# Universität Stuttgart

Institut für Nachrichtenvermittlung und Datenverarbeitung
Prof. Dr.-Ing. habil P. J. Kühn

## 59. Bericht über verkehrstheoretische Arbeiten

# ENTWURF UND IMPLEMENTIERUNG EINER SIMULATIONSBIBLIOTHEK UNTER ANWENDUNG OBJEKTORIENTIERTER METHODEN

von

**Hartmut Kocher**

1994

Druck: E. Kurz & Co., Druckerei + Reprografie GmbH., Stuttgart

ISBN 3-922403-69-7

# University of Stuttgart

Institute of Communications Switching and Data Technics

Prof. Dr.-Ing. habil. P. J. Kühn

**59th Report on Studies in Congestion Theory**

# DESIGN AND IMPLEMENTATION OF A SIMULATION LIBRARY USING AN OBJECT-ORIENTED APPROACH

by

**Hartmut Kocher**

1994

# Summary

Today's systems are becoming more and more complex. The human mind is unable to comprehend complex systems in their entirety. Simulations are often the only viable way to verify the functionality of a system, or to estimate its performance.

Simulating a complex system is itself a complex task. Therefore, it is important to structure simulation software in a suitable way. The complexity of simulation programs can be reduced by decomposing the simulation model into a hierarchy of submodels that can be refined in further steps. Although many parts of a simulation model could potentially be reused across different applications, this is not supported very well in current simulation programs due to strong coupling between model components. Reuse of models and submodels would be an important step towards economic development of simulation programs that could be used to evaluate several design choices in a cost effective way. This is a general problem of software development and is one of the reasons for the so-called software crisis. The object-oriented paradigm promises to improve the situation dramatically. That's why more and more simulation libraries are written in an object-oriented programming language.

Unfortunately, most simulation libraries focus on implementation issues rather than using abstractions from the problem domain. There is a semantic gap between the low-level abstractions they offer, such as process classes, and the problem-oriented abstractions the user wants. Simulation libraries that are more problem oriented are just emerging. However, most systems don't offer support for hierarchical systems. This makes it difficult to implement reusable submodels and components that can be further refined as the design evolves. As already pointed out, these are essential features for simulating complex systems.

This report describes a flexible general purpose library for the simulation of complex hierarchical systems. The library is implemented in C++ and uses high-level abstractions that are closely related to the problem domain. Therefore, users can easily map their simulation models to actual simulation programs. The library can be easily extended and takes advantage of the latest additions to the C++ language, namely templates and the exception handling mechanism. It uses few basic abstractions and emphasizes a clean software architecture. The library supports hierarchical decomposition of simulation models into submodels and model components. Model components are strictly encapsulated and communicate with each other using a

handshake protocol. This offers the ability to highly reuse standardized model components and quickly create or modify a simulation model using a 'plug-and-play' approach.

**Chapter 1** gives a brief introduction and summarizes the goals of the performed research.

**Chapter 2** introduces object-oriented programming. After a short review of the history and the basic concepts of object-oriented programming, new approaches of developing object-oriented software are discussed. These methods are compared to conventional methods that are used for software development. It is followed by a discussion of the benefits of the object-oriented approach. The chapter closes with an introduction to the C++ programming language.

**Chapter 3** describes various approaches for implementing simulation programs. The principles of event driven simulations are presented. The potential benefits of applying object technology to the field of computer simulation are discussed. The simulation of complex systems needs many computer resources. Therefore, it is desirable to distribute large simulations and execute them on a number of computers in parallel. All common methods for distributing simulation runs across computers are presented. The feasibility for implementing them in a general purpose simulation library is evaluated. Finally, a number of known simulation environments are described. Their shortcomings are used as a starting point for developing the requirements of a good simulation library.

**Chapter 4** is the main part of the study. Starting from the design goals of the simulation library, the basic software architecture is shown. Although the main abstractions are discussed in detail, the focus is on describing possible alternative solutions. The importance of a clean software architecture building on a few well-chosen abstractions is emphasized.

**Chapter 5** discusses the application of the simulation library. A complex simulation model of a satellite switching system has been implemented using both a conventional approach and the object-oriented paradigm. A comparison of both programs shows the benefits of the object-oriented solution. By extending the simulation model, the superior maintainability of programs that use the simulation library is demonstrated. Several approaches for extending the library for distributed simulations are presented. They prove the flexibility of the underlying concepts. Finally, some general observations found during development are discussed.

**Chapter 6** closes the report with a summary of the most important results and an outlook of possible directions for future enhancements of the simulation library.

# Inhaltsverzeichnis

**Inhaltsverzeichnis** ..... **1**

**Verzeichnis der Abbildungen** ..... **5**

**1 Einleitung** ..... **7**

1.1 Simulation komplexer verteilter Systeme ..... 7

1.2 Zweck der Arbeit ..... 8

1.3 Übersicht über die Arbeit ..... 9

**2 Objektorientierte Programmierung** ..... **11**

2.1 Geschichte der objektorientierten Programmierung ..... 11

2.2 Grundlagen der objektorientierten Programmierung ..... 13

2.2.1 Modularisierung ..... 13

2.2.2 Datenabstraktion ..... 14

2.2.3 Klassen ..... 14

2.2.4 Vererbung ..... 15

2.2.5 Polymorphie ..... 17

2.2.6 Weitere Konzepte ..... 18

2.3 Objektorientierter Entwurf von Software ..... 18

2.3.1 Vorgehensweise ..... 19

2.3.2 Vergleich mit konventionellen Software-Entwurfsmethoden ..... 19

2.4 Vorteile der objektorientierten Programmierung ..... 20

2.5 Die Programmiersprache C++ ..... 22

2.5.1 Eigenschaften von C++ ..... 22

2.5.2 Unterstützung des objektorientierten Ansatzes durch C++ ..... 22

2.5.2.1 Datenkapselung ..... 22

2.5.2.2 Abstrakte Datentypen ..... 24

2.5.2.3 Vererbung ..... 25

2.5.2.4 Polymorphie ..... 26

2.5.2.5 Parametrisierte Klassen ..... 26

2.5.2.6 Ausnahmebehandlung ..... 26

**3 Die zeitdiskrete Simulation ..... 27**
3.1 Überblick über verschiedene Simulationsarten ..... 28
3.1.1 Die ereignisorientierte Simulation ..... 29
3.1.2 Die prozeßorientierte Simulation ..... 30
3.1.3 Die transaktionsorientierte Simulation ..... 31
3.1.4 Bewertung der Simulationsverfahren ..... 32
3.2 Grundlagen der ereignisorientierten Simulation ..... 33
3.2.1 Komponenten des Simulationsmodells ..... 33
3.2.2 Ablauf der Simulation ..... 34
3.3 Objektorientierte Simulation ..... 35
3.4 Parallele Simulation ..... 36
3.4.1 Gleichzeitiger Ablauf mehrerer Simulationen ..... 37
3.4.2 Parallelisierung von Teiltests ..... 37
3.4.3 Funktionale Aufteilung ..... 38
3.4.4 Zentrale Ereignisverwaltung ..... 38
3.4.5 Verteilte Ereignisverwaltung ..... 39
3.4.6 Modellierung paralleler Vorgänge ..... 40
3.4.7 Abschätzung des Implementierungsaufwandes für parallele Simulation ..... 40
3.5 Eigenschaften bekannter Simulationspakete ..... 42
3.5.1 Eigenständige Simulationsumgebungen ..... 42
3.5.2 Standard-Simulationsbibliotheken ..... 44
3.6 Bewertung, Kritik und Ansätze neuer Methoden ..... 46

**4 Eine objektorientierte Simulationsbibliothek ..... 48**
4.1 Ziele der Simulationsbibliothek ..... 48
4.1.1 Erhöhung der Produktivität ..... 48
4.1.2 Handhabung komplexer Systeme ..... 49
4.2 Objektorientierter Entwurf der Simulationsbibliothek ..... 50
4.2.1 Einfache Abstraktionen ..... 51
4.2.2 Abstrakte Basisklassen ..... 51
4.2.3 Beziehungen zwischen Klassen ..... 52
4.2.4 Klassenhierarchie ..... 52
4.2.5 Konsistenz ..... 53
4.3 Architektur ..... 54

4.4 Modellkomponenten ... 56
4.4.1 Eigenschaften ... 56
4.4.2 Hierarchische Modellkomponenten ... 58
4.4.3 Definition neuer Modellkomponenten ... 62
4.5 Verbindung von Modellkomponenten ... 64
4.5.1 Das Port-Konzept ... 64
4.5.2 Das Verbindungskonzept ... 65
4.5.3 Zusammenwirken von Modellkomponenten und Ports ... 67
4.5.4 Nachrichtenfilter ... 71
4.6 Ereignissteuerung ... 73
4.6.1 Ereignisse ... 73
4.6.2 Bearbeitung von Ereignissen ... 74
4.6.3 Verwaltung von Ereignissen ... 75
4.7 Die Simulationssteuerung ... 78
4.7.1 Aufgaben der Simulationssteuerung ... 78
4.7.2 Aktualisierung des Systemzustandes ... 79
4.7.3 Ablauf der Ereignissteuerung ... 82
4.7.4 Ausgabe der Ergebnisse ... 83
4.7.5 Einlesen von Simulationsparametern ... 83
4.8 Erzeugung von Zufallsverteilungen ... 84
4.8.1 Erzeugung von Zufallszahlen ... 84
4.8.2 Erzeugung von Zufallsverteilungen ... 86
4.8.3 Anwendung von Zufallsvariablen ... 87
4.9 Statistik ... 88
4.9.1 Statistikklassen ... 89
4.9.2 Statistische Aussagesicherheit ... 90
4.10 Messen am Modell ... 91
4.10.1 Zähler ... 92
4.10.2 Messung von Durchlaufzeiten ... 93
4.11 Ergebnisaufbereitung ... 93
4.11.1 Das Ausgabekonzept ... 94
4.12 Entwicklungsunterstützung ... 95

**5 Anwendung und Bewertung der Simulationsbibliothek ..... 97**
5.1 Simulation eines Satellitensystems ..... 97
5.1.1 Modular On-Board Switching System ..... 97
5.1.2 Simulationsmodell ..... 100
5.1.3 Entwurf des Simulationsprogrammes ..... 102
5.1.4 Vergleich mit konventioneller Implementierung ..... 104
5.1.4.1 Aufwand ..... 104
5.1.4.2 Allgemeine Erkenntnisse ..... 106
5.1.5 Erweiterung des Simulationsprogrammes ..... 109
5.1.5.1 Erweitertes Modell ..... 109
5.1.5.2 Änderung des Simulationsprogrammes ..... 111
5.1.5.3 Erkenntnisse ..... 113
5.2 Erweiterungen für parallele Simulation ..... 114
5.2.1 Parallelisierung von Teiltests ..... 114
5.2.2 Konservative Verfahren ..... 116
5.3 Umfang und Bewertung der Simulationsbibliothek ..... 117

**6 Zusammenfassung und Ausblick ..... 123**

**Literaturverzeichnis ..... 126**

**Anhang ..... 132**
A.1 Booch-Notation für OOD-Diagramme. ..... 132
A.1.1 Klassendiagramme ..... 132
A.1.2 Objektdiagramme ..... 133
A.2 Programmierrichtlinien ..... 135
A.2.1 Erzeugung und Vernichtung von Objekten ..... 135
A.2.2 Parameterübergabe ..... 135
A.2.3 Namenskonventionen ..... 137
A.2.3.1 Typvereinbarungen ..... 137
A.2.3.2 Funktionsnamen ..... 137
A.2.3.3 Variablennamen ..... 137
A.2.3.4 Zusammengesetzte Namen ..... 138

# Verzeichnis der Abbildungen


2.1 Stammbaum der wichtigsten objektorientierten Programmiersprachen ..... 12

2.2 Vergleich von Einfach- und Mehrfachvererbung ..... 16

3.1 Klassifikation von Modellen nach Art der Zustandsübergänge ..... 28

3.2 Einfaches Modell eines Netzknotens ..... 29

3.3 Ereignisse im Modell des Netzknotens ..... 30

4.1 Architektur der Simulationsbibliothek ..... 54

4.2 TEntity Klassendiagramm ..... 57

4.3 Beispiel eines Modells eines einfachen Warteschlangennetzes ..... 59

4.4 Aufteilung des Modells in hierarchische Modellkomponenten ..... 59

4.5 Darstellung des Modells als Objektdiagramm ..... 59

4.6 Handshake-Protokoll zum Nachrichtenaustausch zwischen Ports ..... 65

4.7 Austausch einer Nachricht zwischen zwei Modellkomponenten ..... 71

4.8 Nachrichtenaustausch mit installierten Nachrichtenfiltern ..... 72

4.9 Klassendiagramm des Port-Konzeptes ..... 73

4.10 Eintragen eines Ereignisses in den Kalender ..... 76

4.11 Hierarchische Ereignisbearbeitung ..... 77

4.12 Klassendiagramm der Ereignisverwaltung ..... 77

4.13 Klassendiagramm der Simulationssteuerung ..... 80

4.14 Übergang von der Warmlaufphase zum ersten Teiltest ..... 81

4.15 Klassendiagramm der Zufallszahlengeneratoren ..... 85

4.16 Klassendiagramm der Zufallsverteilungen ..... 86

4.17 Klassendiagramm der Statistikklassen ..... 89

4.18 Klassendiagramm für statistische Schätzverfahren ..... 91

4.19 Klassendiagramm der Standard-Meßgeräte ..... 92

5.1 Protokollarchitektur des Satellitensystems ..... 98

5.2 Signalisierszenario eines vollständigen Verbindungsauf- und -abbaues ..... 98

5.3 Blockschaltbild des Satellitensystems ..... 99

5.4 Simulationsmodell des Satellitensystems ..... 101

5.5 Vergleich der Beispielprogramme anhand von Programmzeilen und Anzahl der ausführbaren Statements ..... 105

5.6 Verhältnis zwischen Vereinbarungen und ausführbarem Code ..... 105

5.7 Vergleich der Anzahl von Funktionen bzw. Methoden und Anzahl Statements pro Funktion. ..... 106

5.8 Erweitertes Simulationsmodell mit Fehlerkorrektur auf der Uplink-Satellitenstrecke ..... 110

5.9 Anzahl Programmzeilen und Anzahl ausführbarer Statements für Basis-bibliothek, Standard-Modellkomponenten und Beispielprogramm ..... 118

5.10 Aufteilung der Programmzeilen nach Compileranweisungen, Deklarationen und Programmcode prozentual und absolut ..... 118

5.11 Vergleich der Anzahl der Klassen und Aufteilung in Basis- und abgeleitete Klassen. Außerdem Vergleich der Anzahl der mehrfach abgeleiteten und parametrisierten Klassen ..... 119

5.12 Vergleich von Anzahl Methoden pro Klasse und Anzahl der Statements pro Methode ..... 120

A.1 Klassendiagramme ..... 133

A.2 Objektdiagramme ..... 134

# Kapitel 1

# Einleitung

## 1.1 Simulation komplexer verteilter Systeme

Verteilte Systeme sind teilweise so komplex, daß sie in ihrer Gesamtheit nicht oder nur schwer zu verstehen sind. Entsprechend kompliziert ist auch der Entwurf solcher Systeme. Meist gibt es mehrere Alternativen, ein System zu strukturieren. Um die Komplexität eines Systems beherrschbar zu machen, wird es in einfachere Teilsysteme und Module aufgespalten. Jedes Teilsystem kann einzeln entwickelt, aufgebaut und getestet werden. Anschließend werden die Teilsysteme zu einem Gesamtsystem integriert. Diese Vorgehensweise kann hierarchisch über mehrere Ebenen fortgesetzt werden, bis die Komplexität eines Moduls überschaubar wird. Dennoch ist es oft schwierig, eine optimale Aufteilung zu finden. Häufig stehen für die Realisierung eines Teilsystems mehrere Alternativen zur Auswahl. Die Schwierigkeit liegt darin, die Vor- und Nachteile verschiedener Entwurfsalternativen zu bewerten und gegeneinander abzuwägen. Bei einfachen Systemen lassen sich Aussagen über Leistungsfähigkeit und Verhalten mit Hilfe analytischer Methoden treffen. Sobald die Komplexität jedoch ein gewisses Maß überschreitet, kommen diese Verfahren an ihre Grenze. Der Aufbau von Prototypen, um verschiedene Entwurfsalternativen auszuprobieren, ist häufig nicht durchführbar. Zum einen können Termin- und Kostengründe dagegen sprechen, zum anderen kann es aus Sicherheitsgründen zu gefährlich sein, ein reales System aufzubauen. Bei dem Versuch komplexe Zusammenhänge aufzudecken, kann es außerdem passieren, daß der Prototyp ähnlich kompliziert und aufwendig wird wie das reale System. In diesen Fällen kann ein Ausweg in der Simulation des Gesamtsystems oder von Teilen davon liegen. Mit Hilfe der Simulation lassen sich sowohl Erkenntnisse über die Funktionalität als auch über die Leistungsfähigkeit eines Systems gewinnen.

Bei der Simulation wird das System in einem mehr oder weniger detaillierten Modell nachgebildet. Dieses Modell wird anschließend von einem Computerprogramm simuliert. Es gibt verschiedene Arten der Computersimulation mit entsprechenden Vor- und Nachteilen. Dabei ist es oft nicht einfach, eine dem Problem angepaßte Methode zu finden. Die meisten Simulationsprogramme werden direkt in einer Standard-Programmiersprache wie Pascal oder C

geschrieben. Daneben gibt es fertige Simulationsumgebungen, die dem Anwender eine einfachere Umsetzung vom Simulationsmodell zum lauffähigen Programm versprechen. Allerdings sind diese Umgebungen oft auf einen speziellen Problembereich zugeschnitten und für viele Anforderungen zu unflexibel.

Die Simulation komplexer Systeme ist selbst ein sehr komplexes Problem. Es ist deshalb wichtig, Simulationsprogramme geeignet zu strukturieren. So wie sich ein Gesamtsystem in Teilsysteme zerlegen läßt, ist es sinnvoll, ein Simulationsmodell in hierarchische Teilmodelle aufzuspalten. In konventionellen Programmiersprachen geschriebene Simulationsprogramme und Simulationsumgebungen versagen in diesem wichtigen Bereich häufig.

Ein weiteres Problem stellt die Kosten-/Nutzen-Frage dar. Die Erstellung komplexer Simulationsprogramme ist sehr aufwendig und entsprechend teuer. Obwohl Teilmodelle in verwandten Anwendungen in ähnlicher Form auftreten, konnte dies mit bisherigen Ansätzen nur unzureichend genutzt werden. Die Wiederverwendbarkeit von Modellen und Teilen von Simulationsprogrammen wäre ein wichtiger Schritt zur rationellen Entwicklung neuer Simulationen und zur kostengünstigen Untersuchung mehrerer Entwurfsvarianten.

Das zuletzt angesprochene Problem ist nicht auf die Erstellung von Simulationsprogrammen beschränkt. Es handelt sich um ein allgemeines Problem der Softwareerstellung und ist ein Grund für die immer wieder zitierte Softwarekrise. Objektorientierte Ansätze versprechen einen Ausweg aus dieser Situation. Nach objektorientierten Grundsätzen erstellte Programme sollen einfacher zu entwerfen, programmieren und zu testen sein. Außerdem sollen sie änderungsfreundlicher, leichter wiederverwendbar und damit kostengünstiger in der Herstellung und Wartung sein. Deshalb kommen in letzter Zeit objektorientierte Technologien bei Entwurf und Programmierung von Software auf allen Gebieten verstärkt zum Einsatz. Um die Vorteile der objektorientierten Programmierung zu nutzen, werden Simulationsprogramme mit dieser Technologie erstellt. Die bisherigen Ansätze auf diesem Gebiet sind noch nicht in allen Punkten befriedigend. Es stellt sich die Frage, inwieweit die gestellten Anforderungen und Probleme durch den Einsatz modernster Softwaretechnologie gelöst werden können. Die vorliegende Arbeit versucht eine mögliche Antwort auf diese Frage zu finden.

## 1.2 Zweck der Arbeit

Im Rahmen dieser Arbeit soll untersucht werden, ob mit Hilfe modernster Softwaretechnologie eine Simulationsumgebung geschaffen werden kann, die die im letzten Abschnitt angesprochenen Probleme löst.

Es soll gezeigt werden, wie durch den Einsatz objektorientierter Methoden bei Entwurf und Programmierung eine sehr flexible Simulationsbibliothek entsteht, die leicht an unterschiedlichste Anforderungen angepaßt werden kann. Gleichzeitig soll anhand einer realen Anwendung überprüft werden, wie weit die Versprechungen objektorientierter Technologie in Wirklichkeit zutreffen.

Der Schwerpunkt der Arbeit liegt dabei nicht in der Vorstellung einer neuen Simulationsbibliothek, die für ein breites Feld von Anwendungen geeignet ist, sondern in der Beschreibung der Verfahren und Erkenntnisse, die beim Entwurf dieses umfangreichen Softwarepaketes gewonnen wurden. Insbesondere soll hervorgehoben werden, wie sich eine durchgehende Softwarearchitektur, die sich auf grundlegende Abstraktionen aus dem Problembereich stützt, auf das Verständnis, die Erstellung, Wartung und Wiederverwendbarkeit von Software auswirkt. Viele der gemachten Aussagen beziehen sich deshalb nicht nur auf die vorgestellte Simulationsbibliothek, sondern können allgemein auf objektorientierte Software angewandt werden.

Anhand eines Beispielprogrammes sollen die Vorteile einer objektorientierten Lösung gegenüber der konventionellen Vorgehensweise praktisch aufgezeigt werden. Insbesondere soll die bessere Strukturierung, Wiederverwendbarkeit und Erweiterung anhand konkreter Beispiele demonstriert werden.

## 1.3 Übersicht über die Arbeit

Bevor die Architektur der im Rahmen dieser Arbeit entstandenen Simulationsbibliothek vorgestellt werden kann, müssen zunächst die Grundlagen objektorientierter Softwaretechnologie und unterschiedliche Simulationsverfahren betrachtet werden.

Kapitel 2 beschäftigt sich mit der objektorientierten Programmierung. Nach einem kurzen Überblick über Geschichte und Grundlagen objektorientierter Programmierung wird die Vorgehensweise bei der Erstellung objektorientierter Software erläutert und mit den Methoden der konventionellen Softwareentwicklung verglichen. Es folgt eine kurze Zusammenfassung der Vorteile des objektorientierten Lösungsansatzes. Das Kapitel schließt mit einer Einführung in die Programmiersprache C++, die im Rahmen dieser Arbeit zur Erstellung der Simulationsbibliothek verwendet wurde.

Im Kapitel 3 werden zunächst einige Simulationsverfahren vorgestellt. Anschließend wird auf die Grundlagen der ereignisorientierten Simulation eingegangen. Es folgt eine Beschreibung der Vorteile, die ein Anwender vom Einsatz objektorientierter Technologien bei der Erstellung von Simulationen erwarten kann. Da komplexe Simulationen große Anforderungen an

die Ressourcen der Rechner stellen, besteht häufig der Wunsch, Simulationsprogramme auf mehreren Rechnern parallel auszuführen. Es werden alle gängigen Möglichkeiten zur parallelen Abarbeitung von Simulationsprogrammen vorgestellt und auf ihre Verwendbarkeit für allgemeine Simulationsbibliotheken untersucht. Eine Beschreibung einiger ausgewählter Ansätze für die Realisierung von bekannten Simulationsumgebungen liefert Stoff für eine anschließende Kritik, die die Anforderungen an eine gute Simulationsbibliothek herausarbeiten soll.

Kapitel 4 bildet den Kern der Arbeit und stellt die eigentliche Simulationsbibliothek vor. Dabei wird ausführlich auf die Ziele und die Architektur der Bibliothek eingegangen. Obwohl die einzelnen Abstraktionen detailliert vorgestellt werden, liegen Schwerpunkte immer wieder auf der Beschreibung möglicher Entwurfsalternativen und der Begründung für den gewählten Ansatz. Es wird vor allem auf die Bedeutung einer durchgängigen Architektur eingegangen, die das Zusammenwirken einzelner Abstraktionen unterstützt. Die Aufstellung und Einhaltung einheitlicher Regeln erleichtern dem Anwender das Verständnis und verbessern die Möglichkeiten zur Wiederverwendung einzelner Abstraktionen erheblich.

Das 5. Kapitel steht ganz im Zeichen der Anwendung. Anhand eines umfangreichen Beispiels, das sowohl in konventioneller als auch in objektorientierter Form vorliegt, werden die Vorteile der objektorientierten Lösung aufgezeigt. Eine nachträgliche Erweiterung des Beispielprogrammes demonstriert die leichte Wartbarkeit der mit Hilfe der Simulationsbibliothek erstellten Programme. Die Vorstellung einiger möglicher Ansätze zur Erweiterung der Simulationsbibliothek für parallele Simulationen stellt die Flexibilität der gewählten Lösung unter Beweis. Einige allgemeine Eindrücke, die auch generell auf objektorientierte Software übertragbar sind, runden das Kapitel ab.

Die Arbeit schließt in Kapitel 6 mit der Zusammenfassung der wichtigsten Ergebnisse und gibt einen Ausblick auf mögliche zukünftige Erweiterungen der behandelten Ansätze.

# Kapitel 2

# Objektorientierte Programmierung

## 2.1 Geschichte der objektorientierten Programmierung

Obwohl sich der objektorientierte Ansatz erst in letzter Zeit durchzusetzen beginnt, handelt es sich dabei um keine neue Entwicklung. Die Grundsätze der objektorientierten Programmierung reichen mehr als 25 Jahre zurück. Kristen Nygaard, Ole-Johan Dahl und Bjrn Myhrhaug entwickelten damals die Sprache Simula 67 [48, 49]. Interessanterweise wurde Simula für den gleichen Zweck entworfen, den auch diese Arbeit behandelt, nämlich der Unterstützung von Simulationsprogrammen. Sie entstand mit dem Ziel, Simulationssysteme besser abstrahieren und Gemeinsamkeiten zusammenfassen zu können. Eine erste Version, Simula 1, wurde unter dem Einfluß von ALGOL 60 entwickelt und um die heute als Grundlagen der objektorientierten Programmierung anerkannten Konzepte erweitert. Aus ihr entwickelte sich Simula 67, eine Sprache, die als Urvater aller objektorientierten Programmiersprachen gilt. Die meisten modernen objektorientierten Sprachen sind mehr oder weniger stark an Simula angelehnt. Unter diesem Einfluß wurde in den frühen 70er Jahren am Xerox Palo Alto Research Center unter Federführung von Alan Kay die bekannte Sprache Smalltalk entwickelt, die sich heute noch großer Beliebtheit erfreut. Sie ist eine rein objektorientierte Sprache. Alle Elemente, einschließlich Zahlen und Klassen, werden durch Objekte repräsentiert. Smalltalk ist nicht nur eine Sprache, sondern stellt gleichzeitig eine komplette Entwicklungsumgebung mit graphischer Benutzeroberfläche zur Verfügung [28, 31]. Die momentan erfolgreichste objektorientierte Sprache ist C++, eine Weiterentwicklung der weitverbreiteten Sprache C. Sie wurde Mitte der 80er Jahre von Bjarne Stroustrup entworfen [58]. Da sie die Grundlage für die hier vorgestellte Arbeit ist, wird sie in einem eigenen Unterkapitel näher betrachtet. Ende der 80er Jahre entwickelte Bertrand Meyer die eigenständige, moderne Sprache Eiffel, die auch aus der Simula-Tradition stammt [45]. Man unterscheidet objektbasierte und objektorientierte Sprachen, wobei erstere keinen Vererbungsmechanismus besitzen (vgl. Kap. 2.2.4). Die objektbasierte Sprache Ada wird zur Zeit um zusätzliche objektorientierte Eigenschaften erweitert [10, 61]. Andere Programmiersprachen wurden teilweise um objektorientierte Dialekte bereichert, so z.B. Pascal. Im Anhang von [8] sind die Eigenschaften der

wichtigsten objektorientierten Programmiersprachen zusammengefaßt. Bild 2.1 zeigt den Stammbaum dieser Sprachen.

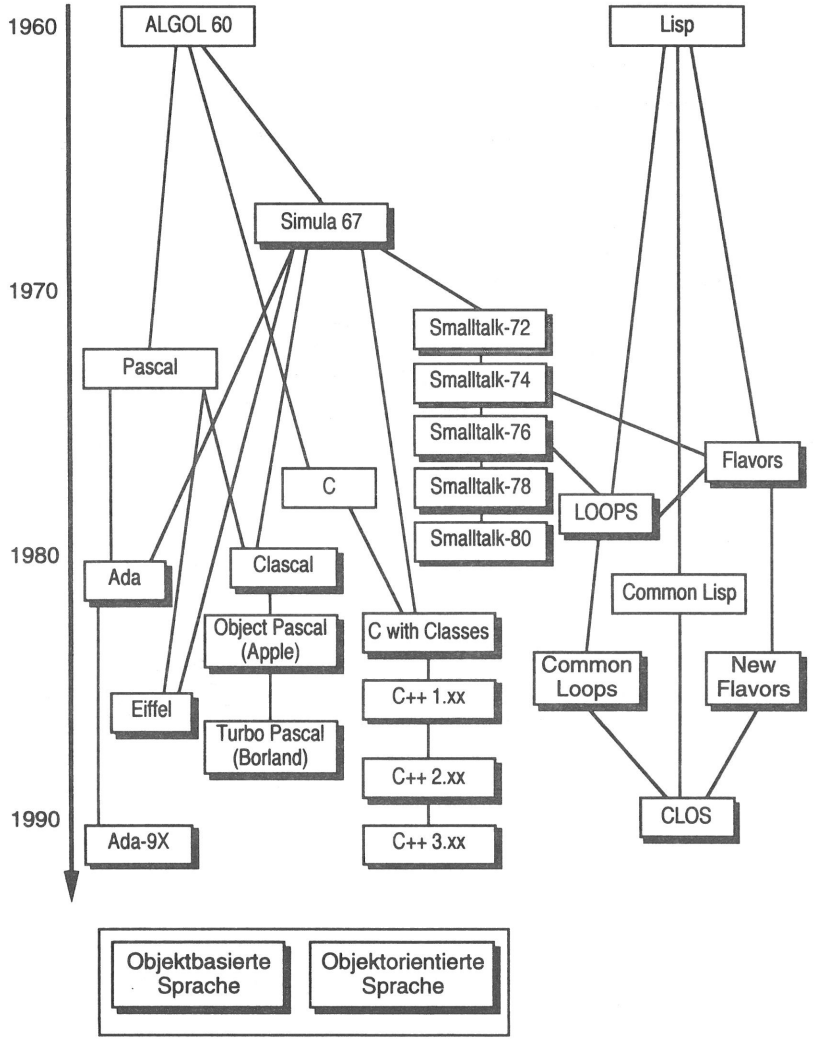



**Bild 2.1:** Stammbaum der wichtigsten objektorientierten Programmiersprachen (Erweiterung von [8], Seite 473)

## 2.2 Grundlagen der objektorientierten Programmierung

Der wichtigste Begriff der objektorientierten Programmierung ist das Objekt. Objekte haben einen lokalen Zustand, der für jedes Objekt verschieden sein kann. Außerdem besitzen sie eine festgelegte Anzahl von Operationen, die auf das Objekt angewandt werden können. Im Gegensatz zur prozeduralen Programmierweise besteht hier ein enger Verbund zwischen Daten und den Funktionen, die diese Daten manipulieren. Die Daten eines Objektes sind in der Regel von außen nicht zugänglich, sondern werden nur indirekt über Operationen verändert. Die Operationen eines Objektes definieren somit seine nach außen sichtbare Schnittstelle. Sie wird auch das Protokoll des Objektes genannt. Objekte kommunizieren miteinander durch den Austausch von Nachrichten. Beim Eintreffen einer Nachricht wird die entsprechende Operation des Objektes ausgelöst. Methode wird häufig als Synonym für Operation verwendet.

Unter objektorientierter Programmierung versteht man die Implementierung eines Programmes als eine Menge von Objekten, die miteinander kommunizieren.

Welche Anforderungen ein Ansatz erfüllen muß, um als „objektorientiert" zu gelten, wird vielfach noch diskutiert. Nach der Definition von Bertrand Meyer [46], müssen folgende Elemente bei einem objektorientierten Ansatz vorhanden sein:

- Modularisierung
- Datenabstraktion
- Klassen
- Vererbung
- Polymorphie

Die folgenden Abschnitte stellen diese Konzepte im einzelnen vor. Ein Einführungsartikel über objektorientierte Programmierung findet sich in [3]. Die unterschiedlichen Ansätze, von der prozeduralen bis zur objektorientierten Programmierung, werden in [57] anhand von Beispielen vorgestellt und miteinander verglichen.

### 2.2.1 Modularisierung

Ein wichtiger Schritt bei der Zerlegung eines Systems ist das Aufspalten in einzelne Module. Jedes Modul sollte die Realisierung einer bestimmten Aufgabe darstellen. Die Kopplung zwischen einzelnen Modulen sollte möglichst gering sein und ist ein Maß für die optimale Zerlegung des Systems. Die Schnittstellen eines Moduls definieren klare Grenzen innerhalb eines Programmes. Der Anwender eines Moduls kennt nur dessen Schnittstellen, die darun-

terliegende Implementierung bleibt ihm verborgen. Die Vorteile einer solchen Unterteilung sind bekannt. Modulkonzepte wurden bereits in mehreren Programmiersprachen realisiert, so z.B. in Modula-2. Beim objektorientierten Ansatz stellt das Objekt die kleinste Einheit der Modularisierung dar. Es ist darüber hinaus sinnvoll, Objekte, die zum selben Problembereich gehören, zusätzlich in herkömmliche Module einzubetten.

### 2.2.2 Datenabstraktion

Eine Weiterführung des Modulgedankens ist die Datenabstraktion. Bei ihr werden Datenstrukturen nur durch ihre nach außen sichtbaren Operationen definiert. Daten und Operationen werden dabei als eine Einheit betrachtet. Die internen Abläufe und Datenstrukturen sollen dem Anwender verborgen bleiben. Auf diese Weise lassen sich benutzerdefinierte Datentypen realisieren. Idealerweise sollten sich selbstdefinierte Datentypen in ihrer Anwendung von eingebauten Datentypen nicht unterscheiden. Dazu ist es notwendig, daß sich die in der Programmiersprache vorhandenen Operationen (wie +, -, *, ...) für eigene Datentypen neu definieren lassen. Dieses Konzept wird „Operator Overloading" genannt und ist in Sprachen wie Ada oder C++ realisiert.

### 2.2.3 Klassen

Da viele Objekte ähnliche oder identische Eigenschaften aufweisen, ist es sinnvoll, gleichartige Objekte zu Klassen zusammenzufassen. Eine Klasse beschreibt eine Anzahl von Objekten mit gleichem Aufbau und gleichem Verhalten. Ein einzelnes Objekt stellt eine Instanz einer Klasse dar. Obwohl alle Objekte einer Klasse die gleiche Datenstruktur besitzen, kann der Inhalt dieser Struktur und damit der interne Zustand eines Objektes für jedes Objekt verschieden sein. Als Beispiel kann die Abstraktion eines Fahrzeuges herangezogen werden. Die Klasse *Fahrzeug* könnte z.B. Datenfelder zur Speicherung des Namens des Besitzers, des Modells, des Alters, etc. definieren. Entsprechende Operationen würden das Verändern dieser Werte erlauben. Obwohl der Aufbau und das Verhalten für alle Objekte der Klasse *Fahrzeug* gleich sind, hängt das Ergebnis der Operationen, z.B. die Berechnung der Kfz-Steuer, vom Zustand des einzelnen Objektes ab.

In [44] beschreibt Meyer die objektorientierte Programmierung als eine Form der Auftragsabwicklung. Das Problem wird in kleine Teilprobleme zerlegt und diese werden durch Auftragsvergabe an Objekte gelöst. Das einzelne Objekt hat dabei eine Aufgabe innerhalb des Systems zu lösen. Zwischen Anwender und Klasse kommt ein Vertrag zustande, der die Aufgaben und das Verhalten aller ihr zugeordneten Objekte definiert. Die Schnittstelle einer Klasse beschreibt alle Operationen, die von Objekten der Klasse zur Verfügung gestellt werden. Wie diese Dienste erbracht werden, d.h. der interne Aufbau der Klasse sowie der

genaue Ablauf der Operationen, bleibt dem Anwender verborgen. Auf diese Weise werden die innere und äußere Sicht einer Klasse unterschieden und die Regeln der Datenabstraktion erfüllt.

Eine Klasse kann selbst ein Objekt mit eigenen Daten und Operationen sein. Eine typische Klassenoperation wäre z.B. die Erzeugung von Objekten dieser Klasse oder das Abrufen der Anzahl erzeugter Objekte. In den Klassenoperationen werden normalerweise die Gemeinsamkeiten aller Objekte dieser Klasse zusammengefaßt. Diese Klassen werden oft Metaklassen genannt.

Zwischen Klassen kann es unterschiedliche Beziehungen geben. Die wichtigste Beziehung stellt die Vererbung von Klassen dar. Sie wird im nächsten Abschnitt gesondert betrachtet. Eine weitere Beziehung ist die „Benutzen"-Beziehung. Um die Aufgaben einer Klasse zu erfüllen, werden oft Teilaufgaben an andere Klassen weitergegeben, d.h. sie werden zur Lösung von Problemen benutzt. Dies entspricht der bereits angesprochenen Auftragsvergabe. Klassen können auch andere Klassen erzeugen. Parametrisierte Klassen sind allgemeine Klassen, die erst durch Angabe von Parametern zu speziellen Klassen werden. Aus einer allgemeinen Listenklasse lassen sich auf diese Weise Listen von Fahrzeugen, Listen von Zahlen, Namen, etc. erzeugen. Ada unterstützt dieses Konzept durch „Generic Packages", in C++ gibt es dafür den „Template"-Mechanismus.

### 2.2.4 Vererbung

Das Vererbungskonzept ist vielleicht das wichtigste Element der objektorientierten Programmierung. Es erlaubt, Hierarchien von Klassen aufzubauen. Bei der Festlegung der Klassen *PKW* und *Bus* stellt sich schnell heraus, daß es viele Gemeinsamkeiten zwischen beiden Klassen gibt. Es bietet sich deshalb an, diese Gemeinsamkeiten in einer Oberklasse *Fahrzeug* zusammenzufassen. Von einer Oberklasse abgeleitete Klassen „erben" alle Daten und Operationen der Oberklasse. Zusätzlich ist es möglich, in der abgeleiteten Klasse neue Datenfelder und neue Operationen hinzuzufügen. Eine Klasse *Bus* könnte z.B. ein Datenfeld mit der Anzahl der Sitzplätze enthalten. Außer dem Hinzufügen neuer Operationen ist es auch möglich, geerbte Operationen neu zu definieren. Dies erlaubt das Anpassen von Operationen an die jeweilige Unterklasse. So könnte z.B. die Methode zur Ermittlung der Kfz-Steuer bei Bussen und PKWs verschieden sein.

Auf diese Weise müssen gemeinsame Merkmale von Klassen nur einmal, nämlich in der Oberklasse, definiert werden. In den Unterklassen werden nur die Änderungen und Erweiterungen spezifiziert. Unterklassen stellen somit im allgemeinen Spezialisierungen der Oberklassen dar. Spezialisierungen im Problembereich lassen sich dadurch sehr einfach in der Programmiersprache ausdrücken. Diese Art der Programmierung wird auch „Programming-

by-Difference“ genannt. Sie ist kompakt und nicht redundant, da gemeinsame Merkmale nur einmal kodiert werden. Außerdem ist sie änderungsfreundlich, da durch Vererbung einfach neue Klassen mit unterschiedlichen Eigenschaften entstehen können, ohne daß an vorhandenen Klassen etwas geändert werden muß.

Vererbung dient gleichzeitig dazu, Ähnlichkeiten zwischen Klassen auszudrücken. Ein Bus **ist** gleichzeitig ein Fahrzeug, d.h. alle Operationen, die auf ein Fahrzeug anwendbar sind, können auch auf einen Bus angewandt werden. Dies kann die Programmierung stark vereinfachen, da Oberklassen und abgeleitete Klassen im Programm gleich behandelt werden können. Der Name des Besitzers kann z.B. von Fahrzeugen, Bussen und PKWs auf dieselbe Art erfragt werden.

Man unterscheidet verschiedene Arten der Vererbung. Bei der einfachen Vererbung (*single inheritance*) kann jede Klasse maximal eine Oberklasse haben. Die Klassenhierarchie eines Systems besteht in diesem Fall aus einem Wald mit verzweigten Bäumen. Die Sprache Smalltalk fordert darüber hinaus, daß jede Klasse, mit Ausnahme der Wurzelklasse *Object*, genau eine Oberklasse haben muß. Es entsteht eine Baumstruktur mit *Object* als Wurzelelement. Kann eine Klasse gleichzeitig mehrere Oberklassen haben, so spricht man von Mehrfachvererbung (*multiple inheritance*). Dies erlaubt es, mehrere Beziehungen gleichzeitig auszudrücken, so ist ein Bus z.B. gleichzeitig ein Fahrzeug und ein öffentliches Verkehrsmittel. Mit Hilfe der Mehrfachvererbung kann dieser Sachverhalt auf natürliche Weise in kompakter Form ausgedrückt werden. Die Klassenhierarchie eines Systems läßt sich durch einen gerichteten Graphen, dessen Kanten die Vererbungsrelationen darstellen, beschreiben, der normalerweise keine Zyklen enthalten darf. Bild 2.2 zeigt je ein Beispiel für einfache und mehrfache Vererbung.

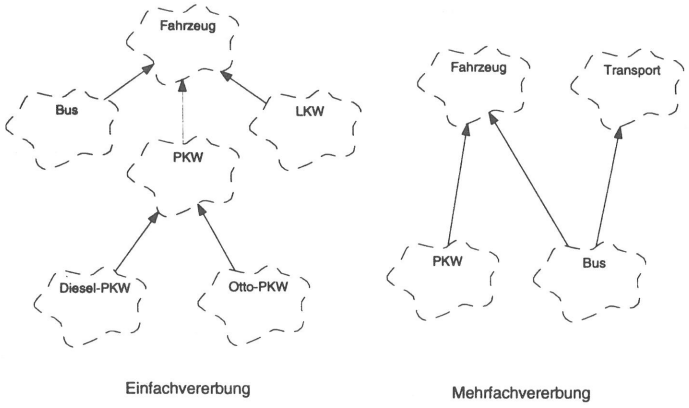



**Bild 2.2:** Vergleich von Einfach- und Mehrfachvererbung

Bei Mehrfachvererbung kann es zu Konflikten kommen, wenn eine Operation mit dem gleichen Namen von mehreren Oberklassen gleichzeitig geerbt wird. Für diesem Fall muß die Programmiersprache eine Regel definieren, welcher Operation der Vorzug zu geben ist.

Sprachen wie Ada, die nur Modulkonzepte und Datenabstraktion unterstützen, werden als objektbasiert bezeichnet. Erst das Vorhandensein eines Vererbungskonzeptes macht aus einer objektbasierten eine objektorientierte Sprache.

### 2.2.5 Polymorphie

Das Vererbungskonzept erlaubt es, geerbte Operationen in abgeleiteten Klassen neu zu definieren. Außerdem ist es möglich, abgeleitete Klassen anstelle der Oberklasse zu verwenden. Im obigen Beispiel könnte man für Busse, PKWs und LKWs jeweils eigene Methoden zur Berechnung der Kfz-Steuer definieren. Das Finanzamt kann dann eine Liste mit Fahrzeugen unterhalten, in die auch die von *Fahrzeug* abgeleiteten Objekte eingetragen werden können. Um die Summe der Steuereinnahmen zu errechnen, müßte man durch Senden einer Nachricht die Operation „Berechne-Kfz-Steuer" auf alle Objekte in der Liste anwenden und die Ergebnisse aufsummieren. Da es sich aus Sicht des Programmes um eine Liste von Fahrzeugen handelt, würde bei statischer Bindung von Nachrichten jeweils die in der Klasse *Fahrzeug* definierte Version der Operation ausgelöst werden, was nicht zum gewünschten Erfolg führen würde. Unter Polymorphie versteht man die dynamische Bindung von Nachrichten an Objekte. Beim Eintreffen einer Nachricht wird in Abhängigkeit des Objekttyps die richtige Operation gefunden und ausgeführt. Im Beispiel würde dies dazu führen, daß je nachdem, ob es sich um einen PKW, LKW oder Bus handelt, die passende Methode zur Steuerberechnung automatisch gefunden wird.

Dies ist ein wesentliches Konzept zur Vereinheitlichung und leichten Erweiterbarkeit objektorientierter Software. Der Programmteil zur Berechnung der Steuereinnahmen kann ohne Wissen der einzelnen Unterklassen, nur mit Kenntnis der Oberklasse *Fahrzeug*, geschrieben werden. Die unterschiedlichen Methoden zur Steuerberechnung für verschiedene Fahrzeugtypen bleiben in den Unterklassen verborgen. Selbst bei Einführung einer neuen Berechnungsmethode, z.B. für PKWs mit Katalysator, muß das Hauptprogramm nicht geändert werden. Es wird nur von *PKW* eine neue Klasse *PKW-mit-Katalysator* abgeleitet, die die neue Berechnungsmethode enthält. Sobald Objekte dieser Klasse in die Liste aller Fahrzeuge eingetragen werden, wird automatisch die neue Methode zur Steuerberechnung angewandt.

### 2.2.6 Weitere Konzepte

Es gibt objektorientierte Sprachen, die vom hier vorgestellten Modell abweichen, indem sie kein Klassenkonzept besitzen. An die Stelle von Klassen treten Prototypen. Im Gegensatz zu Klassen, die die gemeinsamen Merkmale ihrer Elemente definieren, ist ein Prototyp ein spezielles, repräsentatives Element dieser Menge. Andere Objekte können durch Delegation auf diesen Prototyp Bezug nehmen. Unter Delegation wird das Weiterreichen von Nachrichten, die das Objekt nicht selbst versteht, an den jeweiligen Prototyp verstanden. Alle Objekte, die der gleichen Klasse angehören würden, delegieren ihre Nachrichten zum selben Prototyp. Vererbung kann als eine spezielle Form der Delegation betrachtet werden. Kann eine Klasse eine bestimmte Nachricht nicht verstehen, so reicht sie diese an eine ihrer Oberklassen zur Bearbeitung weiter. Vertreter dieser Philosophie sind vor allem die sog. Actor-Sprachen.

Die meisten objektorientierten Sprachen besitzen keine Nebenläufigkeit. Alle Aktionen werden streng sequentiell ausgeführt. Objekte können jedoch auch als nebenläufige Einheiten gesehen werden, die sich durch das Senden von Nachrichten gegenseitig synchronisieren. Diese Idee wurde in den Actor-Sprachen und verschiedenen anderen Dialekten objektorientierter Sprachen, z.B. Concurrent Smalltalk, realisiert.

Auf beide Konzepte soll an dieser Stelle nicht weiter eingegangen werden, da sie zur Durchführung dieser Arbeit nicht relevant sind.

## 2.3 Objektorientierter Entwurf von Software

Während es objektorientierte Programmiersprachen schon einige Zeit gibt, ist die Methodik des objektorientierten Softwareentwurfs ein relativ neues Feld. Entsprechend groß ist die Vielzahl der unterschiedlichen Ansätze, und es wird noch einige Zeit dauern, bis diese konvergieren. Dennoch zeichnen sich bereits einige Gemeinsamkeiten ab, die im folgenden kurz vorgestellt werden sollen. In [23] werden verschiedene objektorientierte Entwurfsmethoden mit herkömmlichen Methoden verglichen. [17] vergleicht mehrere objektorientierte Methoden miteinander.

Der Entwurf von Software gliedert sich normalerweise in zwei große Phasen, der Analyse und dem Entwurf. In der Analysephase werden die Anforderungen aus dem Problembereich identifiziert und zu einer Spezifikation zusammengefaßt. Nach der Analyse erhält man ein Modell des realen Systems, das sämtliche wichtigen Aspekte des Problembereiches nachbildet. In der anschließenden Entwurfsphase wird eine abstrakte Lösung unter Berücksichtigung der gegebenen Randbedingungen, wie z.B. Kosten, Laufzeit, Qualität, etc. gesucht. Diese Lösung wird anschließend im realen System implementiert.

