

#include "rail_proc1.h"

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Bediener

class OfficeServer : public Server
{
    public:
```

```
        OfficeServer    (RailModel&     owner,
                         ProcessQueue&  ServiceQ,
                         ProcessQueue&  CallQ,
                         ProcessQueue&  CustQ
                        )
            :   Server  (owner, ServiceQ, CallQ),
                custQ   (CustQ)
            {}

        virtual bool        CustomersAreWaiting();
        virtual Customer&   NextCustomer();

    private:
        ProcessQueue&   custQ;
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell1

class RailModel1a : public RailModel
{
    public:
            // Zeitbasis: Sekunden
        RailModel1a (Model* owner = 0, const String& name = "Rail_1a");

        virtual bool        ReadParameters ();
                                // Liest die Modellparameter ein und liefert
                                // true, oder false, falls die Standard-
                                // Parameter benutzt werden sollen

        virtual String      Description() const;
                                // Liefert die Modellparameter
    protected:
        virtual Customer&   NewCustomer ();
                                // erzeugt einen neuen Kunden
        virtual Server&     NewServer ();
                                // erzeugt eine neue Bedienkraft

        // Modellparameter
        double          callerProb;         // Wahrsch. fuer Telefonkunden

        // Zufallszahlenstroeme
        BoolDistBernoulli   custTypeStream; // true => Telefonkunde

        // Warteschlangen
        ProcessQueue    custQ;              // wartende Buerokunden

        // Statistiken
        Count           custCounter,        // alle Buerokunden
                        servedNoWaitCust,   // Buerokunden mit sofortiger
                                            // Bedienung
                        aftWaitUnservedCust;// Buerokunden ohne Bedienung

                                            // Verweilzeiten im System von
        Tally           nonServedCust,      // nicht bedienten Buerokunden
                        nonZWaitsCust;      // bedienten Buerokunden, die
                                            // warten mußten
};

// ----------------------------------------------------------------------------

#endif  // RAIL1A_H
```

rail_proc2.cc

```
// Dateiname    : rail_proc2.cc
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ----------------------------------------------------------------------------

#include "rail_proc2.h"

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Bediener

bool OfficeServer::CustomersAreWaiting()
{   // Buerokunde oder Telefonkunde?
    return !custQ.Empty() || Server::CustomersAreWaiting();
}

// ----------------------------------------------------------------------------
```

```
Customer& OfficeServer::NextCustomer()
{   // erst in der Bueroschlange nachsehen:
    if (!custQ.Empty())
        return (Customer&) custQ.First();
    else
        // sonst wie gehabt:
        return Server::NextCustomer();
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Modell

RailModel1a::RailModel1a (Model* owner, const String& name)
    :   RailModel       (owner, name),

        // Modellparameter
        callerProb      (0.4),

        // Zufallszahlenstroeme
        custTypeStream  (*this, "CustomerType", callerProb),

        // Warteschlangen
        custQ           (*this, "WaitingCust"),

        // Zaehler
        custCounter     (*this, "No.Customers"),
        servedNoWaitCust(*this, "ServNoWaitCu"),
        aftWaitUnservedCust (*this, "AftWaitUnsCust"),

        // Tallies
        nonServedCust   (*this, "NonServedCust", waitingTime),
        nonZWaitsCust   (*this, "NonZWaitsCust", waitingTime)
{}

// ------------------------------------------------------------------------

bool RailModel1a::ReadParameters ()
{
    if (!RailModel::ReadParameters())
        return false;
    else
    {
        Out()   << "Warsch. fuer Telefonkunden     (   0.4): ";
        In()    >> callerProb;
            custTypeStream.ChangeParameter (callerProb);
        return true;
    }
}

// ------------------------------------------------------------------------

#include "strstr.h"     // fuer strstream
#include <iomanip.h>    // fuer setw(), setprecision()

String RailModel1a::Description () const
{
    strstream ss;
    ss.flags(ios::showpoint | ios::fixed | ios::right);
    ss
    << RailModel::Description() << endl
    << "Wahrsch. fuer Telefonkunden   : " << setw(10) << setprecision(3)
    << callerProb << ends;
    return ss;
}

// ------------------------------------------------------------------------

Customer& RailModel1a::NewCustomer ()
{
    if (custTypeStream.Sample())
        // Telefonkunde wie gehabt erzeugen:
        return RailModel::NewCustomer();
    else
        // Buerokunde:
        return *new Customer(
                        *this,
                        "Customer",
                         tripStream,    serviceStream,  patienceStream,
                         custQ,         serviceQ,
                         custCounter,   servedNoWaitCust,aftWaitUnservedCust,
                         nonServedCust, nonZWaitsCust);
}

// ------------------------------------------------------------------------

Server& RailModel1a::NewServer ()
{   // erzeugt einen Bediener, der auch Buerokunden behandeln kann
```

```
    return *new OfficeServer (*this, serviceQ, callQ, custQ);
}

// ------------------------------------------------------------------------
```

rail_proc2.rpt

```
                          Clock Time = 360000.000
********************************************************************************
*                                                                              *
*                          Experiment: rail_proc2                              *
*                                                                              *
*                                 Report                                       *
*                                                                              *
********************************************************************************


                          Clock Time = 360000.000
********************************************************************************
*                                                                              *
*                            Model: Rail_1a                                    *
*                                                                              *
********************************************************************************

Rail_1a reset at: 0.000

*** Telephonische Eisenbahn-Reservierung ***

Eingabedaten :         (auf Sekundenbasis)
-----------------------------------------

Anzahl der Leitungen          :     18
Anzahl der Bedienkraefte      :      5
Wahrsch. fuer Hin-Rueckfahrt  :      0.750
Zwischenankunftszeit          :     20.000
mittlere Bedienzeit           :     60.000
mittlere Wartezeit            :    240.000
mit Standardabweichung von    :    120.000
Wahrsch. fuer Telefonkunden   :      0.400

                                Distributions
                                -------------

Title           (Re)set    Obs Type              P a r a m e t e r s      Seed
-------------------------------------------------------------------------------
Arrival           0.000  18033 Neg-Expon.       20.0000               33427485
Service           0.000  15573 Neg-Expon.       60.0000               22276755
Patience          0.000  18033 Normal          240.0000     120.0000  46847980
RoundTrip         0.000  18033 Bernoulli         0.7500               43859043
CustomerType      0.000  18034 Bernoulli         0.4000               64042082

                                Queues
                                ------

Title           (Re)set    Obs    Qmax    Qnow      Qavg.  Zeros   avg.Wait
-------------------------------------------------------------------------------
Servers           0.000   4232       5       0      0.448      1     38.091
WaitingCalls      0.000   7264      14       3      1.967   1841     97.443
WaitingCust       0.000  10766      15       0      1.013   2738     33.872

                                Counts
                                ------

Title           (Re)set    Obs
-------------------------------
No.Callers        0.000   7267
UnservedCall      0.000      0
ServNoWaitCa      0.000   1723
AftWaitUnsCa      0.000   1848
No.Customers      0.000  10766
ServNoWaitCu      0.000   2509
AftWaitUnsCu      0.000    609

                                Tallies
                                -------

Title           (Re)set    Obs       Mean    Std.Dev        Min        Max
-------------------------------------------------------------------------------
NonServedCal      0.000   1848    170.700    106.866      0.000    539.878
NonZWaitsCal      0.000   3691    106.173     85.919      0.012    494.842
NonServedCus      0.000    609     41.284     52.503      0.000    265.676
NonZWaitsCus      0.000   7645     44.394     41.279      0.014    324.145

                          Clock Time = 360000.000
********************************************************************************
*                                                                              *
```

```
*                          End of Model: Rail_1a                              *
*                                                                             *
*******************************************************************************
```

5.2.4.3 Variante mit Pausen

rail_proc3.h

```
// Dateiname    : rail_proc3.h
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************)
(*                                                                           *)
(*      Dateiname:        Rail_B1.MOD                                        *)
(*                                                                           *)
(*      Datum:            01.05.1992                                         *)
(*                                                                           *)
(*      Autoren:          Rolf Boelckow, Andreas Heymann,                    *)
(*                        Ralf Kadler,   Hansjoerg Liebert                   *)
(*      Modifiziert von:  Dirk Martinssen fuer                               *)
(*                        Uebungen zu Simulation II, WS 1997/98              *)
(*                        Aufgabe 1                                          *)
(*                                                                           *)
(*      Programminhalt:   Telefonische Eisenbahnreservierung                 *)
(*                        mit zusaetzlichen Buerokunden                      *)
(*                        und Pausen der Bedienkraefte                       *)
(*                        (prozessorientiert)                                *)
(*                                                                           *)
(*****************************************************************************)
(* Das Modell dient zur Engpassanalyse einer Eisenbahnreservierungsstelle.   *)
(* Buchungen koennen sowohl telephonisch als auch durch Kunden im Buero      *)
(* erfolgen. Wartende Buerokunden haben Prioritaet gegenueber Telefonkunden.*)
(* Es stehen 18 Telephonleitungen und 5 Reservierungskraefte zur Verfuegung.*)
(* Kunden werden entweder sofort bedient, warten eine bestimmte Zeit oder   *)
(* oder erhalten keine freie Leitung (Verlustsystem).                        *)
(* Die Zwischenankunftszeiten der Anrufer und die Bedienzeiten sind negativ-*)
(* exponential-, die "Geduld" normalverteilt.                                *)
(* Die Wahrscheinlichkeit fuer die Buchung einer Hin- und Rueckfahrt be-     *)
(* traegt 0.75. In diesem Fall verdoppelt sich die Bedienzeit.               *)
(* Von den ankommenden Kunden sind 40% Telefonkunden.                        *)
(* Ferner macht jede Bedienkraft alle 30 Minuten eine Pause zwischen 5 und   *)
(* 15 Minuten, in der sie fuer neue Reservierungen nicht zur Verfuegung      *)
(* steht. Eine laufende Bedienung wird dabei noch vor der Pause beendet.     *)
(* Der Zeitpunkt fuer die neue Pause wird erst bestimmt, wenn die Bedien-    *)
(* kraft wieder ihre Arbeit aufnimmt.                                        *)
(* Zeitangaben erfolgen auf Sekundenbasis.                                   *)
(*                                                                           *)
(* Eingabedaten:                                                             *)
(*    - Simulationsdauer,                                                    *)
(*    - mittlere Zwischenankunftszeiten                                      *)
(*    - mittlere Bedienzeiten                                                *)
(*    - mittlere "Geduld" und deren Standardabweichung                       *)
(*                                                                           *)
(*    Mittlwert der Zwischenankunftszeiten   :  20 Sekunden                  *)
(*    Mittlwert fuer eine einfache Fahrt     :   1 Minute                    *)
(*    Mittlwert fuer eine Hin- und Rückfahrt :   2 Minuten                   *)
(*    mittlere Wartezeit eines Kunden        :   4 Minuten                   *)
(*    Abweichung von der Wartezeit           :   2 Minuten                   *)
(*                                                                           *)
(* Ausgabedaten (Mindestanforderung):                                        *)
(*    - mittlere Wartezeit und Warteschlangenlaenge,                         *)
(*    - Anrufe ohne freie Leitung                                            *)
(*    - Anrufe ohne Bedienung nach Ablauf der "Geduldsphase"                 *)
(*                                                                           *)
(* Zusaetzlich fuer die Aufgabe 1, Teil a:                                   *)
(*    - die Anzahl aller Buerokunden                                         *)
(*    - die mittlere Warteschlangenlaenge der Buerokunden                    *)
(*    - die mittlere Wartezeit der Buerokunden (inkl. der sofort Bedienten   *)
(*    - die noch wartenden, nicht bedienten Buerokunden                      *)
(*                                                                           *)
(* Zusaetzlich fuer die Aufgabe 1, Teil b:                                   *)
(*    - die mittlere Pusenlaenge der Bedienkraefte                           *)
(*                                                                           *)
(*****************************************************************************/

#ifndef RAIL1B_H
#define RAIL1B_H

#include "rail_proc2.h"

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Bediener
```

```
class ServerWithBreak : public OfficeServer
{
    public:
        ServerWithBreak (RailModel&     owner,
                         ProcessQueue&  ServiceQ,
                         ProcessQueue&  CallQ,
                         ProcessQueue&  CustQ,
                         RealDist&      BreakStream,
                         RealDist&      BreakLengthStream,
                         Tally&         MeanBreak
                        )
            :   OfficeServer        (owner, ServiceQ, CallQ, CustQ),
                breakPoint          (CurrentTime()),
                breakStart          (0.0),
                breakStream         (BreakStream),
                breakLengthStream   (BreakLengthStream),
                meanBreak           (MeanBreak)
            {}

                SimTime     BreakStart() { return breakStart; }
    protected:
        virtual void        LifeCycle();

        virtual void        CleanUpService();
                                // wird nach Bearbeitung eines Kunden
                                // aufgerufen
                void        Wait();
                                // hiermit setzt sich die Bedienkraft zur
                                // Ruhe, wenn kein Kunde wartet

    private:
        void        GenerateBreak ();
                                // generiert die naechste Pause
        SimTime     breakPoint,         // Zeitpunkt, wann Pause zu machen ist
                    breakStart;         // Zeitpunkt, wann Pause gemacht wurde

        RealDist&   breakStream;
        RealDist&   breakLengthStream;

        Tally&      meanBreak;
};

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Beobachtungsgroessen

class BreakTime : public ValueSupplier
{
    public:
        BreakTime (RailModel& owner)
            :   ValueSupplier   (owner, "BreakTime")
            {}

        virtual double  Value () const
            {
                ServerWithBreak& currentServer =
                                    (ServerWithBreak&)CurrentProcess();
                return (CurrentTime() - currentServer.BreakStart()).Time();
            }
};

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Modell

class RailModel1b : public RailModel1a
{
    public:
            // Zeitbasis: Sekunden
        RailModel1b (Model* owner = 0, const String& name = "Rail_1b");

        virtual bool        ReadParameters ();
                                // Liest die Modellparameter ein und liefert
                                // true, oder false, falls die Standard-
                                // Parameter benutzt werden sollen

        virtual String      Description() const;
                                // Liefert die Modellparameter
    protected:
        virtual Server&     NewServer ();
                                // erzeugt eine neue Bedienkraft

        // Modellparameter
        SimTime             meanBreakTime;      // mittl. Zeit zw. Pausen
        SimTime             breakTimeMin;       // Mindestdauer der Pause
        SimTime             breakTimeMax;       // Maximaldauer der Pause

        // Zufallszahlenstroeme
```

```
        RealDistExponential breakStream;        // Pausenzeiten
        RealDistUniform     breakLengthStream;  // Pausenlaengen

        // Statistiken
        BreakTime           breakTime;          // Beobachtungsgroesse
        Tally               meanBreak;          // mittl. Pausenlaenge
};

// ---------------------------------------------------------------------------

#endif  // RAIL1B_H
```

rail_proc3.cc

```
// Dateiname    : rail_proc3.cc
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ---------------------------------------------------------------------------

#include "rail_proc3.h"

#include "process.h"    // fuer Klasse Process

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Bediener

void ServerWithBreak::LifeCycle()
{
    // erste Pause generieren:
    GenerateBreak();

    // weiter wie gehabt:
    OfficeServer::LifeCycle();
}

// ---------------------------------------------------------------------------

void ServerWithBreak::GenerateBreak()
{
    breakPoint = CurrentTime() + breakStream.Sample();
}

// ---------------------------------------------------------------------------

void ServerWithBreak::CleanUpService()
{
    if (CurrentTime() >= breakPoint)
    {
        breakStart = CurrentTime();
        Hold (breakLengthStream.Sample());
        meanBreak.Update();

        // naechste Pause generieren:
        GenerateBreak();
    }
}

// ---------------------------------------------------------------------------

void ServerWithBreak::Wait()
{
    Hold (breakPoint - CurrentTime());
}

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Modell

RailModel1b::RailModel1b (Model* owner, const String& name)
    :   RailModel1a     (owner, name),

        // Modellparameter
        meanBreakTime   (1800),
        breakTimeMin    ( 300),
        breakTimeMax    ( 900),

        // Zufallszahlenstroeme
        breakStream     (*this, "InterBreakTime", meanBreakTime.Time()),
        breakLengthStream(*this, "BreakLength", breakTimeMin.Time(),
                                                breakTimeMax.Time()),
        // Beobachtungsgroesse fuer Tally
        breakTime       (*this),
```

```
        // Statistiken
        meanBreak       (*this, "Breaks", breakTime)
{}

// ------------------------------------------------------------------------

bool RailModel1b::ReadParameters ()
{
    if (!RailModel1a::ReadParameters())
        return false;
    else
    {
        Out()   << "Mittl. Zeit zw. zwei Pausen    (  1800): ";
        In()    >> meanBreakTime;
            breakStream.ChangeParameter (meanBreakTime.Time());
        Out()   << "Mindestpausenlaenge            (   300): ";
        In()    >> breakTimeMin;
        Out()   << "Maximale Pausenlaenge          (   900): ";
        In()    >> breakTimeMax;
            breakLengthStream.ChangeParameter (breakTimeMin.Time(),
                                               breakTimeMax.Time());
        return true;
    }
}

// ------------------------------------------------------------------------

#include "strstr.h"     // fuer strstream
#include <iomanip.h>    // fuer setw(), setprecision()

String RailModel1b::Description () const
{
    strstream ss;
    ss.flags(ios::showpoint | ios::fixed | ios::right);
    ss
    << RailModel1a::Description() << endl
    << "Mittl. Zeit zw. zwei Pausen   : " << setw(10) << setprecision(3)
                                          << meanBreakTime<< endl
    << "Mindestpausenlaenge           : " << setw(10) << breakTimeMin << endl
    << "Maximale Pausenlaenge         : " << setw(10) << breakTimeMax << endl
    << ends;
    return ss;
}

// ------------------------------------------------------------------------

Server& RailModel1b::NewServer ()
{
    return *new ServerWithBreak (*this, serviceQ, callQ, custQ,
                                 breakStream, breakLengthStream, meanBreak);
}

// ------------------------------------------------------------------------
```

rail_proc3.rpt

```
                            Clock Time = 360000.000
*******************************************************************************
*                                                                             *
*                            Experiment: rail_proc3                           *
*                                                                             *
*                                   Report                                    *
*                                                                             *
*******************************************************************************


                            Clock Time = 360000.000
*******************************************************************************
*                                                                             *
*                               Model: Rail_1b                                *
*                                                                             *
*******************************************************************************

Rail_1b reset at: 0.000

*** Telephonische Eisenbahn-Reservierung ***

Eingabedaten :          (auf Sekundenbasis)
-----------------------------------------

Anzahl der Leitungen          :     18
Anzahl der Bedienkraefte      :      5
Wahrsch. fuer Hin-Rueckfahrt  :      0.750
Zwischenankunftszeit          :     20.000
mittlere Bedienzeit           :     60.000
```

```
mittlere Wartezeit            :    240.000
mit Standardabweichung von    :    120.000
Wahrsch. fuer Telefonkunden   :      0.400
Mittl. Zeit zw. zwei Pausen   :   1800.000
Mindestpausenlaenge           :    300.000
Maximale Pausenlaenge         :    900.000
```

Distributions

Title	(Re)set	Obs	Type	P a r a m e t e r s		Seed
Arrival	0.000	18033	Neg-Expon.	20.0000		33427485
Service	0.000	12463	Neg-Expon.	60.0000		22276755
Patience	0.000	18033	Normal	240.0000	120.0000	46847980
RoundTrip	0.000	18033	Bernoulli	0.7500		43859043
CustomerType	0.000	18034	Bernoulli	0.4000		64042082
InterBreakTi	0.000	709	Neg-Expon.	1800.0000		44366385
BreakLength	0.000	706	R-Uniform	300.0000	900.0000	41357879

Queues

Title	(Re)set	Obs	Qmax	Qnow	Qavg.	Zeros	avg.Wait
Servers	0.000	1573	5	0	0.152	1	34.896
WaitingCalls	0.000	7262	14	5	3.339	761	165.391
WaitingCust	0.000	10763	15	3	2.223	1180	74.331

Counts

Title	(Re)set	Obs
No.Callers	0.000	7267
UnservedCall	0.000	0
ServNoWaitCa	0.000	624
AftWaitUnsCa	0.000	3991
No.Customers	0.000	10766
ServNoWaitCu	0.000	910
AftWaitUnsCu	0.000	1571

Tallies

Title	(Re)set	Obs	Mean	Std.Dev	Min	Max
NonServedCal	0.000	3991	207.147	111.265	0.000	611.333
NonZWaitsCal	0.000	2647	141.424	99.614	0.032	562.018
NonServedCus	0.000	1571	100.747	83.536	0.000	383.903
NonZWaitsCus	0.000	8279	77.493	67.008	0.017	449.090
Breaks	0.000	704	599.373	175.027	300.610	899.672

```
                          Clock Time = 360000.000
*******************************************************************************
*                                                                             *
*                           End of Model: Rail_1b                             *
*                                                                             *
*******************************************************************************
```

5.2.5 Simulation eines Steinbruchs

quarry.h

```
// Dateiname    : quarry.h
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ----------------------------------------------------------------------------

#ifndef QUARRY_H
#define QUARRY_H

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Simulation eines Steinbruchs
//
// Der betrachtete Steinbruch besitzt eine breite Front mit
// durchschnittlicher Steinqualitaet und hat an einer Ecke einen engen
// Floez mit qualitativ hochwertigen Steinen. Jeden Morgen werden Steine
// der entsprechenden Qualitaet abgebrochen, um die Tagesnachfrage zu
// erfuellen.
//
// Der Steinbruch wird von zwei LKW-Typen angefahren. Es kommen grosse
// LKW, welche die Steine durchschnittlicher Qualitaet abfahren, und
// kleine LKW, die die qualitativ besseren Steine transportieren. Im
// Steinbruch selbst arbeiten drei mechanische Bagger. Die vorhandenen
// beiden grossen Bagger beladen ausschliesslich grosse LKW. Sie sind zu
// gross, um in den Bereich der qualitativ hochwertigen Steine
// hineinzufahren. Die Gegebenheiten machen es somit notwendig, dass zur
// Beladung der qualitativ hochwertigen Steine ein dritter, kleinerer
// Bagger eingesetzt wird. Die grossen Bagger arbeiten mit einer hoeheren
// Geschwindigkeit als der kleine Bagger. Wenn der kleine Bagger nicht
// beschaeftigt ist und gleichzeitig beide grossen Bagger arbeiten, kann
// der kleine Bagger einen weiteren ankommenden grossen LKW ebenfalls
// beladen. Sollte jedoch in dieser Zeit ein kleiner LKW ankommen, der
// qualitativ hochwertige Steine haben moechte, so wird der kleine Bagger
// in seiner Beladung unterbrochen und wendet sich dem kleinen LKW zu und
// belaedt diesen. Es ist nicht noetig, dass der unvollstaendig beladene
// LKW jetzt auf den kleinen Bagger wartet, sondern er wird mit dem
// naechsten freiwerdenen Bagger weiterbeladen. Der kleine Bagger wird
// ebenfalls bei der Beladung eines grossen LKW unterbrochen, wenn ein
// grosser Bagger frei wird. Der grosse Bagger uebernimmt die
// Restbeladung, da er schneller belaedt.
//
//
// Modelldaten:
//
// Ladekapazitaet der LKW:
//     grosser LKW :  20t
//     kleiner LKW :   5t
//
// Laderaten der Bagger:
//     grosser Bagger: 240t/h
//     kleiner Bagger:  60t/h
//
// Zwischenankuftszeiten (neg.-exponetional):
//     kleine LKW: 1/10h
//     grosse LKW: 1/22h
//
// Reihenfolge der Zufallszahlenstroeme:
//     Ankunft der kleinen LKW
//     Ankunft der grossen LKW
//
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------

#include "model.h"      // fuer Klasse Model

#include "process.h"    // fuer Process
#include "realdist.h"   // fuer Klasse RealDistExponential & -Normal
#include "waitq.h"      // fuer Klasse WaitQueue
#include "count.h"      // fuer Klasse Count
#include "accumula.h"   // fuer Klasse Accumulate
#include "tally.h"      // fuer Klasse Tally

// ----------------------------------------------------------------------------

class QuarryModel;      // forward
class Truck;
class SmallTruck;
```

```
class LargeTruck;
class Digger;
class SmallDigger;
class LargeDigger;

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Bagger und LKW

class QuarryModelProcess : public Process
{   // Klasse fuer allgemeine Prozesse im Steinbruch
    public:
        QuarryModelProcess  (QuarryModel&       owner,
                             const String&      name,
                             WaitQueue&         rendevouzPoint);

        // Type-Cast-Funktionen (in Unterklassen ueberschreiben)
        virtual Truck*          AsTruck ()          { return 0; }
        virtual SmallTruck*     AsSmallTruck ()     { return 0; }
        virtual LargeTruck*     AsLargeTruck ()     { return 0; }
        virtual Digger*         AsDigger ()         { return 0; }
        virtual SmallDigger*    AsSmallDigger ()    { return 0; }
        virtual LargeDigger*    AsLargeDigger ()    { return 0; }

    protected:
        QuarryModel&    quarry;         // Zugriff aufs Modell
        WaitQueue&      rendevouzPoint; // Synchronisationspunkt fuer
                                        // Bagger und LKW
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// LKW

class Truck : public QuarryModelProcess
{   // Klasse fuer allgemeine LKW im Steinbruch
    public:
        Truck   (QuarryModel&       owner,
                 const String&      name,
                 WaitQueue&         rendevouzPoint,
                 double             capacity);

        // Type-Cast-Funktion
        virtual Truck*      AsTruck ()          { return this; }

        SimTime     ArrivalTime() const         { return arrivalTime;       }
        double      RemainingCapacity() const   { return remainingCapacity; }
        void        ClearCapacity ()            { remainingCapacity = 0;    }
        void        Load (double load)          { remainingCapacity -= load;}
        SimTime     Delay ()           { return CurrentTime() - arrivalTime;}

    protected:
        void        Arrive()            {   arrivalTime = CurrentTime();    }

    private:
        SimTime         arrivalTime;
        double          remainingCapacity;  // noch zu beladen
};

// ----------------------------------------------------------------------------
// kleiner LKW

class SmallTruck : public Truck
{   // Klasse fuer allgemeine LKW im Steinbruch
    public:
        SmallTruck  (QuarryModel&       owner,
                     WaitQueue&         rendevouzPoint,
                     double             capacity,
                     ProcessQueue&      interruptableDiggers,
                     InterruptCode&     smallTruckArrived);

        // Type-Cast-Funktionen
        virtual SmallTruck*     AsSmallTruck ()     { return this; }

        virtual void            LifeCycle();

    private:
        // ------------------------------------------------------------------
        // Bedingung zum finden kleiner Bagger in der WaitQueue:
        class OnlySmallDigger : public Condition
        {
            public:
                OnlySmallDigger (Model& owner)
                    :   Condition   (owner, "small digger")
                    {}

                virtual bool Check (const Entity& slave) const;
        };
        // ------------------------------------------------------------------
```

```
        ProcessQueue&   interruptableDiggers;
        OnlySmallDigger onlySmallDigger;
        InterruptCode   smallTruckArrived;
};

// ---------------------------------------------------------------------------
// grosser LKW

class LargeTruck : public Truck
{   // Klasse fuer allgemeine LKW im Steinbruch
    public:
        LargeTruck  (QuarryModel&       owner,
                     WaitQueue&         rendevouzPoint,
                     double             capacity);

        // Type-Cast-Funktionen
        virtual LargeTruck*     AsLargeTruck ()     { return this; }

        virtual void            LifeCycle();
};

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// Bagger

class Digger : public QuarryModelProcess
{   // Klasse fuer allgemeine LKW im Steinbruch
    public:
        Digger  (QuarryModel&       owner,
                 const String&      name,
                 WaitQueue&         rendevouzPoint,
                 double             loadRate,
                 ProcessQueue&      interruptableDiggers);

        // Type-Cast-Funktionen
        virtual Digger*         AsDigger ()         { return this; }

        virtual void            BeginOfCoop (Truck& truck) = 0;
        virtual void            EndOfCoop   (Truck& truck) = 0;

        double                  LoadRate() const    { return loadRate;  }

    protected:
        // Kooperation
        // ---------------------------------------------------------------------
        class Mating : public ProcessCooperation
        {
            public:
                Mating (Model& owner)
                :   ProcessCooperation  (owner, "Mating", false)
                {}                                          // nicht im Trace

                virtual void    Cooperation (Process& master, Process& slave);
        };
        // ---------------------------------------------------------------------

        Mating          mating;
        ProcessQueue&   interruptableDiggers;

    private:
        double      loadRate;
};

// ---------------------------------------------------------------------------
// kleine Bagger

class SmallDigger : public Digger
{   // Klasse fuer allgemeine LKW im Steinbruch
    public:
        SmallDigger (QuarryModel&       owner,
                     WaitQueue&         rendevouzPoint,
                     double             loadRate,
                     ProcessQueue&      interruptableDiggers,
                     Count&             sdCompletedLT);

        // Type-Cast-Funktionen
        virtual SmallDigger*    AsSmallDigger ()    { return this; }

        virtual void            BeginOfCoop (Truck& truck);
        virtual void            EndOfCoop   (Truck& truck);

        virtual void            LifeCycle();
    private:
        Count&                  completedWorkOnLargeTrucks;

};

// ---------------------------------------------------------------------------
```

```
// grosse Bagger

class LargeDigger : public Digger
{   // Klasse fuer allgemeine LKW im Steinbruch
    public:
        LargeDigger (QuarryModel&       owner,
                     WaitQueue&         rendevouzPoint,
                     double             loadRate,
                     ProcessQueue&      interruptableDiggers,
                     InterruptCode&     largeDiggerFree);

        // Type-Cast-Funktionen
        virtual LargeDigger*    AsLargeDigger ()    { return this; }

        virtual void            BeginOfCoop (Truck& truck);
        virtual void            EndOfCoop   (Truck& truck);

        virtual void            LifeCycle();

        // ------------------------------------------------------------------
        // Bedinungen (Synchronisationsbedingungen)
        class OnlyLargeTruck : public Condition
        {   // Check liefert true, wenn der slave ein grosser LKW ist
            public:
                OnlyLargeTruck (Model& owner)
                    :   Condition   (owner, "large truck")
                    {}

                virtual bool Check (const Entity& slave) const;
        };
        // ------------------------------------------------------------------
        class OnlyInterruptedTruck : public Condition
        {   // Check liefert true, wenn der slave ein unterbrochener LKW ist
            public:
                OnlyInterruptedTruck (Model& owner)
                    :   Condition   (owner, "interrupted")
                    {}

                virtual bool Check (const Entity& slave) const;
        };
        // ------------------------------------------------------------------

    protected:
        OnlyLargeTruck          onlyLargeTrucks;
        OnlyInterruptedTruck    onlyInterruptedTrucks;
        InterruptCode           largeDiggerFree;
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell: Der Steinbruch

class QuarryModel : public Model
{
    public:
        QuarryModel (   Model*      owner                   =  0,
                const   String&     name                    = "Quarry",
                        unsigned    NofSmallDiggers         = 1,
                        unsigned    NofLargeDiggers         = 2,
                        SimTime     meanSmallArrivalTime    = 1.0/10,//h
                        SimTime     meanLargeArrivalTime    = 1.0/22,//h
                        double      smallDiggerLoadRate     =  60,   // t/h
                        double      largeDiggerLoadRate     = 240,   // t/h
                        double      smallTruckCapacity      =   5,   // t
                        double      largeTruckCapacity      =  20    // t
                        );

        virtual bool        ReadParameters ();
                                // Liest die Modellparameter ein und liefert
                                // true, oder false, falls die Standard-
                                // Parameter benutzt werden sollen

        virtual String      Description() const;
                                // Liefert die Modellparameter

                void        ActivateNewSmallTruck ();
                                // erzeugt einen neuen kleinen LKW und
                                // aktiviert ihn in einer von der
                                // Zwischenankunftszeit abhaengenden Zeit

                void        ActivateNewLargeTruck ();
                                // erzeugt einen neuen grossen LKW und
                                // aktiviert ihn in einer von der
                                // Zwischenankunftszeit abhaengenden Zeit

                void        UpdateDiggerUtil (SmallDigger&, int i);
                void        UpdateDiggerUtil (LargeDigger&, int i);
                                // veraendert die Anzahl der freien Bagger
                                // um i und aktualisiert die Statistik
```

```
            void        UpdateTruckDelay (SmallTruck&);
            void        UpdateTruckDelay (LargeTruck&);
                            // aktualisiert die entspr. LKW-Statistik
protected:
    virtual void        DoInitialSchedules ();
                            // Generiert und aktiviert
                            // - die Bagger
                            // - je einen Kleinen und grossen LKW

private:
    // ------------------------------------------------------------------
    // Beobachtungsgroessen
    class DiggerUtilSuppl : public ValueSupplier
    {
        public:
            DiggerUtilSuppl (QuarryModel& owner, unsigned& max,
                                                 unsigned& work)
                :   ValueSupplier   (owner, "Digger Util"),
                    quarry          (owner),
                    maxDiggers      (max),
                    workingDiggers  (work)
                {}

            virtual double Value () const
                {
                    return (double(workingDiggers) / maxDiggers) * 100;
                }
        private:
            QuarryModel&    quarry;
            unsigned&       maxDiggers;
            unsigned&       workingDiggers;
    };
    // ------------------------------------------------------------------
    class TruckDelaySuppl : public ValueSupplier
    {   //
        public:
            TruckDelaySuppl (Model& owner)
                :   ValueSupplier   (owner, "Truck Delay")
                {}

            virtual double Value () const;
    };
    // ------------------------------------------------------------------

    bool        canChangeParam; // koennen Parameter noch geaendert werden

    // Modellparameter
    unsigned    nOfSmallDiggers,    // Anzahl der kleinen Bagger
                nOfLargeDiggers;    // Anzahl der grossen Bagger

    SimTime     meanSmallArrival,   // mittl. Zwischenankunftszeit
                meanLargeArrival;   // der kleinen bzw. grossen LKW
    double      smallLoadRate,      // Laderaten der kleinen
                largeLoadRate,      // bzw. grossen Bagger
                smallCapacity,      // Ladekapazitaeten der kleinen
                largeCapacity;      // bzw. grossen LKW

    // Modellvariablen
    unsigned    workingSmallDiggers,// Anzahl arbeitender kleiner Bagger
                workingLargeDiggers,// Anzahl arbeitender grosser Bagger
                nOfAllDiggers,      // fuer Statistik
                allWorkingDiggers;  // fuer Statistik

    // Zufallszahlenstroeme
    RealDistExponential smallArrivalStream; // Zwischenankunftszeiten d.
    RealDistExponential largeArrivalStream; // kleinen bzw. grossen LKW

    // Synchronisationspunkt
    WaitQueue       rendevouzPoint;     // Bagger (M) und LKW (S)

    // Warteschlange
    ProcessQueue    interruptableDiggers;// grosse LKW beladende Bagger

    // Unterbrechungsursachen
    InterruptCode   smallTruckArrived,
                    largeDiggerFree;

    // Statistik
    DiggerUtilSuppl sdUtilSuppl,
                    ldUtilSuppl,
                    allUtilSuppl;
    TruckDelaySuppl truckDelaySuppl;
    Count           sdCompletedLtCount;
    Accumulate      smallDiggerUtil,
                    largeDiggerUtil,
                    allDiggerUtil;
    Tally           smallTruckDelay,
                    largeTruckDelay,
                    allTruckDelay;
```

```
};

// ----------------------------------------------------------------------------

#endif  // QUARRY_H
```

quarry.cc

```
// Dateiname    : quarry.cc
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ----------------------------------------------------------------------------

#include "quarry.h"

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Bagger und LKW

QuarryModelProcess::QuarryModelProcess  (QuarryModel&       owner,
                                         const String&      name,
                                         WaitQueue&         rendevouz)
    :   Process         (owner, name),
        quarry          (owner),
        rendevouzPoint  (rendevouz)
    {}

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// LKW

Truck::Truck    (QuarryModel&       owner,
                 const String&      name,
                 WaitQueue&         rendevouzPoint,
                 double             capacity)
    :   QuarryModelProcess  (owner, name, rendevouzPoint),
        arrivalTime         (0.0),
        remainingCapacity   (capacity)
    {}

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// kleiner LKW

SmallTruck::SmallTruck  (QuarryModel&       owner,
                         WaitQueue&         rendevouzPoint,
                         double             capacity,
                         ProcessQueue&      diggerQ,
                         InterruptCode&     sTruckArrived)
    :   Truck                   (owner, "S-Truck", rendevouzPoint, capacity),
        interruptableDiggers    (diggerQ),
        onlySmallDigger         (owner),
        smallTruckArrived       (sTruckArrived)
    {}

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------

void SmallTruck::LifeCycle()
{
    // Attribute setzen:
    Arrive();

    // Nachfolger erzeugen:
    quarry.ActivateNewSmallTruck();

    // Prioritaet maximieren, um einen kl. Bagger vor den grossen Trucks
    // zu bekommen
    SetPriority (2);

    Process* dummy;
    if (    !rendevouzPoint.Avail (dummy, onlySmallDigger)
        &&  !interruptableDiggers.Empty())
    {   // es arbeitet ein kl. Bagger an gr. LKW! => unterbrechen
        interruptableDiggers.First().Interrupt (smallTruckArrived);
    }
    rendevouzPoint.Wait ();

    quarry.UpdateTruckDelay (*this);

    DeleteOnTermination();
}
```

```
// ------------------------------------------------------------------------
// Suchbedingungen fuer kleine LKW, um kleine Bagger zu finden

bool SmallTruck::OnlySmallDigger::Check (const Entity& slave) const
{
    SmallDigger* digger = ((QuarryModelProcess&)slave).AsSmallDigger();
    if (digger)
        return true;
    else
        return false;
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// grosser LKW

LargeTruck::LargeTruck  (QuarryModel&       owner,
                         WaitQueue&         rendevouzPoint,
                         double             capacity)
    :   Truck   (owner, "L-Truck", rendevouzPoint, capacity)
    {}

// ------------------------------------------------------------------------

void LargeTruck::LifeCycle()
{
    // Attribute setzen
    Arrive();
    // Nachfolger erzeugen:
    quarry.ActivateNewLargeTruck();

    do {
        rendevouzPoint.Wait ();
        // fuer etwaige Unterbrechung Prioritaet erhoehen:
        SetPriority (1);
    } while (RemainingCapacity() > 0);

    quarry.UpdateTruckDelay (*this);

    DeleteOnTermination();
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Bagger

Digger::Digger  (QuarryModel&       owner,
                 const String&      name,
                 WaitQueue&         rendevouzPoint,
                 double             rate,
                 ProcessQueue&      diggers)
    :   QuarryModelProcess  (owner, name, rendevouzPoint),
        mating              (owner),
        interruptableDiggers(diggers),
        loadRate            (rate)
    {}

// ------------------------------------------------------------------------

void Digger::Mating::Cooperation (Process& master, Process& slave)
{
    const char* where = "Digger::Mating::Cooperation";

    // Pruefen, ob masster ein Bagger ist:
    Digger* digger = ((QuarryModelProcess&)master).AsDigger();
    if (!digger)
    {
        Warning ("Master is no Digger", where);
        return;
    }

    // Pruefen, ob slave ein LKW ist:
    Truck* truck = ((QuarryModelProcess&)slave).AsTruck();
    if (!truck)
    {
        Warning ("Slave is no Truck", where);
        return;
    }

    // Pruefen, ob bereits ein InterruptCode gesetzt war:
    if (GetInterruptCode() != InterruptCode::NoInterrupt())
    {
        Warning ("Cooperation starts with set InterruptCode", where,
                 "Code is cleared");
        ClearInterruptCode();
    }

    // Bagger evntl. in die WS fuer unterbrechbare Bagger einreihen:
    digger->BeginOfCoop (*truck);
```

```
    // eigentliche Kooperation:
    SimTime matingStart = CurrentTime();
    Hold (truck->RemainingCapacity() / digger->LoadRate());

    // Ist die Kooperation unterbrochen worden?
    if (GetInterruptCode() == InterruptCode::NoInterrupt())
        // nein, LKW voll beladen:
        truck->ClearCapacity();
    else
    {   // ja!!!
        if (digger->AsLargeDigger())
            // grosse Bagger duerfen nicht unterbrochen werden!!!
            Warning ("Large Diggers must not be interrupted!!!", where);

