Projektbericht '98

Projekt SIGN
Universität Bremen
Arbeitsgruppe Datenbanksysteme
stp@informatik.uni-bremen.de

Vorwort

Das Projektstudium ist ein wichtiger Bestandteil des Bremer Informatik-Studiums. In diesen Projekten schließen sich jährlich einige Studenten für zwei Jahre zusammen, um eine Aufgabe selbstorganisiert durchzuführen. In unserem Fall trafen sich Anfang des Wintersemesters 1996/1997 19 Studenten, um das von Prof. Dr. Martin Gogolla, Dipl.-Inf. Mark Richters und Dipl.-Inf. Heino Gärtner betreute Projekt zu gestalten. Lediglich ein Student wechselte noch in der ersten Semesterwoche in ein anderes Projekt, alle anderen hielten bis zum Ende des Sommersemesters 1998 durch. Auch Heino Gärtner verließ uns leider vorzeitig, um sich verstärkt seiner Dissertation zu widmen. Bis zum Ende wurde das Projekt von folgenden Studenten getragen: Ersin Canli, Christian Cordes, Derya Dogan, Siyamak Ghotbi, Hamid Habet, Marcus Koch, Arne Lindow, Martin Mann, Holger Neumann, Frank Patz, Marc Plaßmeier, Andreas Rugo, Verena Schmidt, Oliver Spittel, Rainer Spittel, Karlo Wagner, Sandro Wendt und Chun-Hon Yap.

Jeder von uns brachte natürlich eigene Vorstellungen mit, wie so ein Projekt ablaufen sollte. Und so war es teilweise nicht einfach, den Ablauf zu organisieren. Aus diesem Grund teilten wir uns in verschiedene Gruppen auf, die sich den einzelnen Aufgaben im Projekt zuwenden sollten. Dies waren zunächst die Management-, Sprachen-, Werkzeug- und Dokumentationsgruppe; im weiteren Verlauf des Projektes bildeten sich dann weitere Gruppen und auch die starre Einteilung der Gruppen löste sich teilweise auf. So gelang es, daß wir zielstrebig auf unsere Ziele hinarbeiten konnten.

Zu den Zielen eines solchen studentischen Projektes gehören natürlich nicht nur die fachlichen Ziele. Besonders wichtig ist es, einmal Erfahrungen bei der Entwicklung eines umfangreichen Softwaresystems in einem großen Team zu sammeln. Daß dies nicht einfach ist, konnten wir alle im Laufe der zwei Jahre ausgiebig feststellen.

Inhaltsverzeichnis

1 Einleitung **6**

2 Vorbereitungsphase **8**
2.1 Projektplan . . . 8
2.1.1 Ausgangssituation . . . 8
2.1.2 Ziele . . . 11
2.2 Anforderungsdefinition . . . 15
2.2.1 Funktionalität . . . 15
2.2.2 Benutzungsschnittstelle . . . 16
2.3 OMT Vorträge . . . 21
2.3.1 Objektmodell . . . 21
2.3.2 Erweitertes Objektmodell . . . 37
2.3.3 Dynamisches Modell . . . 52
2.3.4 Funktionales Modell . . . 70
2.4 Fallstudien . . . 76
2.4.1 Autovermietung . . . 76
2.4.2 Garagentor . . . 80
2.4.3 Konferenzorganisation . . . 85
2.4.4 Netztopologie . . . 90
2.4.5 Scotland Yard . . . 97

3 Projekt-Infrastruktur **101**
3.1 Corporate Website . . . 103
3.1.1 Sitemap . . . 104
3.2 SGML . . . 106
3.2.1 Konzeption . . . 106
3.2.2 Motivation . . . 107
3.2.3 SGML Tools . . . 108
3.2.4 SGML in der Praxis . . . 110
3.3 CVS Versionskontrolle . . . 113
3.3.1 Was ist CVS ? . . . 113
3.3.2 Wie wird CVS konfiguriert ? . . . 113
3.3.3 Wie wird mit CVS gearbeitet ? . . . 113
3.4 Java . . . 116
3.4.1 Einleitung . . . 116
3.4.2 Einführungsbeispiel . . . 118
3.4.3 Sprachelemente . . . 120
3.4.4 Objektorientierte Programmierung . . . 128