Beim objektorientierten Ansatz wird zwischen Analyse (**O**bject-**O**riented **A**nalysis OOA) und Entwurf (**O**bject-**O**riented **D**esign OOD) von Softwaresystemen unterschieden. Erfolgt die Implementierung der Software ebenfalls objektorientiert, so spricht man von objektorientierter Programmierung (OOP).

### 2.3.1 Vorgehensweise

Grady Booch faßt die einzelnen Schritte beim Entwurf objektorientierter Software wie folgt zusammen [8]:

Zunächst werden die wichtigen Abstraktionen des Problembereiches gesucht und in Klassen und Objekte zusammengefaßt. Anschließend wird die Bedeutung dieser Abstraktionen festgelegt, z.B. die Aufgaben der einzelnen Klassen und Objekte. Beziehungen zwischen Objekten oder zwischen Klassen werden festgehalten und können zum Aufbau einer Klassenhierarchie herangezogen werden. In nachfolgenden Schritten wird dieser Vorgang weiter verfeinert, d.h. es wird versucht, die Aufgaben einer Klasse so weit aufzuschlüsseln, bis sie mit Hilfe einfacher neuer Klassen oder bereits vorhandener Klassen aus einer Bibliothek erledigt werden können. Der ganze Vorgang ist ein iterativer Prozeß, da sich neu gefundene Abstraktionen durchaus auf bekannte Beziehungen zwischen Klassen auswirken können.

Bei der objektorientierten Analyse wird versucht, das Verhalten der wichtigen Abstraktionen aus dem Problembereich zu definieren. Es wird hauptsächlich auf die Schnittstellen und Operationen der Klassen geachtet. Beim anschließenden Design ist die Vorgehensweise ähnlich, jedoch wird jetzt mehr Wert auf die innere Sicht der Klassen, also deren Implementierung gelegt.

Der Übergang zwischen Analyse und Design ist bei der objektorientierten Entwurfsmethodik fließend. Deshalb eignet sie sich besonders für Anwendungen mit „Rapid Prototyping". Da die wichtigsten Abstraktionen meist früh in der Analysephase gefunden werden und diese, dank der Nähe zum Problembereich, sehr stabil bleiben, kann schnell ein lauffähiger Prototyp entwickelt werden, der mit der Zeit weiter verfeinert und mit neuer Funktionalität ausgestattet wird.

### 2.3.2 Vergleich mit konventionellen Software-Entwurfsmethoden

Bei der strukturierten Analyse (Structured Analysis, SA, z.B. DeMarco [18]) erfolgt die Zerlegung des Systems prozeßorientiert, d.h. es wird hauptsächlich Wert auf die durchzuführenden Operationen gelegt. Information Engineering (IE, z.B. Martin [40]) legt in der Analysephase mehr Wert auf die Datenflüsse, da es sich gezeigt hat, daß diese wesentlich stabiler sind als eine funktionelle Aufteilung. Während sich SA und OOA fundamental

unterscheiden, kann OOA als eine Erweiterung des IE aufgefaßt werden, bei der gleichzeitig Daten und Operationen betrachtet werden.

Sowohl der strukturierte Entwurf (Structured Design, SD, z.B. Yourdon [66]) als auch IE unterscheiden sich in der anschließenden Entwurfsphase sehr stark vom objektorientierten Entwurf (OOD). In herkömmlichen Methoden werden die Anforderungen aus der Analysephase in eine Hierarchie von Funktionen heruntergebrochen, d.h. die Entwurfsphase ist funktionsorientiert. Dies führt dazu, daß zwischen Analyse und Design eine Kluft besteht, die durch eine Transformation überbrückt werden muß. Diese Transformation ist nicht zu automatisieren und kann erhebliche Anstrengungen erfordern. Im Gegensatz dazu beschäftigt sich das OOD mit den Beziehungen zwischen Klassen und Objekten. Es ist eine natürliche Fortführung der Analysephase, nur wird beim Design der Schwerpunkt auf die internen Abläufe gelegt, während die Analyse mehr Wert auf die problemnahen Zusammenhänge zwischen Klassen legt.

Während bei der herkömmlichen Methodik idealerweise von einem schrittweisen Durchlaufen der einzelnen Phasen von der Analyse bis hin zu Kodierung, Test und Integration ausgegangen wird (sog. Wasserfallmodell), geht der objektorientierte Ansatz mehr von einer iterativen Vorgehensweise aus. Nach einer groben Analyse folgt der Entwurf und eine erste Implementierung. Im Laufe der Zeit wird das System weiter verfeinert, um neue Funktionalität erweitert und alle Schritte auf tieferer Ebene erneut durchlaufen. Dies erlaubt nicht nur die Erstellung von Prototypen, sondern bezieht alle Phasen eines Softwareproduktes, inklusive Wartung und Weiterentwicklung, mit ein. Letztere sind nur neue Zyklen im Rahmen eines iterativen Entwurfes.

## 2.4 Vorteile der objektorientierten Programmierung

Wie die letzten Kapitel teilweise schon gezeigt haben, ergeben sich aus den Möglichkeiten der objektorientierten Programmierung und des objektorientierten Entwurfs von Software erhebliche Vorteile gegenüber herkömmlichen Methoden. Die Hauptvorteile sollen im folgenden nochmals kurz zusammengefaßt werden.

Durch die relativ direkte Abbildung von realen Objekten des Problembereiches in Objekte innerhalb der Software entsteht ein direkter Bezug zwischen Realität und Modell. Die gefundenen Abstraktionen sind deshalb meist sehr stabil und ändern sich selbst bei nachträglichen Erweiterungen oft nur geringfügig.

Die strenge Kapselung von Daten und Operationen schafft klar umrissene Abstraktionen, die in sich geschlossen sind. Dies wiederum erleichtert die Wiederverwendbarkeit der Abstrak-

tionen in neuen Anwendungen. Mehrere Arten der Wiederverwendung von Software können unterschieden werden. Die Wiederverwendung eines Entwurfes erlaubt es, Abstraktionen, die für einen bestimmten Bereich gefunden wurden, auf ähnliche Aufgaben zu übertragen. Selbst wenn nur die Schnittstellen und Operationen einer Abstraktion übernommen werden können, kann dadurch viel Zeit gespart werden, da der Kodieraufwand im Verhältnis zum Entwurfsaufwand gering ist. Vererbung erlaubt eine andere Form der Wiederverwendbarkeit. Die abgeleitete Klasse benutzt die Schnittstellen und die Implementierung der Oberklasse. Nur Änderungen müssen neu programmiert werden. Damit kann sowohl Code als auch Aufwand beim Entwurf gespart werden. Parametrisierte Klassen erlauben die Wiederverwendung von Source-Code für verschiedene Anwendungen. Wiederverwendbarkeit bekommt man allerdings durch einen objektorientierten Entwurf nicht automatisch geschenkt. Der objektorientierte Ansatz erlaubt es nur, wiederverwendbare Komponenten leichter zu identifizieren. Trotzdem muß Wiederverwendbarkeit von Softwarekomponenten bereits beim Entwurf berücksichtigt werden.

Vererbung und Polymorphie vereinfachen die einheitliche Behandlung unterschiedlicher Objekte. Sie fördern damit den Entwurf generischer Softwaremodule und erleichtern die Erweiterbarkeit bestehender Software. Änderungen und Erweiterungen führen oft nur zur Definition neuer Unterklassen. Operationen, die diese Objekte manipulieren, sind davon in vielen Fällen nicht betroffen.

Die strikte Kapselung von Objekten ermöglicht nicht nur die bessere Wiederverwendbarkeit, sondern sorgt dafür, daß sich Änderungen der Implementierung häufig nur lokal auswirken. Auch der Test von Objekten wird durch die klar umrissene Aufgabenstellung erleichtert.

Der iterative Ansatz beim Entwurf objektorientierter Software ist optimal auf die Anforderungen langlebiger Softwaresysteme, die häufig geändert und ergänzt werden, angepaßt. Er bietet eine durchgängige Methodik vom ersten Prototypen bis zur Wartung und Erweiterung von Softwaresystemen. Er ist damit viel besser an die Erfordernisse heutiger komplexer Softwaresysteme angepaßt als das bisherige Wasserfallmodell, bei dem alle Phasen möglichst sequentiell durchlaufen werden.

Insgesamt verspricht der objektorientierte Ansatz Vorteile in allen wichtigen Bereichen der Softwareentwicklung.

## 2.5 Die Programmiersprache C++

C++ ist eine noch relativ junge Programmiersprache, deren Definition teilweise immer noch im Fluß ist. Trotzdem ist sie bereits heute die meistverwendetste objektorientierte Sprache. Sie wurde Anfang der 80er Jahre von Bjarne Stroustrup in den AT&T Bell Laboratories entworfen. Mittlerweile hat sie einige signifikante Änderungen erfahren. Ein Komitee arbeitet seit einiger Zeit an der Standardisierung der Sprache. In letzter Zeit wurden einige wichtige Erweiterungen definiert, die noch nicht in allen Compilern umgesetzt wurden. Zur Einführung in die Sprache empfiehlt sich [37]. Eine eingehende Betrachtung findet sich in [59]. Die genaue Sprachdefinition kann [21] entnommen werden. Dieses Werk ist gleichzeitig das Basisdokument für die Standardisierung der Sprache.

### 2.5.1 Eigenschaften von C++

Der große Erfolg von C++ rührt von der Kompatibilität mit der Programmiersprache C her. C++ ist keine rein objektorientierte Sprache wie Smalltalk, sondern unterstützt unterschiedliche Ansätze, vom prozeduralen bis zum objektorientierten Programmieren. Daher erlaubt sie ihren Anwendern einen fließenden Übergang von gewohnten Programmieransätzen, die sie aus der Sprache C kennen, zum objektorientierten Paradigma. Aufgrund der strengeren Typüberprüfung lohnt sich der Umstieg von C auf C++ sogar für diejenigen, die die objektorientierten Erweiterungen gar nicht nutzen wollen. Da außer normalen Compilern auch ein Übersetzer von C++ nach C zur Verfügung steht, ist die Sprache auf fast allen Plattformen verfügbar. Bei allen objektorientierten Erweiterungen wurde auf Effizienz bei der Implementierung geachtet, so daß das Laufzeitverhalten im Vergleich zu C ähnlich gut ist.

Weitere Gründe für die Wahl von C++ als Implementierungssprache der hier vorgestellten Simulationsbibliothek werden in Kapitel 4.2 gegeben.

Im folgenden soll ein kurzer Überblick über die Erweiterungen der Sprache C++ gegenüber C gegeben werden. Die Kenntnis der Sprache C wird dabei vorausgesetzt. Natürlich können an dieser Stelle nicht alle Details der Sprache behandelt werden. Für eine ausführliche Diskussion wird auf die bereits zitierte Literatur verwiesen.

### 2.5.2 Unterstützung des objektorientierten Ansatzes durch C++

#### 2.5.2.1 Datenkapselung

Zur Kapselung von Daten stellt C++ ein Klassenkonzept zur Verfügung. In einer Klasse können Daten und Operationen zusammengefaßt werden. Ein abgestuftes Konzept von

Zugriffsberechtigungen ermöglicht die strikte Trennung von Schnittstelle und Implementierung. Verletzungen der Zugriffsberechtigung werden vom Compiler erkannt und gemeldet. Der folgende Codeausschnitt zeigt die Deklaration einer Klasse *Fahrzeug* sowie die Definitionen der Operationen *SetzeBesitzerName* und *BerechneKfzSteuer*.

```
// Deklarationen
class Fahrzeug {
public:
    // Öffentlich zugängliche Operationen und Daten
    void SetzeBesitzerName(char * name);
    float BerechneKfzSteuer();
...

private:
    // Nur innerhalb der Klasse zugängliche Operationen und Daten
    static float GetSteuersatz();  // Klassenmethode

    char *   fBesitzerName;
    int      fHubraum;
};


// Definitionen
void Fahrzeug::SetzeBesitzerName(char * name)
{
    fName = strdup(name);       // Kopie des Namens anfertigen
}

float Fahrzeug::BerechneKfzSteuer()
{
    return fHubraum * GetSteuersatz();
}
```

Deklaration und Definition erfolgen normalerweise in getrennten Dateien, so daß Anwender nur die Deklaration zu sehen bekommen. Die eigentliche Implementierung bleibt verborgen. Getrennte Dateien sind die einzige Möglichkeit, zusammenhängende Klassen in einem Modul zusammenzufassen. Ein echtes Modulkonzept wird nicht unterstützt. Klassen können geschachtelt innerhalb anderer Klassen definiert werden. Damit kann die globale Sichtbarkeit von Klassen kontrolliert werden.

Operationen und Daten werden in C++ *members* genannt. Die sog. *member functions* heißen auch Methoden. Klassenmethoden oder -daten werden in der Klassendeklaration durch das Schlüsselwort *static* gekennzeichnet. Sie sind dann nicht in jedem Objekt vorhanden, sondern nur einmal für die ganze Klasse.

Innerhalb des eigenen Objektes kann der *this*-Zeiger verwendet werden, um auf die eigene Instanz zu verweisen, z.B. würde innerhalb von *BerechneKfzSteuer* sowohl *fHubraum* als auch *this->fHubraum* jeweils dasselbe Datum bezeichnen.

Das Senden von Nachrichten an Objekte erfolgt durch den Aufruf der jeweiligen Methoden:

```
Fahrzeug     auto1;      // Objekt vom Typ Fahrzeug
Fahrzeug     auto2;

auto1.SetzeBesitzerName("Kocher");  // Weise Name zu

float steuer = auto1.BerechneKfzSteuer() + auto2.BerechneKfzSteuer();
```

### 2.5.2.2 Abstrakte Datentypen

Abstrakte Datentypen werden in C++ durch zwei Mechanismen unterstützt. Zum einen erlaubt die Sprache das Überladen aller eingebauten Operatoren. Dadurch ist es möglich, z.B. eine Klasse *KomplexeZahl* zu definieren, die in ihrer Anwendung von eingebauten Datentypen nicht zu unterscheiden ist. Funktionen können ebenso überladen werden, d.h. es ist möglich, verschiedene Funktionen mit dem gleichen Namen aber unterschiedlichen Parametertypen zu vereinbaren. Anhand der Aufrufargumente entscheidet der Compiler automatisch, welche Version der Funktion aufgerufen werden muß.

Der zweite Mechanismus sorgt für die automatische Initialisierung selbstdefinierter Datentypen. In jeder Klasse können sog. *Konstruktoren* definiert werden. Ein Konstruktor wird beim Erzeugen eines Objektes automatisch aufgerufen, um das Objekt zu initialisieren und in einen definierten Zustand zu bringen. Ein Konstruktor ist eine Member-Funktion, die den gleichen Namen wie die Klasse hat. Durch Überladen ist es möglich, mehrere Konstruktoren mit unterschiedlichen Parametern zu definieren. Der Compiler sucht sich auch hier immer den richtigen aus. Der Destruktor ist das Gegenstück zum Konstruktor. Er wird am Ende der Lebenszeit eines Objektes aufgerufen, um die zum geordneten Zerstören des Objektes notwendigen Aktionen durchzuführen (z.B. zur Rückgabe allokierten Speichers). Das folgende Beispiel zeigt, wie die Klasse Fahrzeug um einen Konstruktor erweitert werden kann, der das Objekt gleich mit einem Besitzernamen versieht. Der Destruktor sorgt automatisch dafür, daß der für den Besitzernamen angelegte Speicher an das Betriebssystem zurückgegeben wird.

```
class Fahrzeug {
public:
  Fahrzeug(char * name);                    // Konstruktor
  ~Fahrzeug();                              // Destruktor
  void SetzeBesitzerName(char * name);
  float BerechneKfzSteuer();
...
};

Fahrzeug::Fahrzeug(char * name)
{
  SetzeBesitzerName(name);
}
```

```
Fahrzeug::~Fahrzeug()
{
  delete [] fBesitzerName; // Von strdup angelegten Speicher freigeben
}
```

Der Aufruf erfolgt implizit beim Anlegen eines Objektes der Klasse *Fahrzeug*.

```
Fahrzeug     auto1 ("Kocher");       // auto1 hat Besitzer "Kocher"
Fahrzeug     auto2;                  // Fehler: Kein Name angegeben
```

Beide Mechanismen zusammen erlauben die Definition von Klassen, die sich wie eingebaute Datentypen verhalten.

### 2.5.2.3 Vererbung

C++ unterstützt Mehrfachvererbung. Die normale Vererbung wird durch das Schlüsselwort *public* gekennzeichnet. Die abgeleitete Klasse ist dann ein echter Untertyp der Oberklasse und kann überall anstelle dieser eingesetzt werden. Zusätzlich gibt es eine private Vererbung. In diesem Fall werden alle Daten- und Funktions-Member zwar ererbt, sie sind jedoch, analog zu privaten Membern, von außen nicht zugänglich. Es handelt sich dabei um keinen Untertyp, so daß die Unterklasse nicht anstelle der Oberklasse verwendet werden kann. Diese Art der Vererbung ist nur sinnvoll, wenn die Implementierung, nicht jedoch die Schnittstelle einer Klasse vererbt werden soll. Meist kann und sollte sie durch Datenfelder innerhalb der Klasse ersetzt werden.

Der folgende Code zeigt die Deklaration der abgeleiteten Klasse *Bus* aus den Klassen *Fahrzeug* und *Transport* (vgl. Bild 2.2).

```
class Fahrzeug { … };
class Transport { … };

class Bus : public Fahrzeug, public Transport { … };
```

Namenskonflikte bei Mehrfachvererbung dürfen nicht auftreten. Sie können dadurch gelöst werden, daß die Unterklasse eine Methode mit demselben Namen anbietet wie die Methoden der Oberklassen. Die Methoden der Oberklasse werden auf diese Weise verdeckt, und der Aufruf der Methode ist eindeutig.

Wenn eine Oberklasse im Vererbungsgraphen mehrfach auftritt, ist sie auch mehrfach im Objekt vorhanden. Dies kann gewünscht sein, ist es aber nicht immer. Wenn z.B. die Klassen *Fahrzeug* und *Transport* jeweils von einer gemeinsamen Klasse *Transportmittel* erben, würde ein *Bus* zwei Kopien von *Transportmittel* enthalten. Um dies zu verhindern, kann eine Vererbung mit dem Schlüsselwort *virtual* als virtuell gekennzeichnet werden. In diesem Fall wird nur eine Kopie der Oberklasse eingefügt.

#### 2.5.2.4 Polymorphie

Polymorphie kann in C++ für jede Methode getrennt festgelegt werden. Normale Methodendeklarationen sind nicht polymorph und werden durch den Compiler statisch gebunden. Methoden, denen das Schlüsselwort *virtual* vorangestellt wurde, verhalten sich polymorph und werden dynamisch zur Laufzeit des Programmes gebunden. Dieses abgestufte Konzept erlaubt es, einen optimalen Kompromiß zwischen Laufzeit und Funktionalität für jede Klasse getrennt festzulegen.

#### 2.5.2.5 Parametrisierte Klassen

C++ erlaubt die Definition parametrisierter Klassen, sogenannter *Templates*. Durch Angabe verschiedener Parameter kann der Compiler aus einer Vorlage eine Vielzahl von Klassen automatisch erzeugen. Das folgende Beispiel zeigt, wie aus einer allgemeinen Listenklasse verschiedene Listen von Fahrzeugen, Bussen und normalen Zahlen erzeugt werden können:

```
template <class T>
    class List { … };

List<int>          intListe;         // Liste mit ganzen Zahlen
List<Fahrzeug>     fahrzeugListe;    // Fahrzeugliste
List<Bus>          busListe;         // Liste mit Bussen
List<Fahrzeug *>   fzPtrListe;       // Liste mit Zeigern auf Objekte
                                     // der Klasse Fahrzeug oder davon
                                     // abgeleiteten Klassen
```

#### 2.5.2.6 Ausnahmebehandlung

Seit kurzem wurde ein Konzept zur Behandlung von Ausnahmen in C++ eingebunden [21, 11]. Es erlaubt allen Programmen, Ausnahmesituationen zu signalisieren und auf sie zu reagieren. Es arbeitet nach einem Blockkonzept. Mit *try* wird ein Block eingeleitet, bei dem eine Ausnahmesituation erwartet wird. *Throw* signalisiert eine Ausnahmesituation. Mit *catch* kann auf eine signalisierte Ausnahmesituation reagiert werden.

```
try {
    …
    // Aktionen, die möglicherweise schiefgehen…

    throw "Ausnahmesituation";       // Signalisiere Ausnahme

    …
} catch(char * nachricht) {
    // Bearbeitung der Ausnahmesituation, z.B. Ausdrucken
    cout << "Ausnahme " << nachricht << " empfangen." << endl;
}
```

Zu jedem *try*-Block darf es mehrere überladene *catch*-Statements geben. Der Compiler wählt, je nach Typ der signalisierten Ausnahme, die passende Version aus.

# Kapitel 3

# Die zeitdiskrete Simulation

Unter einer Simulation wird die Nachbildung eines realen Systems mit Hilfe eines geeigneten Modells verstanden. Im Rahmen der Modellbildung werden Aspekte des realen Systems abstrahiert und idealisiert. Das nachgebildete reale System wird durch diesen Vorgang vereinfacht, die Komplexität reduziert sich entsprechend. Modelle lassen sich nach verschiedenen Kriterien klassifizieren, je nachdem, welche Aspekte des Modells betrachtet werden sollen [50]. Grundsätzlich kann man analytische Modelle und Simulationsmodelle unterscheiden, wobei in dieser Arbeit nur die letzteren betrachtet werden. In analytischen Modellen werden die Beziehungen im realen System durch mathematische Gleichungen ausgedrückt, deren Lösungen den Systemzustand beschreiben. Bei der Simulation dagegen wird der Modellzustand schrittweise verändert, um daraus die gewünschten Aussagen abzuleiten. Während sich analytische Lösungsansätze nur für eine bestimmte Gruppe von Problemen finden lassen, können mit Hilfe der Simulation auch sehr komplexe Systeme untersucht werden.

Je nach Art der Zustandsübergänge im Modell können statische und dynamische Modelle unterschieden werden. Während im statischen Modell keine Zustandsänderungen auftreten, kann man dynamische Modelle nach Art der Zustandsänderungen in kontinuierliche und diskrete Modelle unterteilen. Die Zustandsvariablen eines kontinuierlichen Modells lassen sich durch stetige Funktionen beschreiben. Im Gegensatz dazu ändern sich die Zustände im diskreten Fall nur zu bestimmten, ebenfalls diskreten Zeitpunkten. Bild 3.1 zeigt, daß weitere Unterteilungen in deterministische und stochastische Modelle möglich sind. Von deterministischen Modellen wird gesprochen, wenn ein eindeutiger Zusammenhang zwischen einer Eingabe in einem bestimmten Zustand und dem jeweiligen Folgezustand besteht. Ist dies nicht der Fall und lassen sich die Reaktionen des Modells nur durch Wahrscheinlichkeiten beschreiben, so handelt es sich um ein stochastisches Modell.

Stochastische Modelle können auch dazu verwendet werden, die Komplexität eines an sich deterministischen Problems entscheidend zu vereinfachen, indem komplizierte Zustandsübergänge nicht deterministisch, sondern durch Wahrscheinlichkeitsverteilungen ausgedrückt werden.

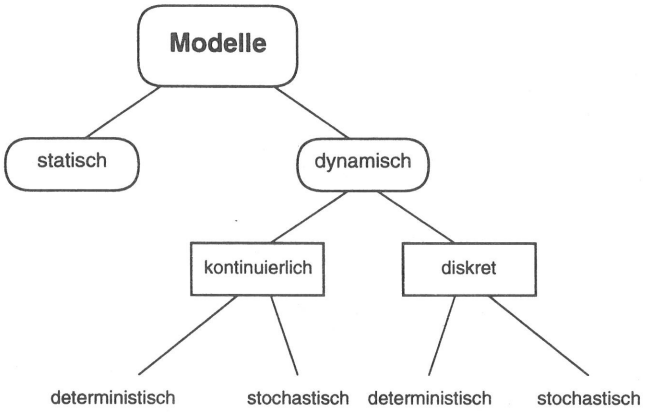



**Bild 3.1:** Klassifikation von Modellen nach Art der Zustandsübergänge

Computersimulationen sind prinzipbedingt immer zeitdiskret. Kontinuierliche Modelle können jedoch durch diskrete Modelle angenähert werden. In diesem Fall wird von quasikontinuierlichen Modellen gesprochen. Meist wird dazu die zeitgesteuerte Simulation verwendet. In diesem Fall wird die Systemzeit in festen Intervallen von $\Delta t$ verändert. Nach jeder Zeitänderung wird der neue Systemzustand bestimmt. Im folgenden sollen deshalb nur zeitdiskrete Modelle betrachtet werden.

Dieses Kapitel gibt eine kurze Einführung in verschiedene Aspekte der Simulationstechnik. Außerdem werden einige bekannte Simulationsumgebungen diskutiert und daraus die Anforderungen für die im Rahmen dieser Arbeit entstandene Simulationsbibliothek abgeleitet.

## 3.1 Überblick über verschiedene Simulationsarten

Während der Simulation ändert sich der Systemzustand in Abhängigkeit von der Zeit. Bei der diskreten Simulation können sich Zustände nur zu bestimmten diskreten Zeitpunkten verändern. Das Modell hat die Aufgabe, den Zusammenhang zwischen der statischen Struktur des Systems und seinem dynamischen Verhalten herzustellen. Anhand eines einfachen Modells für einen Netzknoten in einem Paketvermittlungsnetz soll dies verdeutlicht werden. Ein stark vereinfachtes Modell eines Netzknotens könnte z.B. aus einer Eingangswarteschlange und einer nachgeschalteten Bedieneinheit bestehen (vgl. Bild 3.2). Ankommende Pakete werden

in der Warteschlange solange zwischengespeichert, bis die Bedieneinheit frei wird. Die Bedieneinheit hat die Aufgabe, den Paketkopf auszuwerten und daraus die Zieladressse des nächsten Knotens im Netzwerk zu bestimmen.

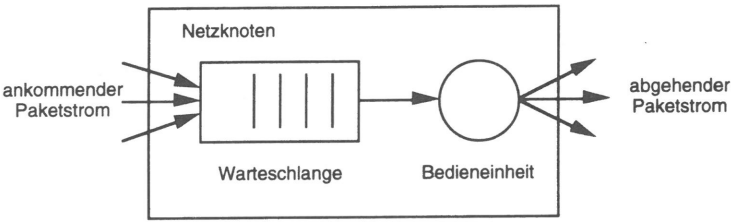



**Bild 3.2:** Einfaches Modell eines Netzknotens

In einer Simulation könnte die Zeitdauer dieses Vorgangs durch eine Wahrscheinlichkeitsverteilung beschrieben werden. Neue Pakete können nur zu bestimmten diskreten Zeitpunkten ankommen, und auch das Bedienende eines Paketes findet zu festen Zeitpunkten statt. Zwischen diesen Zeitpunkten ändert sich der Zustand des Systems nicht. Die Zeit zwischen diesen Ereignissen ist deshalb für die Simulation nicht relevant.

Im folgenden werden verschiedene zeitdiskrete Simulationsarten vorgestellt und ihre Vor- und Nachteile diskutiert.

### 3.1.1 Die ereignisorientierte Simulation

Bei der ereignisorientierten Simulation werden nur die Zeitpunkte betrachtet, bei denen sich Zustandsänderungen des Modells ergeben, nicht jedoch die Zeitspannen und Aktivitäten, welche dazwischen stattfinden. Jeder Zustandsänderung ist ein Ereignis auf der Zeitachse zugeordnet und wird von einer geeigneten Ereignisroutine bearbeitet. Zeitlich ausgedehnte Vorgänge werden auf diese Weise zu einer Folge von Ereignissen reduziert. Die Zeit zwischen zwei Ereignissen wird übersprungen, d.h. sie benötigt keine Rechenzeit während der Simulation. Obwohl während der Ausführung der Ereignisroutinen die eigentliche Rechenzeit verbraucht wird, ändert sich die Simulationszeit nicht. Im obigen Beispiel wären Ereignisse z.B. die Ankunft oder das Bedienende eines Paketes. Die zugehörigen Ereignisroutinen würden z.B. Statistiken über die durchschnittliche Wartezeit von Paketen in der Warteschlange führen oder den Zeitpunkt für das nächste Bedienendeereignis vorausberechnen. Bild 3.3 macht nochmals den Zusammenhang zwischen den Ereigniszeitpunkten und der Dauer der einzelnen Aktivitäten deutlich. Beim Bedienende eines Paketes kann z.B. die Wartezeit des nächsten Paketes aus der Differenz der Ereigniszeitpunkte für die Paketankunft

und dem Bedienende des vorausgegangenen Paketes berechnet werden, ohne daß die wirkliche Wartezeit simuliert werden muß.

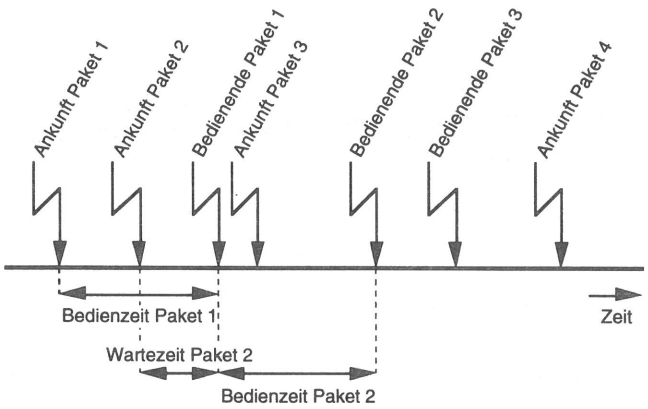



**Bild 3.3:** Ereignisse im Modell des Netzknotens

Die Ablaufsteuerung einer ereignisorientierten Simulation ist relativ einfach. Bei der Auswertung der Ereignisse muß nur das Kausalitätsprinzip beachtet werden. Es sagt aus, daß der Zustand eines Systems zum Zeitpunkt $t_j$ nur von Ereignissen mit Zeitmarken $t_e < t_j$ bestimmt wird, nicht jedoch von Ereignissen, die erst in der Zukunft eintreten. Um das Kausalitätsprinzip einzuhalten, ist es demnach erforderlich, alle Ereignisse nach ihrem Ereigniszeitpunkt geordnet zu bearbeiten. Dazu werden in einer Ereignisliste, auch Kalender genannt, alle Ereignisse nach ihrem Ereigniszeitpunkt geordnet gespeichert. Diese Liste wird während der Programmausführung der Reihe nach abgearbeitet. Dabei wird die Simulationszeit auf den jeweiligen Ereigniszeitpunkt gesetzt und anschließend die zugehörige Ereignisroutine durchgeführt. Die Ausführung dieser Routine kann u.U. zum Eintrag neuer Folgeereignisse in den Kalender führen. Die Simulationszeit ändert sich während der Ausführung der Ereignisroutine nicht. Nach Abschluß der Routine wird die Simulationsuhr zum nächsten Ereigniszeitpunkt vorgestellt und dieses Ereignis bearbeitet. Die Zeitdauer zwischen zwei Ereignissen wird dabei übersprungen.

### 3.1.2 Die prozeßorientierte Simulation

Bei der prozeßorientierten Sichtweise werden Komponenten und ihre Attribute zu Prozessen zusammengefaßt. Die Prozesse führen während der aktiven Phasen Zustandsänderungen durch. Die Simulationszeit ändert sich während der aktiven Prozeßphase nicht. Das während

einer aktiven Phase ausgeführte Teilstück einer Prozeßroutine ist mit einer Ereignisroutine vergleichbar. Nachdem der Prozeß eine Zustandsänderung durchgeführt hat, kann er sich suspendieren. Der Prozeß kann zu einem späteren Zeitpunkt weitergeführt werden. Im Gegensatz dazu wird eine Ereignisroutine immer beendet, bevor das nächste Ereignis eintritt.

In einer prozeßorientierten Simulation könnte der Bedienprozeß der Bedieneinheit aus folgenden Schritten aufgebaut sein: Zunächst wartet der Prozeß auf die Übergabe eines Paketes von der Warteschlange. Die Bedienzeit wird simuliert, indem sich der Prozeß für die Dauer der Bedienphase selbst suspendiert. Anschließend wird das Paket weitergegeben, und der Prozeß startet in einer Endlosschleife erneut. Während in der ereignisorientierten Sicht für den Beginn und das Ende der Bedienphase jeweils eigene Ereignisroutinen zuständig sind, wird in der prozeßorientierten Simulation nur eine Routine für den jeweiligen Prozeß benötigt.

Die Ablaufsteuerung einer prozeßorientierten Simulation ist relativ komplex. Die Prozesse müssen in der richtigen Reihenfolge und zum richtigen Zeitpunkt ablaufen. Dies kann z.B. durch Verwalten der Aktivierungszeitpunkte in einem Kalender geschehen. Da aber die Prozesse nicht immer von Beginn an laufen, muß der jeweilige Prozeßzustand zwischen den Aktivierungen gespeichert werden. Die Ablaufsteuerung verwaltet die Prozesse daher wie ein kleines Betriebssystem, was mit entsprechendem Aufwand verbunden ist und Probleme bei der Portierung auf andere Hardwareumgebungen bereitet.

### 3.1.3 Die transaktionsorientierte Simulation

Bei der transaktionsorientierten Sichtweise wird das System in Blöcke mit fester Funktionalität aufgeteilt. Die Transaktionen werden auf dem Weg durch die einzelnen Blöcke verändert. Sie repräsentieren den dynamischen Teil des Systems. Zustandsänderungen werden durch Transaktionen in Blöcken hervorgerufen. Bei der Modellierung des Netzknotenbeispiels werden die Warteschlange und die Bedieneinheit zu Blöcken, die Pakete sind die Transaktionen, die durch die Blöcke manipuliert werden. Das Auffinden und Eintragen der Zieladresse des nächsten Knotens in ein Paket könnte eine Transaktion sein, die von der Bedieneinheit durchgeführt wird.

Im Gegensatz zur prozeßorientierten Sichtweise, bei der mehr das Geschehen an den Bedienstationen im Vordergrund steht, wird bei der transaktionsorientierten Simulation mehr Wert auf die Modellierung des Datenflusses gelegt. Abgesehen von den unterschiedlichen Schwerpunkten bei der Modellierung, unterscheiden sich beide Verfahren nicht grundlegend. Die transaktionsorientierte Simulation wird deshalb oft als ein Teilgebiet der prozeßorientierten Simulation gesehen [34].

Sofern man das Zusammentreffen einer Transaktion mit einem Block als Ereignis betrachtet, kann die Ablaufsteuerung ereignisgesteuert aufgebaut werden. Die Transaktionen werden im Kalender nach dem Zeitpunkt des Eintretens in den nächsten Block und nach ihrer Priorität geordnet.

### 3.1.4 Bewertung der Simulationsverfahren

Der Hauptvorteil von prozeß- und transaktionsorientierten Simulationsverfahren liegt in der Übersichtlichkeit der Prozesse. Zusammenhängende Aktionen können innerhalb eines Prozesses modelliert werden. Die Abläufe innerhalb eines Prozesses sind leicht überschaubar. Dies wird mit einigen Nachteilen erkauft. Der Datenaustausch zwischen Prozessen erfordert eine Prozeßsynchronisation bzw. eine Entkopplung über Warteschlangen. Interaktionen zwischen Prozessen können zu komplexen Aktivierungs- / Deaktivierungsaufrufen anderer Prozesse führen, so daß die Übersichtlichkeit des Gesamtsystems darunter leidet. Die Implementierung selbst ist aufwendig und muß normalerweise an die jeweilige Hardwareumgebung angepaßt werden. Aufgrund der aufwendigen Prozeßverwaltung sind die Laufzeiten größer als bei vergleichbaren ereignisgesteuerten Simulationen.

Bei der ereignisorientierten Simulation werden alle Ereignisse von Ereignisroutinen bearbeitet. Dies führt zu einer einheitlichen Sichtweise innerhalb der Simulation. Der Hauptnachteil besteht darin, daß eine Folge von Aktionen in mehrere Ereignisroutinen aufgespalten wird. Die Übersichtlichkeit kann dabei verlorengehen. Durch die Möglichkeiten der objektorientierten Programmierung lassen sich jedoch mehrere Ereignisroutinen, die sich auf ein Objekt beziehen, in einer Klasse zusammenfassen, so daß die Übersichtlichkeit gewahrt bleibt. Ein Vorteil der ereignisorientierten Simulation liegt in der einfacheren Implementierbarkeit, die keine hardwarenahe Portierung voraussetzt und kurze Laufzeiten garantiert.

Der Hauptvorteil aber liegt in der Behandlung hierarchischer Systeme. Um die Komplexität heutiger Systeme beherrschbar zu machen, werden diese meist in eine Hierarchie von Teilsystemen abgebildet. Bei der Modellierung dieser Systeme bietet es sich an, ebenfalls hierarchische Modelle zu verwenden. Im prozeßorientierten Fall müßten Prozesse in der Lage sein, ihren momentanen Zustand an übergeordnete Prozesse weiterzugeben, was normalerweise nicht vorgesehen ist. Es müßten Ereignisse definiert werden, die den übergeordneten Prozessen die notwendigen Informationen zur Verfügung stellen. Im Gegensatz dazu kann dies bei ereignisorientierten Simulationssystemen automatisch erfolgen. Da sowieso jede Zustandsänderung zu einem Ereignis führt, muß nur dafür gesorgt werden, daß die Ereignisse zuerst von den höheren Hierarchiestufen bearbeitet werden können, bevor ein Eintrag in die Ereignisliste erfolgt. Im Kapitel 4 wird dieses Konzept eingehend beschrieben.

Da eine hierarchische Modellbildung als essentiell angesehen werden muß, ist dies ein wichtiger Grund, der für eine ereignisorientierte Simulation spricht. Mit Ausnahme des Übersichtlichkeitsarguments, welches aber zumindest teilweise bei einer objektorientierten Implementierung entkräftet werden kann, spricht somit alles für eine ereignisorientierte Simulation. Prozeß- und transaktionsorientierte Simulationen werden daher im folgenden nicht weiter betrachtet.

## 3.2 Grundlagen der ereignisorientierten Simulation

Im folgenden Unterkapitel sollen die wichtigsten Aspekte der ereignisorientierten Simulation zusammengefaßt werden.

### 3.2.1 Komponenten des Simulationsmodells

Zunächst folgt nochmals eine kurze Übersicht über die wichtigsten Komponenten einer ereignisgesteuerten Simulation. Einzelne Modellkomponenten haben die Aufgabe, Instanzen des realen Systems nachzubilden, z.B. Bedieneinheiten oder Warteschlangen. Modellkomponenten kommunizieren miteinander durch den Austausch von Nachrichten. Der Inhalt dieser Nachrichten hängt von dem zu simulierenden Problem ab. Der innere Zustand der Modellkomponenten kann sich nur zu festen Zeitpunkten verändern. Jede Zustandsänderung wird als Ereignis in der Simulation aufgefaßt. Es wird zwischen Ereignis und Ereigniszeitpunkt unterschieden. Das Ereignis gibt die Aktion an, die beim Eintreten des Ereigniszeitpunktes ausgeführt werden soll. Es gibt sichere und unsichere Ereignisse. Sichere Ereignisse treten auf jeden Fall ein. Unsichere Ereignisse können in Abhängigkeit von weiteren Vorgängen eintreten oder auch nicht. Ein typisches Beispiel für ein unsicheres Ereignis stellt eine Timeout-Bedingung dar. Dieses Ereignis tritt nur ein, wenn z.B. eine Nachricht nicht rechtzeitig empfangen wurde und dadurch eine Zeitbedingung verletzt wurde. Alle Ereignisse werden in einer Ereignisliste gespeichert. Die Simulationssteuerung hat die Aufgabe, alle Ereignisse, die in der Ereignisliste gespeichert sind, der Reihe nach abzuarbeiten. Während der Bearbeitung eines Ereignisses kann sich der Zustand von Modellkomponenten ändern, werden Nachrichten zwischen Modellkomponenten ausgetauscht und evtl. notwendige Folgeereignisse vorgeplant. Falls unsichere Ereignisse nicht eintreten, müssen diese aus der Ereignisliste entfernt werden können.

### 3.2.2 Ablauf der Simulation

Der Ablauf einer Simulation erfolgt in mehreren Phasen. Zunächst müssen alle Datenstrukturen der Modellkomponenten und der Simulationssteuerung initialisiert werden. Beim Start der Simulation werden die ersten Ereignisse in die Ereignisliste eingetragen. Anschließend übernimmt die Simulationssteuerung die Abarbeitung der Ereignisse. Wann ein Simulationslauf zu Ende ist, hängt von der jeweiligen Aufgabenstellung ab. Sofern es sich um eine funktionelle Simulation handelt, wird die Simulation fortgeführt, bis alle Ereignisse simuliert wurden oder ein bestimmtes Endekriterium erreicht wurde. Meist werden in Simulationen jedoch statistische Verfahren verwendet, um Aussagen über die Leistungsfähigkeit eines Systems zu erlangen. In diesen Fällen ist es notwendig, die Gesetze der Stochastik bei der Durchführung der Simulation zu beachten.

Während der Simulation werden Meßwerte für alle wichtigen Größen erfaßt. Da für die Messungen nur eine endliche Zahl von Ereignissen zur Verfügung stehen, können die Ergebnisse nur aus einer begrenzten Anzahl von Stichproben erzeugt werden. Bei den Ergebnissen handelt es sich strenggenommen nur um Schätzungen, deren Aussagesicherheit mit den Methoden der beurteilenden Statistik berechnet werden muß. Bei der allgemeinen Vorgehensweise werden wiederholte Messungen durchgeführt. Wenn man die statistische Unabhängigkeit dieser Messungen annimmt, kann man unter Verwendung statistischer Testverfahren zu Aussagen über die Qualität der Meßergebnisse gelangen. Es gibt verschiedene Methoden, wie Simulationsläufe organisiert werden müssen, damit die Messungen möglichst voneinander unabhängig sind [52]. Die einfachste Methode besteht im mehrmaligen Wiederholen der gesamten Simulation mit verschiedenen Startwerten. Man erhält damit mehrere Stichproben, die anschließend ausgewertet werden können. Dieses Verfahren eignet sich hauptsächlich für die Simulation instationärer Vorgänge. Bei der stationären Simulation will man Aussagen über das System im eingeschwungenen Zustand erhalten. Auch hier ist es möglich, mehrere Stichproben durch Wiederholung der Simulation zu erhalten. Allerdings dürfen die Messungen erst nach einer Warmlaufphase beginnen, da das System zu Beginn der Simulation noch instationär ist und erst den eingeschwungenen Zustand erreichen muß. Anstatt mehrere unabhängige Simulationsläufe durchzuführen, kann man auch einen langen Simulationslauf in mehrere Abschnitte, sog. Teiltests („Batches“) unterteilen. Diese Teiltests werden im weiteren als unabhängige Stichproben verwendet. Das größte Problem aller Verfahren ist der Nachweis der Unabhängigkeit der Stichproben. Nur dann lassen sich die Verfahren der beurteilenden Statistik anwenden.

Ein Problem der Simulationssteuerung ist es also, Kriterien für das Ende der Warmlaufphase und der einzelnen Teiltests zu finden. Dies kann z.B. durch Messung der Korrelation zwischen Teiltests geschehen. Die Teiltestlänge kann in diesem Fall sogar dynamisch angepaßt werden. In vielen Fällen reicht es jedoch aus, eine hinreichend große Zahl von Ereignissen bzw. eine entsprechend lange Zeitdauer zu simulieren. Alle Möglichkeiten zur Simulations-

steuerung an dieser Stelle zu betrachten, würde den Rahmen der Arbeit bei weitem sprengen. Stattdessen soll nur darauf aufmerksam gemacht werden, daß die Simulationssteuerung einer Simulationsbibliothek sehr flexibel ausgelegt sein muß, um alle wichtigen Verfahren unterstützen zu können. Insbesondere sollte sie nicht auf eine Methode beschränkt sein, da je nach Aufgabenstellung das eine oder andere Verfahren besser geeignet ist.

Die Simulationssteuerung hat also sowohl die Aufgabe, die Ereignisse während der Simulation abzuarbeiten als auch die einzelnen Phasen eines Simulationslaufes zu steuern.

## 3.3 Objektorientierte Simulation

Wie bereits in Kapitel 2 dargelegt wurde, ist die erste objektorientierte Programmiersprache für die Anwendung bei Simulationen entworfen worden. Trotzdem ist der Einsatz objektorientierter Sprachen nicht sehr verbreitet. Über die Gründe kann nur spekuliert werden. Eine wichtige Rolle dabei hat sicherlich die bei den Anwendern bislang wenig bekannte objektorientierte Denkweise und das schlechte Laufzeitverhalten von bekannten Programmiersprachen wie Smalltalk gespielt. Obwohl Simulationen schon länger in objektorientierten Sprachen implementiert wurden, wählten die meisten Anwender einen prozeduralen Ansatz. Da viele Simulationen in normalen Programmiersprachen implementiert werden, ist es nicht verwunderlich, daß viele Anwender ihnen bekannte Sprachen, wie Pascal oder C, verwendeten. Erst seit sich das objektorientierte Paradigma bei der normalen Anwendungsprogrammierung durchzusetzen beginnt und auch die Entwurfsmethodik besser verstanden wird, entstehen mehr Simulationsprogramme auf objektorientierter Basis.

Die objektorientierte Simulation versucht die Vorteile der objektorientierten Programmierung für Simulationen nutzbar zu machen. An erster Stelle ist hier die konzeptionelle Nähe zum Anwendungsbereich zu nennen. Sie erlaubt eine natürliche Abbildung vom Problembereich in das Simulationsmodell. Objekte des realen Systems lassen sich oft direkt als Simulationsobjekte modellieren. Dadurch wird nicht nur die eigentliche Modellbildung beschleunigt und die Transparenz des Modells erhöht, sondern auch die Programmstruktur ist für den Anwender leichter nachvollziehbar, da sie der Struktur des realen Systems ähnlich ist.

Die Wiederverwendbarkeit von Modellkomponenten ist ein weiteres wichtiges Argument, das für eine objektorientierte Simulation spricht. Zum einen wird durch den Einsatz fertiger Komponenten die Entwicklungszeit deutlich verkürzt, zum anderen nimmt die Zuverlässigkeit bei Verwendung bereits getesteter, ausgereifter Klassen zu. Bei vielen Simulationsmodellen der gleichen Art treten auf Programmebene oft gleiche Komponenten auf, z.B. werden Modelle von Warteschlangen bei vielen verkehrstheoretischen Modellen benötigt. Werden diese Komponenten als Objekte modelliert, so lassen sie sich leicht in verschiedene Simula-

tionsprogramme einbinden. Durch die Eigenschaften der Vererbung ist es auf einfache Art möglich, vorhandene Komponenten zu modifizieren und diese somit an neue Anwendungsgebiete anzupassen. Eine andere Möglichkeit der Erweiterung vorhandener Komponenten besteht darin, neue Komponenten aus bereits existierenden Objekten aufzubauen. Die hierarchische Modellbildung im Verbund mit den bereits angesprochenen Vorteilen trägt wesentlich zur Reduzierung der Komplexität bei.

Die objektorientierte Methodik der evolutionären Systementwicklung erlaubt es darüber hinaus, in kurzer Zeit lauffähige Prototypen der Simulation zu erstellen. Die Verifikation von Systemmodellen wird dadurch wesentlich erleichtert, und es ist auf einfachere Weise möglich, die Auswirkungen beim Einsatz unterschiedlicher Teilmodelle auszuprobieren. Durch Vererbung können zu einem späteren Zeitpunkt einzelne Komponenten weiter verfeinert und in das bestehende Simulationsprogramm integriert werden, ohne daß die übrigen Programmteile geändert werden müssen.

Natürlich kommen diese Vorteile beim Einsatz objektorientierter Methoden nicht von allein. Ein ausgewogener, durchdachter Entwurf ist eine wichtige Voraussetzung zum Erreichen dieser Ziele. Da aber durch die Wiederverwendbarkeit der Komponenten bei späteren Entwicklungen viel Aufwand und Zeit gespart werden, lohnt sich der erhöhte Aufwand beim ersten Entwurf auf jeden Fall. Dazu gehören einheitliche Schnittstellen zwischen Modellkomponenten ebenso wie eine durchgängige Architektur bei der Entwicklung eines Simulationssystems. Erst durch den Einsatz objektorientierter Methoden ist es möglich geworden, universelle Simulationsbibliotheken zu entwerfen, die für eine Vielzahl zu lösender Probleme geeignet sind.