        // LKW teilweise beladen und Prioritaet erhoehen:
        truck->Load (   (CurrentTime() - matingStart).Time()
                     *   digger->LoadRate());
        truck->SetPriority (1);
        // Bagger evntl. uas der WS fuer unterbrechbare Bagger entfernen:
    }

    digger->EndOfCoop (*truck);
    ClearInterruptCode();
}

// ---------------------------------------------------------------------------
// kleine Bagger

SmallDigger::SmallDigger    (QuarryModel&       owner,
                             WaitQueue&         rendevouzPoint,
                             double             loadRate,
                             ProcessQueue&      interruptableDiggers,
                             Count&             sdCompletedLT)
    :   Digger  (owner, "S-Digger",
                 rendevouzPoint, loadRate, interruptableDiggers),
        completedWorkOnLargeTrucks  (sdCompletedLT)
    {}

// ---------------------------------------------------------------------------

void SmallDigger::LifeCycle()
{
    while (true)
        rendevouzPoint.Cooperate (mating);
}

// ---------------------------------------------------------------------------

void SmallDigger::BeginOfCoop (Truck& truck)
{
    quarry.UpdateDiggerUtil (*this, 1);
    if (truck.AsLargeTruck())
        // grosser LKW
        interruptableDiggers.Insert (*this);
}

// ---------------------------------------------------------------------------

void SmallDigger::EndOfCoop (Truck& truck)
{
    quarry.UpdateDiggerUtil (*this, -1);
    if (truck.AsLargeTruck())
    {
        // grosser LKW
        interruptableDiggers.Remove (*this);
        if (truck.RemainingCapacity() == 0)
            completedWorkOnLargeTrucks.Update();
    }
}

// ---------------------------------------------------------------------------
// grosse Bagger

LargeDigger::LargeDigger    (QuarryModel&       owner,
                             WaitQueue&         rendevouzPoint,
                             double             loadRate,
                             ProcessQueue&      diggers,
                             InterruptCode&     lDiggerFree)
    :   Digger  (owner, "L-Digger",
                 rendevouzPoint, loadRate, diggers),
        onlyLargeTrucks         (owner),
        onlyInterruptedTrucks   (owner),
        largeDiggerFree         (lDiggerFree)
    {}

// ---------------------------------------------------------------------------

void LargeDigger::BeginOfCoop (Truck&)
```

```
{
    quarry.UpdateDiggerUtil (*this, 1);
}

// ----------------------------------------------------------------------------

void LargeDigger::EndOfCoop (Truck&)
{
    quarry.UpdateDiggerUtil (*this, -1);
}

// ----------------------------------------------------------------------------

void LargeDigger::LifeCycle()
{
    // grosse Bagger stehen vor kleinen in der WaitQueue:
    SetPriority (1);

    while (true)
    {
        if (interruptableDiggers.Empty())
            rendevouzPoint.Cooperate (mating, onlyLargeTrucks);
        else
        {   // ein kleiner Bagger arbeitet an grossem Truck
            interruptableDiggers.First().Interrupt (largeDiggerFree);
            rendevouzPoint.Cooperate (mating, onlyInterruptedTrucks);
        }
    }
}

// ----------------------------------------------------------------------------
// Kooperationsbedingungen fuer grosse Bagger

bool LargeDigger::OnlyLargeTruck::Check (const Entity& slave) const
{
    LargeTruck* truck = ((QuarryModelProcess&)slave).AsLargeTruck();
    if (truck)
        return true;
    else
        return false;
}

// ----------------------------------------------------------------------------

bool LargeDigger::OnlyInterruptedTruck::Check (const Entity& slave) const
{
    // nur unterbrochene LWK haben Prioritaet 1
    return slave.GetPriority() == 1;
}

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Das Modell: Steinbruch

QuarryModel::QuarryModel (  Model*      owner,
                    const   String&     name,
                            unsigned    NofSmallDiggers,
                            unsigned    NofLargeDiggers,
                            SimTime     meanSmallArrivalTime,
                            SimTime     meanLargeArrivalTime,
                            double      smallDiggerLoadRate,
                            double      largeDiggerLoadRate,
                            double      smallTruckCapacity,
                            double      largeTruckCapacity)
    :   Model               (owner, name),
        canChangeParam      (true),
        // Modellparameter
        nOfSmallDiggers     (NofSmallDiggers),
        nOfLargeDiggers     (NofLargeDiggers),
        meanSmallArrival    (meanSmallArrivalTime),
        meanLargeArrival    (meanLargeArrivalTime),
        smallLoadRate       (smallDiggerLoadRate),
        largeLoadRate       (largeDiggerLoadRate),
        smallCapacity       (smallTruckCapacity),
        largeCapacity       (largeTruckCapacity),
        // Modellvariablen
        workingSmallDiggers (0),
        workingLargeDiggers (0),
        nOfAllDiggers       (nOfSmallDiggers + nOfLargeDiggers),
        allWorkingDiggers   (0),
        // Zufallszahlenstroeme
        smallArrivalStream      (*this, "Small Arrival",
                                 meanSmallArrival.Time()),
        largeArrivalStream      (*this, "Large Arrival",
                                 meanLargeArrival.Time()),
        // WaitQueue
        rendevouzPoint          (*this, "Mating WQ"),
        // Warteschlange
        interruptableDiggers    (*this, "SD on LT"),
```

```
        // Unterbrechungsursachen
        smallTruckArrived       ("S-Truck arrived"),
        largeDiggerFree         ("L-Digger free"),
        // Statistik
        // ValueSupplier
        sdUtilSuppl             (*this, nOfSmallDiggers, workingSmallDiggers),
        ldUtilSuppl             (*this, nOfLargeDiggers, workingLargeDiggers),
        allUtilSuppl            (*this, nOfAllDiggers, allWorkingDiggers),
        truckDelaySuppl         (*this),
        // Count
        sdCompletedLtCount      (*this, "SD compl. LT"),
        // Accumulate
        smallDiggerUtil         (*this, "S-Digger Util", sdUtilSuppl, false),
        largeDiggerUtil         (*this, "L-Digger Util", ldUtilSuppl, false),
        allDiggerUtil           (*this, "  Digger Util", allUtilSuppl, false),
        // Tally
        smallTruckDelay         (*this, "S-Truck Delay", truckDelaySuppl),
        largeTruckDelay         (*this, "L-Truck Delay", truckDelaySuppl),
        allTruckDelay           (*this, "  Truck Delay", truckDelaySuppl)
    {}

// ------------------------------------------------------------------------

void QuarryModel::DoInitialSchedules ()
{
    canChangeParam = false;

    // kleine Bagger erzeugen und vormekren:
    for (int i= 1; i <= nOfSmallDiggers; ++i)
        (new SmallDigger (*this, rendevouzPoint, smallLoadRate,
                          interruptableDiggers, sdCompletedLtCount)
        )->Activate (0.0);

    // grosse Bagger erzeugen und vormekren:
    for (int i= 1; i <= nOfLargeDiggers; ++i)
        (new LargeDigger (*this, rendevouzPoint, largeLoadRate,
                          interruptableDiggers, largeDiggerFree)
        )->Activate (0.0);

    // ersten kleinen LKW erzeugen und vormekren:
    ActivateNewSmallTruck();

    // ersten grossen LKW erzeugen und vormekren:
    ActivateNewLargeTruck();
}

// ------------------------------------------------------------------------

void QuarryModel::ActivateNewSmallTruck ()
{
    Process* newTruck = new SmallTruck (*this, rendevouzPoint, smallCapacity,
                                        interruptableDiggers,
                                        smallTruckArrived);
    newTruck->Activate (smallArrivalStream.Sample());
}

// ------------------------------------------------------------------------

void QuarryModel::ActivateNewLargeTruck ()
{
    Process* newTruck = new LargeTruck (*this, rendevouzPoint, largeCapacity);
    newTruck->Activate (largeArrivalStream.Sample());
}

// ------------------------------------------------------------------------

void QuarryModel::UpdateDiggerUtil (SmallDigger&, int i)
{   // SmallDigger wir nur zum Ueberladen benutzt
    const char* where = "QuarryModel::UpdateSmallDiggerUtil";

    workingSmallDiggers += i;
    if (workingSmallDiggers < 0)
    {
        Warning ("Attempt to release more small Diggers than exist", where);
        workingSmallDiggers = 0;
        return;
    }
    if (workingSmallDiggers > nOfSmallDiggers)
    {
        Warning ("Attempt to use more small Diggers than available", where);
        workingSmallDiggers = nOfSmallDiggers;
        return;
    }
    // Statistik:
    allWorkingDiggers = workingSmallDiggers + workingLargeDiggers;
    smallDiggerUtil.Update();
    allDiggerUtil.Update();
}
```

```
// ------------------------------------------------------------------------

void QuarryModel::UpdateDiggerUtil (LargeDigger&, int i)
{   // LargeDigger wir nur zum Ueberladen benutzt
    const char* where = "QuarryModel::UpdateLargeDiggerUtil";

    workingLargeDiggers += i;
    if (workingLargeDiggers < 0)
    {
        Warning ("Attempt to release more large Diggers than exist", where);
        workingLargeDiggers = 0;
        return;
    }
    if (workingLargeDiggers > nOfLargeDiggers)
    {
        Warning ("Attempt to use more large Diggers than available", where);
        workingLargeDiggers = nOfLargeDiggers;
        return;
    }
    // Statistik:
    allWorkingDiggers = workingSmallDiggers + workingLargeDiggers;
    largeDiggerUtil.Update();
    allDiggerUtil.Update();
}

// ------------------------------------------------------------------------

void QuarryModel::UpdateTruckDelay (SmallTruck&)
{   // SmallTruck wir nur zum Ueberladen benutzt
    smallTruckDelay.Update();
    allTruckDelay.Update();
}

// ------------------------------------------------------------------------

void QuarryModel::UpdateTruckDelay (LargeTruck&)
{   // LargeTruck wir nur zum Ueberladen benutzt
    largeTruckDelay.Update();
    allTruckDelay.Update();
}

// ------------------------------------------------------------------------

double QuarryModel::TruckDelaySuppl::Value() const
{
    const char* where = "QuarryModel::TruckDelaySuppl::Value";

    QuarryModelProcess& p     = (QuarryModelProcess&) CurrentProcess();
    Truck*              truck = p.AsTruck();

    if (!truck)
    {
        Warning ("To evaluate Truck Delay, a truck must be current!", where,
                 "0.0 is returned");
        return 0.0;
    }
    return truck->Delay().Time();
}

// ------------------------------------------------------------------------

#include <iomanip.h>

bool QuarryModel::ReadParameters ()
{
    if (!canChangeParam)
        return false;

    Out() << Description() << endl << endl;
    Out() << "Standardparameter benutzen (j/n)? ";
    char c;
    In() >> c;
    if (c == 'j' || c == 'J')
        return false;   // Nicht einlesen, sondern Standard verwenden
    else
    {
        int oflags = Out().flags (ios::showpoint |  ios::fixed | ios::right);
        Out()   << "(in Klammern die Referenzparameter)\n";
        Out()   << "Anzahl der kleinen Bagger       (" << setw( 6)
                << nOfSmallDiggers << "   )       : ";
        In()    >> nOfSmallDiggers;
        Out()   << "Anzahl der grossen Bagger       (" << setw( 6)
                << nOfLargeDiggers << "   )       : ";
        In()    >> nOfLargeDiggers;
            nOfAllDiggers = nOfSmallDiggers + nOfLargeDiggers;
        Out()   << "Zw.ankunftszeit der kleinen LKW (" << setw(10)
                << meanSmallArrival << ") [h]  : ";
        In()    >> meanSmallArrival;
            smallArrivalStream.ChangeParameter (meanSmallArrival.Time());
```

```
        Out()   << "Zw.ankunftszeit der grossen LKW (" << setw(10)
                << meanLargeArrival << ") [h]  : ";
        In()    >> meanLargeArrival;
            largeArrivalStream.ChangeParameter (meanLargeArrival.Time());
        Out()   << "Laderate der kleinen Bagger     (" << setw(10)
                << setprecision(3) << smallLoadRate << ") [t/h]: ";
        In()    >> smallLoadRate;
        Out()   << "Laderate der grossen Bagger     (" << setw(10)
                << setprecision(3) << largeLoadRate << ") [t/h]: ";
        In()    >> largeLoadRate;
        Out()   << "Kapazitaet der kleinen Bagger   (" << setw(10)
                << setprecision(3) << smallCapacity << ") [t]  : ";
        In()    >> smallCapacity;
        Out()   << "Kapazitaet der grossen Bagger   (" << setw(10)
                << setprecision(3) << largeCapacity << ") [t]  : ";
        In()    >> largeCapacity;
        Out().flags(oflags);
    }
    return true;
}

// ------------------------------------------------------------------------

#include "strstr.h" // fuer strstream

String QuarryModel::Description () const
{
    strstream ss;
    ss.flags(ios::showpoint | ios::fixed | ios::right);
    ss
    << "*** Steinbruch ***\n\n"
    << "Eingabedaten :         \n"
    << "---------------------------------------\n\n"
    << "Anzahl der kleinen Bagger       : " << setw( 6) << nOfSmallDiggers
                                            << endl
    << "Anzahl der grossen Bagger       : " << setw( 6) << nOfLargeDiggers
                                            << endl
    << "Zw.ankunftsz. der kleinen LKW(h): " << setw(10) << meanSmallArrival
                                            << endl
    << "Zw.ankunftsz. der grossen LKW(h): " << setw(10) << meanLargeArrival
                                            << endl
    << "Laderate der kleinen Bagger(t/h): " << setw(10) << setprecision(3)
                                            << smallLoadRate << endl
    << "Laderate der grossen Bagger(t/h): " << setw(10) << setprecision(3)
                                            << largeLoadRate << endl
    << "Kapazitaet der kleinen LKW (t)  : " << setw(10) << setprecision(3)
                                            << smallCapacity << endl
    << "Kapazitaet der grossen LKW (t)  : " << setw(10) << setprecision(3)
                                            << largeCapacity << endl
    << ends;
    return ss;
}

// ------------------------------------------------------------------------
```

quarry.rpt

```
                            Clock Time = 240.000
********************************************************************************
*                                                                              *
*                             Experiment: quarry                               *
*                                                                              *
*                                  Report                                      *
*                                                                              *
********************************************************************************


                            Clock Time = 240.000
********************************************************************************
*                                                                              *
*                                Model: Quarry                                 *
*                                                                              *
********************************************************************************

Quarry reset at: 0.000

*** Steinbruch ***

Eingabedaten :
---------------------------------------

Anzahl der kleinen Bagger       :      1
Anzahl der grossen Bagger       :      2
Zw.ankunftsz. der kleinen LKW(h):      0.100
Zw.ankunftsz. der grossen LKW(h):      0.045
Laderate der kleinen Bagger(t/h):     60.000
```

```
Laderate der grossen Bagger(t/h):    240.000
Kapazitaet der kleinen LKW (t)  :      5.000
Kapazitaet der grossen LKW (t)  :     20.000
```

Distributions

Title	(Re)set	Obs	Type	P a r a m e t e r s	Seed
Small Arriva	0.000	2455	Neg-Expon.	0.1000	33427485
Large Arriva	0.000	5239	Neg-Expon.	0.0455	22276755

Queues

Title	(Re)set	Obs	Qmax	Qnow	Qavg.	Zeros	avg.Wait
SD on LT	0.000	891	1	0	0.102	0	0.027

Counts

Title	(Re)set	Obs
SD compl. LT	0.000	0

Accumulates

Title	(Re)set	Obs	Mean	Std.Dev	Min	Max
S-Digger Uti	0.000	6689	95.353	21.051	0.000	100.000
L-Digger Uti	0.000	10474	89.641	25.637	0.000	100.000
Digger Uti	0.000	17163	91.545	19.812	0.000	100.000

Tallies

Title	(Re)set	Obs	Mean	Std.Dev	Min	Max
S-Truck Dela	0.000	2453	0.320	0.242	0.083	1.350
L-Truck Dela	0.000	5236	0.233	0.151	0.083	0.888
Truck Dela	0.000	7689	0.261	0.189	0.083	1.350

Wait-Queues

Title	(Re)set	Obs	Qmax	Qnow	Qavg.	Zeros	avg.Wait
Mating WQ M	0.000	8583	3	0	0.254	7567	0.007
Mating WQ S	0.000	8583	26	0	5.611	1909	0.157

```
                          Clock Time = 240.000
********************************************************************************
*                                                                              *
*                          End of Model: Quarry                                *
*                                                                              *
********************************************************************************
```

5.2.6 Hafenmodell

5.2.6.1 Grundmodell

harbour1.h

```
// Dateiname    : harbour1.h
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************
(*                                                                           *
(*      Autor:            Dirk Martinssen                                    *
(*                                                                           *
(*      Programminhalt:   Hafenmodell                                        *
(*                        Andocken von Containerschiffen mit Hafenschleppern *
(*                        Aufgabe zu Simulation                              *
(*                                                                           *
(*****************************************************************************
(* An einem Containerterminal stehen zwei Kais zur Entladung jeweils eines   *
(* Containerschiffes zur Verfuegung. Die Schiffe laufen den Hafen mit expo-  *
(* nentialverteilter Zwischenankunftszeit (u=10 Std.) an. Sie muessen dort   *
(* warten, bis ein Kai frei wird, um unmittelbar anzudocken und entladen zu  *
(* werden. Anschliessend verlassen die Schiffe den Hafen. Das An- bzw. Ab-   *
(* docken ist normalverteilt mit einem Mittelwert von 2 Std. und einer       *
(* Standardabweichung von 0.2 Std. Beim Anlegen sind jeweils 2 Hafenschlep-  *
(* per notwendig, beim Ablegen jeweils einer. Waehrend ein Schiff ablegt,    *
(* kann ein weiteres parallel dazu schon anlegen. Insgesamt stehen 3 Schlep- *
(* per zur Verfuegung. Die Entladezeit sei normalverteilt mit einem Mittel-  *
(* wert von 14 Std. und einer Standardabweichung von 3 Std.                  *
(* Die Simulationszeit betrage 365 Tage von 24 Std.                          *
(*****************************************************************************/

#ifndef HARBOUR1_H
#define HARBOUR1_H

#include "model.h"      // fuer Klasse Model

#include "process.h"    // fuer Process
#include "realdist.h"   // fuer Klasse RealDistExponential & -Normal
#include "res.h"        // fuer Klasse Res

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Schiffe

class HarbourModel;     // forward

class Ship : public Process
{   // Klasse fuer allgemeine Schiffe
    // Unterklassen koennen das Verhalten durch Ueberschreiben von
    // 'Dock' und 'UnDock' anpassen
    public:
        Ship    (HarbourModel&      owner,
                 RealDist&          DockingStream,
                 RealDist&          LoadingStream,
                 Res&               Tugs,
                 Res&               Quays,
                 const String&      name = "Ship");

    protected:
        virtual void    LifeCycle();
        virtual void    Dock();     // Andocken des Schiffes
        virtual void    UnDock();   // Abdocken des Schiffes

        HarbourModel&   harbour;    // um Nachfolger zu erzeugen

    private:
        RealDist&           dockingStream;
        RealDist&           loadingStream;
        Res&                tugs;
        Res&                quays;
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell: Der Hafen
```

```
class HarbourModel : public Model
{
    public:
            // Zeitbasis: Stunden
        HarbourModel (  Model*      owner   =  0,
                const   String&     name    = "Harbour",
                        unsigned    NofTugs = 3,
                        unsigned    NofQuays= 2);

        virtual bool        ReadParameters ();
                                // Liest die Modellparameter ein und liefert
                                // true, oder false, falls die Standard-
                                // Parameter benutzt werden sollen

        virtual String      Description() const;
                                // Liefert die Modellparameter

                void        ActivateNewShip ();
                                // erzeugt ein neues Schiff mit Hilfe von
                                // 'NewShip' und aktiviert es in einer von
                                // der Zwischenankunftszeit abhaengenden Zeit

    protected:
        virtual void        DoInitialSchedules ();
                                // Generiert und aktiviert
                                // - das erste Schiff

        virtual Ship&       NewShip ();
                                // erzeugt ein neues Schiff

    private:
        bool        canChangeParam; // koennen Parameter noch geaendert werden

        // Modellparameter
        unsigned    ntugs;          // Anzahl der Schlepper
        unsigned    nquays;         // Anzahl der Kais

        SimTime     meanArrivalTime,// mittl. Zw.ankunftszeit d. Schiffe
                    meanDockingTime,// mittl. An-/Abdockzeit
                    stdDevDocking,  // mit Standardabweichung
                    meanLoadingTime,// mittlere Entladezeit
                    stdDevLoading;  // mit Standardabweichung

        // Zufallszahlenstroeme
        RealDistExponential arrivalStream;  // Zw.ankunftszeiten d. Schiffe
    protected:
        RealDistNormal      dockingStream;  // An-/Abdockzeiten
        RealDistNormal      loadingStream;  // Be-/Entladezeiten

        // Ressourcen
        Res             tugs,           // Schlepper
                        quays;          // Kais
};

// ----------------------------------------------------------------------------

#endif  // HARBOUR1_H
```

harbour1.cc

```
// Dateiname    : harbour1.cc
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ----------------------------------------------------------------------------

#include "harbour1.h"

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell: Der Hafen

HarbourModel::HarbourModel (Model* owner, const String& name,
                            unsigned    NofTugs,
                            unsigned    NofQuays)
    :   Model   (owner, name),
        canChangeParam  (true),
        ntugs           (NofTugs),
        nquays          (NofQuays),
        meanArrivalTime (    10  ),
        meanDockingTime (     2  ),
        stdDevDocking   (     0.2),
        meanLoadingTime (    14  ),
        stdDevLoading   (     3  ),
```

```
        arrivalStream   (*this, "Arrival", meanArrivalTime.Time()),
        dockingStream   (*this, "Landing", meanDockingTime.Time(),
                                           stdDevDocking.Time()),
        loadingStream   (*this, "Unload",  meanLoadingTime.Time(),
                                           stdDevLoading.Time()),
        tugs            (*this, "Tugs",  ntugs),
        quays           (*this, "Quays", nquays)
    {}

// ------------------------------------------------------------------------

bool HarbourModel::ReadParameters ()
{
    if (!canChangeParam)
        return false;

    Out() << Description() << endl << endl;
    Out() << "Standardparameter benutzen (j/n)? ";
    char c;
    In() >> c;
    if (c == 'j' || c == 'J')
        return false;   // Nicht einlesen, sondern Standard verwenden
    else
    {
        Out()   << "*** Hafenmodell mit Schleppern und Kais ***\n\n"
                << "Foldende Angaben bitte auf Stundenbasis:\n"
                << "(in Klammern die Referenzparameter)\n";
        Out()   << "Zwischenankunftszeit           (  10  ): ";
        In()    >> meanArrivalTime;
            arrivalStream.ChangeParameter (meanArrivalTime.Time());
        Out()   << "Mittlere Andockzeit            (   2  ): ";
        In()    >> meanDockingTime;
        Out()   << "mit Standardabweichung von     (   0.2): ";
        In()    >> stdDevDocking;
            dockingStream.ChangeParameter (meanDockingTime.Time(),
                                           stdDevDocking.Time());
        Out()   << "Mittlere Be-/Entladezeit       (  14  ): ";
        In()    >> meanLoadingTime;
        Out()   << "mit Standardabweichung von     (   3  ): ";
        In()    >> stdDevLoading;
            loadingStream.ChangeParameter (meanLoadingTime.Time(),
                                            meanLoadingTime.Time());
    }
    return true;
}

// ------------------------------------------------------------------------

#include "strstr.h"     // fuer strstream
#include <iomanip.h>    // fuer setw(), setprecision()

String HarbourModel::Description () const
{
    strstream ss;
    ss.flags(ios::showpoint | ios::fixed | ios::right);
    ss
    << "*** Hafenmodell mit Schleppern und Kais ***\n\n"
    << "Eingabedaten :          (auf Stundenbasis)\n"
    << "-------------------------------------\n\n"
    << "Anzahl der Schlepper          : " << setw( 6)<< ntugs          << endl
    << "Anzahl der Kais               : " << setw( 6)<< nquays         << endl
    << "Zwischenankunftszeit          : " << setw(10)<< meanArrivalTime<< endl
    << "mittlere Andockzeit           : " << setw(10)<< meanDockingTime<< endl
    << "mit Standardabweichung von    : " << setw(10)<< stdDevDocking  << endl
    << "mittlere Be-/Entladezeit      : " << setw(10)<< meanLoadingTime<< endl
    << "mit Standardabweichung von    : " << setw(10)<< stdDevLoading
    << ends;
    return ss;
}


// ------------------------------------------------------------------------

void HarbourModel::ActivateNewShip ()
{
    NewShip().Activate (arrivalStream.Sample());
}

// ------------------------------------------------------------------------

void HarbourModel::DoInitialSchedules ()
{
    NewShip().Activate (0.0);
}

// ------------------------------------------------------------------------

Ship& HarbourModel::NewShip ()
{
```

```
    return *new Ship (*this, dockingStream, loadingStream, tugs, quays);
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Schiffe

Ship::Ship  (HarbourModel&      owner,
             RealDist&          DockingStream,
             RealDist&          LoadingStream,
             Res&               Tugs,
             Res&               Quays,
             const String&      name)
    :   Process         (owner, name),
        harbour         (owner),
        dockingStream   (DockingStream),
        loadingStream   (LoadingStream),
        tugs            (Tugs),
        quays           (Quays)
    {}

// ------------------------------------------------------------------------

void Ship::LifeCycle ()
{
    // Nachfolger erzeugen:
    harbour.ActivateNewShip();

    // Kai anfordern und andocken:
    quays.Acquire (1);
    Dock();
    // Entladen:
    Hold (loadingStream.Sample ());
    // Abdocken:
    UnDock();
    DeleteOnTermination();
}

// ------------------------------------------------------------------------

void Ship::Dock ()
{
    // Schlepper anfordern und anlegen:
    tugs.Acquire (2);
    Hold (dockingStream.Sample());
    tugs.Release (2);
}

// ------------------------------------------------------------------------

void Ship::UnDock ()
{
    // Schlepper anfordern und ablegen:
    tugs.Acquire (1);
    quays.Release (1);
    Hold (dockingStream.Sample());
    tugs.Release (1);
}

// ------------------------------------------------------------------------
```

harbour1.rpt

```
                              Clock Time = 8760.000
*******************************************************************************
*                                                                             *
*                              Experiment: harbour1                           *
*                                                                             *
*                                    Report                                   *
*                                                                             *
*******************************************************************************


                              Clock Time = 8760.000
*******************************************************************************
*                                                                             *
*                                 Model: Harbour                              *
*                                                                             *
*******************************************************************************

Harbour reset at: 0.000

*** Hafenmodell mit Schleppern und Kais ***

Eingabedaten :          (auf Stundenbasis)
-----------------------------------------
```

```
Anzahl der Schlepper          :      3
Anzahl der Kais               :      2
Zwischenankunftszeit          :     10.000
mittlere Andockzeit           :      2.000
mit Standardabweichung von    :      0.200
mittlere Be-/Entladezeit      :     14.000
mit Standardabweichung von    :      3.000

                              Distributions
                              -------------

Title           (Re)set    Obs Type              P a r a m e t e r s      Seed
-------------------------------------------------------------------------------
Arrival           0.000    837 Neg-Expon.       10.0000              33427485
Landing           0.000   1672 Normal            2.0000       0.2000  22276755
Unload            0.000    836 Normal           14.0000       3.0000  46847980

                                Resources
                                ---------

Title           (Re)set  Users  Limit    Min    Now  Usage [%]   avg.Wait QMaxL
-------------------------------------------------------------------------------
Tugs              0.000   1671      3      0      1     19.213      0.061     1
Quays             0.000    835      2      0      0     76.774     13.231    10

                            Clock Time = 8760.000
*******************************************************************************
*                                                                             *
*                            End of Model: Harbour                            *
*                                                                             *
*******************************************************************************
```

5.2.6.2 Variante mit Gezeiten

harbour2.h

```
// Dateiname    : harbour2.h
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// Beschreibung uebernommen von :
//
/*****************************************************************************
(*                                                                           *
(*      Autor:            Dirk Martinssen                                    *
(*                                                                           *
(*      Programminhalt:   Hafenmodell                                        *
(*                        Andocken von Containerschiffen mit Hafenschleppern *
(*                        Zusaetzlich ist Ebbe und Flut zu beruecksichtigen  *
(*                        Aufgabe zu Simulation                              *
(*                                                                           *
(*****************************************************************************
(* An einem Containerterminal stehen zwei Kais zur Entladung jeweils eines   *
(* Containerschiffes zur Verfuegung. Die Schiffe laufen den Hafen mit expo-  *
(* nentialverteilter Zwischenankunftszeit (u=10 Std.) an. Sie muessen dort   *
(* warten, bis ein Kai frei wird, um unmittelbar anzudocken und entladen zu  *
(* werden. Anschliessend verlassen die Schiffe den Hafen.                    *
(* Zusaetzlich zum Grundmodell werden hier noch die Gezeiten mit berueck-    *
(* sichtigt. Die beladenen Containerschiffe koennen nur einlaufen, wenn kein *
(* Niedrigwasser herrscht. Das Auslaufen der Schiffe ist dagegen unabhaengig *
(* von dem jeweiligen Wasserstand.                                           *
(* Das An- bzw. Abdocken ist normalverteilt mit einem Mittelwert von 2 Std.  *
(* und einer Standardabweichung von 0.2 Std. Beim Anlegen sind jeweils 2     *
(* Hafenschlepper notwendig, beim Ablegen jeweils einer. Waehrend ein Schiff *
(* ablegt, kann ein weiteres parallel dazu schon anlegen. Insgesamt stehen 3 *
(* Schlepper zur Verfuegung. Die Entladezeit sei normalverteilt mit einem    *
(* Mittelwert von 14 Std. und einer Standardabweichung von 3 Std.            *
(* Der Zeitraum, in der ein Schiff aufgrund von Niedrigwasser nicht ein-     *
(* laufen kann, betraegt 4 Stunden. Die dazwischen liegenden Flutphasen      *
(* betragen jeweils 9 Stunden.                                               *
(*                                                                           *
(* Die Simulationszeit betrage 365 Tage von 24 Std.                          *
(*****************************************************************************/

#ifndef HARBOUR2_H
#define HARBOUR2_H

#include "harbour1.h"       // fuer Klasse HarbourModel

#include "process.h"    // fuer Process
#include "res.h"        // fuer Klasse Res
#include "condq.h"      // fuer Klasse CondQueue, Condition

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Schiffe, die bei Niedrigwasser nicht einfahren koennen

class TideDependentShip : public Ship
{   // Klasse fuer gezeitenabhaengige Schiffe
    // erweitert die Klasse 'Ship' um die Gezeitenabhaengigkeit, indem die
    // Methode 'Dock' so redefiniert wird, dass die CondQueue 'Reede' und
    // die Condition 'HighTideAntTugsAvail' benutzt wird
    // 'Undock' wird so erweitert, dass der CondQueue die Freigabe des
    // Schleppers signalisiert wird
    // ueberschreibt 'WaitForLanding' mit "Warten auf Flut und Schlepper"
    public:
        TideDependentShip   (HarbourModel&  owner,
                             RealDist&      DockingStream,
                             RealDist&      LoadingStream,
                             Res&           Tugs,
                             Res&           Quays,
                             CondQueue&     Reede,
                             Condition&     HighTideAndTugsAvail,
                             const String&  name = "Ship");

    protected:
        virtual void    Dock();
        virtual void    UnDock();

    private:
        CondQueue&          reede;
        Condition&          highTideAndTugsAvail;
};
```

```
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Der Tide-Prozess

class Tide : public Process
{   // Der Tide-Prozess simuliert die Gezeiten
    // 'Low' liefert true bei Niedrigwasser, wenn Schiffe also nicht einlaufen
    // koennen.
    public:
        Tide    (Model&         owner,
                 CondQueue&     Reede,
                 const String&  name = "Tide")
            :   Process (owner, name),
                lowTide (true),         // beginnt mit Ebbe
                reede   (Reede)
            {}

        bool    High () const   { return !lowTide; }
        bool    Low  () const   { return  lowTide; }
    protected:
        virtual void    LifeCycle ();
                        // Tide beginnt mit der 4-stuendigen Niedrigwasser-
                        // Phase, die sich mit der 9-stuendigen Hochwasser-
                        // Phase abwechselt. Das Erreichen der Hochwasser-
                        // Phase wird der CondQueue 'reede' signalisiert
    private:
        bool        lowTide;
        CondQueue&  reede;
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Der Bedingung zum Andocken der Schiffe

class DockCondition : public Condition
{   // Die Bedingugn ist erfuellt, wenn sowohl Hochwasser ist, als auch
    // genuegend Schlepper verfuegbar sind
    public:
        DockCondition   (   Model&          owner,
                            Tide&           theTide,
                            Res&            Tugs,
                            const String&   name = "it can dock")
            :   Condition   (owner, name),
                tide        (theTide),
                tugs        (Tugs)
            {}

        virtual bool    Check (const Entity&) const
                            // der Parameter wird hier nicht benoetigt
                        {   return tide.High() && tugs.Avail() >= 2; }

    private:
        Tide&       tide;   // der Gezeiten-Prozess
        Res&        tugs;   // die Schlepper
};

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell: Der Hafen

class HarbourWithTideModel : public HarbourModel
{   // Das Hafen-Modell erweitert um einen Gezeiten-Prozess
    // Es werden Schiffe erzeugt, die nur bei Nidrigwasser einlaufen koennen.
    public:
            // Zeitbasis: Stunden
        HarbourWithTideModel (  Model*      owner   =  0,
                        const   String&     name    = "HarbourWT",
                                unsigned    NofTugs = 3,
                                unsigned    NofQuays= 2);

        ~HarbourWithTideModel();

    protected:
        virtual void        DoInitialSchedules ();
                                // Generiert und aktiviert
                                // - das erste Schiff
                                // - den Tide-Prozess

        virtual Ship&       NewShip ();
                                // erzeugt ein gezeitenabhaengiges Schiff

    private:
        CondQueue           reede;  // Warteschlange fuer Schiffe
        DockCondition       highTideAndTugsAvail;
                                // Bedingung fuer das Andocken der Schiffe
        Tide                tide;   // Gezeitenprozess
};

// ----------------------------------------------------------------------------
```

```
#endif  // HARBOUR2_H
```

harbour2.cc

```
// Dateiname    : harbour2.cc
//
// Datum        : 08.03.1998
//
// Autor        : Thomas Schniewind
//
// ----------------------------------------------------------------------------

#include "harbour2.h"

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Modell: Der Hafen mit Gezeiten

HarbourWithTideModel::HarbourWithTideModel (Model* owner, const String& name,
                                            unsigned    NofTugs,
                                            unsigned    NofQuays)
    :   HarbourModel            (owner, name),
        reede                   (*this, "Reede"),
        highTideAndTugsAvail    (*this, tide, tugs),
        tide                    (*this, reede)
    {}

// ----------------------------------------------------------------------------

HarbourWithTideModel::~HarbourWithTideModel ()
{
    // Gezeiten-Prozess anhalten, damit es beim Loeschen keine Warnung gibt
    tide.Cancel();
}

// ----------------------------------------------------------------------------

void HarbourWithTideModel::DoInitialSchedules ()
{
    HarbourModel::DoInitialSchedules();
    tide.Activate (0.0);
}

// ----------------------------------------------------------------------------

Ship& HarbourWithTideModel::NewShip ()
{
    return *new TideDependentShip (*this,
                                   dockingStream,  loadingStream,
                                   tugs,           quays,
                                   reede,          highTideAndTugsAvail);
}

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// Gezeitenabhaengige Schiffe

TideDependentShip::TideDependentShip
    (HarbourModel&  owner,
     RealDist&      DockingStream,
     RealDist&      LoadingStream,
     Res&           Tugs,
     Res&           Quays,
     CondQueue&     Reede,
     Condition&     HighTideAndTugsAvail,
     const String&  name)
    :   Ship                    (owner, DockingStream, LoadingStream,
                                 Tugs, Quays, name),
        reede                   (Reede),
        highTideAndTugsAvail    (HighTideAndTugsAvail)
{}

// ----------------------------------------------------------------------------

void TideDependentShip::Dock()
{
    // Auf Tide und Tugs warten:
    reede.WaitUntil (highTideAndTugsAvail);
    // Andocken wie gehabt:
    Ship::Dock();
    // in Ship::Dock() wurden 2 Schlepper freigegeben
    // => Bedingung kann sich geaendert haben:
    reede.Signal();
}
```

```
// ------------------------------------------------------------------------

void TideDependentShip::UnDock()
{
    // Abdocken wie gehabt:
    Ship::UnDock();
    // in Ship::UnDock() wurde ein Schlepper freigegeben
    // => Bedingung kann sich geaendert haben:
    reede.Signal();
}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------
// Der Gezeiten-Prozess

void Tide::LifeCycle ()
{
    while (true)
    {
        Hold (4);
        lowTide = false;
        reede.Signal();
        Hold (9);
        lowTide = true;
    }
}

// ------------------------------------------------------------------------
```

harbour2.rpt

```
                              Clock Time = 8760.000
********************************************************************************
*                                                                              *
*                              Experiment: harbour2                            *
*                                                                              *
*                                     Report                                   *
*                                                                              *
********************************************************************************


                              Clock Time = 8760.000
********************************************************************************
*                                                                              *
*                                Model: HarbourWT                              *
*                                                                              *
********************************************************************************

HarbourWT reset at: 0.000

*** Hafenmodell mit Schleppern und Kais ***

Eingabedaten :          (auf Stundenbasis)
-----------------------------------------

Anzahl der Schlepper          :      3
Anzahl der Kais               :      2
Zwischenankunftszeit          :     10.000
mittlere Andockzeit           :      2.000
mit Standardabweichung von    :      0.200
mittlere Be-/Entladezeit      :     14.000
mit Standardabweichung von    :      3.000

                                  Cond-Queues
                                  -----------

Title           (Re)set    Obs    Qmax    Qnow      Qavg.  Zeros   avg.Wait All
-------------------------------------------------------------------------------
Reede             0.000    837       2       0      0.070    514       0.734 no

                                Distributions
                                -------------

Title           (Re)set    Obs Type              P a r a m e t e r s      Seed
-------------------------------------------------------------------------------
Arrival           0.000    837 Neg-Expon.       10.0000                 33427485
Landing           0.000   1672 Normal            2.0000       0.2000  22276755
Unload            0.000    836 Normal           14.0000       3.0000  46847980

                                  Resources
                                  ---------

Title           (Re)set  Users  Limit    Min    Now  Usage [%]   avg.Wait QMaxL
-------------------------------------------------------------------------------
Tugs              0.000   1671      3      0      1     19.217      0.037     1
Quays             0.000    835      2      0      0     80.063     18.388    12
```