3.4.5 Klassenbibliotheken . . . 132
3.4.6 Techniken der Programmentwicklung . . . 134

4 Konstruktionsphase I **139**
4.1 Systementwurf I . . . 139
4.1.1 Vorwort . . . 139
4.1.2 Entwurf . . . 143
4.1.3 Transformation nach SQL . . . 196
4.1.4 Transformation nach Java . . . 206
4.1.5 Entwurfserweiterungen . . . 214
4.2 Sprachentwurf I . . . 244
4.2.1 Basisnotation . . . 244
4.3 Semantik I . . . 257
4.3.1 Vorwort . . . 257
4.3.2 Syntax der SIGN ML . . . 258
4.3.3 Konventionen . . . 258
4.3.4 Klassen . . . 258
4.3.5 Klassenbeziehungen . . . 262
4.3.6 Qualifikation . . . 264
4.3.7 Candidate Key . . . 264
4.3.8 Constraints . . . 264

5 Konstruktionsphase II **266**
5.1 Systementwurf II . . . 267
5.1.1 Logische Elemente der State-Charts . . . 267
5.1.2 Logische Elemente: Activity Charts . . . 267
5.1.3 Diagramm Elemente StateCharts . . . 269
5.1.4 Diagramm Elemente: Activity Charts . . . 274
5.2 Sprachentwurf II . . . 278
5.2.1 Statecharts . . . 278
5.2.2 Activity-Charts . . . 286
5.2.3 Constraints . . . 294
5.3 Semantik II . . . 311
5.3.1 Vorwort . . . 311
5.3.2 Deterministische Sequentielle Automaten . . . 313
5.3.3 Nichtdeterminismus . . . 326
5.3.4 ϵ -Transitionen . . . 326
5.3.5 Transformation nach CSP . . . 327
5.3.6 Termination . . . 329
5.3.7 Hierarchische Dekomposition . . . 330
5.3.8 Parallele Komposition . . . 345

6 Implementierung **348**
6.1 Voraussetzungen . . . 348
6.2 Styleguide (Programmierrichtlinien) . . . 350
6.2.1 Allgemeines . . . 350
6.2.2 Programmstruktur . . . 350
6.2.3 Bezeichner . . . 351
6.2.4 Kommentare . . . 352
6.2.5 Dateien . . . 353

6.3 Statistik 353

7 Schlußwort **354**

A Anhang **356**
A.1 Handbuch Graphischer Editor deSIGN 356
A.1.1 Einleitung 356
A.1.2 Technische Voraussetzungen 356
A.1.3 Übersicht der Programmfunktionen 357
A.1.4 Beispiel 371
A.1.5 Fehlermeldungen und Problembehebung 373
A.2 Sprachvergleiche 376
A.2.1 OMT 376
A.2.2 OOD 393
A.2.3 BOOCH 396
A.2.4 STATEMATE 406
A.2.5 UML 418
A.3 Autorenliste 448

Stichwortverzeichnis **449**

Literaturverzeichnis **454**

Kapitel 1

Einleitung

Das Projekt SIGN (Software Implementation durch Graphische Notation), ehemals stp (Software Through Pictures), begann im Wintersemester 96/97 und endete mit dem Sommersemester 98. In den vergangenen zwei Jahren haben wir viel von dem, was wir uns als Projektziele vorgenommen hatten, erreicht (siehe Projektplan).

Das erste Semester wurde von uns dazu genutzt, um uns selbst darüber klar zu werden, was wir eigentlich mit dem Projekt erreichen wollen. So verging einige Zeit mit der Formulierung der Projektziele. Des weiteren wurde nach graphischen Sprachen gesucht, die für unsere Zwecke passend erschienen: Die Arbeiten von Booch [Boo91], Rumbaugh [RBP+91], und anderen wurden unter die Lupe genommen und auf ihre Tauglichkeit hin untersucht. Schließlich wurde von uns eine eigene Sprache SIGN entwickelt, die unseren Ansprüchen gerecht wurde. Parallel dazu wurde ermittelt, welche Programmiersprachen geeignet sind, um das Programmiervorhaben zu realisieren. Wichtig war uns dabei vor allen Dingen, daß das fertige Produkt am Ende auf allen wichtigen Plattformen laufen kann. Nach einer kurzen Zeit der Vergleiche entschieden wir uns hier für Java, einer damals neuen Sprache auf dem Softwaremarkt.

Im zweiten Projektsemester wurde dann das Konzept für SIGN fertiggestellt und auch der Entwurf unseres Projektes nahm Gestalt an. Außerdem wurden die ersten Prototypen implementiert.

Das dritte Semester war ein Programmier-Semester. Aber auch an unserer Sprache wurde weitergearbeitet und so ein zweiter Sprachentwurf geschaffen. Damit dieser dann auch durch unser Programm dargestellt werden kann, mußte der Entwurf angepaßt werden. So entstand schließlich der Entwurf II.