Nicht zuletzt trägt ein einheitliches Konzept zur leichteren Erlernbarkeit von Simulationsbibliotheken und zur Reduzierung der Komplexität von Simulationsprogrammen bei. In Kapitel 3.5 werden die Eigenschaften einiger Simulationsumgebungen genauer untersucht. Daran anschließend folgt eine Bewertung inwieweit die obengenannten Ziele erreicht wurden. Gleichzeitig lassen sich daraus Anforderungen für die in Kapitel 4 vorgestellte Simulationsbibliothek ableiten.

## 3.4 Parallele Simulation

Die steigende Komplexität von Informationssystemen drückt sich in immer anspruchsvolleren Simulationen aus. Um komplexe Systeme in einem vernünftigen Zeitrahmen simulieren zu können, reicht die Rechenleistung heutiger sequentieller Computersysteme oftmals nicht mehr aus. Ein weiteres Problem stellt die Anzahl der Ereignisse in einem großen System dar, z.B. bei der Simulation des extrem großen Nachrichtenverkehrs auf schnellen Kommunika-

tionsnetzen. Die Simulation seltener Ereignisse, wie z.B. extrem niedriger Bitfehlerraten auf Glasfasernetzen, erfordert ebenfalls eine riesige Zahl zu simulierender Ereignisse, um die gewünschte statistische Aussagesicherheit zu gewährleisten. In all diesen Fällen ist eine Steigerung der Simulationsgeschwindigkeit nur zu erwarten, wenn es gelingt, die Simulationsmodelle auf massiv parallele Rechnerstrukturen, Multiprozessorsysteme oder Rechner-Cluster abzubilden.

Es gibt verschiedene Arten, eine Simulation zu parallelisieren, von denen die wichtigsten im folgenden kurz vorgestellt werden sollen. Obwohl im Rahmen dieser Arbeit die parallele Simulation nicht im Vordergrund steht, wurden diesbezüglich bereits einige Gesichtspunkte beim Entwurf der Simulationsbibliothek berücksichtigt. Es sollte daher möglich sein, die Grundarchitektur der Simulationsbibliothek bei einer späteren Erweiterung für parallele Simulation beizubehalten.

### 3.4.1 Gleichzeitiger Ablauf mehrerer Simulationen

Die einfachste Methode, eine sequentielle Simulation zu beschleunigen, besteht darin, die für eine Parameterstudie erforderlichen Simulationsläufe nicht hintereinander, sondern gleichzeitig auf mehreren Rechnern auszuführen. Dabei wird auf jedem Rechner eine Kopie des gleichen sequentiellen Simulationsprogrammes, jedoch mit unterschiedlichen Eingangsparametern, gestartet. Die einzelnen Simulationsläufe finden zwar zur gleichen Zeit statt, sind aber ansonsten völlig voneinander getrennt. Sofern die einzelnen Programme von Hand auf dem jeweiligen Rechner gestartet werden, kann dazu jedes sequentielle Simulationsprogramm ohne Änderungen verwendet werden. Sollen die Läufe automatisiert werden, so bleibt das eigentliche Simulationsprogramm unverändert. Es wird lediglich eine zusätzliche Schicht um das bestehende Simulationsprogramm gelegt, welche dafür verantwortlich ist, die einzelnen Simulationsläufe auf verschiedenen Rechnern mit unterschiedlichen Parametern zu starten, nach Abschluß der Simulation die Ergebnisse zu sammeln und in geeigneter Weise aufzubereiten.

### 3.4.2 Parallelisierung von Teiltests

Wie im vorangegangenen Unterkapitel bereits beschrieben wurde, laufen die meisten Simulationen in mehreren Phasen ab. Nach einer einmaligen Warmlaufphase folgen mehrere sog. Teiltests, die anschließend ausgewertet werden. Um einen **einzelnen** Simulationslauf zu beschleunigen, bietet es sich an, die Teiltests parallel auf mehreren Rechnern durchzuführen. Dazu wird das Simulationsprogramm auf mehreren Rechnern mit den gleichen Eingangsparametern, jedoch unterschiedlichen Startwerten für die Zufallszahlenerzeugung, gestartet. Nach der Warmlaufphase wird jeweils ein Teiltest durchgeführt und die Ergebnisse werden

zur Auswertung an den Steuerrechner übermittelt. Im Gegensatz zur sequentiellen Methode liegen die Ergebnisse bereits nach der Dauer der Warmlaufphase und eines Teiltests vor. Werden allerdings zu einer Parameterstudie mehrere komplette Simulationsläufe benötigt, erhält man das Gesamtergebnis normalerweise nicht schneller als bei der Methode nach 3.4.1 Sofern jedoch sehr viele Rechner zur Verfügung stehen, können beide Methoden kombiniert werden, um auf diese Weise mehrere Simulationsläufe und die einzelnen Teiltests der Simulationsläufe gleichzeitig durchzuführen.

Ein Nachteil dieser Methode, nämlich daß für jeden Teiltest ein getrennter Warmlauf erforderlich ist, kann behoben werden, indem der Zustand des Systems nach der Warmlaufphase abspeichert und an alle beteiligten Rechner verteilt wird. Ausgehend von diesem Grundzustand werden dann die Teiltests mit unterschiedlichen Startwerten für die Zufallszahlenerzeugung ausgeführt.

### 3.4.3 Funktionale Aufteilung

Im Gegensatz zu den bisher vorgestellten Ansätzen, bei denen die eigentliche Simulation im Prinzip weiterhin sequentiell abläuft, werden bei diesem Ansatz die verschiedenen Grundfunktionen einer Simulation auf mehrere Rechner verteilt. Die Aufteilung erfolgt dabei nach funktionalen Gesichtspunkten, z.B. könnten Zufallszahlen von einer Spezialhardware auf Vorrat erzeugt und an die Verbraucher verteilt werden. Mit Hilfe einer Rauschquelle lassen sich auf diese Weise sogar „echte" Zufallszahlen erzeugen.

Da bei einer normalen Simulation viele verschiedene funktionelle Einheiten eng zusammenarbeiten, kann sich, abhängig von der Art der Aufteilung, ein erheblicher zusätzlicher Kommunikationsaufwand ergeben. Diese Art der Simulation ist deshalb nur für spezielle Anwendungsgebiete geeignet.

### 3.4.4 Zentrale Ereignisverwaltung

Bei dieser Art der Parallelisierung wird das Simulationsmodell in mehrere Teilmodelle unterteilt, die auf jeweils eigenen Rechnern ausgeführt werden. Ein Problem bei dieser Art der Aufteilung besteht darin, daß die Simulationen der Teilmodelle, abhängig von der Komplexität des Modells und der Leistungsfähigkeit des Rechners, unterschiedlich schnell ablaufen. Sobald jedoch einzelne Nachrichten zwischen den Teilmodellen ausgetauscht werden sollen, muß sichergestellt sein, daß die Simulationsuhren der beteiligten Modelle synchron laufen. Sonst kann es passieren, daß ein Modell eine Nachricht für einen Zeitpunkt in der Vergangenheit erhält. Um dieses Problem zu umgehen, wird der Ereigniskalender weiterhin zentral geführt. Da alle Ereignisse in diesem Kalender registriert werden, synchronisieren sich die

Teilmodelle auf diese Weise automatisch. Bei der Ausführung werden die Ereignisse aus dem zentralen Kalender ausgetragen und an das jeweilige Teilmodell zur Ausführung übergeben. Dabei muß auf die Einhaltung des Kausalitätsprinzipes geachtet werden. Der Nachteil dieser Methode wird an dieser Stelle ebenfalls sichtbar: Der zentrale Kalender stellt den Flaschenhals des Gesamtsystems dar. Durch den Einsatz spezieller Hardware und geeigneter Kommunikationsstrukturen kann dieser Nachteil jedoch teilweise wieder ausgeglichen werden. Der Vorteil dieses Ansatzes ist in der einfachen Synchronisation des Systems über den Kalender zu suchen.

### 3.4.5 Verteilte Ereignisverwaltung

Dieses Verfahren unterscheidet sich vom vorhergehenden dadurch, daß auch die Ereigniskalender verteilt sind. Der zentrale Kalender mit seiner Flaschenhalsproblematik entfällt somit. Jedes Teilmodell bekommt eine eigene Kalenderverwaltung und eine eigene „virtuelle" lokale Zeit. Um den oben angeführten Fehlerfall auszuschließen, müssen die lokalen Uhren miteinander synchronisiert werden. Dabei gibt es im wesentlichen zwei verschiedene Ansätze:

Beim sog. **pessimistischen** oder **konservativen** Ansatz wird von vornherein sichergestellt, daß der genannte Fehler gar nicht erst auftreten kann. Die lokale Zeit eines Teilmodells darf erst dann weitergestellt werden, wenn sichergestellt ist, daß von keinem anderen Modell eine Nachricht älteren Datums mehr eintreffen kann. Ein Problem dieser Strategie ist, daß es bei diesem Vorgang zu Verklemmungen kommen kann. Es gibt unterschiedliche Algorithmen [14, 15], die alle einen mehr oder weniger großen Aufwand zur Synchronisation und zur Vermeidung bzw. Auflösung von Verklemmungen erfordern.

Der **optimistische** Ansatz geht davon aus, daß der Fehlerfall nicht auftritt und alle eintreffenden Nachrichten einen Zeitstempel haben, der in der Zukunft liegt. Somit entfällt zunächst der notwendige Aufwand, um die lokalen Uhren zu synchronisieren. Wenn doch eine Nachricht eintrifft, die älter als die momentane lokale Zeit ist, muß die Simulation abgebrochen und von diesem Zeitpunkt an neu gestartet werden. Alle Ereignisse und deren Auswirkungen zwischen diesem Zeitpunkt und dem momentanen lokalen Zeitpunkt müssen rückgängig gemacht werden (Rollback-Algorithmus). Es gibt verschiedene Ansätze, wie das Wiederaufsetzen der Simulation erfolgen kann. Allen Ansätzen gemein ist die Notwendigkeit, den Zustand der Simulation regelmäßig abzuspeichern, um die Simulation nach einem Rollback wieder von einer früheren Stelle erneut starten zu können. Das bekannteste optimistische Verfahren ist der sog. „Time Warp"-Mechanismus [32, 33].

Je nachdem, wie häufig ein Rollback erforderlich und wie groß der dazu nötige Aufwand ist, kann der optimistische oder der pessimistische Ansatz besser geeignet sein. Im ersten Fall

entsteht der Hauptaufwand, wenn der Fehlerfall tatsächlich eintritt. Im anderen Fall ist ständig ein gewisser Aufwand zu betreiben, der jedoch im Einzelfall einfacher ausfällt. Welcher Ansatz zu besseren Ergebnissen führt, hängt demnach stark vom zu untersuchenden System ab. Der Hauptnachteil konservativer Verfahren liegt darin, daß sie Parallelität oft nicht nutzen können, da sie immer sicherstellen müssen, daß kein Ereignis eintrifft, dessen Zeitstempel in der Vergangenheit liegt. Dies ist selbst dann der Fall, wenn die Ereignisse sich nicht gegenseitig beeinflussen. Der optimistische Ansatz hingegen nutzt vorhandene Parallelität ohne Zutun des Programmierers automatisch aus, da bei unabhängigen Ereignissen weniger Rollbacks erforderlich sind. In Untersuchungen schließt der optimistische Ansatz deshalb meist wesentlich besser ab [24, 25]. Ein guter Einführungsartikel über parallele Simulation findet sich in [26].

### 3.4.6 Modellierung paralleler Vorgänge

Sofern die Vorgänge an sich schon paralleler Natur sind, können spezielle Algorithmen gefunden werden, die ein optimales Simulationsergebnis erwarten lassen. Da solche Lösungsansätze nur für eine bestimmte Form von Problemen geeignet sind, lassen sich keine allgemeinen Lösungswege finden. Der Einsatz allgemeiner Simulationsumgebungen stößt hier an seine Grenzen, da nicht für jedes spezielle Problem eine entsprechende Lösung vorgesehen werden kann. Probleme dieser Art sollen deshalb im folgenden nicht weiter betrachtet werden.

### 3.4.7 Abschätzung des Implementierungsaufwandes für parallele Simulation

Die in den vergangenen Unterkapiteln vorgestellten Methoden erfordern einen unterschiedlich hohen Aufwand bei der Realisierung. Im folgenden soll abgeschätzt werden, mit welchem Aufwand sich eine durchdachte Simulationsbibliothek, die für die sequentielle Simulation entworfen wurde, für parallele Anwendungen erweitern läßt.

Die in 3.4.1 und 3.4.2 gezeigten Ansätze laufen im wesentlichen auf eine Modifizierung der Simulationssteuerung hinaus. Die eigentliche Simulation läuft nach wie vor sequentiell ab. Die im Rahmen dieser Arbeit vorgestellte Simulationsbibliothek läßt sich sehr leicht für diese Anforderungen modifizieren, so daß sogar bereits fertiggestellte Simulationsprogramme mit minimalem Aufwand umgestellt werden können. Die Simulationssteuerung erfolgt durch ein eigenes Objekt der Klasse *TSimulation*. Eine davon abgeleitete Klasse könnte die zusätzlichen Aufgaben der Verteilung auf verschiedene Rechner übernehmen, ohne daß das eigentliche Simulationsprogramm verändert werden muß. Nur wenn der gesamte Zustand zur Vermeidung mehrerer Warmlaufphasen abgespeichert werden soll, ist es erforderlich, daß der

Systemzustand in einer Datei abgelegt und von dort wieder gelesen werden kann. In diesem Fall bietet sich der Einsatz einer objektorientierten Datenbank an, die es erlaubt, C++-Objekte persistent zu verwalten.

Die funktionale Aufteilung, wie sie in 3.4.3 beschrieben ist, eignet sich weniger für eine Erweiterung der Simulationsbibliothek, da die einzelnen Komponenten in einem sehr engen funktionalen Zusammenhang stehen, der sich nicht ohne weiteres auf verschiedene Rechner verteilen läßt. Eine solche Aufteilung ist sowieso sehr problemabhängig und schon aus diesem Grunde nicht für den Einsatz einer universellen Simulationsbibliothek geeignet. Ebenso kommt der in 3.4.6 gezeigte Ansatz für eine universelle Anwendung nicht in Frage.

Dagegen können die in 3.4.4 und 3.4.5 diskutierten Methoden auf die Simulationsbibliothek angewendet werden. Aus der Möglichkeit, hierarchische Modelle direkt abzubilden, folgt unmittelbar, daß eine Aufteilung in verschiedene Teilmodelle möglich ist. Die Architektur unterstützt in der sequentiellen Form bereits die Möglichkeit, mehrere Kalender gleichzeitig einzusetzen. Das Handshake-Protokoll, welches zum Nachrichtenaustausch zwischen Komponenten eingesetzt wird, könnte problemlos auf Netzverbindungen ausgedehnt werden. Mit einer entsprechend abgeänderten Simulationssteuerung wäre es daher möglich, sequentielle Simulationsprogramme mit vergleichsweise geringfügigem Aufwand zu parallelisieren. Der Hauptaufwand müßte dabei nicht vom jeweiligen Simulationsprogramm erbracht werden, sondern müßte nur einmal in eine erweiterte Version der Simulationsbibliothek gesteckt werden. Pessimistische Verfahren lassen sich dabei einfacher implementieren als optimistische. Das liegt daran, daß sich der Rollback-Mechanismus optimistischer Verfahren auf die einzelnen Modellkomponenten auswirkt. Bei der Beschreibung der Simulationsbibliothek in Kapitel 4 und speziell in Kapitel 5.2 wird nochmals auf den notwendigen Änderungsaufwand eingegangen.

Insgesamt kommen die Vorteile einer objektorientierten Implementierung hier deutlich zum Tragen. Durch die gute Kapselung in Blöcke mit überschaubarer Funktionalität und der dynamischen Bindung zur Laufzeit, ist es leicht möglich, einzelne Komponenten völlig neu zu implementieren, ohne daß andere Teile des Simulationsprogrammes davon betroffen sind. Eine wichtige Voraussetzung dafür ist jedoch ein vorausblickender Systementwurf, der bereits sinnvolle Schnittstellen zwischen den einzelnen Blöcken vorsieht. Ein Block mit veränderter Funktionalität läßt sich nur dann leicht austauschen, wenn die Schnittstellen zwischen den Blöcken weiterhin erhalten bleiben. Die Bedeutung abstrakter Basisklassen, die nur Schnittstellen, jedoch keine Implementierung festlegen, muß in diesem Zusammenhang besonders hervorgehoben werden. Im Kapitel 4 wird dieser wichtige Punkt beim Systementwurf noch ausführlich behandelt.

## 3.5 Eigenschaften bekannter Simulationspakete

In den folgenden Abschnitten sollen die Eigenschaften einiger Simulationsumgebungen kurz vorgestellt werden. Dabei wird besonderer Wert auf die Architektur und die möglichen Anwendungsgebiete gelegt. Eine vollständige Beschreibung kann im Rahmen dieser Arbeit leider nicht erfolgen, es wird daher auf die entsprechende Literatur verwiesen. Auf die Beschreibung prozeduraler Simulationssprachen wie GPSS wurde verzichtet, da die Vorteile objektorientierter Programmierung nicht zum Tragen kommen. Umgebungen für parallele Simulation wurden ebenfalls nicht betrachtet, da sie nicht das Hauptziel dieser Arbeit darstellen. Die angesprochenen Probleme lassen sich aber durchaus auch auf diese Umgebungen übertragen.

Simulationsumgebungen lassen sich grob in zwei Kategorien einteilen. Zum einen gibt es eigenständige Umgebungen, die meist auf spezielle Probleme zugeschnitten sind. Zum anderen gibt es Simulationsbibliotheken, die in standardisierten Programmiersprachen geschrieben wurden und die vom Anwender erweiterbar sind. Beide Konzepte sollen in den folgenden Unterkapiteln anhand von Beispielen betrachtet werden.

### 3.5.1 Eigenständige Simulationsumgebungen

Eigenständige Simulationsumgebungen zeichnen sich dadurch aus, daß sie meist sehr problemorientiert sind. Sie bieten problemnahe Sprachen, die es dem Anwender erlauben, Simulationen mit hohem Abstraktionsniveau zu spezifizieren. In manchen Fällen kann ein Simulationsmodell sogar mit einer graphischen Benutzeroberfläche komfortabel erstellt werden. Allerdings hat diese Problemnähe einen Preis. Aufgrund der Spezialisierung eignen sich die Simulationsumgebungen oft nur für eine bestimmte Klasse von Anwendungen. Eigene Erweiterungen können meist nur in beschränktem Umfang vorgenommen werden. Der Vorteil einer problemnahen Sprache wird durch den Aufwand zum Erlernen einer neuen Sprache relativiert. Da es sich nicht lohnt, für jede Simulationssprache einen entsprechenden Compiler zu entwickeln, werden viele dieser Sprachen interpretiert oder in eine Standardprogrammiersprache übersetzt. Dadurch können sich Laufzeitprobleme bei der Simulation komplexer Systeme ergeben. Da die meisten Simulationsumgebungen nur für gängige Plattformen zur Verfügung stehen, wird ihr Anwendungsgebiet weiter eingeschränkt.

Ein bekanntes Beispiel für eine komplette Simulationsumgebung stellt ModSim dar. ModSim steht für „Modular Simulation Language“ und wurde von der amerikanischen Armee für die Simulation von Kriegsszenarien entwickelt [30, 4]. Sie vereinigt Eigenschaften normaler Programmiersprachen mit objektorientierten Ansätzen und unterstützt die zeitdiskrete ereignisgesteuerte Simulation. Das Hauptanwendungsgebiet ist allerdings die prozeßorientierte Simulation. Alle Klassen, die von der Klasse *ProcessObject* abgeleitet wurden, stellen einen

eigenen Prozeß dar. Jedesmal wenn ein solches Objekt eine spezielle Art von Methoden (sog. *Tell*-Methoden) aufruft, wird ein neuer Prozeß erzeugt, der unabhängig weiterarbeitet. Da für diese Methoden keine Rückgabewerte erlaubt sind, kann die aufrufende Routine sofort weiterlaufen. Durch Warteanweisungen können Objekte miteinander synchronisiert werden. ModSim unterstützt Mehrfachvererbung und hat ein Modula-ähnliches Modulkonzept, das streng zwischen Definition und Implementierung unterscheidet. Da ModSim eine eigene Programmiersprache darstellt, kann sie für sehr unterschiedliche Anwendungen eingesetzt werden. Es existiert deshalb kein klares Konzept, wie Modelle aufzubauen sind. Insbesondere findet sich keine Unterstützung für die Bildung hierarchischer Modelle, außer durch Aggregation vorhandener Komponenten. Die Kommunikation zwischen Komponenten erfolgt durch direkten Methodenaufruf. Dadurch ist es notwendig, Referenzen auf andere Komponenten in der Komponente selbst zu verwalten, was die Wiederverwendbarkeit einschränkt. Wie Leistungsuntersuchungen zeigen [5], zählt ModSim eher zu den leistungsschwächeren Simulationspaketen. Der Hauptvorteil besteht in der eigenständigen Programmiersprache, die direkt Konstrukte zur Simulationsunterstützung bereitstellt und die beliebige Erweiterungen durch den Anwender gestattet.

Im Gegensatz zu ModSim ist Q+ eine Simulationsumgebung, die sehr spezifisch auf einen Problembereich zugeschnitten ist. Es handelt sich um ein Softwareprodukt, das von AT&T entwickelt wurde [42, 43]. Q+ ist eine Simulationsumgebung, die auf die Simulation von Warteschlangennetzen spezialisiert ist. Der Anwender kann mit Hilfe einer graphischen Oberfläche ein Modell eines beliebigen Warteschlangennetzes entwerfen. Anschließend können das Modell simuliert und die Ergebnisse statistisch ausgewertet werden. Es ist sogar möglich, die Simulation schrittweise interaktiv ablaufen zu lassen. Mittels Animation kann der Anwender die Vorgänge direkt nachvollziehen, was besonders in der Testphase hilfreich ist. Der Anwender kann zwar Objekte, wie Warteschlangen oder Bedieneinheiten, verwenden, eigene Erweiterungen sind jedoch nicht möglich. Dies schränkt den allgemeinen Einsatz dieses Werkzeugs ein. Objektorientierte Konzepte wurden überhaupt nicht realisiert, selbst die Implementierung erfolgte in einer prozeduralen Sprache. Auch das hierarchische Konzept beschränkt sich auf die Aggregation einzelner Komponenten zu neuen Komponenten. Somit wird die einfache Handhabung des Systems durch Einschränkungen in der Funktionalität und der Erweiterbarkeit erkauft.

Die Reihe der Beispiele ließe sich weiter fortsetzen. Vor allem sehr spezialisierte Simulationsumgebungen zeichnen sich durch große Anwenderfreundlichkeit und den Einsatz graphischer Bedienoberflächen aus. Für viele Problemstellungen wie z.B. der Simulation von Materialflüssen durch eine Fabrik, stellt die fehlende Erweiterbarkeit kein Problem dar, da das Anwendungsgebiet überschaubar und meist durch die spezialisierte Lösung vollständig abgedeckt wird. Es fällt aber auf, daß zwischen allgemein verwendbaren und sehr spezialisierten Umgebungen eine Lücke klafft. Es fehlen problemorientierte Simulationsumgebungen, die leicht erweitert werden können.

### 3.5.2 Standard-Simulationsbibliotheken

Traditionelle Simulationsbibliotheken verwenden normale Programmiersprachen, um ein Simulationsprogramm zu erstellen. Sie beschränken sich auf häufig benutzte Komponenten, die der Anwender unverändert übernehmen kann. Der Rest des Simulationsprogrammes wird in herkömmlicher Weise programmiert. In letzter Zeit werden immer mehr Bibliotheken in objektorientierten Programmiersprachen, hauptsächlich in C++, entworfen, um die Vorteile der objektorientierten Programmierung in diesem Bereich nutzen zu können. Im Gegensatz zu eigenständigen Simulationsumgebungen, können diese Bibliotheken jederzeit mit allen Mitteln der eingesetzten Programmiersprache erweitert werden. Der Anwender hat den Vorteil, daß er keine neue Programmiersprache lernen muß. Da das resultierende Simulationsprogramm in einer Standardprogrammiersprache geschrieben ist, stellt die Portierung auf neue Plattformen meist kein Problem dar.

Wie sehr die erhöhte Flexibilität dieser Lösung durch mangelnden Komfort bei der Erstellung des Simulationsprogrammes zunichte gemacht wird, hängt stark von der Architektur der Simulationsbibliothek ab. In letzter Zeit sind verstärkt Anstrengungen zu verzeichnen, durch geeignete objektorientierte Abstraktionen eine sehr problemnahe Realisierung von Simulationsbibliotheken zu erreichen. Dabei wird versucht, mit einer durchdachten Architektur der Simulationsbibliothek die Lücke zwischen Anwendernähe und Erweiterbarkeit zu schließen.

Es ist auffallend, daß die interessantesten Ansätze in diesem Bereich auf Abstraktionen aus dem Problembereich aufbauen. Bibliotheken wie DOSE (Discrete-Event Object-Oriented Simulation Environment) [38] oder PRISM [63, 64] haben eine klare Architektur, bei der die Simulationsmodelle aus einzelnen Modellkomponenten bestehen, die auch hierarchisch aufgebaut sein dürfen. In beiden Fällen erfolgt die Simulation ereignisorientiert. Während bei DOSE die Ereignisroutinen direkt an einen Kalender übergeben werden, abstrahiert PRISM diesen Vorgang mit einer eigenen Ereignisklasse. Neue Komponenten lassen sich durch Aggregation aus bestehenden Komponenten aufbauen oder durch Vererbung von vorhandenen Komponenten ableiten. Obwohl der hierarchische Aufbau von Modellkomponenten auf diese Weise unterstützt wird, ist eine hierarchische Ereignisverarbeitung nicht vorgesehen. Um möglichst abgeschlossene Komponenten zu erhalten, die sich leicht wiederverwenden lassen, verwendet DOSE ein Port-Konzept. Modellkomponenten kommunizieren nur über explizite Schnittstellen (sog. Ports) miteinander und können durch ihr Verhalten an den Ein- und Ausgängen beschrieben werden. Auf unterster Ebene findet der Informationsaustausch in PRISM nur über Ereignisse statt. Darauf aufbauend wurden Klassen zur Simulation von Warteschlangennetzen implementiert. Diese benutzen ebenfalls ein ereignisgesteuertes Port-Konzept zum Nachrichtenaustausch zwischen Modellkomponenten.

Port-Konzepte sind ein beliebtes Mittel zur Entkopplung von Modellkomponenten und tauchen auch in anderen Simulationsbibliotheken auf [1, 67]. Allerdings haben sie alle den

Nachteil, daß Informationen, die von einer Modellkomponente ausgegeben werden, von den nachfolgenden Komponenten abgenommen werden müssen; eine Blockierung ist nicht vorgesehen. Dadurch ergeben sich Einschränkungen in der Wahl der Komponentenaufteilung. So müssen alle Komponenten an ihren Schnittstellen blockierungsfrei sein.

Bei prozeßorientierten Simulationsbibliotheken fällt allgemein auf, daß sie sich stark an der jeweiligen Implementierung orientieren. Der Hauptgrund dürfte die schwierige Umsetzung eines Prozeßmodells auf einem sequentiellen Rechner sein. Objektorientierte Ansätze erlauben zwar die Abstraktion von Prozeßinteraktionen als Klassen, trotzdem werden die anwendernahen Aspekte oft unzureichend berücksichtigt. Simulationsbibliotheken wie SimPOL (Simulation Process-driven Object-oriented Language) [39] oder das an der Universität Karlsruhe entwickelte SIM_PP (SIM PlusPlus) [54] sowie einige andere benutzen Erweiterungen der Sprache C++, um ein prozeßorientiertes Modell zu realisieren. Diese Erweiterungen müssen mit Hilfe von Präprozessoren in C++ umgesetzt werden. Dabei sind Konventionen einzuhalten, die von einem normalen Präprozessor schwer zu kontrollieren sind. Er müßte dazu eine Übermenge von C++ und der Simulationssprache verstehen. Die Typsicherheit von C++ geht dabei teilweise wieder verloren. Es gibt keine Konzepte für den Nachrichtenaustausch zwischen Modellkomponenten. Außerdem werden hierarchische Modelle nicht unterstützt. Die Aufteilung komplexer Systeme und die Wiederverwendbarkeit der entworfenen Komponenten sind damit in Frage gestellt. Während SimPOL ein portables C++-Programm erzeugt, das normal compiliert werden kann, erfordert die Portierung von SIM_PP auf neue Plattformen die Anpassung der Prozeßumschaltung. Sie muß (in Assembler) an die jeweilige Hardwareumgebung angepaßt werden.

Neben diesen für allgemeine Anwendungen gedachten Simulationsbibliotheken gibt es natürlich auch in diesem Bereich spezialisierte Lösungen. Ein Beispiel dafür ist SMURPH, eine prozeßorientierte Simulationsbibliothek, die ebenfalls in C++ geschrieben wurde und speziell die Simulation von Kommunikationsprotokollen unterstützt [27]. Andere spezialisierte Anwendungen bauen auf allgemeinen Bibliotheken auf. SimEnv basiert z.B. auf SIM_PP und erweitert diese um die Möglichkeiten der Netzwerkanalyse [55]. Für dieses spezielle Einsatzgebiet wurde eine eigene Sprache entworfen, die es erlaubt, vorgefertigte Komponenten zusammenzufügen und daraus ein lauffähiges Programm zu erzeugen. Dies erleichtert die Anwendung bereits vorgefertigter Komponenten. Am weitesten fortgeschritten ist in diesem Bereich die PRISM-Bibliothek, die sogar die graphische Modelleingabe, automatische Umsetzung in Simulationsobjekte und interaktive Simulation unterstützt.

Leider fehlt in den meisten Fällen eine durchgängige Architektur, so daß der Übergang von der anwendernahen Abstraktion zur darunterliegenden Implementierung bei vielen Simulationsbibliotheken schwierig ist. Der Vorteil gegenüber integrierten Simulationsumgebungen, aufgrund der standardisierten Programmiersprache eigene Erweiterungen vornehmen zu können, wird dadurch relativiert.

Am besten schneiden DOSE und PRISM ab, die beide die durchgängigsten Konzepte anbieten. Besonders PRISM zeigt, daß zwischen problemnahen Abstraktionen und einfacher Erweiterbarkeit kein Widerspruch sein muß. Allerdings wurden auch hier noch nicht alle Konzepte konsequent zu Ende gedacht.

## 3.6 Bewertung, Kritik und Ansätze neuer Methoden

Die Betrachtung der im vorigen Kapitel vorgestellten Simulationsumgebungen zeigt, daß es sehr viele unterschiedliche Ansätze gibt, die teilweise interessante Eigenschaften haben. Es gibt jedoch keine ideale Lösung für alle Simulationsprobleme. Vor allem fällt auf, daß die Möglichkeiten der objektorientierten Programmierung bisher nur in Ansätzen ausgenutzt werden. Meist gibt es zwar einige objektorientierte Konzepte, mit deren Hilfe Simulationsprogramme vereinfacht werden können. Es fehlt jedoch an einer durchgängigen Architektur, die dem Anwender klare Richtlinien zur Handhabung und Erweiterung bestehender Simulationsbibliotheken an die Hand gibt.

Spezialisierte Simulationsumgebungen vereinfachen durch ihre konzeptionelle Nähe zum Problembereich die Modellbildung erheblich. Aufgrund fehlender Erweiterungsmöglichkeiten und zu starker Spezialisierung ist ihre Verwendung jedoch auf ein enges Anwendungsgebiet beschränkt. Allgemeine Simulationsbibliotheken versprechen durch die Verwendung standardisierter Programmiersprachen und objektorientierter Ansätze einfachere Anpassung an neue Aufgabengebiete. Die vorgegebenen Abstraktionen sind sehr implementierungsbezogen. Es fehlt eine klare Konzeption, wie der Problembereich auf die in der Simulationsbibliothek realisierten Konzepte abgebildet werden soll. Damit entsteht eine Kluft zwischen den Abstraktionen des Problembereichs und denen der Simulationsbibliothek. Das objektorientierte Paradigma ist mit dem Anspruch angetreten, genau diese Kluft zu beseitigen. Daß dies mit den vorliegenden Ansätzen bisher nicht zufriedenstellend gelungen ist, hat vielfältige Gründe. Der wichtigste Grund dürfte darin liegen, daß die meisten Entwickler von Simulationsbibliotheken die Denkweise des objektorientierten Entwurfes noch nicht verinnerlicht haben und deshalb die größte Aufmerksamkeit funktionellen implementierungsnahen Gesichtspunkten widmen. Weitere Gründe sind sicher auch die mangelnde Erfahrung mit dem Einsatz solcher Bibliotheken und objektorientierter Software allgemein.

Welche Eigenschaften müßte demnach eine optimale Simulationsumgebung zur Verfügung stellen? Sie müßte die Vorteile einer am hohen Abstraktionsniveau des Problembereichs ausgerichteter Simulationsumgebung mit der einfachen Erweiterbarkeit einer in einer standardisierten Programmiersprache geschriebenen Simulationsbibliothek verbinden. Die Niederungen implementierungsabhängiger Konzepte, die zur Durchführung der Simulation erforderlich sind, sollten dem Anwender soweit wie möglich verborgen bleiben. Insbesondere

sollte es dem Anwender möglich sein, problemnahe Abstraktionen direkt in die Konzepte der jeweiligen Simulationsumgebung umzusetzen.

Wie können diese Eigenschaften erreicht werden? Dazu ist es erforderlich, die Abstraktionen einer Simulationsbibliothek nicht an den Konzepten der Implementierung auszurichten, sondern sich auf die Schlüsselabstraktionen im Problembereich zu konzentrieren. Im Falle einer Simulation sind dies: Das Modell, Modellkomponenten aus denen sich Modelle aufbauen lassen, Interaktionen zwischen Modellkomponenten sowie die Steuerung des Simulationsablaufes. Sofern diese Abstraktionen von einer Simulationsbibliothek direkt unterstützt werden, fällt es dem Anwender leicht, die in seinem Problembereich auftretenden Komponenten auf Elemente der Simulationsbibliothek abzubilden. Die Implementierung der Abstraktionen in einer standardisierten Sprache wie C++ stellt dann kein Problem dar. Durch die Ausrichtung auf den Problembereich des Anwenders kann das Potential des objektorientierten Ansatzes, wie die Wiederverwendung ganzer Modellkomponenten, besser genutzt werden, als wenn man sich auf die objektorientierte Implementierung einiger funktionaler Aspekte beschränkt. Im nächsten Schritt kann der Anwender durch Bereitstellung problemorientierter Modellkomponenten oder ganzer Teilmodelle das Abstraktionsniveau und den Komfort auf das Niveau spezialisierter Simulationsumgebungen anheben. Sogar Erweiterungen durch graphische Benutzeroberflächen können dann auf dem Fundament einer anwendernahen Simulationsbibliothek vorgenommen werden.

Das folgende Kapitel beschreibt die Architektur einer solchen, möglichst universell gehaltenen Simulationsbibliothek. Sie soll zeigen, wie sich die hier vorgestellten Ziele durch die Wahl einiger weniger grundlegender Abstraktionen und den konsequenten Einsatz objektorientierter Entwurfsmethoden verwirklichen lassen.

# Kapitel 4

# Eine objektorientierte Simulationsbibliothek

## 4.1 Ziele der Simulationsbibliothek

Wie bereits aus der Zusammenfassung des letzten Kapitels zu ersehen ist, gibt es keine ideale Simulationsumgebung. Die Einsatzgebiete sind zu verschieden, um eine optimal geeignete Simulationsbibliothek zu finden, die alle denkbaren Anwendungen unterstützt. Aber auch für bestimmte Anwendungsgebiete sind die meisten Bibliotheken nur bedingt geeignet. Entweder sind sie zu speziell auf ein Gebiet zugeschnitten und lassen sich nicht oder nur schwer erweitern, oder sie sind so allgemein gehalten, daß sie nur ein paar generelle Ansätze anbieten, ohne jedoch eine einheitliche Gesamtarchitektur zu unterstützen. Es ist deshalb auch nicht das Ziel dieser Arbeit, eine völlig universell verwendbare Simulationsumgebung zu schaffen, die alle möglichen Anwendungen unterstützt, sondern ein Konzept zu erarbeiten, das die Simulation komplexer verteilter Systeme erleichtert. Dabei wird größter Wert auf eine durchgängige Architektur gelegt, die es dem Anwender ermöglicht, die Simulationsbibliothek leicht zu verstehen und an seine speziellen Bedürfnisse anzupassen. Im folgenden werden die wesentlichen Probleme und Ziele beim Entwurf der Simulationsbibliothek zusammengefaßt.

### 4.1.1 Erhöhung der Produktivität

Die Erstellung einer komplexen Simulation erfordert einen hohen Aufwand. Der Erfolg vorgefertigter Simulationsbibliotheken liegt darin begründet, daß sie versprechen, diesen Aufwand drastisch zu verringern. Um die Produktivität zu erhöhen, reicht es allerdings nicht aus, nur einige vorgefertigte Komponenten, die häufig gebraucht werden, anzubieten, sondern es muß ein umfassendes Konzept vorhanden sein, das den Anwender in jeder Phase - vom Entwurf bis zum Test der Simulation - unterstützt. Der größte Produktivitätszuwachs erfolgt durch die Wiederverwendung bestehender Komponenten. Diese können jedoch nur wiederverwendet werden, wenn die Komponenten klare Schnittstellen mit einem definierten Verhalten haben, so daß man sie isoliert betrachten und einsetzen kann. Wenn z.B. die Implementierung einer Warteschlange so entworfen wurde, daß sie als nachfolgende Modell-

komponente immer eine Bedieneinheit benötigt, so kann sie nicht verwendet werden, wenn zwischen ihr und der Bedieneinheit eine weitere Modellkomponente, z.B. ein Multiplexer, eingefügt werden soll. Oftmals entsprechen vorhandene Komponenten nicht genau den Erfordernissen, sondern müssen angepaßt werden. Eine gute Simulationsbibliothek muß sich deshalb flexibel an neue Erfordernisse anpassen und erweitern lassen. Dabei sollten klare Regeln vorschreiben, wie diese Erweiterung zu erfolgen hat, so daß die neu entstehenden Komponenten für zukünftige Projekte einfach zu nutzen sind. Eine klare Architektur der Simulationsbibliothek erleichtert aber nicht nur die Wiederverwendung einzelner Komponenten, sondern verkürzt vor allem auch den Zeitaufwand, den ein potentieller Anwender zum Verstehen der Konzepte benötigt. Sowohl die Wartung bestehender Simulationen als auch die Erstellung einer neuen Simulation aus vorhandenen Komponenten wird vereinfacht, wenn die Grundregeln klar definiert und von allen Anwendern eingehalten werden. Der letzte Punkt erfordert zwar eine gewisse Disziplin vom Anwender, der er sich jedoch leichter unterziehen wird, wenn die Vorteile offensichtlich sind. Vor allem kann er sich nur an Regeln halten, die auch existieren. Viele Simulationsumgebungen machen hierzu nur sehr vage Aussagen, die dem Anwender oft nicht weiterhelfen. Zu viele Regeln wiederum überfordern den Anwender und erhöhen die Komplexität unnötig. Eine gute Simulationsbibliothek sollte deshalb einige Schlüsselkonzepte bereitstellen, deren Vorteile bei der Anwendung unmittelbar einleuchten.

### 4.1.2 Handhabung komplexer Systeme

Ein großes Problem stellt die Komplexität heutiger Systeme dar. Dabei überträgt sich natürlich die Komplexität des realen Systems auf die Simulation. Obwohl sich durch geeignete Abstraktionen und Idealisierungen eine erhebliche Reduktion der Komplexität erreichen läßt, ist das verbleibende Modell trotzdem oft noch sehr kompliziert. Eine Möglichkeit, die Komplexität weiter zu reduzieren, kann durch eine hierarchische Modellbildung erreicht werden. Analog zur schrittweisen Verfeinerung von Blockschaltbildern beim Hardwareentwurf bzw. zum Top-Down-Entwurf im Softwarebereich, wird das Modell in Teilmodelle unterteilt, die miteinander verbunden sind. Diese Teilmodelle können dann im nächsten Schritt weiter verfeinert werden. Dadurch ergibt sich eine Reihe von hierarchischen Modellen, die jedes für sich eine überschaubare Komplexität besitzen. Auch beim Testen des Simulationsprogrammes ergeben sich Vorteile, wenn jedes Teilmodell einzeln getestet werden kann. Bei der Bildung von Teilmodellen ergeben sich die gleichen Schwierigkeiten wie bei der Wiederverwendung einzelner Komponenten. Auch hier werden klare Schnittstellen und Regeln benötigt, um das Zusammenwirken der Teilmodelle zu erleichtern. Insbesondere ist es für viele Anwendungen interessant, ganze Teilmodelle wiederzuverwenden.

Wie man aus dieser kurzen Diskussion bereits sieht, können beide Konzepte nicht isoliert voneinander betrachtet werden. Die Architektur einer Simulationsbibliothek sollte deshalb durchgängige Konzepte vom Entwurf einfacher Modellkomponenten bis zum Aufbau kom-

pletter Teilmodelle bereitstellen. Erst die problemlose Wiederverwendbarkeit komplexer Modellkomponenten und Teilmodelle erlaubt es dem Anwender, problemnahe Lösungen zu verwenden und damit die semantische Lücke zwischen einer universellen Simulationsbibliothek und dem Problembereich des Anwenders zu schließen.

Eine weitere Schwierigkeit bei der Simulation komplexer Systeme besteht häufig darin, daß die Rechenzeit zur Durchführung des Simulationslaufes auf einem Rechner zu groß wird. Es sollte deshalb möglich sein, ein bestehendes Simulationsprogramm nachträglich mit möglichst geringen Änderungen auf mehrere Rechner zu verteilen und parallel abzuarbeiten.

Die folgenden Unterkapitel beschreiben, wie diese Ziele mit Hilfe der objektorientierten Programmierung und einer klaren Architektur unterstützt werden können.

## 4.2 Objektorientierter Entwurf der Simulationsbibliothek

Nur weil ein Entwurf nach objektorientierten Gesichtspunkten vollzogen wurde, muß das Ergebnis noch lange nicht alle Erwartungen erfüllen. Meist gibt es mehrere Möglichkeiten, ein gewähltes Konzept in ein objektorientiertes System umzusetzen. Im folgenden sollen einige grundsätzliche Vorgehensweisen vorgestellt werden, die sich beim Entwurf der Simulationsbibliothek als nützlich erwiesen haben. Es handelt sich dabei um allgemeine Aussagen, die auch auf andere Anwendungsgebiete übertragbar sind. Ein wichtiger Aspekt dieser Arbeit ist darin zu sehen, alternative Lösungsvorschläge gegeneinander abzuwägen und auf diese Weise einige Einblicke zu gewähren, die bei der Erstellung objektorientierter Software hilfreich sein können. Natürlich können an dieser Stelle nicht alle Grundsätze des objektorientierten Entwurfs wiederholt werden, vielmehr sollen einige interessante Punkte hervorgehoben werden. Weitere spezielle Hinweise in den entsprechenden Unterkapiteln ergänzen die hier vorgestellten allgemeinen Aussagen.

Auch die Wahl der Programmiersprache sollte nicht unterschätzt werden. Sie muß die direkte Umsetzung der beim Entwurf verwandten Konzepte erlauben. Außerdem werden die Randbedingungen wie Entwicklungsumgebung und Codeeffizienz stark von der Programmiersprache beeinflußt. In dieser Arbeit wurde C++ gewählt, da sie alle wesentlichen Konzepte des objektorientierten Paradigmas direkt unterstützt. Die effiziente Implementierung dieser Konzepte verspricht ein gutes Laufzeitverhalten. Die große Verbreitung dieser Sprache bringt weitere Vorteile mit sich. Die Wahrscheinlichkeit, daß potentielle Anwender die Sprache bereits kennen, ist groß. Gute Entwicklungsumgebungen, ein schnell wachsender Markt vorgefertigter Bibliotheken, Schnittstellen zu allen objektorientierten Datenbanken und die Verfügbarkeit auf allen wichtigen Plattformen erleichtern unter anderem die Softwareentwicklung. Der große Erfolg dieser Sprache stellt sicher, daß wichtige Entwicklungen, wie

Analyse-Tools oder Code-Generatoren, auf jeden Fall C++-Anwendern zur Verfügung stehen werden.

### 4.2.1 Einfache Abstraktionen

Es hat sich gezeigt, daß ein guter Entwurf mit wenigen grundlegenden Abstraktionen auskommt. Meist ist es einfach, einige Kandidaten für eine solche Abstraktion im Problembereich zu finden. Schwieriger wird es dann schon, wenn man versucht, den Umfang einer Abstraktion genauer zu fassen und diese auf eine Klasse oder eine ganze Klassenhierarchie abzubilden. Dabei hat es sich gezeigt, daß es von Vorteil ist, möglichst einfache Abstraktionen zu finden und den Aufgabenumfang, der von einer Klasse erfüllt werden soll, auf eine spezielle Aufgabe zu beschränken. Mächtigere Abstraktionen lassen sich dann durch das Zusammenwirken von mehreren Klassen erreichen. Das Verständnis einer Klasse wird durch diese Vorgehensweise sehr erleichtert. Ein Fehler, der häufig gemacht wird, ist der Versuch, möglichst viele allgemeine Funktionen zur Verfügung zu stellen, um die Wiederverwendbarkeit von Klassen zu erhöhen. Zum einen kann aber niemand alle möglichen zukünftigen Anwendungen vorhersehen, so daß ein Entwurf selten alle denkbaren Fälle abdecken wird. Zum anderen werden oft nur Teile einer Abstraktion benötigt. Eine zu mächtige Klasse kann dann nicht sinnvoll wiederverwendet werden. Abgesehen davon erschwert eine umfangreiche Schnittstelle mit sehr vielen Methoden das Verständnis der zugrundeliegenden Abstraktion. Im Gegensatz dazu ist es bequem möglich, mehrere einfache Abstraktionen zu einer neuen, mächtigeren Abstraktion zusammenzufassen. Es ist deshalb sinnvoll, nur die Funktionalität zur Verfügung zu stellen, die für eine gegebene Abstraktion wirklich notwendig ist. Im Zweifelsfall läßt sich eine Klasse immer leichter erweitern als in ihrer Funktionalität einschränken.

### 4.2.2 Abstrakte Basisklassen

Unter einer abstrakten Basisklasse versteht man eine Klasse, die nur die Schnittstellen einer Abstraktion definiert, jedoch keine Implementierung vorgibt. Je nach Einsatzgebiet wird die Funktionalität in den abgeleiteten Klassen unterschiedlich implementiert. So kann z.B. ein Ereigniskalender abhängig von der Anzahl zu erwartender Einträge als einfache lineare Liste, als Baum oder als Hash-Tabelle implementiert werden. Da die Objekte nur über die von der abstrakten Basisklasse vorgegebenen Schnittstellen manipuliert werden, können Objekte mit unterschiedlicher Implementierung später problemlos ausgetauscht werden. Die Kopplung zwischen Klassen wird auf diese Weise stark verringert. Ein weiterer Vorteil kommt beim Entwurf zum Tragen. Wenn man die Schnittstellen einer Klasse problemorientiert definiert, ohne bereits die Implementierung vor Augen zu haben, erhält man meist wesentlich robustere Abstraktionen. Obwohl die Implementierung der Simulationsbibliothek mehrmals, teilweise drastisch geändert wurde, waren die Auswirkungen oft nur lokal zu spüren, da die Schnittstel-

len relativ stabil blieben. Der Wert abstrakter Basisklassen wird leider bei vielen Entwürfen unterschätzt. Er kann deshalb nur nochmals hervorgehoben werden. Eine gute Einführung in die Thematik gibt Robert Martin in [41].

### 4.2.3 Beziehungen zwischen Klassen

Die Vererbung zwischen Klassen ist ein fundamentales Konzept der objektorientierten Sichtweise. Leider wird sie deshalb von vielen als Allheilmittel angesehen und überall eingesetzt. Beim Entwurf dieser Simulationsbibliothek hat sich jedoch gezeigt, daß es oft sehr viel sinnvoller sein kann, Funktionalität nicht zu ererben, sondern diese durch Delegierung von Aufgaben an andere Objekte erbringen zu lassen. Bei der Vererbung erhält man genau die Funktionalität der Basisklasse. Im anderen Fall kann man durch abstrakte Basisklassen und Polymorphismus die Funktionalität flexibel durch Austausch der Objekte anpassen. Durch entsprechende Schnittstellen ist dies sogar dynamisch zur Laufzeit des Programmes möglich.