```
                              Clock Time = 8760.000
*******************************************************************************
*                                                                             *
*                           End of Model: HarbourWT                           *
*                                                                             *
*******************************************************************************
```

6 Referenz der Schnittstellenobjekte

6.1 Übersicht

In diesem Abschnitt sollen die Klassen beschrieben werden, die der Modellprogrammierer benötigt, um ein Modell zu implementieren. Die Beschreibungen der Klassen und ihrer Verwendung sind alphabetisch geordnet, um ein schnelles Auffinden zu ermöglichen. Eine Einführung und Übersicht über die behandelten Klassen ist den Beschreibungen vorangestellt. Diese Übersicht ist thematisch gegliedert.

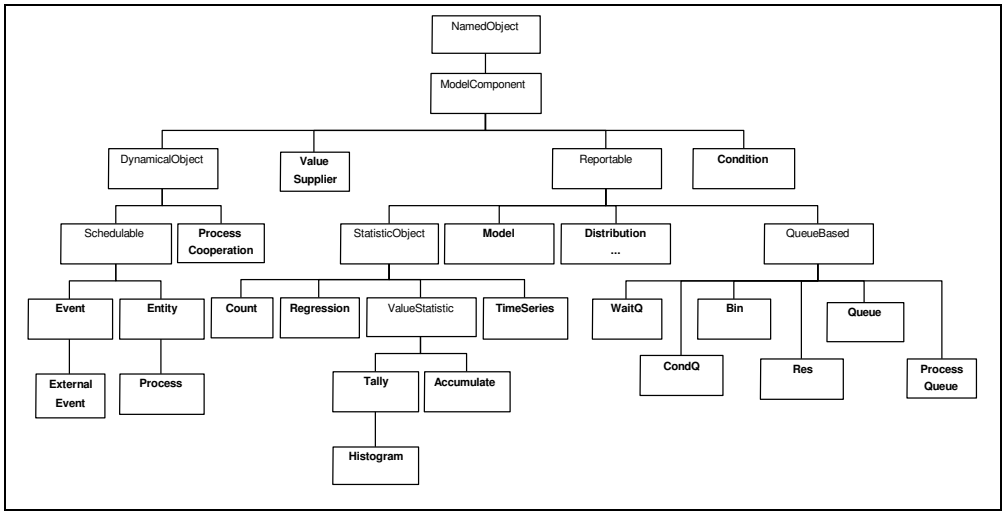

Abbildung 6-1: Klassendiagramm der Modellkomponenten

6.1.1 Experiment, Modell und Modellkomponenten

Um ein Simulationsexperiment durchführen zu können, muß ein Experiment (**`Experiment`**) erzeugt werden, über das die Simulation gesteuert wird. Es wird mit genau einem Modell (**`Model`**) verbunden, dem Hauptmodell. Jedes Modell hat ein Eigentümermodell, das bei der Erzeugung stets angegeben werden muß (eine Ausnahme bildet das Hauptmodell, das keinem anderen Modell untergeordnet ist). Ein Modell kann aus Modellkomponenten (**`ModelComponent`**) zusammengesetzt werden. Da Modelle Modellkomponenten sind, können sie auch aus anderen Modellen zusammengesetzt werden. So kann eine komplexe Modellhierarchie aufgebaut werden.

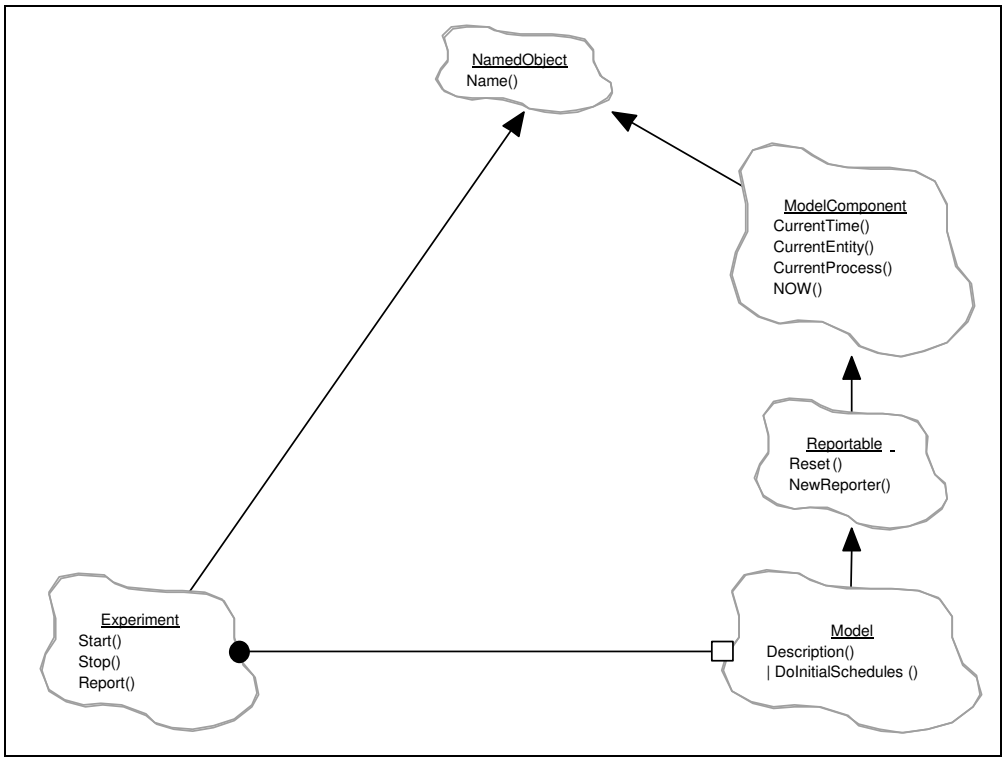

Abbildung 6-2: Klassendiagramm zu Experiment, Modell und Modellkomponente

6.1.2 Reportfähige Objekte (`Reportable`)

Über reportfähige Objekte können Informationen im Report ausgegeben werden. Hierzu zählen die Zufallszahlenströme (**`Distribution`**), Modelle (**`Model`**), Warteschlangenbasierte (**`QueueBased`**) und die statistischen Datensammelobjekte (**`StatisticObject`**).

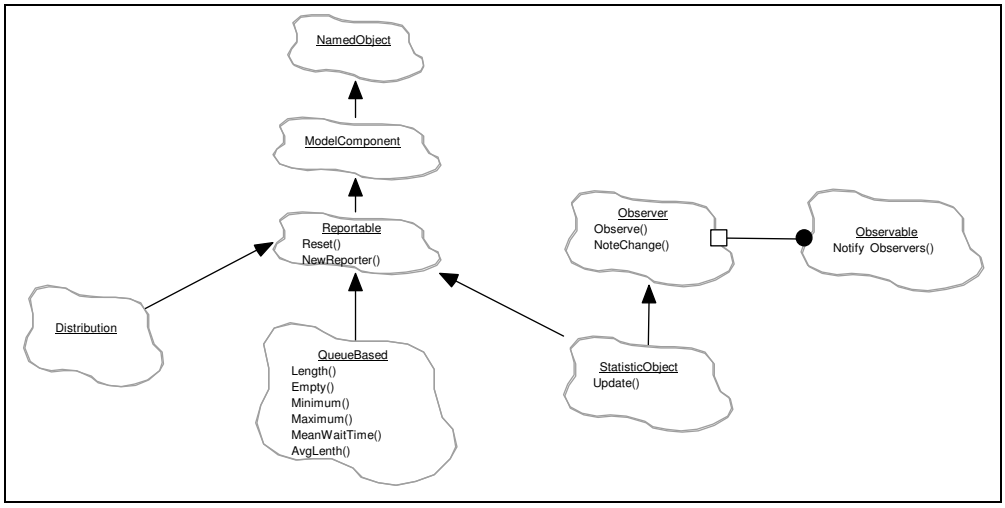

Abbildung 6-3: Klassendiagramm zu den reportfähigen Objekten (ohne die Klasse `Model`)

6.1.2.1 Zufallszahlenströme (`Distribution`)

Es gibt drei Arten von Zufallszahlenströmen (ZZ-Ströme), die sich im Typ der erzeugten Zufallszahlen unterscheiden. Ihre Bezeichner beginnen jeweils mit dem Namen der Oberklasse. Es sind dies boolesche (**`BoolDist`**), ganzzahlige (**`IntDist`**) und reelwertige ZZ-Ströme (**`RealDist`**).

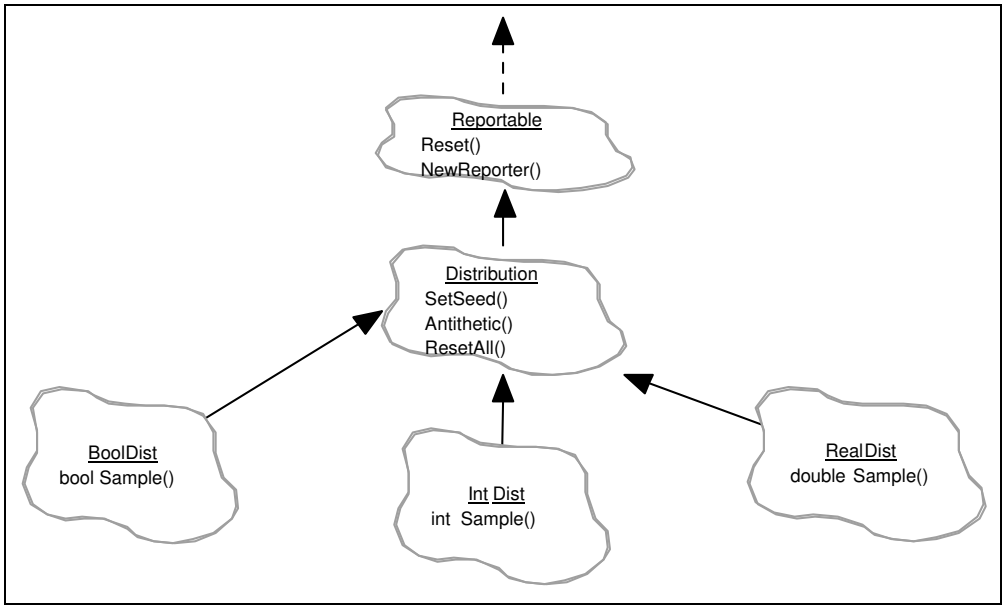

Abbildung 6-4: Klassendiagramm für drei Arten von Zufallszahlenströmen

Zu jeder Art ist jeweils ein konstanter ZZ-Strom verfügbar (**`BoolDistConst`**, **`IntDistConst`**, **`RealDistConst`**). Desweiteren sind verfügbar:

`BoolDistBernoulli`	ein anzugebender Wert bestimmt die Wahrscheinlichkeit, daß true geliefert wird
`IntDistEmpirical`	ganzzahliger ZZ-Strom mit empirischer Verteilung
`IntDistPoisson`	ganzzahliger ZZ-Strom mit Poisson-Verteilung
`IntDistUniform`	ganzzahliger ZZ-Strom
`RealDistEmpirical`	reelwertiger ZZ-Strom mit empirischer Verteilung
`RealDistErlang`	reelwertiger ZZ-Strom mit k-Erlang-Verteilung
`RealDistExponential`	reelwertiger ZZ-Strom mit negativ-exponentieller Verteilung
`RealDistNormal`	reelwertiger ZZ-Strom mit Gaußscher Normal-Verteilung
`RealDistUniform`	reelwertiger ZZ-Strom mit Gleichverteilung in einem Intervall

Tabelle 6-1: Die verschiedenen ZZ-Stromtypen

6.1.2.2 Warteschlangen und darauf basierende Objekte

Die Klasse **`QueueBased`** stellt zunächst die statistischen Funktionen bereit, die alle Unterklassen benötigen, deren Implementierung auf Warteschlangen beruht. Methoden zur Manipulation werden erst in den Unterklassen angeboten. Allgemeine Warteschlangen (**`Queue`**) können zusammen mit Entities verwendet werden. Für den Fall, daß in einer Warteschlange ausschließlich Prozesse warten sollen, können Prozeßwarteschlangen (**`ProcessQueue`**) benutzt werden. Die Klassen für Objekte zur Modellierung höherer Synchronisationsmechanismen erben ebenfalls von **`QueueBased`**, bieten jedoch keine Methoden zur direkten Manipulation der impliziten Warteschlange (`Insert`, `Remove`, …) an.

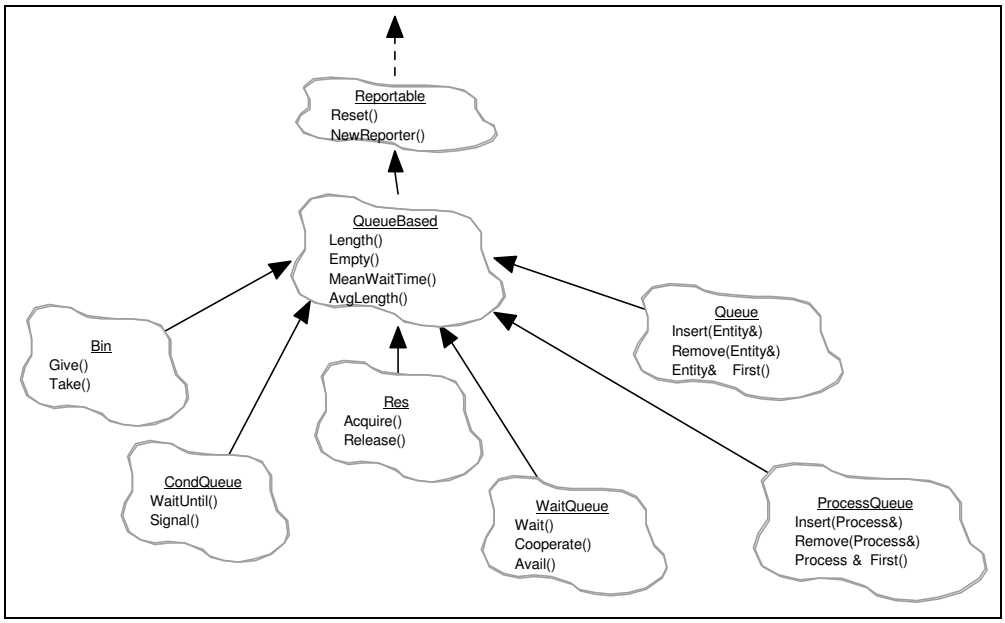

Abbildung 6-5: Klassendiagramm zu den Warteschlangenbasierten Objekten

6.1.2.3 Höhere Synchronisationsmechanismen

Prozesse können synchronisiert werden über Behälter (**`Bin`**) für zählbare, identitätslose Einheiten, mit denen Produzenten-Konsumenten-Beziehungen dargestellt werden können, über Warteschlangen für bedingtes Warten (**`CondQueue`**), in denen Prozesse solange blockiert werden, bis eine bestimmte Bedingung erfüllt ist, über die Nutzung Ressourcen (**`Res`**) begrenzter Kapazität sowie über die Äußerung zum Kooperationswunsch (**`WaitQueue`**) mit anschließender direkter Kooperation zweier Prozesse.

6.1.2.4 Statistische Datensammelobjekte (`StatisticObject`)

Mit Hilfe der Objekte dieses Teilbaums werden während der Simulation Daten gesammelt und statistisch ausgewertet. Bis auf Zähler (**`Count`**) für einfache Zählfunktionen werden diese mit einem Objekt verbunden, das den aktuellen Wert einer Beobachtungsgröße liefern kann (**`ValueSupplier`**). Ferner finden siche hier Klassen für Regressionsanalyse (**`Regression`**), die mit zwei Beobachtungsgrößen verbunden wird, Zeitreihen (**`TimeSeries`**) zur Protokollierung einer Beobachtungsgröße in eine Datei, zeitgewichtete Beobachtung (**`Accumulate`**), bei der die Werte der Beobachtungsgröße mit der jeweiligen Dauer ihres Auftretens gewichtet werden, nicht zeitgewichtete Beobachtung (**`Tally`**) und Histogramme (**`Histogram`**) zur Aufbereitung nicht zeitgewichteter Statistik in Form einer zeichenbasierten Balkengrafik. **`ValueStatistc`** ist dieOberklasse der Datensammelobjekte, die mit genau einem `ValueSupplier` verbunden werden. `TimeSeries` wird hier jedoch nicht mit untergeordnet, da die Werte nur protokolliert werden sollen, ohne sie statistische auszuwerten.

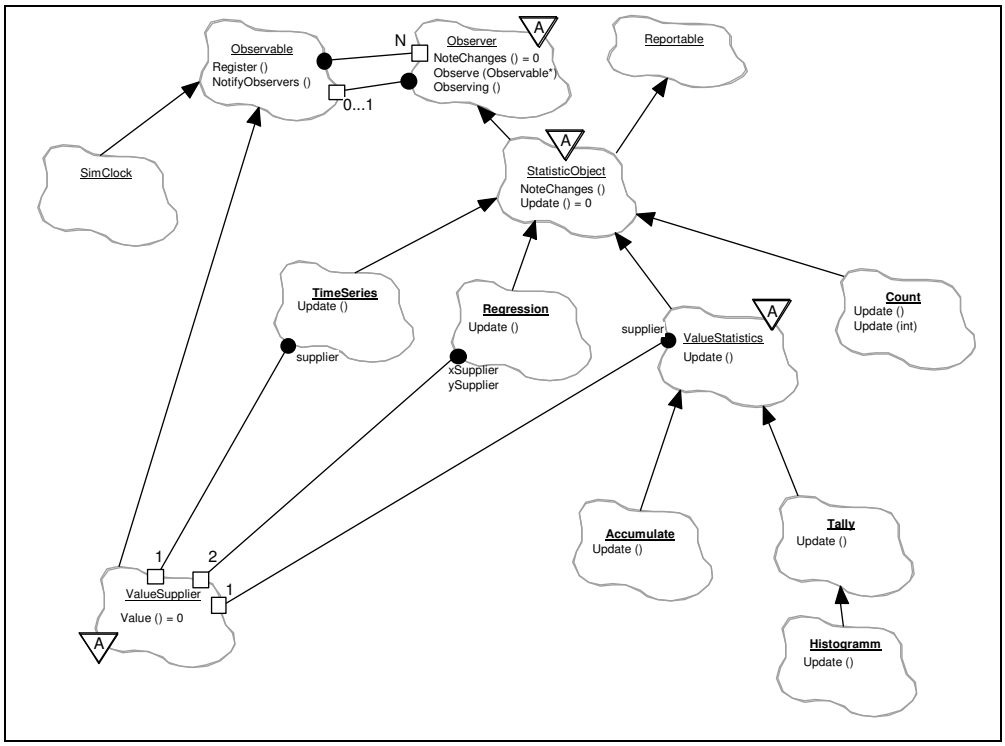

Abbildung 6-6: Klassendiagram der Datensammelobjekte

6.1.3 Dynamische Objekte (`DynamicalObject`)

Die Erzeugung von dynamischen Objekten wird in der Regel erst während der Simulation bestimmt und sollte stets über den new-operator von C++ erfolgen. So finden zunächst die Objekte, die vorgemerkt und so auf die Ereignisliste gesetzt werden können (**`Schedul-able`**). Dazu gehören Ereignisse (**`Event`**), Entities (**`Entity`**) und Prozesse (**`Process`**). Ereignisse werden entweder allein, als externe Ereignisse (**`ExternalEvent`**), vorgemerkt oder in Assoziation mit einem Entity. Prozesse können aktiviert und passiviert werden. Darüber hinaus können sie aber auch wie Entities behandelt werden, indem sie zusammen mit einem Ereignis vorgemerkt werden. Ein weiteres dynamisches Objekt tritt bei der direkten Prozeßkooperation in Erscheinung, und zwar Objekte, die die Kooperation zweier Prozesse repräsentieren (**`ProcessCooperation`**). Alle dynamischen Objekte werden spätestens beim Zerstören des zugehörigen Modells ihrerseits zerstört. Es kann aber auch eingestellt werden, daß die Zerstörung unmittelbar nach der Terminierung eines solchen Objekts stattfindet.

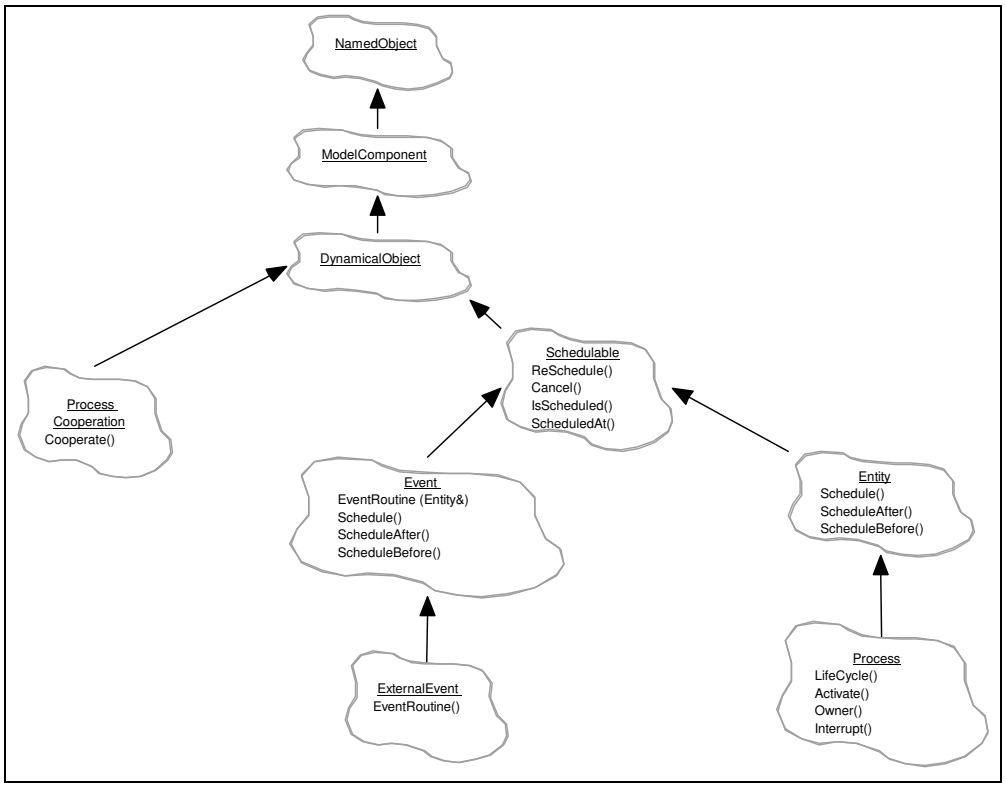

Abbildung 6-7: Klassendiagram für dynamische Objekte

6.1.4 Verdrängung mit Hilfe von 'NOW'

Um aktive Prozesse (`Current`) zu verdrängen, kann den Methoden zur Vormerkung (`Schedule`, `Activate`, etc.) als Simulationszeit der spezielle Wert `NOW` übergeben werden, was bewirkt, daß der gerade laufende Prozeß an die Spitze der Ereignisliste gesetzt und passiviert wird. Anschließend wird das auf diese Weise aktivierte Objekt (Ereignis oder anderer Prozeß) zum aktuellen (`Current`).

Ereignisse können jedoch nicht verdrängt werden. In diesem Fall wird das neue Objekt an die Spitze der Ereignisliste gesetzt und folglich erst behandelt, wenn die aktuelle Ereignisroutine abgearbeitet ist.

6.1.5 Objekte dynamisch anlegen

Die Implementierung der Koroutinen, die im DESMO-Kern verwendet wird, erfordert, daß Objekte, auf die von vielen stellen aus zugegriffen werden muß, sich auf dem Heap befinden. D.h. sie müssen mittels `new` dynamisch erzeugt oder Element eines dynamisch erzeugten Objekts sein. Dies gilt insbesondere für die Klassen Experiment und Modell.

6.1.6 Von DESMO nach DESMO-C

In DESMO-C haben siche einige Bezeichnungen von Methoden bzw. Klassen gegenüber den Moduln bzw. Prozeduren in DESMO geändert. Die folgende Tabelle soll es erleichtern, die aus DESMO bekannten Bezeichner in DESMO-C wiederzufinden.

Modul	**Bezeichner**	**Klasse**	**Methode**
CondQ	Condition	Condition	Check
eQueue	Condition	Condition	Check
eQueue	Find	Queue	First (Condition&)
eQueue	FindNext	Queue	Succ (Entity&, Condition&)
EventSimulation	Current	ModelComponent	CurrentEntity
EventSimulation	EvTime	Schedulable	ScheduledAt
EventSimulation	Idle	Schedulable	!IsScheduled
EventSimulation	NextEv	Schedulable	NextEntity
EventSimulation	NullEntity	ModelComponent	NullEntity
EventSimulation	Time	ModelComponent	CurrentTime
ProcessSimulation	Avail	Process	CanCooperate
ProcessSimulation	ClearInterrupt	Process	ClearInterruptCode
ProcessSimulation	CoOpt	Process	Cooperate
ProcessSimulation	Current	ModelComponent	CurrentProcess
ProcessSimulation	DisposeOnTermination	DynamicalObject	DeleteOnTermination
ProcessSimulation	EvTime	Schedulable	ScheduledAt
ProcessSimulation	Idle	Schedulable	!IsScheduled
ProcessSimulation	Interrupted	Process	GetInterruptCode
ProcessSimulation	Interrupted	ProcessCooperation	GetInterruptCode
ProcessSimulation	NextEv	ProcessCooperation	NextEntity
ProcessSimulation	NextEv	Schedulable	NextProcess
ProcessSimulation	NullEntity	ModelComponent	NullProcess
ProcessSimulation	Owner	Process	Master
ProcessSimulation	ReSchedule	Process	ReActivate
ProcessSimulation	Schedule	Process	Activate
ProcessSimulation	ScheduleAfter	Process	ActivateAfter
ProcessSimulation	ScheduleBefore	Process	ActivateBefore
ProcessSimulation	Time	ModelComponent	CurrentTime
Queue	Condition	Condition	Check
Queue	Find	ProcessQueue	First (Condition&)
Queue	FindNext	ProcessQueue	Succ (Process&, Condition&)
Res	DeadLockCheck	Experiment	DeadLockCheck
Res	Level	Experiment	DeadLockLevelT
WaitQ	Condition	Condition	Check
WaitQ	CoOpt	WaitQueue	Cooperate
WaitQ	Find	WaitQueue	Cooperate (ProcessCooperation&)

Tabelle 6-2: Abbildung von Bezeichenrn aus DESMO nach DESMO-C

6.2 Katalog

Im Folgenden wird in jedem Abschnitt eine eigene Klasse dargestellt. Dabei besteht eine Beschreibung aus sechs bis sieben Bereichen:

1. Basisklassen: Hier sind die Oberklassen aufgelistet und mit einer in eckigen Klammern angegebenen Nummer versehen, die die Ebene der Vererbungshierarchie wiedergibt. Je Größer die Nummer, desto kleiner ist der Abstand auf dem "Vererbungspfad". Die letzte Klasse in der Liste ist also stets die direkte Oberklasse. Die einzelnen Klassen sind durch Kommata getrennt, oder durch ein '+'-Zeichen, um Mehrfachvererbung darzustellen.

2. Assoziationen: Hier sind Klassen aufgelistet, zu denen zusätzliche Verbindungen bestehen, etwa weil sie als Parameter einer Methode übergeben oder von einer zurückgeliefert werden.

3. Datei: In dieser Zeile steht der Name, in der die beschriebene Klasse deklariert ist.

4. Beschreibung: Eine allgemeine Beschreibung der Klasse soll helfen, sich über ihren Sinn und Zweck sowie ihrer Verwendung zu informieren. Dabei werden manchmal die wichtigsten Methoden grob skizziert oder auf andere Klassen verwiesen, deren Beschreibung zum Verständnis beitragen.

5. geerbte Methoden: Hier sind alle Methoden aufgelistet, die von Oberklassen direkt oder indirekt geerbt werden. Die Liste ist alphabetisch sortiert. Außerdem trägt jede Methode in eckigen Klammern die Information, ob sie öffentlich (Kürzel 'ö') oder geschützt (Kürzel 'g'). Zusätzlich ist die Nummer der Oberklasse angegeben, in der die Methode zuletzt definiert wurde, so daß man dort nachschlagen kann, um weitere Informationen zu bekommen.

6. Methoden: Es werden alle öffentlichen Methoden aufgeführt und beschrieben, die in der Klasse als `public` deklariert sind.

7. geschützte Methoden: Evtl. folgt ein Abschnitt mit den als protected deklarierten Methoden.

6.2.1 Accumulate

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5], ValueStatistics[6]`

Assoziationen zu: `Model, Reporter, String, ValueSupplier`

in Datei: `accumula.h`

Beschreibung: `Accumulate` ist die Klasse von Datensammelobjekten zur statistischen Erfassung einer Beobachtungsgröße mit zeitlicher Gewichtung. Dabei wird ein `Accumulate`-Objekt mit einem Wertlieferanten (`ValueSupplier`) verbunden, der auf Verlangen den zu beobachtenden Wert (Beobachtungsgröße) berechnet und übergibt. Mit `Update` wird die Beobachtungsgröße ermittelt und die Statistik unter Berücksichtigung der zeitlichen Gewichtung aktualisiert.

Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, immer dann die Statistik fortzuschreiben, wenn sich die Beobachtungsgröße geändert hat. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Aktualisierung der Statistik mit Hilfe des vom `ValueSupplier`-Objekt gelieferten Wertes. `Observable` kann als Mixin-Klasse verwendet werden, um Objekte beobachtbar zu machen. Eine andere Alternative besteht in der Wahl für automatische Aktualisierung vor jedem Stellen der Simulationsuhr.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0.`

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Maximum[6ö], Mean[6ö], Minimum[6ö], Name[1ö], NewReporter[3ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[6ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], StdDev[6ö], TraceIsOn[2g],`

```
TraceNote[2g], TraceOff[2g], TraceOn[2g],
Update[6ö], Valid[2ö], valid[2g], Value[6ö],
Warning[2g]
```

Methoden:

```
Accumulate   (      Model&          owner,
              const String&         name,
                    ValueSupplier&  vs,
                    bool            automatic    = false,
                    bool            showInReport = true,
                    bool            showInTrace  = false)
```

Dem Konstruktor müssen mindestens das zugehörige Modell, ein Name und ein `ValueSupplier`-Objekt übergeben werden. Die Verbindung zwischen Modell und Datensammelobjekt bleibt über die gesamte Lebensdauer bestehen. In `vs` wird das Objekt übergeben, das auf Verlangen die Beobachtungsgröße berechnen und liefern kann. Wird für `automatic true` übergeben, so wird das Datensammelobjekt automatisch vor jedem Stellen der Simulationsuhr aktualisiert. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
Accumulate   (Model&          owner,
              ValueSupplier&  vs,
              bool            automatic    = false,
              bool            showInReport = true,
              bool            showInTrace  = false)
```

Der Konstruktor bietet eine Alternative zum ersten Konstruktor und unterscheidet sich von diesem nur dadurch, daß hier die Angabe für den Namen entfallen kann.

`void` **`Update`** `()`

virtuell, läßt sich vom `ValueSupplier`-Objekt den aktuellen Wert der Beobachtungsgröße liefern, um die Statistik zu aktualisieren. Wurde automatische Aktualisierung bei der Konstruktion eingestellt, so ist der Aufruf von `Update` überflüssig.

`double` **`Mean`** `() const`

liefert den zeitlich gewichteten Mittelwert der Beobachtungsgröße seit dem letzten Rücksetzen.

`double` **`StdDev`** `() const`

liefert die zeitlich gewichtete Standardabweichung der Beobachtungsgröße seit dem letzten Rücksetzen.

`void` **`Reset`** `() const`

setzt die bisher geführte Statistik zurück.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über das Datensammelobjekt berichten kann.

6.2.2 Bin

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], QueueBased[4]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `bin.h`

Beschreibung: Mit Hilfe der Klasse `Bin` läßt sich eine Form der Prozeßsynchronisation modellieren, bei der Prozesse voneinander unabhängig beliebige, nicht unterscheidbare Objekte ("Produkte") in einem Behälter ("bin") oder Puffer bereitstellen bzw. aus diesem entnehmen. Bei der Erzeugung eines Behälters kann angegeben werden, wieviele Produkte er bereits enthält. Mittels `Give` werden Produkte in den Behälter gegeben, mittels `Take` wieder entnommen. Ein negativer Inhalt ist nicht möglich.

Es wird eine implizite Warteschlange für Prozesse geführt, deren Nachfrage nicht sofort befriedigt werden kann. Nach Priorität und dann nach FIFO-Strategie wird entschieden, in welcher Reihenfolge den wartenden Prozessen die "Produkte" zugeteilt werden. Konsumierende Prozesse werden ggf. solange blockiert, bis ihr Bedarf befriedigt werden kann. Dabei blockiert ein wartender Prozeß mit einer hohen Anforderung u.U. hinter ihm wartende Prozesse, obwohl deren Anforderung vielleicht schon befriedigt werden könnte. Produzierende Prozesse brauchen nie zu warten.

Die von `QueueBased` geerbten Methoden beziehen sich auf die implizite Warteschlange der blockierten Konsumenten. Darüber hinaus stellt `Bin` Informationen über die Anzahl der produzierenden bzw. konsumierenden Zugriffe sowie maximale, mittlere und aktuelle Anzahl verfügbarer Produkte zur Verfügung.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0`.

geerbte Methoden: `AvgLength[4ö], AvgWaitTime[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Empty[4ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Length[4ö], MaxLength[4ö], MaxLengthAt[4ö], MaxWaitTime[4ö], MaxWaitTimeAt[4ö], MinLength[4ö], MinLengthAt[4ö], Name[1ö], NewReporter[4ö], NOW[2ö], NullEntity[2ö],`

```
NullEvent[2ö], NullProcess[2ö],
Observations[3ö], Out[2ö], QuotedName[1ö],
Rename[1ö], Reset[4ö], ResetAt[3ö],
SendMessage[2g], ShowInReport[3ö],
ShowInTrace[2ö], SkipTraceNote[2ö],
StdDevLength[4ö], StdDevWaitTime[4ö],
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g],
Warning[2g], ZeroWaits[4ö]
```

Methoden:

```
Bin     (Model& owner, const String& name = "",
         unsigned long initial      = 0,
         bool          showInReport = true
         bool          showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `initial` gibt an, wieviele Produkte der Behälter zu Beginn enthält. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

Bin `(const Bin& obj)`

Der Kopier-Konstruktor erzeugt aus `obj` einen neuen Behälter mit den selben statistischen Daten wie `obj`, jedoch enthält die implizite Warteschlange keine Prozesse, d.h. sie ist leer und somit gilt: `MinLength = 0` und `MinLengthAt = CurrentTime`.

~Bin `()`

Der Destruktor entfernt alle noch wartenden Prozesse aus der Warteschlange. Die Prozesse selbst werden nicht gelöscht.

`void` **Take** `(unsigned long n)`

Anforderung von `n` Produkten aus dem Puffer. Bei zu geringem Vorrat wird der betreffende Prozeß (nach Priorität bzw. FIFO) in eine Warteschlange eingefügt und solange blockiert, bis die Anforderung erfüllt werden kann. Anforderungen mit `n` = 0 werden auf jeden Fall ohne Verzögerungen bedient.

`void` **Give** `(unsigned long n)`

Bereitstellung von `n` Produkten im Puffer. Diese werden ggf. wartenden Prozessen gemäß der Reihenfolge in der Warteschlange zugeteilt. Der produzierende Prozeß wird weder blockiert noch verzögert.

`void` **Reset** `()`

virtuell, setzt die Statistik des Puffers zurück.

`unsigned long` **`Producers`** `() const`

liefert die Anzahl der produzierenden Zugriffe seit dem letzten Rücksetzen.

`unsigned long` **`Consumers`** `() const`

liefert die Anzahl der konsumierenden Zugriffe seit dem letzten Rücksetzen.

`unsigned long` **`Users`** `() const`

synonym zu `Producers`

`unsigned long` **`Initial`** `() const`

liefert die Anzahl von Produkten, die der Puffer bei seiner Erzeugung hatte (entspricht dem Konstruktor-Argument).

`unsigned long` **`Maximum`** `() const`

liefert den größten Inhalt des Puffers sei dem letzten Rücksetzen.

`unsigned long` **`Avail`** `() const`

liefert den aktuellen Inhalt des Puffers.

`double` **`AvgAvail`** `() const`

liefert den zeitgewichteten durchschnittlichen Inhalt des Puffers seit dem letzten Rücksetzen.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über den Puffer berichten kann.

6.2.3 BoolDist

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4]`

Assoziationen zu: `Model, String`

in Datei: `booldist.h`

Beschreibung: `BoolDist` ist Oberklasse für alle booleschen Zufallszahlenströme. Sie führt die Methode `Sample` ein, die einen Wert vom Typ `bool` liefert, der der gezogenen Zufallszahl entspricht. Die Methode wird erst in den Unterklassen definiert, die jeweils einen eigenen Verteilungstyp repräsentieren.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

`bool` **`Sample`** `() = 0`

virtuell, wird in Unterklassen definiert, so daß der gezogene boolesche Wert geliefert wird.

geschützte Methoden:

```
BoolDist      (Model& owner, const String& name = "",
               bool   showInReport = true
               bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann. Der Konstruktor ist geschützt, da nur Objekte der Unterklassen erzeugt werden sollen.

6.2.4 BoolDistBernoulli

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], BoolDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `booldist.h`

Beschreibung: `BoolDistBernoulli` ist die Klasse von Zufallszahlenströmen mit Bernoulli-Verteilung. Ein dem Konstruktor übergebener Wert bestimmt, mit welcher Trefferwahrscheinlichkeit die Methode `Sample` den Wert `true` liefert.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
BoolDistBernoulli  (      Model&  owner,
                    const String& name        = "",
                          double  probability  = 0.0,
                          bool    showInReport = true,
                          bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. Der Parameter `probability` gibt die Trefferwahrscheinlichkeit an, mit der zukünftige Aufruf von `Sample` den Wert `true` liefern. Er muß im Intervall [0,1] liegen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
bool Sample ()
```

liefert eine Zufallszahl aus dem ZZ-Strom.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String

`double` **`GetProbability`** `() const`

liefert den dem Konstruktor übergebenen Wert für die Trefferwahrscheinlichkeit.

`void` **`ChangeParameter`** `(double newProb)`

setzt die Trefferwahrscheinlichkeit auf den Wert `newProb`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void` **`checkProb`** `(const char* where)`

wird nach dem Ändern der Trefferwahrscheinlichkeit dazu verwendet, evtl. unzulässige Werte nach Ausgabe einer Warnung zu korrigieren.

6.2.5 BoolDistConst

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], BoolDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `booldist.h`

Beschreibung: `BoolDistConst` ist die Klasse von booleschen Zufallszahlenströmen, deren `Sample`-Aufrufe stets denselben Wert liefern. Konstante ZZ-Ströme werden benutzt, um ein Modell zunächst prototypisch zu entwickeln. Später können dann die konstanten ZZ-Ströme durch solche mit einer passenden Verteilung ersetzt werden. Ein dem Konstruktor übergebener Wert bestimmt das Ergebnis des `Sample`-Aufrufs.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
BoolDistConst      (      Model&  owner,
                    const String& name         = "",
                          bool    value        = true,
                          bool    showInReport = true,
                          bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. Der Parameter `value` gibt den Wert an, den zukünftige Aufrufe von `Sample` liefern. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`bool` **`Sample`** `()`

liefert den dem Konstruktor übergebenen Wert.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetValue`** `() const`

liefert den dem Konstruktor übergebenen Wert.

`void` **`ChangeParameter`** `(bool newValue)`

setzt den von `Sample` zu liefernden Wert auf `newValue`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

6.2.6 Condition

Basisklassen: `NamedObject[1], ModelComponent[2]`

Assoziationen zu: `Entity, Model, String`

in Datei: `conditio.h`

Beschreibung: Bedingungen werden benötigt, um z.B. in Warteschlangen bestimmte Entities zu finden (siehe z.B. bei `Queue::First`). In Unterklassen muß die rein virtuelle Methode `Check` definiert werden, um die gewünschte Bedingung zu implementieren. Sie soll `true` liefern, wenn das übergebene Entity die Bedingung erfüllt, andernfalls `false`.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], Name[1ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
Condition    (      Model& owner,
               const String& name        = "Condition"
                     bool    showInTrace = true)
```

Dem Konstruktor kann außer dem Modell, innerhalb dessen die Bedingung benutzt wird, auch ein Name übergeben werden, der z.B. in Warnmeldungen in Erscheinung treten kann. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann.

```
bool Check (const Entity& e) const = 0
```

rein virtuell, soll `true` liefern, wenn das Entity die Bedingung erfüllt, andernfalls `false`. Wird die Bedingung bei einer `WaitQueue` eingesetzt, so wird der Methode `Check` der Slave übergeben. Müssen für das Testen einer Kooperationsbedingung Master und Slave miteindader verglichenwerden, so kann in der Unterklasse von `Condition` der Master als Attribut der Bedingung angelegt werden. In der Methode `Check` kann dann der Slave, der als Parameter übergeben wird, mit dem Master verglichen werden.

6.2.7 CondQueue

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], QueueBased[4]`

Assoziationen zu: `Model, Reporter, SimTime, String`

in Datei: `condq.h`

Beschreibung: Mit Hilfe von der Klasse `CondQueue` lassen sich Synchronisationsmechanismen für bedingtes Warten modellieren, mit denen sich Prozesse blockieren können, bis eine anzugebende Bedingung erfüllt ist. Ein Prozeß reiht sich mittels `WaitUntil` in die Warteschlange ein und gibt dabei eine Bedingung an. Er wird nun solange blockiert, bis die Bedingung erfüllt ist. Eine wiederholte Überprüfung auf evtl. Erfüllung der Bedingungen muß jeweils explizit mittels `Signal` veranlaßt werden, liegt also voll in der Verantwortung des Modellprogrammierers. Auf eine automatische Überprüfung aller Wartenden Prozesse wird verzichtet, da sie einen erheblichen Verwaltungsaufwand erfordert und somit eine hohe Experimentlaufzeit bewirkt.