Im vierten und letzten Semester überschlugen sich förmlich die Ereignisse: Immer wieder wurden in (fertig?) implementierten Programmabschitten neue Fehler entdeckt, die behoben werden wollten. Aber für jeden Bug, der verschwand, schienen neue aufzutauchen. Des weiteren rückte der Projekttag, auf dem SIGN sich von seiner besten Seite präsentieren sollte, immer näher und wir waren froh, daß wir am 17.07.1998 eine wirklich lauf- und präsentierfähige Version vorliegen hatten. Aber für den Projekttag mußte nicht nur implementiert werden - auch die Vorführung und die Organisation des Projekttages kostete viel Arbeit.

Der hier vorliegende Abschlußbericht soll noch einmal die geleistete Arbeit im Projekt verdeutlichen. Obwohl längst nicht jede Arbeit schriftlich dokumentiert wurde, ist auch so schon eine Menge Material zusammengekommen, um einen Einblick in unsere Arbeit zu geben. Der Bericht gliedert sich in die Bereiche „Einleitung“, „Vorbereitungsphase“, „Infrastruktur“, „Konstruktionsphase 1“, „Konstruktionsphase 2“, „Implementierung“, „Schlußwort“ und „Anhang“.

In der Vorbereitungsphase werden der Projektplan einschließlich der Projektziele, die Anforderungsdefinition an unser Programm deSIGN, im Rahmen der Sprachentwicklung gehaltene Vorträge und Fallstudien beschrieben.

Im Kapitel Infrastruktur sind Dokumente enthalten, die zwar für den Fortgang des Projektes wichtig waren, sich jedoch nicht direkt mit der Sprache oder dem Editor beschäftigen. So sind hier Einführungen in SGML (hiermit wurde das komplette Projekt dokumentiert - auch der Abschlußbericht wurde so verfaßt) und Java (die Programmiersprache, in der deSIGN implementiert wurde) enthalten.

Die Konstruktionshasen 1 und 2 beschäftigen sich mit dem Entwurf der Sprache SIGN und des graphischen Editors deSIGN. In der ersten Phase wird dabei nur auf das statische Modell eingegangen, da uns dies am wichtigsten für die Modellierung und den Softwareentwurf im allgemeinen erschien. Im zweiten Abschnitt haben wir uns dann mit dem dynamischen und funktionalen Modell auseinandergesetzt und hierfür dann auch den zweiten Entwurf angefertigt, nachdem der Sprachentwurf II abgenommen war.

Im Kapitel Implementierung gehen wir auf den Softwareentwicklungsprozeß in unserem Projekt ein.

Außer diesen eben genannten Dokumentteilen gehören jedoch noch weitere zu einer solch umfangreichen Arbeit. Leider lassen sich diese nicht immer einfach bestimmten Themenkreisen zuordnen, und so entschieden wir, diese in den Anhang zu verbannen. Allerdings sei darauf hingewiesen, daß diese Dokumente nicht weniger interessant sind, als die anderen. In diesem Kapitel findet man ein Handbuch für den von uns erstellten Editor deSIGN, Sprachvergleiche zwischen den bekanntesten objektorientierten Modellierungssprachen und eine Autorenliste. Zum letzten Punkt an dieser Stelle noch eine Anmerkung: Jeder im Projekt hat in gewisser Weise an allen Dokumenten mitgearbeitet - schließlich wurden alle Entscheidungen ausführlich im Projektplenum diskutiert und man konnte sich darauf verlassen, daß immer jemand Rat wußte, wenn man selber als Autor einmal Hilfe benötigte.

Kapitel 2

Vorbereitungsphase

Dieses Kapitel beschreibt die Grundaktivitäten des Projektes SIGN. Die Grundaktivitäten setzen sich aus den Festlegungen der Projektziele, sowie den Anforderungen an das zu entwickelnde Produkt (Projektplan bzw. Anforderungsdefinition) und die Beschäftigung mit verschiedenen Sprachkonzepten der Modellierungssprache OMT im Zuge der Wissensaneignung (OMT-Vorträge) zusammen. Die Wissensaneignung wird durch verschiedene Szenarien intensiviert (Fallstudien).