Häufig werden virtuelle Methoden eingesetzt, um für jede abgeleitete Klasse unterschiedliche Werte darzustellen. Meist lassen sich Daten mit weniger Aufwand durch ein entsprechendes Feld in jedem Objekt repräsentieren. Jede Klasse kann dann das Feld individuell initialisieren. Virtuelle Methoden sollten nur dort verwendet werden, wo es durch unterschiedliches Verhalten der Klassen erforderlich ist. In [12, Kap. 1] diskutiert Tom Cargill diese Problematik ausführlich.

Man sollte beim Entwurf deshalb darauf achten, welche Beziehungen zwischen Klassen bestehen. Gute Beschreibungen dieser Beziehungen finden sich in [8, S.75 ff; 16, Kap. 6.4].

### 4.2.4 Klassenhierarchie

Prinzipiell gibt es zwei grundlegende Klassenhierarchien. Die bekannteste ist der Baum, bei dem alle Klassen von einer Oberklasse, der sog. Wurzel, abgeleitet sind. Diese Art der Klassenhierarchie ist bei Sprachen wie Smalltalk vorherrschend, da hier keine statische Typüberprüfung stattfindet. Man ist deshalb nicht von vornherein sicher, welcher Klasse ein bestimmtes Objekt angehört. Um die Eigenschaften der Klassen zur Laufzeit abfragen zu können, definiert das Wurzelobjekt die gemeinsame Funktionalität aller Objekte. Vor allem sog. Containerklassen werden so definiert, daß sie alle von der Wurzelklasse abgeleiteten Klassen verwalten können. Die Vererbungshierarchie wird so zur Wiederverwendbarkeit benutzt. Allerdings geht dabei der genaue Typ eines Objektes verloren und muß zur Laufzeit bestimmt werden. Für Sprachen mit strenger statischer Typprüfung wie Eiffel oder C++ ist es dagegen sinnvoller, die Klassenhierarchie wie einen Wald zu strukturieren, d.h. mehrere baumartige Hierarchien aufzubauen, die zusätzlich durch Mehrfachvererbung miteinander

vernetzt sein können, jedoch keine gemeinsame Wurzel haben. Dies führt im allgemeinen zu übersichtlicheren Hierarchien, da es meist nur 2 bis 3 Ebenen gibt, die zudem genau auf das Problem angepaßt sind. Die Kopplung zwischen den Hierarchien ist loser, was die Wiederverwendbarkeit erleichtert. Containerklassen werden bei diesem Ansatz als parametrisierte Klassen definiert, so daß keine gemeinsame Basisklasse erforderlich ist. Durch die Typüberprüfung des Compilers wird sichergestellt, daß alle Objekte vom gleichen Typ sind; eine Abfrage zur Laufzeit erübrigt sich daher. Unterschiedliche Klassenbibliotheken lassen sich einfacher zusammen verwenden, als wenn jede eine bestimmte Wurzel (oft mit dem gleichen Namen „Object") erwartet. Bei der baumartigen Hierarchie enthält jedes Objekt den Overhead der Wurzelklasse, auch wenn deren Funktionalität nicht benötigt wird. Insgesamt ergeben sich klare Vorteile, die für eine verteilte Klassenhierarchie ohne gemeinsame Wurzel sowie den Einsatz parametrisierter Klassen sprechen. In [22] werden anhand von Containerklassen die Vor- und Nachteile beider Ansätze diskutiert.

### 4.2.5 Konsistenz

Ein wichtiger Gesichtspunkt, der zum leichteren Verständnis und zur fehlerfreien Anwendung einer Klassenbibliothek beiträgt, ist die Konsistenz mit der sie entworfen wurde. Unter diesem Punkt lassen sich mehrere Aspekte zusammenfassen, die kurz erläutert werden sollen:

- **Einheitliche Schnittstellen**
  Diese sollten so gewählt werden, daß sie vollständig und in sich konsistent sind. Der Anwender sollte nicht durch Ausnahmen überrascht werden.
- **Einheitliche Konventionen**
  Innerhalb der Klassenbibliothek sollten einheitliche Konventionen definiert werden, z.B. muß dokumentiert werden, wer Objekte erzeugen und diese vernichten kann. Ähnliche Probleme stellen sich bei der Verwendung von Zeigern und Referenzen als Parameter bei Funktionsaufrufen. Auch hier ist das Ziel, möglichst durchgängige Konzepte zu finden, die keine Überraschungen bieten.
- **Namensgebung**
  Ein weiterer wichtiger Punkt betrifft die sinnvolle Namensgebung von Variablen, Klassen und Methoden. Sofern die Regeln vernünftig gewählt wurden, ist es zweitrangig, wie die Namenskonventionen genau aussehen. Hier hat jeder Programmierer seinen eigenen Stil. Wichtiger ist es, diese Regeln konsistent anzuwenden. Dem Anwender wird damit das Verständnis, selbst ohne zusätzliche Dokumentation, sehr erleichtert.

Die Konventionen, die bei dieser Simulationsbibliothek verwendet wurden, sind im Anhang A.2 zusammengefaßt.

## 4.3 Architektur

Dieses Kapitel beschreibt die Architektur, die der Simulationsbibliothek zugrunde liegt. In Bild 4.1 sind die Teile, aus denen ein Simulationsprogramm besteht, dargestellt. Im wesentlichen kann man zwei Blöcke unterscheiden. Zum einen die Simulationsunterstützung, in der alle Komponenten, die zur Verwaltung und zur Steuerung des Simulationsablaufes notwendig sind, zusammengefaßt werden; zum anderen das eigentliche Modell. Das Modell kann wiederum hierarchisch aus Teilmodellen und Modellkomponenten aufgebaut sein.

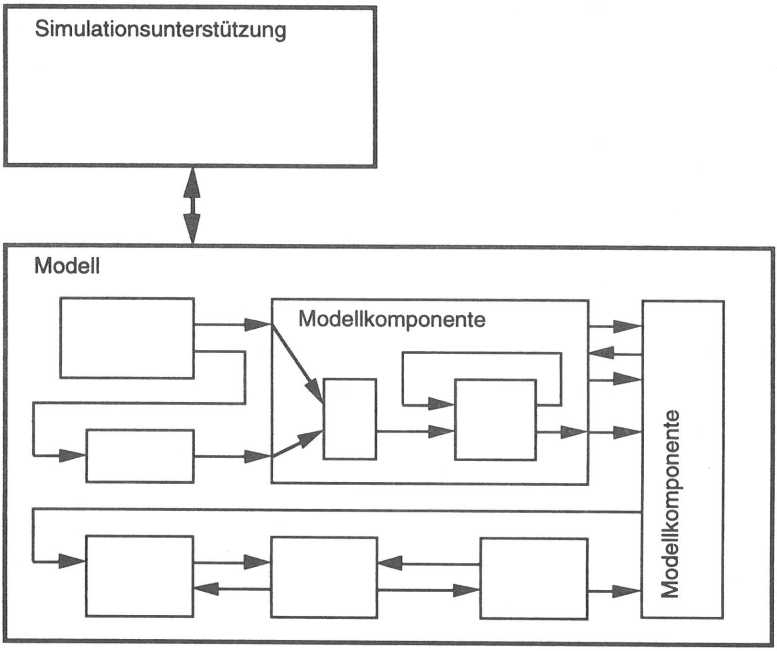



**Bild 4.1:** Architektur der Simulationsbibliothek

Modellkomponenten kommunizieren durch den Austausch von Nachrichten miteinander. Alle Nachrichten sind von einer abstrakten Nachrichtenklasse abgeleitet, die einige gemeinsame Eigenschaften, z.B. den Nachrichtentyp, definiert. Der weitere Inhalt der Nachrichten und ihre Bedeutung hängen vom jeweiligen Modell ab und sind nicht festgelegt. Jede Modellkomponente kann die Nachrichten nach eigenen Gesichtspunkten auswerten. Modellkomponenten verfügen über festgelegte Schnittstellen zur Außenwelt, sogenannte Ports, über die Nachrichten ausgetauscht werden können. Um zwei Modellkomponenten miteinander

kommunizieren zu lassen, müssen nur die entsprechenden Ein- und Ausgangsschnittstellen der Komponenten miteinander verbunden werden. Der Nachrichtenaustausch erfolgt durch Punkt-zu-Punkt-Verbindungen mit Hilfe eines Handshake-Protokolls, das sicherstellt, daß Nachrichten nur dann gesendet werden, wenn beide Instanzen dazu bereit sind. Eine Modellkomponente kann als Black-Box gesehen werden, deren Verhalten nach außen nur durch ihre Ports repräsentiert wird. Dieses Konzept erlaubt eine klare Trennung zwischen dem Verhalten von Modellkomponenten und ihrer strukturellen Anordnung innerhalb des Modells. Deshalb ist es jederzeit möglich, neue Modellkomponenten zwischen bestehende einzufügen, ohne an den bestehenden Modellkomponenten etwas ändern zu müssen. Modellkomponenten können wiederum aus anderen Modellkomponenten aufgebaut sein, wobei auch hier der Nachrichtenaustausch über die vorgegebenen Schnittstellen erfolgt. Modellkomponenten werden im nächsten Unterkapitel, der Nachrichtenaustausch in Kapitel 4.5 genauer beschrieben.

Bei der ereignisorientierten Simulation werden Ereignisse dazu verwendet, zukünftige Schritte mit Hilfe eines Kalenders vorzuplanen und diese später auszuführen. Man kann Ereignisse jedoch auch als eine weitere Möglichkeit sehen, um mit der Umgebung zu kommunizieren. Die Bedeutung eines Ereignisses hängt von der jeweiligen Modellkomponente ab. Ein Ereignis wird zur Bearbeitung an die Modellkomponente übergeben. Diese kann das Ereignis selbst bearbeiten oder es an die jeweils übergeordnete Modellkomponente weitergeben. Auf diese Weise wird automatisch eine hierarchische Bearbeitung von Ereignissen gewährleistet.

Ein Modell unterscheidet sich von anderen Modellkomponenten durch einen eingebauten Ereigniskalender. Es steht an der höchsten Hierarchieebene aller Modellkomponenten. Da ein Modell aus mehreren Teilmodellen bestehen darf, ist es auch möglich, mehr als nur einen Kalender zu verwenden. Verteilte Kalender werden aus Effektivitätsgründen oft eingesetzt. Diese müssen dann allerdings miteinander synchronisiert werden. Der gleiche Mechanismus könnte auch bei einer parallelen Version der Simulationsbibliothek von Nutzen sein. Die Ereignissteuerung wird in Kapitel 4.6 näher behandelt.

Die Simulationsunterstützung beinhaltet die eigentliche Simulationssteuerung sowie allgemeine Komponenten, die die Entwicklung von Simulationsprogrammen vereinfachen, wie z.B. ein modulares Ein- / Ausgabekonzept. Die Simulationssteuerung sorgt für den korrekten Ablauf der Simulation. Sie kontrolliert das Einlesen von Simulationsparametern, die Teilteststeuerung sowie das Ausgeben der Simulationsergebnisse. Die Simulationssteuerung ist modular aufgebaut. Durch Überschreiben einzelner Methoden lassen sich alle Aspekte der Steuerung an die individuellen Gegebenheiten auf einfache Weise anpassen.

Die wesentlichen Abstraktionen dieser Simulationsbibliothek sind eine hierarchische Modellbildung, standardisierte Schnittstellen zum Nachrichtenaustausch zwischen Modellkomponenten, eine hierarchische Ereignisverwaltung sowie die Simulationssteuerung. Die

klare Abgrenzung der Modellkomponenten führt dazu, daß Abhängigkeiten zwischen den Modellkomponenten normalerweise gering sind, was sich positiv auf die Wiederverwendbarkeit auswirkt. Diese grundlegenden Konzepte reichen bereits aus, eine gut strukturierte und dennoch universell einsetzbare Simulationsbibliothek aufzubauen. In den folgenden Unterkapiteln werden diese Konzepte verfeinert und detailliert beschrieben.

## 4.4 Modellkomponenten

### 4.4.1 Eigenschaften

Ein Simulationsmodell besteht aus Modellkomponenten, die miteinander vernetzt sind. Alle Modellkomponenten müssen von der Klasse *TEntity* abgeleitet sein, welche die grundsätzlichen Eigenschaften der Komponenten festlegt.

Jede Modellkomponente besitzt einen Namen, der beim Anlegen der Komponente frei gewählt werden kann. Hierarchische Modellkomponenten besitzen außer ihrem lokalen Namen einen globalen Namen, der sich aus der Aneinanderreihung des lokalen Namens und der Namen aller übergeordneten Modellkomponenten ergibt. Dies entspricht in etwa dem Pfadnamen einer verschachtelten Verzeichnisstruktur bei Dateisystemen. Der Name einer Modellkomponente dient hauptsächlich zur eindeutigen Identifizierung bei der Ausgabe von Ergebnissen und zur genauen Bestimmung von Fehlermeldungen während der Programmentwicklung.

*TEntity* besitzt außerdem Methoden zur Verwaltung der Standardschnittstellen. Ports können damit dem System global bekannt gemacht und miteinander verbunden werden. Alle Ports werden dazu in einem globalen internen Portmanager gespeichert. Die genaue Arbeitsweise wird in Kapitel 4.5 erläutert.

Die Ereignissteuerung erfolgt ebenfalls mit Methoden der Klasse *TEntity*. Ereignisse werden vom Programm zusammen mit dem geplanten Ereigniszeitpunkt der Modellkomponente zur Bearbeitung übergeben. Diese findet dann automatisch den zu diesem Ereignistyp passenden Event-Handler, der die eigentliche Bearbeitung bzw. die Vorplanung in einem Kalender vornimmt. Alle Event-Handler werden von einem internen Managerobjekt verwaltet. In Kapitel 4.6 wird die Ereignissteuerung ausführlich vorgestellt.

Bild 4.2 zeigt das Klassendiagramm der Klasse *TEntity*. Aus ihm kann man entnehmen, daß sich alle Objekte der Klasse *TEntity* bzw. einer von ihr abgeleiteten Klasse, genau je ein globales Objekt der Managerklassen *TPortManager* und *TEventHandlerManager* miteinander

teilen. Alle Modellkomponenten werden zusätzlich noch in einer globalen Liste gespeichert, die bei der Ereignisverwaltung benötigt wird. *TEntity* ist von der Klasse *TPrintServer* abgeleitet, da sie bereits die Grundfunktionalität zur Ausgabe von Modellkomponenten bereitstellt. Das Ausgabekonzept wird in Kapitel 4.11 vorgestellt. Da *TEntity* Ports, Ereignisse und Handler verwaltet, tauchen diese Klassen in der Schnittstellendefinition der Klasse und somit auch im Diagramm auf. Hier wird das in Kapitel 4.2.1 beschriebene Konzept, möglichst einfache Abstraktionen zu definieren, angewandt. Die Managerklassen haben jeweils eine fest umrissene Aufgabenstellung mit überschaubarer Komplexität. Die sehr viel kompliziertere Abstraktion einer Modellkomponente erfolgt durch das Zusammenwirken der Klasse *TEntity* mit den Managerklassen. Die Komplexität der Klasse *TEntity* verringert sich, da viele Aufgaben an die entsprechenden Managerklassen delegiert werden.

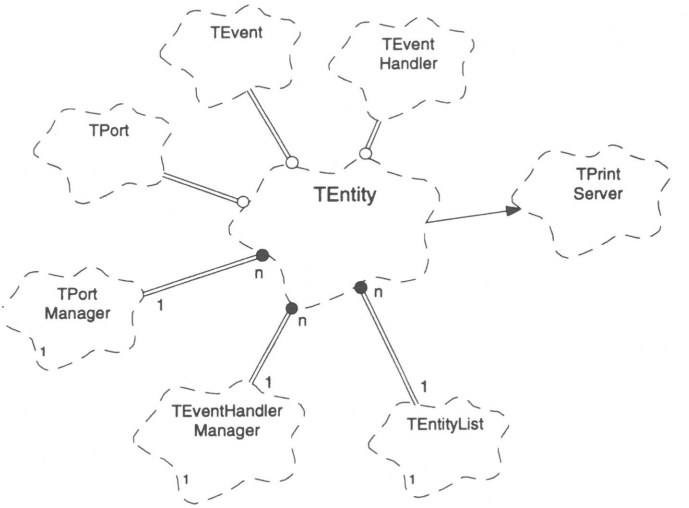



**Bild 4.2:** *TEntity* Klassendiagramm

Die Basismodellkomponente besitzt keinerlei Ein- / Ausgabeschnittstellen (Ports) und hat auch noch keine festgelegte Verhaltensweise. Beides wird erst in den abgeleiteten Klassen definiert. Die Basisklasse stellt somit nur ein Gerüst dar, aus dem sich allgemeine Modellkomponenten entwickeln lassen. Es handelt sich also um eine teilweise abstrakte Basisklasse, die nur die Schnittstellen für die wichtigsten Dienste zur Verfügung stellt. Je nach Aufgabenstellung sorgen unterschiedliche Implementierungen in den abgeleiteten Klassen für die notwendige Funktionalität. *TEntity* ist allerdings keine rein abstrakte Basisklasse, da sie für viele Methoden bereits Standardversionen zur Verfügung stellt. Trotzdem würde es keinen Sinn machen, Instanzen der Klasse *TEntity* in einem Simulationsprogramm zu verwenden, da

sie weder entsprechende Funktionalität, noch die benötigten Ports zur Kommunikation mit anderen Modellkomponenten besitzen.

Die Anzahl der Ports hängt von der jeweiligen Aufgabe der Modellkomponente ab, so besitzt eine Warteschlange normalerweise genau einen Eingangs- und einen Ausgangsport. Ein Multiplexer kann N Eingänge und einen Ausgang aufweisen. Es ist sogar möglich, die Anzahl der Ports dynamisch an die Anwendung anzupassen.

Modellkomponenten können auf verschiedene Arten miteinander kommunizieren. Modellkomponenten auf derselben Hierarchieebene sollten normalerweise nur über ihre Ein- / Ausgabeschnittstellen miteinander kommunizieren. Prinzipiell besteht zwar die Möglichkeit, daß einzelne Objekte direkt die Methoden einer anderen Modellkomponente aufrufen, dies führt jedoch zu einer engen Kopplung zwischen den Komponenten, was die Wiederverwendbarkeit stark einschränkt. Daher sollte der direkte Methodenaufruf vermieden werden. In den meisten Fällen kann das gleiche Ergebnis auch durch den Austausch von Nachrichten über die normalen Schnittstellen in viel allgemeinerer Form erreicht werden.

Für die Kommunikation zwischen Modellkomponenten, die hierarchisch geordnet sind, gibt es mehrere Möglichkeiten, die im nächsten Unterkapitel näher beleuchtet werden.

### 4.4.2 Hierarchische Modellkomponenten

Wie bereits mehrfach angesprochen, stellt der hierarchische Aufbau von Modellen ein wichtiges Konzept zur Strukturierung und zur Verringerung der Komplexität dar. Entsprechend große Aufmerksamkeit wurde ihm beim Entwurf der Simulationsbibliothek gewidmet. Jede Modellkomponente kann aus mehreren Modellkomponenten aufgebaut sein, die wiederum intern aus beliebig vielen weiteren Modellkomponenten bestehen können. So entsteht letztendlich eine Baumstruktur, an deren Wurzel sich das eigentliche Modell befindet. Jedes Objekt der Klasse *TEntity* besitzt eine Referenz auf die ihr übergeordnete Modellkomponente. Die Bilder 4.3 - 4.5 zeigen anhand eines einfachen Beispiels wie ein Warteschlangenmodell auf Modellkomponenten bzw. in ein Objektmodell übertragen werden kann. Dabei wurde die Kombination aus Warteschlange und Bedieneinheit, wie sie in Bild 3.2 zu sehen ist, zu einem Netzknoten zusammengefaßt. An diesem Beispiel ist auch die hierarchische Namensgebung deutlich zu sehen. Während die Namen der übergeordneten Modellkomponenten noch frei gewählt werden können, werden die Namen der tieferliegenden Komponenten bereits bei der Erstellung der hierarchischen Modellkomponenten festgelegt und können nachträglich nicht mehr ohne weiteres geändert werden. Dies ist auch nicht notwendig, da z.B. ein Netzknoten immer aus einer Warteschlange und einer Bedieneinheit besteht. Eine bestimmte Warteschlange ist durch ihren hierarchischen Namen, der den Netzknoten enthält, eindeutig gekennzeichnet.

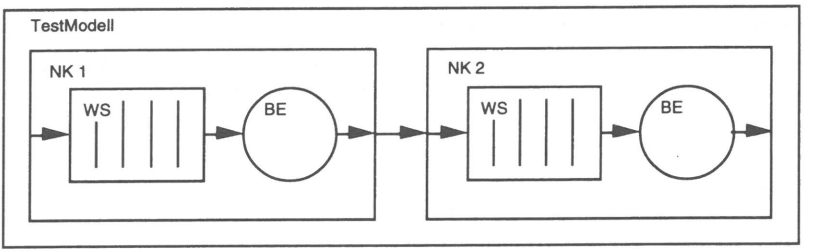



**Bild 4.3:** Beispiel eines Modells eines einfachen Warteschlangennetzes

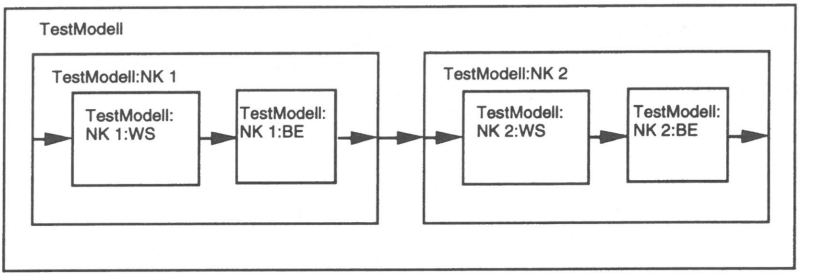



**Bild 4.4:** Aufteilung des Modells in hierarchische Modellkomponenten

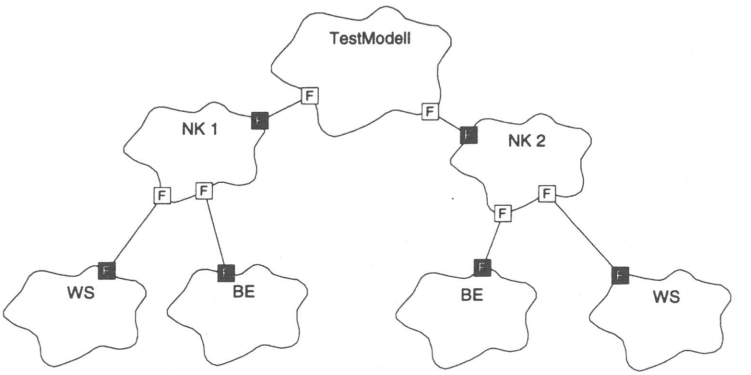



**Bild 4.5:** Darstellung des Modells als Objektdiagramm

Im Sinne der objektorientierten Vorgehensweise **ist** eine hierarchische Modellkomponente einerseits eine Modellkomponente, weshalb sie von *TEntity* abgeleitet sein muß, andererseits **enthält** sie weitere Modellkomponenten, aus denen sie aufgebaut ist. Letztere können z.B. als ganz normale Felder in der Klassendeklaration auftauchen. Der Konstruktor der übergeordneten Modellkomponente initialisiert alle Instanzen der enthaltenen Komponenten. Ein Beispiel dazu findet sich am Ende dieses Kapitels. Zusätzlich können hierarchische Modellkomponenten dynamisch erzeugt werden. Dies ist z.B. notwendig, um Modelle in Abhängigkeit von Eingabeparametern aufzubauen. In diesem Fall werden die Modellkomponenten von der übergeordneten Modellkomponente dynamisch erzeugt, initialisiert und z.B. in einer Liste gespeichert.

Natürlich reicht es nicht aus, die Modellkomponenten nur hierarchisch aufzubauen; wichtig ist vor allem, wie hierarchische Komponenten zusammenwirken. Ein bedeutender Gesichtspunkt dabei ist die Kopplung zwischen Modellkomponenten in verschiedenen Hierarchieebenen. Um die Wiederverwendbarkeit verschiedener Modellkomponenten nicht einzuschränken, sollte die Kopplung zwischen Komponenten nur so eng wie absolut notwendig sein. Dies kann durch die Beachtung einiger einfacher Regeln erreicht werden.

Jede Komponente bekommt bei der Initialisierung eine Referenz auf die ihr übergeordnete Modellkomponente geliefert. Dies ermöglicht ihr, Dienste der übergeordneten Instanz in Anspruch zu nehmen. Da sie den wirklichen Typ der übergeordneten Instanz nicht kennt, können dies nur Dienste sein, die allgemein bekannt sind, d.h. die bereits in der *TEntity*-Klasse definiert wurden. Aufgrund der dynamischen Bindung vieler Methoden hängt das Ergebnis eines Aufrufs jedoch trotzdem vom wirklichen Typ der übergeordneten Modellkomponente ab. Auf diese Weise lassen sich viele Aufgaben delegieren, ohne den genauen Aufbau der höheren Hierarchieebenen zu kennen. Als Beispiel kann hier wieder der Aufbau eines hierarchischen Namens herangezogen werden. Wenn der Name einer Modellkomponente erfragt wird, läßt diese sich zunächst den Namen der übergeordneten Modellkomponente geben, d.h. sie delegiert den Aufruf an die nächsthöhere Instanz. Anschließend hängt sie ihren eigenen lokalen Namen an, um den vollständigen Namen zu erhalten. Das Ganze kann sich rekursiv über mehrere Ebenen erstrecken, ohne daß eine Komponente wissen muß, wie die Modellhierarchie wirklich aufgebaut ist. Auch das Erfragen der aktuellen Systemzeit läuft ähnlich ab. In einem sequentiellen Simulationsprogramm gibt es normalerweise nur eine globale Systemzeit. In diesem Fall würde eine globale Variable ausreichen. Bei verteilten Systemen kann es jedoch mehrere lokale Zeiten geben, die bei Bedarf miteinander synchronisiert werden müssen. Es wurde deshalb eine Methode *GetSystemTime* eingeführt, mit deren Hilfe man die Systemzeit erfragen kann. Die Standardversion der Methode *GetSystemTime* delegiert den Aufruf an die nächsthöhere Instanz, da sie die Systemzeit selbst nicht kennt. Erst die Modellkomponente, die auch den Ereigniskalender enthält, kann über die Systemzeit Auskunft geben. Sie überschreibt die Standardmethode mit einer Version, die die intern gespeicherte Zeit zurückliefert. Bei einer parallelen Version der Simulationsbibliothek könnte

eine spezielle Modellklasse beim Aufruf der Systemzeit gleichzeitig die Synchronisation zwischen Teilmodellen ermöglichen, ohne daß untergeordnete Modellkomponenten betroffen wären. Pessimistische Verfahren könnten auf diese Weise relativ einfach realisiert werden. Optimistische Verfahren müßten allerdings zusätzlich Unterstützung für Rollback-Verfahren bieten. Dieses Beispiel zeigt, wie durch dynamische Bindung von Methoden immer die richtige Aktion veranlaßt wird, ohne daß der wirkliche Typ der übergeordneten Instanz bekannt sein muß. Die Delegation von Teilen von Aufgaben an andere Objekte stellt ein wesentliches Konzept beim Entwurf eines objektorientierten Programmes dar. Es trägt nicht nur zur loseren Kopplung zwischen Objekten, sondern auch zur flexibleren Anpassung an neue Gegebenheiten bei. Im zweiten Fall wird einfach eine neue abgeleitete Klasse mit unterschiedlicher Implementierung entwickelt. Da die Aufrufschnittstelle gleich bleibt, müssen die Objekte nur ausgetauscht werden, um eine neue Funktionalität zu erhalten. Dies unterstreicht nochmals die Vorteile beim Einsatz von abstrakten Basisklassen, die in heutigen Entwicklungen häufig noch unterschätzt werden.

Während untergeordnete Komponenten nur allgemeine Methoden der übergeordneten Modellkomponente aufrufen dürfen, um das Prinzip der Datenkapselung nicht zu verletzen, kennt die übergeordnete Instanz ihren internen Aufbau. Sie kann deshalb ohne weiteres direkt Methoden der Modellkomponenten aufrufen, aus denen sie aufgebaut ist. Allerdings darf auch sie dabei keine Hierarchieebene überspringen, da sie der interne Aufbau weiter unten liegender Instanzen nichts angeht. Sofern die internen Komponenten einer Modellkomponente als private Felder innerhalb des Objektes deklariert wurden, wird allerdings bereits der Compiler einen derartigen Versuch unterbinden.

Außer dem direkten Methodenaufruf ist es auch möglich, Nachrichten zwischen Modellkomponenten auf benachbarten Hierarchieebenen über die Standardschnittstellen auszutauschen. Näheres dazu findet sich in Kapitel 4.5.

Wie bereits früher angedeutet, werden Ereignisse ebenfalls hierarchisch bearbeitet. Ein Ereignis wird zunächst an die übergeordnete Instanz übergeben. Diese kann das Ereignis direkt bearbeiten, das Ereignis verändern oder gleich an die nächsthöhere Instanz weitergeben. Auch hier wird also das Prinzip der Delegierung verwendet. Das genaue Vorgehen ist in Kapitel 4.6 beschrieben.

Zusammenfassend kann man sagen, daß der hierarchische Aufbau eines Modells viele neue Möglichkeiten eröffnet und einen sehr wichtigen Stellenwert innerhalb des Entwurfs der Simulationsbibliothek einnimmt. Insbesondere wurde darauf geachtet, daß der Informationsfluß von tieferen zu höheren Hierarchieschichten anonym verläuft, Modellkomponenten ihre übergeordneten Instanzen also nicht wirklich kennen müssen. Umgekehrt liegt das Wissen über den Aufbau tieferliegender Schichten vor, so daß hier Komponenten auch gezielt angesprochen werden können. Die Einhaltung dieses Konzepts sorgt für eine strenge Datenkapse-

lung und erlaubt deshalb die Wiederverwendung von Modellkomponenten, ja ganzen Teilmodellen, in unterschiedlichen Anwendungen, ohne daß weitere Anstrengungen unternommen werden müßten. Man kann deshalb ohne Übertreibung an dieser Stelle von einem zentralen Grundsatz beim Entwurf der Simulationsbibliothek sprechen.

### 4.4.3 Definition neuer Modellkomponenten

Alle neuen Modellkomponenten müssen von der Klasse *TEntity* abgeleitet werden. Normalerweise muß keine der geerbten Methoden überschrieben werden. Dies ist nur erforderlich, falls die Funktionalität der Standardversionen nicht ausreichend ist, z.B. bei der Implementierung einer dynamischen Anzahl von Ports. Entsprechende Hinweise finden sich in den jeweiligen Unterkapiteln.

In jeder Modellkomponente müssen die Schnittstellen nach außen definiert und das interne Verhalten programmiert werden. Hierarchische Modellkomponenten lassen sich einfach dadurch aufbauen, daß die internen Komponenten als Felder der neuen Instanz definiert und im Konstruktor initialisiert werden. Der folgende Ausschnitt zeigt die prinzipielle Definition der Modellkomponenten aus obigem Beispiel:

```
// Abstract base class
class TEntity {
public:
    // Constructor
    TEntity(const TString & name, TEntity * owner = 0);
...
};


// Queue
class TWarteschlange : public TEntity {
public:
    // Constructor
    TWarteschlange(const TString & name, TEntity * owner = 0) :
       TEntity(name, owner) {}       // Initialize base class
...
};


// Server
class TBedieneinheit : public TEntity {
public:
    TBedieneinheit (const TString & name, TEntity * owner = 0) :
       TEntity(name, owner) {}       // Initialize base class
...
};
```

```
// Network node
class TNetzknoten : public TEntity {
public:
    TNetzknoten (const TString & name, TEntity * owner = 0) :
       TEntity(name, owner),          // Initialize base class
       fWarteschlange("WS", this),   // Initialize embedded entities
       fBedieneinheit("BE", this)
    {
        // Connect internal ports…
    }
…
private:
    TWarteschlange    fWarteschlange;     // Embedded entity fields
    TBedieneinheit    fBedieneinheit;
};

// A model is a special entity that has a calendar
class TModel : public TEntity { … };

// Our model is a special model that contains two network nodes
class TTestModell : public TModel {
public:
    TTestModell (const TString & name, TEntity * owner = 0) :
       TModel(name, owner),     // Initialize base class
       fNK1("NK 1", this),      // Initialize embedded entities
       fNK2("NK 2", this)
    {
        // Connect internal ports…
    }
…
private:
    TNetzknoten       fNK1;    // Embedded network node entities
    TNetzknoten       fNK2;
};
```

Aus dem Code kann man erkennen, daß eine Warteschlange direkt von *TEntity* abgeleitet wird. In ihrem Konstruktor wird der Name und ein Zeiger auf eine eventuell vorhandene übergeordnete Instanz übergeben und mit diesen Werten die Oberklasse *TEntity* initialisiert. Für die Bedieneinheit erfolgt das Vorgehen analog. Beim Netzknoten zeigt sich dann, wie diese neuen Modellkomponenten wiederverwendet werden können. Der Netzknoten beinhaltet je eine Instanz der Klassen *TWarteschlange* und *TBedieneinheit*. Diese werden im Konstruktor mit einem festen Namen („WS" bzw. „BE") initialisiert. Außerdem wird ein Zeiger auf die eigene Instanz übergeben, so daß diese Modellkomponente von beiden als ihre übergeordnete Instanz erkannt wird. Die interne Verknüpfung der Ports müßte ebenfalls im Konstruktor erfolgen und wird im nächsten Kapitel behandelt. Analog zu diesem Vorgehen, wird das Modell aus mehreren Netzknoten aufgebaut. In einem realen Beispiel hätten die Konstruktoren der Modellkomponenten noch weitere Argumente, z.B. die Anzahl der Warteplätze oder die Verteilungsfunktion der Bedienzeit. Auf diese Weise entsteht ein Baukasten, der sich einfach erweitern und wiederverwenden läßt.

## 4.5 Verbindung von Modellkomponenten

### 4.5.1 Das Port-Konzept

Ports stellen die Schnittstellen dar, zwischen denen Nachrichten während der Simulation ausgetauscht werden können. Es werden Ein- und Ausgabeports unterschieden, um sicherzustellen, daß nicht versehentlich z.B. die Ausgabeports zweier Modellkomponenten miteinander verbunden werden. Bei allen Verbindungen handelt es sich um gerichtete Punkt-zu-Punkt-Verbindungen zwischen zwei Ports. Jeder Port hat einen eigenen lokalen Namen und gehört zu einer Modellkomponente. Der globale Name ergibt sich aus dem Namen der Modellkomponente und dem lokalen Namen des Ports (z.B. „TestModell:NK 1.Eingang"). Beim Anlegen eines Ports registriert sich dieser automatisch bei seiner Modellkomponente und wird vom globalen Portmanager gespeichert. Durch Angabe der Modellkomponente und des lokalen Portnamens kann somit auf jeden Port innerhalb des Systems zugegriffen werden. Damit diese Zuordnung eindeutig ist, müssen die Namen aller Ports **einer** Modellkomponente verschieden sein. Zur Verwaltung der Ports stehen u.a. folgende Methoden in der Klasse *TEntity* zur Verfügung:

```
static void RegisterPort(TPort &);

static void Connect(TEntity & fromEntity,
                    const TString & fromLocalPortName,
                    TEntity & toEntity,
                    const TString & toLocalPortName);

Boolean IsPortKnown(const TString & localPortName) const;

void AliasPort(TEntity & originalPortEntity,
               const TString & originalLocalPortName,
               const TString & aliasLocalPortName);

void RenamePort(const TString & oldPortName,
                const TString & newPortName);
```

*RegisterPort* macht einen Port dem System bekannt. Diese Methode wird beim Anlegen eines Ports automatisch aufgerufen. Zwei Ports können mit der *Connect*-Methode miteinander verbunden werden. Dabei wird geprüft, ob eine solche Verbindung überhaupt zulässig ist. Diese Aufgaben werden von der *TEntity*-Klasse nicht direkt ausgeführt, sondern intern an das globale *TPortManager*-Objekt delegiert, so daß eine klare Trennung der Aufgaben stattfindet. *IsPortKnown* erlaubt die Abfrage, ob ein Port mit dem vorgegebenen Namen existiert. Die *Connect*-Methode fragt mit dieser Methode ab, ob beide Ports existieren, bevor eine Verbindung aufgebaut wird. Sofern eine Modellkomponente eine dynamische Anzahl von Ports benötigt, kann sie *IsPortKnown* überschreiben und jedesmal, wenn ein unbekannter Port abgefragt wird, diesen dynamisch erzeugen. *AliasPort* erlaubt die Verwendung eines Ports

unter neuem Namen und wird im übernächsten Unterkapitel genauer beschrieben. Mit *RenamePort* kann der Name eines Ports nachträglich geändert werden. Nachrichten zwischen Ports werden mit Hilfe eines Handshake-Protokolls übertragen, das im nächsten Unterkapitel behandelt wird.

Alle Nachrichten müssen von der Klasse *TMessage* abgeleitet sein. Der Nachrichteninhalt hängt von dem zu simulierenden Problem ab und wird in abgeleiteten Klassen definiert. Allgemeine Modellkomponenten wie z.B. Warteschlangen brauchen den genauen Nachrichtentyp nicht zu kennen und manipulieren allgemeine Objekte vom Typ *TMessage*. Andere Modellkomponenten werten den Inhalt der Nachricht aus und müssen dazu den genauen Nachrichtentyp wissen. Die Klasse *TMessage* bietet deshalb eine Methode zum Erfragen des Typs einer Nachricht. Sobald die vorgesehene Einführung von Laufzeitinformationen in C++ verfügbar ist, kann die Typabfrage vom Compiler unterstützt werden [60].

Der Nachrichtenaustausch zwischen Ports und Modellkomponenten wird in Kapitel 4.5.3 näher erläutert. Die Funktionalität von Ein- und Ausgangsports ist sehr ähnlich. Es hat sich gezeigt, daß es in diesem Fall sinnvoller ist, die gesamte Funktionalität in der gemeinsamen Basisklasse *TPort* zu implementieren und die wenigen Ausnahmen, z.B. die Prüfung, ob eine Verbindung zulässig ist, durch eine Typabfrage zu behandeln. Die abgeleiteten Klassen dienen hauptsächlich dazu, die Typüberprüfung durch den Compiler zu ermöglichen.

### 4.5.2 Das Verbindungskonzept

Bild 4.6 zeigt das Handshake-Protokoll zwischen zwei Ports anhand eines Objektdiagramms.


**Bild 4.6:** Handshake-Protokoll zum Nachrichtenaustausch zwischen Ports

Der Austausch von Nachrichten zwischen zwei Ports läuft folgendermaßen ab: Nachdem einem Ausgabeport eine Nachricht zur Übermittlung übergeben wurde, ruft dieser die *MessageIndication*-Methode des Empfängerports auf. Dadurch wird der empfangenden Modellkomponente mitgeteilt, daß eine neue Nachricht zur Verfügung steht. Es steht der empfangenden Komponente frei, ob sie die Nachricht sofort annimmt oder diese erst zu einem späteren Zeitpunkt bearbeiten will. Im letzteren Fall wäre der sendende Port bis zur endgültigen Abnahme blockiert. Soll die Nachricht gleich bearbeitet werden, so ruft der

Empfangsport die *GetMessage*-Methode des Sendeports auf. Will der Empfänger zu einem späteren Zeitpunkt wissen, ob der Sender noch eine Nachricht zum Versenden hat, so kann er *IsMessageAvailable* aufrufen. *GetMessage* darf nur direkt innerhalb der *MessageIndication*-Methode oder nach einem erfolgreichen *IsMessageAvailable*-Aufruf aufgerufen werden. Andernfalls bricht das Programm mit einer Fehlermeldung ab.

Ein kurzes Beispiel soll das Protokoll verdeutlichen. Sobald die erste Nachricht in der Warteschlange eintrifft, wird sie mit *MessageIndication* der nachfolgenden Bedieneinheit angeboten. Sofern die Bedieneinheit nicht belegt ist, kann sie die Nachricht sofort abholen. Falls sie gerade eine frühere Nachricht bearbeitet, wird der Aufruf einfach ignoriert. Am Ende eines Bedienprozesses fragt die Bedieneinheit bei der Warteschlange mit *IsMessageAvailable* nach, ob eine weitere Nachricht zur Bearbeitung bereitsteht. Wenn dies der Fall ist, wird sie durch *GetMessage* abgeholt. Im anderen Fall geht sie in den nichtbelegten Zustand über. Sobald eine neue Nachricht eintrifft, wird sie automatisch durch den *MessageIndication*-Aufruf geweckt. Durch mehrmaliges Wiederholen von *IsMessageAvailable* und *GetMessage* können auch mehrere Nachrichten auf einmal aus der Warteschlange entfernt werden.

Durch dieses einfache Handshake-Verfahren können Nachrichten zwischen Ports ausgetauscht werden, ohne daß die beteiligten Modellkomponenten wissen müssen, wie die Nachrichten erzeugt oder verarbeitet werden, oder ob eine Komponente in einem Zustand ist, in der sie neue Nachrichten akzeptieren kann. Die sehr lose Bindung zwischen zwei Ports ermöglicht es, neue Modellkomponenten zwischen bestehende einzufügen, ohne die Übertragung von Nachrichten zu stören. So könnte z.B. eine Komponente, die übertragene Meldungen zählt, völlig transparent zwischen Warteschlange und Bedieneinheit eingefügt werden. Dies erhöht die Flexibilität beträchtlich, bestehende Modelle durch Einfügen neuer Komponenten an neue Gegebenheiten anzupassen.

Im Gegensatz zu Ansätzen, bei denen einmal verschickte Nachrichten auf jeden Fall vom Empfänger abgenommen werden müssen (vgl. z.B. Kapitel 3.5.2, DOSE), erlaubt die Entkopplung von Anbieten und Versenden einer Nachricht auch die Möglichkeit, daß die Nachricht erst später abgenommen wird, der Sender also blockiert ist. Wo dies nicht erlaubt ist, z.B. bei einem Nachrichtengenerator, muß die Modellkomponente eine entsprechende Fehlermeldung generieren. Wäre diese Möglichkeit nicht vorhanden, so könnten Modellkomponenten nur so entworfen werden, daß zwischen ihnen keine Blockierung auftreten kann. Eine nachfolgende Modellkomponente hat bei diesem Ansatz keine Möglichkeit, sich gegen das Senden weiterer Nachrichten zu wehren. Ein Aufspalten von Warteschlange und Bedieneinheit in unabhängige Modellkomponenten wäre nicht zulässig, da eine ankommende Nachricht von der Warteschlange sofort weitergesendet würde, obwohl die Bedieneinheit noch mit der Bearbeitung der vorhergehenden Nachricht beschäftigt wäre. Warteschlange und Bedieneinheit müssen deshalb eng gekoppelt in einer Modellkomponente untergebracht werden (was im DOSE-Beispiel in [38] auch gut zu sehen ist). Das hier vorgestellte Protokoll erlaubt die

flexible Unterteilung in eigenständige Modellkomponenten und besitzt keine derartigen Einschränkungen.

Die Flexibilität des Protokolls soll am Beispiel eines Multiplexers, der zwischen eine Warteschlange und mehrere Bedieneinheiten gesetzt wird, demonstriert werden. Immer wenn er eine Nachricht von der Warteschlange angeboten bekommt, bietet er sie nach einer festgelegten Strategie den Bedieneinheiten an. Die erste freie Bedieneinheit nimmt die Nachricht durch Aufruf von *GetMessage* an. Weder die Warteschlange, noch die Bedieneinheit müssen für diesen Vorgang angepaßt werden. Allerdings darf sich eine beschäftigte Bedieneinheit nicht merken, daß ihr eine Nachricht angeboten wurde, sonst würde sie später *GetMessage* aufrufen, obwohl die fragliche Nachricht längst einer anderen Bedieneinheit übergeben wurde. Es ist deshalb wichtig, daß alle Modellkomponenten das Protokoll einhalten und durch den vorherigen Aufruf von *IsMessageAvailable* prüfen, ob eine Nachricht vorliegt.

Das Port-Konzept erlaubt es, beliebige Modellkomponenten miteinander zu verbinden, sofern diese über die notwendigen Ports verfügen. Ob eine solche Verbindung sinnvoll ist, hängt vom modellierten System ab und kann von der Simulationsbibliothek nicht überprüft werden. Diese Aufgabe übernimmt weiterhin der Anwender, obwohl man sich für spezielle Aufgabenstellungen Werkzeuge vorstellen könnte, die in der Lage sind, eine semantische Prüfung eines Modells vorzunehmen.

### 4.5.3 Zusammenwirken von Modellkomponenten und Ports

Bisher wurde nur der Nachrichtenaustausch zwischen zwei Ports betrachtet. Im folgenden soll nun erläutert werden, wie die Nachrichten zwischen Modellkomponenten und Ports ausgetauscht werden. Außerdem soll die Übergabe von Nachrichten zwischen hierarchischen Modellkomponenten betrachtet werden.

Portobjekte werden normalerweise als Felder in einer Modellkomponente angelegt. Die Modellkomponente kennt deshalb alle ihre Ports und hat direkten Zugriff auf sie. Eine Nachricht wird einem Ausgabeport durch den Aufruf der *MessageIndication*-Methode angeboten. Dieser ruft automatisch dieselbe Methode im Empfangsport auf. Nun stellt sich allerdings die Frage, wie die empfangende Modellkomponente von diesem Angebot unterrichtet wird. Eine Möglichkeit würde darin bestehen, eine virtuelle Methode der *TEntity*-Klasse aufzurufen. Dies hätte den Nachteil, daß die gleiche Methode von allen Empfangsports aufgerufen würde. Deshalb wird eine neue Klasse *TMessageHandler* eingeführt, die als Bindeglied zwischen Port und Modellkomponente dient. Entsprechend der verschiedenen Ein- / Ausgabeports werden auch *TInputMessageHandler* und *TOutputMessageHandler* unterschieden, die beide von *TMessageHandler* abgeleitet sind. Jeder Port besitzt eine Methode *SetMessageHandler*, mit der ein Message-Handler mit dem Port assoziiert werden kann. Wenn einem

Empfangsport eine Nachricht angeboten wird, so ruft dieser die *HandleMessageIndication*-Methode des Message-Handlers auf. Wie dieser auf das Angebot reagiert, kann durch Überschreiben der Methode in einer abgeleiteten Klasse festgelegt werden. Normalerweise wird er eine vorher festgelegte Methode der Modellkomponente aufrufen. Da für jeden Port ein eigener Handler angegeben werden kann, lassen sich für jeden Port auch unterschiedliche Methoden in der Modellkomponente aufrufen.

Wenn man Bild 4.4 betrachtet, findet man einerseits Verbindungen, die auf der gleichen Hierarchieebene verlaufen, z.B. zwischen Warteschlange und Bedieneinheit bzw. zwischen den Netzknoten. Diese können mit den gerade besprochenen Methoden behandelt werden. Andererseits gibt es auch Verbindungen auf benachbarten Hierarchieebenen. So muß z.B. eine Nachricht, die an einem Netzknoten ankommt, an den Empfangsport der Warteschlange weitergegeben werden. Mit den bisherigen Mitteln müßte der Netzknoten einen *TInputMessage-Handler* installieren, der die Nachricht an seinem Empfangsport entgegennimmt und sie direkt an den Warteschlangenport übergibt. Dies wäre mit unnötigem Aufwand verbunden, da der Netzknoten nur zur Datenkapselung dient und keine eigene Funktionalität besitzt, er also an den Nachrichten gar nicht interessiert ist. Um diesen Fall zu optimieren, ist es deshalb erlaubt, zwei Eingangs- bzw. Ausgangsports, die auf benachbarten Hierarchieebenen stehen, direkt miteinander zu verbinden. Der Eingangsport des Netzknotens reicht empfangene Nachrichten in diesem Fall unmittelbar an den Eingangsport der Warteschlange weiter. Natürlich kann sich dieser Vorgang rekursiv über mehr als zwei Hierarchieebenen erstrecken. Eine zusätzliche Optimierung stellt die Möglichkeit dar, von außen direkt auf einen internen Port zuzugreifen. Dazu gibt es in der *TEntity*-Klasse die Funktion *AliasPort*, die einen internen Port als virtuellen Port der umschließenden Modellkomponente der Öffentlichkeit zugänglich macht. Eine Verbindung zu einem Netzknoten spricht dann direkt den Port der Warteschlange an. Alle drei geschilderten Verfahren verhalten sich nach außen hin gleich, die Datenkapselung bleibt gewahrt. Es bleibt dem Entwickler einer Modellkomponente überlassen, welche Methode er anwendet. Sogar nachträgliche Änderungen bleiben ohne Auswirkungen auf das übrige Programm.