Auf die automatische Statistik kann über die von `QueueBased` geerbten Methoden zugegriffen werden.

geerbte Methoden: `AvgLength[4ö], AvgWaitTime[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Empty[4ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Length[4ö], MaxLength[4ö], MaxLengthAt[4ö], MaxWaitTime[4ö], MaxWaitTimeAt[4ö], MinLength[4ö], MinLengthAt[4ö], Name[1ö], NewReporter[4ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[4ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], StdDevLength[4ö], StdDevWaitTime[4ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g], ZeroWaits[4ö]`

Methoden:

```
CondQueue     (Model& owner, const String& name = "",
               bool checkAll     = false,
               bool showInReport = true
               bool showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `checkAll` gibt an, ob bei Zustandsänderungen stets für alle oder nur für einige der in der `CondQueue` wartenden Prozesse eine erneute Auswertung ihrer Synchronisationsbedingung erfolgen soll (s. `Signal`). `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`CondQueue` `(const CondQueue& obj)`

Der Kopier-Konstruktor erzeugt aus `obj` eine neue Warteschlange für bedingtes Warten mit den selben statistischen Daten wie `obj`, jedoch warten in ihr keine Prozesse, d.h. sie ist leer und somit gilt: `MinLength = 0` und `MinLengthAt = CurrentTime`.

`~CondQueue` `()`

Der Destruktor entfernt alle noch wartenden Prozesse aus der Warteschlange. Die Prozesse selbst werden nicht gelöscht.

`void` **`WaitUntil`** `(const Condition& c)`

Ein Prozeß wird durch Aufruf dieser Prozedur solange blockiert, bis die übergebene Synchronisationsbedingung erfüllt ist. Dabei wird es gemäß Priorität und FIFO-Strategie in die Warteschlange eingereiht. Ist beim Aufruf die Bedingung erfüllt, so erfolgt keine Blockierung.

`void` **`Signal`** `()`

Durch `Signal` wird eine erneute Überprüfung der Synchronisationsbedingung der in der `CondQueue` wartenden 'Prozesse veranlaßt. Die Prüfung endet beim ersten Prozeß, dessen Bedingung nicht erfüllt werden kann.

Wurde der `CondQueue` bei ihrer Erzeugung das Attribut `checkAll = true` zugeordnet, so werden sukzessiv jeweils sämtliche wartenden Prozesse überprüft. Dieses Vorgehen ist z.B. dann sinnvoll, wenn Prozesse mit verschiedenen Bedingungen in einer `CondQueue` eingetragen sind. Der Aufruf von `Signal` sollte jeweils dann erfolgen, wenn eine Zustandsänderung auch zur Erfüllung mindestens einer Bedingung in der betreffenden `CondQueue` führen könnte. Bei leerer Warteschlange hat die Prozedur keine Wirkung.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über die Warteschlange berichten kann.

6.2.8 Count

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5]`

Assoziationen zu: `Model, String`

in Datei: `count.h`

Beschreibung: `Count` ist die Klasse für einfache Zähler mit automatischer Reportausgabe. Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, Objektänderungen zu zählen. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Inkrementierung des Zählers.

`Count` implementiert die von `StatisticObject` geerbte rein virtuelle Methode `Update` so, daß der Zähler um 1 inkrementiert wird. Zusätzlich wird `Update` mit einem Parameter vom Typ `unsigned long` angeboten, der ein Inkrementierung um einen anzugebenden Wert ermöglicht. Die Methode Value liefert den aktuellen Zählerstand. Sie ist synonym zu der von `Reportable` geerbten Methode `Observations`. Die Rücksetzung eines `Count`-Objekts durch die von `Reportable` geerbte Methode `Reset` setzt den Zähler auf 0.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Update[5ö], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
Count  (Model& owner, const String& name = "",
        bool   showInReport = true
        bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Zähler bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`void` **`Update`** `()`

virtuell, inkrementiert den Zähler um 1. Diese Methode wird bei einer automatischen Aktualisierung durch ein beobachtetes Objekt aufgerufen.

`void` **`Update`** `(unsigned long n)`

inkrementiert den Zähler um `n`. Diese Methode wird bei einer automatischen Aktualisierung durch ein beobachtetes Objekt **nicht** aufgerufen.

`unsigned long` **`Value`** `() const`

liefert den aktuellen Zählerstand und entspricht der von `Reportable` geerbten Methode `Observations`.

6.2.9 Distribution

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3]`

Assoziationen zu: `Model, SimTime, String`

in Datei: `distrib.h`

Beschreibung: `Distribution` ist Oberklasse für alle Zufallszahlenströme. DESMO-C bietet drei Unterklassen von Distribution an: `BoolDist` für boolesche, `IntDist` für ganzzahlige und `RealDist` für reelwertige ZZ-Ströme. Jede dieser drei Unterklassen führt eine Methode `Sample` ein, die jeweils einen Wert vom Typ `bool` bzw. `int` bzw. `double` liefert, der der gezogenen Zufallszahl entspricht.

Startwerte für die erzeugten Zufallszahlenströme werden automatisch erzeugt, können aber auch "von Hand" (durch `SetSeed`) eingestellt werden. Unter Verwendung des Startwertgenerators können mehr als 275 Zufallszahlenströme generiert werden, aus denen jeweils über 120000 Zufallszahlen gezogen werden können, ohne daß es zu Überlappungen kommt.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

`String` **`GetType`**`() const = 0`

virtuell, wird in Unterklassen so definiert, daß die Typ-Bezeichnung des ZZ-Stroms als String geliefert wird.

`long` **`Seed`** `() const`

liefert den Startwert des ZZ-Stroms.

`void` **`SetSeed`** `(long newSeed)`

setzt den Startwert auf `newSeed`.

`void` **`SeedGenerator`** `(long newSeed) const`

setzt den Startwert des Startwertegenerators auf `newSeed`.

`void` **`Antithetic`** `()`

setzt den ZZ-Strom auf 'antithetisch'. Dabei wird auch der Startwert zurückgesetzt.

`void` **`AntitheticAll`** `()`

setzt alle zum selben Experiment gehörenden ZZ-Ströme auf 'antithetisch'. Dabei werden auch die Startwerte zurückgesetzt.

`void` **`ResetAll`** `()`

setzt alle zum selben Experiment gehörenden ZZ-Ströme zurück. Die Rücksetzung betrifft lediglich die Zählung der Ziehungen. Der ZZ-Strom selbst bleibt unberührt. Die Rücksetzung des Startwertes kann aber durch den Aufruf von `SetSeed(Seed())` erreicht werden.

geschützte Methoden:

```
Distribution (Model& owner, const String& name = "",
              bool   showInReport = true
              bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann. Die Konstruktoren sind geschützt, da nur Objekte der Unterklassen erzeugt werden sollen.

`double` **`random`** `()`

liefert eine Zufallszahl zwischen 0 und 1 und wird von den Unterklassen verwendet, um neue Verteilungsformen für ZZ-Ströme zu implementieren. Sie ist für Modellprogrammierer nicht von Bedeutung.

6.2.10 DynamicalObject

Basisklassen: `NamedObject[1], ModelComponent[2]`

Assoziationen zu: `Model, String`

in Datei: `dyobject.h`

Beschreibung: `DynamicalObject` ist Oberklasse für alle Objekte, deren Erzeugung sich in der Regel erst während des Simulationslaufes ergibt. Da in einem Modell meistens sehr viele Objekte dieser Klasse erzeugt werden und ihre Lebensdauer sehr unterschiedlich ist, wäre es unzumutbar, dem Modellprogrammierer aufzubürden, sich um die ordnungsgemäße Zerstörung zu kümmern. Dynamische Objekte werden daher vom System verwaltet und entweder bei ihrer Terminierung, falls dies gewünscht ist, oder mit dem Löschen des zugehörigen Modells zerstört. Dabei hängt es von der konkreten Klasse des Objekts ab, wann es als terminiert gilt:

`Event:`	nach dem Ende der Ereignisroutine
`Entity:`	nie
`Process:`	nach Ende des Lebenszyklus
`ProcessCooperation:`	nach Kooperationsende

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], Name[1ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
DynamicalObject    (      Model&  owner,
                    const String& name        = "",
                          bool    showInTrace = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Komponente bleibt über die gesamte Lebensdauer bestehen. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann. Das Objekt wird registriert, so daß es spätestens mit dem Löschen des zugehörigen Modells zerstört werden kann.

DynamicalObject `(const DynamicalObject&)`

Kopier-Konstruktor, registriert das neu konstruierte Objekt.

`void` **DeleteOnTermination** `()`

`DeleteOnTermination` markiert das Objekt für automatische Zerstörung nach seiner Terminierung. Dabei hängt es von der konkreten Klasse des Objekts ab, wann dies ist (s.o.). Der Aufruf `DeleteOnTermination` von kann nicht revidiert werden. Ist das Objekt danach terminiert, sollte es nicht mehr benutzt werden.

`void` **CheckDeleteOnTermination** `()`

liefert true, wenn zuvor `DeleteOnTermination` mindestens einmal aufgerufen wurde.

Die Methode `IsGarbage` wird intern benötigt und deshalb hier nicht beschrieben.

6.2.11 Entity

Basisklassen: `NamedObject[1], ModelComponent[2], DynamicalObject[3], Schedulable[4]`

Assoziationen zu: `Event, Model, Schedulable, SimTime, String`

in Datei: `entity.h`

Beschreibung: `Entity` ist Oberklasse für alle dynamischen Simulationsobjekte, die in Verbindung mit Ereignissen vorgemerkt werden können. Mit Entities können Objekte modelliert werden, die Informationen tragen und durch Eintreten von auf sie bezogenen Ereignissen manipuliert werden. So können in einem Fertigungssystem beispielsweise Aufträge als Entities betrachtet werden, die durch ihren Bearbeitungszustand charakterisiert sind. Ereignisse sorgen dafür, daß sich dieser Bearbeitungszustand mit der Zeit ändert, bis die Aufträge schließlich das System verlassen können. In diesem Sinne sind Entities passive Komponenten, die von anderen Teilen des Modells manipuliert werden.

Eine Erweiterung von Entities stellen Prozesse dar, die eigenes Verhalten in Form einer die Handlungen des Prozesses beschreibenden Methode aufweisen, und Entities so zu aktiven Komponenten machen.

Entities kann eine Priorität zugewiesen werden, die Einfluß auf die Behandlung in Warteschlangen hat. Dabei werden Objekte höherer Priorität beim Einfügen in Warteschlangen vor Objekten niedrigerer Priorität eingereiht, so daß sie eher wieder entnommen werden. Auf die Ereignisliste haben Prioritäten keinen Einfluß. Außerdem kann eingestellt werden, ob Entities in maximal einer Warteschlange zur Zeit oder in mehreren warten darf. Letzteres ist nützlich, wenn man Warteschlangen z.B. nur für statische Zwecke nutzt.

geerbte Methoden: `Cancel[4ö], CheckDeleteOnTermination[3ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], DeleteOnTermination[3ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsCurrent[4ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IsNull[4ö], IsScheduled[4ö], Name[1ö], Next[4ö], NextEntity[4ö], NextEvent[4ö], NextProcess[4ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], ReSchedule[4ö], ScheduledAt[4ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö],`

```
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]
```

Methoden:

Entity (Model& owner, const String& name = "",
bool showInTrace = true)

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und `Entity` bleibt über die gesamte Lebensdauer bestehen. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann. Entities besitzen eine Identität, weswegen Kopier-Konstruktor und Zuweisungsoperator privat deklariert und nicht implementiert sind.

~Entity ()

Der Destruktor entfernt das Entity zunächst aus allen Warteschlangen, in denen es noch wartet. Handelt es sich um das aktuelle (Current) Entity, so wird eine Fehlermeldung ausgegeben und das Experiment angehalten. Um das aktuelle Entity zu terminieren, benutze man `Delete`.

void **Delete** ()

bewirkt, daß das Entity gelöscht wird, sobald es nicht mehr das aktuelle (Current) ist. Steht es noch auf der Ereignisliste oder wartet es noch in einer Warteschlange, wird es nach Ausgabe einer Warnung aus diesen entfernt.

bool **IsNullEntity** () const

liefert `true`, wenn das Entity das Pseudo-Objekt `NullEntity` ist, sonst `false`

bool **IsProcess** () const

liefert `true`, wenn das Entity ein Prozeß ist, sonst `false`

void **Schedule** (SimTime dt, Event& ev)

setzt das Entity zusammen mit dem Ereignis `ev` zum Zeitpunkt `now + dt` auf die Ereignisliste. Ist das Entity oder `ev` bereits vorgemerkt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

void **ScheduleBefore** (Schedulable& before, Event& ev)

setzt das Entity zusammen mit dem Ereignis `ev` unmittelbar vor `before` auf die Ereignisliste. `before` muß vorgemerkt oder der laufende (`Current`) Prozeß sein, das Entity und `ev` dürfen nicht auf der Ereignisliste stehen, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert. Ist `before` der laufende Prozeß, so wird dieser verdrängt.

```
void ScheduleAfter (Schedulable& after, Event& ev)
```

setzt das Entity zusammen mit dem Ereignis `ev` unmittelbar nach `after` auf die Ereignisliste. `after` muß vorgemerkt oder der laufende (`Current`) Prozeß sein, das Entity und `ev` dürfen nicht auf der Ereignisliste stehen, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

```
PriorityT GetPriority () const
```

liefert die Priorität des Entities. Wurde sie nicht explizit gesetzt, so wird 0 zurückgegeben. Ein höherer Wert bedeutet eine höhere Priorität und umgekehrt. Entities mit höherer Priorität werden vor Entities mit niedrigerer Priorität in Warteschlangen eingereiht.

```
PriorityT SetPriority (const PriorityT newPriority)
```

setzt die Priorität des Entities auf `newPriority`.

```
QueueOption GetQueueOption() const
```

liefert `OnlyOneQueue`, wenn zuvor nicht explizit die andere Option eingestellt wurde, sonst `MultipleQueue`.

```
QueueOption SetQueueOption (QueueOption newOption)
```

setzt die Queue-Option für das Entity auf `newOption`. Achtung: wird die Option von MultipleQueue auf OnlyOneQueue zurückgesetzt, während sich das Entity in mehreren Warteschlangen befindet, so wird es, wenn es das nächste mal in eine Warteschlange eingefügt wird, automatisch aus den übrigen entfernt.

```
bool operator== (const Entity& en) const
```

```
bool operator!= (const Entity& en) const
```

Vergleich auf Identität.

Die folgenden Vergleichsoperatoren sind eine Abkürzung für den Vergleich der Prioritäten der Entities:

```
bool operator<  (const Entity& en) const
```

```
bool operator>  (const Entity& en) const
```

```
bool operator<= (const Entity& en) const
```

```
bool operator>= (const Entity& en) const
```

6.2.12 Event

Basisklassen: `NamedObject[1], ModelComponent[2], DynamicalObject[3], Schedulable[4]`

Assoziationen zu: `Entity, Model, Schedulable, SimTime, String`

in Datei: `event.h`

Beschreibung: `Event` ist Oberklasse für alle Simulationsereignisse. Ereignisse werden als externe Ereignisse (Unterklasse `ExternalEvent`) oder in Verbindung mit einem Entity angesetzt. Jede Unterklasse von `Event` stellt einen eigenen Ereignistyp dar. Das Modell reagiert auf das Eintreten eines Ereignisses mit bestimmten Aktionen, die den Systemzustand verändern, indem z.B. Attribute des assoziierten Entities manipuliert werden. Diese Aktionen werden in Ereignisroutinen zusammengefaßt, so daß jedem Ereignistyp, genau eine Ereignisroutine zugeordnet werden kann. Indem eine Ereignisroutine als eine Methode des korrespondierenden Ereignisses implementiert wird, kann diese 1:1-Beziehung auf natürliche Weise realisiert werden. Mit der Einführung eines neuen Ereignistyps, einer Unterklasse von `Event` also, muß die rein virtuelle Methode `EventRoutine` definiert werden, um die Reaktionen des Modells auf das Eintreten des Ereignisses zu beschreiben.

Ereignisroutinen bekommen als Parameter eine Referenz auf ein Entity übergeben, auf das sie sich beziehen. Sie werden stets komplett abgearbeitet, bevor andere Objekte aktiviert werden. Es ist also im Gegensatz zu Prozessen nicht möglich die Bearbeitung eines Ereignisses zu verdrängen oder zu unterbrechen.

In der Regel sind Ereignisse temporäre Objekte, d.h. jedes Ereignis wird neu erzeugt und nach Abarbeiten der Ereignisroutine wieder gelöscht. Es ist jedoch auch möglich, Ereignisse wiederzuverwenden. Daher werden Ereignisse nicht automatisch gelöscht. Es sollte jedoch nur in ganz besonderen Ausnahmen auf diese Technik zurückgegriffen. Statt dessen sollte in der Ereignisroutine stets der Aufruf von `DeleteOnTermination` erfolgen, was bewirkt, daß das Ereignis nach vollständiger Abarbeitung der Ereignisroutine automatisch gelöscht wird.

geerbte Methoden: `Cancel[4ö], CheckDeleteOnTermination[3ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], DeleteOnTermination[3ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsCurrent[4ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IsNull[4ö], IsScheduled[4ö], Name[1ö], Next[4ö], NextEntity[4ö], NextEvent[4ö],`

```
NextProcess[4ö], NOW[2ö], NullEntity[2ö],
NullEvent[2ö], NullProcess[2ö], Out[2ö],
QuotedName[1ö], Rename[1ö], ReSchedule[4ö],
ScheduledAt[4ö], SendMessage[2g],
ShowInTrace[2ö], SkipTraceNote[2ö],
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]
```

Methoden:

```
Event  (Model& owner, const String& name = "",
        bool   showInTrace = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und `Event` bleibt über die gesamte Lebensdauer bestehen. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann.

`bool` **`IsNullEvent`** `() const`

liefert `true`, falls das Ereignis das Pseudo-Ereignis `NullEvent` ist, sonst `false`.

`bool` **`IsExternal`** `() const`

liefert `true`, falls das Ereignis das externes Ereignis ist, sonst `false`.

`void` **`Schedule`** `(SimTime dt, Entity& en)`

Vormerken des Ereignisses zum Zeitpunkt `now + dt`. `en` ist das mit dem Ereignis assoziierte Entity. Weder das Ereignis noch en dürfen vorgemerkt sein.

`void` **`ScheduleBefore`** `(Schedulable& before, Entity& en)`

Vormerken des Ereignisses unmittelbar vor `before`. `before` muß vorgemerkt oder der laufende (`Current`) Prozeß sein. `en` ist das mit dem Ereignis assoziierte Entity. Weder das Ereignis noch en dürfen vorgemerkt sein.

`void` **`ScheduleAfter`** `(Schedulable& after, Entity& en)`

Vormerken des Ereignisses unmittelbar nach `after`. `after` muß vorgemerkt oder das aktuelle (`Current`) `Schedulable` sein. `en` ist das mit dem Ereignis assoziierte Entity. Weder das Ereignis noch en dürfen vorgemerkt sein.

geschützte Methoden:

`void` **`EventRoutine`** `(Entity& entity)`

rein virtuell, muß in den Unterklassen definiert werden, um zu beschreiben, wie auf das Eintreten des Ereignisses reagiert werden soll. In `entity` wird das Entity übergeben, das beim Ansetzen des Ereignisses mit diesem assoziiert wurde.

6.2.13 Experiment

Basisklassen: `NamedObject[1]`

Assoziationen zu: `ExperimentOpts, Message, Model, Output, SimTime, String`

in Datei: `experime.h`

Beschreibung: Um mit einem Modell zu experimentieren, muß zunächst ein Experiment erzeugt werden. Mit dem Experiment werden alle notwendigen internen Datenstrukturen für die Ablaufsteuerung des Simulationslaufes angelegt, in einer internen Datenstruktur (`ExperimentAccessory`) zusammengefaßt und mit der Zerstörung des Experiments wieder aus dem Speicher entfernt. Dadurch ist es möglich, innerhalb eines Programms beliebig viele Experimente hintereinander durchzuführen.

Jedes Experiment muß mit genau einem Modell verbunden werden, dem Hauptmodell, bevor es gestartet werden kann. Diese Verbindung wird von DESMO-C vorgenommen. Dazu wird das erste Modell ausgewählt, das als nach einem Experiment erzeugt wurde und diesem zugeordnet. Z.B. kann das Hauptmodell Element einer Unterklasse von Experiment sein, wodurch beide automatisch miteinander verbunden werden. Es kann aber auch ein Objekt von Experiment erzeugt werden und anschließend eines vom Hauptmodell. Z.B. wird in

```
1. Experiment* e = new Experiment ("MyExperiment");
2. Model*      m = new MyModel    (0, "MyModel");
```

`das Model m` mit dem aktuellen Experiment `e` verbunden.

Ein Experiment kann mittels `Start` bzw. `Stop` gestartet bzw. angehalten werden. Ein angehaltenes Experiment kann mittels `Continue` fortgesetzt werden.

Ein Experiment besitzt vier Ausgabekanäle: Debug, Error, Report und Trace. Zu jedem dieser Ausgabekanäle wird je eine Standardausgabedatei bereitgestellt, deren Name sich aus dem Namen des Experiments und der Endung ".dbg", ".err", ".rpt" oder ".trc" zusammensetzen (für Debug-, Error-, Report und Trace-Datei). Durch Aufruf von Report werden aktuelle Ergebnisse in den Reportausgabekanal geschrieben. Zur Unterstützung der Modellvalidierung kann über die Methoden `TraceOn` und `TraceOff` eine Ablaufverfolgung ein- bzw. ausgeschaltet werden. Bei eingeschalteter Debug-Funktion (`DebugOn`, `DebugOff`) werden nach jeder Änderung die Inhalte von Ereignisliste und Warteschlangen in den Debug-Kanal ausgegeben, um die Suche nach Programmierfehlern zu unterstützen. In seltenen Fällen können Fehler auftreten, die keinem Experiment zugeordnet werden können. Diese werden in die Standardfehlerausgabe von DESMO-C ausgegeben.

Weiterhin nutzt jedes Experiment drei Stream-Objekte für die Standardausgabe und -eingabe sowie die Fehlerausgabe. Als Vorgabe sind dies `cout`, `cin` und `cerr` aus `iostream.h`. Es können dem Experiment jedoch auch andere Objekte zugewiesen werden (`SetDesmoOut`, `SetDesmoErr`, `SetDesmoIn`). Um eigene Experimente und Modelle dieses Konzept unterstützen zu lassen können statt `cout`, `cerr` und `cin` die Methoden `Out`, `Err` und `In` genutzt werden, die das jeweilige aktuelle Stream-Objekt liefern.

geerbte Methoden: `ClassName[1ö], Name[1ö], QuotedName[1ö], Rename[1ö], Valid[1ö]`

Methoden:

`Experiment` `(const String& name, const ExperimentOpts& = ExperimentOpts())`

Konstruktor, setzt den Namen des Experiments auf name und nimmt evtl. gewünschte Optionen für das Experiment auf, wenn welche angegeben sind. Andernfalls werden die Standardoptionen eingestellt. Es werden alle zur Durchführung des Experiments benötigten Objekte angelegt mit Ausnahme des Modells, mit dem experimentiert werden soll. Dieses ist im Anschluß zu erzeugen, wodurch es automatisch mit dem aktuellen Experiment verknüpft wird. Erst dann kann das Experiment gestartet werden.

Der Kopier-Konstruktor ist als privat deklariert und nicht implementiert, um das Kopieren von Experimenten zu verhindern. Gleiches gilt für den Zuweisungsoperator.

`~Experiment` `()`

virtueller Destruktor, zerstört alle Datenstrukturen der Simulationskontrolle dieses Experiments. Das entsprechende Hauptmodell wird nicht zerstört. Dies muß vom Modellprogrammierer gewährleistet werden.

`void` **`Rename`** `(const String& newName)`

virtuelle Methode der Oberklasse `NamedObject`, wird überschrieben, um das Umbenennen eines Experiments zu verhindern. Der Aufruf wird ignoriert.

`void` **`Start`** `(SimTime t = 0.0)`

startet das Experiment mit Anfangszeit `t`. Das Experiment muß bereits mit einem Modell verknüpft sein, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert. Die Simulationsuhr wir auf die Anfangszeit t gestellt, bevor das erste Modell seine ersten Vormerkungen vornimmt. Das Experiment wir angehalten, wenn es entweder explizit mittels `Stop` angehalten wird, die Ereignisliste leer oder ein Fehler aufgetreten ist. Im Fehlerfall wird das Experiment umgehend gelöscht.

```
void Stop (SimTime dt = SimTime::Now())
```

hält das Experiment in der angegebenen Zeit an. Wird kein Zeitintervall angegeben, so wird das Experiment bei aktivem Prozeß unmittelbar (verdrängend), bei Bearbeitung eines Ereignisses nach Bearbeitung der zugehörigen Ereignisroutine angehalten. Wird eine von NOW verschiedene Zeit angegeben, so wird das Experiment erst nach Bearbeiten aller für den Zeitpunkt vorgemerkten Objekte angehalten. Ein unterbrochenes Experiment kann mittels `Continue` fortgesetzt werden.

```
void Continue ()
```

setzt ein unterbrochenes Experiment fort.

```
void Reset (SimTime dt = 0.0)
```

führt in `dt` Zeiteinheiten eine Rücksetzung aller zum Experiment gehörenden reportfähigen Objekte durch. Dabei wird bei laufendem Experiment bis zum Ende des aktuellen Simulationszeitpunktes gewartet. Ist dies nicht erwünscht, so kann die Zeitkonstante NOW() übergeben werden.

```
void Report (SimTime dt = 0.0)
```

gibt in `dt` Zeiteinheiten die aktuellen Ergebnisse in den Reportkanal aus. Dabei wird bei laufendem Experiment bis zum Ende des aktuellen Simulationszeitpunktes gewartet. Ist dies nicht erwünscht, so kann die Zeitkonstante NOW() übergeben werden.

```
String Description ()
```

virtuell, soll eine Beschreibung des Experiments liefern, die im Report ausgegeben wird. Hierzu muß Description überschrieben werden. Die Vorgabeimplementierung liefert einen leeren String.

```
bool TraceIsOn ()
```

liefert `true`, wenn die Ablaufverfolgung eingeschaltet ist.

```
void TraceOn (SimTime dt = SimTime::Now())
```

schaltet in `dt` Zeiteinheiten die Ablaufverfolgung ein.

```
void TraceOff (SimTime dt = SimTime::Now())
```

schaltet in `dt` Zeiteinheiten die Ablaufverfolgung aus.

```
bool DebugIsOn ()
```

liefert `true`, wenn der Debug-Modus eingeschaltet ist.

```
void DebugOn (SimTime dt = SimTime::Now())
```

schaltet in `dt` Zeiteinheiten den Debug-Modus ein.

`void` **`DebugOff`** `(SimTime dt = SimTime::Now())`

schaltet in `dt` Zeiteinheiten den Debug-Modus aus.

`SimTime` **`NOW`** `()`

static, liefert die Pseudo-Simulationszeit "jetzt sofort" und wirkt u.U. verdrängend. Die Übergabe von `NOW` als Aktivierungszeitpunkt von `Schedule` bzw. `Activate` hat bei aktiven Prozessen eine Verdrängung zur Folge, d.h. die Vormerkung bewirkt einen Ereignislisteneintrag vor dem aktiven Prozeß. Falls gerade eine Ereignisroutine behandelt wird, bewirkt die Vormerkung mit `NOW` die Eintragung auf der Ereignisliste unmittelbar nach dem gerade behandelten Ereignis. Die Übergabe von 0.0 hingegen bewirkt eine Vormerkung hinter allen anderen Ereignislisteneinträgen zum aktuellen Zeitpunkt.

`Model&` **`GetModel`** `()`

liefert eine Referenz auf das Hauptmodell des Experiments.

`const ExperimentOpts&` **`GetOpts`** `() const`

liefert eine Referenz auf die Optionen des Experiments (siehe dort). Die Experimentoptionen können im Nachhinein nicht verändert werden. Sie können lediglich bei der Erzeugung des Experiment dem Konstruktor übergeben werden.

Die folgenden Methoden erlauben einen direkten Zugriff auf die gleichnamigen Methoden der ExperimentOpts des Experiments:

```
int TimeWidth () const

int TimePrecision () const

int NameWidth () const

int NameNumberWidth () const

SimTime Epsilon () const
```

Die folgenden Methoden ermöglichen, ein Objekt der Klasse `Output` an jeweils einen der vier Ausgabekanäle zu binden bzw. diese Bindung wieder aufzuheben:

```
void AddDebugOutput (Output&)

void AddErrorOutput (Output&)

void AddReportOutput (Output&)

void AddTraceOutput (Output&)

void RemoveDebugOutput (Output&)

void RemoveErrorOutput (Output&)

void RemoveReportOutput (Output&)

void RemoveTraceOutput (Output&)
```

Die folgenden Methoden ermöglichen den Zugriff auf die jeweiligen Stream-Objekte der DESMO-C-Standardein- und ausgabe bzw. erlauben diese Objekte zu setzen, um Ein- und Ausgabe in andere Stream-Objekte zu leiten. Die Methoden sind als `static` deklariert, da sie sich nur auf ein Experiment sondern auf das ganze System beziehen:

```
ostream& Out ()

ostream& Err ()

ostream& In ()

void SetDesmoOut (ostream&)

void SetDesmoErr (ostream&)

void SetDesmoIn  (istream&)

void ResetDesmoIO ()
```

```
void SetSeed (long newSeed)
```

setzt den Startwert des Startwertgenerators für die Zufallszahlenströme. Alle bereits existierenden Zufallszahlenströme des Experiments erhalten ebenfalls einen neuen Startwert.

```
void Antithetic ()
```

setzt alle (auch bereits existierende) Zufallszahlenströme des Experiments auf 'antithetisch'.

```
enum DeadlockLevelT {Off, DynamicA, DynamicB}
void DeadlockLevel (DeadlockLevelT level)
```

Wird im Modell das Ressourcen-Konzept benutzt (`Res`), so können über DeadlockLevel verschiedene Verfahren zur Prüfung auf Zyklenfreiheit im Ressourcenallokationsgraph eingestellt werden:

`DynamicA`: jedesmal, wenn ein Prozeß versucht, eine Ressource zu belegen

`DynamicB`: jedesmal, wenn ein Prozeß auf Ressourcen-Zuteilung warten muß

Nach der ersten Ressourcen-Anforderung (durch `Acquire`) kann der Überwachungs-Level nicht mehr um-, sondern nur noch abgeschaltet werden (`Off`).

geschützte Methoden:

```
void Warning (const String& what, const String& where,
              const String& consequences = "",
              const String& hint         = "") const
```

gibt eine Warnung in den Error-Kanal des Experiments. `what` beschreibt die Warnung, `where`, wo sie aufgetreten ist, `consequences` die behandelnde Maßnahme und `hint` kann einen Hinweis zur zukünftigen Vermeidung dieser Meldung beinhalten. Eine Warnung hat keinen Einfluß auf den weiteren Programmverlauf.

```
void Error   (const String& what, const String& where,
              const String& consequences = "",
              const String& hint         = "") const
```

gibt eine Fehlermeldung in den Error-Kanal des Experiments. Anschließend wird versucht, das Experiment anzuhalten. `Error` sollte nur verwendet werden, wenn es nicht gelingt einen Fehler sinnvoll zu korrigieren, ansonsten sollte statt dessen `Warning` aufgerufen werden. Parameter siehe `Warning`.

```
void FatalError    (const String& what,
                    const String& where,
                    const String& consequences = "",
                    const String& hint = "") const
```

gibt eine Fehlermeldung in den Error-Kanal des Experiments. Anschließend wird das Programm mittels `abort` abgebrochen. In der Regel sollte die Verwendung von `Error` jedoch ausreichen. Parameter siehe `Warning`.

```
void SendMessage (const Message& msg) const
```

leitet eine beliebige Nachricht an den Nachrichtenmanager des Experiments, der diese an die entsprechenden Kanäle weiterleitet.

6.2.14 ExperimentOpts

Basisklassen: keine

Assoziationen zu: `SimTime`

in Datei: `expotps.h`

Beschreibung: Die Klasse ExperimentOpts stellt eine Datenstruktur zur Aufnahme der Optionen eines Experiments. Diese werden einem Experiment bei seiner Erzeugung übergeben. Werden sie nicht explizit angegeben, so werden die Standardoptionen übernommen, so wie sie im ExperimentOpts-Konstruktor durch die Standardargumente definiert werden.

Zu den Optionen gehören die Ausgabebreite und -genauigkeit von Simulationszeitwerten, die Breite, in der Namen ausgegeben werden, die Anzahl der Stellen, mit denen vormerkbare Objekte numeriert werden und die kleinste zu berücksichtigende Zeiteinheit Epsilon. Alle Werte können über entsprechende Zugriffsmethoden gelesen werden, die ebenfalls von der Klasse Experiment bereitgestellt werden.

geerbte Methoden: keine

Methoden:

```
ExperimentOpts      (int     timewidth     = 10,
                     int     timeprecision =  4,
                     int     namewidth     = 12,
                     int     numberwidth   =  2,
                     SimTime epsilon       =  0.00001)
```

Der Konstruktor kann als Standard-Konstruktor verwendet werden, da er ausschließlich Standardargumente besitzt. `timewidth` gibt an, in welcher Breite Objekte des Typs `SimTime` inkl. der Nachkommastellen ausgegeben werden sollen. `timeprecision` gibt die Zahl der auszugebenden Nachkommastellen der `SimTime`-Objekte inklusive des Dezimalpunktes an. Mit `namewidth` wird die Ausgabebreite von allen benannten Objekten (`NamedObject`) eingestellt. `numberwidth` bezieht sich auf die automatisch numerierten Objekte (`Schedulable`). So bedeutet der Wert 2, daß zwei Stellen für die Numerierung bereitgestellt werden, also wird modulo 2 numeriert (0..99). Die Zeitkonstante `epsilon` bestimmt, ab wann zwei Zeitpunkte als unterschiedlich angesehen werden. Dies hat Konsequenzen beim Stellen der Simulationsuhr, was erst geschieht, wenn die Differenz zwischen aktueller Zeit und nächstem Zeitpunkt größer als epsilon ist.

Die folgenden Methoden liefern den entsprechenden Wert, der dem Konstruktor übergeben wurde:

```
int TimeWidth () const

int TimePrecision () const

int NameWidth () const

int NameNumberWidth () const

SimTime Epsilon () const
```

6.2.15 ExternalEvent

Basisklassen: `NamedObject[1], ModelComponent[2], DynamicalObject[3], Schedulable[4], Event[5]`

Assoziationen zu: `Entity, Model, Schedulable, SimTime, String`

in Datei: `event.h`

Beschreibung: `ExternalEvent` ist Oberklasse für alle externen Simulationsereignisse. Externe Ereignisse sind Ereignisse, die nicht im Zusammenhang mit einem Entity stehen und als solche meist global auf ein Modell wirken. Im Gegensatz zu `Event` muß in Unterklassen von `ExternalEvent` die rein virtuelle, parameterlose Methode `ExternalEventRoutine` definiert werden, um die Reaktionen des Modells auf das Eintreten des externen Ereignisses zu beschreiben.

Die Methoden zum Vormerken aus der Oberklasse `Event` werden durch gleichnamige Methoden ersetzt, jedoch ohne den `Entity`-Parameter.

In der Regel sind Ereignisse temporäre Objekte, d.h. jedes Ereignis wird neu erzeugt und nach Abarbeiten der Ereignisroutine wieder gelöscht. Es ist jedoch auch möglich, Ereignisse wiederzuverwenden. Daher werden Ereignisse nicht automatisch gelöscht. Es sollte jedoch nur in ganz besonderen Ausnahmen auf diese Technik zurückgegriffen. Statt dessen sollte in der Ereignisroutine stets der Aufruf von `DeleteOnTermination` erfolgen, was bewirkt, daß das Ereignis nach vollständiger Abarbeitung der Ereignisroutine automatisch gelöscht wird.

geerbte Methoden: `Cancel[4ö], CheckDeleteOnTermination[3ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], DeleteOnTermination[3ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsCurrent[4ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IsNull[4ö], IsNullEvent[5ö], IsScheduled[4ö], Name[1ö], Next[4ö], NextEntity[4ö], NextEvent[4ö], NextProcess[4ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], ReSchedule[4ö], ScheduledAt[4ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
ExternalEvent       (      Model& owner,
                     const String& name = "",
                     bool  showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und `Event` bleibt über die gesamte Lebensdauer bestehen. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann.

`void` **`Schedule`** `(SimTime dt)`

Vormerken des Ereignisses zum Zeitpunkt `now + dt`. Das Ereignis darf nicht vorgemerkt sein.

`void` **`ScheduleBefore`** `(Schedulable& before)`

Vormerken des Ereignisses unmittelbar vor `before`. `before` muß vorgemerkt oder der laufende (`Current`) Prozeß sein. Das Ereignis darf nicht vorgemerkt sein.

`void` **`ScheduleAfter`** `(Schedulable& after)`

Vormerken des Ereignisses unmittelbar nach `after`. `after` muß vorgemerkt oder das aktuelle (`Current`) `Schedulable` sein. Das Ereignis darf nicht vorgemerkt sein.

geschützte Methoden:

`void` **`ExternalEventRoutine`** `() = 0`

rein virtuell, muß in den Unterklassen definiert werden, um zu beschreiben, wie auf das Eintreten des Ereignisses reagiert werden soll.

`void` **`EventRoutine`** `(Entity& entity)`

soll nicht benutzt werden und gibt bei Nichtbeachtung eine Warnung aus. Anschließend wird `ExternalEventRoutine` aufgerufen.

6.2.16 Histogram

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5], ValueStatistics[6], Tally[7]`

Assoziationen zu: `Model, Reporter, String, ValueSupplier`

in Datei: `histogra.h`

Beschreibung: `Histogram` ist die Klasse von Datensammelobjekten zur statistischen Erfassung einer Beobachtungsgröße ohne zeitliche Gewichtung. Sie erweitert die Funktionalität der Klasse Tally um ein Histogramm. Dabei wird ein `Histogram`-Objekt mit einem Wertlieferanten (`ValueSupplier`) verbunden, der auf Verlangen den zu beobachtenden Wert (Beobachtungsgröße) berechnet und übergibt. Mit `Update` wird die Beobachtungsgröße ermittelt und die Statistik aktualisiert. Für die Berechnung des Mittelwertes wird die Anzahl der `Update`-Aufrufe herangezogen.

Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, immer dann die Statistik fortzuschreiben, wenn sich die Beobachtungsgröße geändert hat. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Aktualisierung der Statistik mit Hilfe des vom `ValueSupplier`-Objekt gelieferten Wertes. `Observable` kann als Mixin-Klasse verwendet werden, um Objekte beobachtbar zu machen.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0.`

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Maximum[6ö], Mean[7ö], Minimum[6ö], Name[1ö], NewReporter[7ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[7ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö],`

```
SkipTraceNote[2ö], StdDev[7ö], TraceIsOn[2g],
TraceNote[2g], TraceOff[2g], TraceOn[2g],
Update[7ö], Valid[2ö], valid[2g], Value[6ö],
Warning[2g]
```

Methoden:

```
Histogram    (      Model&          owner,
               const String&        name,
                     ValueSupplier& vs,
                     double         lower = 0.0,
                     double         upper = 0.0,
                     unsigned       cells = 1,
                     bool           showInReport = true,
                     bool           showInTrace  = false)
```

Dem Konstruktor müssen mindestens das zugehörige Modell, ein Name und ein `ValueSupplier`-Objekt übergeben werden. Die Verbindung zwischen Modell und Datensammelobjekt bleibt über die gesamte Lebensdauer bestehen. In `vs` wird das Objekt übergeben, das auf Verlangen die Beobachtungsgröße berechnen und liefern kann. `lower` und `upper` geben das Intervall an, das in `cells` Zellen geteilt wird. Für jede dieser Zellen wird festgehalten, wie oft bei `Update` die Beobachtungsgröße in den Bereich der jeweiligen Zelle fiel. Zusätzlich wird je eine Zelle für Unter- und Überläufe angelegt. Es gilt also: In Zelle 0 werden die Unter- in Zelle `cells+1` die Überläufe registriert. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
Histogram    (const Histogram& obj)
```

Der Kopier-Konstruktor sorgt dafür, daß die interne Datenstruktur zur Zählung innerhalb der Zellen ordnungsgemäß kopiert wird.

```
void ChangeParameter     (double lower,
                          double upper,
                          unsigned cells)
```

ändert die dem Konstruktor übergebene Einstellung für die Zellaufteilung des Histogramms. Dies ist jedoch nur möglich, wenn bisher keine Beobachtungen aufgezeichnet sind (d.h. entweder vor dem ersten `Update` oder nach einem `Reset`).

```
void Update ()
```

virtuell, läßt sich vom `ValueSupplier`-Objekt den aktuellen Wert der Beobachtungsgröße liefern, um die Statistik zu aktualisieren.

```
unsigned Cells () const
```

liefert die Anzahl der Zellen im Histogramm und entspricht dem im Konstruktor oder `ChangeParameter` angegebenen untere Parameter. Es existieren jedoch zwei zusätzliche Zellen für Unter- und Überläufe.