2.1 Projektplan

2.1.1 Ausgangssituation

Zu den allgemein anerkannten Grundsätzen erfolgreicher Projektarbeit gehört ein Höchstmaß an Transparenz. Informationen über das Projekt werden in der Regel allen Projektmitgliedern mitgeteilt. Die Aktivitäten jedes einzelnen Teammitglieds im Projekt sind ebenfalls allen bekannt. Ein weiterer Schlüsselfaktor für den Projekterfolg ist die Zieldefinition. Projektziele werden iterativ durch schrittweise Verfeinerung konkretisiert und werden von allen Projektteilnehmern getragen. Nicht zu übersehen ist auch die Bedeutung realistischer Zielsetzungen unter Berücksichtigung verfügbarer Resourcen und einzuhaltender Termine. Das Projektmanagement hat hier die Aufgabe, Planabweichungen frühzeitig zu erkennen und ist gegebenenfalls zur Initiative verpflichtet. Da das Projektmanagement in diesem Projekt nicht mit Kompetenzen ausgestattet ist, besteht die Initiative des Projektmanagements in der Regel in einer Vorlage zur Beschlussfassung durch das Plenum.

Dieses Dokument enthält die Zieldefinition, der Projektaktivitäten und des -fortschrittes. Es wurde in regelmäßigen Abständen durch das Projektmanagement aktualisiert und allen Projektteilnehmer frei verfügbar gemacht. Es ist ebenfalls Gegenstand einer genehmigenden Beschlussfassung durch das Plenum.

2.1.1.1 Die Bedeutung von Softwaretechnik-Methoden wächst

Komplexere Anwendungsgebiete

Die entwickelten Softwaresysteme werden komplexer und sind schneller Veränderungen unterworfen. Während in früheren Phasen des Einsatzes von Informationstechnik die Automatisierung von Arbeitsabläufen (Datenverarbeitungs-Ära) das Hauptanwendungsgebiet war, geht es in den nachfolgenden Phasen zunehmend um die Bereitstellung von Informationen für das Management (Mikro-Ära) sowie um die systemübergreifende Integration von Anwendungen (Netzwerk-Ära). Die Bedeutung von Informationstechnik für ein Unternehmen und die Entwicklung der Einflußnahme der Unternehmensführung auf die Systementwicklung haben sich analog entwickelt. Hier werden die Phasen DP-Era (Data-Processing), MIS-Era (Management Information Systems) und SIS-Era (Strategic Information Systems) unterschieden [KR96]. Zugleich wird die Aufgabe der Systementwicklung heute umfassender gesehen. Der Softwareentwicklungsprozeß befaßt sich nicht mehr nur mit technischen Aspekten, sondern mit dem gesamten soziotechnischen System. Sinn und Zweck von Softwaretechnik-Methoden ist die Beherrschung von Komplexität. Dies geschieht durch Konzentration auf wichtige Aspekte des Anwendungsgebiets und der Systemaufgaben. Die Entwicklung der Einsatzgebiete und der Bedeutung von Informationstechnik machen die zunehmende Bedeutung von Softwaretechnik-Methoden deutlich. Softwareentwicklung findet zunehmend auf einer konzeptionellen Ebene statt. Gleichzeitig sind die heutigen Systeme (MIS, SIS) mehr datenorientiert. Die funktionale Komplexität tritt demgegenüber in den Hintergrund. Diese Entwicklung erklärt, warum sich objektorientierte Methoden bereits während der Analyse- und Design-Phase immer grösserer Beliebtheit erfreuen.

Qualität

Bei methodischem Vorgehen in der Analyse- und Design-Phase des Softwareentwicklungsprozesses werden überprüfbare Ergebnisse erzeugt. Eine formale Softwarespezifikation, z.B. mittels graphischer Modelle, kann stets validiert werden - d.h. auf Übereinstimmung mit den Anforderungen des Anwendungsgebiets überprüft werden. Das fertige System kann anhand der formalen Spezifikation verifiziert werden - d.h. die Einhaltung der Spezifikation kann überprüft werden. Die formale Spezifikation des Systems hilft, konzeptionelle Fehler zu vermeiden. Insbesondere Methoden mit graphischer Notation bieten gute Voraussetzungen zur Förderung der Kommunikation von Systementwicklern bzw. Analytikern und Experten des jeweiligen Anwendungsgebiets. Im Sinne der soziotechnischen Systemgestaltung ist die Kommunikation über das Modell ein wesentlicher Bestandteil des Entwicklungsprozesses. Die Beteiligung der Benutzer am Entwicklungsprozess wird durch eine leicht verständliche Notation gefördert.

Wartbarkeit

Durch die systematische, formale Spezifikation des Systems wird eine hohe Transparenz erreicht. Das System läßt sich leichter auf geänderte Anforderungen anpassen.