Zusammenfassend kann man folgende Regeln für das Verbinden von Ports aufstellen:

- Ein- und Ausgangsports können nur auf derselben Hierarchieebene verbunden werden. Dabei können beide Ports zu verschiedenen oder zur gleichen Modellkomponente gehören, so daß auch Rückkopplungen erlaubt sind.
- Zwei Eingabe- bzw. zwei Ausgabeports dürfen nur verbunden werden, wenn sie zu benachbarten Hierarchieebenen gehören und eine Modellkomponente die andere enthält.
- Mit *AliasPort* ist es möglich, einen Port einer enthaltenen Modellkomponente auf der nächst höheren Hierarchieebene als externen Port direkt anzusprechen.

Das folgende Beispiel zeigt, welche Erweiterungen an Warteschlange und Netzknoten vorgenommen werden müssen, um Ports in das bereits bekannte Beispiel zu integrieren. Die übrigen Modellkomponenten werden analog dazu geändert.

```
class TWarteschlange : public TEntity {
public:
    // Constructor now initializes ports and sets message handlers
    TWarteschlange(const TString & name, TEntity * owner = 0) :
       TEntity(name, owner),
       fInputPort(*this, "In"), fOutputPort(*this, "Out")
    {
        fInputPort.SetMessageHandler(fInputMessageHandler);
        fOutputPort.SetMessageHandler(fOutputMessageHandler);
    }
...
private:
    TInputPort                  fInputPort;       // Input port
    TWSInputMessageHandler      fInputMessageHandler; // Message handler
    TOutputPort                 fOutputPort;      // Same for output
    TWSOutputMessageHandler     fOutputMessageHandler;
};

class TNetzknoten : public TEntity {
public:
    TNetzknoten (const TString & name, TEntity * owner = 0) :
       TEntity(name, owner),          // Base class constructor
       fOutputPort(*this, "Output"), // Output port constructor
       fWarteschlange("WS", this),    // Embedded entities constructors
       fBedieneinheit("BE", this)
    {
         // Make internal input port of fWarteschlange public
         this->AliasPort(fWarteschlange, "In", "Input");
         // Connect internal ports
         this->Connect(fWarteschlange, "Out", fBedieneinheit, "In");
         this->Connect(fBedieneinheit, "Out", *this, "Output");
    }
private:
    TOutputPort       fOutputPort;        // Output port
    TWarteschlange    fWarteschlange;     // Embedded entities
    TBedieneinheit    fBedieneinheit;
};

class TTestModell : public TModel {
public:
    TTestModell (const TString & name, TEntity * owner = 0) :
       TModel(name, owner),                // Base class constructor
       fNK1("NK 1", this), fNK2("NK 2", this)
    {
         // Connect internal ports
         this->Connect(fNK1, "Output", fNK2, "Input");
    }
...
private:
    TNetzknoten       fNK1;    // Embedded entities
    TNetzknoten       fNK2;
};
```

Die Warteschlange erhält je einen Eingangs- und Ausgangsport sowie die dazugehörigen Message-Handler. Im Konstruktor werden die Namen und die zugehörige Modellkomponente der Ports festgelegt und die Message-Handler der Ports gesetzt. Die Methoden, die beim Senden und Empfangen von Nachrichten von den Message-Handlern aufgerufen werden, sind nicht gezeigt. Beim Netzknoten verhält es sich ähnlich. Im Konstruktor erfolgt der Verbindungsaufbau zwischen den internen Modellkomponenten Warteschlange und Bedieneinheit. Der Ausgangsport der Bedieneinheit wird mit dem Ausgangsport des Netzknotens verbunden. Mit *AliasPort* wird der Eingangsport der Warteschlange unter einem neuen Namen als Eingangsport des Netzknotens bekannt gemacht. Beim Aufbau des Modells wird dieser Name zum Verbinden der Netzknoten verwendet. In beiden Fällen sind keine zusätzlichen Message-Handler erforderlich, da die Nachrichten nur transparent durchgereicht werden. Diese Funktionalität wird bereits durch die direkte Verkettung von Ports ermöglicht. Insgesamt kann man sehen, daß der Aufwand zur Vernetzung eines Modells relativ gering ist. Da jede Modellkomponente ihre interne Vernetzung selbst durchführt, müssen in höheren Hierarchieebenen nur die Verbindungen zwischen den jeweils neuen Modellkomponenten betrachtet werden.

Um die Programmierung der Message-Handler zu erleichtern, gibt es mehrere parametrisierte Klassen. Die Standard-Message-Handler (*TStdInputMessageHandler* bzw. *TStdOutputMessageHandler*) rufen eine feste Methode der Modellkomponente auf. Sie sind immer dann geeignet, wenn pro Modellkomponente nur ein Eingangs- und Ausgangsport vorhanden sind. Sie benötigen im Konstruktor eine Referenz auf die Modellkomponente, bei der der Methodenaufruf stattfinden soll. Die allgemeinen Message-Handler (*TGenInputMesssageHandler* bzw. *TGenOutputMessageHandler*) lassen zusätzlich noch die Angabe der Methode zu, so daß für jeden Port eine andere Methode aufgerufen werden kann. Somit ist es nur für ganz spezielle Anforderungen notwendig, eine eigene Klasse von der Basis-Handlerklasse abzuleiten. Dieses abgestufte Konzept hat sich an mehreren Stellen der Simulationsbibliothek bewährt, da es die Benutzung der Bibliothek vereinfacht, ohne die Flexibilität einzuschränken.

Bild 4.7 zeigt abschließend die Übergabe einer Nachricht von Netzknoten 1 an Netzknoten 2. Um die Übersichtlichkeit zu erhöhen, wurde die Reihenfolge der Methodenaufrufe durchnumeriert. Ausgehend von der Bedieneinheit wird die Nachricht mit *MessageIndication* nacheinander dem eigenen Ausgabeport, dem Port des 1. Netzknotens und schließlich dem Eingangsport der Warteschlange des 2. Netzknotens angeboten. Dort wird die *HandleMessageIndication*-Methode des Empfangs-Handlers aufgerufen, der die Nachricht an die Modellkomponente weitergibt bzw. sie selbst bearbeitet. Im positiven Fall nimmt die Komponente die Nachricht mit *GetMessage* an. Dieser Aufruf geht den umgekehrten Weg, bis er über den Sende-Handler bei der Bedieneinheit eintrifft, die daraufhin die Nachricht zur Verfügung stellt. Man erkennt deutlich den streng hierarchischen Ablauf. Durch die Verwendung von *AliasPort* muß der Netzknoten 2 keinen eigenen Eingangsport besitzen und die Nachricht trifft direkt am Port der Warteschlange ein. Die gleiche Verbesserung könnte am

Ausgangsport des Netzknotens 1 vorgenommen werden. Dadurch ließen sich mehrere Methodenaufrufe des Handshake-Protokolls einsparen, weshalb diese Optimierung immer durchgeführt werden sollte. Da dieser Fall bei hierarchischen Modellen mehrfach vorkommen kann, wurde die Implementierung durch ein Cache-Verfahren optimiert. Alle Ports, die Nachrichten nur an den nächsten Port weiterleiten, werden dabei automatisch übersprungen. Mit dieser Methode lassen sich die gleichen Laufzeitvorteile wie durch Verwendung von *AliasPort* erreichen. *AliasPort* besitzt allerdings darüber hinaus den Vorteil, daß kein eigener Port allokiert und initialisiert werden muß, was die Verwaltung vereinfacht und Speicherplatz spart.

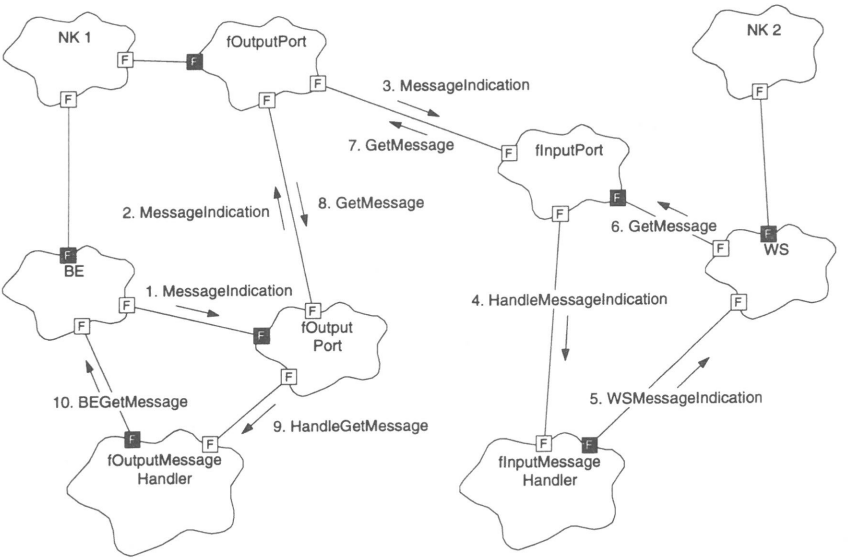



**Bild 4.7:** Austausch einer Nachricht zwischen zwei Modellkomponenten

### 4.5.4 Nachrichtenfilter

Für bestimmte Aufgaben wäre es oft ausreichend, wenn man einen Blick auf die verschickten Nachrichten werfen oder ein Attribut einer Nachricht ändern könnte. Ein Beispiel wäre ein Durchflußzähler, der eine Statistik über alle Nachrichten eines bestimmten Typs führt. In solchen Fällen ist es nicht gerechtfertigt, eine eigene Modellkomponente mit Ports und Message-Handlern zu entwerfen. Deshalb wurde die Möglichkeit geschaffen, in jedem Port sog. Nachrichtenfilter zu installieren. Diese Filter werden für jede Methode des Handshake-Protokolls aufgerufen, bevor die Methode des nächsten Ports bzw. der normale Message-

Handler aufgerufen werden. Der Filter kann die Nachricht auswerten oder manipulieren. Die Methoden des Basisfilters geben die Nachricht einfach an den nächsten Filter weiter, bzw. der letzte Filter gibt sie an den Port zurück. Es ist deshalb nur erforderlich, diejenigen Methoden zu überschreiben, die auch gefiltert werden sollen. Mit der *AddMessageFilter*-Methode können beliebig viele Filter in jedem Port installiert werden. Die Reihenfolge, in der Filter installiert werden, spielt keine Rolle, da jeder Filter die entsprechende Methode des nächsten Filters aufrufen muß. Dies wird am einfachsten dadurch erreicht, daß nach Abschluß der eigenen Bearbeitung die Methode des Basisfilters aufgerufen wird.

Ein Zähler würde z.B. nur die *HandleGetMessage*-Methode des Filters überschreiben, um alle Nachrichten eines bestimmten Typs zu zählen. Bild 4.8 zeigt den Ablauf, wenn am Eingangsport der Warteschlange zwei zusätzliche Filter installiert wurden. Insgesamt eröffnet dieses Konzept vielfältige Möglichkeiten, den Nachrichtenfluß in einem Simulationsmodell zu untersuchen, ohne Eingriffe in bestehende Modellkomponenten vornehmen zu müssen, da sich die Filter auch nachträglich installieren lassen.

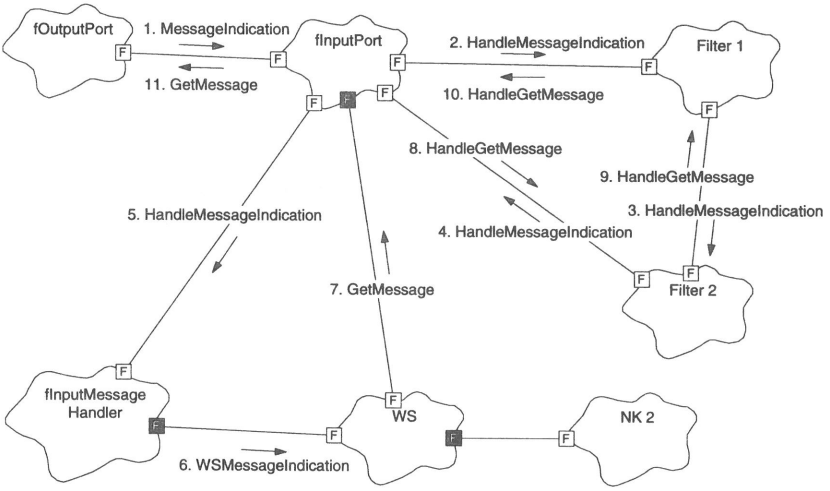



**Bild 4.8:** Nachrichtenaustausch mit installierten Nachrichtenfiltern

Ähnlich wie bei den Message-Handlern gibt es auch hier eine parametrisierte Klasse, die das Schreiben von Filtern wesentlich erleichtert. Die Klasse *TGenMessageFilter* erlaubt die Angabe der Methoden des Handshake-Protokolls, die gefiltert werden sollen. Zum Abschluß dieses Kapitels sind alle Klassen des Port-Konzeptes in Bild 4.9 zusammengefaßt.

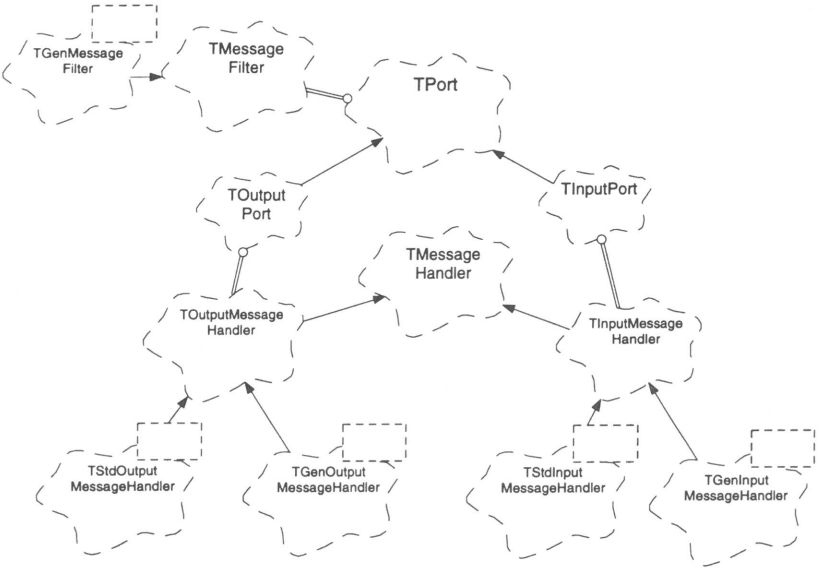



**Bild 4.9:** Klassendiagramm des Port-Konzeptes

## 4.6 Ereignissteuerung

Normalerweise läuft die Ereignissteuerung so ab, daß eine Modellkomponente ein zukünftiges Ereignis zusammen mit dem Ereigniszeitpunkt an den Kalender übergibt. Sobald die Simulationszeit mit dem Ereigniszeitpunkt übereinstimmt, wird das Ereignis aus dem Kalender ausgetragen und bearbeitet. In der hier vorgestellten Simulationsbibliothek wird dieses Konzept um die Möglichkeit der hierarchischen Ereignisbearbeitung erweitert.

### 4.6.1 Ereignisse

Kern der Ereignisverarbeitung bilden die eigentlichen Ereignisse. *TEvent* stellt eine Basisklasse dar, deren Methoden *HandleProcessEvent* und *HandleCancelEvent* in abgeleiteten Klassen überschrieben werden können. Die vorhandenen Implementierungen sind leer.

Bei der Ausführung des Ereignisses wird die *HandleProcessEvent*-Methode, beim Abbruch eines bereits vorgemerkten Ereignisses die *HandleCancelEvent*-Methode des Ereignisses

aufgerufen. Auch hier gibt es wieder ein abgestuftes Konzept parametrisierter Klassen, das es erlaubt, feste (*TStdEvent*) bzw. beliebige Methoden (*TGenEvent*) der Modellkompente aufzurufen.

### 4.6.2 Bearbeitung von Ereignissen

Die Bearbeitung von Ereignissen erfolgt hierarchisch. Dabei muß zwischen dem Zeitpunkt, zu dem das Ereignis vorgemerkt und dem eigentlichen Ereigniszeitpunkt, zu dem es ausgeführt werden soll, unterschieden werden. Um ein Ereignis vorzumerken, wird die *PostEvent*-Methode der Modellkomponente aufgerufen. Als Parameter werden das Ereignis und der Ereigniszeitpunkt übergeben. Ein vorgemerktes, aber noch nicht bearbeitetes Ereignis kann mit *CancelEvent* abgebrochen werden. Das nachfolgend vorgestellte Konzept erlaubt es, in jeder Hierarchieebene, sowohl beim Vormerken als auch bei der Ausführung des Ereignisses einzugreifen.

Beim Vormerken kann jedes Ereignis durch sog. Event-Handler manipuliert werden. Dazu lassen sich jeder Modellkomponente Event-Handler zuordnen. Die Event-Handler werden von der globalen Instanz der Klasse *TEventHandlerManager* zentral für alle Modellkomponenten verwaltet. Jedes Ereignis besitzt einen bestimmten Ereignistyp. Event-Handler können entweder nur Ereignisse eines bestimmten Typs oder Ereignisse mit beliebigem Typ bearbeiten. Sobald die bereits erwähnte Laufzeitinformation in C++ zur Verfügung steht, kann diese zur dynamischen Bestimmung des Ereignistyps verwendet werden. Mit Hilfe der *AddEventHandler*-Methode der *TEntity*-Klasse lassen sich Event-Handler einer Modellkomponente zuordnen.

Nachdem *PostEvent* (analog für *CancelEvent*) aufgerufen wurde, versucht die Modellkomponente einen passenden Event-Handler zu finden. Zunächst wird versucht, einen Handler zu finden, der den speziellen Ereignistyp bearbeiten kann. Falls keiner gefunden wird, so wird als nächstes ein Handler gesucht, der alle Ereignistypen verarbeiten kann. Schlägt auch dies fehl, wird die *PostEvent*-Methode der übergeordneten Modellkomponente aufgerufen. Dies wird rekursiv solange fortgeführt, bis entweder ein Event-Handler gefunden wurde oder die höchste Hierarchieebene erreicht ist und ein Fehler generiert wird. Es handelt sich hier um einen Bottom-Up-Ansatz, bei dem alle Ereignisse die Hierarchieebenen von unten nach oben durchlaufen. Wie im nächsten Unterkapitel erläutert wird, ist in der höchsten Ebene normalerweise der Kalender installiert, der alle Ereignisse behandelt. Aus Effizienzgründen wird der zuletzt benutzte Event-Handler in der Modellkomponente gespeichert. Der Event-Handler kann zur eigenen Modellkomponente oder einer übergeordneten Instanz gehören, so daß Hierarchieebenen, in denen kein eigener Handler installiert wurde, automatisch ausgelassen werden. Dieser Cache hat meist eine hohe Trefferrate, da viele Modellkomponenten nur Ereignisse eines Typs erzeugen. Die Suchzeit nach einem geeigneten Event-Handler ver-

ringert sich durch diese Maßnahme erheblich. Bei der Installation eines neuen Event-Handlers muß der Cache gelöscht werden. Da die meisten Handler während der Initialisierungsphase installiert werden, hat dies keine Auswirkungen auf die Trefferrate des Caches.

Sobald ein passender Event-Handler gefunden wurde, wird ihm das Ereignis zur Behandlung übergeben. Ein Handler kann dabei wie ein Filter arbeiten, indem er das Ereignis nach der Behandlung an die nächst höhere Instanz übergibt. Er kann das Ereignis aber auch speichern wie ein Kalender und es zum angegebenen Zeitpunkt ausführen, indem er die *ProcessEvent*-Methode des Ereignisses aufruft. Eine weitere Möglichkeit besteht darin, den Inhalt des Ereignisses sofort auszuwerten und direkt darauf zu reagieren.

Um auch zum eigentlichen Ereigniszeitpunkt eingreifen zu können, muß der Event-Handler ein eigenes neues Ereignis in das bestehende Ereignis einbetten und dieses dann weitergeben. Bei der Ausführung des ursprünglichen Ereignisses wird das eingebettete Ereignis automatisch mitabgearbeitet. Die Abarbeitung geschieht in einem Top-Down-Ansatz, von hohen zu niederen Hierarchieebenen, so daß die eingebetteten Ereignisse zuerst ausgeführt werden.

Somit kann ein Handler sowohl beim Vormerken eines Ereignisses als auch bei der Ausführung eingreifen. Eine übergeordnete Modellkomponente hat auf diese Weise die Möglichkeit, alle Ereignisse der enthaltenen Modellkomponenten zu manipulieren. Die untergeordneten Komponenten müssen dazu nicht verändert werden. Dieses Prinzip kann sogar dazu verwendet werden, den momentanen Zustand einer Modellkomponente allen übergeordneten Instanzen mitzuteilen, ohne diese kennen zu müssen.

Das hier vorgestellte Ereigniskonzept eignet sich demnach sowohl zur konventionellen Ereignisbearbeitung wie auch als Kommunikationsmedium zum Informationsaustausch zwischen Hierarchieebenen.

### 4.6.3 Verwaltung von Ereignissen

Die Ereignisse werden von Kalendern verwaltet. Normalerweise gibt es einen systemweiten Kalender, der in einer von *TModel* abgeleiteten Klasse installiert wird. Ein Kalender ist ein spezieller Event-Handler. *PostEvent* trägt ein Ereignis in den Kalender ein, *CancelEvent* löscht es wieder. Die *GetNextEvent*-Methode liefert das zeitlich nächste Ereignis, das anschließend ausgeführt werden kann. Die Ausführung geschieht durch Aufruf der *ProcessEvent*-Methode des Ereignisses. Diese sorgt dafür, daß die *HandleProcessEvent*-Methoden aller eingebetteten Ereignisse und anschließend die des ursprünglichen Ereignisses aufgerufen werden. Das Ganze erfolgt im Zusammenspiel mit dem Modell und der Systemsteuerung und wird im nächsten Unterkapitel behandelt. Ein Kalender kann prinzipiell in jeder Modellkomponente installiert werden, so daß auch verteilte Kalender realisierbar sind. Bei parallelen Systemen müssen die Teilmodelle dafür sorgen, daß die Kalender konsistent geführt werden.

Die Klasse *TCalendar* ist eine abstrakte Basisklasse. Je nach Anzahl der zu speichernden Ereignisse können unterschiedliche Implementierungen realisiert werden. Der normale Kalender der Klasse *TStdCalendar* verwendet in seiner Implementierung eine sortierte Liste.

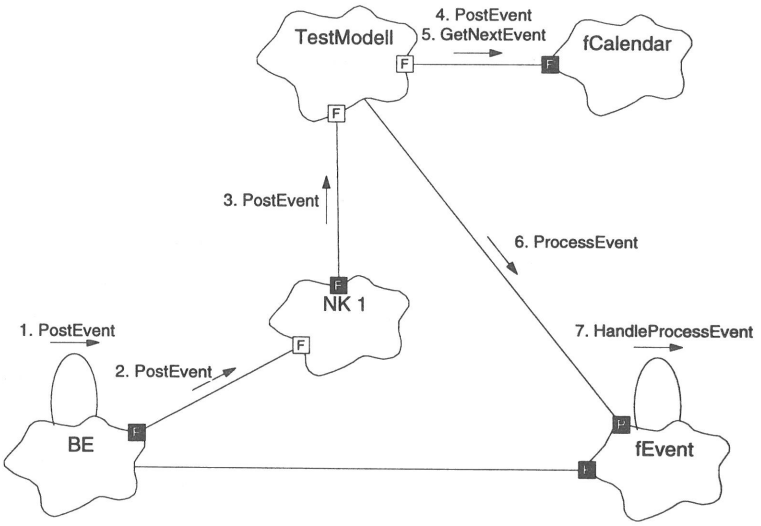



**Bild 4.10:** Eintragen eines Ereignisses in den Kalender

Bild 4.10 zeigt den Ablauf beim Eintragen des Bedienende-Ereignisses der Bedieneinheit. Zunächst wird das Ereignis, zusammen mit dem Ereigniszeitpunkt, als Parameter an die *PostEvent*-Methode der Bedieneinheit übergeben. Da weder die Bedieneinheit noch der Netzknoten einen eigenen Event-Handler spezifiziert haben, wird das Ereignis vom Kalender der Modellkomponente vorgemerkt und in die Ereignisliste eingetragen. Zu einem späteren Zeitpunkt erhält die Modellkomponente mit *GetNextEvent* das Ereignis vom Kalender und führt dieses durch Aufruf der *ProcessEvent*-Methode aus. Bild 4.11 zeigt das gleiche Beispiel, erweitert um einen zusätzlichen Event-Handler, der im Netzknoten 1 installiert wurde. In diesem Fall wird das Ereignis vom Netzknoten 1 an den Event-Handler zur Bearbeitung übergeben. Dieser bettet ein eigenes Ereignis in das bestehende ein. Anschließend übergibt er das ursprüngliche Ereignis der nächsten Hierarchieebene. Nach dem Austragen des Ereignisses aus dem Kalender erfolgt die Abarbeitung durch Aufruf der *ProcessEvent*-Methode des ursprünglichen Ereignisses. Diese ruft intern zunächst die *HandleProcessEvent*-Methode des eingebetteten Ereignisses und anschließend die eigene Version dieser Methode auf. Auf diese Weise ist höheren Hierarchieebenen möglich, sowohl beim Vormerken als auch bei der Ausführung von Ereignissen tieferliegender Ebenen einzugreifen.

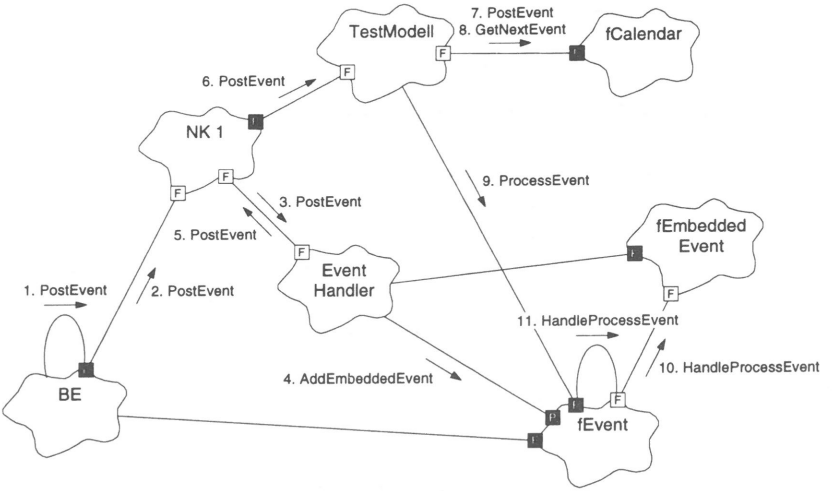



**Bild 4.11:** Hierarchische Ereignisbearbeitung

Bild 4.12 zeigt das Klassendiagramm aller an der Ereignisverarbeitung beteiligten Komponenten.

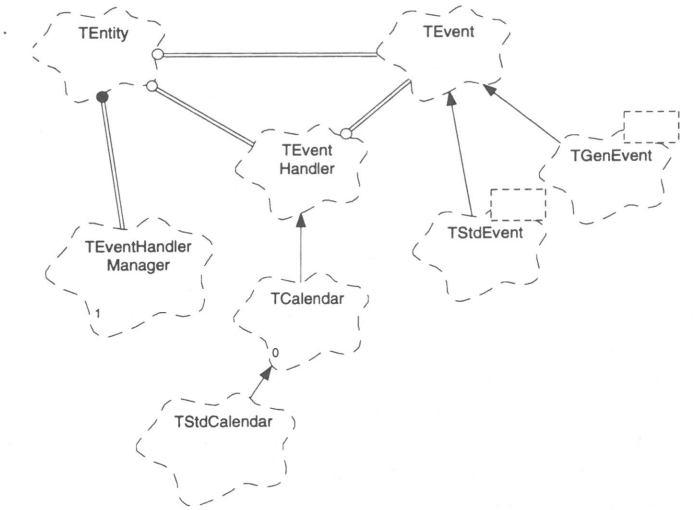



**Bild 4.12:** Klassendiagramm der Ereignisverwaltung

## 4.7 Die Simulationssteuerung

### 4.7.1 Aufgaben der Simulationssteuerung

Die Simulationssteuerung übernimmt die Aufgabe, den Ablauf des Simulationsprogrammes von der Initialisierung bis zur Auswertung der Ergebnisse durchzuführen und zu kontrollieren. Obwohl die genaue Ausführung von der jeweiligen Problemstellung abhängt, gibt es einige gemeinsame Aufgaben, die in jeder Simulation durchgeführt werden müssen. Die Klassen der Simulationssteuerung ergeben einen Rahmen, der für die jeweilige Simulation flexibel angepaßt werden kann. Die abstrakte Basisklasse *TSimulation* definiert die allgemeinen Schritte, die bei jeder Simulation durchgeführt werden müssen. Dazu zählen folgende Aufgaben:

- Initialisierung der Datenstrukturen
- Einlesen und Verarbeiten der Simulationsparameter
- Starten der Simulation
- Durchführung des Simulationslaufes
- Ausgeben der Ergebnisse
- Aufräumarbeiten

Zunächst müssen alle dynamischen Datenstrukturen aufgebaut und initialisiert werden. Anschließend werden die Simulationsparameter eingelesen und ausgewertet. Je nach Aufgabenstellung kann es notwendig sein, Teile des Simulationsmodells dynamisch in Abhängigkeit von den Eingabeparametern zu erzeugen. Beim Start der Simulation müssen alle Zähler und Statistiken in einen definierten Zustand gebracht werden. Die eigentliche Durchführung des Simulationslaufes hängt von der angewandten Methode ab. Im Normalfall wird die Simulation gestartet, ein Warmlauf und eine Anzahl von Teiltests durchgeführt. Anschließend werden die Ergebnisse berechnet und ausgegeben. Die abschließenden Aufräumarbeiten dienen dazu, dynamische Datenstrukturen zu löschen und offene Dateien zu schließen. Für jede dieser Aufgaben besitzt *TSimulation* eine virtuelle Methode, die in einer abgeleiteten Klasse überschrieben werden kann. Die *Run*-Methode ruft diese Methoden der Reihe nach auf. Die Klasse *TStdSimulation* implementiert die Steuerung für den normalen Ablauf einer Simulation, inklusive Warmlauf- und Teilteststeuerung. Für die Steuerung einer Simulation werden die Initialisierungsmethoden in einer eigenen Steuerklasse überschrieben. Das eigentliche Hauptprogramm ist dann sehr einfach:

```
TSimulation *       gSimulation = 0;  // Global pointer to simulation

main()
{
    gSimulation = new TMySimulation;       // Create simulation object
    gSimulation->Run();                    // Run it
    delete gSimulation;                    // Delete it
    return 0;
}
```

Die folgenden Unterkapitel beschäftigen sich damit, wie die Simulationssteuerung im einzelnen aufgebaut ist.

### 4.7.2 Aktualisierung des Systemzustandes

Während des Simulationslaufes benötigen viele Komponenten die Information, in welcher Phase sich die Simulation gerade befindet. So müssen z.B. Zähler für statistische Auswertungen nach jedem Teiltest ausgewertet und anschließend neu initialisiert werden. Die Simulationssteuerung hat deshalb die Aufgabe, allen Komponenten, die diese Information benötigen, jeden Übergang in eine andere Phase mitzuteilen. Im folgenden sind dabei zwei unterschiedliche Gesichtspunkte zu beachten. Zum einen stellt sich die Frage, wie der Übergang von einer Phase in die nächste vorgenommen wird. Zum anderen muß ein Weg gefunden werden, alle interessierten Komponenten von diesem Übergang zu unterrichten. Da beide Probleme stark von der eigentlichen Problemstellung abhängen, gibt es keine standardisierte Lösung für die Vorgehensweise. Es wurde deshalb ein verteilter Lösungsansatz gewählt, der sehr flexibel an unterschiedliche Anforderungen angepaßt werden kann. Dabei entscheidet die Simulationssteuerung nicht selbst, ob eine Phase abgeschlossen ist, sondern läßt sich von anderen Komponenten darüber unterrichten. Zu diesem Zweck enthält die Steuerung drei Objekte der Klasse *TNotificationHandler*, je eines für das Ende der Warmlaufphase, das Ende der Teiltests und das Simulationsende. Jedes Objekt der Klasse *TNotificationHandler* kann mehrere Objekte der Klasse *TNotifier* verwalten. Am Ende einer Phase wird die Steuerung durch Aufruf der *Notify*-Methode davon in Kenntnis gesetzt. Das wirkliche Ende einer Phase ist jedoch erst erreicht, wenn alle *TNotifier*-Objekte, die für eine Phase registriert sind, ihre *Notify*-Methode aufgerufen haben. Auf diese Weise lassen sich auch komplexe Ende-Bedingungen beschreiben. So könnte z.B. ein Teiltest dann beendet werden, wenn in einigen Pfaden jeweils eine Mindestanzahl von Nachrichten ausgetauscht und eine kritische Statistik eine gewünschte Aussagesicherheit erreicht hat. Dazu würde man in die Pfade Zählermodellkomponenten einfügen, die nach einer festgelegten Anzahl von durchlaufenden Meldungen ihre *Notify*-Methode aufrufen. Eine spezielle Statistikkomponente könnte ebenfalls so modifiziert werden, daß sie nach Erreichen einer vorgegebenen statistischen Aussagesicherheit die *Notify*-Methode aufruft. Auf diese Weise lassen sich mehrere Bedingungen angeben, die alle erfüllt sein müssen, bevor mit der nächsten Phase fortgefahren werden kann. Selbst Zeitbe-

dingungen können durch das Eintragen spezieller Ereignisse in den Kalender realisiert werden.

Mit Hilfe der abstrakten Klasse *TSimulationControl* können alle an einem Phasenübergang interessierte Instanzen benachrichtigt werden. Wie folgender Ausschnitt aus der Klassendeklaration zeigt, gibt es für jeden Übergang eine virtuelle Methode, die bei Bedarf überschrieben werden kann:

```
class TSimulationControl {
public:
    virtual void InitSimulation();          // Handle initialization
    virtual void StartSimulation(TSimulation &); // Start simulation
    virtual void StopSimulation();          // Stop simulation
    virtual void StartTransientPhase();     // Begin of transient phase
    virtual void StopTransientPhase();      // End of transient phase
    virtual void StartBatch(int batchNr);   // Start batch # batchNr
    virtual void StopBatch(int batchNr);    // Stop batch # batchNr
    ...
};
```

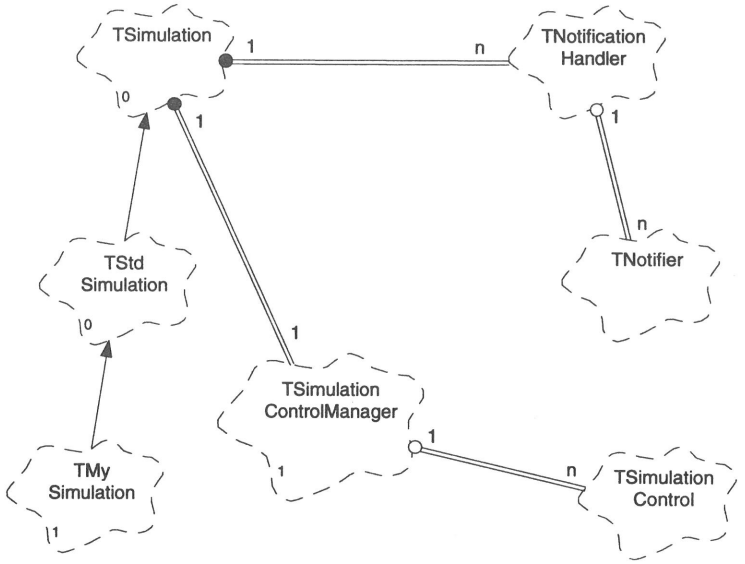



**Bild 4.13:** Klassendiagramm der Simulationssteuerung

Natürlich müssen nur die Methoden überschrieben werden, die für die jeweilige Komponente relevant sind. Eine Statistikklasse kann z.B. mit Hilfe der *StartBatch*- und *StopBatch*-Methoden ihre internen Zähler zurücksetzen bzw. auswerten. Bild 4.13 zeigt das zugehörige Klassendiagramm.

Alle Objekte der Klasse *TSimulationControl* werden von einem globalen Objekt der Klasse *TSimulationControlManager* verwaltet, bei dem sich die Objekte bei ihrer Generierung automatisch anmelden. Die Managerklasse stellt außer den Methoden zur Verwaltung der Objekte dieselben Methoden wie die *TSimulationControl*-Klasse zur Verfügung. Diese werden bei Phasenübergängen von der Simulationsklasse aufgerufen und sorgen dafür, daß die entsprechenden Methoden aller registrierten Objekte aufgerufen werden.

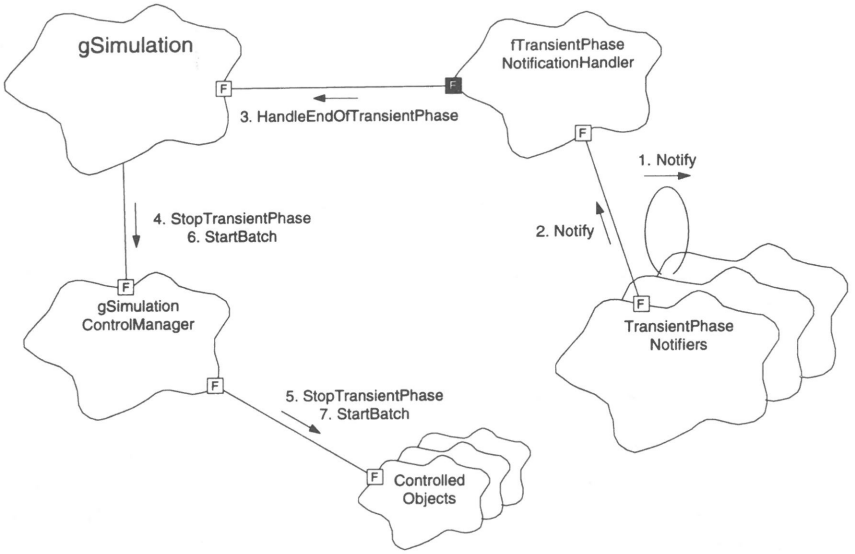



**Bild 4.14:** Übergang von der Warmlaufphase zum ersten Teiltest

Bild 4.14 zeigt anhand eines einfachen Beispiels den Übergang von der Warmlaufphase zum ersten Teiltest. Mehrere Objekte der Klasse *TNotifier* zeigen dem *TNotificationHandler* das Ende der Warmlaufphase durch den Aufruf der *Notify*-Methode an. Das *TNotification-Handler*-Objekt unterrichtet die Simulationsklasse durch den Aufruf von *HandleEndOf-TransientPhase*. Dies erfolgt aber erst, nachdem alle *TNotifier*-Objekte ihre *Notify*-Methode aufgerufen haben. Dadurch werden beim globalen *TSimulationControlManager*-Objekt nacheinander die Methoden *StopTransientPhase* und *StartBatch(1)* ausgelöst, da das Warm-

laufende gleichzeitig den Beginn des ersten Teiltests markiert. Das Managerobjekt verteilt die Aufrufe an alle registrierten *TSimulationControl*-Objekte. Auf diese Weise lassen sich sehr mächtige und dennoch einfach zu überschauende Simulationssteuerungen aufbauen.

Anhand der Klasse *TSimulationControl* läßt sich ein interessanter Implementierungsaspekt aufzeigen. Oft stellt sich die Frage, ob eine Klasse privat ererbt oder als Unterobjekt implementiert werden soll. Alle Klassen, die die Funktionalität von *TSimulationControl* brauchen, benötigen diese für ihre interne Logik, z.B. zum Initialisieren von Zählern. Eine öffentliche Vererbung scheidet also aus, da nur die Implementierung der Klasse benötigt wird, nicht jedoch ihr Interface. Normalerweise würde man deshalb ein Objekt der Klasse *TSimulationControl* in die fragliche Klasse einbetten. Dies ist aber nicht möglich, da zur Nutzung der Funktionalität einige Methoden der Klasse *TSimulationControl* überschrieben werden müssen. Es handelt sich hier um einen der wenigen Fälle, bei denen eine private Vererbung angebracht ist. Bei der privaten Vererbung wird nur die Implementierung, nicht jedoch das Interface vererbt. Trotzdem hat man noch die Möglichkeit, vererbte Methoden zu überschreiben. Sobald jedoch mehr als ein Unterobjekt derselben Basisklasse benötigt wird, scheidet diese Möglichkeit aus. In diesem Fall müssen die Objekte eingebettet werden. Wenn garantiert wäre, daß jede Modellkomponente nur einen Ein- und Ausgangsport besitzt, könnte man diese z.B. privat ererben und die Methoden direkt überschreiben. Die Notwendigkeit für Message-Handler würde dann entfallen. Da aber jede Komponente beliebig viele Ports besitzen kann, ist diese einfache Lösung nicht allgemeingültig anwendbar. Eine gute Diskussion dieser Problematik gibt Scott Meyers in [47].

### 4.7.3 Ablauf der Ereignissteuerung

Während der Warmlaufphase und den einzelnen Teiltests läuft die Simulation ereignisgesteuert ab. Die Ereignissteuerung erfolgt durch Zusammenarbeit der Simulationssteuerung mit den Teilmodellen. Die Steuerung ist nur für das Anstoßen des Vorganges verantwortlich. Die Speicherung und Ausführung von Ereignissen geschieht dezentral in den Teilmodellen bzw. deren Kalendern. Die Simulationssteuerung enthält eine Liste aller Teilmodelle der Simulation. Nach dem Start der Warmlaufphase wird der Reihe nach bei allen registrierten Modellen die Methode *ProcessPendingEvents* aufgerufen. Diese Methode ist dafür verantwortlich, Ereignisse aus dem Kalender auszutragen und diese auszuführen. Die Methode kehrt erst zurück, wenn entweder keine weiteren Ereignisse im Kalender vorhanden sind oder die Ausführung aufgrund eines Phasenendes vorzeitig abgebrochen werden soll. Bei verteilten Kalendern sind die Teilmodelle dafür verantwortlich, die Kalender miteinander zu synchronisieren. Der Übergang von der Warmlaufphase zu den Teiltests erfolgt wie oben beschrieben. Auch für die einzelnen Teiltests wird die *ProcessPendingEvents*-Methode reihum für alle Teilmodelle aufgerufen. Nach dem Ende des letzten Teiltests wird die Simulation beendet.

### 4.7.4 Ausgabe der Ergebnisse

Die Aufbereitung und Ausgabe der Ergebnisse wird von der Simulationssteuerung veranlaßt und automatisch durchgeführt. Die Simulationsergebnisse können dabei am Ende der Simulation sowie wahlweise auch nach jedem Teiltest ausgedruckt werden. Die Möglichkeiten, wie die standardisierte Ausgabe an unterschiedliche Gegebenheiten angepaßt werden kann, werden ausführlich in Kapitel 4.11 erörtert. Die Ausgabe kann auf jeden zur C++-Standard-Bibliothek kompatiblen Ausgabe-Stream erfolgen. Somit ist es einfach, Ergebnisse auf dem Bildschirm darzustellen oder sie in Dateien abzuspeichern. Die Simulationssteuerung benötigt hierzu nur die Angabe des Ausgabe-Streams sowie eines Druckformates. Das Druckformat legt fest, wie die Ausgabe formatiert werden soll und ist ebenfalls in Kapitel 4.11 beschrieben. Insgesamt hat der Anwender nur sehr wenig mit der Steuerung der Ergebnisausgabe zu tun und kann sich auf die eigentliche Simulation und die Formatierung der Ausgabe konzentrieren.

### 4.7.5 Einlesen von Simulationsparametern

Um ein Simulationsprogramm flexibler zu gestalten, werden einige Simulationsparameter nicht fest kodiert, sondern zu Beginn der Simulation eingelesen. Die Startwerte des Simulationslaufes sowie teilweise auch die Konfiguration des zugrundeliegenden Modells werden danach in Abhängigkeit von den eingelesenen Werten angepaßt. Um diesen Vorgang zu unterstützen, enthält die Simulationsbibliothek einige Hilfsklassen, die es erlauben, Parameter von einem beliebigen Eingabe-Stream zu lesen und umzusetzen. Die Klasse *TParser* ist in der Lage, eine dynamisch festgelegte Sprache mit C-ähnlicher Syntax zu parsen. Klassen, die von der Klasse *TParserObject* abgeleitet wurden, können an den Parser eine Liste von Tokens übergeben, die anschließend beim Parsen erkannt werden. Die Token beschreiben Attribute einzelner Sprachkonstrukte, wie z.B. Schlüsselwörter, einzuhaltende Reihenfolge, Bereichsgrenzen von Werten, etc. Wenn ein Token vom Parser erkannt wird, ruft er die *Init-Parameter*-Methode der Klasse *TParserObject* auf. Eine abgeleitete Klasse kann dann den übergebenen Wert beliebig weiterverwenden, z.B. zur Initialisierung von Objekten. Mit der Klasse *TParseEnvironment* wird die Umgebung festgelegt, z.B. kann hier der Eingabe-Stream angegeben werden, von dem die Parameter eingelesen werden. Auf diese Art kann die Initialisierung von Simulationsparametern auf einheitliche Weise erfolgen. Eine genauere Beschreibung des Parsers würde den Rahmen dieser Arbeit sprengen und soll daher nicht weiter vertieft werden.

## 4.8 Erzeugung von Zufallsverteilungen

Um die Komplexität einer Simulation zu verringern, werden viele Vorgänge nicht exakt nachgebildet, sondern durch Zufallsgrößen ersetzt. Dabei ist es notwendig, Zufallsvariablen mit einer vorgegebenen Verteilungsfunktion zu erzeugen. Die Simulationsbibliothek stellt die gängigsten Zufallsverteilungen zur Verfügung. Beim Entwurf der Klassenhierarchie wurde auf universelle Verwendbarkeit und leichte Erweiterbarkeit geachtet. Im folgenden wird dieses Konzept genauer vorgestellt.

### 4.8.1 Erzeugung von Zufallszahlen

Es gibt viele Methoden, echte oder sog. Pseudozufallszahlen zu erzeugen. Echte Zufallszahlen lassen sich z.B. aus Rauschquellen ableiten. Der Vorteil, eine wirklich zufällig Zahlenfolge zu erhalten, stellt sich in realen Anwendungen oft eher als Nachteil heraus. Während des Testens einer Simulation ist es nämlich vorteilhaft, immer dieselben Zahlenfolgen zu erhalten, um z.B. die Abfolge von Ereignissen wiederholen zu können. Dies ist mit echten Zufallszahlen nicht möglich, außer man speichert die erzeugte Zufallszahlenfolge ab, um sie später wiederverwenden zu können. Mittlerweile sind solche echten Zufallszahlenfolgen auf CD-ROM erhältlich. Da der Aufwand für diese Methode in vielen Fällen zu hoch ist und auch die Eigenschaften von hardware-unterstützten Generatoren nicht immer genau bekannt sind, beschränken sich die meisten Simulationsprogramme auf die Erzeugung von quasi-zufälligen Zahlen. Diese sog. Pseudozufallszahlen werden ausgehend von einem Startwert nach einer vorgegebenen Rechenvorschrift erzeugt. Durch Verwenden des gleichen Startwertes lassen sich die Zahlenfolgen beliebig wiederholen. Die zufälligen Eigenschaften dieser Zahlenfolgen hängen stark von der benutzten Rechenvorschrift ab. Da die gesamte statistische Aussagesicherheit einer Simulation von der Qualität der erzeugten Zufallszahlen abhängt, sollte man auf die Eigenschaften des verwendeten Zufallszahlengenerators achten. In der Literatur wird eine Vielzahl solcher Generatoren mit unterschiedlichsten Eigenschaften beschrieben [20]. Da es keinen optimalen Generator gibt, der für alle Zwecke gleich gut geeignet ist, wurde dies beim Entwurf der Simulationsbibliothek berücksichtigt. Die Klasse *TRandomNumberGenerator* ist eine abstrakte Basisklasse, die nur eine Methode zum Erzeugen der nächsten Zufallszahl zur Verfügung stellt. Erst die von dieser Klasse abgeleiteten Klassen implementieren die verschiedenen Rechenvorschriften. Da in der Simulation nur auf die in der Basisklasse definierte Methode zugegriffen wird, wirkt sich der Austausch verschiedener Generatoren im Programm nicht weiter aus. Da die Klasse nur eine einzige Aufgabe hat, wurde der Funktionsoperator *„operator ()“* überladen. Dies erlaubt die folgende einfache Schreibweise, um die nächste Zufallszahl der Folge zur erhalten:

```
TStdRandomNumberGenerator rng(123456);     // Init. with seed

for(int i = 0; i < 10; i++) {              // Print 10 random numbers
    Random number = rng();                 // Get next random number
    // Print to standard output stream
    cout << "Random number = " << number << endl;
}
```

Alle von *TRandomNumberGenerator* abgeleiteten Klassen müssen den Funktionsoperator überschreiben und bei jedem Aufruf eine im Intervall )0, 1( gleichverteilte Zahl zurückgeben. Die Grenzen des Intervalls sind nicht eingeschlossen, um Sonderfälle bei einer nachfolgenden Skalierung zu vermeiden. In der ersten Ausbaustufe wurden drei verschiedene Zufallszahlengeneratoren implementiert. Der Standardgenerator ist ein linear kongruenter Generator. Er benutzt den in [51] beschriebenen Algorithmus. Seine guten statistischen Eigenschaften und die einfache Rechenvorschrift machen ihn zur ersten Wahl für die meisten Anwendungen. Die Klasse *TMixedRandomNumberGenerator* implementiert einen Generator, der durch Mischung zweier linear kongruenter Generatoren hervorgeht. Dieser Generator hat sehr gute statistische Eigenschaften und eine sehr große Periode von $\approx 2{,}3\ 10^{18}$ [19]. Der *TVax-RandomNumberGenerator* bildet aus Kompatibilitätsgründen den für die ursprünglich am Institut verwendete Simulationsbibliothek entwickelten Generator nach. Bild 4.15 zeigt das einfache Klassendiagramm. Je nach Bedarf können weitere Generatoren hinzugefügt werden, z.B. könnte ein Generator entwickelt werden, der die Zufallszahlenfolge von CD-ROM liest und diese ausgibt.