`double` **`Lower`** `(unsigned cell = 1) const`

liefert die untere Grenze der Zelle `cell`. Zelle 0 ist die Zelle für die Unterläufe, also erhält man für `cell = 1` genau die dem Konstruktor oder `ChangeParameter` übergebene untere Grenze des ganzen Intervalls.

`double` **`Upper`** `() const`

liefert die obere Grenze des ganzen Histogramm-Intervalls und entspricht dem im Konstruktor oder `ChangeParameter` angegebenen untere Parameter.

`double` **`CellWidth`** `() const`

liefert die Breite einer einzelnen Zelle.

`unsigned long` **`ObservationsInCell`** `(unsigned cell) const`

liefert die Anzahl der registrierten Wert innerhalb der Zelle `cell`.

`unsigned` **`MostFrequentedCell`** `() const`

liefert die Nummer derjenigen Zelle, in der die meisten Werte registriert wurden. Existieren mehrere Zellen mit der gleichen Anzahl von Beobachtungen, so wird die Nummer der unteren Zelle geliefert.

`void` **`Reset`** `() const`

setzt die bisher geführte Statistik zurück.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über das Datensammelobjekt berichten kann.

6.2.17 IntDist

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4]`

Assoziationen zu: `Model, String`

in Datei: `intdist.h`

Beschreibung: `IntDist` ist Oberklasse für alle ganzzahligen Zufallszahlenströme. Sie führt die Methode `Sample` ein, die einen Wert vom Typ `int` liefert, der der gezogenen Zufallszahl entspricht. Die Methode wird erst in den Unterklassen definiert, die jeweils einen eigenen Verteilungstyp repräsentieren.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

`int` **`Sample`** `() = 0`

rein virtuell, wird in Unterklassen definiert, so daß der gezogene ganzzahlige Wert geliefert wird.

geschützte Methoden:

```
IntDist(Model& owner, const String& name = "",
              bool   showInReport = true
              bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann. Der Konstruktor ist geschützt, da nur Objekte der Unterklassen erzeugt werden sollen.

6.2.18 IntDistConst

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `intdist.h`

Beschreibung: `IntDistConst` ist die Klasse von ganzzahligen Zufallszahlenströmen, deren `Sample`-Aufrufe stets denselben Wert liefern. Konstante ZZ-Ströme werden benutzt, um ein Modell zunächst prototypisch zu entwickeln. Später können dann die konstanten ZZ-Ströme durch solche mit einer passenden Verteilung ersetzt werden. Ein dem Konstruktor übergebener Wert bestimmt das Ergebnis des `Sample`-Aufrufs.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
IntDistConst (Model& owner, const String& name = "",
              bool   value        = 0,
              bool   showInReport = true,
              bool   showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. Der Parameter `value` gibt den Wert an, den zukünftige Aufrufe von `Sample` liefern. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`int` **`Sample`** `()`

liefert den dem Konstruktor übergebenen Wert.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetValue`** `() const`

liefert den dem Konstruktor übergebenen Wert.

`void` **`ChangeParameter`** `(int newValue)`

setzt den von `Sample` zu liefernden Wert auf `newValue`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

6.2.19 IntDistEmpirical

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `intdist.h`

Beschreibung: `IntDistEmpirical` ist die Klasse von ganzzahligen Zufallszahlenströmen mit empirischer Verteilung. Dabei wird die empirische Verteilungsfunktion durch (x,y)-Wertepaare angegeben, wobei y der Wert der kumulativen Häufigkeit für x ist. Die Wertepaare werden über die Methode `AddEntry` eingebracht, wobei für alle n Paare (x,y) gilt:

(1) $x_i \leq x_{i+1}$
(2) $0 \leq y_i \leq y_{i+1} \leq y_n = 1$

Die Wertepaare sind in aufsteigender Reihenfolge durch wiederholte Aufrufe von `AddEntry` zu übergeben.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
IntDistEmpirical   (      Model& owner,
                    const String& name = "",
                    bool  showInReport = true,
                    bool  showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`int` **`Sample`** `()`

liefert eine Zufallszahl, die von der mittels `AddEntry` angegebenen Verteilung abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`void` **`AddEntry`** `(int newValue, double cumulativeFrequency)`

fügt ein neues Wertepaar mit x = `newValue` und y = `cumulativeFrequency` hinzu, wobei die in der Klassenbeschreibung angegebenen Bedingungen eingehalten werden müssen. Andernfalls wird der Aufruf nach Ausgabe einer Fehlermeldung ignoriert. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Erst wenn ein Wertepaar mit einer kumulativen Häufigkeit von 1 übergeben wurde, und nur dann ist der ZZ-Strom für `Sample`-Aufrufe bereit.

`unsigned` **`CountEntries`** `() const`

liefert die Anzahl der bisher mittels `AddEntry` übergebenen Wertepaare.

`int` **`GetValue`** `(unsigned n) const`

liefert den x-Wert des `n`-ten Eintrags ($0 \leq n <$ `CountEntries()`). Wird ein größerer Wert für n übergeben, so wird nach Ausgabe einer Warnung der größte x-Wert geliefert, oder 0, falls noch keine Wertepaare definiert sind.

`int` **`GetCumulativeFrequency`** `(unsigned n) const`

liefert den y-Wert des `n`-ten Eintrags ($0 \leq n <$ `CountEntries()`). Wird ein größerer Wert für n übergeben, so wird nach Ausgabe einer Warnung der größte y-Wert geliefert, oder 0, falls noch keine Wertepaare definiert sind.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

6.2.20 IntDistPoisson

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `intdist.h`

Beschreibung: `IntDistPoisson` ist die Klasse von ganzzahligen Zufallszahlenströmen mit Poisson-Verteilung. Ein dem Konstruktor übergebener Wert ist der Mittelwert der erzeugten Zufallszahlen.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
IntDistPoisson     (      Model&  owner,
                    const String& name         = "",
                          double  mean         = 0.0,
                          bool    showInReport = true,
                          bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `mean` ist der Mittelwert der zu ziehenden Zufallszahlen und darf nicht negativ sein. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`int` **`Sample`** `()`

liefert eine Zufallszahl, die vom dem Konstruktor übergebenen Mittelwert abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetMean`** `() const`

liefert den Mittelwert der zu ziehenden Zufallszahlen.

`void` **`ChangeParameter`** `(double newMean)`

setzt den Mittelwert der zu ziehenden Zufallszahlen auf `newMean`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Wird ein negativer Wert übergeben, so wird er nach Ausgabe einer Warnung invertiert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void checkMean (const char* where)`

wird nach dem Ändern des Mittelwertes dazu verwendet, evtl. negative Werte nach Ausgabe einer Warnung zu invertieren. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

6.2.21 IntDistUniform

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `intdist.h`

Beschreibung: `IntDistUniform` ist die Klasse von ganzzahligen Zufallszahlenströmen mit Gleichverteilung auf einem anzugebenden Intervall.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
IntDistUniform     (      Model& owner,
                    const String& name = "",
                    int   low          = 0,
                    int   high         = 0,
                    bool  showInReport = true,
                    bool  showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `low` und high bilden das Intervall, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind. Ist `high` < `low`, werden die Werte nach Ausgabe einer Warnung vertauscht. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`int` **`Sample`** `()`

liefert eine Zufallszahl, die vom dem Konstruktor übergebenen Intervall abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`int` **`GetLow`** `() const`

liefert die untere Intervallgrenze, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind.

`int` **`GetHigh`** `() const`

liefert die obere Intervallgrenze, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind.

`void` **`ChangeParameter`** `(int newLow, int newHigh)`

setzt das Intervall, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind, auf [`newLow`, `newHigh`]. Ist `high` < `low`, werden die Werte nach Ausgabe einer Warnung vertauscht. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void` **`checkHiLo`** `(const char* where)`

wird nach dem Ändern der Intervallgrenzen dazu verwendet, evtl. vertauschte Werte nach Ausgabe einer Warnung auszuwechseln. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

6.2.22 InterruptCode

Basisklassen: `NamedObject[1]`

Assoziationen zu: `String`

in Datei: `interrup.h`

Beschreibung: Um Prozesse zu unterbrechen, wird an ihnen die Methode `Interrupt` aufgerufen. Als Grund für die Unterbrechung wird ein Objekt der Klasse `InterruptCode` übergeben, anhand dessen der unterbrochene Prozeß entscheiden kann, ob er unterbrochen wurde und wenn ja, welche Ursache für die Unterbrechung vorliegt. Für jeden Unterbrechungsgrund muß also genau ein Objekt neu erzeugt werden. Zuweisung und Kopierkonstruktion erhalten jedoch die Identität, die auf der intern vergebenen Seriennummer basiert, nicht auf den Adressen der Objekte. `NoInterrupt` liefert das vordefinierte Objekt, das den Fall "keine Unterbrechung" repräsentiert.

geerbte Methoden: `ClassName[1ö], Name[1ö], QuotedName[1ö], Rename[1ö], Valid[1ö]`

Methoden:

`InterruptCode` `(const String& name = "")`

der Konstruktor liefert eine neue Unterbrechungsursache mit einem neuen Code und dem Namen `name`. Da der Vergleich auf Gleichheit über den internen Code implementiert ist, können Interrupt-Codes kopiert werden, d.h. die Übergabe als Wert ist möglich.

`const InterruptCode&` **`NoInterrupt`** `()`

static, liefert den InterruptCode, der "keine Unterbrechung" signalisiert.

`bool` **`operator==`** `(const InterruptCode& rhs) const`

liefert true, wenn die Codes der beiden Unterbrechungsursachen gleich sind, andernfalls false.

`bool` **`operator!=`** `(const InterruptCode& rhs) const`

liefert true, wenn die Codes der beiden Unterbrechungsursachen verschieden sind, andernfalls false.

`unsigned long` **`GetCode`** `() const`

liefert den internen Code der Unterbrechungsursache.

6.2.23 Message

Basisklassen: keine

Assoziationen zu: `Entity, Event, Experiment, Model, ModelComponent, Process, Schedulable, SimTime, String`

in Datei: `message.h`

Beschreibung: `Message` ist Oberklasse aller im System verwendeten Nachrichten. Jede Nachricht besitzt einen Typ und einen Code. Bei den Typen wird unterschieden zwischen `globalError`, `error`, `report`, `trace` und `debug`. Er wird herangezogen, um die Nachricht in einen entsprechenden Ausgabekanal zu lenken. Der Code dient der Kennzeichnung besonderer Unterarten. Z.B. können Meldungen des Typs error mit Hilfe des Codes in fatale Fehler, normale Fehler und Warnungen differenziert werden. Die Schnittstelle ist relativ breit, da alle Nachrichtentypen abgedeckt werden müssen.
Der erste Methodenblock dient dem Zugriff auf aktuelle Informationen des laufenden Experiments wie aktuelle Zeit, aktuelles Ereignis etc., welche erst im Moment des Zugriffs beschafft werden. Aus diesem Grund dürfen Nachrichten nur als temporäre Objekte verwendet und nicht längerfristig gespeichert werden.
Die virtuelle Methode `Description` wird in der Regel von Unterklassen überschrieben und charakterisiert den eigentlichen Inhalt der Nachricht. Der geschützten Methoden bedienen sich Unterklassen, um den Inhalt etwas aufzubereiten.

geerbte Methoden: keine

Methoden:

```
enum MessageType   {globalError, error, report,
                    trace,       debug}
```

Aufzählungstyp mit den fünf verwendeten Nachrichtentypen:

`globalError`:	Fehlermeldungen, die nicht einem bestimmten Experiment zugeordnet werden können
`error`:	allgemeine Fehlermeldungen
`report`:	Nachrichten für die Reportausgabe
`trace`:	Nachrichten für die Traceausgabe
`report`:	Nachrichten für die Debugausgabe

```
Message (MessageType mt, codeType ct = normal)
```

Dem Konstruktor muß ein Nachrichtentyp übergeben werden, damit die Nachricht zu den entsprechenden Ausgabekanälen gelangt. Der Code wird insbesondere bei Fehlermeldungen benötigt, um die Fehlerarten zu unterscheiden.

`SimTime` **`Time`** `() const`

liefert die aktuelle Simulationszeit des gerade laufenden Experiments.

`const Event&` **`GetEvent`** `() const`

liefert das aktuelle Ereignis des gerade laufenden Experiments. Kann keines bestimmt werden, so wird das Pseudo-Ereignis `NullEvent` zurückgegeben.

`const Entity&` **`GetEntity`** `() const`

liefert das aktuelle Entity des gerade laufenden Experiments. Kann keines bestimmt werden, so wird das Pseudo-Ereignis `NullEntity` zurückgegeben.

`const Process&` **`GetProcess`** `() const`

liefert den aktuellen Prozeß des gerade laufenden Experiments. Kann keiner bestimmt werden, so wird der Pseudo-Prozeß `NullProcess` zurückgegeben.

`const Model&` **`GetModel`** `() const`

liefert das aktuelle Modell des gerade laufenden Experiments.

`const Experiment&` **`GetExperiment`** `() const`

liefert das gerade laufende Experiment.

`MessageType` **`Type`** `() const`

liefert den Typ der Nachricht, der dem Konstruktor übergeben wurde.

`int` **`Code`** `() const`

liefert den Code der Nachricht, der dem Konstruktor übergeben wurde.

`String` **`CodeText`** `() const`

liefert den Code der Nachricht in textueller Form.

`String` **`Description`** `() const`

virtuell, muß von Unterklassen überschrieben werden, um den Inhalt der Nachricht wiederzugeben.

`String` **`Location`** `() const`

virtuell, kann von Unterklassen überschrieben werden, um Informationen über den Entstehungsort der Nachricht wiederzugeben. Wird in der Regel nur von Fehlermeldungen benötigt.

`String` **`Consequences`** `() const`

virtuell, kann von Unterklassen überschrieben werden, um zu kommentieren, wie auf die Nachricht reagiert wird. Wird in der Regel nur von Fehlermeldungen benötigt.

`String` **`Hint`** `() const`

virtuell, kann von Unterklassen überschrieben werden, um zu dokumentieren, wie die Entstehung dieser Nachricht vermieden werden kann. Wird in der Regel nur von Fehlermeldungen benötigt.

geschützte Methoden:

`String` **`Quote`** `(const String& s) const`

setzt s in Hochkommata und liefert das Ergebnis zurück.

`String` **`NameAndModel`** `(const ModelComponent& mc) const`

liefert einen String zusammengesetzt aus dem Namen von mc und dem Namen des zugehörigen Modells zurück.

`String` **`TxtDtToAt`** `(const SimTime& dt) const`

wandelt die Zeitdifferenz `dt` in eine absolute Zeit um und stellt das Wort 'at' voran. Ggf. wird die Textkonstante 'NOW' substituiert, falls `dt` = NOW, oder die Textkonstante 'now', falls `dt` = 0.0.

`String` **`TxtTimeToAt`** `(const SimTime& dt) const`

stellt `dt` das Wort 'at' voran. Ggf. wird die Textkonstante 'now' substituiert, falls `dt` = 0.0.

`String` **`TxtItselfIfCurrent`** `(const Schedulable& s) const`

falls s ein gerade aktuelles vormerkbares Objekt ist, wird die Textkonstante 'itself' geliefert, andernfalls der Name des Objekts in Hochkommata.

6.2.24 MessageReceiver

Basisklassen: keine

Assoziationen zu: `Message, Reporter`

in Datei: `messager.h`

Beschreibung: `MessageReceiver` ist die Oberklasse von allen Objekten, die Nachrichten (`Message`) empfangen können. Hierzu zählen insbesondere die Stdandardausgaben von DESMO-C, die jeweils Objekte der Klassen `StdDebug`, `StdError`, `StdReport` und `StdTrace` sind. Reportausgaben müssen zudem in der Lage sein, mit Reportern umzugehen und die Methode `TakeReporter` überschreiben. Beide Methoden sind mit einer leeren Implementierung versehen, so daß nur die jeweils benötigte überschrieben werden muß.

geerbte Methoden: keine

Methoden:

`void` **`Note`** `(const Message& msg)`

virtuell, soll von Unterklassen überschrieben werden, um die Reaktion auf eine Nachricht zu implementieren.

`void` **`TakeReporter`** `(Reporter& r)`

virtuell, soll von Unterklassen überschrieben werden, um die Informationen des Reporters auszugeben.

6.2.25 Model

Basisklassen: `NamedObject[1]`, `ModelComponent[2]`, `Reportable[3]`

Assoziationen zu: `Experiment`, `ModelComponent`, `Reporter`, `QueueOption`, `SimTime`, `String`

in Datei: `model.h`

Beschreibung: Ein Modell beschreibt den Aufbau des zu untersuchenden Systems. Es enthält alle Informationen über die in ihm gekapselten Objekte (Modellkomponenten) sowie deren Interaktionen und Verhalten. Außerdem beschreibt es, wie es von außen beeinflußt werden kann und welche Reaktionen es nach außen haben kann. Die Konstruktion eines Modells sollte komplett im Konstruktor geschehen. Dort sollten alle statischen Komponenten des Modells erzeugt werden (z.B. Warteschlangen, Synchronisationspunkte, Statistikobjekte). Der Destruktor muß diese Komponenten entsprechend wieder zerstören. Da `Model` von `ModelComponent` (über `Reportable`) erbt, sind Modelle ebenfalls Modellkomponenten, so daß ein Modell beliebig viele Submodelle enthalten kann. Auf diese Weise lassen sich ganze Modellhierarchien erzeugen. Das Modell an der Spitze dieser Hierarchie ist das Hauptmodell.

Um ein Experiment an einem Modell durchzuführen, müssen Modell und Experiment miteinander verbunden werden. Hierfür wird zunächst das Experiment erzeugt. Aber ohne ein Modell kann das Experiment nicht gestartet werden. Deswegen wird das erste Hauptmodell, was nach einem Experiment erzeugt wird, diesem zugeordnet. In einer Modellhierarchie kann nur das Hauptmodell mit einem Experiment verbunden werden. Es zeichnet sich dadurch aus, daß dem Konstruktor anstelle eines anderen als übergeordnetes Modell, ein Zeiger auf sich selbst oder ein Null-Zeiger übergeben wird.

So ergeben sich zwei Varianten zur Erzeugung von Experiment und Modell:

1. Es wird eine Unterklasse von Experiment gebildet, die ein Modell enthält:

```
 1. class MyModel : public Model {...};
 2.
 3. class MyExperiment : public Experiment {
 4.     public:
 5.         MyExperiment () : Experiment ("anExperiment"),
 6.                           myModel (0, "aModel")
 7.                           {}
 8.     private:
 9.         MyModel   myModel;
10. };
11.
12. MyExperiment* myExp = new MyExperiment;
```

Damit würde mit der Erzeugung eines Objektes der Klasse `MyExperiment` in dessen Konstruktor-Initialisiererliste das Modell `myModel` als Hauptmodell angelegt und automatisch mit dem zuletzt erzeugten Experiment (also `myExp`) verbunden.

2. Es wird erst ein Experiment erzeugt und anschließend ein Hauptmodell:

```
1. Experiment* myExp   = new Experiment ("anExperiment");
2. MyModel*    myModel = new MyModel    (0, "aModel");
3. ...
4. myExp->Start();
5. myExp->Report();
6. ...
7. delete myModel;
8. delete myExp;
```

mit der zweiten Zeile wird das Modell `myModel` automatisch mit dem Experiment `myExp` verbunden. Es kann sodann gestartet und ein Report generiert werden. Anschließend muß darauf geachtet werden, daß Modell und Experiment in umgekehrter Reihenfolge gelöscht werden:

Die virtuelle Methode `DoInitialSchedules` kann in Unterklassen überschrieben werden, um die ersten Vormerkungen vorzunehmen. Sie wird automatisch aufgerufen, nachdem `Experiment::Start` aufgerufen wurde. Alle Modelle vom Hauptmodell bis zu den Modellen an den Blättern der Hierarchie, werden aufgefordert, ihre ersten Vormerkungen vorzunehmen, indem jeweils die Methode `DoInitialSchedules` aufgerufen wird. Dabei sollten Modelle so konzipiert sein, daß das Verhalten von `DoInitialSchedules` über die Parameter des Modells gesteuert werden kann, um eine Verwendung als Submodell zu ermöglichen.

Bei der Verwendung von Modellen als Submodell gibt es jedoch ein Problem, wenn Modellkomponenten zwischen zwei Modellen ausgetauscht werden oder miteinander interagieren sollen. Es wird in verschiedenen Fällen notwendig sein, Modellkomponenten (insbes. Entities) von einem allgemeinen Typ in einen speziellen Typ zu konvertieren (z.B. von `Entity` nach `Job`), um auf die speziellen

Eigenschaften zugreifen zu können. Bei Systemen ohne RTTI[63] ist dies nur gefahrlos möglich, wenn gewährleistet ist, daß alle Komponenten, die in das Modell gelangen, in diesem bekannt sind. Es kann dann in eine dem Modell bekannte Unterklasse der allgemeinen Komponente konvertiert werden, die nähere Informationen zum speziellen Typ liefern kann. Dieser Mechanismus muß innerhalb des Modellkontextes implementiert werden.

Um die Kompatibilität zweier Komponenten sicherzustellen, wird bei allen Operationen, bei denen Komponenten übergeben werden (z.B. `Queue::Insert`) geprüft, ob die Komponenten zum selben Modell gehören. Für diesen Test wird jeweils die Methode `IsModelCompatible` aufgerufen, die die Überprüfung an die Methode `CheckCompatibility` des zugehörigen Modells weiterleitet. Dort wird in der Regel geprüft, ob beide Komponenten zum selben Modell (Vergleich der Adressen) gehören.

Stehen geeignetere Verfahren zur Beurteilung der Kompatibilität zur Verfügung, so kann CheckCompatibility entsprechend überschrieben werden. Dadurch kann erreicht werden, daß Modellkomponenten zwischen Modellen ausgetauscht werden können.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

63 RTTI = Runtime Type Information ist eine ANSI-Erweiterung zu C++, die es ermöglicht, über den `dynamic_cast`-operator einen Downcast ohne Gefahr durchzuführen. RTTI wurde jedoch zum Zeitpunkt der Implementierung von vielen Compilern noch nicht oder nur unzureichend unterstützt.

Methoden:

```
Model   (Model* owner, const String& name = "",
         bool   showInReport = true
         bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens ein Zeiger auf das übergeordnete Modell übergeben werden. Das neue Modell wird dann zum Submodell des übergeordneten. Die Verbindung zwischen Submodell und übergeordnetem Modell bleibt über die gesamte Lebensdauer bestehen. Im Gegensatz zu anderen Modellkomponenten wird ein Zeiger anstelle einer Referenz übergeben. Wird ein Null-Zeiger eingesetzt, so wird das Modell zum Hauptmodell einer neuen Hierarchie. Dies ist nur dann zulässig, wenn dem zuletzt erzeugten[64] Experiment noch kein Modell zugeordnet wurde. Modell und Experiment werden sodann miteinander verbunden.

`showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann. Die Konstruktoren sind geschützt, da nur Objekte der Unterklassen erzeugt werden sollen.

Kopier-Konstruktor und Zuweisungsoperator sind privat deklariert und nicht implementiert. Sie sollen nicht benutzt werden.

`~Model` `()`

Mit dem Löschen des Modells werden alle zum Modell gehörenden dynamischen Objekte gelöscht, die noch im Umlauf sind.

`void` **`Reset`** `()`

setzt alle zum Modell gehörenden reportfähigen Objekte (`Reportable`) einschließlich der Submodelle unmittelbar zurück.

`void` **`Report`** `()`

gibt unmittelbar Informationen über das Modell und seine Komponenten in den Report aus..

`bool` **`Connected`** `() const`

liefert true, wenn das Modell mit einem Experiment verbunden ist

`Experiment&` **`GetExperiment`** `() const`

virtuell, liefert das Experiment das (direkt oder indirekt über die Modellhierarchie) mit dem Modell verbunden ist.

[64] Dafür ist ausreichend, daß der Konstruktor der Klasse `Experiment` abgearbeitet ist

```
QueueOption GetQueueOption () const
```

liefert `OnlyOneQueue`, wenn neue Entities in maximal einer Warteschlangen warten können, andernfalls `MultipleQueue`.

```
void SetQueueOption (QueueOption newOption)
```

virtuell, setzt die Warteschlangenoption des Modells und aller seiner Submodelle auf `newOption`, kann jedoch von Unterklassen redefiniert werden, um die Einstellmöglichkeit zu sperren, z.B. wenn in bestimmten (Unter-)Modellen die `MultipleQueue`-Option benötigt wird.

```
String Description () const
```

virtuell, liefert als Standard einen leeren String. `Description` kann in Unterklassen redefiniert werden, um eine beliebige Beschreibung des Modells zu liefern, die im Report vor allen anderen Komponenten des Modells erscheint.

```
bool CheckCompatibility  (const ModelComponent* m1,
                          const ModelComponent* m2)
const
```

virtuell, liefert `true`, wenn `m1` und `m2` kompatibel sind. Als Vorgabe ist der Vergleich der zu den Komponenten gehörenden Modelle auf Identität implementiert. `CheckCompatibility` kann in Unterklassen überschrieben werden, um die vom System angebotene strenge Kompatibilitätsüberprüfung von Modellkomponenten abzuschwächen, damit es möglich wird, Komponenten zwischen Modellen auszutauschen.

```
bool InDestruction () const
```

liefert `true`, wenn das Modell gerade destruiert wird. Diese Phase wird vom Destruktor der Klasse `Model` eingeleitet.

```
Reporter* NewReporter () const
```

liefert einen mittels `new` neu erzeugten Reporter, der über das Modell berichten kann.

geschützte Methoden:

```
void DoInitialSchedules ()
```

virtuell, soll in Unterklassen überschrieben werden, um die ersten Simulationsobjekte in die Ereignisliste einzutragen. Die Methode wird vom System automatisch aufgerufen, nachdem das Experiment über `Start` angestoßen wurde. Anschließend werden alle Submodelle aufgefordert, ihre ersten Eintragungen vorzunehmen. Erst wenn `DoInitialSchedules` für alle Modelle aufgerufen wurde, beginnt der Scheduler die Ereingisliste abzuarbeiten. Sobald die Ereignisliste leer ist, beendet er seine Arbeit, und die Kontrolle geht wieder an den Kontext, der `Experiment::Start` aufgerufen hat.

6.2.26 ModelComponent

Basisklassen: `NamedObject[1]`

Assoziationen zu: `ExperimentOpts, Message, Model, SimTime, String`

in Datei: `modelcom.h`

Beschreibung: ModelComponent ist Oberklasse für die Objekte, aus denen ein Modell aufgebaut werden kann. Sie müssen einem Modell zugeordnet werden, dem Eigentümer der Modellkomponente. Diese Zuordnung muß bei der Erzeugung der Komponente festgelegt werden und kann im Nachhinein nicht geändert werden.

In der Regel sollten Modellkomponenten entweder dynamisch erzeugt werden, oder direkte oder indirekte Elemente des Modells sein. Auf jeden Fall muß der Modellprogrammierer dafür Sorge tragen, daß die Komponenten vor dem Destruktor-Aufruf des Modells zerstört werden, d.h. spätestens im Destruktor der entsprechenden von `Model` abgeleiteten Klasse.

Eine Ausnahme hiervon bilden die von `DynamicalObject` abgeleiteten Klassen, da diese vom Modell verwaltet und bei dessen Zerstörung automatisch gelöscht werden, falls noch nicht geschehen.

Eine Konsequenz der Modellzugehörigkeit ist die Modellkompatibilität. Sie erlaubt es festzustellen, ob eine Komponente im Kontext eines Modells benutzt werden darf. Dadurch soll gewährleistet werden, daß keine "fremden" Komponenten in ein Modell gelangen. Würde z.B. ein Entity E aus Modell M1 in die Warteschlange Q2 des Modells M2 eingereiht, so könnte es passieren, daß in M2 E aus Q2 entnommen und auf Attribute von E zugegriffen würde, die E gar nicht besitzt, da es ein Entity einer Klasse ist, die wohl in M1 nicht jedoch in M2 bekannt ist.[65] Daher wird der Versuch, ein Entity in eine Warteschlange eines anderen Modells einzufügen, mit Hilfe der Kompatibilitätsprüfung zurückgewiesen.

Zwei Modellkomponenten sind zueinander kompatibel, wenn sie zum selben Modell gehören, nicht zu verschiedenen Modellen der selben Klasse. Außerdem sind die Pseudo-Objekte `NullEvent`, `NullEntity` und `NullProcess` zu jeder Modellkomponente kompatibel. Diese Prüfung kann jedoch ersetzt werden, indem die Methode `CheckCompatibility` der Klasse `Model` überschrieben wird. Sie ist dann für alle Kompatibilitätsprüfungen des Modells zuständig, und sollte daher sehr sorgfältig implementiert werden.

---

65 Für Compiler, die RTTI unterstützen, könnte eine die Kompatibilitätsprüfung entfallen, da der Downcast vom Entity zum speziellen Entity mit Hilfe des `dynamic_cast`-operators ohne Gefahr vorgenommen werden kann. Es müßte dann nur gewährleistet werden, daß beide Komponenten zum selben Experiment gehören.

geerbte Methoden: `ClassName[1ö], Name[1ö], QuotedName[1ö], Rename[1ö], Valid[1ö]`

Methoden:

```
ModelComponent      (      Model&  owner,
                     const String& name        = "",
                           bool    showInTrace = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Komponente bleibt über die gesamte Lebensdauer bestehen. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann.

`ModelComponent` `(const ModelComponent&)`

Kopier-Konstruktor.

`~ModelComponent` `()`

virtueller Destruktor.

`Model&` **`GetModel`** `() const`

liefert das Modell, in das die Modellkomponente "eingebaut" ist.

`const ExperimentOpts&` **`GetExperimentOpts`** `() const`

liefert eine Referenz auf die Optionen des zugehörigen Experiments.

`bool` **`IsExperimentCompatible`** `(const ModelComponent& m) const`

liefert true, wenn die Modellkomponente zum selben Experiment gehört wie m. Die Zugehörigkeit ergibt sich aus den Modellen er Komponenten. Diese Methode wird immer aufgerufen, wenn Modellkomponenten interagieren.

`bool` **`IsModelCompatible`** `(const ModelComponent& m) const`

liefert true, delegiert die Kompatibilitätsprüfung an das Modell der Komponente, die der Methode `Model::CheckCompatibility` (siehe dort) als erster Parameter übergeben wird. `m` wird als zweiter Parameter eingesetzt. Durch Überschreiben von `CheckCompatibility` kann auf die Prüfung Einfluß genommen werden.

Die folgenden drei statischen Methoden liefern Referenzen auf die drei Pseudo-Objekte, die zurückgegeben werden, wenn eine Methode ein Objekt liefern muß, jedoch keines gefunden wurde. Diese drei Objekte bestehen die Gültigkeitsprüfung mittels `Valid` nicht und können nicht in ein Modell hineingebracht werden (z.B. über das Einfügen in eine Warteschlange). Sie existieren unabhängig von anderen Modellen und Experimenten und werden erst bei Programmende gelöscht.

`Event& **NullEvent** () const`

`Entity& **NullEntity** () const`

`Process& **NullProcess** () const`

`SimTime **NOW** () const`

static, liefert die Pseudo-Simulationszeit "Jetzt sofort" und wirkt u.U. verdrängend. Die Übergabe von `NOW` als Aktivierungszeitpunkt von `Schedule` bzw. `Activate` hat bei aktiven Prozessen eine Verdrängung zur Folge, d.h. die Vormerkung bewirkt einen Ereignislisteneintrag vor dem aktiven Prozeß. Falls gerade eine Ereignisroutine behandelt wird, bewirkt die Vormerkung mit `NOW` die Eintragung auf der Ereignisliste unmittelbar nach dem gerade behandelten Ereignis. Die Übergabe von 0.0 hingegen bewirkt eine Vormerkung hinter allen anderen Ereignislisteneinträgen zum aktuellen Zeitpunkt.

`SimTime **Epsilon** () const`

liefert die kleinste zu berücksichtigende Simulationszeit, die bestimmt, ob die Simulationsuhr weitergestellt wird. Ferner spielt sie bei statistischen Berechnungen eine Rolle.

`SimTime **CurrentTime** () const`

liefert die aktuelle Simulationszeit. Sie wird vom Experiment verwaltet und kann von außen nicht manipuliert werden. Ist sie einmal größer als 0, kann sie nicht mehr zurückgestellt werden, jedoch kann ein neues Experiment angelegt werden, daß seine eigene Simulationszeit führt.

`Schedulable& **Current** () const`

liefert das aktuelle Ereignis oder den gerade laufenden Prozeß. Wird gerade weder ein Ereignis bearbeitet noch eine Prozeßroutine ausgeführt, wird `NullEntity` zurückgegeben.

`Event& **CurrentEvent** () const`

liefert das aktuelle Ereignis. Wird gerade kein Ereignis bearbeitet, weil z.B. ein Prozeß aktiv ist, wird `NullEvent` zurückgegeben.

`Entity& **CurrentEntity** () const`

liefert das aktuelle Entity. Wird gerade kein Entity bearbeitet, weil z.B. ein externes Ereignis behandelt wird, wird `NullEntity` zurückgegeben. Ist ein Prozeß aktiv, so gilt:

```
CurrentEntity() == CurrentProcess().
```

`Process& **CurrentProcess** () const`

liefert den aktiven Prozeß. Ist kein Prozeß aktiv, weil z.B. ein Ereignis behandelt wird, wird `NullProcess` zurückgegeben. Ist ein Prozeß aktiv, so gilt außerdem:

```
CurrentEntity() == CurrentProcess().
```

`Model& **CurrentModel** () const`

liefert den aktuelle Modell. Ist ein Prozeß aktiv, so wird das Modell dieses Prozesses geliefert, andernfalls das Modell des gerade behandelten Ereignisses.

`bool **ShowInTrace** () const`

liefert true, wenn die Modellkomponente Trace-Ausgaben produzieren soll.

`void **ShowInTrace** (bool show)`

stellt ein, ob die Modellkomponente im Trace angezeigt werden soll, d.h. Trace-Ausgaben produzieren soll. Über diese Eigenschaft kann erreicht werden, daß eingebettete Komponenten, die für die Implementierung einer komplexeren Komponente verwendet wird, daß die Trace-Ausgabe für eine bestimmte Aktion von der komplexen Komponente generiert wird, und die Low-Level-Aktionen der eingebetteten Komponente nicht zusätzlich in den Trace schreiben.
Wird die Trace-Ausgabe für ein Modell ausgeschaltet, so gilt dies ebenso für alle Komponenten des Modells.

`void **SkipTraceNote** (unsigned skip = 1) const`

bewirkt, daß die nächsten `skip` Trace-Notizen unterdrückt werden. Falls `ShowInTrace` `false` liefert, wird der Aufruf von `SkipTraceNote` ignoriert.

`bool **Valid** () const`

liefert true, wenn `NamedObject::Valid` true ergibt und es sich bei der Komponente nicht um eins der Pseudo-Objekte (`NullEvent`, `NullEntity`, `NullProcess`) handelt.

`String **Debug** () const`

virtuell, kann in Unterklassen überschrieben werden, um Informationen zu liefern, die bei eingeschaltetem Debug-Modus in die Debug-Ausgabe geschrieben werden sollen.

Die folgenden drei Methoden sind statisch und liefern jeweils eine Referenz auf das entsprechende Stream-Objekt, das DESMO-C benutzt, um Standard-Ein- bzw. Ausgaben zu tätigen. Wenn die Modellkomponente also diese Stream-Objekte anstatt `cout`, `cin` oder `cerr` aus der Standardbibliothek von C++ benutzt, so können bei Bedarf alle Ein- und Ausgaben leicht umgelenkt werden.
`cout << "any text";` wird zu `Out() << "any text";`

```
ostream& Out ()
```

```
ostream& Err ()
```

```
istream& In ()
```

geschützte Methoden:

```
bool TraceIsOn () const
```

liefert true, wenn die Modellkomponente Trace-Notizen generieren darf und die Trace-Ausgabe eingeschaltet ist.

```
void TraceOn () const
```

schaltet die Trace-Ausgabe ein.

```
void TraceOff () const
```

schaltet die Trace-Ausgabe aus.

```
bool DebugIsOn () const
```

liefert true, wenn der Debug-Modus eingeschaltet ist.

```
void DebugOn () const
```

schaltet den Debug-Modus ein.

```
void DebugOff () const
```

schaltet den Debug-Modus aus.

in den folgenden drei Methoden bedeuten

what:	der Text der Nachricht
where:	der Ort der Entstehung (in der Regel eine Methode einer Klasse)
consequences:	wie wird auf die Nachricht reagiert (z.B. ändern eines ungültigen Parameters)
hint:	Hinweis, wie die Meldung zu vermeiden ist

```
void Warning (const String& what,
              const String& where,
              const String& consequences = "",
              const String& hint         = "") const
```

sendet eine Warnmeldung an das Nachrichtensystem des Experiments. Warnungen weisen auf behebbare Fehler hin. Diese Meldung wird an die Fehlerkanäle weitergeleitet und das Programm normal fortgesetzt.

```
void Error   (const String& what,
              const String& where,
              const String& consequences = "",
              const String& hint         = "") const
```

sendet eine Fehlermeldung an das Nachrichtensystem des Experiments. Fehlermeldungen weisen auf nicht behebbare Fehler des Experiments hin. Die Meldung wird an die Fehlerkanäle weitergeleitet und das Experiment angehalten. Das Programm wird ansonsten normal fortgesetzt.

```
void FatalError    (const String& what,
                    const String& where,
                    const String& consequences = "",
                    const String& hint         = "")
const
```

sendet eine Meldung über einen fatalen Fehler an das Nachrichtensystem des Experiments. Fatale Fehler weisen auf nicht behebbare Fehler des Programms hin. Die Meldung wird an die Fehlerkanäle weitergeleitet und das Experiment angehalten. Das Programm wird ansonsten abgebrochen.

```
void TraceNote (const String& what) const
```

schreibt eine Notiz in die Trace-Ausgabe, sofern die Modellkomponente Trace-Ausgaben produzieren darf.

```
void SendMessage (const Message& msg) const
```

sendet eine beliebige Nachricht an das Nachrichtensystem des Experiments. Achtung: falls `msg` eine Trace-Notiz ist, wird diese auf jeden Fall gesendet, auch wenn die Modellkomponente keine Trace-Ausgaben produzieren darf! In diesem Falle ist vorher mittels `TraceIsOn` abzufragen, ob Trace-Notizen sinnvoll sind.

Die folgenden beiden Methoden dienen der Gültigkeitsprüfung. Beide geben eine entsprechende Meldung aus. Der Parameter `where` gibt jeweils an, wo die Ungültigkeit entdeckt wurde. Er wird als C-String übergeben, um aus Effizienzgründen die u.U. überflüssige Konstruktion und Destruktion eines Strings zu vermeiden. Der Klassenname muß ebenfalls übergeben werden, da er bei einer ungültigen Modellkomponente nicht über die virtuelle Methode `ClassName` ermittelt werden kann, denn die Auswirkungen des Aufrufs hätten unvorhersehbare Konsequenzen.

```
bool valid    (const ModelComponent& m,
               const char*           className,
               const char*           where) const
```

überprüft die Modellkomponente `m` auf Gültigkeit. `valid` liefert false, wenn `m` ungültig ist und gibt eine Warnung aus.