Wiederverwendbarkeit

Objektorientierte Methoden versprechen eine durchgängige Darstellung der Arbeitsergebnisse. Dadurch wird die Konsistenz zeitlich aufeinanderfolgender Arbeitsergebnisse gesichert. Objektorientierte Methoden fördern die Orientierung an fachlichen Begriffen des Anwendungsgebiets. Invariante Eigenschaften des Anwendungsgebiets werden so leichter erkannt. Im Ergebnis entstehen stabile, wiederverwendbare Spezifikationen. Nicht nur Teile der implementierten Software lassen sich wiederverwenden, sondern auch die Ergebnisse der Analyse- und Design-Phase. Zudem wird leichter erkannt, welche Teile von bereits existierenden Systemen wiederverwendet werden können, wenn dies bereits auf einer konzeptionellen Ebene geschieht.

Effizienter Entwicklungsprozess

Aufgrund der zunehmenden Komplexität der entwickelten Systeme wächst die Gefahr spät erkannter konzeptioneller Fehler. Die Konzentration auf konzeptionelle Eigenschaften des Systems und deren formale Spezifikation verspricht daher eine deutliche Reduzierung von aufwendigen Nachbesserungen. Dem erhöhten Aufwand für die formale Spezifikation steht eine verbesserte Qualität der Ergebnisse, eine bessere Wartbarkeit sowie ein höherer Grad an Wiederverwendung von Arbeitsergebnissen gegenüber.

2.1.1.2 Die Anwendung von Softwaretechnik-Methoden bereitet in der Praxis Schwierigkeiten

Methodenauswahl

Eine Vielzahl unterschiedlicher Methoden bietet sich zur Auswahl an. Die Methoden besitzen unterschiedliche Eigenschaften hinsichtlich der Modellierung von verschiedenen Sichten auf ein System. Für unterschiedliche Systeme sind daher auch unterschiedliche Methoden geeignet.

Werkzeugunterstützung

CASE-Tools sind teuer und können nur sinnvoll eingesetzt werden, wenn in einem Unternehmen bereits ein formalisierter Software-Entwicklungsprozeß stattfindet. Die Anschaffung eines Werkzeugs bedeutet praktisch die Festlegung auf eine bestimmte Methode. Der Wechsel zu einer anderen Methode, die in einem neuen Projekt vielleicht adäquater wäre, wird erschwert. Die Anwendung von formellen Entwurfsmethoden (insbesondere graphischen) ohne spezialisiertes Werkzeug erhöht den Aufwand.

Übergang zur Implementation

Viele Methoden überlassen es der Phantasie des Entwicklers, die im Entwurf modellierte Software in einer konkreten Zielsprache zu implementieren. Werkzeuge, die den Übergang von einem Entwurf zu einer Implementation in einer konkreten Programmiersprache unterstützen, sind kaum verbreitet.

Fehlendes Reverse Engineering

Die Wiederverwendung bereits bestehender Module kann in der Regel erst in der Implementationsphase erfolgen, da keine Modelle hiervon vorliegen. Eine automatische Übernahme bereits bestehender Sourcen in ein Modell ist kaum möglich. Da das hierzu notwendige Reverse Engineering Verfahren in der Regel nicht vorhanden ist, müssen die Metadaten des Modells von Hand eingegeben werden.

Vorgehensmodelle

Ein Grund für mangelnde Sorgfalt bei Analyse und Entwurf im Software-Entwicklungsprozess dürfte in falsch verstandenen Vorgehensmodellen liegen. So kann ein zyklisches Vorgehensmodell (Spiralmodell) und der Einsatz von Prototyping dazu verleiten, Analyse und Entwurf zu vernachlässigen. Der Aufwand für eine formale Spezifikation und eine ausgedehnte Analysephase wird eingespart. Stattdessen dient ein Prototyp als Grundlage der Benutzer-Entwickler Kommunikation. Bei diesem Verfahren wird versucht, die benötigten funktionellen Eigenschaften des Systems quasi im Versuch-und-Irrtum-Verfahren über die Eigenschaften der Benutzerschnittstelle zu ergründen. Nachprüfbare und wiederverwendbare Arbeitsergebnisse werden hierbei jedoch kaum erzeugt.