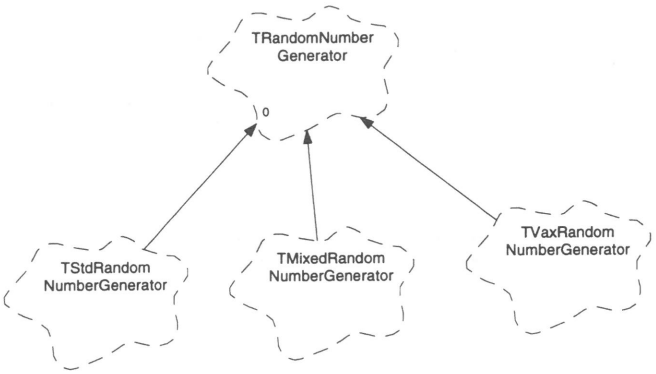



**Bild 4.15:** Klassendiagramm der Zufallszahlengeneratoren

### 4.8.2 Erzeugung von Zufallsverteilungen

Für die meisten Anwendungen ist es nicht ausreichend, nur gleichverteilte Zufallszahlen zur Verfügung zu haben. In vielen Fällen werden Zufallsvariablen mit vorgegebenen Verteilungsfunktionen benötigt. Anstatt die Verteilungen direkt zu erzeugen, bietet es sich an, diese Verteilungen aus einer gleichverteilten Zufallsvariable zu bestimmen. Da alle Verteilungsfunktionen den Eingangsbereich auf Werte zwischen 0 und 1 abbilden, kann man mit Hilfe der jeweiligen Umkehrfunktion die entsprechenden Eingangswerte erhalten.

Mit Hilfe von abstrakten Basisklassen wurde ein Gerüst von Zufallsverteilungen realisiert, vgl. Bild 4.16. Die Basisklasse *TDistribution* verwaltet den zugrundeliegenden, gleichverteilten Zufallsgenerator. Jede Verteilung wird mit einer Referenz auf einen systemweiten Zufallsgenerator initialisiert, kann aber bei Bedarf mit einem eigenen Generator arbeiten. Um Korrelationen zwischen Zufallszahlengeneratoren zu vermeiden, ist es jedoch sinnvoller, nur einen systemweiten Generator zu verwenden. Eine Alternative besteht darin, mehrere gleichartige Generatoren mit unterschiedlichen Startwerten zu verwenden. Die Startwerte sind dabei so zu wählen, daß sie gleichmäßig über die Periode des Generators verteilt sind. Da die Zahlenfolgen in diesem Fall nur gegeneinander verschoben sind, muß mit entsprechenden Korrelationen gerechnet werden. Eine ausführliche Betrachtung dieser Problematik findet sich in [20].

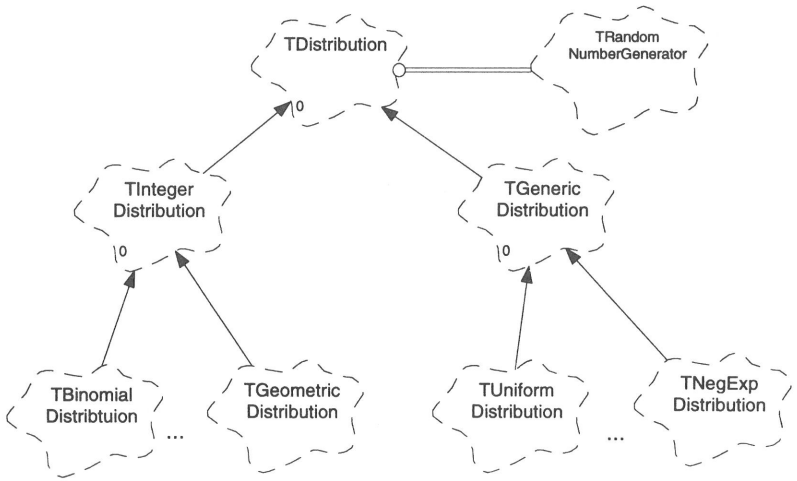



**Bild 4.16:** Klassendiagramm der Zufallsverteilungen

Zwei weitere abstrakte Klassen wurden von der Basisklasse abgeleitet. *TIntegerDistribution* ist die gemeinsame Basisklasse für alle Verteilungen, bei denen nur ganzzahlige Werte erlaubt sind. *TGenericDistribution* dient als Basisklasse für alle übrigen Verteilungen. Bei beiden Varianten wurde wieder der Funktionsoperator zur Erzeugung des nächsten Wertes herangezogen. Die von diesen Klassen abgeleiteten Klassen realisieren die häufig verwendeten Standardverteilungen.

### 4.8.3 Anwendung von Zufallsvariablen

Die Bedienzeit einer Bedienstation wird häufig von einer Zufallsvariablen gesteuert. Wenn man im Konstruktor der Modellkomponente eine Referenz auf die Klasse *TGeneric-Distribution* übergibt, lassen sich Bedieneinheiten mit beliebig verteilten Bedienzeiten erzeugen, ohne daß der Code der Bedieneinheit angepaßt werden müßte. Dies ist möglich, da in der Implementierung der Bedieneinheit nur auf das Interface der abstrakten Basisklasse zugegriffen wird. Die wirkliche Verteilung wird erst bei der Übergabe im Konstruktor festgelegt, und das Ergebnis wird erst zur Laufzeit ermittelt. Auf diese Weise lassen sich allgemeine Modellkomponenten entwerfen, deren statistische Eigenschaften erst bei der Instanzierung der Objekte festgelegt werden.

Bei manchen Anwendungen sollen die Parameter der Zufallsverteilungen bzw. der Typ der Verteilung selbst, erst zur Laufzeit des Programmes als Eingabeparameter festgelegt werden. Dieser Fall wird durch eine spezielle Managerklasse unterstützt. Parallel zur Klassenhierarchie der Verteilungen wurde eine weitere Hierarchie von Parserklassen entworfen. Für jede Verteilungsfunktion wird eine spezielle Parserklasse von der abstrakten Klasse *TGeneric-DistributionParser* abgeleitet. Sie muß in der Lage sein, alle Parameter einer Verteilung einzulesen. Dazu bedient sie sich den von *TParserObject* bereitgestellten Hilfsmitteln (vgl. Kap. 4.7.5). Alle Parserklassen melden sich automatisch bei einem globalen Objekt der Klasse *TGenericDistributionManager* an. Die Methode *ReadGenericDistribution* dieser Klasse liest den Namen der Verteilung ein und sucht den passenden Parser aus der gespeicherten Liste. Dieser liest dann selbständig die Parameter der Verteilung ein. Ein Aufruf von *CreateGenericDistribution* erzeugt eine Instanz dieser Verteilung, die anschließend in allgemeinen Modellkomponenten verwendet werden kann. Durch den Einsatz des Managerobjekts und der verschiedenen Parserklassen erhält man eine verteilte Lösung, in die neue Verteilungen automatisch eingebunden werden können, ohne daß Änderungen am Managerobjekt vorgenommen werden müssen. Um eine neue Verteilung zu implementieren, muß nur jeweils die Verteilungsklasse und der zugehörige Parser neu implementiert werden. Beides läßt sich meistens aufgrund des bereits vorhandenen Rahmens mit wenigen Zeilen Code erledigen. Für ganzzahlige Verteilungen gibt es eine analoge Klassenhierarchie (*TIntegerDistributionManager, TIntegerDistributionParser, ...*).

Das folgende Beispiel zeigt, wie einfach es ist, eine beliebige Verteilung einzulesen und eine Bedieneinheit mit dieser Verteilung zu initialisieren.

```
class TBedieneinheit : public TEntity {
public:
    TBedieneinheit (TGenericDistribution & d, const TString & name,
                    TEntity * owner = 0) :
        TEntity(name, owner), // Init. base class and distribution
        fServiceTime(&d) {}
...
    TGenericDistribution *  fServiceTime; // Service time distribution
};

main()
{
    TParseEnvironment  env(cin);    // Parse from standard input stream
    TGenericDistribution & d =      // Read distribution from stream
       gGenericDistributionManager->ReadGenericDistribution(env);
    TBedienEinheit be (d, "BE 1"); // Init. server with distribution
    ...
}
```

Zunächst wird eine um die Verteilung der Bedienzeit erweiterte Klasse *TBedieneinheit* definiert. Das Hauptprogramm zeigt, wie mit Hilfe des globalen Managerobjektes eine beliebige Verteilung eingelesen und erzeugt werden kann. Diese Verteilung wird anschließend dem Konstruktor der Bedieneinheit übergeben. Eine Schwierigkeit wurde in diesem Beispiel nicht berücksichtigt. Die erzeugte Verteilung wird von niemandem wieder gelöscht. Sofern man kein Speicherleck in Kauf nehmen will, muß dies entweder von Hand oder im Destruktor der Bedieneinheit erfolgen. Natürlich könnte man eine spezielle Bedieneinheit entwerfen, die die Verteilung direkt einliest und erzeugt. An diesem Beispiel sieht man, wie sehr die Möglichkeiten der objektorientierten Programmierung die Definition flexibler Modellkomponenten vereinfachen.

## 4.9 Statistik

Die während eines Simulationslaufes anfallenden Meßwerte, wie z.B. Wartezeiten, Auslastungen von Bedieneinheiten und Verlustwahrscheinlichkeiten, müssen von der Anwendung ermittelt werden. Spezielle Statistikklassen unterstützen die Aufbereitung und Auswertung der Stichproben. Mit Hilfe der Methoden der beurteilenden Statistik kann die statistische Aussagesicherheit der Ergebnisse bestimmt werden. Die folgenden Abschnitte geben einen Überblick über die vorhandenen Statistikklassen. Der theoretische Hintergrund soll an dieser Stelle nicht weiter betrachtet werden. Es wird auf die entsprechende Literatur verwiesen [56].

### 4.9.1 Statistikklassen

Je nach Art der anfallenden Meßwerte gibt es unterschiedliche Verfahren zur Auswertung von Stichproben. Von der Basisklasse *TStatistic* wurden deshalb mehrere Klassen abgeleitet, die die verschiedenen Meßverfahren definieren, vgl. Bild 4.17. Die Basisklasse übernimmt die Aufgaben der Simulationssteuerung, d.h. sie veranlaßt die Initialisierung sowie die Berechnung der Zwischenergebnisse nach jedem Teiltest. Sie benutzt dazu die Klasse *TSimulationControl*, mit Hilfe deren Methoden die einzelnen Simulationsphasen bestimmt werden können (vgl. Kap. 4.7.2). Außerdem ist sie von *TPrintServer* abgeleitet, um Ergebnisse ausgeben zu können.

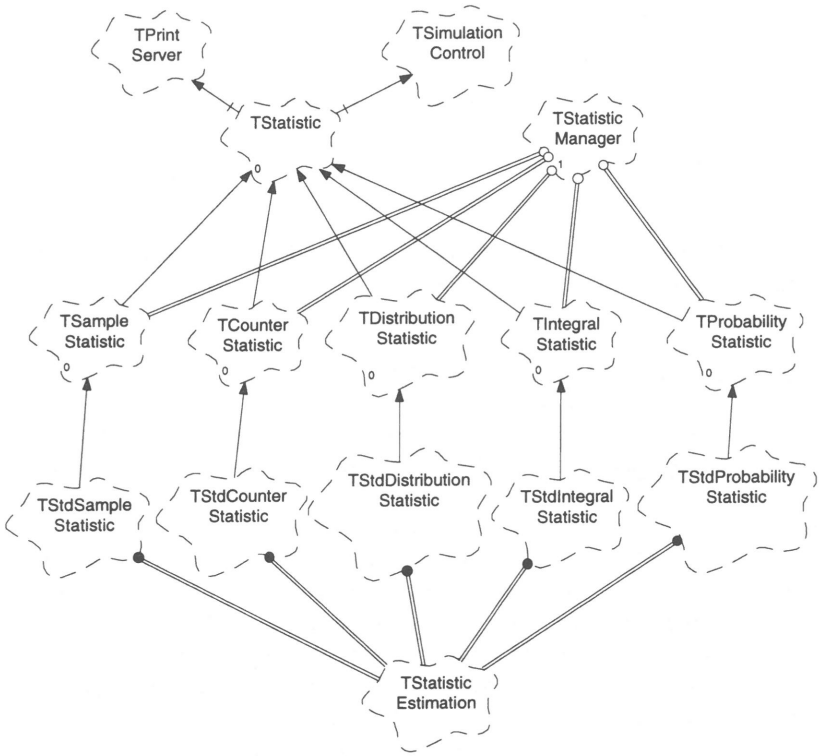



**Bild 4.17:** Klassendiagramm der Statistikklassen

Die von *TStatistic* abgeleiteten Klassen stellen abstrakte Basisklassen dar, die die Operationen zur Messung und Auswertung für jeweils eine Statistikart definieren. Die Klasse *TSample-Statistic* besitzt z.B. eine *Update*-Methode, mit der neue Stichprobenwerte an die Statistik übergeben werden. Nach jedem Teiltest bzw. am Ende der Simulation, können Mittelwert, Standardabweichung, Variationskoeffizient sowie Vertrauensintervalle für die Zufallsvariable bestimmt werden. Da es zur Berechnung dieser Werte unterschiedliche Algorithmen gibt, erfolgt die eigentliche Implementierung erst in weiter abgeleiteten Klassen, so z.B. in der Klasse *TStdSampleStatistic*. Für spezielle Anforderungen können andere Algorithmen herangezogen werden, die in neuen Klassen definiert werden. Die Klasse *TStatisticEstimation* wird zur Bestimmung der statistischen Aussagesicherheit benutzt und wird im folgenden Unterkapitel beschrieben.

Eine spezielle globale Managerklasse dient zur Erzeugung von Statistikobjekten. Modellkomponenten erzeugen ihre Statistikobjekte durch Aufruf von entsprechenden *Create-Statistic*-Methoden der Klasse *TStatisticManager*. Spezielle Methoden der Managerklasse erlauben die Auswahl verschiedener Implementierungen. Dadurch ist es möglich, in den Modellkomponenten Statistiken mit unterschiedlichen Algorithmen anzuwenden, ohne den Code der Modellkomponenten verändern zu müssen. Alle Modellkomponenten verwenden nur die von den abstrakten Basisklassen zur Verfügung gestellten Schnittstellen. Die indirekte Erzeugung der Statistikobjekte ermöglicht die Trennung von Modellkomponenten und Meßmethoden.

Alle Statistikklassen verwenden das in Kapitel 4.11 vorgestellte flexible Ausgabekonzept, um Ergebnisse aufzubereiten und diese auszudrucken.

### 4.9.2 Statistische Aussagesicherheit

Um die statistische Sicherheit einer Zufallsvariablen zu bestimmen, muß deren Varianz bekannt sein. Dies ist im allgemeinen nicht der Fall, sondern es liegt nur ein Schätzwert vor, der aus den Messungen gewonnen wurde. Es gibt verschiedene Verfahren, die es erlauben, aufgrund der Schätzfunktion der Varianz ein Vertrauensintervall für die statistische Sicherheit zu bestimmen. Ein bekanntes Verfahren ist z.B. die Student-t-Verteilung. Die Klasse *TStatisticEstimation* ist eine abstrakte Oberklasse, die von allen Statistikklassen zur Bestimmung von Vertrauensintervallen verwendet wird. Davon abgeleitete Klassen implementieren die unterschiedlichen Verfahren zur Bestimmung von Vertrauensintervallen. Bild 4.18 zeigt die Klassenhierarchie. Die Student-t-Verteilung wurde in mehreren Klassen implementiert. Die Klasse *TStudentSearch* verwendet eine Tabelle für die gebräuchlichsten Werte. *TStudent-Calc* berechnet die Werte jeweils neu, und *TStudentMixed* benutzt die beiden vorhergehenden Klassen, um zuerst den Wert in der Tabelle zu suchen oder ihn zu berechnen, falls er in keiner Tabelle gefunden wurde.

Die Klasse *TStatisticEstimationManager* dient auch hier zur Entkopplung von Statistikklassen und den eigentlichen Schätzmethoden zur Bestimmung der Vertrauensintervalle. Alle Statistikklassem bedienen sich der globalen Instanz der Managerklasse, um Objekte einer von *TStatisticEstimation* abgeleiteten Klasse zu erzeugen. Durch Änderung der Managerklasse können verschiedene Schätzverfahren verwendet werden.

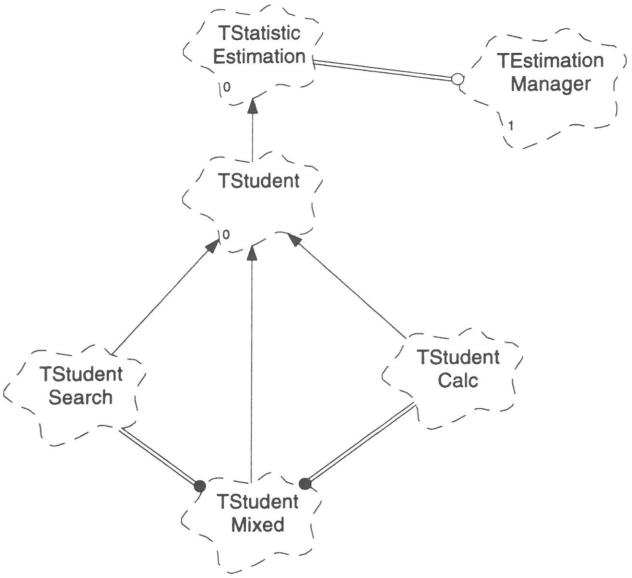



**Bild 4.18:** Klassendiagramm für statistische Schätzverfahren

## 4.10 Messen am Modell

Die im vorangegangenen Unterkapitel behandelten Statistikklassen werden normalerweise direkt in Modellkomponenten integriert, um dort Messungen durchzuführen. Es gibt jedoch auch andere Anwendungsgebiete, bei denen es nicht sinnvoll bzw. gar nicht möglich ist, alle Meßfunktionen innerhalb einer Modellkomponente durchzuführen. Beispiele hierfür wären das Zählen von Nachrichten an einer bestimmten Stelle des Modells oder die Messung von Durchlaufzeiten zwischen mehreren Modellkomponenten. Zur Lösung dieser Probleme wurden spezielle Meßgeräteklassen entworfen, die im folgenden kurz vorgestellt werden sollen. Bild 4.19 zeigt die Klassenhierarchie der Standardmeßgeräte. Weitere spezielle Meßgeräte können durch Vererbung jederzeit abgeleitet werden.

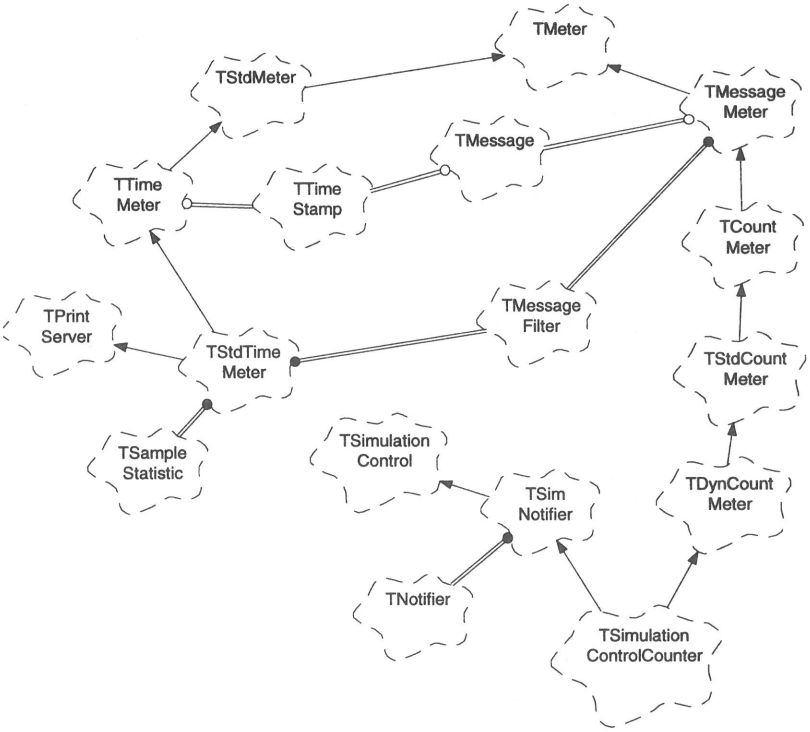



**Bild 4.19:** Klassendiagramm der Standard-Meßgeräte

### 4.10.1 Zähler

In vielen Fällen ist es nützlich, die Anzahl von Nachrichten zu kennen, die eine bestimmte Stelle des Modells durchlaufen. Oft wird z.B. das Teiltestende dadurch bestimmt, daß eine vorgegebene Anzahl von Nachrichten bearbeitet wurde. Die Zählerklasse erlaubt es, die Anzahl von Nachrichten zu ermitteln, die an einem beliebigen Port des Modells übergeben werden. Es ist sogar möglich, gleichzeitig mehrere Ports zu beobachten und die Ergebnisse aufzusummieren. Dazu installiert die *TCountMeter*-Klasse an jedem interessierenden Port einen Nachrichtenfilter, der alle durchlaufenden Nachrichten registriert und sie der Zählerklasse zur Auswertung übergibt. Auf diese Weise kann an jeder beliebigen Stelle des Modells gemessen werden, ohne in bestehende Modellkomponenten eingreifen zu müssen. *TStdCountMeter* unterstützt das Zählen der Nachrichten an einem einzigen Port, während

*TDynCountMeter* den Durchfluß gleichzeitig an mehreren Stellen messen und aufsummieren kann. Speziell abgeleitete Zählerklassen würden auch das Zählen von Nachrichten eines bestimmten Typs erlauben. Kombiniert man eine Zählerklasse mit einer *TNotifier*-Klasse, so kann der Zähler zur Simulationssteuerung eingesetzt werden. Nach Erreichen einer Mindestanzahl von Nachrichten kann z.B. das Teiltestende signalisiert werden. Die Klasse *TSimulationControlCounter* stellt einen solchen Zähler zur Verfügung, bei dem die Anzahl durchfließender Nachrichten für jede Simulationsphase getrennt festgelegt werden kann.

### 4.10.2 Messung von Durchlaufzeiten

Um die Durchlaufzeiten innerhalb einer Komponente z.B. einer Warteschlange zu erfassen, sind in der Regel keine besonderen Vorkehrungen zu treffen, da alle benötigten Informationen direkt vorliegen. Schwieriger wird diese Aufgabe, wenn die Messung der Durchlaufzeit von Nachrichten über mehrere Modellkomponenten hinweg vorgenommen werden soll. Zum Glück kann man mit Hilfe von Nachrichtenfiltern auch hier eine sehr flexible Lösung finden. Am Startpunkt der Messung wird ein Nachrichtenfilter installiert, der jede Nachricht mit einem Zeitstempel versieht. Am Zielpunkt der Messung wird dieser Zeitstempel mit Hilfe eines weiteren Nachrichtenfilters ausgewertet und dem Meßgerät übergeben. Da es möglich ist, mehrere zum Teil überlappende Messungen gleichzeitig durchzuführen, wird jeder Zeitstempel zusätzlich mit einer Meßgerätekennung versehen, so daß eine eindeutige Zuordnung zwischen Zeitstempel und Messung besteht. Die Nachrichtenklasse *TMessage* stellt für diesen Zweck Methoden zur Zeitstempelverwaltung zur Verfügung. Die Basisklasse *TTimeMeter* erzeugt und verwaltet aus Effizienzgründen die Zeitstempel der Klasse *TTimeStamp* mit Hilfe einer sog. Freiliste. *TStdTimeMeter* stellt ein komplettes Meßgerät zur Verfügung, das zwischen beliebigen Punkten messen kann. Die von einer Statistik ausgewerteten Meßwerte können anschließend ausgegeben werden. Durch Installation mehrerer Nachrichtenfilter an verschiedenen Zielpunkten lassen sich auch Alternativwege berücksichtigen. Insgesamt ergibt sich ein sehr flexibles Konzept, das sich mit minimalem Aufwand selbst nachträglich zur Laufzeit in jede bestehende Simulation integrieren läßt.

## 4.11 Ergebnisaufbereitung

Ein Problem bei der Erstellung wiederverwendbarer Modellkomponenten stellt die Aufbereitung und Ausgabe der Simulationsergebnisse dar. Obwohl man durch Angabe des Namens einer Modellkomponente die Ergebnisse verschiedener Komponenten unterscheiden kann, reicht es normalerweise nicht aus, eine Standardausgabe für jede Komponente zu programmieren. Alternativ dazu könnte man in jeder Modellkomponente eine virtuelle Druckmethode definieren, die bei Bedarf überschrieben werden kann. Dies würde jedoch dazu führen, daß

von vielen funktionell gleichen Modellkomponenten neue abgeleitet werden müßten, nur um die Ergebnisausgabe anzupassen. Dies ist aus mehreren Gründen nicht wünschenswert. Erstens würde dadurch die Anzahl der Klassen in einem Simulationsprogramm drastisch ansteigen, was zu Lasten der Übersichtlichkeit ginge. Zweitens würde sich der Aufwand zum Programmieren, Verwalten und Warten dieser Komponenten entsprechend erhöhen. Drittens wäre dadurch die direkte Wiederverwendbarkeit von Modellkomponenten stark eingeschränkt.

Aus diesem Grund wurde ein völlig anderes Ausgabekonzept verwirklicht. Dabei wird ähnlich dem von Serienbriefen bekannten Konzept vorgegangen, Variablen in einen vorgegebenen Text einzufügen. Jede Modellkomponente bezeichnet dabei ihre Ergebnisse durch einzelne Schlüsselworte. Der Anwender kann nun Druckformate definieren, die aus einer Mischung von normalem Text, Schlüsselworten und anderen Druckformaten bestehen. Bei der eigentlichen Ausgabeoperation werden die Schlüsselworte automatisch durch die berechneten Ergebnisse ersetzt. Auf diese Weise erhält man ein extrem flexibles Ausgabekonzept, das sich dynamisch an verschiedene Ausgabewünsche anpassen läßt. Es ist sogar möglich, die Druckformate von einer Textdatei einzulesen, die zur Laufzeit interpretiert wird und so die Ausgabe anzupassen, ohne das Simulationsprogramm zu verändern oder neu compilieren zu müssen.

### 4.11.1 Das Ausgabekonzept

Das Ausgabekonzept wird durch das Zusammenwirken von drei Klassen erreicht. Die Klasse *TPrintFormat* definiert die zur Ausgabe nötigen Druckformate. Ein Druckformat besteht aus dem Druckformatnamen und einem beliebigen Text, der auch die Namen anderer Druckformate sowie Schlüsselworte enthalten darf. *TPrintServer* ist eine abstrakte Basisklasse, von der alle Klassen, welche an der Ausgabe teilnehmen wollen, abgeleitet sein müssen. Diese Klasse nimmt die Umsetzung von Schlüsselworten zu Ergebnissen vor. Die Klasse *TPrintManager* schließlich verwaltet alle Druckformate und Server und ist für die Steuerung der Ausgabe verantwortlich. Im Prinzip stellt diese Klasse einen Interpreter dar. Um die Ergebnisse auf einen beliebigen Ausgabe-Stream auszugeben, ruft man die *PrintResults*-Methode auf. Als Parameter übergibt man den Stream sowie den Namen des Druckformates, das zur Anwendung kommen soll. Die Methode gibt den Text des Druckformates auf den Stream aus. Sobald sie auf ein Schlüsselwort, den Namen eines Servers bzw. eines Druckformates stößt, wird diese Information ausgewertet und die damit verbundenen Ausgaben vorgenommen. Anschließend wird der weitere Text ausgegeben, bis entweder ein neues Sonderzeichen auftritt oder das Druckformat vollständig ausgegeben wurde. Schlüsselworte bzw. Namen werden durch ein %-Zeichen eingeleitet (ähnlich der *printf*-Funktion der Standard-C-Bibliothek). Druckformatnamen werden in geschweifte Klammern ({}) eingeschlossen, Servernamen in spitze Klammern (<>) und Schlüsselworte in eckige Klammern ([]) gesetzt. Sobald

ein Druckformatname erkannt wird, wird der Inhalt des Druckformates ausgegeben, so daß Druckformate rekursiv geschachtelt werden können. Wird der Name eines Servers erkannt, so wird im folgenden dieser Server als neuer Kontext verwendet. Nach dem Servernamen kann in Klammern eingeschlossen ein neues Druckformat angegeben werden, das zur Ausgabe des Servers verwendet wird. Sobald ein Schlüsselwort erkannt wurde, wird es zur Auswertung an den momentan gültigen Server übergeben. Die Auswertung erfolgt kontextsensitiv durch Aufruf der *PrintItem*-Methode des jeweiligen PrintServers. Von *TPrintServer* abgeleitete Klassen können diese Methode verwenden, um Schlüsselworte zu erkennen und in eine entsprechende Ausgabe umzusetzen. Dabei kann es auch passieren, daß die Umsetzung einiger Schlüsselworte an eingebettete Objekte bzw. an die Oberklasse delegiert werden. Unbekannte Schlüsselworte führen zum Abbruch des Ausgabevorgangs.

Bei der Initialisierung der TPrintServer-Objekte melden diese sich automatisch beim globalen Druckmanager an. Der Name ist frei wählbar, bei Modellkomponenten wird der Name der Komponente verwendet. Die Druckformate können wahlweise einzeln beim Druckmanager registriert werden, oder sie können von einem Eingabe-Stream eingelesen werden. Jedem *TPrintServer*-Objekt kann der Name eines Standarddruckformates zugeordnet werden, das verwendet wird, wenn beim Ausdruck kein anderes Format angegeben wird. Dies hat den Vorteil, daß neue Druckformate nur für die Komponenten generiert werden müssen, bei denen das Standardformat unzureichend ist. Außerdem kann die Standardausgabe für alle Komponenten einfach geändert werden, indem die jeweiligen Standarddruckformate angepaßt werden. Der Name des Standarddruckformates entspricht normalerweise dem Klassennamen, kann jedoch beliebig gewählt werden.

Insgesamt ergibt sich ein äußert flexibles Druckkonzept, das sich leicht an eigene Bedürfnisse anpassen läßt. Insbesondere unterstützt es die Wiederverwendbarkeit von Modellkomponenten, da nicht für jede Ausgabevariante neue Komponenten definiert werden müssen.

## 4.12 Entwicklungsunterstützung

Für die effiziente Entwicklung objektorientierter Simulationsprogramme sind zunächst alle Hilfsmittel nützlich, die allgemein den Entwurf objektorientierter Software erleichtern, insbesondere Browser zur Anzeige von Klassenhierarchien und CASE-Tools, die den OOA/OOD-Ansatz unterstützen. Darüber hinaus sollte die Architektur der Simulationsbibliothek grundsätzlich möglichst viele Fehlerquellen ausschließen. Dies geschieht zum einen durch die klare Gliederung der Konzepte in

- hierarchische Modellkomponenten
- Nachrichtenaustausch mittels Port-Konzept
- hierarchischer Ereignisbearbeitung

und deren Zusammenwirken. Die einheitliche Handhabung dieser Konzepte erleichtert das Verständnis und schließt viele grundsätzliche Fehler von vornherein aus. Zum anderen wird die Einhaltung der Konzepte durch die strenge Typprüfung von C++ weitgehend erzwungen. Wo dies zur Compilierzeit nicht möglich ist, wurden Laufzeitüberprüfungen vorgesehen, wobei möglichst viele Plausibilitätsprüfungen vorgenommen werden. Da die meisten Prüfungen beim Aufbau des Modells, also vor dem eigentlichen Start der Simulation, erfolgen, haben sie keine negativen Auswirkungen auf die Programmlaufzeit.

Um das Anwenderprogramm nicht mit Abfragen, die die erfolgreiche Ausführung einer Aktion überprüfen, zu überfrachten, wurde der Ausnahmebehandlungsmechanismus von C++ verwendet. Das eigentliche Anwenderprogramm kann so programmiert werden, als ob keine Fehler auftreten könnten. Immer wenn das Programm einen Laufzeitfehler erkennt, wird eine entsprechende Ausnahme signalisiert. Meist bedeuten diese Ausnahmen fatale Fehler im Entwurf des Simulationsprogrammes, z.B. das Verbinden nicht vorhandener Ports, so daß das Simulationsprogramm an dieser Stelle abgebrochen und korrigiert werden muß. In anderen Fällen kann eine Ausnahme vom Programm bearbeitet und anschließend das Programm fortgeführt werden. Mit Hilfe der Ausnahmebehandlung wird dem Anwender ein Fehler auf jeden Fall mitgeteilt, ohne daß zusätzliche Maßnahmen erforderlich wären. Im Gegensatz dazu können durch vergessene Abfragen manche Fehler gar nicht oder erst sehr spät entdeckt werden. Das Verwenden von Ausnahmen erleichtert somit nicht nur die Anwenderprogrammierung, sondern stellt auch die frühzeitige Erkennung von Fehlern sicher und steigert damit die Softwarequalität.

Konzeptionelle Fehler, wie die falsche Wahl von Modellkomponenten oder die falsche Implementierung eines Algorithmuses innerhalb einer Modellkomponente, können mit Hilfe dieser Verfahren natürlich nicht erkannt werden. Dies muß nach wie vor im Rahmen der Validierung des Modells und während der Testphase der Softwareentwicklung erfolgen. Zusätzlich zu den Möglichkeiten, die moderne Debugger in diesem Bereich bieten, wurde ein umfangreiches Trace-Konzept integriert [36], welches es erlaubt, Meldungen selektiv für bestimmte Modellkomponenten oder Nachrichten während des Simulationslaufes auszugeben. Damit ist es möglich, während der Testphase bestimmte Modellkomponenten bzw. ausgesuchte Nachrichten innerhalb des Systems, genauer zu untersuchen. Eine Trace-Maske erlaubt es, die Meldungen ganz an- und abzuschalten bzw. den Detaillierungsgrad der Ausgabe zu bestimmen. Da die Masken dynamisch verändert werden können, kann die Trace-Ausgabe flexibel auf das Eintreten bestimmter Ereignisse reagieren. Die Verifikation bestimmter Abläufe innerhalb der Simulation wird auf diese Weise sehr erleichtert.

Insgesamt wird der Entwurf von Simulationsprogrammen weitgehend unterstützt, wobei aufgrund der Problemabhängigkeit einige Aufgaben weiterhin beim Anwender der Simulationsbibliothek verbleiben.

# Kapitel 5

# Anwendung und Bewertung der Simulationsbibliothek

Dieses Kapitel beschreibt die Erfahrungen, die bei der Entwicklung und Anwendung der Simulationsbibliothek gemacht wurden. Zunächst wird anhand eines konkreten Beispiels ein konventionell in der Sprache Pascal geschriebenes Simulationsprogramm einer objektorientierten Lösung gegenübergestellt. Anschließend wird demonstriert, wie einfach sich das objektorientierte Programm bei neuen Anforderungen erweitern läßt. Außerdem werden konkrete Wege zur Erweiterung der Simulationsbibliothek für parallele Simulationen aufgezeigt. Abschließend werden einige interessante Aspekte der Simulationsbibliothek hervorgehoben.

## 5.1 Simulation eines Satellitensystems

### 5.1.1 Modular On-Board Switching System

„MOBS" steht für „Modular On-Board Switching System". Es handelt sich hierbei um eine Studie, die in Zusammenarbeit mit der Firma ANT für das BMFT erstellt wurde. MOBS ist ein modulares Vermittlungssystem, das für die Anforderungen einer Satellitenanwendung optimiert wurde. Der Satellit dient dabei erstmals als Vermittlungsstelle, um eine Vielzahl von Erdeterminals mit teilweise unterschiedlichen Kommunikationsprotokollen miteinander zu verbinden. Aufgrund der begrenzten Rechenleistung, die an Bord eines Satelliten zur Verfügung steht, können aufwendige herkömmliche Signalisiersysteme wie z.B. das Signalisiersystem CCITT No. 7 [13], nicht verwendet werden. Im Rahmen des Projektes wurde deshalb ein eigenes Protokoll zur Signalisierung entworfen. Dies zeichnet sich durch Einfachheit und Flexibilität aus. Es unterscheidet im wesentlichen zwei Pakettypen. Sogenannte Control Messages (CM) werden verwendet, um Verbindungen zwischen Terminals und dem Satelliten auf- bzw. abzubauen. Mit Hilfe von User Messages (UM) können Signalisiernachrichten

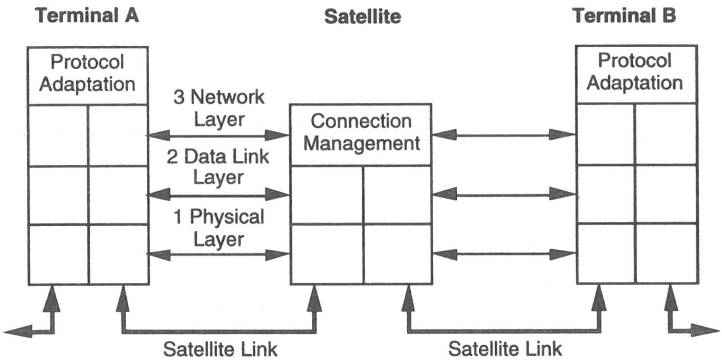



**Bild 5.1:** Protokollarchitektur des Satellitensystems

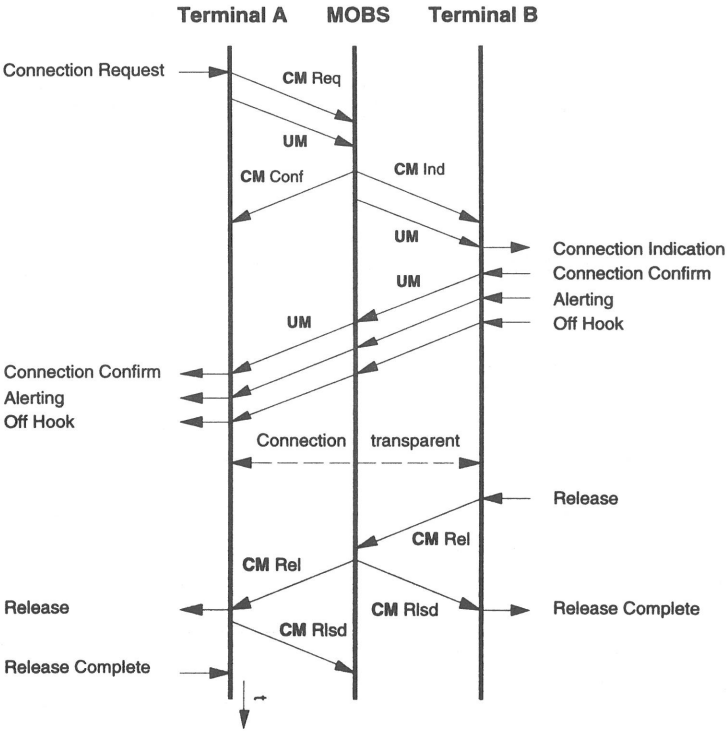



**Bild 5.2:** Signalisierszenario eines vollständigen Verbindungsauf- und -abbaues

zwischen den beteiligten Endsystemen ausgetauscht werden. Alle User Messages werden transparent über den Satelliten übertragen. Die Pakete haben eine feste Größe von 32 Bytes und sind durch Prüfsummen und Reihenfolgesicherungsnummern gegen Störungen auf dem Kanal geschützt. Auf der Schicht 2 wird dabei ein HDLC-Protokoll [62] mit Selective Repeat verwendet. Längere Nachrichten können durch Kettung mehrerer Pakete übertragen werden. Der Satellit stellt einen Vermittlungsknoten im Netz dar und benötigt daher nur die Protokollschichten 1 - 3 des ISO/OSI-Referenzmodelles. Bild 5.1 zeigt die Protokollarchitektur des Satellitensystems. In den Erdeterminals erfolgt die Umsetzung von anwendungsspezifischen Protokollen auf das spezielle Satellitenprotokoll.

Bild 5.2 zeigt das Signalisierszenario für einen erfolgreichen Verbindungsauf- und -abbau, wobei die Umsetzung anwenderprotokollspezifischer Nachrichten in User Messages zu sehen ist.

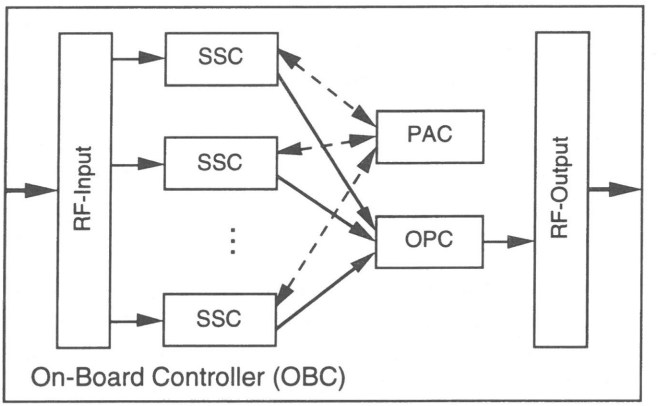



**Bild 5.3:** Blockschaltbild des Satellitensystems

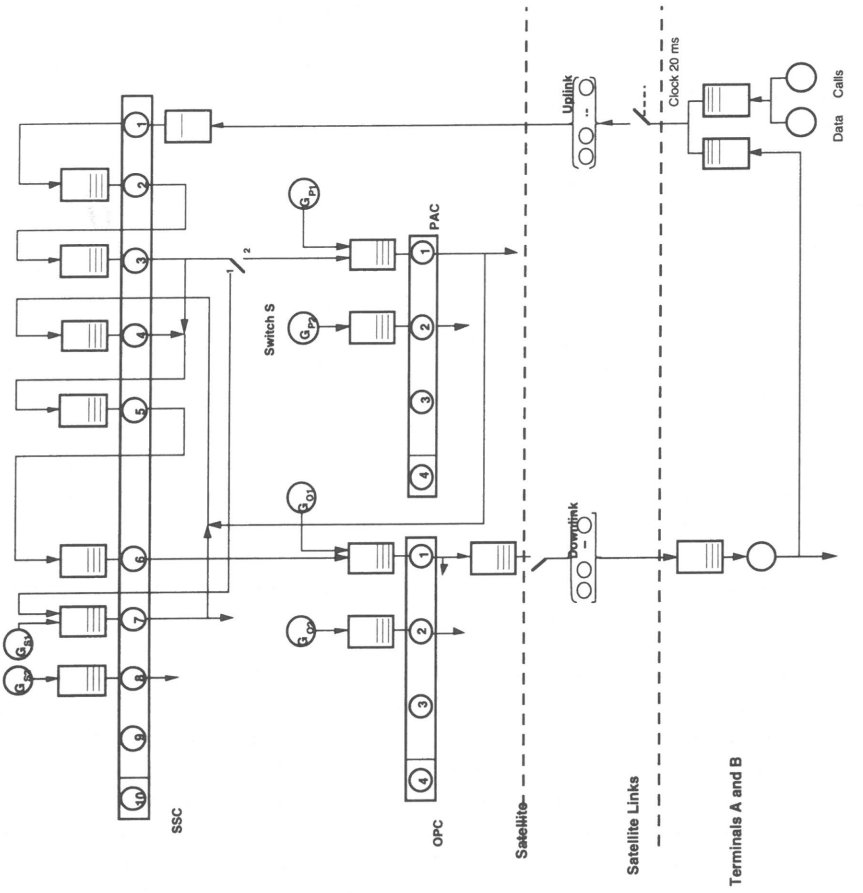
Innerhalb des Satelliten erfolgt die Vermittlung mit Hilfe eines Multiprozessorsystems, dem „On-Board Controller" (OBC), das sich die Aufgaben der Vermittlung teilt, vgl. Bild 5.3. Nach der Demodulation am Eingang werden die Pakete auf die „Signalling and Switching Controller" (SSC) verteilt, die für die Verwaltung der Verbindungen verantwortlich sind. Jeder SSC bekommt eine Reihe von Erdeterminals zugeteilt. Ein „Path Allocation Controller" (PAC) verwaltet die Ressourcen, die zu jeder Verbindung gehören, zentral für den gesamten Satelliten, um Probleme bei der parallelen Belegung von Betriebsmitteln durch mehrere SSCs zu vermeiden. Ein „Output Processing Controller" (OPC) ist für die Aufbereitung und das Senden von Paketen zwischen Satellit und Erdeterminals verantwortlich. Der abschließende Modulator sorgt für die Kanalkodierung und HF-Anpassung des Signals. Bei Ausfall des

PACs können dessen Aufgaben von einem der SSCs mit übernommen werden. Eine ausführlichere Beschreibung findet sich in [53].

### 5.1.2 Simulationsmodell

Ein im Rahmen des Projektes erstelltes Simulationsprogramm hat zwei Aufgaben. Zum einen soll die Funktionsweise des Signalisierprotokolls bestätigt, zum anderen die Leistungsfähigkeit des Gesamtsystems, insbesondere des Satellitenteils, betrachtet werden. Es handelt sich also um eine kombinierte funktionelle und Performance-Simulation. Die Abläufe innerhalb des Satelliten müssen daher genau nachgebildet werden, was zu dem in Bild 5.4 gezeigten ausführlichen Modell führt. Zum Aufbau einer Verbindung werden zwei Terminals A und B benötigt. In Terminal A erzeugt ein Rufgenerator Verbindungsaufbauwünsche für das Satellitensystem. Die User Messages werden durch einen zweiten Datengenerator simuliert. Terminal B empfängt Verbindungsauf- und -abbauwünsche und reagiert auf diese. Der Verbindungsabbau muß an den Satelliten bestätigt werden. Die Nachrichten von Terminal A und B werden kombiniert und über einen sog. Infinite Server, der die Laufzeit der Satellitenstrecke simuliert, an den SSC geschickt. Ein getaktetes Element vor der Satellitenstrecke simuliert den getakteten Zugriff auf den Satelliten. Innerhalb des Satelliten sind die einzelnen Arbeitsphasen zur Abarbeitung des Protokolls genau nachgebildet. Im wesentlichen gibt es Phasen zur Reihenfolgesicherung der Pakete in den Schichten 2 und 3, der Abarbeitung von CM und UM und zum Zusammenfügen neuer Pakete für das Weitersenden an die Erdeterminals.