```
bool valid    (const char* className, const char* where,
               Message::CodeType treatAs =
                                  Message::normalError)
const
```

kann verwendet werden, um die Modellkomponente selbst innerhalb einer ihrer Methoden auf Gültigkeit zu überprüfen. valid liefert false, wenn die Modellkomponente ungültig ist und gibt eine Fehlermeldung aus. In diesem Falle kann auf dem normalen Wege kein zugehöriges Experiment ermittelt werden. Aus diesem Grund wird eine globale Fehlermeldung ausgegeben. In `treatAs` kann angegeben werden, ob das Fehlschlagen der Gültigkeitsprüfung evtl. nur einer Warnung bedarf (`Message::warning`) oder das Programm sogar abzubrechen ist (`Message::fatalError`). Wird nichts angegeben, so wird nach Ausgabe der Meldung das gerade laufende Experiment angehalten.

6.2.27 NamedObject

Basisklassen: keine

Assoziationen zu: `String`

in Datei: `nobject.h`

Beschreibung: `NamedObject` dient als Basisklasse für alle Objekte, die einen Namen tragen können. Dieser wird bei Erzeugung des Objekts mit angegeben oder später mittels `Rename` gesetzt. `Rename` ist als virtuelle Methode deklariert, damit Unterklassen die Möglichkeit haben, auf Namensänderungen zu reagieren.

Desweiteren steht ein Mechanismus zur Validitätsprüfung der Objekte über die Methode `Valid` zur Verfügung. Hiermit kann geprüft werden, ob ein Objekt bereits gelöscht wurde. D.h. nach Aufruf des Konstruktors liefert `Valid true`, nach Aufruf des Destruktors `false`.

geerbte Methoden: keine

Methoden:

`NamedObject` `(const String& name = "")`

Konstruktor, erzeugt ein neues Objekt mit dem initialen Namen `name`.

`NamedObject` `(const NamedObject&)`

Kopier-Konstruktor, erzeugt ein neues Objekt und kopiert den Namen des angegebenen Objekts.

`~NamedObject` `()`

virtueller Destruktor, sorgt dafür, daß `selfRef` zurückgesetzt wird.

`const String&` **`Name`** `() const`

`String` **`Name`** `()`

Zugriffsmethode auf den Namen des Objekts.

`String` **`QuotedName`** `() const`

liefert den Namen des Objekts in Hochkommata.

`String` **`ClassName`** `() const`

virtuell, liefert den Namen der Klasse und wird in einigen Fällen benutzt, um z.B. bei Fehlermeldungen nähere Informationen zu Objekten zu bieten. Kann in Unterklassen angepaßt werden, wie dies bei den meisten DESMO-C-Klassen der Fall ist und deswegen dort nicht noch einmal beschrieben wird.

`bool` **`Valid`** `() const`

liefert `true`, wenn das Objekt noch nicht zerstört wurde

`String` **`Rename`** `(const String& newName)`

virtuell, benennt das Objekt mit `newName`. Die Methode ist virtuell, damit Unterklassen auf ein Umbenennen reagieren (z.B. verweigern) können, indem sie `Rename` überschreiben. Soll ein Umbenennen durchgeführt werden, ist der Methodenaufruf an die Oberklasse weiterzuleiten.

`NamedObject&` **`operator=`** `(const NamedObject& rhs)`

wenn ein benanntes Objekt einem anderen zugewiesen wird, bedeutet dies, daß der Name de zugewiesenen Objekts übernommen wird.

6.2.28 Observable

Basisklassen: keine

Assoziationen zu: `Observer`

in Datei: `observab.h`

Beschreibung: `Observable` setzt zusammen mit `Observer` das Beobachtermuster um. Ein Beobachter (`Observer`) registriert sich bei maximal einem beobachtbaren Objekt (`Observable`), über dessen Änderungen er informiert werden möchte. Bei einer Änderung können Unterklassen über die Methode `NotifyObservers` veranlassen, daß alle angemeldeten Beobachter deren Methode `NoteChange` aufgerufen wird. Der Destruktor löst die Verbindungen zu allen angemeldeten Beobachtern. `Observable` eignet sich gut als Mixin-Klasse, um Objekte einer anderen Klasse beobachtbar zu machen.

geerbte Methoden: keine

Methoden:

`Observable` `()`

Der Konstruktor legt eine interne Liste an, in der die Angemeldeten Beobachter verwaltet werden. Kopier-Konstruktor und Zuweisungsoperator sind als private deklariert und nicht implementiert.

`~Observable` `()`

Der Destruktor sorgt dafür, daß alle angemeldeten Beobachter abgemeldet werden.

`void` **`NotifyObservers`** `()`

bewirkt, daß alle angemeldeten Beobachter benachrichtigt werden. Dabei bestimmt der Zeitpunkt des Aufrufs in der Unterklasse, ob die Benachrichtigung vor oder nach der Änderung erfolgt, und sollte daher genau dokumentiert werden.

6.2.29 Observer

Basisklassen: keine

Assoziationen zu: `Observable`

in Datei: `observer.h`

Beschreibung: `Observer` setzt zusammen mit `Observable` das Beobachtermuster um. Ein Beobachter (`Observer`) registriert sich bei maximal einem beobachtbaren Objekt (`Observable`), über dessen Änderungen er informiert werden möchte. Das beobachtete Objekt ruft für alle angemeldeten Observer bei einer Änderung deren Methode `NoteChange` auf und übergibt einen Zeiger auf sich selbst. Um sich abzumelden, kann der Beobachter die Methode `Observe` mit einem Null-Zeiger aufrufen. Er kann sich dann erneut anmelden, auch bei einem anderen Objekt. Der Destruktor nimmt diese Abmeldung automatisch vor. Ebenso geschieht dies, wenn das beobachtete Objekt gelöscht wird.

geerbte Methoden: keine

Methoden:

`Observer` `(Observable* obs = 0)`

Der Konstruktor führt eine Anmeldung bei `obs` durch, sofern es sich nicht um einen Null-Zeiger handelt.

`Observer` `(const Observer& obj)`

Der Kopier-Konstruktor führt eine Anmeldung bei dem von `obj` beobachteten Objekt durch, sofern dieses existiert.

`~Observer` `()`

Der Destruktor führt eine Abmeldung bei dem beobachteten Objekt durch, sofern dieses existiert.

`void` **`NoteChange`** `(Observable* obs)`

rein virtuell, muß in Unterklassen implementiert werden, um zu beschreiben, wie auf die Änderung des beobachteten Objekts reagiert werden soll.

`void` **`Observe`** `(Observable* obs = 0)`

meldet den Beobachter bei `obs` an. War er schon bei einem anderen Objekt angemeldet, so wird er dort zuvor abgemeldet. Ist `obs` = 0, wird ebenfalls eine Abmeldung vorgenommen. Außerdem meldet sich so ein beobachtetes Objekt bei allen angemeldeten Beobachtern ab, wenn es gelöscht wird.

```
bool Observing () const
```

liefert `true`, wenn der Beobachter bei einem beobachtbaren Objekt angemeldet ist, andernfalls `false`.

6.2.30 Output

Basisklassen: `MessageReceiver[1]`

Assoziationen zu: `ostream, String`

in Datei: `stdoutp.h`

Beschreibung: `Output` ist die Oberklasse für alle DESMO-C-Ausgaben. Sie erbt die Schnittstelle von `MessageReceiver` die aus den virtuellen Methoden `Note` und `TakeReporter` besteht, wobei letztere nur für Reports gebraucht wird. `Output` bietet zwei Konstruktoren an, die ein Stream-Objekt an die Ausgabe binden. Der erste Konstruktor nimmt ein allgemeines Stream-Objekt entgegen, während der zweite einen Dateinamen und -erweiterung erwartet. Beiden Konstruktoren kann ein Wert für die Ausgabebreite übergeben werden.

`Output`-Objekte können an die Ausgabekanäle eines Experiments gehängt werden (z.B. `AddTraceOuptut`). Damit ein Output-Objekt funktioniert muß die Methode `Note` überschrieben werden, um auf die verschiedenen Nachrichten zu reagieren. Auf diese Weise können eigene Ausgaben implementiert werden, die die Informationen anders aufbereiten, als die Stdandardausgaben.

Eine weitere Möglichkeit besteht darin, eine der Unterklassen (`StdDebug`, `StdError`, `StdReport` und `StdTrace`) mit einem eigenen Stream-Objekt zu verbinden, und so die Standardformatierung in einen eigenen Stream zu leiten.

geerbte Methoden: Note[1ö], TakeReporter[1ö]

Methoden:

```
Output (ostream& os, unsigned width = 100)
```

Konstruktor der die Ausgabe mit dem Stream `os` verbindet und die Ausgabebreite `width` benutzt.

```
Output (const String&    fileName,
        const String&    extension,    // inkl. '.'
              unsigned   width = 100)
```

Konstruktor der die Ausgabe mit einer Datei verbindet, die den Namen `fileName` und die Dateierweiterung `extension` trägt. Dabei muß `extension` mit einem Punkt beginnen. Es wird die Ausgabebreite `width` benutzt.

```
void Rename (const String& name)
```

Rename hat nur Auswirkungen auf eine dateibasierte Ausgabe. Und zwar wird die alte Datei geschlossen und eine neue mit Namen name angelegt. Die Dateierweiterung wird beibehalten. Existiert bereits eine Datei mit demselben Namen, so wird diese zuvor gelöscht. War die alte Datei bisher leer, so wird sie ebenfalls gelöscht.

`bool` **`Empty`**`() const`

Mit `Empty` kann geprüft werden, ob bereits Ausgaben erfolgt sind.

`unsigned` **`GetWidth`**`() const`

liefert die dem Konstruktor übergebene Ausgabebreite.

geschützte Methoden:

`ostream&` **`GetOstream`**`()`

liefert eine Referenz auf das zugehörige Stream-Objekt.

6.2.31 Process

Basisklassen: `NamedObject[1], ModelComponent[2], DynamicalObject[3], Schedulable[4], Entity[5]`

Assoziationen zu: `InterruptCode, Model, ProcessCooperation, Res, Schedulable, SimTime, String`

in Datei: `process.h`

Beschreibung: `Process` ist Oberklasse für alle aktiven dynamischen Simulationsobjekte, die eigenes Verhalten aufweisen. Nachdem ein Prozeß das erste mal aktiviert wurde, folgt er der Beschreibung seiner Handlungen. Dabei kann er sich selbst wieder passivieren, entweder für eine bestimmte Zeit, nach der er automatisch wieder aktiviert wird, oder für unbestimmte Zeit. In diesem Fall muß er von außen angestoßen werden, um weiter arbeiten zu können.

Es gibt zwei Kategorien, mit denen ein Prozeß vorgemerkt werden kann. Mit den von `Entity` geerbten `Schedule`-Methoden kann er wie alle Entities zusammen mit einem Ereignis vorgemerkt werden. Er verhält sich passiv, solange er während der Bearbeitung des Ereignisses manipuliert wird. Die hier neu eingeführten `Activate`-Methoden bewirken, daß der Prozeß seine Handlungen fortführt bzw. beginnt.

Prozesse können in Warteschlangen warten sowie sich untereinander synchronisieren (siehe `Bin`, `CondQueue`, `Res`, `WaitQueue`). Der Interrupt-Mechanismus ermöglicht es, einen Prozeß zu aktivieren, und ihm eine Unterbrechungsursache zu übermitteln. (Die Metapher "Interrupt" rührt daher, daß die konzeptionell aktiven Phasen über mehrere Zeitpunkte verteilt wird, zwischen denen er jeweils technisch passiv, d.h. nicht der aktuelle Prozeß, ist. Um einen Prozeß in seiner konzeptionell aktiven Phase zu unterbrechen, muß er technisch aktiviert werden.

geerbte Methoden: `Cancel[4ö], CheckDeleteOnTermination[3ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], DeleteOnTermination[3ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetPriority[5ö], GetQueueOption[5ö], In[2ö], IsCurrent[4ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IsNull[4ö], IsNullEntity[5ö], IsProcess[5ö], IsScheduled[4ö], Name[1ö], Next[4ö], NextEntity[4ö], NextEvent[4ö], NextProcess[4ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], operator==[5ö], operator!=[5ö], operator<[5ö],`

```
operator<=[5ö], operator>[5ö], operator>=[5ö],
Out[2ö], QuotedName[1ö], Rename[1ö],
ReSchedule[4ö], Schedule[5ö],
ScheduleAfter[5ö], ScheduleBefore[5ö],
ScheduledAt[4ö], SendMessage[2g],
SetPriority[5ö], SetQueueOption[5ö],
ShowInTrace[2ö], SkipTraceNote[2ö],
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]
```

Methoden:

```
Process(Model& owner, const String& name = "",
              bool   showInTrace = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Prozeß bleibt über die gesamte Lebensdauer bestehen. Es wird ein neuer Prozeß erzeugt, der aktiviert oder auch als Entity behandelt werden kann. Mit seiner ersten Aktivierung wird die Methode `LifeCycle` aufgerufen, die in der Unterklasse zu implementieren ist, und die Handlungen des Prozesses beschreibt. `showInTrace` gibt an, ob für diesen Prozeß evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann. Prozesse besitzen Identität, weswegen Kopier-Konstruktor und Zuweisungsoperator privat deklariert und nicht implementiert sind.

```
bool IsNullProcess () const
```

liefert `true`, wenn der Prozeß der Pseudo-Prozeß `NullProcess` ist, sonst `false`.

```
bool Terminated () const
```

liefert `true`, wenn der Prozeß terminiert ist, sonst `false`. Ein Prozeß ist terminiert, wenn die Methode `LifeCycle` auf "normalem Wege" verlassen wird (über `return` oder durch Erreichen des Endes der Methode). Ein terminierter Prozeß kann nicht mehr aktiviert, wohl aber zusammen mit einem Ereignis vorgemerkt werden.

```
bool Blocked () const
```

liefert `true`, wenn der Prozeß blockiert ist, d.h. auf das Eintreten einer bestimmten Bedingung wartet (z.B. bei Synchronisationsmechanismen). Andernfalls wird `false` zurückgegeben. Ein blockierter Prozeß kann nicht unterbrochen werden. Er kann zwar aktiviert werden, aber dies hat keinen Einfluß auf die Umstände, die seine Blockade bedingen.

```
void Activate (SimTime dt)
```

der Prozeß wird zum Zeitpunkt `now + dt` aktiviert. Der Prozeß darf nicht auf der Ereignisliste stehen. In diesem Fall ist entweder `ReActivate` oder vorher `Cancel` zu verwenden. Wird für `dt NOW` übergeben und ist gerade ein Prozeß aktiv (`Current`), so wird dieser verdrängt. Ereignisse können jedoch nicht verdrängt werden.

`void` **`ReActivate`** `(SimTime dt)`

hat den gleichen Effekt wie `ReSchedule`, sollte bei Prozessen jedoch vorgezogen werden. Der Prozeß muß vorgemerkt und darf nicht terminiert sein, dann wird er um `dt` auf `now + dt` auf der Ereignisliste verschoben.

`void` **`ActivateBefore`** `(Schedulable& before)`

aktiviert den Prozeß unmittelbar vor `before`. `before` muß vorgemerkt oder der laufende (`Current`) Prozeß sein, der zu aktivierende Prozeß darf nicht auf der Ereignisliste stehen, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert. Ist `before` der laufende Prozeß, so wird dieser verdrängt.

`void` **`ActivateAfter`** `(Schedulable& after)`

aktiviert den Prozeß unmittelbar nach `after`. `after` muß vorgemerkt oder das aktuelle (`Current`) `Schedulable` sein, der zu aktivierende Prozeß darf nicht auf der Ereignisliste stehen, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

`void` **`Interrupt`** `(const InterruptCode& reason)`

unterbricht den Prozeß, und setzt `reason` in sein `InterruptCode`-Attribut ein. Dieses Attribut bleibt solange gesetzt, bis `ClearInterrupt` aufgerufen wird. Der unterbrochene Prozeß kann mit `GetInterruptCode` die Ursache für die Unterbrechung abfragen. Terminierte oder blockierte Prozesse können nicht unterbrochen werden. Ist der unterbrochene Prozeß Slave in einer Kooperationsbeziehung, so wird die Unterbrechung an den entsprechenden Master weitergeleitet.

`InterruptCode` **`GetInterruptCode`**`() const`

liefert die Unterbrechungsursache die durch den letzten Aufruf von `Interrupt` gesetzt wurde. Wurde die Unterbrechung behandelt, muß `ClearInterrupt` aufgerufen werden, um das `InterruptCode`-Attribut des Prozesses wieder zurückzusetzen.

`bool` **`Interrupted`** `() const`

liefert `true`, falls eine Unterbrechungsursache vorliegt, sonst `false`.

`void` **`ClearInterruptCode`** `()`

setzt das `InterruptCode`-Attribut des Prozesses wieder zurück auf `NoInterrupt`.

`Process&` **`Master`** `() const`

liefert den Master eines Prozesses, falls er sich als Slave in Kooperation mit einem anderen Prozeß befindet, sonst wird das Pseudo-Objekt `NullProcess` zurückgegeben.

```
bool CanCooperate () const
```

liefert `true`, wenn der Prozeß für die Kooperation als Slave bereit ist. Dies setzt voraus, daß er in einer `WaitQueue` wartet.

```
void Cooperate (ProcessCooperation& coop)
```

der Prozeß wird als Slave für die direkte Prozeßkooperation ausgewählt. Dies setzt voraus, daß er in einer `WaitQueue` wartet, dort also zuvor `Wait` aufgerufen hat.

```
bool ReleasedAll (Res*& r, unsigned long& n)
```

liefert true, wenn alle Ressourcen zurückgegeben wurden. Dann bleibt `r` unverändert und `n` wird auf 0 gesetzt. Andernfalls false geliefert und in `r` eine Ressource zurückgegeben, von der der Prozeß noch `n` Einheiten belegt.

geschützte Methoden:

```
void LifeCycle () = 0
```

rein virtuell, muß in Unterklassen definiert werden, um die Handlungen des Prozesses zu beschreiben. `LifeCycle` wird aufgerufen, wenn der Prozeß aktiviert wurde, und der Simulationzeitpunkt seiner Aktivierung erreicht wurde, so daß der Prozeß am Anfang der Ereignisliste steht. Wird `LifeCycle` über `return` oder das Erreichen des Methodenendes verlassen, so gilt der Prozeß als terminiert. Er kann dann nicht mehr aktiviert werden. Muß auf die Attribute des Prozesses nach seiner Terminierung nicht mehr zugegriffen werden, so sollte `DeleteOnTermination` aufgerufen werden, damit der Prozeß automatisch gelöscht werden kann. Andernfalls muß der Prozeß explizit gelöscht werden, damit er nicht bis zur Zerstörung seines Modells unnötig Speicher belegt.

```
void Hold (SimTime dt)
```

Die Handlungen des (aktiven) Prozesses werden für `dt` Zeiteinheiten ausgesetzt, indem seine Reaktivierung für `now + dt` vorgemerkt und er anschließend passiviert wird.

```
void Passivate ()
```

bewirkt eine Passivierung des (aktiven) Prozesses für unbestimmte Zeit. Der Prozeß kann nur noch durch andere Objekte wieder aktiviert werden.

6.2.32 ProcessCooperation

Basisklassen: `NamedObject[1], ModelComponent[2], DynamicalObject[3]`

Assoziationen zu: `Event, InterruptCode, Model, Process, Schedulable, SimTime, String`

in Datei: `coop.h`

Beschreibung: `ProcessCooperation` dient als Oberklasse für Objekte, die eine Kooperation zwischen zwei Prozessen repräsentieren. Sobald eine Kooperation zustande kommt, wird die Methode `Cooperation` aufgerufen, die in abgeleiteten Klassen definiert werden muß, um die gemeinsamen Handlungen der beiden Prozesse zu beschreiben. Dabei übernimmt einer der beiden Prozesse die Rolle des Masters, der die gemeinsamen Handlungen ausführt. Der andere Prozeß wird zum Slave. Während der Kooperation gilt der Master als aktiv, der Slave als passiv. Nach Beendigung der Kooperation wird zuerst der Master und anschließend der Slave angestoßen, falls dieser nicht während der Kooperation aktiviert wurde.

Alle Methoden sind geschützt, da sie nur zur Vereinfachung der Handlungsbeschreibung dienen. Unterklassen sollten auf keinen Fall öffentliche Methoden implementieren, die Methoden von `ProcessCooperation` aufrufen. Denn alle implizit an den Master weitergeleiteten Methodenaufrufe, werden an den aktiven Prozeß weitergeleitet, da ein Verweis auf den Master nur innerhalb der Methode Cooperation verfügbar ist. Der Aufruf einer öffentlichen Methode während einer inaktiven Phase des Masters könnte dazu führen, daß z.B. `Hold` nicht an den Master sondern an den gerade aktiven Prozeß weitergeleitet würde.

Eine Besonderheit ist beim Interrupt-Mechanismus zu beachten:

Wird ein Slave während einer Kooperation unterbrochen, so wird der Interrupt-Befehl automatisch an den Master weitergeleitet, so daß es letztendlich zur Unterbrechung der Kooperation kommt. Wird der Interrupt dort nicht behandelt, d.h. `ClearInterruptCode` wird nicht aufgerufen, so wird der Interrupt-Code nach Ende der Kooperation auf den Slave übertragen.

geerbte Methoden: `CheckDeleteOnTermination[3ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], DeleteOnTermination[3ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], Name[1ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö],`

```
Rename[1ö], SendMessage[2g], ShowInTrace[2ö],
SkipTraceNote[2ö], TraceIsOn[2g],
TraceNote[2g], TraceOff[2g], TraceOn[2g],
Valid[2ö], valid[2g], Warning[2g]
```

Methoden:

ProcessCooperation (Model& owner,
const String& name = "")

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Kooperation bleibt über die gesamte Lebensdauer bestehen.

geschützte Methoden:

void **Cooperation** (Process& master, Process& slave) = 0

rein virtuell, muß in Unterklassen überschrieben werden, um die gemeinsamen Handlungen von Master und Slave zu beschreiben. Referenzen auf Master und Slave werden als Parameter übergeben, wobei die Reihenfolge zu beachten ist. Der Master ist beim Aufruf von `Cooperation` aktiv.

Die übrigen Methoden dienen der Vereinfachung bei der Implementierung von `Cooperation`. Sie beziehen sich alle auf den Master (bzw. den aktiven Prozeß) und werden an diesen durchgereicht (siehe `Process`).

```
void           Activate (SimTime dt)

void           ReActivate (SimTime dt)

void           ActivateBefore (Schedulable& before)

void           ActivateAfter (Schedulable& after)

void           Hold (SimTime dt)

void           Passivate ()

bool           Interrupted () const

InterruptCode  GetInterruptCode () const

void           ClearInterruptCode ()

void           Schedule (SimTime dt, Event& ev)

void           ScheduleBefore (Schedulable& before,
                               Event&       ev)

void           ScheduleAfter  (Schedulable& after,
                               Event& ev)
```

```
PriorityT       GetPriority () const

PriorityT       SetPriority (const PriorityT newPriority)

QueueOption     GetQueueOption() const

Event&          NextEvent () const

Entity&         NextEntity () const

Process&        NextProcess () const
```

6.2.33 ProcessQueue

Basisklassen: `NamedObject[1]`, `ModelComponent[2]`, `Reportable[3]`, `QueueBased[4]`

Assoziationen zu: `Condition`, `Process`, `Model`, `Reporter`, `SimTime`, `String`

in Datei: `pqueue.h`

Beschreibung: `ProcessQueue` ist die Klasse der Warteschlangen, die beliebig viele Prozesse aufnehmen können. Sie entspricht der Klasse `Queue`, jedoch lassen sich ausschließlich Prozesse einfügen. Dadurch läßt sich erreichen, daß der Warteschlange entnommene Objekte garantiert Prozesse sind, was bei `Queue` nicht der Fall ist.

`ProcessQueue` erbt von `QueueBased` alle Merkmale für die Warteschlangenstatistik und ergänzt sie um Methoden zum Einfügen, Entfernen und Auffinden von Prozessen. Einfügungen erfolgen mittels `Insert` etc., Löschungen mittels `Remove`. Zum Auffinden von in einer Warteschlange wartenden Prozessen können die Methoden `First`, `Last`, `Pred` und `Succ` verwendet werden. `Pred` liefert den Vorgänger und `Succ` den Nachfolger eines wartenden Prozesses. Alle vier Methoden sind um eine Suchbedingung (`Condition`) ergänzbar, so daß jeweils der erste Prozeß ermittelt wird, der die Bedingung erfüllt. Das Einfügen erfolgt nach Priorität und innerhalb gleich priorisierter Prozesse nach dem FIFO-Prinzip, d.h. Prozesse höherer Priorität stehen vor allen anderen mit niedrigerer Priorität, Prozesse gleicher Priorität in der (zeitlichen) Reihenfolge ihres Eintrags.

Bei unzulässigen Operationen und fehlerhaften Zugriffen auf Warteschlangen oder Prozesse werden Warnungen oder Fehlermeldungen ausgegeben. In solchen Fällen wird von Methoden, die einen Prozeß liefern das Pseudo-Objekt `NullProcess` zurückgegeben.

Ein Prozeß kann in der Regel nur in einer Warteschlange zur Zeit warten. Beim Versuch einer weiteren Einfügung wird er zuvor aus der alten Warteschlange entfernt. Es besteht jedoch die Möglichkeit, über die Entity-Methode `SetQueueOption` für einen Prozeß festzulegen, ob er in mehreren Warteschlangen warten kann. Diese Einstellung läßt sich auch für ein ganzes Modell vornehmen (`Model::SetQueueOption`), was jedoch keine Auswirkung auf bereits existierende Prozesse hat. Vielmehr ist es eine Vorgabe für neue Prozesse. Diese übernehmen bei ihrer Erzeugung die `QueueOption` ihres Modells.

geerbte Methoden: `AvgLength[4ö]`, `AvgWaitTime[4ö]`, `ClassName[1ö]`, `CurrentTime[2ö]`, `CurrentEntity[2ö]`, `CurrentEvent[2ö]`, `CurrentProcess[2ö]`, `CurrentModel[2ö]`, `Debug[2ö]`, `DebugIsOn[2g]`, `DebugOn[2g]`, `DebugOff[2g]`, `Empty[4ö]`,

```
Epsilon[2ö], Err[2ö], Error[2g],
FatalError[2g], GetModel[2ö], In[2ö],
IsExperimentCompatible[2ö],
IsModelCompatible[2ö], IncObservations[3g],
Length[4ö], MaxLength[4ö], MaxLengthAt[4ö],
MaxWaitTime[4ö], MaxWaitTimeAt[4ö],
MinLength[4ö], MinLengthAt[4ö], Name[1ö],
NewReporter[4ö], NOW[2ö], NullEntity[2ö],
NullEvent[2ö], NullProcess[2ö],
Observations[3ö], Out[2ö], QuotedName[1ö],
Rename[1ö], Reset[4ö], ResetAt[3ö],
SendMessage[2g], ShowInReport[3ö],
ShowInTrace[2ö], SkipTraceNote[2ö],
StdDevLength[4ö], StdDevWaitTime[4ö],
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g],
Warning[2g], ZeroWaits[4ö]
```

Methoden:

```
ProcessQueue (Model& owner, const String& name = "",
              bool   showInReport = true
              bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
ProcessQueue (const ProcessQueue& obj)
```

Der Kopier-Konstruktor erzeugt aus `obj` eine neue Warteschlange mit den selben statistischen Daten wie `obj`, jedoch enthält sie keine Prozesse, d.h. sie ist leer und somit gilt: `MinLength = 0` und `MinLengthAt = CurrentTime`.

```
~Queue ()
```

Der Destruktor entfernt alle noch wartenden Prozesse aus der Warteschlange. Die Prozesse selbst werden nicht gelöscht.

```
void Insert (Process& p)
```

fügt den Prozeß `p` in die Warteschlange nach Priorität bzw. FIFO ein. Ist für den Prozeß die Queue-Option `OnlyOneQueue` eingestellt, so wird er ggf. vorher aus anderen Warteschlangen entfernt.

`void` **`InsertAfter`** `(Process& p, Process& where)`

fügt den Prozeß `p` direkt hinter dem Prozeß `where` ohne Berücksichtigung der Priorität in die Warteschlange ein. Ist für den einzufügenden Prozeß die Queue-Option `OnlyOneQueue` eingestellt, so wird er ggf. vorher aus anderen Warteschlangen entfernt. `where` muß sich in der Warteschlange befinden, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

`void` **`InsertBefore`** `(Process& p, Process& where)`

fügt den Prozeß `p` direkt vor dem Prozeß `where` ohne Berücksichtigung der Priorität in die Warteschlange ein. Ist für den einzufügenden Prozeß die Queue-Option `OnlyOneQueue` eingestellt, so wird er ggf. vorher aus anderen Warteschlangen entfernt. `where` muß sich in der Warteschlange befinden, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

`void` **`Remove`** `(Process& p)`

entfernt den Prozeß `p` aus der Warteschlange. Befindet sich `p` nicht in der Warteschlange, so wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Process&` **`First`** `() const`

liefert den vordersten Prozeß der Warteschlange oder `NullProcess`, falls sie leer ist.

`Process&` **`Last`** `() const`

liefert den letzten Prozeß der Warteschlange oder `NullProcess`, falls sie leer ist.

`Process&` **`Pred`** `(const Process& p) const`

liefert den Vorgänger des Prozesses `p` in der Warteschlange oder `NullProcess`, falls sie leer ist oder `p` das vorderste Element der Warteschlange ist. Befindet sich `p` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullProcess` zurückgegeben.

`Process&` **`Succ`** `(const Process& p) const`

liefert den Nachfolger des Prozesses `p` in der Warteschlange oder `NullProcess`, falls sie leer ist oder `p` das letzte Element der Warteschlange ist. Befindet sich `p` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullProcess` zurückgegeben.

Die folgenden Methoden arbeiten wie die vier vorhergehenden, jedoch liefern sie nicht den unmittelbar nächsten Prozeß, sondern suchen ggf. weiter, bis der erste Prozeß gefunden wurde, der eine bestimmte Bedingung erfüllt.

`Process&` **`First`** `(Condition& c) const`

liefert den ersten Prozeß der Warteschlange, der die Bedingung `c` erfüllt, oder `NullProcess`, falls die Warteschlange leer ist oder keiner der wartenden Prozesse die Bedingung erfüllt.

```
Process& Last (Condition& c) const
```

liefert den letzten Prozeß der Warteschlange, der die Bedingung `c` erfüllt, oder `NullProcess`, falls die Warteschlange leer ist oder keiner der wartenden Prozesse die Bedingung erfüllt.

```
Process& Pred (const Process& p, Condition& c) const
```

liefert den nächsten Vorgänger des Prozesses `p`, der die Bedingung `c` erfüllt, oder `NullProcess`, falls sie leer ist oder `p` das vorderste Element der Warteschlange ist. Befindet sich `p` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullProcess` zurückgegeben.

```
Process& Succ (const Process& p, Condition& c) const
```

liefert den nächsten Nachfolger des Prozesses `p`, der die Bedingung `c` erfüllt, oder `NullProcess`, falls sie leer ist oder `p` das letzte Element der Warteschlange ist. Befindet sich `p` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullProcess` zurückgegeben.

6.2.34 Queue

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], QueueBased[4]`

Assoziationen zu: `Condition, Entity, Model, Reporter, SimTime, String`

in Datei: `queue.h`

Beschreibung: `Queue` ist die Klasse der Warteschlangen, die beliebig viele Entities aufnehmen können. Sie erbt von QueueBased alle Merkmale für die Warteschlangenstatistik und ergänzt sie um Methoden zum Einfügen, Entfernen und Auffinden von Entities. Einfügungen erfolgen mittels `Insert` etc., Löschungen mittels `Remove`. Zum Auffinden von in einer Warteschlange wartenden Entities können die Methoden `First`, `Last`, `Pred` und `Succ` verwendet werden. `Pred` liefert den Vorgänger und `Succ` den Nachfolger eines wartenden Entities. Alle vier Methoden sind um eine Suchbedingung (`Condition`) ergänzbar, so daß jeweils das erste Entity ermittelt wird, das die Bedingung erfüllt. Das Einfügen erfolgt nach Priorität und innerhalb gleich priorisierter Entities nach dem FIFO-Prinzip, d.h. Entities höherer Priorität stehen vor allen anderen mit niedrigerer Priorität, Entities gleicher Priorität in der (zeitlichen) Reihenfolge ihres Eintrags.

Bei unzulässigen Operationen und fehlerhaften Zugriffen auf Warteschlangen oder Entities werden Warnungen oder Fehlermeldungen ausgegeben. In solchen Fällen wird von Methoden, die ein Entity liefern das Pseudo-Objekt `NullEntity` zurückgegeben.

Ein Entity kann in der Regel nur in einer Warteschlange zur Zeit warten. Beim Versuch einer weiteren Einfügung wird es zuvor aus der alten Warteschlange entfernt. Es besteht jedoch die Möglichkeit, über die Entity-Methode `SetQueueOption` für ein Entity festzulegen, ob es in mehreren Warteschlangen warten kann. Diese Einstellung läßt sich auch für ein ganzes Modell vornehmen (`Model::SetQueueOption`), was jedoch keine Auswirkung auf bereits existierende Entities hat. Vielmehr ist es eine Vorgabe für neue Entities. Diese übernehmen bei ihrer Erzeugung die `QueueOption` ihres Modells.

In Objekten der Klasse Queue können sowohl Entities als auch Prozesse warten. D.h. bei er Entnahme werden immer 'nur' Entities geliefert. Will man sicher gehen, daß es sich um Prozesse handelt, müßte man `Entity::IsProcess` aufrufen und ggf. einen Down-Cast vornehmen, um das Entity in einen Process umzuwandeln. Für Warteschlangen, in denen ausschließlich Prozesse warten sollen, ist daher die Klasse `ProcessQueue` geeigneter.

geerbte Methoden: `AvgLength[4ö], AvgWaitTime[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö],`

```
CurrentModel[2ö], Debug[2ö], DebugIsOn[2g],
DebugOn[2g], DebugOff[2g], Empty[4ö],
Epsilon[2ö], Err[2ö], Error[2g],
FatalError[2g], GetModel[2ö], In[2ö],
IsExperimentCompatible[2ö],
IsModelCompatible[2ö], IncObservations[3g],
Length[4ö], MaxLength[4ö], MaxLengthAt[4ö],
MaxWaitTime[4ö], MaxWaitTimeAt[4ö],
MinLength[4ö], MinLengthAt[4ö], Name[1ö],
NewReporter[4ö], NOW[2ö], NullEntity[2ö],
NullEvent[2ö], NullProcess[2ö],
Observations[3ö], Out[2ö], QuotedName[1ö],
Rename[1ö], Reset[4ö], ResetAt[3ö],
SendMessage[2g], ShowInReport[3ö],
ShowInTrace[2ö], SkipTraceNote[2ö],
StdDevLength[4ö], StdDevWaitTime[4ö],
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g],
Warning[2g], ZeroWaits[4ö]
```

Methoden:

```
Queue  (Model& owner, const String& name = "",
        bool   showInReport = true
        bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`Queue` `(const Queue& obj)`

Der Kopier-Konstruktor erzeugt aus `obj` eine neue Warteschlange mit den selben statistischen Daten wie `obj`, jedoch enthält sie keine Entities, d.h. sie ist leer und somit gilt: `MinLength = 0` und `MinLengthAt = CurrentTime`.

`~Queue` `()`

Der Destruktor entfernt alle noch wartenden Entities aus der Warteschlange. Die Entities selbst werden nicht gelöscht.

`void` **`Insert`** `(Entity& e)`

fügt das Entity `e` in die Warteschlange nach Priorität bzw. FIFO ein. Ist für das Entity die Queue-Option `OnlyOneQueue` eingestellt, so wird es ggf. vorher aus anderen Warteschlangen entfernt.

`void` **`InsertAfter`** `(Entity& e, Entity& where)`

fügt das Entity `e` direkt hinter dem Entity `where` ohne Berücksichtigung der Priorität in die Warteschlange ein. Ist für das einzufügende Entity die Queue-Option `OnlyOneQueue` eingestellt, so wird es ggf. vorher aus anderen Warteschlangen entfernt. `where` muß sich in der Warteschlange befinden, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

`void` **`InsertBefore`** `(Entity& e, Entity& where)`

fügt das Entity `e` direkt vor dem Entity `where` ohne Berücksichtigung der Priorität in die Warteschlange ein. Ist für das einzufügende Entity die Queue-Option `OnlyOneQueue` eingestellt, so wird es ggf. vorher aus anderen Warteschlangen entfernt. `where` muß sich in der Warteschlange befinden, andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

`void` **`Remove`** `(Entity& e)`

entfernt das Entity e aus der Warteschlange. Befindet sich e nicht in der Warteschlange, so wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Entity&` **`First`** `() const`

liefert das vorderste Entity der Warteschlange oder `NullEntity`, falls sie leer ist.

`Entity&` **`Last`** `() const`

liefert das letzte Entity der Warteschlange oder `NullEntity`, falls sie leer ist.

`Entity&` **`Pred`** `(const Entity& e) const`

liefert den Vorgänger des Entity `e` in der Warteschlange oder `NullEntity`, falls sie leer ist oder `e` das vorderste Element der Warteschlange ist. Befindet sich `e` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullEntity` zurückgegeben.

`Entity&` **`Succ`** `(const Entity& e) const`

liefert den Nachfolger des Entity `e` in der Warteschlange oder `NullEntity`, falls sie leer ist oder `e` das letzte Element der Warteschlange ist. Befindet sich `e` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullEntity` zurückgegeben.

Die folgenden Methoden arbeiten wie die vier vorhergehenden, jedoch liefern sie nicht das unmittelbar nächste Entity, sondern suchen ggf. weiter, bis das erste Entity gefunden wurde, das eine bestimmte Bedingung erfüllt.

`Entity&` **`First`** `(Condition& c) const`

liefert das erste Entity der Warteschlange, das die Bedingung `c` erfüllt, oder `NullEntity`, falls die Warteschlange leer ist oder keines der wartenden Entities die Bedingung erfüllt.

`Entity& **Last** (Condition& c) const`

liefert das letzte Entity der Warteschlange, das die Bedingung `c` erfüllt, oder `NullEntity`, falls die Warteschlange leer ist oder keines der wartenden Entities die Bedingung erfüllt.

`Entity& **Pred** (const Entity& e, Condition& c) const`

liefert den nächsten Vorgänger des Entity `e`, der die Bedingung `c` erfüllt, oder `NullEntity`, falls sie leer ist oder `e` das vorderste Element der Warteschlange ist. Befindet sich `e` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullEntity` zurückgegeben.

`Entity& **Succ** (const Entity& e, Condition& c) const`

liefert den nächsten Nachfolger des Entity `e`, der die Bedingung `c` erfüllt, oder `NullEntity`, falls sie leer ist oder `e` das letzte Element der Warteschlange ist. Befindet sich `e` nicht in der Warteschlange, wird eine Warnung ausgegeben und `NullEntity` zurückgegeben.

6.2.35 QueueBased

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3]`

Assoziationen zu: `Model, Reporter, SimTime, String`

in Datei: `qbased.h`

Beschreibung: `QueueBased` ist Oberklasse für alle Warteschlangenbasierten Klassen. Sie bietet eine Reihe von Methoden an, um auf eine automatisch geführte kumulierende Statistik zugreifen zu können. Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0.`

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
QueueBased   (Model& owner, const String& name = "",
              bool   showInReport = true
              bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
QueueBased (const QueueBased& obj)
```

Der Kopier-Konstruktor erzeugt aus `obj` ein neues `QueueBased`-Objekt mit den selben statistischen Daten, mit Ausnahme von: `MinLength = 0` und `MinLengthAt = CurrentTime.`

```
void Reset ()
```

virtuell, setzt die Statistik der Warteschlange zurück. `MinLength` und `MaxLength` werden auf die aktuelle Warteschlangenlänge gesetzt. Die statistische Berechnung wird neu begonnen mit Anfangswert 0.

```
bool Empty () const
```

liefert `true`, falls die Warteschlange leer ist, andernfalls `false`.

```
unsigned long Length () const
```

liefert die aktuelle Länge der Warteschlange.