2.1.2 Ziele

Das Projekt SIGN (Software Implementation durch Grafische Notation) - ehemals StP (Software through Pictures) - ist ein studentisches Projekt. Die hier verfolgten Ziele sind deshalb im Kontext der durch die Studienordnung für das Hauptstudium im Studiengang Informatik vorgegebenen Ziele zu sehen. Es ist danach zu berücksichtigen, daß sich Projekte „nicht nur an dem angestrebten Resultat, sondern auch an den bei ihrer Durchführung gesammelten Erfahrungen“ orientieren. „Sie sind eine Form des Lehrens und Lernens [UB93].“ Dies impliziert eine starke Gewichtung der hier unter Vorgehensziele genannten Lernziele (Fachwissen, Teamarbeit, Präsentation). Allgemeine Ergebnisziele ergeben sich aus dem in der Studienordnung genannten Gegenstand von Projekten: der Analyse, Planung, Gestaltung, Einsatz und Bewertung rechnergestützter Systeme und Verfahren. Ferner wird in der Studienordnung die Berücksichtigung von Sozialzielen, insbesondere praktische Relevanz und Bezugnahme auf gesellschaftliche Zusammenhänge gefordert. Eine erste Konkretisierung der Ergebnisziele des Projekts SIGN geht aus dem Projektantrag hervor. „Ziel des Projekts ist die Erarbeitung einer exemplarischen Software-Entwicklungsumgebung für Programme mit persistenter Haltung von Daten [Gog96b].“ Die Software-Entwicklungsumgebung soll die Software-Spezifikation mittels graphischer Formalismen unterstützen. Damit soll die Konzentration auf konzeptionelle, anwendungsbezogene Strukturen (gegenüber konkreten, implementierungsnahen Strukturen) gefördert werden. SIGN beschäftigt sich nicht mit dem Problem von Analyse und Entwurf von Systemen im Allgemeinen. Ziel ist die Erarbeitung eines Werkzeugs, das es ermöglicht, die Ergebnisse der Systemanalyse und des Entwurfs zu spezifizieren. Dazu gehört eine formelle Notation sowie eine rechnergestützte Entwicklungsumgebung. Das SIGN Projektergebnis soll die Vorgehensweise nach einer objektorientierten Methode sinnvoll unterstützen. Es soll jedoch nicht eine neue Methode entwickelt werden.

2.1.2.1 Ergebnisziele

Funktionsziele

Die als Basisziele aufgeführten Ziele sind bereits hinreichend klar, um ihre Erreichbarkeit und ihren Nutzen einschätzen zu können. Diese Ziele gelten als Grundkonsens aller Teilnehmer im Projekt SIGN. Die unter Erweiterungen aufgeführten Ziele bedürfen weiterer Prüfung. Aus ihnen können im Projektverlauf iterativ neue konkrete Zielvorstellungen entwickelt werden.

Basisziele

Das grundsätzliche Ziel des Projektes ist der Entwurf einer graphische Sprache für ein statisches Objektmodell zur Spezifikation der Datensicht von Programmen. Desweiteren ist ein graphischer Editor für diese Objektmodelle vorgesehen, der eine Abbildung und strukturelle Implementierung des Schemas in der Zielsprache (Java oder SQL) ermöglicht. Language Mapping: Übersetzung in relationales Datenbankschema bzw. SQL-Script mit DDL-Befehlen.

Erweiterungen

Da sich die Mächtigkeit des Projektes und der gesamte Zeitaufwand für die Basisziele zum Zeitpunkt der Zieldefinition noch nicht abschätzen lassen, lassen sich Erweiterungen nur als optionale Ziele festlegen. Möglich wären die Einbindung weiterer existenter Sichten in den Editor, wie das dynamische Modell (Verhaltenssicht) sowie das funktionelle Modell (Funktionssicht) sowie die Einbettung in eine Methode. Desweiteren wäre eine Übersetzung in C++-Source-Code möglich, um die Effizienz des Editors zu erhöhen. Weitere Erweiterungen könnten Sichtkonzept, logische Datenunabhängigkeit sowie die Möglichkeit zum Reverse Engineering sein.

Sozialziele

Die Sozialziele lassen sich grob in zwei Funktionsgruppen aufteilen: Präsentation und Anwendung. Der Hauptpunkt ist hierbei die praktische Relevanz, d.h. sowohl Sprache als auch Editor sollen praktisch sinnvoll entwickelt werden, so daß beide einfach und verständlich für den Einsteiger als auch mächtig und funktional genug für den geübten Anwender sind. Weiterhin soll eine Softwarespezifikation leicht verständlich abgebildet werden, so dass eine Präsentation auch für Nicht-Experten möglich ist.

2.1.2.2 Vorgehensziele

Termine

Die Meilensteintermine des jeweiligen Semesters sind unter Berücksichtigung der Semesterferien bis zum Beginn des nächsten Semesters zu erfüllen.