In der Simulation hat jede Phase eine vom jeweiligen Nachrichtentyp abhängige Bedienzeit, die von der Eingabedatei gelesen werden kann. Ebenso kann die Reihenfolge der Abarbeitung der Phasen eines Prozessors in der Eingabedatei festgelegt werden. Der Schalter S erlaubt die Simulation von getrennten PAC und SSCs und der Version, bei der der PAC in einen SSC integriert ist. Es wird nur ein SSC simuliert. Die zusätzliche Last weiterer Terminals und SSCs wird durch mehrere Lastgeneratoren innerhalb des Modells nachgebildet. Das Simulationsprogramm hat die Aufgabe, das Signalisierprotokoll funktionell zu überprüfen und die Verbindungsaufbauzeit bei realistischer Verkehrslast zu bestimmen.




| Prozessor | Phase | Aufgabe | Prozessor | Phase | Aufgabe |
|---|---|---|---|---|---|
| SSC | ① | Empfang Schicht 2 | SSC | ② | Empfang Schicht 3 |
| SSC | ③ | Vermittlung UM<br>Bearbeitung CM | SSC | ④ | Generierung CM<br>(Conf, Ind, Rel, Rlsd) |
| SSC | ⑤ | Senden Schicht 3 | SSC | ⑥ | Senden Schicht 2 |
| SSC | ⑦ | PAC-Phase bei PAC in SSC | SSC | ⑧ | Lastphase |
| SSC | ⑨ | Overhead Allgemein | SSC | ⑩ | Overhead Phasenumschaltung |
| PAC | ① | Interne Ressourcenverwaltung | OPC | ① | Paketaufbereitung für Senden |
| PAC, OPC | ② | Lastphase | PAC, OPC | ③ | Overhead Allgemein |
| PAC, OPC | ④ | Overhead Phasenumschaltung | $G_{XX}$ | | Lastgeneratoren |

**Bild 5.4:** Simulationsmodell des Satellitensystems

### 5.1.3 Entwurf des Simulationsprogrammes

Wie bereits in Kapitel 4 hervorgehoben wurde, kann die Aufteilung des Simulationsprogrammes in einzelne Einheiten beim objektorientierten Ansatz in natürlicher Weise erfolgen. Aus dem Simulationsmodell folgt sofort eine Hierarchie von Modellkomponenten. Eine grobe Aufteilung liefert zunächst die Komponenten Terminal, Satellit und Verbindungskanal. Der Satellit wird weiter unterteilt in die Prozessoren SSC, PAC und OPC. Alle diese Modellkomponenten kann man im Simulationsprogramm als Objekte wiederfinden.

Der Vorteil eines fertigen Rahmens, den die Simulationsbibliothek bietet, zeigt sich bereits während der Entwicklungsphase. Aufgrund der Abgeschlossenheit der einzelnen Modellkomponenten, kann zu jedem Zeitpunkt der Implementierung ein Teilmodell gebildet werden, das getrennt simuliert werden kann. Später werden die einzelnen Modellkomponenten in hierarchischen Modellkomponenten zusammengefaßt und gemeinsam simuliert. Diese evolutionäre Art der Programmentwicklung ist einer der wesentlichen Vorteile gegenüber der konventionellen Vorgehensweise. Zu jedem Zeitpunkt existiert ein lauffähiges Programm. Der Test vereinfacht sich, da bereits einzelne Komponenten getrennt getestet werden können. Das Problem, ein komplexes Simulationsprogramm, das aus vielen Komponenten besteht, auf einmal zu integrieren, existiert nicht. Da die einzelnen Komponenten bereits ausgetestet sind, kann die Gesamtlogik des Simulationsprogrammes leichter verifiziert werden. Dies ist ein nicht zu unterschätzender Vorteil.

Während der Entwurfsphase stellt sich die Frage, welche Komponenten neu entwickelt werden müssen. Für gängige Probleme wie Warteschlangen oder Laufzeitglieder (Infinite Server) stellt bereits die Basisversion der Simulationsbibliothek entsprechende Modellkomponenten zur Verfügung, die sofort wiederverwendet werden können. Andere Komponenten sind sehr problemspezifisch und müssen daher neu entwickelt werden. Es hat sich gezeigt, daß neue Komponenten oft durch Vererbung und einfachen Modifikationen aus bestehenden Standardkomponenten gewonnen werden können. So wurde z.B. der Datengenerator des Terminals von einem Standardgenerator abgeleitet und um die Logik zur Reihenfolgesicherung auf Schicht 2 und 3 erweitert. Auf der anderen Seite bieten sich während des Entwurfs häufig Möglichkeiten, durch Verallgemeinerung neue Standardkomponenten zu schaffen. So wurde ein allgemeines Prozessormodell mit beliebigen Phasen entwickelt, mit dem sich alle Prozessoren innerhalb des Satelliten modellieren ließen, das gleichzeitig aber so allgemein gehalten wurde, daß es als neue Standardkomponente in die Simulationsbibliothek aufgenommen werden konnte. Die Vorausplanung, problemspezifische Modellkomponenten zu verallgemeinern, um sie später wiederverwenden zu können, erfordert zunächst einen gewissen Mehraufwand. Dieser zahlt sich langfristig in neuen Projekten durch weniger Entwurfsaufwand aus. Auch ein kurzfristiger Nutzen ist zu verzeichnen. Durch die Verallgemeinerung erhält man meist eine klarere Abstraktion, so daß die Aufteilung des Problems übersichtlicher wird. Für das Beispiel konnten die allgemeinen Prozessorkomponenten sofort eingesetzt

werden. Die problemabhängigen Prozessorphasen wurden durch Vererbung und Modifikation aus den allgemeinen Phasen gewonnen. Durch die bessere Abstraktion konnten selbst die einzelnen Phasen sofort wiederverwendet werden. Die Bearbeitungsphase des PACs kann einerseits als Phase im PAC vorkommen; sie kann aber auch ein Teil des SSCs sein, wenn dieser die Aufgaben des PAC mit übernehmen muß. Während bei konventioneller Programmierweise die schlechte Kapselung und Typinkompatibilitäten dazu führten, daß große Teile des Codes dupliziert werden mußten, konnte beim objektorientierten Ansatz die Phase problemlos in beiden Prozessoren verwendet werden. Dies führt nicht nur zur Einsparung von Code, sondern auch zu besserer Wart- und Testbarkeit, da sichergestellt ist, daß der gleiche Code in beiden Fällen zur Anwendung kommt. Die Konsistenz ist somit von vornherein gesichert.

Während der Implementierungsphase hat sich das flexible Meß- und Filterkonzept sehr bewährt. Da Meßgeräte und Nachrichtenfilter ohne Eingriff in bestehende Modellkomponenten an jedem Port installiert werden können, lassen sich zu jedem Zeitpunkt auf einfache Art Messungen zwischen beliebigen Punkten im Modell durchführen. Auch die in der Simulation geforderte Messung der Verbindungsaufbauzeit konnte mit Hilfe eines Meßgerätes der Klasse *TStdTimeMeter* realisiert werden, das die Zeit vom Rufgenerator bis zur Ankunft der Bestätigungsmeldung im Terminal mißt (vgl. Kap. 4.10.2). Durch Installation eines Message-Trace-Filters können an jedem Port alle durchfließenden Nachrichten beobachtet werden, was während der Testphase sehr nützlich ist.

Erst nachdem das Modell mit im Code fest vorgegebenen Parametern bereits fehlerfrei funktionierte, wurde zum Einlesen der Simulationsparameter eine Hierarchie von Parserklassen entworfen, die die Hierarchie der Modellkomponenten nachbildet. Da die Parserklassen von der allgemeinen Klasse *TParserObject* abgeleitet sind, verfügen sie automatisch über alle Eigenschaften der allgemeinen Parserklasse der Simulationsbibliothek, wie kontextsensitive Hilfe oder formatfreies Einlesen von Parametern. Der Aufwand zum Einlesen aller Parameter war dann auch sehr gering.

Die Ausgabe der Ergebnisse erforderte gar keinen zusätzlichen Code, sondern nur die Erstellung einer einfachen Textdatei, in der die Durckformatangaben definiert wurden. Durch Anpassung dieser Datei kann die Ausgabe beliebig verändert werden.

Insgesamt konnte die Beispielsimulation in sehr kurzer Zeit implementiert werden und produzierte auf Anhieb fehlerfreie Ergebnisse. Im nächsten Abschnitt sollen beide Implementierungen sowohl qualitativ als auch quantitativ miteinander verglichen werden.

## 5.1.4 Vergleich mit konventioneller Implementierung

### 5.1.4.1 Aufwand

Das Pascal-Programm ist in [65] beschrieben. Es verwendet die am Institut entwickelte Bibliothek zur Erzeugung von Zufallszahlen und zur Statistikauswertung. Weitere Teile wie Kalender und Warteschlangen wurden aus bestehenden Simulationsprogrammen übernommen und den Bedürfnissen entsprechend angepaßt. Dies trifft die Definition von Wiederverwendbarkeit nicht ganz, da der ursprüngliche Code hierzu verändert (bzw. kopiert) werden muß. Es wird nur Schreibarbeit jedoch kein Entwicklungsaufwand eingespart, da die „wiederverwendeten" Module neu durchdacht, angepaßt und komplett neu getestet werden müssen. Das Programm ist in 9 Module unterteilt, die nach funktionellen Gesichtspunkten organisiert sind, z.B. Ein-/Ausgabe, Ereignisbearbeitung, etc.

Das objektorientierte Pendant dazu verwendet die in dieser Arbeit vorgestellte Simulationsbibliothek. Zusätzlich wurden 19 kleinere Dateien erstellt, die nach Modellkomponenten organisiert sind, z.B. gibt es eigene Dateien für die Prozessoren, das Terminal, die Generatoren, usw.

Die folgenden Schaubilder sollen einen Überblick über den Codeumfang beider Programme geben. Bild 5.5 vergleicht die Anzahl der Codezeilen und die Anzahl der Statements. Beide Programme wurden einheitlich formatiert. Die Kommentare wurden entfernt, so daß sich unterschiedliche Kodierstile nicht auswirken. Das objektorientierte Simulationsprogramm (OOS) ist in zwei Varianten abgebildet: Die erste Variante (OOS 1) beinhaltet den gesamten Code, der für das Beispielprogramm geschrieben wurde. In der zweiten Variante (OOS 2) wurde der wiederverwendbare Code nicht berücksichtigt. Es sind dies die Prozessorabstraktion und das getaktete Gate, die beide allgemein weiterverwendet werden können. Wie man dem Bild entnehmen kann, spart der Einsatz der Simulationsbibliothek viel Aufwand. Der Codeumfang ist bei konventioneller Programmierung um 27%, bzw. bei Abzug der wiederverwendbaren Teile sogar um 42% größer als beim Einsatz der Simulationsbibliothek. Beim Vergleich der Anzahl der Statements fällt der Unterschied noch drastischer aus: Die konventionelle Lösung benötigt sogar 2,7 (2,9) mal soviel Statements wie die objektorientierte Lösung. Der Grund, daß mehr Statements als Codezeilen eingespart wurden, liegt darin begründet, daß in C++ alle Klassen und Funktionen vor dem ersten Gebrauch deklariert sein müssen. Bei der Definition werden diese Teile wiederholt. Dieses Konzept erlaubt die Aufspaltung von Dateien in Deklarationen und Definitionen, die getrennt compiliert werden können. Es handelt sich hierbei zwar um vermehrte Schreibarbeit, die jedoch keinen zusätzlichen Entwicklungsaufwand bedeutet. Bild 5.6 zeigt die prozentuale Codeaufteilung in Deklarationen, Definitionen und Compileranweisungen. Letztere bestehen hauptsächlich aus Anweisungen an den C++-Präprozessor, um andere Dateien einzufügen und sind eine Folge der modularen Aufteilung der Klassen in einzelne Dateien. Zum Vergleich enthält das Bild

die Aufteilung von Typdeklarationen und normalem Code für das Pascal-Programm. Wie man sieht, ist der Overhead für die C++-Version beträchtlich, was sich allerdings nicht auf die Komplexität des Codes auswirkt, sondern nur eine erhöhte Redundanz bedeutet, die vom Compiler zu Prüfungen über Modulgrenzen hinweg genutzt werden kann. Vergleicht man den reinen Codeanteil beider Programme miteinander, bekommt man wieder ein ähnliches Verhältnis wie beim Vergleich der Statements.

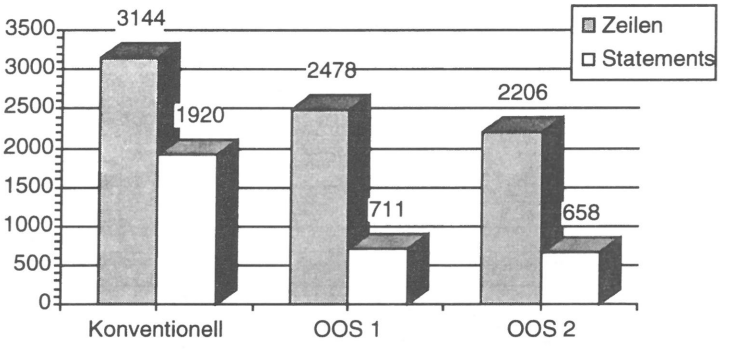



**Bild 5.5:** Vergleich der Beispielprogramme anhand von Programmzeilen und Anzahl der ausführbaren Statements

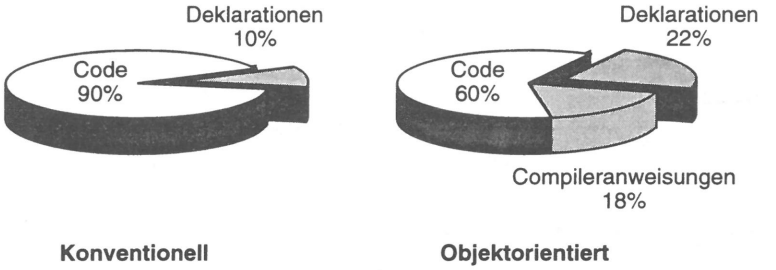



**Bild 5.6:** Verhältnis zwischen Vereinbarungen und ausführbarem Code

Bei der Betrachtung beider Programme fallen noch weitere interessante Eigenschaften auf. Das Pascal-Programm besteht aus nur 65 Prozeduren und Funktionen, die teilweise sehr umfangreich sind. Im Gegensatz dazu ist das objektorientierte Programm in sehr viele Klassen und Funktionen unterteilt, die jedoch in ihrer Komplexität viel überschaubarer sind.

Bild 5.7 stellt die Anzahl von Funktionen bzw. Methoden und die Anzahl von Statements pro Funktion einander gegenüber. Die große Anzahl der Methoden im C++-Programm spiegelt die Entwurfsphilosophie objektorientierter Software wieder, die jeder Methode möglichst genau eine überschaubare Aufgabe zuweist. Dies unterstützt die Wiederverwendbarkeit von Klassen, da in einer abgeleiteten Klasse selektiv nur die Methoden überschrieben werden müssen, in denen sich das Verhalten der abgeleiteten Klasse von der ursprünglichen unterscheidet. Sofern eine Methode mehr als eine Aufgabe erfüllt, muß sie überschrieben werden, sobald auch nur eine der Teilaufgaben neu implementiert werden muß. Der Code der übrigen Teilaufgaben wird dann unnötigerweise dupliziert. Deshalb sollte jede Methode nur ein begrenztes Problem lösen. Das Ändern einer Methode ist dadurch überschaubar und erfordert nur sehr wenig neuen Code. Aus diesem Grund verbessert ein modularer Entwurf die Codeeffizienz erheblich.

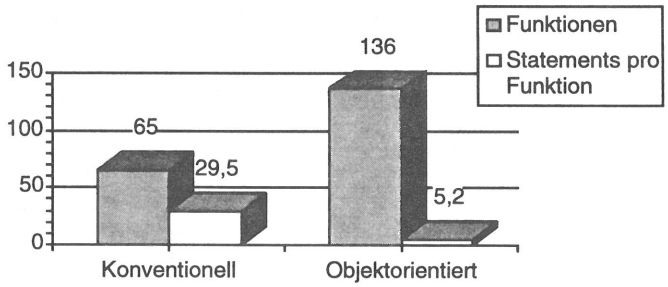



**Bild 5.7:** Vergleich der Anzahl von Funktionen bzw. Methoden und Anzahl Statements pro Funktion.

Auch dieses Beispiel zeigt deutlich, daß bei objektorientierter Software mehr Aufwand in der Entwurfsphase getrieben werden muß. Die eigentliche Kodierung ist meist sehr einfach und überschaubar, was sich auch positiv auf die Fehlerrate auswirkt.

### 5.1.4.2 Allgemeine Erkenntnisse

Im vorhergehenden Abschnitt wurde hauptsächlich ein quantitativer Vergleich beider Simulationsprogramme vorgenommen. Obwohl bereits dieser Vergleich klar für eine objektorientierte Lösung spricht, sollen an dieser Stelle noch einige qualitative Aussagen hinzugefügt werden.

Aufgrund der unvollständigen Dokumentation des ursprünglichen Simulationsprogrammes, mußten einige Vorgänge direkt aus dem Pascal-Code des Programmes nachvollzogen werden, um die Funktionalität der objektorientierten Lösung richtig entwerfen zu können. Dabei hat

sich gezeigt, daß der prozedurale Ansatz das Verständnis des Programmes erschwert. Zum einen sind Anweisungen, die eine Komponente des Simulationsmodells beeinflussen, gemäß der funktionalen Aufteilung des Programmes über viele Funktionen und Module verteilt. Es ist schwierig, alle Programmteile wiederzufinden, die eine bestimmte Komponente betreffen. Der Einsatz globaler Variablen macht es fast unmöglich, alle Stellen zu finden, an denen ihr Wert geändert wird. Zum anderen wird das Verständnis zusätzlich dadurch erschwert, daß sich die einzelnen Modellkomponenten im Programm teilweise nicht direkt wiederfinden, sondern aus funktionellen Gründen zusammengefaßt wurden. Darüberhinaus beeinflussen sie sich oft noch gegenseitig.

Im Gegensatz dazu ist das objektorientierte Simulationsprogramm sehr einfach zu verstehen. Alle Komponenten des Simulationsmodells finden sich auch als Klassen im Programm wieder. Um die Funktion einer Modellkomponente zu verstehen, muß nur die Funktionalität der jeweiligen Klasse verstanden werden. Da alle Methoden einer Klasse normalerweise in einer Datei implementiert werden, ist der Code einer Modellkomponente leicht zu finden und zu verstehen. Aufgrund der Kapselung aller Modellkomponenten erfolgt eine gegenseitige Beeinflussung nur über die explizit vorgesehenen Schnittstellen. Versteckte Änderungen, die nur schwer zu finden sind, gibt es nicht.

Anhand zweier ausgewählter Beispiele soll dies verdeutlicht werden.

- Zur Nachbildung der Reihenfolgesicherung der Schicht 2 werden Zähler benötigt, die die Sende- und Empfangsnummern verwalten. Konzeptionell befinden sich diese Zähler an den Endpunkten der Schicht-2-Verbindungen, nämlich in den Terminals und im Satellit. Aufgrund der funktionalen Aufteilung des ursprünglichen Simulationsprogrammes werden diese Zähler von vielen unterschiedlichen Funktionen manipuliert. Sie mußten deshalb als globale Variablen definiert werden. Es war nicht einfach herauszufinden, an welchen Stellen im Programm welcher Zähler verändert wird. Abgesehen davon würde eine Änderung des Algorithmuses zur Reihenfolgesicherung Änderungen an vielen Stellen des Programmes nach sich ziehen. Bei der objektorientierten Lösung konnten die Zähler genau als Teile der Modellkomponenten verwirklicht werden, zu denen sie konzeptionell gehören. Die Reihenfolgesicherung wird hier wirklich als eine verteilte Lösung implementiert, die sie ja auch im realen System ist. Außerdem wurden die Reihenfolgezähler als abstrakte Datentypen in einer eigenen Klasse definiert, die gleichzeitig den Algorithmus zur Manipulation der Zähler zur Verfügung stellt. Änderungen des Algorithmuses würden sich nur an einer Stelle auswirken und nicht im ganzen Programm. Insgesamt ist dieser Ansatz problemnäher, übersichtlicher, spart Code, ist leichter zu ändern und einfacher zu testen als die konventionelle Implementierung.
- Die Flexibilität des Simulationsprogrammes, sowohl eine Variante mit eigenständigem PAC als auch mit im SSC integriertem PAC simulieren zu können, erfordert die dyna-

mische Erzeugung und Anpassung des Simulationsmodells zur Laufzeit des Programmes. In der Pascal-Version werden nicht nur während der Initialisierungsphase unterschiedliche Datenstrukturen erzeugt, sondern es erfolgen auch während der Laufzeit an verschiedenen Stellen des Programmes Fallunterscheidungen, z.B. um Nachrichten an die richtige Komponente weiterzugeben. Die Fehlermöglichkeiten bei dieser Art der Programmierung sind beträchtlich höher, da immer die Gefahr besteht, die Fallunterscheidung an einer Stelle zu vergessen. Beim Einsatz der Simulationsbibliothek ist die Fallunterscheidung nur an einer Stelle, nämlich bei der Erzeugung und Vernetzung der Modellkomponenten notwendig. Zur Laufzeit des Programmes sind keine zusätzlichen Fallunterscheidungen notwendig. Dies wird durch das allgemeine Port-Konzept ermöglicht, das für eine klare Trennung zwischen den Modellkomponenten sorgt und es erlaubt, beliebige Modellkomponenten miteinander zu verbinden. Während z.B. die Einbeziehung einer Variante, bei der OPC und PAC in einem Prozessor kombiniert würden, schwerwiegende Auswirkungen auf das Pascal-Programm hätte, müßte im objektorientierten Fall nur die Arbeitsphase des OPC in den PAC verlegt und die Vernetzung des Satellitenmodells angepaßt werden. Beides würde lokal in der Satellitenklasse erfolgen und hätte keine weiteren Auswirkungen auf den Rest des Programmes.

Ein weiterer Vorteil der Simulationsbibliothek zeigt sich bei der Aufbereitung und Ausgabe der Ergebnisse. Während im Pascal-Programm umfangreiche Funktionen zur Formatierung der Ausgabe bereitgestellt werden mußten, benötigt die objektorientierte Lösung keinen zusätzlichen Code. Dies ist eine Folge objektorientierter Programmierung, die es ermöglicht, allgemeine Methoden zu schreiben, die Objekte einer Basisklasse und aller von ihr abgeleiteten Klassen gleichermaßen manipulieren. Das Druckkonzept nutzt diesen Umstand, um alle druckbaren Komponenten auszudrucken, selbst solche, die bei der Erstellung der Simulationsbibliothek noch gar nicht existiert haben. Wie bereits in Kapitel 4.11 erläutert wurde, kann der Anwender durch Angabe von Druckformaten bestimmen, welche Modellkomponenten in welchem Format ausgedruckt werden sollen. Die Druckformate können in einer Textdatei abgelegt werden und werden vom Druckmanager der Simulationsbibliothek zur Laufzeit interpretiert. Dies hat den Vorteil, daß Änderungen der Ausgabe keine Programmänderungen mit Neu-Compilierung des Codes erfordern, sondern nur eine Anpassung der Textdatei. Diese Datei benötigt selbst für den Ausdruck komplexer Modellkomponenten nur wenige Zeilen Text. Insgesamt hat sich das Druckkonzept sehr bewährt und nicht unerheblich zur Codeeinsparung beigetragen.

Insgesamt ergeben sich durch den Einsatz der Simulationsbibliothek und Programmierung in einer objektorientierten Programmiersprache eine ganze Reihe von Vorteilen, die abschließend im Überblick zusammengefaßt werden sollen.

- Direkte Abbildung des Simulationsmodells in die Modellkomponenten des Programmes. Dies erhöht die Übersichtlichkeit des Programmes und erleichtert das Verständnis

und die Strukturierung erheblich. Die hierarchische Modellbildung unterstützt die schrittweise Verfeinerung von Modellkomponenten und Teilmodellen.

- Konzentration auf die problemnahen Teile der Simulation möglich. Die grundlegenden Eigenschaften eines Simulationsprogrammes werden bereits von der Simulationsbibliothek erbracht. Es müssen nur die problemabhängigen Teile der Simulation entworfen und neu programmiert werden.
- Codeeinsparung durch Verminderung der Redundanz und Modifikation bestehender Komponenten.
- Geringerer Entwicklungsaufwand durch Wiederverwendung von Standardkomponenten und Einbindung in den vorgegebenen Rahmen der Simulationsbibliothek.
- Ein iterativer Programmentwurf erlaubt das Austesten von Teilmodellen und sorgt dafür, daß bereits in einem sehr frühen Entwicklungsstadium ein lauffähiges Simulationsprogramm zur Verfügung steht. Fehlersuche, Test- und Validierungsaufwand reduzieren sich durch diese Möglichkeit erheblich.
- Änderungen und Erweiterungen des Simulationsprogrammes haben meist nur lokale Auswirkungen. Sie sind deshalb leicht zu implementieren und zu testen. Die Anpassung kann einfach als ein weiterer Schritt des iterativen Programmentwurfs betrachtet werden.

Der letzte Punkt soll im folgenden Kapitel anhand eines Beispiels nochmals genauer betrachtet werden. Insgesamt ergibt sich eine Steigerung der Produktivität auf allen Ebenen, was auch durch andere Forschungsvorhaben bestätigt wird [35].

### 5.1.5 Erweiterung des Simulationsprogrammes

Anhand einer Erweiterung des Simulationsmodells soll gezeigt werden, wie einfach neue Anforderungen in ein bestehendes Simulationsprogramm integriert werden können.

#### 5.1.5.1 Erweitertes Modell

Im ursprünglichen Simulationsmodell wurde die Reihenfolgesicherung auf der Schicht 2 zwar vorgenommen und geprüft, der Fehlerfall und die Auswirkungen des Selective-Repeat-Algorithmuses wurden jedoch nicht betrachtet. Aufgrund der angenommenen geringen Bitfehlerwahrscheinlichkeit von $\leq 10^{-6}$ auf dem Satellitenkanal dürfte sich dies kaum auf die gemessenen Verbindungsaufbauzeiten auswirken. Außerdem stellt sich das Problem, daß sehr viele Ereignisse simuliert werden müssen, um bei dieser geringen Wahrscheinlichkeit gesicherte Aussagen zu erhalten. Dies würde die Laufzeit des Simulationsprogrammes drastisch erhöhen.

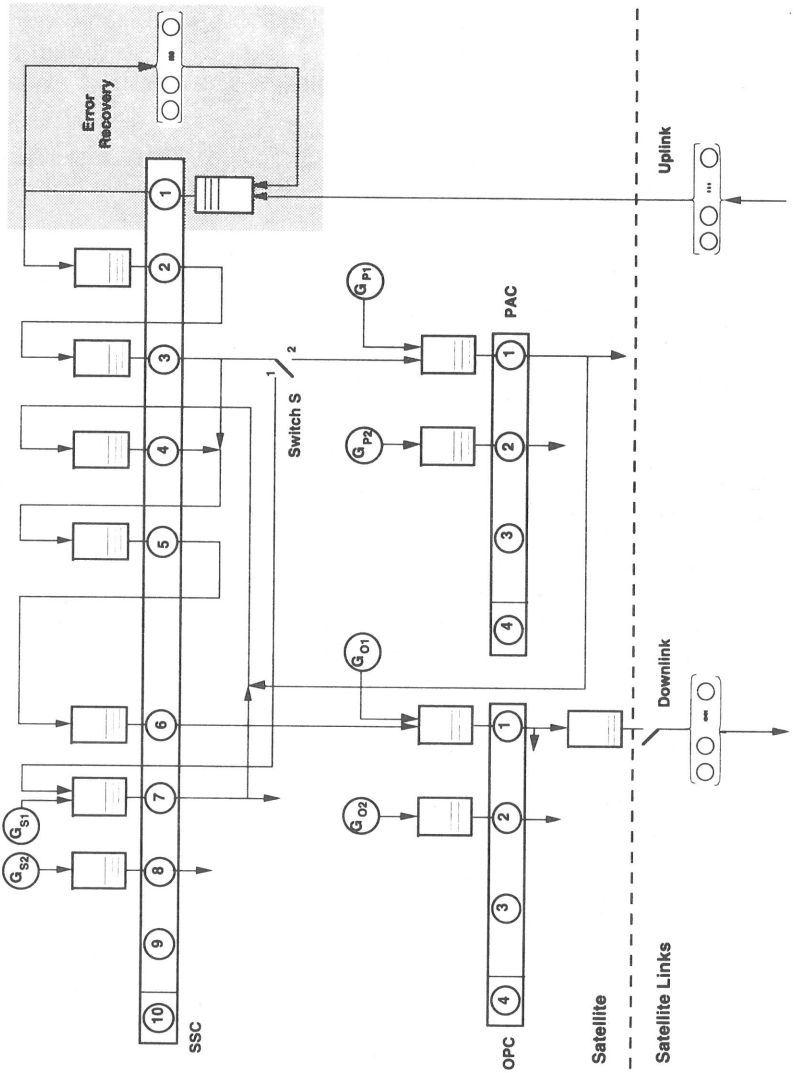



**Bild 5.8:** Erweitertes Simulationsmodell mit Fehlerkorrektur auf der Uplink-Satellitenstrecke

Angenommen, in einer Zusatzstudie sollen die Auswirkungen gestörter Pakete auf der Satellitenstrecke nicht mehr vernachlässigt werden, so müßte das Modell entsprechend angepaßt werden. Da die Bitfehlerwahrscheinlichkeit u.a. vom Wetter und der verwendeten Antennengröße abhängt, könnte man mit Hilfe einer Parameterstudie Erkenntnisse über die optimale Auslegung der Antennen in den Erdeterminals gewinnen. Da die kritische Strecke mit der höchsten Fehlerrate zwischen Terminal und Satellit verläuft, sollen nur die Auswirkungen gestörter Pakete im Uplink berücksichtigt werden. Bild 5.8 zeigt das geänderte Simulationsmodell. Ein gestörtes Paket, das über Selective-Repeat neu angefordert wird, wirkt sich in der Simulation dadurch aus, daß alle Pakete, die zwischenzeitlich eintreffen, gespeichert werden müssen, bis das angeforderte Paket eintrifft. Anschließend kann die Empfangsphase der Schicht 2 alle Pakete in der richtigen Reihenfolge zur Bearbeitung an die Empfangsphase der Schicht 3 übergeben. Da dieser Vorgang von der hohen Signallaufzeit zwischen Erde und Satellit dominiert wird, kann er im Modell durch ein einfaches Laufzeitglied nachgebildet werden, das ein als gestört angenommenes Paket nach der doppelten Signallaufzeit wieder in die Empfangswarteschlange einreiht. Wie man dem Bild entnehmen kann, sind also nur Komponenten innerhalb des Satelliten betroffen.

#### 5.1.5.2 Änderung des Simulationsprogrammes

Um das Simulationsprogramm an die geänderten Anforderungen anzupassen, gibt es mehrere Möglichkeiten. Zum einen können die betroffenen Klassen direkt geändert bzw. erweitert werden, zum anderen können die Anpassungen auch durch Bildung neuer Unterklassen durchgeführt werden. Für kleinere Änderungen ist der erste Schritt durchaus legitim. Bei größeren Änderungen oder falls beide Versionen des Simulationsprogrammes weiterhin benötigt werden, empfiehlt es sich jedoch, neue Klassen aus den bestehenden abzuleiten und diese an deren Stelle zu verwenden. Auf diese Weise kann leicht zur Laufzeit des Programmes entschieden werden, welche Version simuliert werden soll. Es müssen dann nur Objekte der entsprechenden Klassen erzeugt werden. Es soll nochmals hervorgehoben werden, daß sich beide Verfahren nur lokal auswirken. Alle übrigen Programmteile sind von einer Änderung nicht betroffen und müssen daher auch nicht neu getestet werden.

Im genannten Beispiel wird die Empfangsphase der Schicht 2 des SSCs (Phase 1 im Bild) komplett ersetzt. Die neue Phase übernimmt die Aufgabe, das Selective-Repeat-Protokoll abzuarbeiten. Bei jedem empfangenen Paket wird über eine Wahrscheinlichkeitsverteilung ausgewürfelt, ob das Paket korrekt oder gestört empfangen wurde. Gestörte Pakete werden über einen zweiten Port ausgegeben. Außer der Reihe empfangene Pakete werden in einer internen Warteschlange zwischengespeichert. Nach Eintreffen des fehlenden Paketes wird das Paket zusammen mit den zwischengespeicherten Paketen an den normalen Port der Phase ausgegeben.

Von der ursprünglichen Satelliten-Modellkomponente wird eine erweiterte Modellkomponente abgeleitet, die das Laufzeitglied und einen zusätzlichen Multiplexer zum Mischen der Paketströme von Uplink und Laufzeitglied enthält. Der folgende Programmausschnitt zeigt den kompletten Code, der erforderlich ist, um die neue Satelliten-Modellkomponente zu definieren und zu initialisieren.

```
// Class declaration of original satellite entity
class TSatellite : public TEntity {
public:
   // Constructor
   TSatellite(const TString name, TEntity * owner = 0);
   // Add satellite parts that are parsed seperately
   void AddSatelliteParts(TSSC & ssc,
                          TProcessor & opc,
                          TProcessor * pac,
                          TLoadGenerator * sscG1,
                          TLoadGenerator * sscG2,
                          TLoadGenerator * opcG1,
                          TLoadGenerator * opcG2,
                          TLoadGenerator * pacG1,
                          TLoadGenerator * pacG2);

   virtual void ConnectModel(); // Connect internal satellite parts

protected:
   // Components of satellite model
   ...
   // Satellite receive queue for phase 1
   TUnboundedFIFOQueue  fSscPhase1Queue;
};

// New derived satellite class with error recovery
class TECCSatellite : public TSatellite {
public:
   // Constructor
   TECCSatellite(const TString name, TEntity * owner = 0);

   virtual void ConnectModel();      // Connect extended satellite

protected:
   // New satellite entities for error recovery
   TNTo1Multiplexer<2u> fStandardAndECTraffic;// Multiplexer component
   TDInfiniteServer     fRejectDelay;         // Time delay component
};


// Constructor
TECCSatellite::TECCSatellite(const TString name, TEntity * owner) :
   TSatellite(name, owner),           // Construct original satellite
   fStandardAndECTraffic("CombineStdAndECCTraffic", this),
   fRejectDelay(270.0, "RejectDelay", this)      // Double hop = 270ms
{
}
```

```
// Connect extended satellite
void TECCSatellite::ConnectModel()
{
   TSatellite::ConnectModel();  // Connect original satellite first
   // Rename original satellite input, so we can reuse port name
   this->RenamePort("input", "trafficInput");
   // New satellite input port (retains old name)
   this->AliasPort(fStandardAndECTraffic, "input 1", "input");
   // Connect phase -> Infinite Server -> Multiplexer
   fSSC->Connect("SSC_P1_reject", fRejectDelay, "input");
   fRejectDelay.Connect("output", fStandardAndECTraffic, "input 2");
   // Connect multiplexer with input receive queue
   fStandardAndECTraffic.Connect("output", fSscPhase1Queue, "input");
}
```

Der Konstruktor der erweiterten Komponente initialisiert die ursprüngliche Satellitenklasse sowie die zusätzlichen Modellkomponenten. Die Methode *ConnectModel* wird überschrieben, um das erweiterte Teilmodell zu verbinden. Zunächst wird die ursprüngliche Methode aufgerufen, die das bisherige Satellitenmodell verbindet. Der frühere Eingangsport wird umbenannt. *AliasPort* definiert einen Eingangsport des Multiplexers als neue Schnittstelle der Satellitenkomponente. Anschließend werden die neu hinzugekommenen Modellkomponenten verknüpft. Die neue SSC-Phase taucht hier nicht auf, da sie zusammen mit dem SSC und anderen konfigurierbaren Komponenten von den Parserklassen erzeugt und mit *AddSatelliteParts* an die Satellitenklasse übergeben wird. Da sich die Schnittstellen zwischen Satellit und den übrigen Teilmodellen nicht geändert haben, können letztere unverändert bleiben.

Als letzte Anpassung müssen die Parserklassen von SSC und Satellit überschrieben bzw. geändert werden, damit sie die neue SSC-Phase und das neue Satellitenobjekt erzeugen. Dies ist absolut trivial und erfordert jeweils nur wenige Zeilen Code. Durch Auswahl der entsprechenden Parserklassen können nun wahlweise Simulationen mit oder ohne Fehlerbehandlung erfolgen. Weitere Anpassungen sind nicht erforderlich. Auch der Test kann sich auf die neu hinzugekommenen Modellkomponenten beschränken.

#### 5.1.5.3 Erkenntnisse

Wie dieses Beispiel gezeigt hat, kann ein bestehendes Simulationsprogramm auf natürliche Art und Weise erweitert und verfeinert werden. Die Möglichkeiten der objektorientierten Programmierung sowie die Flexibilität der Simulationsbibliothek zahlen sich hierbei aus. Lokale Änderungen wirken sich auch nur lokal aus. Das Ziel einer vollkommen hierarchischen Modellbildung konnte kompromißlos erreicht werden und unterstützt die schrittweise Verfeinerung von Modellkomponenten. Dadurch erhöht sich nicht nur die Übersichtlichkeit der Programmstruktur, sondern es verringert sich vor allem der Aufwand zum Testen von Änderungen. Insgesamt kann die Erweiterung von Programmen als integraler Bestandteil in der Evolution des Entwicklungsprozesses betrachtet werden.

## 5.2 Erweiterungen für parallele Simulation

Dieses Kapitel soll einen Überblick über die Schritte geben, die notwendig sind, um ein vorhandenes sequentielles Simulationsprogramm zu parallelisieren. Anhand dieser Lösungen soll nochmals die Flexibilität der entworfenen Simulationsbibliothek deutlich gemacht werden.

### 5.2.1 Parallelisierung von Teiltests

In Kapitel 3.4.2 wurde bereits das grundsätzliche Verfahren zur parallelen Abarbeitung von Teiltests beschrieben. In diesem Abschnitt soll nun konkreter darauf eingegangen werden, welche Teile der Simulationsbibliothek angepaßt werden müssen.

Das Starten von Prozessen auf fremden Rechnern und der Austausch von Daten zwischen Prozessen hängt stark vom jeweiligen Betriebssystem ab. Wie genau diese Aufgaben zu lösen sind, soll deshalb hier nicht näher untersucht werden. Eine Möglichkeit wäre z.B. der Einsatz von „Remote Procedure Calls“ (RPC), die es erlauben, Funktionen auf anderen Rechnern aufzurufen.

Die Simulationssteuerung wird von der Klasse *TSimulation* oder einer von ihr abgeleiteten Klasse übernommen. In der einfachsten Version der Steuerung werden N identische Kopien des Programmes auf verschiedenen Rechnern gestartet. Damit jede Kopie mit einem anderen Startwert für die Zufallszahlenerzeugung startet, kann dieser als Teil der Initialisierung z.B. über eine Eingabedatei an das Programm übergeben werden. Die eigentliche Simulation läuft dann normal ab, beinhaltet jedoch nur die Warmlaufphase und einen Teiltest. Am Simulationsende werden die Ergebnisse in einer Ergebnisdatei abgelegt. Die Teilergebnisse werden entweder vom Hauptprogramm automatisch oder manuell eingesammelt. Die Auswertung der Ergebnisse kann mit Hilfe eines speziellen Programmes oder einem normalen Tabellenkalkulationsprogramm erfolgen. Die flexible Druckformatsteuerung ermöglicht es, die Ergebnisdateien in einem Format zu erzeugen, das von allen gängigen Tabellenkalkulationen gelesen werden kann. Sofern diese Art der Steuerung verwendet wird, sind keinerlei Änderungen am Simulationsprogramm vorzunehmen. Der Rahmen zur Steuerung des parallelen Ablaufs der Programme kann wahlweise durch ein eigenständiges Steuerprogramm erzeugt werden oder in eine spezielle von *TSimulation* abgeleitete Klasse integriert werden. Mit derselben Methode können auch mehrere komplette Simulationsläufe für Parameterstudien parallelisiert werden (vgl. Kap. 3.4.1).

Das eigentliche Problem bei der Parallelisierung stellt die Auswertung der Ergebnisse dar. Die in der einfachen Form angegebene Lösung, die Ergebnisse nach dem Einsammeln durch spezielle Programme oder eine Tabellenkalkulation auswerten zu lassen, ist problemabhängig, läßt sich nicht ohne weiteres verallgemeinern und erfordert zusätzlichen Aufwand

vom Anwender. Die beste Lösung ist daher, die Auswertung den Komponenten zu überlassen, die auch die Ergebnisse erzeugen. Im Normalfall sind dies Objekte einer der Statistikklassen. Man kann nun von allen Statistikklassen neue Unterklassen bilden, die über Kommunikationsmodule in der Lage sind, die Ergebnisse zwischen gleichartigen Objekten in anderen Prozessen auszutauschen. Damit würden die Ergebnisse am Ende der Simulation allen Statistikobjekten zur Verfügung stehen und könnten in herkömmlicher Weise normal ausgedruckt werden. Bei sequentiellen Simulationen würden die bekannten Standard-Statistikklassen verwendet, bei paralleler Ausführung kämen die erweiterten Statistikklassen zum Einsatz. Bei der Umstellung einer Simulation von sequentieller auf parallele Ausführung müßten dann allerdings alle Statistikobjekte in allen Modellkomponenten angepaßt werden. Um dies zu vermeiden, wurde von vornherein ein Statistikmanager vorgesehen, der die Erzeugung von Statistikobjekten vornimmt. Alle Modellkomponenten, die Statistikfunktionen benötigen, lassen sich ihre Statistikobjekte von dieser Managerklasse erzeugen. Die einzige Änderung, die notwendig ist, um ein Simulationsprogramm umzustellen, besteht darin, dem Statistikmanager mitzuteilen, welche Art von Statistikobjekten er erzeugen soll. Wie man an diesem Beispiel sieht, können Managerklassen in diesem Fall sehr nützlich sein. Die Managerklasse erzeugt abhängig von den momentan gültigen Einstellungen die passenden Objekte mit unterschiedlicher Implementierung, die vom Benutzer nur über die allgemein zugänglichen Schnittstellen der Basisklasse manipuliert werden. Auf diese Weise erfolgt eine Entkopplung von Erzeuger und Benutzer von Objekten, welches die Flexibilität insgesamt steigert.

Sofern man die Warmlaufphase nur einmal durchführen und sie nicht in jedem Prozeß erneut wiederholen möchte, muß der Systemzustand am Ende der Warmlaufphase auf alle Prozesse verteilt werden. Diese führen dann nur jeweils einen Teiltest mit unterschiedlichen Startwerten für die Zufallszahlenerzeugung durch. Um auch diese Version für den Anwender transparent zu gestalten, bietet sich der Einsatz einer objektorientierten Datenbank an [2]. Eine entsprechend modifizierte Steuerungsklasse sorgt dafür, daß am Ende der Warmlaufphase die Zustände aller Modellkomponenten in der Datenbank abgelegt werden. Dann werden die parallelen Prozesse gestartet, die statt der Warmlaufphase den Systemzustand aus der Datenbank einlesen. Die Teiltestsimulation läuft wie gewohnt ab. Die Ergebnisauswertung kann wie oben beschrieben durchgeführt werden.

Insgesamt zeigt sich, daß sich aufgrund der flexiblen Gesamtarchitektur der Simulationsbibliothek die parallele Abarbeitung von Teiltests ohne größere Schwierigkeiten und vor allem für den Anwender völlig transparent durchführen läßt. Dies ermöglicht es dem Anwender, ein Simulationsprogramm erst in sequentieller Form zu erstellen und es später ohne Zusatzaufwand auf mehreren Rechnern parallel auszuführen.

### 5.2.2 Konservative Verfahren

Wie in Kapitel 3.4.5 bereits angesprochen wurde, stellt das Hauptproblem bei verteilten konservativen Verfahren die Synchronisation der Teilmodelle miteinander dar. Aufgrund der hierarchischen Modellbildung, die von der Simulationsbibliothek voll unterstützt wird, können beliebige Teilmodelle gebildet werden. Jedes Teilmodell kann einen eigenen Kalender besitzen und prinzipiell auf einem anderen Rechner ablaufen. Die Möglichkeit, Teilmodelle bereits bei der sequentiellen Form der Simulationsbibliothek zu bilden, erleichtert das Experimentieren bei der Aufteilung des Modells und erlaubt es, die Validierung des Simulationsprogrammes auf einem Rechner durchzuführen. Später werden die Teilmodelle auf verschiedene Rechner verteilt und die direkte Synchronisation wird durch Inter-Prozeßkommunikation ersetzt.

Eine von *TModel* abgeleitete Klasse muß so modifiziert werden, daß sie vor Ausführung eines Ereignisses sicherstellt, daß von außen kein Ereignis mit einem in der Vergangenheit liegenden Zeitstempel eintreffen kann, da sonst das Kausalitätsprinzip verletzt würde. Diese Synchronisation kann auf unterschiedliche Weise erfolgen. In einem ersten Ansatz sollen die Teilmodelle in sich abgeschlossen sein und nicht über normale Ports miteinander kommunizieren. Der Datenaustausch zwischen den Teilmodellen erfolgt über spezielle Modellkomponenten. Das eigentliche Teilmodell ist damit in sich konsistent und läuft sequentiell ab. Synchronisationsprobleme können nur bei den speziellen Modellkomponenten auftreten. In diesem Fall kann das Modell in normale Modellkomponenten und solche, die synchronisiert werden müssen, aufgeteilt werden. Letztere melden sich beim Erzeugen automatisch beim Modell an. Vor der Ausführung eines Ereignisses werden alle angemeldeten Modellkomponenten abgefragt. Erst nachdem alle Komponenten zugestimmt haben, wird das Ereignis ausgeführt. Während der Abfrage kann es erforderlich sein, daß einzelne Modellkomponenten Kontakt zu Modellkomponenten auf anderen Rechnern aufnehmen müssen, um sich gegenseitig zu synchronisieren.

Die Entscheidung, wann synchronisiert werden muß und wie Verklemmungen verhindert werden können, liegt bei dieser Lösung bei den speziellen Modellkomponenten und letztendlich beim Anwender, der diese Modellkomponenten implementieren muß. Die Simulationsbibliothek ist deshalb nur wenig betroffen und kann diesen Ansatz problemlos unterstützen. Modellkomponenten, die synchronisiert werden müssen, werden von der Klasse *TSyncEntity* abgeleitet, die eine Unterklasse von *TEntity* darstellt. Alle Objekte, die zu einer von *TSyncEntity* abgeleiteten Klasse gehören, melden sich automatisch bei der modifizierten Modellklasse an. *TSyncEntity* stellt die Methode *Synchronize* zur Verfügung, die vom Modell vor Ausführung eines Ereignisses aufgerufen wird und die von abgeleiteten Klassen überschrieben werden kann. Die Kapselung der Synchronisation in einer speziellen Klasse *TSyncEntity* erlaubt es, diesen Ansatz generell in einer erweiterten Version der Simulationsbibliothek zu unterstützen, ohne daß sich dies auf die Architektur oder die Abläufe innerhalb

der Bibliothek auswirken würde. Normale Modellkomponenten, die innerhalb der Teilmodelle verwendet werden, müssen nicht modifiziert werden.

Obwohl dieser Ansatz einen Teil der Verantwortung auf den Anwender abschiebt, ist er für eine Reihe von Problemen geeignet. In einer Fallstudie wurde dieses Verfahren zur verteilten Simulation eines Protokolls für die Kommunikation über eine Ringstruktur verwendet. Jede Ringstation wurde dabei auf einem eigenen Rechner simuliert, die über ein Netzwerk miteinander verknüpft waren. Die Simulation konnte in kürzester Zeit entworfen und implementiert werden und funktioniert zur vollsten Zufriedenheit.

Die Simulationssteuerung muß nur angepaßt werden, falls die Ergebnisse von allen Prozessen automatisch gesammelt und aufbereitet werden sollen. Es können die normalen Statistikklassen verwendet werden, da die Simulationsphasen insgesamt sequentiell ablaufen und nur die Modellkomponenten auf verschiedene Rechner verteilt sind. Sofern von den speziellen Modellkomponenten Ereignisse asynchron (z.B. bei Eintreffen einer Nachricht über das Netz) erzeugt werden, muß zusätzlich darauf geachtet werden, daß Zugriffe auf den Kalender über Semaphore geschützt werden, um dessen Konsistenz sicherzustellen.

In einem zweiten Verfahren könnten die Teilmodelle über die normalen Port-Schnittstellen miteinander gekoppelt werden. Das Handshake-Protokoll, das zur Übertragung von Nachrichten zwischen Ports angewandt wird, kann prinzipiell auch über einen Kommunikationskanal ablaufen. Spezielle von *TPort* abgeleitete Klassen könnten die Synchronisation sowie den Datenaustausch zwischen Teilmodellen steuern. Dies hätte den Vorteil, daß ein sequentielles Simulationsprogramm nachträglich durch Austausch der Ports an den Schnittstellen der Teilmodelle und Verwendung der parallelen Version der Modellklasse parallelisiert werden könnte.