Die folgenden Methoden liefern die entsprechenden Werte der kumulierenden Statistik der Warteschlange seit der letzten Rücksetzung. Bei Mehrdeutigkeiten gilt der Zeitpunkt des ersten Auftretens eines Extremwertes (Suffix "`At`"). `AvgLength` und `StdDevWaitTime` sind zeitlich gewichtet.

```
unsigned long MinLength () const

unsigned long MaxLength () const

double AvgLength () const

double StdDevLength () const

SimTime MinLengthAt () const

SimTime MaxLengthAt () const


unsigned long ZeroWaits () const

SimTime MaxWaitTime () const

SimTime AvgWaitTime () const

SimTime StdDevWaitTime () const

SimTime MaxWaitTimeAt () const


Reporter* NewReporter() const
```

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über das Warteschlangenbasierte Objekt berichten kann. Der gelieferte Reporter berichtet so, als handele es sich um eine Warteschlange.

6.2.36 RealDist

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4]`

Assoziationen zu: `Model, String`

in Datei: `realdist.h`

Beschreibung: `RealDist` ist Oberklasse für alle reelwertigen Zufallszahlenströme. Sie führt die Methode `Sample` ein, die einen Wert vom Typ `double` liefert, der der gezogenen Zufallszahl entspricht. Die Methode wird erst in den Unterklassen definiert, die jeweils einen eigenen Verteilungstyp repräsentieren.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

`double` **`Sample`** `() = 0`

rein virtuell, wird in Unterklassen definiert, so daß der gezogene reelwertige Wert geliefert wird.

geschützte Methoden:

```
RealDist      (Model& owner, const String& name = "",
               bool   showInReport = true
               bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann. Der Konstruktor ist geschützt, da nur Objekte der Unterklassen erzeugt werden sollen.

6.2.37 RealDistConst

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `realdist.h`

Beschreibung: `RealDistConst` ist die Klasse von reelwertigen Zufallszahlenströmen, deren `Sample`-Aufrufe stets denselben Wert liefern. Konstante ZZ-Ströme werden benutzt, um ein Modell zunächst prototypisch zu entwickeln. Später können dann die konstanten ZZ-Ströme durch solche mit einer passenden Verteilung ersetzt werden. Ein dem Konstruktor übergebener Wert bestimmt das Ergebnis des `Sample`-Aufrufs.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
RealDistConst      (      Model& owner,
                    const String& name         = "",
                          double  value        = 0.0,
                          bool    showInReport = true,
                          bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. Der Parameter `value` gibt den Wert an, den zukünftige Aufrufe von `Sample` liefern. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`double` **`Sample`** `()`

liefert den dem Konstruktor übergebenen Wert.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetValue`** `() const`

liefert den dem Konstruktor übergebenen Wert.

`void` **`ChangeParameter`** `(double newValue)`

setzt den von `Sample` zu liefernden Wert auf `newValue`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

6.2.38 RealDistEmpirical

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `realdist.h`

Beschreibung: `RealDistEmpirical` ist die Klasse von reelwertigen Zufallszahlenströmen mit empirischer Verteilung. Dabei wird die empirische Verteilungsfunktion durch (x,y)-Wertepaare angegeben, wobei y der Wert der kumulativen Häufigkeit für x ist. Die Wertepaare werden über die Methode `AddEntry` eingebracht, wobei für alle n Paare (x,y) gilt:

(1) $x_i \leq x_{i+1}$
(2) $0 \leq y_i \leq y_{i+1} \leq y_n = 1$

Die Wertepaare sind in aufsteigender Reihenfolge durch wiederholte Aufrufe von `AddEntry` zu übergeben.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
RealDistEmpirical  (       Model&  owner,
                    const  String& name         = "",
                           bool    showInReport = true,
                           bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`double` **`Sample`** `()`

liefert eine Zufallszahl, die von der mittels `AddEntry` angegebenen Verteilung abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

```
void AddEntry       (double newValue,
                     double cumulativeFrequency)
```

fügt ein neues Wertepaar mit x = `newValue` und y = `cumulativeFrequency` hinzu, wobei die in der Klassenbeschreibung angegebenen Bedingungen eingehalten werden müssen. Andernfalls wird der Aufruf nach Ausgabe einer Fehlermeldung ignoriert. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Erst wenn ein Wertepaar mit einer kumulativen Häufigkeit von 1 übergeben wurde, und nur dann ist der ZZ-Strom für `Sample`-Aufrufe bereit.

`unsigned` **`CountEntries`** `() const`

liefert die Anzahl der bisher mittels `AddEntry` übergebenen Wertepaare.

`double` **`GetValue`** `(unsigned n) const`

liefert den x-Wert des `n`-ten Eintrags ($0 \le n <$ `CountEntries()`). Wird ein größerer Wert für n übergeben, so wird nach Ausgabe einer Warnung der größte x-Wert geliefert, oder 0, falls noch keine Wertepaare definiert sind.

`double` **`GetCumulativeFrequency`** `(unsigned n) const`

liefert den y-Wert des `n`-ten Eintrags ($0 \le n <$ `CountEntries()`). Wird ein größerer Wert für n übergeben, so wird nach Ausgabe einer Warnung der größte y-Wert geliefert, oder 0, falls noch keine Wertepaare definiert sind.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

6.2.39 RealDistErlang

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `realdist.h`

Beschreibung: `RealDistPoisson` ist die Klasse von reelwertigen Zufallszahlenströmen mit k-Erlang-Verteilung.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
RealDistErlang     (      Model&   owner,
                    const String&  name         = "",
                          unsigned k            = 1,
                          double   mean         = 0.0,
                          bool     showInReport = true,
                          bool     showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `mean` ist der Mittelwert der zu ziehenden Zufallszahlen und darf nicht negativ sein. `k` gibt die Ordnung der Erlang-Verteilung an und muß größer als 0 sein. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`double` **`Sample`** `()`

liefert eine Zufallszahl, die vom dem Konstruktor übergebenen Mittelwert und der Ordnung der Erlang-Verteilung abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetMean`** `() const`

liefert den Mittelwert der Erlang-Verteilung.

`void` **`ChangeParameter`** `(unsigned newK, double newMean)`

setzt den Mittelwert und die Ordnungszahl k der Erlang-Verteilung auf `newMean` bzw. `newK`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Wird ein negativer Wert für `newMean` übergeben, so wird er nach Ausgabe einer Warnung invertiert. Bei `newK` = 0 wird eine Warnung ausgegeben und die Ordnungszahl der Erlang-Verteilung auf 1 gesetzt.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void` **`checkK`** `(const char* where)`

wird nach dem Ändern der Ordnungszahl k dazu verwendet, nach Ausgabe einer Warnung Werte von 0 auf 1 zu korrigieren. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

`void` **`checkMean`** `(const char* where)`

wird nach dem Ändern des Mittelwertes dazu verwendet, evtl. negative Werte nach Ausgabe einer Warnung zu invertieren. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

6.2.40 RealDistExponential

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `realdist.h`

Beschreibung: `RealDistExponential` ist die Klasse von reelwertigen Zufallszahlenströmen mit negativ-exponentieller-Verteilung. Ein dem Konstruktor übergebener Wert ist der Mittelwert der Verteilung.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
RealDistExponential (      Model& owner,
                     const String& name         = "",
                           double  mean         = 0.0,
                           bool    showInReport = true,
                           bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `mean` ist der Mittelwert der zugrundeliegenden Verteilung und darf nicht negativ sein. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
double Sample ()
```

liefert eine Zufallszahl, die vom dem Konstruktor übergebenen Mittelwert abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetMean`** `() const`

liefert den Mittelwert der zugrundeliegenden Verteilung.

`void` **`ChangeParameter`** `(double newMean)`

setzt den Mittelwert der zugrundeliegenden Verteilung auf `newMean`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Wird ein negativer Wert übergeben, so wird er nach Ausgabe einer Warnung invertiert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void` **`checkMean`** `(const char* where)`

wird nach dem Ändern des Mittelwertes dazu verwendet, evtl. negative Werte nach Ausgabe einer Warnung zu invertieren. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

6.2.41 RealDistNormal

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `realdist.h`

Beschreibung: `RealDistNormal` ist die Klasse von reelwertigen Zufallszahlenströmen mit Gaußscher Normal-Verteilung.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
RealDistNormal     (      Model& owner,
                    const String& name         = "",
                          double  mean         = 0.0,
                          double  stddev       = 0.0,
                          bool    showInReport = true,
                          bool    showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. In `mean` wird der Mittelwert der zugrundeliegenden Verteilung übergeben. `stddev` ist die Standardabweichung und darf nicht negativ sein. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`double` **`Sample`** `()`

liefert eine Zufallszahl, die von der zugrundeliegenden Verteilung abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetMean`** `() const`

liefert den Mittelwert der zugrundeliegenden Verteilung.

`double` **`GetStddev`** `() const`

liefert die Standardabweichung der zugrundeliegenden Verteilung.

`void` **`ChangeParameter`** `(double newMean, double newStddev)`

setzt den Mittelwert der zugrundeliegenden Verteilung auf `newMean` und die Standardabweichung auf `newStddev`. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Wird ein negativer Wert für `stddev` übergeben, so wird er nach Ausgabe einer Warnung invertiert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void` **`checkStddev`** `(const char* where)`

wird nach dem Ändern der Standardabweichung dazu verwendet, evtl. negative Werte nach Ausgabe einer Warnung zu invertieren. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

6.2.42 RealDistUniform

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], Distribution[4], IntDist[5]`

Assoziationen zu: `Model, Reporter, String`

in Datei: `realdist.h`

Beschreibung: `RealDistUniform` ist die Klasse von reelwertigen Zufallszahlenströmen mit Gleichverteilung auf einem anzugebenden Intervall.

geerbte Methoden: `Antithetic[4ö], AntitheticAll[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], GetType[4ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], random[4ö], Rename[1ö], Reset[3ö], ResetAll[4ö], ResetAt[3ö], Seed[4ö], SeedGenerator[4ö], SendMessage[2g], SetSeed[4ö], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
RealDistUniform    (      Model& owner,
                    const String& name        = "",
                          double low          = 0.0,
                          double high         = 0.0,
                          bool   showInReport = true,
                          bool   showInTrace  = false)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und ZZ-Strom bleibt über die gesamte Lebensdauer bestehen. `low` und high bilden das Intervall, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind. Ist `high < low`, werden die Werte nach Ausgabe einer Warnung vertauscht. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
double Sample ()
```

liefert eine Zufallszahl, die vom dem Konstruktor übergebenen Intervall abhängt.

`String` **`GetType`** `() const`

liefert die Typ-Bezeichnung des ZZ-Stroms als String.

`double` **`GetLow`** `() const`

liefert die untere Intervallgrenze, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind.

`double` **`GetHigh`** `() const`

liefert die obere Intervallgrenze, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind.

`void` **`ChangeParameter`** `(double newLow, double newHigh)`

setzt das Intervall, innerhalb dessen die zu ziehenden Zufallszahlen gleichverteilt sind, auf [`newLow`, `newHigh`]. Ist `high` < `low`, werden die Werte nach Ausgabe einer Warnung vertauscht. Wurde der ZZ-Strom bereits benutzt, wird eine Warnung ausgegeben und der Aufruf ignoriert.

`Reporter*` **`NewReporter`** `() const`

liefert einen mittels `new` neu erzeugten Reporter, der über den ZZ-Strom berichten kann.

geschützte Methoden:

`void` **`checkHiLo`** `(const char* where)`

wird nach dem Ändern der Intervallgrenzen dazu verwendet, evtl. vertauschte Werte nach Ausgabe einer Warnung auszuwechseln. Diese Methode ist für den Modellprogrammierer nicht von Bedeutung.

6.2.43 Regression

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5]`

Assoziationen zu: `Model, Reporter, SimTime, String, ValueSupplier`

in Datei: `regress.h`

Beschreibung: `Regression` ist die Klasse von Datensammelobjekten zur statistischen Erfassung von Beobachtungsgrößen zwecks linearer Regression. Für jedes Größenpaar ist ein Objekt zu erzeugen, das mit zwei Wertlieferanten (`ValueSupplier`) verbunden wird. Diese berechnen und übergeben auf Verlangen den zu beobachtenden Wert (Beobachtungsgröße). Mit `Update` werden beide Beobachtungsgrößen ermittelt und in die Statistik einbezogen.

Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, immer dann die Statistik fortzuschreiben, wenn sich die Beobachtungsgrößen geändert haben. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Aktualisierung der Statistik mit Hilfe der von den `ValueSupplier`-Objekten gelieferten Werte. `Observable` kann als Mixin-Klasse verwendet werden, um Objekte beobachtbar zu machen.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0`.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Update[5ö], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
Regression   (      Model&          owner,
              const String&         name1,
              const String&         name2,
                    ValueSupplier&  xvs,
                    ValueSupplier&  yvs,
                    bool            showInReport = true,
                    bool            showInTrace  = false)
```

Dem Konstruktor müssen mindestens das zugehörige Modell, zwei Namen und zwei Value-`Supplier`-Objekte übergeben werden. Die Verbindung zwischen Modell und Regressionsanalyseobjekt bleibt über die gesamte Lebensdauer bestehen. `name1` und `name2` benennen die x- bzw. y-Beobachtungsgröße. In `xvs` und `yvs` werden die Objekte übergeben, die auf Verlangen die Beobachtungsgröße für den x- bzw. y-Wert berechnen und liefern können. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
Regression   (Model&          owner,
              ValueSupplier&  xvs,
              ValueSupplier&  yvs,
              bool            showInReport = true,
              bool            showInTrace  = false)
```

Der Konstruktor bietet eine Alternative zum ersten Konstruktor und unterscheidet sich von diesem nur dadurch, daß hier die Angebe der Namen entfallen kann. Sie werden dann einfach auf "X" bzw. "Y" gesetzt.

`void` **`Update`** `()`

virtuell, läßt sich von den ValueSupplier-Objekten die aktuellen Werte der Beobachtungsgrößen, um die Statistik zu aktualisieren.

`String` **`Name2`** `() const`

liefert den zweiten Namen der dem Konstruktor übergeben wurde. Auf den ersten kann mit `Name` zugegriffen werden.

`String` **`QuotedName2`** `() const`

liefert den zweiten Namen in Apostroph eingerahmt.

`void` **`Values`** `(double& x, double& y) const`

setzt die Parameter `x` und `y` auf die bei der letzten Aktualisierung ermittelten Werte der beiden Beobachtungsgrößen.

`double` **`xValue`** `() const`

liefert den bei der letzten Aktualisierung ermittelten Werte der ersten Beobachtungsgröße.

`double` **`yValue`** `() const`

liefert den bei der letzten Aktualisierung ermittelten Werte der zweiten Beobachtungsgröße.

`double` **`xMean`** `() const`

liefert den Mittelwert der seit dem letzten Rücksetzen eingegangenen x-Werte.

`double` **`yMean`** `() const`

liefert den Mittelwert der seit dem letzten Rücksetzen eingegangenen y-Werte.

`double` **`ResStdDev`** `() const`

liefert die Residuale Standardabweichung seit dem letzten Rücksetzen.

`double` **`RegCoeff`** `() const`

liefert den Regressionskoeffizienten seit dem letzten Rücksetzen.

`double` **`Intercept`** `() const`

liefert den Ordinatenabstand der Ausgleichsgeraden zum Nullpunkt seit dem letzten Rücksetzen.

`double` **`StdDevRegCoeff`** `() const`

liefert die Standardabweichung der Regressionskoeffizienten seit dem letzten Rücksetzen.

`double` **`CorrCoeff`** `() const`

liefert den Korrelationskoeffizienten seit dem letzten Rücksetzen.

`bool` **`xConstant`** `() const`

liefert true, wenn die x-Werte seit dem letzten Rücksetzen konstant waren.

`bool` **`yConstant`** `() const`

liefert true, wenn die y-Werte seit dem letzten Rücksetzen konstant waren.

`void` **`Reset`** `() const`

setzt die bisher geführte Statistik zurück.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über das Regressionsanalyseobjekt berichten kann.

6.2.44 Reportable

Basisklassen: `NamedObject[1], ModelComponent[2]`

Assoziationen zu: `Model, Reporter, SimTime, String`

in Datei: `reportab.h`

Beschreibung: `Reportable` ist Oberklasse für alle reportfähigen Objekte, über die Informationen im Report ausgegeben werden können. Sie zeichnen sich vor allem dadurch aus, daß sie auf Verlangen einen `Reporter` erzeugen können, der alle wichtigen Informationen aus dem Objekt extrahieren und für den Report aufbereiten kann. Reporter stellen somit eine Schnittstelle zwischen Report und den reportfähigen Objekten her. Ferner wird ein Zähler geführt, der von Unterklassen meist dazu benutzt wird, um die Zahl der Benutzungen festzuhalten. Über `Reset` können der Zähler und die in Unterklassen gesammelten Daten zurückgesetzt werden. `ResetAt` liefert den Zeitpunkt der letzten Rücksetzung.

Eine globale Rücksetzung aller reportfähigen Objekte kann über die `Reset`-Methode des Experiments oder des Hauptmodells bewirkt werden. Dabei sie auf einen Unterschied hingewiesen:

`Experiment::Reset` wartet auf das Ende des Simulationszeitpunktes, zu dem die Rücksetzung vorgenommen werden soll, während `Model::Reset` alle die zu ihm gehörenden reportfähigen Objekte unmittelbar zurücksetzt.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], Name[1ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
Reportable   (Model& owner, const String& name = "",
              bool   showInReport = true
              bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Komponente bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
bool ShowInReport () const
```

liefert `true`, wenn das Objekt im Report erscheinen soll, sonst `false`.

```
void ShowInReport (bool show)
```

stellt über `show` ein, ob das Objekt im Report erscheinen soll.

```
unsigned long Observations () const
```

liefert die Zahl der seit dem letzten `Reset` erfolgten Benutzungen.

```
void Reset ()
```

virtuell, führt eine Rücksetzung der gesammelten Daten durch. Außerdem wird der Zeitpunkt der letzten Rücksetzung auf den Zeitpunkt des Aufrufs gesetzt. `Reset` kann in Unterklassen überschrieben werden, um dort definierte Daten ebenfalls zurückzusetzen. Es sollte dort aber zunächst die `Reset`-Methode der jeweiligen Oberklasse aufgerufen werden.

```
SimTime ResetAt () const
```

liefert den Zeitpunkt der letzten Rücksetzung.

```
Reporter* NewReporter () const = 0
```

virtuell, muß in Unterklassen definiert werden, so daß ein Zeiger auf einen mittels `new` im Moment des Aufrufs erzeugter Reporter geliefert wird. Kann oder soll kein Bericht für dieses Objekt erzeugt werden, so ist ein Null-Zeiger zu liefern. Falls ein Reporter zurückgegeben wird, ist der aufrufende Kontext für die Zerstörung des Reporters nach Gebrauch mittels `delete` zuständig. Alle im Rahmen von DESMO-C angebotenen Klassen für reportfähige Objekte definieren diese Methode bereits, so daß sie vom Modellprogrammierer außer Acht gelassen werden kann.

geschützte Methoden:

```
void IncObservations (unsigned long inc = 1)
```

erhöht den internen Zähler für die Zahl der Benutzungen/Beobachtungen um `inc`. Diese Methode wird von Unterklassen aufgerufen. Sie wird nur benötigt, falls neue Klassen von reportfähigen Objekten entwickelt werden sollen.

6.2.45 Res

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], QueueBased[4]`

Assoziationen zu: `Model, Process, Reporter, SimTime, String`

in Datei: `res.h`

Beschreibung: Mit Hilfe der Klasse `Res` lassen sich Mechanismen für Ressourcenwettbewerb modellieren. Es können verschiedene kapazitätsbeschränkte Pools von Ressourcen definiert werden, von denen später Prozesse mittels `Acquire` einzelne Einheiten angefordert können. Mittels `Release` können diese wieder freigegeben werden.

Es wird eine implizite Warteschlange für Prozesse geführt, deren Nachfrage nicht sofort befriedigt werden kann. Dabei wird nach Priorität und dann nach FIFO-Strategie entschieden, in welcher Reihenfolge den wartenden Prozessen die Ressourcen zugeteilt werden. Prozesse werden ggf. solange blockiert, bis ihr Bedarf befriedigt werden kann. Dabei blockiert ein wartender Prozeß mit einer hohen Anforderung u.U. hinter ihm wartende Prozesse, obwohl deren Anforderung vielleicht schon befriedigt werden könnte. Ressourcen können nicht direkt zwischen Prozessen ausgetauscht werden. Alle Ressourcen, die ein Prozeß belegt hat, müssen von ihm irgendwann auch wieder freigegeben werden.

Bei Anforderungen von Ressourcen können Deadlocks auftreten. Es wird eine dynamische Überwachung durchgeführt, die sich auf die tatsächliche momentane Ressourcenvergabe bezieht (darstellbar durch einen Allokationsgraf). Die Art der Deadlock-Überwachung kann für ein Experiment eingestellt werden, in dem die Methode `DeadLockCheck` (s. `Experiment`) aufgerufen wird. Werden einzelne Ressource-Einheiten gerade benutzt, kann der Überwachungslevel nur noch ausgeschaltet werden.

Die von `QueueBased` geerbten Methoden beziehen sich auf die implizite Warteschlange der blockierten Prozesse. Darüber hinaus stellt `Res` Informationen über die Anzahl der vollständigen Belegungs-Freigabe-Sequenzen sowie minimale, mittlere und aktuelle Anzahl verfügbarer Ressourcen zur Verfügung.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0`.

geerbte Methoden: `AvgLength[4ö], AvgWaitTime[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g],`

```
DebugOn[2g], DebugOff[2g], Empty[4ö],
Epsilon[2ö], Err[2ö], Error[2g],
FatalError[2g], GetModel[2ö], In[2ö],
IsExperimentCompatible[2ö],
IsModelCompatible[2ö], IncObservations[3g],
Length[4ö], MaxLength[4ö], MaxLengthAt[4ö],
MaxWaitTime[4ö], MaxWaitTimeAt[4ö],
MinLength[4ö], MinLengthAt[4ö], Name[1ö],
NewReporter[4ö], NOW[2ö], NullEntity[2ö],
NullEvent[2ö], NullProcess[2ö],
Observations[3ö], Out[2ö], QuotedName[1ö],
Rename[1ö], Reset[4ö], ResetAt[3ö],
SendMessage[2g], ShowInReport[3ö],
ShowInTrace[2ö], SkipTraceNote[2ö],
StdDevLength[4ö], StdDevWaitTime[4ö],
TraceIsOn[2g], TraceNote[2g], TraceOff[2g],
TraceOn[2g], Valid[2ö], valid[2g],
Warning[2g], ZeroWaits[4ö]
```

Methoden:

```
Res     (Model& owner, const String& name = "",
         unsigned long capacity     = 1,
         bool          showInReport = true
         bool          showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `capacity` gibt an, wieviele Einheiten der Ressourcenpool umfaßt. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

Res `(const Res& obj)`

Der Kopier-Konstruktor erzeugt aus `obj` einen neuen vollständigen Ressourcenpool mit den selben statistischen Daten wie `obj`, jedoch enthält die implizite Warteschlange keine Prozesse, d.h. sie ist leer und somit gilt: `MinLength = 0` und `MinLengthAt = CurrentTime`.

~Res `()`

Der Destruktor entfernt alle noch wartenden Prozesse aus der Warteschlange. Die Prozesse selbst werden nicht gelöscht.

`void` **Acquire** `(unsigned long n)`

Anforderung von `n` Einheiten des Ressourcenpools. Bei zu geringem Vorrat wird der betreffende Prozeß (nach Priorität bzw. FIFO) in eine Warteschlange eingefügt und solange blockiert, bis die Anforderung erfüllt werden kann. Anforderungen mit `n` = 0 werden auf jeden Fall ohne Verzögerungen bedient.

`void` **`Release`** `(unsigned long n)`

Rückgabe von `n` Einheiten an den Ressourcenpool. die ggf. wartenden Prozessen gemäß der Reihenfolge in der Warteschlange zugeteilt werden.

`void` **`ChangeLimit`** `(unsigned long n)`

Ändert die Anzahl der im Pool befindlichen Einheiten an Ressourcen auf n. Dies ist nur vor Benutzung oder nach einem `Reset` möglich, sofern in dem Moment keine Einheiten belegt sind. Andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

`void` **`Reset`** `()`

virtuell, setzt die Statistik des Ressourcenpools zurück.

`unsigned long` **`Limit`** `() const`

liefert die Anzahl maximal verfügbarer Ressourcen (entspricht dem Konstruktor-Argument `capacity`).

`unsigned long` **`Avail`** `() const`

liefert die Anzahl der momentan verfügbaren Ressourcen.

`unsigned long` **`Minimum`** `() const`

liefert die kleinste Anzahl von Ressourcen, die seit dem letzten Rücksetzen verfügbar waren.

`unsigned long` **`Users`** `() const`

liefert die Anzahl der seit dem letzten Rücksetzen erfolgten Rückgaben.

`double` **`AvgUsage`** `() const`

liefert die mittlere Auslastung des Ressourcenpools seit der letzten Rücksetzung.

`bool` **`ReleasedAll`** `(Process& p, unsigned long& n)`

liefert `true`, wenn `p` alle Einheiten der Ressource zurückgegeben hat, sonst `false`. In `n` wird auf jeden Fall die Anzahl der von `p` belegten Einheiten zurückgegeben (ggf. 0).

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über die Ressource berichten kann.

6.2.46 Schedulable

Basisklassen: `NamedObject[1], ModelComponent[2], DynamicalObject[3]`

Assoziationen zu: `Event, Entity, Model, Process, SimTime, String`

in Datei: `schedula.h`

Beschreibung: `Schedulable` ist Oberklasse für alle vormerkbaren Objekte. Über bestimmte Methoden, die teilweise erst in Unterklassen hinzukommen, können sie auf die Ereignisliste gesetzt (vorgemerkt) werden. Es werden Methoden angeboten, die sowohl auf Ereignisse als auch auf Entities bzw. Prozesse anwendbar sind und in drei Kategorien einteilbar sind:

- Abfrage, ob und wenn ja, wann das Objekt vorgemerkt ist
- Nachfolger auf der Ereignisliste
- Verschieben auf oder Entfernen von der Ereignisliste

Außerdem werden vormerkbare Objekte automatisch mit einer Seriennummer versehen, die innerhalb des Modells für jeden Namen einzeln geführt wird. Diese Nummer wird bei der Erzeugung an den Namen gehängt. Evtl. wird der Name vorher gekürzt, um die maximale Namenslänge (siehe `ExperimentOpts`) nicht zu überschreiten. Die Nummer selbst wird modulo 10^n berechnet, wobei n die Anzahl der Stellen ist, die für die Numerierung verwendet werden soll (siehe `ExperimentOpts`).

geerbte Methoden: `CheckDeleteOnTermination[3ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], DeleteOnTermination[3ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], Name[1ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
Schedulable  (Model& owner, const String& name = "",
              bool   showInTrace = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und `Schedulable` bleibt über die gesamte Lebensdauer bestehen. `showInTrace` gibt an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` erfolgen kann. Das Objekt wird zunächst mit dem übergebenen Namen konstruiert und anschließend mittels `Rename` um eine Seriennummer ergänzt.

```
Schedulable (const Schedulable&)
```

Der Kopier-Konstruktor erzeugt ein neues Objekt und übernimmt alle Daten der übergebenen Kopiervorlage. Auch die Seriennummer wird übernommen. Das neue Objekt ist jedoch nicht vorgemerkt, auch wenn das zu kopierende Objekt auf der Ereignisliste steht.

```
~Schedulable ()
```

Der Destruktor überprüft zunächst, ob das Objekt vorgemerkt ist und entfernt es nach Ausgabe einer Warnung von der Ereignisliste. Diese Meldung wird jedoch unterdrückt, wenn die Destruktion im Rahmen der Freispeicherverwaltung stattfindet, die beim Löschen des zugehörigen Modells durchgeführt wird.

```
bool IsNull () const
```

liefert `true`, wenn das Objekt eines der Pseudo-Objekte (`NullEvent`, `NullEntity`, `NullProcess`) ist, andernfalls `false`

```
bool IsCurrent () const
```

liefert `true`, wenn das Objekt das gerade aktive ist (`CurrentEvent`, `CurrentEntity` oder `CurrentProcess`), andernfalls `false`. Das aktuelle Objekt steht nicht auf der Ereignisliste, sofern es nicht bereits wieder vorgemerkt wurde.

```
bool IsScheduled () const
```

liefert `true`, wenn das Objekt bereits vorgemerkt ist und somit auf der Ereignisliste steht, andernfalls `false`.

```
SimTime ScheduledAt () const
```

liefert den Zeitpunkt, zu dem das Objekt vorgemerkt ist, falls es auf der Ereignisliste steht, andernfalls wird eine negative Zeit zurückgegeben.

```
Schedulable& Next() const
```

versucht, das auf der Ereignisliste nachfolgende Entity zu liefern. Ist der Nachfolger ein externes Ereignis, wird dies zurückgegeben. Ist das Objekt selbst weder vorgemerkt noch das aktuelle (`Current`) oder gibt es keinen Nachfolger, so wird das Pseudo-Objekt `NullEntity` geliefert.

`Event& **NextEvent** () const`

liefert das Nachfolgeereignis auf der Ereignisliste. Das Objekt muß vorgemerkt oder das aktive (`Current`) sein, sonst wird nach Ausgabe einer Warnung das Pseudo-Objekt `NullEvent` zurückgegeben. Gibt es keinen Nachfolger oder ist dieser ein Prozeß, wird ebenfalls `NullEvent` geliefert.

`Entity& **NextEntity** () const`

liefert das Nachfolge-Entity auf der Ereignisliste. Das Objekt muß vorgemerkt oder das aktive (`Current`) sein, sonst wird nach Ausgabe einer Warnung das Pseudo-Objekt `NullEntity` zurückgegeben. Gibt es keinen Nachfolger oder ist dieser ein externes Ereignis, wird ebenfalls `NullEntity` geliefert.

`Process& **NextProcess** () const`

liefert den Nachfolgeprozeß auf der Ereignisliste. Das Objekt muß vorgemerkt oder das aktive (`Current`) sein, sonst wird nach Ausgabe einer Warnung das Pseudo-Objekt `NullProcess` zurückgegeben. Gibt es keinen Nachfolger oder ist dieser kein Prozeß, wird ebenfalls `NullProcess` geliefert.

`void **ReSchedule** (SimTime dt)`

verschiebt das Objekt auf der Ereignisliste um `dt` auf `now + dt`. Ist das Objekt nicht vorgemerkt, wird eine Warnung ausgegeben und der Aufruf ignoriert. Ist `dt = NOW`, so wird ein evtl. aktiver Prozeß verdrängt.

`void **Cancel** ()`

entfernt das Objekt von der Ereignisliste, falls es vorgemerkt war. Andernfalls wird eine Warnung ausgegeben und der Aufruf ignoriert.

6.2.47 SimTime

Basisklassen: keine

Assoziationen zu:

in Datei: `simtime.h`

Beschreibung: `SimTime` ist die Klasse Simulationszeitpunkte. Sie besteht im Wesentlichen aus zwei Konstruktoren und einer Reihe von Operatoren für eine intuitive Benutzung. Wird der zur internen Repräsentation verwendete Wert vom Typ `double` benötigt, so kann dieser über die Methode `Time` abgefragt werden. Die Methoden sprechen für sich und sind daher hier nur aufgelistet. Außerdem ist hier die Simulationszeitkonstante `NOW` definiert, die über die Methode `Now` abfragbar ist.

geerbte Methoden: `keine`

Methoden:

```
SimTime ()

SimTime (double t)
```

Zugriffsmethoden:

```
static SimTime Now ()

double Time () const

String AsString (int width, int precision) const
```

Operatoren:

```
SimTime& operator= (const SimTime& t)

SimTime& operator+= (const SimTime& t)

SimTime& operator-= (const SimTime& t)

SimTime& operator*= (const SimTime& t)

SimTime& operator/= (const SimTime& t)

SimTime operator/ (const SimTime& t) const

SimTime operator* (const SimTime& t) const

SimTime operator+ (const SimTime& t) const

SimTime operator- (const SimTime& t) const


int operator< (const SimTime& t) const

int operator> (const SimTime& t) const

int operator<= (const SimTime& t) const

int operator>= (const SimTime& t) const

int operator== (const SimTime& t) const

int operator!= (const SimTime& t) const


friend ostream& operator<< (ostream&, const SimTime&)

friend istream& operator>> (istream&, SimTime&)
```

6.2.48 StatisticObject

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4]`

Assoziationen zu: `Model, Observable, String`

in Datei: `statobj.h`

Beschreibung: `StatObject` ist Oberklasse für alle Datensammelobjekte und erbt sowohl von `Reportable` als auch von `Observer`. Über die Oberklasse `Observer` wird die Fähigkeit, bestimmte Objekte (`Observable`) zu beobachten, zu den Merkmalen eines reportfähigen Objekts (Oberklasse `Reportable`) hinzugefügt. Damit besteht für die Unterklassen die Möglichkeit, die Statistik immer dann fortzuschreiben, wenn sich das beobachtete Objekt (`Observable`) ändert (s. `Observer`). Die von `Observer` geerbte Methode `NoteChange` ist so implementiert, daß sie die neu eingeführte rein virtuelle Methode `Update` aufruft. Diese ist von Unterklassen zu implementieren, so daß die Statistik aktualisiert wird.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0.`

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NoteChange[4ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

```
StatisticObject    (      Model&  owner,
                    const String& name = "",
                          bool    showInReport = true
                          bool    showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann. Das neu konstruierte Objekt beobachtet kein anderes. Für automatische Aktualisierung muß erst ein Aufruf von `Observe` (s. `Observer`) erfolgen.

`void` **`Update`** `()`

rein virtuell, wird in Unterklassen implementiert, um zu beschreiben, wie eine Aktualisierung der Statistik durchzuführen ist.

`void` **`NoteChange`** `(Observable* obs)`

virtuell, implementiert die rein virtuelle Methode der Oberklasse `Observer` durch einen Aufruf von `Update`. Sie wir von einem evtl. zu beobachtenden Objekt aufgerufen, wenn es sich ändert. Ein Zeiger auf dieses Objekt wird in `obs` übergeben.

6.2.49 StdDebug

Basisklassen: `MessageReceiver[1], Output[2], StdOutput[3]`

Assoziationen zu: `Message, ostream, String`

in Datei: `stdoutp.h`

Beschreibung: `StdDebug` kann benutzt werden, um Debug-Nachrichten unter Beibehaltung der Standardformatierung in ein eigenes Stream-Objekt zu leiten. Dazu muß lediglich ein Objekt von `StdDebug` erzeugt werden und über die Methode `AddDebugOutput` des entsprechenden Experiments mit dessen Debug-Kanal verbunden werden.

geerbte Methoden: `Box[3g], Box2[3g], ClockTime[3g], Empty[2ö], GetOstream[2ö], GetOutputTitle[3ö], GetWidth[2ö], Line[3g], Note[1ö], Rename[2ö], SwitchOff[3ö], SwitchOn[3ö], TakeReporter[1ö], wrap[3g], WriteHeader[3g]`

Methoden:

`StdDebug` `(ostream& os, unsigned width = 78)`

Konstruktor der die Ausgabe mit dem Stream `os` verbindet und die Ausgabebreite `width` benutzt.

```
StdDebug      (const String&  fileName,
               const String&  extension = ".dbg",
                     unsigned width     = 78)
```

Konstruktor der die Ausgabe mit einer Datei verbindet, die den Namen `fileName` und die Dateierweiterung `extension` trägt. Dabei muß `extension` mit einem Punkt beginnen. Es wird die Ausgabebreite `width` benutzt.

`String` **`GetOutputTitle`** `() const`

liefert "Debug".

`void` **`Note`** `(const Message& msg)`

bereitet die entsprechende Nachricht für die Debug-Ausgabe auf und gibt sie aus.

6.2.50 StdError

Basisklassen: `MessageReceiver[1], Output[2], StdOutput[3]`

Assoziationen zu: `Message, ostream, String`

in Datei: `stdoutp.h`

Beschreibung: `StdError` kann benutzt werden, um Fehlernachrichten unter Beibehaltung der Standardformatierung in ein eigenes Stream-Objekt zu leiten. Dazu muß lediglich ein Objekt von `StdError` erzeugt werden und über die Methode `AddErrorOutput` des entsprechenden Experiments mit dessen Error-Kanal verbunden werden.

geerbte Methoden: `Box[3g], Box2[3g], ClockTime[3g], Empty[2ö], GetOstream[2ö], GetOutputTitle[3ö], GetWidth[2ö], Line[3g], Note[1ö], Rename[2ö], SwitchOff[3ö], SwitchOn[3ö], TakeReporter[1ö], wrap[3g], WriteHeader[3g]`

Methoden:

`StdError` `(ostream& os, unsigned width = 78)`

Konstruktor der die Ausgabe mit dem Stream `os` verbindet und die Ausgabebreite `width` benutzt.

```
StdError     (const String&  fileName,
              const String&  extension = ".err",
                    unsigned width     = 78)
```

Konstruktor der die Ausgabe mit einer Datei verbindet, die den Namen `fileName` und die Dateierweiterung `extension` trägt. Dabei muß `extension` mit einem Punkt beginnen. Es wird die Ausgabebreite `width` benutzt.

`String` **`GetOutputTitle`** `() const`

liefert "Error".

`void` **`Note`** `(const Message& msg)`

bereitet die entsprechende Nachricht für die Fehlerausgabe auf und gibt sie aus.

6.2.51 StdOutput

Basisklassen: `MessageReceiver[1], Output[2]`

Assoziationen zu: `Message, ostream, String`

in Datei: `stdoutp.h`

Beschreibung: `StdOutput` ist die Oberklasse für alle DESMO-C-Standardausgaben. Sie wiederholt die Konstruktoren der Oberklasse `Output` und stellt weitere von den Unterklassen benötigte geschützte Methoden zur Formatierung der Ausgabe bereit. Über die öffentlichen Methoden steuern die Ausgabemanager die Ausgabe. Durch die rein virtuelle Methode `GetOutputTitle`, die in den Unterklassen definiert werden muß, wird `StdOutput` zur abstrakten Klasse.

geerbte Methoden: `Empty[2ö], GetOstream[2ö], GetWidth[2ö], Note[1ö], Rename[2ö], TakeReporter[1ö]`

Methoden:

`StdOutput` `(ostream& os, unsigned width = 78)`

Konstruktor der die Ausgabe mit dem Stream `os` verbindet und die Ausgabebreite `width` benutzt.

```
StdOutput     (const String&  fileName,
               const String&  extension,    // inkl. '.'
                     unsigned width = 78)
```

Konstruktor der die Ausgabe mit einer Datei verbindet, die den Namen `fileName` und die Dateierweiterung `extension` trägt. Dabei muß `extension` mit einem Punkt beginnen. Es wird die Ausgabebreite `width` benutzt.

`String` **`GetOutputTitle`** `() const = 0`

rein virtuell, muß in den Unterklassen definiert werden und sollte den Titel der Ausgabe zurückliefern (z.B. "Trace").

`void` **`SwitchOn`** `(const Message& dummy)`

virtuell, informiert die Ausgabe darüber, daß der entsprechende Kanal eingeschaltet wurde. Die Nachricht `dummy` dient dem Zugriff auf das Experiment und der aktuellen Simulationszeit. Die Standardimplementierung schreibt einen Rahmen mit Experimentnamen und Ausgabetitel in die Ausgabe.

```
void SwitchOff      (const Message& dummy,
                     const String&  what)
```

virtuell, informiert die Ausgabe darüber, daß der entsprechende Kanal ausgeschaltet wurde. Die Nachricht `dummy` dient dem Zugriff auf das Experiment und der aktuellen Simulationszeit. Der String `what` wird zusammen mit dem Ausgabetitel in einem Rahmen ausgegeben (z.B. `what = "switched off"`)

geschützte Methoden:

```
void Box (const String& title)
```

setzt `title` zentriert in einen Rahmen aus '*' und gibt ihn in den Ausgabestrom aus.

```
void Box2 (const String& title1, const String& title2)
```

setzt `title1 und title2` jeweils zentriert in eine Zeile, umgibt beide mit einen Rahmen aus '*' und gibt ihn in den Ausgabestrom aus.

```
void Line (char c = '-')
```

gibt eine Linie aus `c`-Zeichen in den Ausgabestrom aus.

```
void ClockTime (const SimTime& t)
```

gibt die Zeit `t` zentriert in den Ausgabestrom aus.

```
void WriteHeader ()
```

virtuell, gibt den Unterklassen die Möglichkeit, durch Überschreiben eine Kopfzeile auszugeben, nachdem der jeweilige Kanal eingeschaltet wurde (wird z.B. vom `StdTrace` benötigt).

```
void wrap     (const String&  s,
               unsigned offset,
               unsigned indent = 0)
```

nimmt den String s und gibt ihn in der Breite `width - offset` aus. Ist er länger als diese Breite, wird er umbrochen und bis offset eingerückt. Ist für `indent` ein Wert angegeben, so wird ab der zweiten Zeile um `indent` mehr eingerückt.