1.Semester

- Festlegung des Projektziels
- Wissensaneignung der entsprechenden Funktionsgruppe
- Spezifikation des Prototypen bzw. der formalen Sprache
- Entwicklungsumgebung/Sprache
- Prototyp des Editors entwerfen/implementieren
- Erster Entwurf der graphischen Sprache

2.Semester

- Fallstudie zum Umgang mit dem Editor
- Mögliche Strukturveränderung des Editors
- Analyse möglicher Zielsprachen
- Fertigstellung der graphischen Sprache
- Ausweitung des Umfangs der graphischen Sprache um weitere Konzepte(Sichten, Transformationen)
- Ausweitung des Editors um weitere Konzepte und Funktionen

3.Semester

- Festschreibung der Zielsprache
- Implementation der Zielsprachentransformation
- Qualitätssicherung anhand einer Fallstudie
- Erweiterung um Anwendungskonzepte (DB, verteilte Systeme, Einbindung und Erweiterung existenter Anwendungen)

4.Semester

- Debugging
- Umsetzung für weitere Systeme/Zielsprachen
- Abschlußbericht

Kosten

Für das Projekt sind keinerlei Kosten veranschlagt; Räumlichkeiten sowie Hard- und Software werden von der Universität Bremen gestellt.

Lernen

Darüber hinaus hat das Projekt auch lehrende Aspekte. Zum einen soll das Anwenden bereits erlernter wissenschaftlicher Methoden und die Analyse und Präzisierung einer konkreten Aufgabenstellung geübt werden. Zum anderen wird besonderer Wert auf arbeitstechnische Aspekte gelegt; es soll in der beruflichen Praxis geforderte arbeitsteilige, gruppenorientierte Arbeit sowie teamübergreifende Problemanalyse und -lösung geübt werden. Desweitern sollen Arbeitsergebnisse wissenschaftlichen Anforderungen entsprechend präsentiert werden.

2.2 Anforderungsdefinition

2.2.1 Funktionalität

In diesem Abschnitt soll geklärt werden, welche Funktionen der Editor deSIGN dem Benutzer zur Verfügung stellt.

2.2.1.1 Objekte der Grafik

Jede deSIGN-Grafik besteht aus Grafikobjekten. Dies können z.B. Klassen, Assoziationen und Aggregationen sein.

Grundlegend für das Arbeiten mit deSIGN ist die Möglichkeit, Grafikobjekte zu erzeugen. Dies geschieht über die Toolbar, aus der man sich das gewünschte Objekt per Drag-and-Drop auf die Arbeitsfläche herausziehen kann. Für das weitere Arbeiten mit den Objekten müssen diese nun markiert werden. Dabei kann man entweder einzelne Objekte oder ganze Bereiche markieren.

Zu den Bearbeitungsfunktionen gehören: Lösch-, Verschiebe- und Kopierfunktion. Des weiteren kann man für einzeln selektierte Objekte den Eigenschaftsdialog aufrufen, um dort Einträge wie z.B. Beschriftung vorzunehmen. Es ist aber auch eine Funktion vorgesehen, um Kommentare in die Grafik einzubinden. Hierbei wird zwischen Programmkommentaren (werden in den Programmcode übernommen) und Grafikkommentaren (dienen nur zur Erleichterung des Verständnisses beim Zeichnen) unterschieden.

Zum Schluß bietet deSIGN die Möglichkeit, die erzeugten Objekte automatisch so anzuordnen, daß eine möglichst große Übersicht gegeben ist.

2.2.1.2 Verwaltung der Grafik

Das Wichtigste um Grafiken bearbeiten zu können, ist die Entwicklungsumgebung. Wenn man eine neues SIGN-Diagramm erstellen möchte, dann bekommt man per Knopfdruck eine neue Entwicklungumgebung bereitgestellt.

Nun läßt sich jede beliebige SIGN-Grafik auf dem Bildschirm darstellen. Falls es erwünscht ist, die SIGN-Grafik größer oder kleiner zu betrachten, kann man ganz einfach den Zoom einstellen.

Ist schon eine angefangene oder fertige SIGN-Grafik vorhanden, läßt sich diese in den Editor laden. Hierbei erkennt deSIGN automatisch, ob die gewünschte Datei das richtige Format hat und zeigt entsprechende Warnungen an. Geänderte oder neue SIGN-Grafiken kann man speichern. Es ist auch möglich, sie unter neuem Namen zu speichern bzw. in einem anderen Format.

Natürlich kann man die SIGN-Grafiken auch drucken. Um vorher sehen zu können, wieviel Seiten die Grafik benötigt und wie die Teile der Grafik über die Seiten verteilt sind, gibt es die Möglichkeit der Seitenansicht.

Ein weiteres Konzept von deSIGN ist das Feststellen von mehrfach gleichen Bezeichnungen. Das ist sinnvoll, um nicht aus Versehen mehreren Klassen den gleichen Bezeichner zu geben.