Da in beiden Varianten die Synchronisation und die Verklemmungsvermeidung vom Anwender programmiert werden müssen, kommt es auf den Einzelfall an, welche Methode besser geeignet ist. Auf jeden Fall zeigt auch dieses Beispiel, daß sich die Simulationsbibliothek mit geringem Aufwand an unterschiedliche Erfordernisse anpassen läßt.

## 5.3 Umfang und Bewertung der Simulationsbibliothek

Die vorangegangenen Abschnitte haben einige Erfahrungen bei der Anwendung der Simulationsbibliothek wiedergegeben. Außerdem wurde gezeigt, wie die Bibliothek an unterschiedliche Anforderungen angepaßt werden kann. Dieses Kapitel soll einen kurzen Überblick über den Umfang der Simulationsbibliothek geben und anhand von Schaubildern einige interessante Aspekte hervorheben.

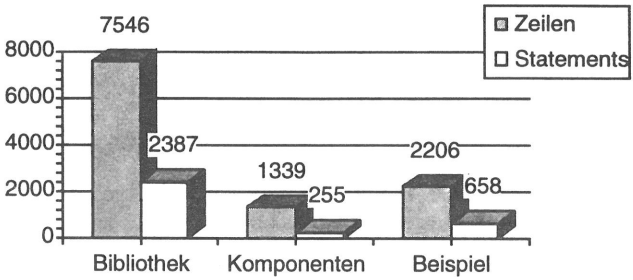



**Bild 5.9:** Anzahl Programmzeilen und Anzahl ausführbarer Statements für Basisbibliothek, Standard-Modellkomponenten und Beispielprogramm

Bild 5.9 zeigt den Umfang der Simulationsbibliothek, aufgeteilt in die Basisversion der Bibliothek und die bereits vorhandenen Standard-Modellkomponenten. Zum Vergleich wurden die Werte für das Beispielprogramm ebenfalls nochmals eingetragen. Wie man dem Bild entnehmen kann, ist der Umfang der Bibliothek im Vergleich zu den Standardkomponenten und dem Beispielprogramm bei weitem nicht so groß wie man aus dem bisher vorgestellten Funktionsumfang schließen würde. Dies kommt zum einen daher, daß die Bibliothek zwar viele Abstraktionen vorgibt, diese jedoch teilweise erst in den Modellkomponenten realisiert werden müssen. Zum anderen werden einige Abstraktionen bereits innerhalb der Simulationsbibliothek wiederverwendet, was zur guten Codeeffizienz beiträgt. Interessant ist auch die Aufteilung des Codes in Compileranweisungen, Deklarationen und echte Codezeilen, die Bild 5.10 zeigt. Hier zeigt sich, daß die Bibliothek prozentual mit Abstand den höchsten Codeanteil besitzt, da sie sehr viel Basisfunktionalität erbringen muß.

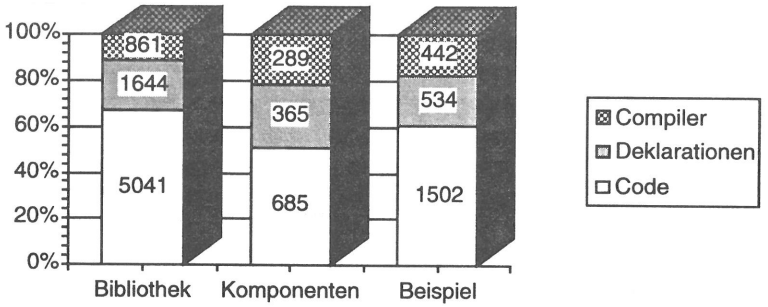



**Bild 5.10:** Aufteilung der Programmzeilen nach Compileranweisungen, Deklarationen und Programmcode prozentual und absolut

Bild 5.11 zeigt einen Überblick über die Anzahl der Klassen. Interessant ist die Aufteilung in Basis- und abgeleitete Klassen. Basisklassen stellen im allgemeinen eigene Abstraktionen dar. Abgeleitete Klassen erweitern bzw. implementieren vorhandene Abstraktionen. Da die Bibliothek gleichzeitig die Abstraktionen vorgibt, besitzt sie die meisten Basisklassen. Die Standard-Modellkomponenten sowie das Beispielprogramm bauen auf diesen Abstraktionen auf. Sie verwenden deshalb fast ausschließlich abgeleitete Klassen. Nur das Beispielprogramm definiert eigene Basisklassen zur Abstraktion der Reihenfolgezähler und des Protokolls. Die Simulationsbibliothek selbst verwendet fast doppelt soviel abgeleitete Klassen wie Basisklassen. Dies ist zum einen darauf zurückzuführen, daß für viele abstrakte Basisklassen Standardimplementierungen vorgenommen wurden, die für normale Anwendungen direkt verwendet werden können. Zum anderen wurden die gemeinsamen Eigenschaften vieler Klassen in Basisklassen verlagert. So besitzen z.B. alle Statistikklassen eine gemeinsame Oberklasse *TStatistic*.

Wie man dem Bild ebenfalls entnehmen kann, ist der Anteil mehrfach vererbter Klassen sehr gering. Dies ist damit zu begründen, daß die neu gebildeten Klassen meist von einer Basisabstraktion dominiert werden, z.B. bei der Definition einer neuen Modellkomponente. Andere Abstraktionen werden zwar innerhalb der Klassen verwendet, jedoch meist in „Benutzen"-Beziehungen oder als Datenobjekt. Dies wird noch dadurch bestätigt, daß in fast allen Fällen, bei denen Mehrfachvererbung verwendet wurde, diese nur zur Unterstützung der Implementierung dient. Klassen die durch Mehrfachvererbung entstanden sind, besitzen deshalb häufig eine öffentliche und eine oder mehrere private Basisklassen. Meistens könnte die Mehrfachvererbung durch eine einfache Vererbung mit entsprechenden Datenobjekten ersetzt werden (vgl. Kap. 2.5.2.3 und Kap. 4.7.2).

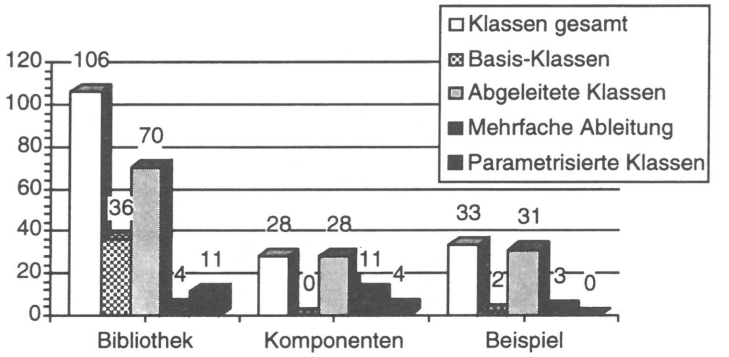



**Bild 5.11:** Vergleich der Anzahl der Klassen und Aufteilung in Basis- und abgeleitete Klassen. Außerdem Vergleich der Anzahl der mehrfach abgeleiteten und parametrisierten Klassen

Parametrisierte Klassen werden verwendet, um eine Abstraktion für unterschiedliche Datentypen nutzbar zu machen. Innerhalb der Simulationsbibliothek werden parametrisierte Klassen hauptsächlich zu sog. Callback-Zwecken angewandt, die Methodenaufrufe in eine andere Klasse umleiten. Als Beispiel seien hier die allgemeinen Nachrichtenfilterklassen genannt, die es erlauben, bei Ankunft einer Nachricht eine fast beliebige Methode der Modellkomponente aufzurufen. Das größte Potential für die Anwendung parametrisierter Klassen steckt in der Definition von Standard-Modellkomponenten. Mit Hilfe parametrisierter Klassen lassen sich dort konfektionierbare Modellkomponenten entwerfen, deren Wiederverwendungspotential deutlich höher ist als bei fest definierten Komponenten. Als Beispiel sei auf eine Multiplexer-Modellkomponente verwiesen, deren Eingangszahl als Parameter festgelegt werden kann. Nachdem die Komponente einmal allgemein entwickelt wurde, kann sie für jede beliebige Anzahl von Eingangsports wiederverwendet werden (vgl. die Deklaration der Multiplexer-Modellkomponente im Codeausschnitt in Kap. 5.1.5.2, die einen Multiplexer mit 2 Eingängen und einem Ausgang definiert). Da das Beispielprogramm auf ein spezielles Problem zugeschnitten ist, wurden hier nur konkrete und keine neuen parametrisierten Klassen benötigt. Allerdings wurden Instanzen vorhandener parametrisierter Klassen bei der Implementierung benutzt.

Es sei noch darauf hingewiesen, daß die Implementierung der Simulationsbibliothek eine käufliche Standardbibliothek für alle internen Datentypen, wie Warteschlangen, Listen, etc. einsetzt. Diese Bibliothek realisiert alle Datentypen ebenfalls mit Hilfe parametrisierter Klassen [6, 7]. Die Standardbibliothek wurde in den Bildern und Aussagen dieses Kapitels nicht berücksichtigt.

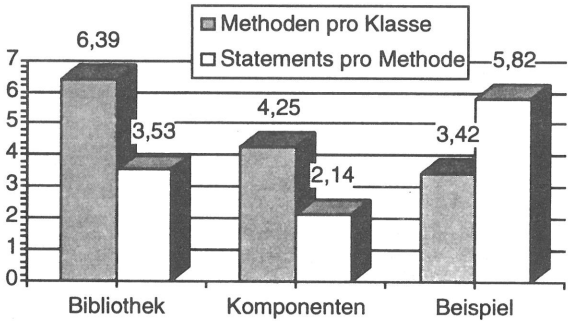



**Bild 5.12:** Vergleich von Anzahl Methoden pro Klasse und Anzahl der Statements pro Methode

Das letzte Bild dieses Kapitels zeigt die Anzahl von Methoden pro Klasse sowie die Anzahl ausführbarer Statements pro Methode. Die Simulationsbibliothek definiert die meisten Methoden in jeder Klasse. Wie bereits in Kapitel 5.1.4.1 ausgeführt, ist dies eine Folge des Versuches, jeder Methode nur eine Teilaufgabe zuzuordnen. Würde man nur Basisklassen in Betracht ziehen, so wäre der Wert noch höher. Dies wurde gemacht, um in abgeleiteten Klassen sehr selektiv und damit effizient einzelne Methoden überschreiben zu können. Trotzdem ist die Anzahl der Methoden pro Klasse insgesamt nicht sehr hoch, was die Übersichtlichkeit der einzelnen Abstraktionen fördert.

An der geringen Zahl von Methoden pro Klasse sowohl bei den Standard-Modellkomponenten als auch im Beispielprogramm, kann man ablesen, daß die Basisabstraktionen bereits sehr viel Funktionalität beinhalten. Zieht man je einen Konstruktor und Destruktor pro Klasse in Betracht, bedeutet dies, daß die meisten Modellkomponenten durch Überschreiben von nur 1 - 2 Methoden angepaßt werden können. Dies zeigt, daß die vorhandenen Abstraktionen zu einem hohen Grad wiederverwendet werden können.

Obwohl die Basisbibliothek die meisten Abstraktionen definiert, überrascht sie mit einem geringen Codeumfang pro Methode. Dies hängt zum einen mit der großen Anzahl von Methoden pro Klasse zusammen, zum anderen ist es eine Folge der Verwendung zumindest teilweise abstrakter Basisklassen. Es gibt deshalb viele Methoden, die keine Statements enthalten, da sie nur eine Schnittstelle definieren und es keine sinnvolle Standardimplementierung gibt. Diese müssen deshalb in abgeleiteten Klassen überschrieben werden. Auf der anderen Seite wird dies durch einige sehr umfangreiche Methoden im Bereich der Ein-/Ausgabe kompensiert. Insgesamt ergibt sich dadurch zwar ein geringer Mittelwert, der jedoch stark variiert.

Der geringe Codeumfang pro Methode bei den Standardkomponenten ist darauf zurückzuführen, daß diese meist einen Großteil der Funktionalität der Basisklasse übernehmen können und die Erweiterungen keine Sonderfälle berücksichtigen müssen, sondern für alle ausgetauschten Nachrichten gleichermaßen gelten. Im Gegensatz dazu benötigt das problemnahe Beispielprogramm mehr Code pro Methode, da hier nachrichtenabhängig unterschiedliche Vorgänge ausgeführt werden.

Ein Vergleich der Werte für das Beispielprogramm in den Abbildungen 5.7 und 5.10 bestätigt dies. In Bild 5.7 wurde die Anzahl Statements pro Methode für das gesamte Beispielprogramm angegeben. Dagegen enthält Bild 5.10 nur den problemspezifischen Teil des Programmes, da die allgemein wiederverwendbaren Modellkomponenten den Standardkomponenten zugerechnet wurden. Dadurch steigt der Anteil des problemspezifischen Codes im Beispielprogramm an, was zur Erhöhung des Mittelwertes von 5,2 auf 5,82 Statements pro Methode führt.

Abschließend sollen die wichtigsten Punkte nochmals zusammengefaßt werden:

- Die Simulationsbibliothek definiert wenige grundlegende Abstraktionen, meist in Form abstrakter Basisklassen, die in abgeleiteten Klassen implementiert bzw. erweitert werden.
- Jede Klasse definiert modular für alle Teilaufgaben eigene Methoden, um ein effizientes Überschreiben in abgeleiteten Klassen zu ermöglichen. Die Anzahl der Methoden ist trotzdem gering, um die Übersichtlichkeit der Abstraktion zu wahren. Komplexere Abstraktionen werden deshalb auf mehreren Klassen verteilt.
- Die einzelnen Methoden erfüllen nur kleine Teilaufgaben. Sie beinhalten in der Regel nur wenige Zeilen Code. Sie sind deshalb leicht zu kodieren, zu verstehen und einfach zu testen.
- Die beiden letzten Punkte unterstreichen nochmals, daß der Entwurfsaufwand zur Bildung geeigneter Abstraktionen bei objektorientierter Softwareentwicklung höher einzuschätzen ist, als bei konventioneller Vorgehensweise. Dafür vereinfachen sich Kodierungs- und Testphase entsprechend.
- Standard- und problemspezifische Modellkomponenten können aus den vorhandenen Abstraktionen mit geringem Aufwand realisiert werden, was für eine ausgereifte Wahl der Basisabstraktionen spricht.

# Kapitel 6

# Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde der Entwurf und die Architektur einer objektorientierten Simulationsbibliothek beschrieben. Ausgehend von den Grundlagen der objektorientierten Programmierung und der ereignisorientierten Simulation wurden Anforderungen an eine möglichst universelle Bibliothek zur Simulation komplexer Systeme formuliert. Anschließend wurden diese mit Hilfe objektorientierter Softwareentwurfsmethoden umgesetzt.

Es hat sich gezeigt, daß sich die Vorteile objektorientierter Softwaretechnologie für Simulationsanwendungen geradezu ideal nutzen lassen. Als Hauptvorteil muß dabei die konzeptionelle Nähe zum Problembereich gesehen werden, die es erlaubt, gefundene Simulationsmodelle direkt in Modellkomponenten der Simulationsbibliothek abzubilden. Dies erleichtert nicht nur das Verständnis, sondern beschleunigt auch den Programmentwurf. Ein wichtiger Gesichtspunkt ist dabei eine durchgängige Softwarearchitektur mit einheitlichen Regeln, die wenige Grundabstraktionen verwendet. Die wichtigsten Abstraktionen der Simulationsbibliothek sind:

- Hierarchische Modellbildung
- Port-Konzept
- Hierarchische Ereignisverarbeitung

Die hierarchische Modellbildung reduziert die Komplexität eines Simulationsmodells erheblich. Sie ist erforderlich, um hierarchisch gegliederte Systeme direkt in Simulationsmodelle abbilden zu können. Außerdem erlaubt sie die schrittweise Verfeinerung von Teilmodellen und Modellkomponenten. Das Port-Konzept fördert die Bildung abgeschlossener Modellkomponenten mit definierten Schnittstellen und erleichtert so deren unabhängige Wiederverwendbarkeit. Das Handshake-Protokoll ist so ausgelegt, daß es bei einer Erweiterung für parallele Simulationen auch über ein Netzwerk abgewickelt werden kann. Die hierarchische Ereignisverarbeitung unterstützt ebenfalls die Bildung hierarchischer Modelle, indem sie die Bearbeitung von Ereignissen auf allen Hierarchieebenen zuläßt. Sie erlaubt außerdem die Verwendung verteilter Kalender und ermöglicht somit den Übergang zu parallelen Verfahren.

Die Konzentration auf Schlüsselabstraktionen aus dem Problembereich anstatt auf Implementierungskonzepte hat sich bewährt. Die gewählten Abstraktionen sind sehr stabil und können auch bei Erweiterungen, die eine andere Implementierung erfordern, weiterverwendet werden. Als Beispiel seien die in Kapitel 5 vorgestellten Erweiterungen für parallele Simulationen genannt, die mit Hilfe der vorhandenen Abstraktionen realisiert werden können. Außerdem wird dem Anwender dadurch die Transformation des Problembereiches in ein Simulationsprogramm auf allen Ebenen erleichtert. Sofern für einen bestimmten Problembereich bereits fertige Standard-Modellkomponenten vorhanden sind, muß der Anwender sie nur noch zusammensetzen, um ein fertiges Simulationsprogramm zu erhalten. Er kann dabei auf einem sehr hohen Abstraktionsniveau arbeiten, ohne sich um die Implementierung kümmern zu müssen. Der Anwender, der eigene Modellkomponenten erstellt, arbeitet zwar auf einem niedrigeren Abstraktionsniveau, da er sich um viele Implementierungsdetails kümmern muß, er kann aber die gleichen Schlüsselabstraktionen weiterverwenden und implementiert diese nur. Die Verwendung einer Standardprogrammiersprache erleichtert die Umsetzung und hat sich ebenfalls bewährt. Insgesamt ergibt sich ein durchgängiges Konzept ohne Kluft zwischen verschiedenen Abstraktionsebenen, was die Bedeutung einer einheitlichen Softwarearchitektur hervorhebt.

Die iterative Vorgehensweise bei der Erstellung objektorientierter Software sorgt dafür, daß Teilmodelle bereits in einem sehr frühen Entwicklungsstadium simuliert werden können. Dies vereinfacht nicht nur den Test, sondern auch die Validierung des Simulationsmodells. Insgesamt beschleunigt diese Vorgehensweise den gesamten Entwicklungsprozeß und vereinheitlicht ihn, da Wartung und Erweiterungen als weitere Entwurfszyklen aufgefaßt werden können.

Wie der Vergleich anhand eines konkreten Beispiels in Kapitel 5 deutlich macht, ist der objektorientierte Ansatz einer konventionellen Lösung in vielerlei Hinsicht überlegen. Das resultierende Programm ist nicht nur besser strukturiert und damit einfacher zu verstehen, sondern kann aufgrund des hohen Wiederverwertungspotentials schneller und mit weniger Aufwand entwickelt werden. Ein Vergleich des jeweils neu zu entwickelnden Codeumfangs für die Beispielprogramme beweist dies drastisch. Die sinnvolle Verwendung von Vererbung und Polymorphie unterstützt die Erstellung generischer Software und trägt entscheidend zum geringen Umfang der Simulationsbibliothek und der damit erstellten Simulationsprogramme bei.

Die Vorteile des objektorientierten Ansatzes können jedoch nur bei einem durchdachten Entwurf realisiert werden. Hier müssen besonders die folgenden Punkte hervorgehoben werden:

- Abstrakte Basisklassen trennen Schnittstellen von der jeweiligen Implementierung. Ihre konsequente Anwendung hat einen großen Beitrag zur Flexibilität und Erweiterbarkeit der Simulationsbibliothek geleistet.

- In sich abgeschlossene Abstraktionen sind leicht zu verstehen und einfach wiederverwendbar. Jede Abstraktion sollte nur eine definierte Aufgabe übernehmen.
- Die Methoden einer Klasse sollten genau eine Teilaufgabe der Abstraktion implementieren. Dies erleichtert das gezielte Überschreiben von Methoden in abgeleiteten Klassen und verbessert die Codeeffizienz.

Insgesamt muß die Bedeutung der Entwurfsphase unterstrichen werden. Sofern der Entwurf modular erfolgt und die obigen Grundsätze berücksichtigt, ist der eigentliche Kodier- und Testaufwand gering und leicht überschaubar. Der geringe Codeumfang pro Methode, der meist nur wenige Zeilen beträgt, bestätigt dies eindrucksvoll.

Zusammenfassend kann gesagt werden, daß sich der Einsatz objektorientierter Methoden voll bewährt hat. Die entstandene Simulationsbibliothek ist sehr flexibel und kann leicht an neue Erfordernisse angepaßt werden.

Wenn man von Detail- und Leistungsverbesserungen, die sich im Laufe des Einsatzes der Simulationsbibliothek ergeben, absieht, gibt es zwei interessante Richtungen, die in Zukunft genauer untersucht werden sollten.

Der erste Bereich wäre die konsequente Weiterentwicklung der Simulationsbibliothek für den Einsatz bei parallelen Simulationen. Erste vielversprechende Ansätze wurden bereits in Kapitel 5 vorgestellt und sollten weiterverfolgt werden. Insbesondere sollte untersucht werden, ob optimistische Verfahren in die Simulationsbibliothek integriert werden können. Der Einsatz objektorientierter Datenbanken für die Speicherung der für einen Rollback notwendigen Informationen, könnte eine für den Anwender transparente Lösung ermöglichen.

Der zweite lohnende Bereich für zukünftige Erweiterungen liegt auf dem Gebiet der graphischen Ein-/Ausgabe. Zum einen wäre es attraktiv, die jetzigen Konzepte für graphische Ein-/Ausgabe zu erweitern. Zum anderen wäre es noch interessanter, zu untersuchen, ob sich die Simulationsbibliothek zu einer integrierten Simulationsumgebung erweitern läßt. Die Modellbildung könnte mit Hilfe graphischer Editoren am Bildschirm entworfen werden. Der Code zur Vernetzung von Modellkomponenten würde automatisch generiert werden. Für unbekannte Modellkomponenten würde der Rahmen mit allen Verbindungsports automatisch erstellt und müßte vom Anwender nur noch um die eigentliche Funktionalität ergänzt werden. Damit würde auch die letzte Lücke zu spezialisierten Simulationsumgebungen mit benutzerfreundlichen Bedienoberflächen geschlossen.

# Literaturverzeichnis

[1] J. A. Abell, R. P. Judd: *Reusable Simulation Object Specification through Arbitrary Message Passing and Aggregation Scheme.* Proceedings of the 1993 SCS Western Simulation Multiconference, S. 9 - 14. La Jolla, CA, 1993.

[2] D. Bartels, J. Robie: *Persistent objects and object-oriented databases for C++.* The C++ Report, Vol. 4, Nr. 7, S. 49 - 56. SIGS Publication, New York, NY, September 1992.

[3] G. Barth, C. Welsch: *Objektorientierte Programmierung.* Informationstechnik 30, Nr. 6, S. 404 - 419. R. Oldenbourg Verlag, 1988.

[4] R. F. Belanger: *MODSIM II: A Modular, Object-Oriented Language.* Proceedings of the 1990 Winter Simulation Conference, S. 118 - 122. New Orleans, 1990.

[5] E. H. Bensley, V. T. Giddings, J. I. Leivent, R. J. Watro: *A Performance Comparison of Object-Oriented Simulation Tools.* Proceedings of the 1992 Object-Oriented Simulation Conference, S. 47 - 51. Newport Beach, CA, 1992.

[6] G. Booch, M. Vilot: *The Design of the C++ Booch Components.* Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), S. 1 - 11. Ottawa, Canada, 1990.

[7] G. Booch, M. Vilot: *Simplifying the Booch Components.* The C++ Report, Vol. 5, Nr. 5, S. 41 - 53. SIGS Publication, New York, NY, Juni 1993.

[8] G. Booch: *Object Oriented Design with Applications.* Benjamin Cummings, Redwood City, CA, 1991.

[9] G. Booch: *The Booch Method: Notation.* Rational, Santa Clara, CA, 1992.

[10] B. M. Brosgol: *Object-oriented programming in Ada 9X.* Object Magazine, Vol. 6, Nr. 2, S. 64 -65. SIGS Publications, New York, NY, März / April 1993.

[11] D. Cameron, P. Faust, D. Lenkov, M. Mehta: *A Portable Implementation of C++ Exception Handling*. Proceedings of the USENIX C++ Technical Conference, S. 225 - 243. Portland, OR, August 1992.

[12] T. Cargill: *C++ Programming Style*. Addison-Wesley, Reading, MA, 1992.

[13] CCITT: Recommendations Q.700 - Q.795: *Specification of Signalling System No. 7.* Blue Book, Vol. VI.7 - VI.9. Genf, 1989.

[14] K. M. Chandy, J. Misra: *Distributed simulation: A case study in design and verification of distributed programs.* IEEE Transactions on Software Engineering, SE-5, Vol. 5, S. 440 - 452. September 1979.

[15] K. M. Chandy, J. Misra: *Asynchronous distributed simulation via a sequence of parallel computations.* Communications of the ACM, Vol. 24, Nr. 11, S. 198 - 205. ACM, New York, NY, November 1981.

[16] J. O. Coplien: *Advanced C++ - Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

[17] J. Cribbs, S. Moon, C. Roe: *An Evaluation of Object-Oriented Analysis and Design Methodologies.* Alcatel Network Systems, SIGS Books, New York, NY, 1992.

[18] T. DeMarco: *Structured Analysis and System Specification.* Yourdon Inc., New York, NY, 1978.

[19] P. L'Ecuyer: *Efficient and Portable Combined Random Number Generators.* Communications of the ACM, Vol. 31, Nr. 6, S. 742 - 749, 774. ACM, New York, NY, Juni 1988. Ergänzt in Communications of the ACM, Vol. 32, Nr. 8, S. 1019 - 1024. ACM, New York, NY, August 1989.

[20] P. L'Ecuyer: *Random Numbers for Simulation.* Communications of the ACM, Vol. 33, Nr. 10, S. 85 - 97. ACM, New York, NY, Oktober 1990.

[21] M. A. Ellis, B. Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[22] M. A. Ellis, M. Carroll: *Inheritance-based vs. template-based containers.* The C++ Report, Vol. 5, Nr. 3, S. 17 - 20. SIGS Publication, New York, NY, März / April 1993.

[23] R. G. Fichman, C. F. Kemerer: *Object-Oriented and Conventional Analysis and Design Methodologies. Comparison and Critique.* IEEE Computer, Vol. 25, Nr. 10, S. 22 - 39. Oktober 1992.

[24] R. M. Fujimoto: *Time Warp on a shared memory multiprocessor.* Transactions of SCS, 6(3). S. 211 - 239. Juli 1989.

[25] R. M. Fujimoto: *Performance of Time Warp under synthetic workloads.* Proceedings of the SCS Multiconference on Distributed Simulation, Vol. 22, Nr. 1, S. 23 - 28. Januar 1990.

[26] R. M. Fujimoto: *Parallel Discrete Event Simulation.* Communications of the ACM, Vol. 33, Nr. 10, S. 30 - 53. ACM, New York, NY, Oktober 1990.

[27] P. Gburzynski, P. Rudnicki: *Object-Oriented Simulation in SMURPH – A Case Study of DQDB Protocol.* Proceedings of the SCS Multiconference on Object-Oriented Simulation, S. 12 - 21. Anaheim, CA, Januar 1991.

[28] A. Goldberg, D. Robson: *Smalltalk-80: The Language.* Addison-Wesley, Reading, MA, 1989.

[29] D. Goldsmith, J. Palevich: *Unofficial C++ Style Guide.* Develop, Vol. 1, Nr. 2, S. 204 - 232. Apple Computer, Inc., Cupertino, CA, April 1990.

[30] C. Herring: *ModSim: A new Object-Oriented Simulation Laguage.* Proceedings of the SCS Multiconference on Object-Oriented Simulation. San Diego, CA, 1990.

[31] D. H. H. Ingalls: *Design Principles Behind Smalltalk.* Byte, Vol. 8, Nr. 6, S. 286 - 298. Juni 1981.

[32] D. R. Jefferson, H. Sowizral: *Fast concurrent simulation using the Time Warp mechanism, part I: Local control.* Tech. Report N-1906-AF. RAND Corporation, Dezember 1982.

[33] D. R. Jefferson: *Virtual Time.* Transactions on Programming Languages and Systems, Vol. 3, Nr. 7, S. 404 - 425. ACM, New York, NY, Juli 1985.

[34] W. Kreutzer: *System Simulation – Programming Styles and Languages.* Addison-Wesley, Reading, MA, 1986.

[35] J. A. Lewis, S. M. Henry, D. G. Kafura, R. S. Schulman: *An Empirical Study of the Object-Oriented Paradigm and Software Reuse.* Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), S. 184 - 196. ACM, SIGPLAN, New York, NY, 1991.

[36] B. Liebler: *Entwurf eines Trace-Konzeptes für die objektorientierte Simulationsbibliothek.* Diplomarbeit Nr. 1178, gefertigt am Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1992.

[37] S. B. Lippman: *C++ Primer.* 2. Auflage. Addison-Wesley, Reading, MA, 1991.

[38] V. W. Mak: *DOSE: A Modular and Reusable Object-Oriented Simulation Environment.* Proceedings of the SCS Multiconference on Object-Oriented Simulation, S. 3 - 11. Anaheim, CA, Januar 1991.

[39] B. Malloy, M. J. Harrold, J. D. McGregor: *The Implementation of a Simulation Language using Dynamic Binding.* Proceedings of the 1993 SCS Western Simulation Multiconference, S. 3 - 8. La Jolla, CA, 1993.

[40] J. Martin: *Information Engineering.* Band I - III. Englewood Cliffs, New York, NY, 1990.

[41] R. Martin: *Abstract classes and pure virtual functions.* The C++ Report, Vol. 4, Nr. 6, S. 46 - 52. SIGS Publication, New York, NY, Juli / August 1992.

[42] B. Melamed, R. J. T. Morris: *Visual Simulation: The Performance Analysis Workstation.* IEEE Computer, Vol. 18, Nr. 8, S. 87 - 94. August 1985.

[43] B. Melamed: *Q+: The AT&T Performance Analysis Workstation. Users Guide and Reference Manual.* AT&T Bell Labs, 1989.

[44] B. Meyer: *Applying "Design by Contract".* IEEE Computer, Vol. 25, Nr. 10, S. 40 - 51. Oktober 1992.

[45] B. Meyer: *Eiffel: The Language.* Prentice Hall, Englewood Cliffs, New York, NY, 1991.

[46] B. Meyer: *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, New York, NY, 1988.

[47] S. Meyers: *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, Reading, MA, 1992.

[48] B. Myhrhaug, K. Nygaard, O-J. Dahl: *Common Base Language.* Norwegian Computing Center, 1968.

[49] K. Nygaard, O-J. Dahl: *The development of the SIMULA language.* History of Programming Languages. Academic Press, New York, NY, 1981.

[50] B. Page: *Diskrete Simulation. Eine Einführung mit Modula-2.* Springer-Verlag, Berlin, 1991.

[51] S. K. Park, K. W. Miller: *Random Number Generators: Good Ones Are Hard To Find.* Communications of the ACM, Vol. 31, Nr. 10, S. 1192 - 1201. ACM, New York, NY, Oktober 1988.

[52] K. Pawlikowski: *Steady-State Simulation of Queuing Processes: A Survey of Problems and Solutions.* ACM Computing Surveys, Vol. 22, Nr. 2, S. 123 - 170. ACM, New York, NY, Juni 1990.

[53] M. Piontek, W. Berner, H. Kocher, M. N. Huber: *Frame organization and signalling for an autonomous switching satellite.* Proceedings of the First European Conference on Satellite Communication. München, 1989.

[54] O. Rose: *SIM PlusPlus – Part I - SIM_PP User Manual.* Institut für Telematik, Universität Karlsruhe, 1992.

[55] O. Rose: *SIM PlusPlus – Part II - SimEnv User Manual.* Institut für Telematik, Universität Karlsruhe, 1992.

[56] L. Sachs: *Angewandte Statistik: Anwendung statistischer Methoden.* 7. Auflage. Springer Verlag, Berlin, 1992.

[57] B. Stroustrup: *Object-Oriented Programming.* AT&T C++ Language System, Release 2.0, Selected Readings. AT&T, 1989.

[58] B. Stroustrup: *The C++ Programming Language.* 1. Auflage. Addison-Wesley, Reading, MA, 1986.

[59] B. Stroustrup: *The C++ Programming Language.* 2. Auflage. Addison-Wesley, Reading, MA, 1991.

[60] B. Stroustrup, D. Lenkov: *Run-Time Type Identification for C++ (Revised).* Proceedings of the USENIX C++ Technical Conference, S. 313 - 339. Portland, OR, August 1992.

[61] S. T. Taft: *Ada 9X: A Technical Summary,* Communications of the ACM, Vol. 35, Nr. 11, S. 77 - 82. ACM, New York, NY, November 1992.

[62] H. L. Truong: *Über die Leistungsfähigkeit von HDLC-gesteuerten Datenverbindungen.* 33. Bericht über verkehrstheoretische Arbeiten (Dissertationsschrift), Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1982.

[63] P. W. Vaughan, D. E. Newton, R. P. Johns: *PRISM: An Object-Oriented System Modeling Environment in C++.* Proceedings of the SCS Multiconference on Object-Oriented Simulation, S. 32- 39. Anaheim, CA, Januar 1991.

[64] P. W. Vaughan, D. E. Newton: *Interactive Graphic MODeling Environment (Igmod).* Proceedings of the 1992 Object-Oriented Simulation Conference. S. 25 - 30. Newport Beach, CA, 1992.

[65] T. Wagner: *Simulative Untersuchung der Steuerung eines Vermittlungssatelliten.* 2. Semesterarbeit Nr. 1039, gefertigt am Institut für Nachrichtenvermittlung und Datenverarbeitung, Universität Stuttgart, 1990.

[66] E. Yourdon, L. Constantine: *Structured Design: Fundamentals of a Discipline of Computer Programming and Design.* 2. Auflage. Prentice Hall, New York, NY, 1979.

[67] Q. Zheng, P. Chow: *EXsim: A General Purpose Object-Oriented Environment for Descrete-Event Simulations.* Proceedings of the 1993 SCS Western Simulation Multiconference, S. 15 - 21. La Jolla, CA, 1993.

# Anhang

## A.1 Booch-Notation für OOD-Diagramme.

Zur Verdeutlichung der Konzepte werden im Rahmen dieser Arbeit Klassen- und Objektdiagramme eingesetzt. Die verwendete Darstellung orientiert sich dabei an der Notation von Grady Booch, die in [8] veröffentlicht ist. Eine neuere Version davon ist in [9] zu finden. Um das Verständnis der Diagramme zu erleichtern, sollen im folgenden die wesentlichen Elemente dieser Notation vorgestellt werden.

### A.1.1 Klassendiagramme

Ein Klassendiagramm beschreibt Beziehungen zwischen einzelnen Klassen. Eine Klasse wird durch eine gestrichelte „Wolke" dargestellt, die den Klassennamen enthält, vgl. Bild A.1. Zahlen innerhalb der Klasse geben an, wieviele Instanzen dieser Klasse vorhanden sein können. Wichtig sind hier besonders die Werte Null und Eins. Abstrakte Klassen werden durch Null gekennzeichnet, da sie keine Instanzen besitzen können. Klassen, die genau eine globale Instanz besitzen (z.B. Managerklassen), erhalten die Zahl Eins. Im Bild nicht gezeigt sind parametrisierte Klassen. Sie werden durch ein gestricheltes Rechteck im rechten oberen Eck markiert (vgl. z.B. Bild 4.9 in Kapitel 4.5.4).

Die wichtigsten Beziehungen zwischen Klassen sind Vererbung, Benutzen anderer Klassen und Enthalten von Instanzen anderer Klassen. Vererbung wird durch einen Pfeil kenntlich gemacht, der von der abgeleiteten Klasse zur Basisklasse gerichtet ist. Bei privater Vererbung wird der Pfeil durchgestrichen. Wenn eine Klasse eine andere benutzt, um von ihr beispielsweise Dienste erbringen zu lassen, so wird dies durch eine Doppellinie mit nicht ausgefülltem Punkt dargestellt. Im Bild benutzt Klasse A die Klasse B. Die Enthalten-Beziehung wird durch einen ausgefüllten Punkt unterschieden. Im Beispiel sind Instanzen der Klasse D in Klasse C enthalten. Wenn die Anzahl von Objekten, die an einer Beziehung beteiligt sind, bekannt ist, so kann sie durch Angabe einer Zahl am Ende der Beziehungslinie deutlich gemacht werden. Wenn z.B. ein Managerobjekt A viele Instanzen der Klasse B manipuliert,

so kann man dies durch zwei Zahlen angeben: In der Nähe von Klasse A wird eine 1 angegeben, da alle Instanzen von Klasse B zu einem gemeinsamen Managerobjekt gehören. Am anderen Ende der Beziehung wird mit einem kleinen „n" gekennzeichnet, daß die Klasse A mehrere Instanzen der Klasse B benutzt.

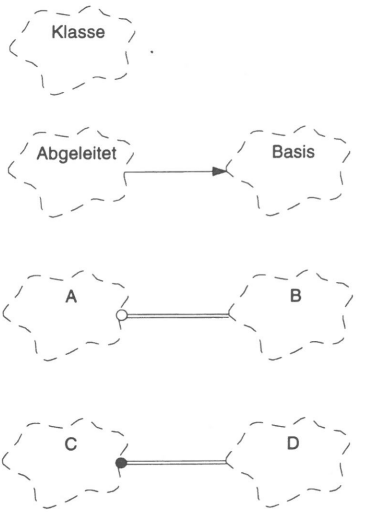



**Bild A.1:** Klassendiagramme

## A.1.2 Objektdiagramme

Objektdiagramme drücken zum einen die Struktur der Objekte zur Laufzeit des Programmes aus, zum anderen können sie zur Illustration des dynamischen Ablaufs herangezogen werden. Objekte werden im Diagramm als „Wolke" mit durchgezogenen Linien dargestellt, vgl. Bild A.2.

Beziehungen zwischen Objekten werden durch Linien gekennzeichnet. Objekte können Methoden anderer Objekte aufrufen. Im Diagramm wird dies durch einen Pfeil mit dem Methodennamen dargestellt, wobei zur Kennzeichnung dynamischer Abläufe die Nachrichten durchnumeriert werden. Im Beispiel ruft Objekt A die Methode mit dem Namen *Methode* von Objekt B auf.

Die Art der Beziehung kann zusätzlich durch Angabe eines Buchstabens in einem Kasten näher qualifiziert werden. Der Buchstabe „F" (für Feld) auf schwarzem Grund im Bild bedeutet, daß das Objekt D in Objekt C enthalten ist (entweder als Instanz oder durch einen Zeiger auf das Objekt). Der ausgefüllte Kasten gibt an, daß D exklusiv zu C gehört. Im Gegensatz dazu würde ein nichtgefüllter Kasten nur eine Referenz auf das Objekt bedeuten. Ein Objekt kann immer nur **einem** anderen Objekt gehören, aber es kann gleichzeitig von **mehreren** Objekten referenziert werden. Wenn ein Objekt nur als Parameter an eine Methode übergeben wird, erfolgt die Kennzeichnung durch den Buchstaben „P". Es gibt noch andere Qualifizierungen, die jedoch im Rahmen dieser Arbeit nicht verwendet wurden und deshalb hier nicht erläutert werden.

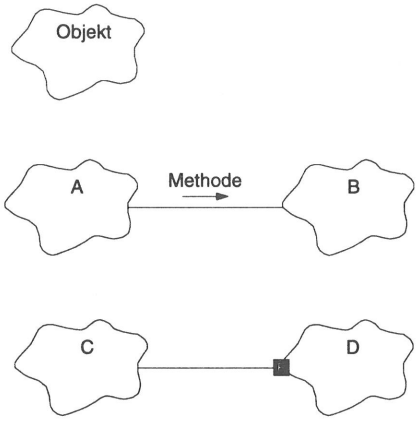



**Bild A.2:** Objektdiagramme

## A.2 Programmierrichtlinien

Dieser Anhang stellt einige Konventionen vor, die beim Entwurf der Simulationsbibliothek beachtet wurden. Richtlinien für die Erstellung von Software können einen weiten Bereich umfassen, von einfachen Regeln zur Namensgebung von Variablen, bis hin zu Vorschriften, wie tief Prozeduren geschachtelt werden dürfen. Die Richtlinien sollen einen einheitlichen Programmierstil sicherstellen und damit dem potentiellen Anwender das Verständnis erleichtern. Die Dokumentation vereinfacht sich ebenfalls, da viele Regeln bereits aus den Programmierrichtlinien ersichtlich sind. Aus diesem Grund wurde bei Entwurf und Implementierung der Simulationsbibliothek auf die Einhaltung der verwendeten Regeln geachtet. Im folgenden werden einige ausgewählte Konventionen vorgestellt, die dem Zweck dienen, die innerhalb dieser Arbeit verwendeten Beispiele transparenter zu machen und deren Verständnis zu erleichtern. Sie basieren teilweise auf eigenen Erfahrungen sowie auf einer Veröffentlichung der Firma Apple, die sich mit diesem Thema beschäftigt [29].

### A.2.1 Erzeugung und Vernichtung von Objekten

Die Simulationsbibliothek hält sich an die Regel, daß derjenige, der ein Objekt erzeugt, dieses auch wieder vernichten muß. Das Einhalten dieser Konvention ist sehr wichtig, da es sonst zu Speicherlecks oder sogar zu Systemabstürzen kommen kann, wenn ein Objekt gar nicht oder mehrmals vernichtet wird.

Alle Teile der Simulationsbibliothek richten sich deshalb nach dieser Konvention; so besitzt z.B. der Statistikmanager Methoden sowohl zum Erzeugen als auch zur Vernichtung von Statistikobjekten. Das Einhalten dieser Regel erfordert einige Umsicht, besonders wenn mehrere Objekte eng zusammenarbeiten. So muß z.B. ein Port darauf achten, daß beim Löschen des Objektes ein evtl. eingetragener Message-Handler nicht mit vernichtet, sondern nur die Assoziation zwischen Port und Handler unterbrochen wird.

### A.2.2 Parameterübergabe

Bei der Parameterübergabe und der Rückgabe von Funktionsergebnissen sollte auf einheitliche Regeln geachtet werden. Es gibt im wesentlichen zwei unterschiedliche Konventionen, die festlegen, wann Zeiger oder Referenzen zu verwenden sind.

Eine Konvention verwendet dann Referenzen auf Objekte, wenn der Anwender ein gültiges Objekt übergeben muß, bzw. wenn garantiert ist, daß ein gültiges Objekt zurückgegeben wird. Zeiger werden immer dann verwendet, wenn ein Objekt optional ist, d.h. auch ein Null-Zeiger erlaubt ist. In diesem Fall müssen die Zeiger explizit auf Null überprüft werden.

Die andere gängige Variante verwendet Zeiger, wenn intern ein Verweis auf das übergebene Objekt gespeichert wird. Referenzen werden verwendet, wenn das Objekt nur während des Funktionsaufrufes gültig sein muß. Diese Methode hat den Nachteil, daß in Fällen, bei denen Null-Zeiger erlaubt sind, trotzdem Zeiger verwendet werden müssen, obwohl kein Verweis auf das Objekt gespeichert wird. Sie ist deshalb nicht konsistent und muß entsprechend kommentiert werden.

Aus diesem Grund verwendet die Simulationsbibliothek die erste Konvention. Die folgenden Beispiele sollen einige Anregungen geben:

```
void RegisterPort(TPort & port);
```

Um einen Port dem System bekannt zu machen, muß dieser existieren. Es macht keinen Sinn, diese Methode ohne einen gültigen Port aufzurufen. Deshalb wurde als Parameter eine Referenz auf den Port gewählt. Der Anwender wird damit gezwungen, ein gültiges Objekt zu übergeben.

```
TEntity(const TString & name, TEntity * owner = 0);
```

Der Konstruktor der Klasse TEntity benötigt zwei Parameter, den Namen der Modellkomponente sowie deren übergeordnete Instanz. Der Name muß angegeben werden, deshalb wird an dieser Stelle eine Referenz verwendet. Der übergebene Name wird intern kopiert und vom Konstruktor nicht verändert. Die Referenz wurde aus diesem Grund als konstant deklariert. Konstante Referenzen werden immer dann eingesetzt, wenn der Aufwand zum Kopieren des Wertes zu groß ist oder die polymorphen Eigenschaften eines Objektes benötigt werden. Wäre der Name als Wert übergeben worden, so wäre erst einmal eine lokale Kopie für den Zugriff innerhalb des Konstruktors angefertigt worden. Die Zuweisung dieser lokalen Kopie an das *fName*-Feld des *TEntity*-Objektes hätte einen zweiten Kopiervorgang ausgelöst. Durch Verwendung der konstanten Referenz konnte ein Kopiervorgang vermieden werden.

Da nicht jede Modellkomponente eine übergeordnete Komponente besitzt, wird der zweite Parameter als Zeiger übergeben. Da er Null als Defaultwert hat, muß er nur angegeben werden, wenn eine übergeordnete Instanz vorhanden ist.

```
const TString & LocalName() const;
```

Das letzte Beispiel zeigt eine Methode der Klasse *TEntity*, die den lokalen Namen der Modellkomponente zurückgibt. Da jede Modellkomponente einen Namen haben muß, garantiert diese Methode, daß ein gültiges String-Objekt zurückgegeben wird. Der Anwender kann deshalb auf eine zusätzliche Prüfung verzichten. Im Gegensatz dazu, muß der Rückgabewert bei allen Methoden, die einen Zeiger zurückgeben, auf Null überprüft werden.

Falls eine Funktion, die eine Referenz zurückgibt, kein gültiges Objekt findet (z.B. weil ein ungültiger Eingabeparameter übergeben wurde), signalisiert sie dies durch eine Ausnahme.

Man könnte in allen Fällen, bei denen dies möglich ist, einen Zeiger zurückgeben. Allerdings muß dann der Anwender jeden Rückgabewert aktiv überprüfen. Ein Vorteil der Ausnahmebehandlung liegt darin, daß sie automatisch abläuft. Der normale Code kann so geschrieben werden, als würde nie ein Fehler auftreten. In Kapitel 4.12 wird näher auf die Vorteile der Ausnahmebehandlung eingegangen.

## A.2.3 Namenskonventionen

Die Lesbarkeit von Programmen läßt sich durch geeignete Konventionen bei der Namensgebung von Variablen, Funktionen, etc. wesentlich verbessern. Die Richtlinien der Simulationsbibliothek werden im folgenden kurz zusammengefaßt, wobei die wichtigste Regel lautet, möglichst aussagekräftige Namen zu verwenden.

### A.2.3.1 Typvereinbarungen

Alle Namen von Typen beginnen mit Großbuchstaben. Zusätzlich beginnen Klassennamen mit einem „T".

Beispiele: Boolean, TEntity

### A.2.3.2 Funktionsnamen

Namen von Funktionen und Methoden beginnen mit einem großen Buchstaben.

### A.2.3.3 Variablennamen

Namen von Variablen beginnen mit einem kleinen Buchstaben. Dabei gelten folgende Regeln:

| Variablentyp | Erster Buchstabe | Beispiel |
|---|---|---|
| Datenmember von Klassen | „f" für „Feld" | fName |
| Globale Variable | „g" für „global" | gPrintManager |
| Private Globale Variable (*static*) | „p" für „privat" | TStatistic::pKeywords |
| Konstante | „k" für „konstant" | kStdEventType |
| Lokale Variable und Parameter | beliebiger Kleinbuchstabe | i, next, ... |

### A.2.3.4 Zusammengesetzte Namen

Werden Namen aus mehreren Wörtern zusammengesetzt, so folgt das erste Wort des Namens den vorangegangenen Konventionen. Alle weiteren Worte schließen daran direkt an und beginnen jeweils mit einem Großbuchstaben.

Beispiele:

| | |
|---|---|
| TPortManager | Name einer Klasse |
| fMessageHandler | Datenmember einer Klasse |
| DrawContent | Member-Funktion einer Klasse |
| gPrintManager | globaler Wert |
| currentValue | lokale Variable/Parameter |
| kMaxSize | Konstante |