6.2.52 StdReport

Basisklassen: `MessageReceiver[1], Output[2], StdOutput[3]`

Assoziationen zu: `Message, ostream, Reporter, String`

in Datei: `stdoutp.h`

Beschreibung: `StdReport` kann benutzt werden, um Fehlernachrichten unter Beibehaltung der Standardformatierung in ein eigenes Stream-Objekt zu leiten. Dazu muß lediglich ein Objekt von `StdReport` erzeugt werden und über die Methode `AddReportOutput` des entsprechenden Experiments mit dessen Report-Kanal verbunden werden.

geerbte Methoden: `Box[3g], Box2[3g], ClockTime[3g], Empty[2ö], GetOstream[2ö], GetOutputTitle[3ö], GetWidth[2ö], Line[3g], Note[1ö], Rename[2ö], SwitchOff[3ö], SwitchOn[3ö], TakeReporter[1ö], wrap[3g], WriteHeader[3g]`

Methoden:

`StdReport` `(ostream& os, unsigned width = 78)`

Konstruktor der die Ausgabe mit dem Stream `os` verbindet und die Ausgabebreite `width` benutzt.

```
StdReport     (const String&  fileName,
               const String&  extension = ".rpt",
                     unsigned width     = 78)
```

Konstruktor der die Ausgabe mit einer Datei verbindet, die den Namen `fileName` und die Dateierweiterung `extension` trägt. Dabei muß `extension` mit einem Punkt beginnen. Es wird die Ausgabebreite `width` benutzt.

`String` **`GetOutputTitle`** `() const`

liefert "Report".

`void` **`Note`** `(const Message& msg)`

bereitet die entsprechende Nachricht für die Report-Ausgabe auf und gibt sie aus.

`void` **`TakeReporter`** `(Reporter& r)`

entnimmt dem Reporter die Informationen und bereitet sie entsprechend für die Report-Ausgabe auf.

geschützte Methoden:

`void` **`WriteBeginOfGroup`** `(Reporter&)`

bereitet einen Reportergruppenwechsel vor und gibt alle Gruppeninformationen aus.

6.2.53 StdTrace

Basisklassen: `MessageReceiver[1], Output[2], StdOutput[3]`

Assoziationen zu: `Message, ostream, String`

in Datei: `stdoutp.h`

Beschreibung: `StdTrace` kann benutzt werden, um Fehlernachrichten unter Beibehaltung der Standardformatierung in ein eigenes Stream-Objekt zu leiten. Dazu muß lediglich ein Objekt von `StdTrace` erzeugt werden und über die Methode `AddTraceOutput` des entsprechenden Experiments mit dessen Trace-Kanal verbunden werden.

geerbte Methoden: `Box[3g], Box2[3g], ClockTime[3g], Empty[2ö], GetOstream[2ö], GetOutputTitle[3ö], GetWidth[2ö], Line[3g], Note[1ö], Rename[2ö], SwitchOff[3ö], SwitchOn[3ö], TakeReporter[1ö], wrap[3g], WriteHeader[3g]`

Methoden:

`StdTrace` `(ostream& os, unsigned width = 90)`

Konstruktor der die Ausgabe mit dem Stream `os` verbindet und die Ausgabebreite `width` benutzt.

```
StdTrace      (const String&  fileName,
               const String&  extension = ".trc",
                     unsigned width     = 90)
```

Konstruktor der die Ausgabe mit einer Datei verbindet, die den Namen `fileName` und die Dateierweiterung `extension` trägt. Dabei muß `extension` mit einem Punkt beginnen. Es wird die Ausgabebreite `width` benutzt.

`String` **`GetOutputTitle`** `() const`

liefert "Trace".

`void` **`Note`** `(const Message& msg)`

bereitet die entsprechende Nachricht für die Trace-Ausgabe auf und gibt sie aus.

6.2.54 String

Basisklassen: keine

Assoziationen zu:

in Datei: `str.h`

Beschreibung: `String` ist die Klasse der in DESMO-C verwendeten Zeichenketten. Sie besteht im Wesentlichen aus einer Reihe von Konstruktoren und Operatoren für eine intuitive Benutzung. Die Methoden sprechen für sich und sind daher hier nur aufgelistet.

geerbte Methoden: `keine`

Methoden:

Konstruktoren:

```
String (void)

String (const char*)

String (const String&)

String (const String&, unsigned)

String (char)

String (char, unsigned)

String (strstream&)

String (const void*)

String (int)

String (unsigned)

String (long)

String (unsigned long)

String (double)

String (bool)

~String (void)
```

Selektoren:

```
const char* Get (void) const

unsigned Length (void) const
```

Vergleichsoperatoren:

```
bool operator== (const String &s2) const

bool operator!= (const String &s2) const

bool operator< (const String &s2) const

bool operator<= (const String &s2) const

bool operator> (const String &s2) const

bool operator>= (const String &s2) const

int Compare (const String&) const

int Compare (const char*) const
```

Gloobale Freund-Vergleichsoperatoren:

```
bool operator== (const char* s1, const String &s2)

bool operator!= (const char* s1, const String &s2)

bool operator< (const char* s1, const String &s2)

bool operator<= (const char* s1, const String &s2)

bool operator> (const char* s1, const String &s2)

bool operator>= (const char* s1, const String &s2)
```

Suche:

```
int Find (const char, unsigned pos = 0) const

int Find (const String&, unsigned pos = 0) const
```

Teilstrings:

```
String Left (int) const

String Right (int) const

String Substr (int from, int n = -1) const
```

Ausgabe (globale Freund-Methode):

```
ostream& operator<< (ostream &o, const String &s)
```

Zuweisung:

```
String& operator= (const String&)

String& operator= (const char*)

String& operator+= (const String&)
```

Konkatenation:

```
String operator+ (const String&) const

friend String operator+ (const char*, const String&)
```

Manipulation:

```
int Replace (const String &substr,
             const String &repstr, unsigned pos = 0)

int Replace (const char substr,
             const char repstr, unsigned pos = 0)

String& Insert (const String&, unsigned pos = 0)

String& LTrim (void)

String& RTrim (void)

String& Trim (void)
```

6.2.55 Tally

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5], ValueStatistics[6]`

Assoziationen zu: `Model, Reporter, String, ValueSupplier`

in Datei: `tally.h`

Beschreibung: `Tally` ist die Klasse von Datensammelobjekten zur statistischen Erfassung einer Beobachtungsgröße ohne zeitliche Gewichtung. Dabei wird ein `Tally`-Objekt mit einem Wertlieferanten (`ValueSupplier`) verbunden, der auf Verlangen den zu beobachtenden Wert (Beobachtungsgröße) berechnet und übergibt. Mit `Update` wird die Beobachtungsgröße ermittelt und die Statistik aktualisiert. Für die Berechnung des Mittelwertes wird die Anzahl der `Update`-Aufrufe herangezogen.

Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, immer dann die Statistik fortzuschreiben, wenn sich die Beobachtungsgröße geändert hat. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Aktualisierung der Statistik mit Hilfe des vom `ValueSupplier`-Objekt gelieferten Wertes. `Observable` kann als Mixin-Klasse verwendet werden, um Objekte beobachtbar zu machen.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0.`

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Maximum[6ö], Mean[6ö], Minimum[6ö], Name[1ö], NewReporter[3ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[6ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], StdDev[6ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g],`

```
Update[6ö], Valid[2ö], valid[2g], Value[6ö],
Warning[2g]
```

Methoden:

```
Tally  (      Model&          owner,
        const String&         name,
              ValueSupplier&  vs,
              bool            showInReport = true,
              bool            showInTrace  = false)
```

Dem Konstruktor müssen mindestens das zugehörige Modell, ein Name und ein `ValueSupplier`-Objekt übergeben werden. Die Verbindung zwischen Modell und Datensammelobjekt bleibt über die gesamte Lebensdauer bestehen. In `vs` wird das Objekt übergeben, das auf Verlangen die Beobachtungsgröße berechnen und liefern kann. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
Tally  (Model&          owner,
        ValueSupplier&  vs,
        bool            showInReport = true,
        bool            showInTrace  = false)
```

Der Konstruktor bietet eine Alternative zum ersten Konstruktor und unterscheidet sich von diesem nur dadurch, daß hier die Angabe für den Namen entfallen kann.

`void` **`Update`** `()`

virtuell, läßt sich vom `ValueSupplier`-Objekt den aktuellen Wert der Beobachtungsgröße liefern, um die Statistik zu aktualisieren.

`double` **`Mean`** `() const`

liefert den Mittelwert der Beobachtungsgröße über die Anzahl der `Update`-Aufrufe seit dem letzten Rücksetzen.

`double` **`StdDev`** `() const`

liefert die Standardabweichung der Beobachtungsgröße seit dem letzten Rücksetzen.

`void` **`Reset`** `() const`

setzt die bisher geführte Statistik zurück.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über das Datensammelobjekt berichten kann.

6.2.56 TimeSeries

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5]`

Assoziationen zu: `Model, Reporter, SimTime, String, ValueSupplier`

in Datei: `timeseri.h`

Beschreibung: `TimeSeries` ist die Klasse für in Dateien auszugebende Zeitreihen. Eine Zeitreihe wird mit einem Wertlieferanten (`ValueSupplier`) verbunden, der auf Verlangen den zu beobachtenden Wert (Beobachtungsgröße) berechnet und übergibt. Mit `Update` werden aktuelle Simulationszeit und Beobachtungsgröße zusammen in eine neue Zeile der Datei geschrieben. Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, immer dann einen Wert zu protokollieren, wenn er sich geändert hat. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Protokollierung des vom `ValueSupplier`-Objekt gelieferten Wertes. `Observable` kann als Mixin-Klasse verwendet werden, um Objekte beobachtbar zu machen. Eine andere Alternative besteht in der Wahl für automatische Aktualisierung vor jedem Stellen der Simulationsuhr.

Die Rücksetzung eines `TimeSeries`-Objekts mittels `Reset` löscht die bisher protokollierten Daten in der Datei und beginnt von neuem.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0`.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö],`

```
SkipTraceNote[2ö], TraceIsOn[2g],
TraceNote[2g], TraceOff[2g], TraceOn[2g],
Update[5ö], Valid[2ö], valid[2g], Warning[2g]
```

Methoden:

```
TimeSeries   (      Model&          owner,
              const String&         name,
              const String&         fileName,
                    ValueSupplier&  vs,
              const SimTime&        start     = 0.0,
              const SimTime&        end       = 0.0,
                    bool            automatic = true,
              const String&         separator = ",")
```

Dem Konstruktor müssen mindestens das zugehörige Modell, ein Name, ein Dateiname und ein `ValueSupplier`-Objekt übergeben werden. Die Verbindung zwischen Modell und Zeitreihe bleibt über die gesamte Lebensdauer bestehen. `name` (falls nicht "") wird als erste Zeile in die mit `fileName` bezeichnete Datei geschrieben. `fileName` muß den Namenskonvention des unterliegenden Betriebssystems entsprechen. In `vs` wird ein Objekt übergeben, daß auf Verlangen die Beobachtungsgröße berechnen und liefern kann. `start` und `end` geben den Zeitraum an, in dem der von `vs` gelieferte Wert protokolliert wird. `end` <= `start` bedeutet, daß während des ganzen Experiments protokolliert wird. Ist `automatic` = `true`, so wird vor jedem Stellen der Simulationsuhr protokolliert. `separator` wird benutzt, um Simulationszeit und Beobachtungsgröße von einander zu trennen. So ist es möglich, das Ausgabeformat anzupassen, damit die Daten in anderen Programmen (z.B. Tabellenkalkulation) importiert werden können.

```
TimeSeries   (const TimeSeries& obj)
```

Der Kopier-Konstruktor erzeugt ein neues Zeitreihenobjekt mit den selben Daten wie obj. Es wird keine neue Datei erzeugt, sondern in die Datei von obj geschrieben. Der Kopier-Konstruktor dient in erster Linie der Initialisierung von Arrays. Der Zuweisungsoperator soll nicht benutzt werden. Er ist als private deklariert und nicht implementiert.

```
double Value () const
```

läßt sich vom ValueSupplier-Objekt den aktuellen Wert der Beobachtungsgröße liefern und gibt ihn zurück. Ist das ValueSupplier-Objekt ungültig (weil inzwischen gelöscht), so wird eine Warnung ausgegeben und -1.0 zurückgegeben.

```
void Update ()
```

virtuell, läßt sich vom ValueSupplier-Objekt den aktuellen Wert der Beobachtungsgröße liefern und protokolliert diesen zusammen mit der aktuellen Simulationszeit in einer neuen Zeile der Protokolldatei. Zeit und Wert werden mit dem im Konstruktor übergebenen Separator-String getrennt.

```
void Reset () const
```

bewirkt neben den in `Reportable` beschriebenen Aktionen auch das Löschen des Protokolldateiinhalts, um die Protokollierung neu zu beginnen.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen Null-Zeiger, da Zeitreihen nicht im Report erscheinen sollen. Daher bleibt der Aufruf von `ShowInReport` bei Zeitreihen ohne Wirkung.

6.2.57 ValueStatistics

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3] + Observer[4], StatisticObject[5]`

Assoziationen zu: `Model, String, ValueSupplier`

in Datei: `valuesta.h`

Beschreibung: `ValueStatistics` ist die Klasse von Datensammelobjekten zur statistischen Erfassung einer Beobachtungsgröße. Dabei wird ein `ValueStatistics`-Objekt mit einem Wertlieferanten (`ValueSupplier`) verbunden, der auf Verlangen den zu beobachtenden Wert (Beobachtungsgröße) berechnet und übergebt. Mit `Update` wird die Beobachtungsgröße ermittelt und die Statistik aktualisiert. Es können Informationen über Maximum, Minimum, Mittelwert und Stdandardabweichung seit dem letzten Rücksetzen abgerufen werden. Dabei hängt die Art der Berechnung von Mittelwert und Standardabweichung davon ab, ob die Unterklasse ihre Statistik zeitgewichtet führt.

Von der Oberklasse `Observer` wird die Fähigkeit geerbt, bestimmte Objekte (`Observable`) zu beobachten. Damit besteht die Möglichkeit, immer dann die Statistik fortzuschreiben, wenn sich die Beobachtungsgröße geändert hat. Das Interesse an den Änderungen eines beobachtbaren Objekts (`Observable`) kann mittels der von `Observer` geerbten Methode `Observe` angemeldet werden. Das beobachtete Objekt sorgt dann bei einer Änderung für eine Aktualisierung der Statistik mit Hilfe des vom `ValueSupplier`-Objekt gelieferten Wertes. `Observable` kann als Mixin-Klasse verwendet werden, um Objekte beobachtbar zu machen.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0`.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Name[1ö], NewReporter[3ö], NoteChange[5ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Observe[4ö], Observing[4ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[3ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö],`

```
SkipTraceNote[2ö], TraceIsOn[2g],
TraceNote[2g], TraceOff[2g], TraceOn[2g],
Update[5ö], Valid[2ö], valid[2g], Warning[2g]
```

Methoden:

```
ValueStatistics     (Model&          owner,
                 const  String&      name,
                        ValueSupplier& vs,
                        bool         showInReport = true,
                        bool         showInTrace  = false)
```

Dem Konstruktor müssen mindestens das zugehörige Modell, ein Name und ein `ValueSupplier`-Objekt übergeben werden. Die Verbindung zwischen Modell und Datensammelobjekt bleibt über die gesamte Lebensdauer bestehen. In `vs` wird das Objekt übergeben, das auf Verlangen die Beobachtungsgröße berechnet und liefern kann. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

```
ValueStatistics     (Model&          owner,
                     ValueSupplier& vs,
                     bool         showInReport = true,
                     bool         showInTrace  = false)
```

Der Konstruktor bietet eine Alternative zum ersten Konstruktor und unterscheidet sich von diesem nur dadurch, daß hier die Angabe für den Namen entfallen kann.

`void` **`Update`** `()`

virtuell, läßt sich vom `ValueSupplier`-Objekt den aktuellen Wert der Beobachtungsgröße liefern, um die Statistik zu aktualisieren.

`double` **`Value`** `() const`

liefert den bei der letzten Aktualisierung ermittelten Wert der Beobachtungsgröße.

`double` **`Minimum`** `() const`

liefert den kleinsten Wert der Beobachtungsgröße seit dem letzten Rücksetzen.

`double` **`Maximum`** `() const`

liefert den größten Wert der Beobachtungsgröße seit dem letzten Rücksetzen.

`double` **`Mean`** `() const`

rein virtuell, wird in Unterklassen implementiert, um den (evtl. zeitgewichteten) Mittelwert der Beobachtungsgröße seit dem letzten Rücksetzen zu liefern.

`double` **`StdDev`** `() const`

rein virtuell, wird in Unterklassen implementiert, um die (evtl. zeitgewichtete) Standardabweichung der Beobachtungsgröße seit dem letzten Rücksetzen zu liefern.

`void` **`Reset`** `() const`

setzt die bisher geführte Statistik zurück.

6.2.58 ValueSupplier

Basisklassen: `NamedObject[1], ModelComponent[2]`

Assoziationen zu: `Model, String`

in Datei: `valuesup.h`

Beschreibung: `ValueSupplier` ist Oberklasse für alle Objekte, die den Wert einer Beobachtungsgröße liefern können. Sie werden von Datensammelobjekten (`StatisticObject`) benutzt, um zur Aktualisierung ihrer Statistik den aktuellen Wert einer Beobachtungsgröße zu ermitteln. Dafür muß in Unterklassen die rein virtuelle Methode `Value` implementiert werden, die den aktuellen Wert der Beobachtungsgröße berechnet und zurückgibt.

Durch Verwendung von `Observable` als Mixin-Klasse kann eine Beobachtungsgröße so eingerichtet werden, daß ein Datensammelobjekt vollautomatisch bei jeder Änderung der Beobachtungsgröße aktualisiert wird. Das Datensammelobjekt muß sich dazu mittels der `Observer`-Methode `Observe` bei der Beobachtungsgröße anmelden.

geerbte Methoden: `ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], Name[1ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Out[2ö], QuotedName[1ö], Rename[1ö], SendMessage[2g], ShowInTrace[2ö], SkipTraceNote[2ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g]`

Methoden:

`ValueSupplier` `(Model& owner, const String& name = "")`

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und `ValueSupplier`-Objekt bleibt über die gesamte Lebensdauer bestehen.

`double` **`Value`** `() const`

rein virtuell, muß in Unterklassen definiert werden, um den Wert einer Beobachtungsgröße zu liefern.

6.2.59 WaitQueue

Basisklassen: `NamedObject[1], ModelComponent[2], Reportable[3], QueueBased[4]`

Assoziationen zu: `Model, Process, Reporter, SimTime, String`

in Datei: `waitq.h`

Beschreibung: Mit Hilfe der Klasse `WaitQueue` sich die direkte Kooperation zweier Prozesse modellieren. Dabei wird von einer "Master-Slave"-Beziehung ausgegangen. Master und Slave signalisieren über die `WaitQueue` mittels `Cooperate` bzw. `Wait` ihren Kooperationswunsch. Kommt eine solche Kooperation zustande, wird eine gemeinsame Handlung der beiden Prozesse durchgeführt. Der Master bestimmt dabei die Art dieser Handlung. Danach laufen die Prozesse asynchron weiter und die Kooperation ist beendet.

Es wird je eine implizite Warteschlange für Master- bzw. Slave-Prozesse geführt, Kooperationswunsch nicht sofort befriedigt werden kann. Dabei wird nach Priorität und dann nach FIFO-Strategie entschieden, in welcher Reihenfolge den wartenden Prozesse einander zugeteilt werden. Prozesse werden ggf. solange blockiert, bis ihre Kooperation zustande gekommen ist. Mit Beginn der gemeinsamen Handlung werden sowohl Master als auch Slave aus den jeweiligen Warteschlangen entfernt. Neben einer Kooperation mit dem jeweils ersten wartenden Slave können Master auch durch die Angabe eines Synchronisationsbedingung (`Condition`: testet, ob der Slave die Bedingung erfüllt) bei `Cooperate` bzw. `Avail` mit geeigneten Kandidaten gezielt kooperieren bzw. nach solchen suchen. Müssen für das Testen einer Kooperationsbedingung Master und Slave miteindader verglichenwerden, so kann in der Unterklasse von Condition der Master als Attribut der Bedingung angelegt werden. In der Methode `Condition::Check` kann dann der Slave, der als Parameter übergeben wird, mit dem Master verglichen werden.

Die von `QueueBased` geerbten Methoden beziehen sich auf die implizite Warteschlange der blockierten Master-Prozesse. Synonym sind die entsprechenden Methoden mit dem Präfix 'm' zu verwenden. Die Methoden für die implizite Slave-Warteschlange beginnen jeweils mit dem Präfix 's'. Darüber hinaus stellt `Res` Informationen über die Anzahl der vollständigen Belegungs-Freigabe-Sequenzen sowie minimale, mittlere und aktuelle Anzahl verfügbarer Ressourcen zur Verfügung.

Bei unzulässigen Operationen werden, sofern kein Programmabbruch erfolgt, definierte Werte zurückgegeben. Falls nicht anders angegeben, gelten für die jeweiligen Typen folgende Standardwerte: `unsigned = 0, bool = false, double = -1.0, Process = NullProcess.`

geerbte Methoden: `AvgLength[4ö], AvgWaitTime[4ö], ClassName[1ö], CurrentTime[2ö], CurrentEntity[2ö], CurrentEvent[2ö], CurrentProcess[2ö], CurrentModel[2ö], Debug[2ö], DebugIsOn[2g], DebugOn[2g], DebugOff[2g], Empty[4ö], Epsilon[2ö], Err[2ö], Error[2g], FatalError[2g], GetModel[2ö], In[2ö], IsExperimentCompatible[2ö], IsModelCompatible[2ö], IncObservations[3g], Length[4ö], MaxLength[4ö], MaxLengthAt[4ö], MaxWaitTime[4ö], MaxWaitTimeAt[4ö], MinLength[4ö], MinLengthAt[4ö], Name[1ö], NewReporter[4ö], NOW[2ö], NullEntity[2ö], NullEvent[2ö], NullProcess[2ö], Observations[3ö], Out[2ö], QuotedName[1ö], Rename[1ö], Reset[4ö], ResetAt[3ö], SendMessage[2g], ShowInReport[3ö], ShowInTrace[2ö], SkipTraceNote[2ö], StdDevLength[4ö], StdDevWaitTime[4ö], TraceIsOn[2g], TraceNote[2g], TraceOff[2g], TraceOn[2g], Valid[2ö], valid[2g], Warning[2g], ZeroWaits[4ö]`

Methoden:

```
WaitQueue    (Model& owner, const String& name = "",
              bool   showInReport = true
              bool   showInTrace  = true)
```

Dem Konstruktor muß mindestens das zugehörige Modell übergeben werden. Die Verbindung zwischen Modell und Warteschlange bleibt über die gesamte Lebensdauer bestehen. `showInTrace` und `showInReport` geben jeweils an, ob für dieses Objekt evtl. Trace-Ausgaben erfolgen sollen bzw. ob es im Report erscheinen soll oder nicht, was auch nachträglich noch mit Hilfe der Methode `ShowInTrace` bzw. `ShowInReport` erfolgen kann.

`WaitQueue` `(const WaitQueue& obj)`

Der Kopier-Konstruktor erzeugt aus `obj` einen neuen vollständigen Synchronisationspunkt mit den selben statistischen Daten wie `obj`, jedoch enthalten die implizite Warteschlangen keine Prozesse, d.h. sie sind leer und somit gilt: m`MinLength = 0` und m`MinLengthAt = CurrentTime` sowie s`MinLength = 0` und s`MinLengthAt = CurrentTime`.

`~WaitQueue` `()`

Der Destruktor entfernt alle noch wartenden Prozesse aus der Warteschlange. Die Prozesse selbst werden nicht gelöscht.

`void` **`Wait`** `()`

Ruft ein Prozeß `Wait` auf, so signalisiert er seinen Kooperationswunsch als Slave. Wartet kein Master, so wird der Prozeß (nach Priorität bzw. FIFO) in eine Warteschlange eingefügt und solange blockiert, bis eine Kooperation erfolgt ist und beendet wurde. Findet sich ein kooperationsbereiter Master, so übernimmt dieser die Kontrolle während der Kooperation. Anschließend wird der Slave automatisch aktiviert, wenn dies nicht schon innerhalb der Kooperation geschehen ist.

`void` **`Cooperate`** `(ProcessCooperation& coop)`

Ruft ein Prozeß `Cooperate` auf, so signalisiert er seinen Kooperationswunsch als Master. Steht ein kooperationsbereiter Slave zur Verfügung, wird die gemeinsame Handlung coop mit dem ersten Element der Slave-Warteschlange ausgeführt. Nach dem Ende der Kooperation geht die Ablaufkontrolle zuerst an den Master und dann an den Slave über. Wartet kein Slave, so wird der Prozeß in die Master-Warteschlange eingereiht (gemäß Priorität und FIFO) und solange blockiert, bis ein kooperationsbereiter Slave eintrifft.

`void` **`Cooperate`** `(ProcessCooperation& coop, Condition& c)`

Ein Master signalisiert seinen Kooperationswunsch mit einem Slave, der zusätzlich die Bedingung `c` erfüllt. Warten mehrere Slaves, werden sie gemäß ihrer Eintragsreihenfolge in der Warteschlange überprüft. Wird ein passender Slave gefunden, kommt die gemeinsame Handlung coop zur Ausführung. Andernfalls wird der Master blockiert (s.o.).

`bool` **`Avail`** `(Process*& slave, Condition& c)`

liefert true, wenn in der `WaitQueue` ein Slave blockiert ist, der die Bedingung c erfüllt, ohne jedoch bei Erfolg eine direkte Kooperation aufzunehmen bzw. den Master bei Mißerfolg zu blockieren. Warten mehrere Slaves, werden sie gemäß ihrer Eintragsreihenfolge in der Warteschlange überprüft. Auf diese Weise kann z.B. gestaffelt mit unterschiedlich "scharfen" Kriterien nach kooperationsbereiten Prozessen gesucht werden. Wird ein passender Slave gefunden, so wird er im Zeiger `slave` zurückgegeben, andernfalls zeigt dieser auf das Pseudo-Objekt `NullProcess`. Mit einem auf diese Weise gefundenen Slave-Prozeß kann eine gemeinsame Handlung über die Prozedur `Process::Cooperate` (s. dort) aufgenommen werden.

`Reporter*` **`NewReporter`** `() const`

virtuell, liefert einen mittels `new` neu erzeugten Reporter, der über die `WaitQueue` berichten kann.

Die folgenden Methoden entsprechen den aus QueueBased geerbten, wobei sich die Präfixe "m" bzw. "s" auf die implizite Master- bzw. Slave-Warteschlangen beziehen:

```
bool mEmpty () const

unsigned long mLength () const

unsigned long mMinLength () const

unsigned long mMaxLength () const

double mAvgLength () const

double mStdDevLength () const

SimTime mMinLengthAt () const

SimTime mMaxLengthAt () const

SimTime mZeroWaits () const

SimTime mMaxWaitTime () const

SimTime mAvgWaitTime () const

SimTime mStdDevWaitTime () const

SimTime mMaxWaitTimeAt () const


bool sEmpty () const

unsigned long sLength () const

unsigned long sMinLength () const

unsigned long sMaxLength () const

double sAvgLength () const

double sStdDevLength () const

SimTime sMinLengthAt () const

SimTime sMaxLengthAt () const

SimTime sZeroWaits () const

SimTime sMaxWaitTime () const
```

```
SimTime sAvgWaitTime () const

SimTime sStdDevWaitTime () const

SimTime sMaxWaitTimeAt () const
```

7 Schlußbemerkungen und Ausblick

7.1 Schlußbemerkungen

Mit DESMO-C ist eine Realisierung der Funktionalität von DESMO in C++ gelungen. Die Klassenbibliothek wurde bereits im Rahmen einer Lehrveranstaltung erfolgreich eingesetzt, bei der die Studierenden Aufgaben mit Hilfe von DESMO-C umgesetzt haben. Die meisten der dabei auftretenden Schwierigkeiten waren eher auf Probleme mit der Sprache C++ zurückzuführen, als auf die simulationsspezifischen Konzepte. Dabei wurde von den Teilnehmern die in Kapitel 6 vorgestellte Referenz der Schnittstellenobjekte als große Hilfe empfunden, sich in der Klassenbibliothek zurechtzufinden.

Ereignis- und prozeßorieniterte Simulation müssen in DESMO-C nicht mehr strikt getrennt werden. Darüber hinaus konnte der Funktionsumfang erweitert werden. DESMO-C kann zur Entwicklung von Experimentierumgebungen eingestezt werden, da es sich gut mit anderen Klassenbibliotheken kombinieren läßt. Die neu eingeführte Klasse `Model` eignet sich dazu, einmal entworfene Modelle wiederzuverwenden und zu komplexeren zu kombinieren.

7.2 Laufzeitvergleich

DESMO-C wurde in erster Linie in Hinsicht auf die Anwendbarkeit konzipiert. Optimierungen in Bezug auf das Laufzeitverhalten konnten im Rahmen dieser Arbeit nicht umgesetzt werden. Dennoch zeigt sich bei einem ersten Laufzeitvergleich mit den Messungen aus [Schni96][66] ein äußerst positives Ergebnis. Abbildung 7-1 zeigt die gemessenen Ausführungszeiten der jeweiligen Modelle mit DESMO sowie mit DESMO-C auf einem Apple PowerMacintosh 6100/66. Dabei muß darauf hingewiesen werden, daß der in [Schni96] verwendete Compiler keinen für den Prozessor (PPC 601) optimierten Code erzeugte, was zu Ungunsten der DESMO Ergebnisse Buche schlägt.

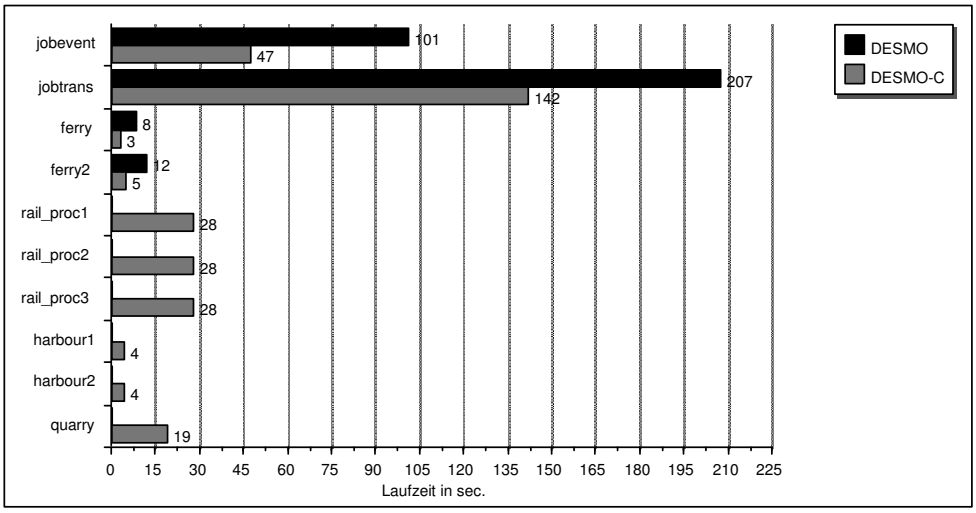

Abbildung 7-1: Laufzeiten der Beispielmodelle in DESMO-C, z.T. auch Vergleichswerte der DESMO-Versionen

66 Vgl. [Schni96], S. 51

7.3 Ausblick

Bei der Entwicklung der vorliegenden Version von DESMO-C war das Erreichen der Funktionalität von DESMO geboten. Die Entwicklung eines vollkommen neuen Simulationskerns, der eine gemeinsame Nutzung beider Weltbilder zuläßt, und die Anpassung der bestehenden Konzepte an diese neue Perspektive ließ diese Arbeit recht umfangreich werden. Daher konnten die angestrebten Konzepte für Experiment- und Modellverwaltung nur ansatzweise umgesetzt werden.

Insbes. bei der Entwicklung der Beispielmodelle fehlte eine Vorrichtung, um Modellparameter verwalten zu können. So wäre eine Klasse `Parameter` als Unterklasse von `Reportable` wünschenswert, die Mechanismen zum Einlesen, Ändern, Ausgeben und Sperren eines Parameters anbietet. Ein anderer wichtiger Punkt ist der Austausch von Informationen zwischen Modellen, der in der vorliegenden Version vollständig vom Modellentwickler zu leisten ist. Besonders wenn Entities von Modell zu Modell wandern, muß gewährleistet sein, daß die konzeptionelle Identität des Entities erhalten bleibt, auch wenn es in unterschiedlichen Modellen durch verschiedene Objekte repräsentiert wird. Hierfür könnte eine Klasse `Port` eingerichtet werden, über deren Objekte Modelle miteinander verbunden werden können. Dies würde eine Verwendung von Modellen auf einer höheren Ebene wesentlich erleichtern.

Eine Erweiterung von DESMO um kontinuierliche Simulation wurde in [Ritsc91] vorgestellt und in SiFrame nachvollzogen. Dabei wurden unterschiedliche Implementierungen verwendet, jenachdem mit welchem diskreten Simulationsstil der kontinuierliche Teil kombiniert wird. Für DESMO-C muß eine solche Unterscheidung nicht mehr getroffen werden, da die Weltbilder vereinbar sind. So kann bei einer zukünftigen Implementierung das günstigste Verfahren zum Einsatz kommen.

Der Kern von DESMO-C ist dafür ausgelegt, daß für die Ereignislistenverwaltung verschiedene Algorithmen zum Einsatz kommen können. Die Auswahl eines implementierten Algorithmus kann zur Laufzeit erfolgen. Damit ist die Entwicklung von Verfahren denkbar, die entsprechend einer Modellklassse einen günstigen Algorithmus auswählen, um so das Laufzeitverhalten des Simulators zu optimieren.

8 Literatur

[Alexa77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel: A Pattern Language. Oxford University Press, New York 1977

[Biele] Carsten Biele, Heiko Weber: Konzept für die Verbesserung der Wartungs- und Portierungsfreundlichkeit von DESMO. Studienarbeit, Fachbereich Informatik, Universität Hamburg

[Birtw79] G. M. Birtwistle: DEMOS – A System for Discrete Event Modeling On Simula. MacMillan, London 1979

[Bölck89] R. Bölckow; A. Heymann, R. Kandler, H. Liebert: Entwurf und Implementation eines Simulators für die Zeitdiskrete Simulation in Modula-2. Diplomarbeit, Fachbereich Informatik, Universität Hamburg 1989

[Booch95] Grady Booch: Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen. Bonn, Paris, Reading, Mass., [u.a.], Addison-Wesley, 1. korrigierter Nachdruck 1995

[Brock86] dtv-Brockhaus-Lexikon in 20 Bänden. Band 5. Deutscher Taschenbuch Verlag GmbH & Co. KG, München 1986

[DalCi89] M. Dal Cin, J. Lutz, T. Risse:Programmierung in Modula-2. Teubner, Stuttgart, 4. Aufl. 1989

[Eckel96] Bruce Eckel: In C++ denken. München, London, [u.a.], Prentice Hall 1996

[Ellis94] Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual. AT&T Bell Laboratories, Murray Hill, New Jersey, April 1994

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, Hohn Vlissides: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, Bonn, 1. Auflage 1996

[Khosh86] S. Khoshafian, G. Copeland: Object Identity. SIGPLAN Notices vol. 21(11), November 1986.

[Meyer90] Bertrand Meyer: Objektorientierte Softwareentwicklung. Wien: Hnaser; London: Prentice Hall Internat. 1990

[Meyer96] Scott Meyers: Effektiv C++ programmieren: 50 Möglichkeiten zur Verbesserung Ihrer Programme. Bonn; Paris; Reading, Mass. [u.a.], Addison-Wesley, 2. Auflage/1. korrigierter Nachdruck 1996

[Meyer98] Scott Meyers: Mehr effektiv C++ programmieren: 35 neue Wege zur Verbesserung Ihrer Programme und Entwürfe. Bonn; Paris [u.a.], Addison-Wesley, 1. Auflage 1998

[Page88] B. Page, R. Bölckow, A. Heymann, R. Kadler, H. Liebert: Simulation und moderne Programmiersprachen. Modula-2, C, Ada. Fachberichte Simulation 8, Springer, Berlin 1988

[Page91] Bernd Page: Diskrete Simulation. Eine Einfphrung mit Modula-2. Berlin; Heidelberg; New York; [u.a.]; Springer 1991

[Page94] Bernd Page, Lorenz M. Hilty: Umweltinformatik – Informatikmethoden für Umweltschutz und Umweltforschung. München; Wien; Oldenbourg 1994 (Handbuch der Informatik, Bd. 13.3)

[Ritsc91] Astrid Ritscher: Entwurf und Realisierung eines kontinuierlichen Simulationsmoduls für das Simulationspaket DESMO in Modula-2. Diplomarbeit, Fachbereich Informatik, Universität Hamburg 1991

[Russe] Edward C. Russel: Building Simulation Models with SIMSCRIPT II.5. CACI, Inc.-Federal Los Angeles, California 1983

[Schni96] Thomas Schniewind: Portierung von DESMO auf Metrowerks Modula-2 unter Macintosh System 7. Studienarbeit, Fachbereich Informatik, Universität Hamburg 1996

[Spani95] Otto Spaniol, Simon Hoff: Ereignisorinetierte Simulation: Konzepte und Systemrealisierung. Bonn, Internat. Thomson Publ., 1. Auflage 1995

[Strou92] Bjarne Stroustrup: Die C++-Programmiersprache. Bonn; München; Paris [u.a.], Addison-Weslay, 2. Auflage 1992

[Trush95] Andreas Trusheim: DESMO in Oberon-2. Diplomarbeit, Fachbereich Informatik, Universität Hamburg 1995

[Weber96] Heiko Weber: Entwurf und Implementation eines objektorientierten Simulationspaketes für die zeitdiskrete und kontinuierliche Simulation in C++. Diplomarbeit, Fachbereich Informatik, Universität Hamburg 1996

9 Verzeichnisse

9.1 Abkürzungen

Abb.	Abbildung
Aufl.	Auflage
bzw.	beziehungsweise
DEMOS	Discrete Event Modelling On Simula, ein von G. Birtwistle entwickelter Simulator in Simula
DESMO	Discrete Event Simulation in Modula-2, eine Modulsammlung für Modula-2 zur Entwicklung von Simulationsprogrammen
dgl.	dergleichen
d.h.	das heißt
ebd.	ebenda; ebendort
etc.	et cetera
evtl.	eventuell
exkl.	exklusive; ausschließlich
FIFO	Bedienstrategie: "First In First Out"
ggf.	gegebenenfalls
i. allg.	im allgemeinen
inkl.	inklusive; einschließlich
insbes.	insbesondere
lat.	lateinisch
lfd.	laufend
LIFO	Bedienstrategie: "Last In First Out"
m.a.W.	mit anderen Worten
Nr.	Nummer
o.a.	oben angegeben
o.ä.	oder ähnlich(es)
OOP	objektorientierte Porgrammierung bzw. Programmiersprache(n)
s.	siehe
s.a.	siehe auch
SiFrame	Simulation Framework, ein Rahmenwerk zur Entwicklung von Simulationsprogrammen in C++
s.o.	siehe oben
sog.	sogenannt
s.S.	siehe Seite
s.u.	siehe unten
u.a.	unter anderem
u.s.w.	und so weiter
u.U.	unter Umständen
u.v.m.	und vieles mehr
vgl.	vergleiche
w.o.	wie oben
z.B.	zum Beispiel
z.T.	zum Teil
ZZ-Strom	Zufallszahlenstrom
z.Zt.	zur Zeit