Für den Fall, daß irgend etwas beim Erstellen einer SIGN-Grafik schiefgeht, gibt es die Option „Letzten Schritt rückgängig" (optional mehrere Schritte), mit der es möglich sein wird, den zuletzt ausgeführten Befehl rückgängig zu machen. Wenn man nach dem Rückgängig-machen feststellen sollte, daß der entsprechende Befehl doch richtig war, gibt es die Möglichkeit des Wiederherstellens. Dies ist einfach nur die Umkehrfunktion zu „Letzten Schritt rückgängig".

Es kann leicht passieren, daß man beim Arbeiten das Zwischendurch-Speichern vergißt. Kommt es in solch einem Fall zu einem Stromausfall oder einem Systemabsturz, so ist dies vor allen Dingen ärgerlich, denn alle nichtgespeicherten Daten sind im Regelfall verloren. Um einem solchen Verlust vorzubeugen, bietet deSIGN eine automatische Sicherung während der Arbeit an.

Große Grafiken wirken leicht unübersichtlich, wenn jede Kleinigkeit ständig sichtbar ist. Damit dies bei SIGN-Grafiken nicht der Fall ist, kann man mit deSIGN die Eigenschaften/Attribute ein- und ausklappen. Damit ist der Name der Klasse noch erkennbar und die Attribute auf Wunsch anzeigbar. Ein weiterer Schritt zur Übersichtlichkeit ist die Black-Box. In ihr können beliebige Gruppen von Klassen, Objekten usw. versteckt werden.

2.2.1.3 Unwandlung von Grafik in Programmtext

Jede fertige SIGN-Grafik soll in Programmtext umgewandelt werden können.

deSIGN kann die Objekte der Grafik erkennen und übersetzen. Hierbei werden die Beschriftungen und Programmkommentare übernommen und die Grafikkommentare ignoriert.

2.2.1.4 Hilfefunktionen

Natürlich kommt es immer mal vor, daß man Hilfe benötigt. Hierzu bietet deSIGN die ausführliche mitgelieferte Dokumentation, die sämtliche Funktionen erläutert. Eine Online-Dokumentation ist ausdrücklich nicht vorgesehen.

2.2.2 Benutzungsschnittstelle

In diesem Teil der Anforderungsdefinition wollen wir uns mit dem Aufbau der Benutzungsschnittstelle unter zwei Gesichtspunkten beschäftigen. Zum einen soll es um die Anordnung der Fenster, zum anderen um die Benutzerdialoge gehen.

2.2.2.1 Fensteranordnung

Bei den Überlegungen zu einer Gestaltung der Oberfläche des Editors sind wir zu dem Ergebnis gekommen, daß drei getrennte Komponenten in irgendeiner Form zusammengebracht werden müssen. Da wären zuerst das Menü, das alle Funktionen, die der Editor ermöglichen soll, enthält, die Iconbar, die die am häufigsten verwendeten Funktionen schneller erreichbar macht, die Toolbar, in der vorgefertigte Objekte für den Editor angezeigt werden sowie das/die Projektfenster. Wie in den meisten Programmen werden das Menü und die Iconbar zu einer Komponente zusammengezogen. Diese drei wichtigen Komponenten lassen sich auf verschiedene Art und Weise anordnen. Im Folgenden werden die Vor- und Nachteile dieser verschiedenen Anordnungen aufgeführt.

Getrennte Komponenten

Das Trennen der Komponenten in eigenständige Programmteile und Fenster ist eine oft verwendete Form des Bildschirmaufbaus. Diese erlaubt es, die Fenster in einer beliebigen Anordnung auf dem Bildschirm zu plazieren. Mittels Drag-and-Drop kann hier in einfachster Art und Weise von der Toolbar ein Objekt auf einem Projektfenster plaziert werden und sogar von einem Projektfenster in ein anderes verschoben/kopiert werden. Außerdem ist es möglich, mit einer automatischen Anordnung eine schnelle Übersicht herzustellen und alle Fenster in eine geeignete Form zu bringen. Allerdings kann es auch schnell unübersichtlich werden, falls zu viele Projektfenster gleichzeitig geöffnet sind; vor allen bei geringeren Bilschirmaufauflösungen. Die eindeutige Zuordnung der Funktionen (Menü/Icon) zu einem der Projektfenster ist nicht mehr gegeben, da gleichzeitig mehrere aktiv sind und die Funktion nicht entscheiden kann, auf welches Fenster sich die aufgerufene Funktion bezieht. Abbildung 2.1 zeigt, wie von einander getrennte Komponenten auf dem Bildschirm aussehen könnten.

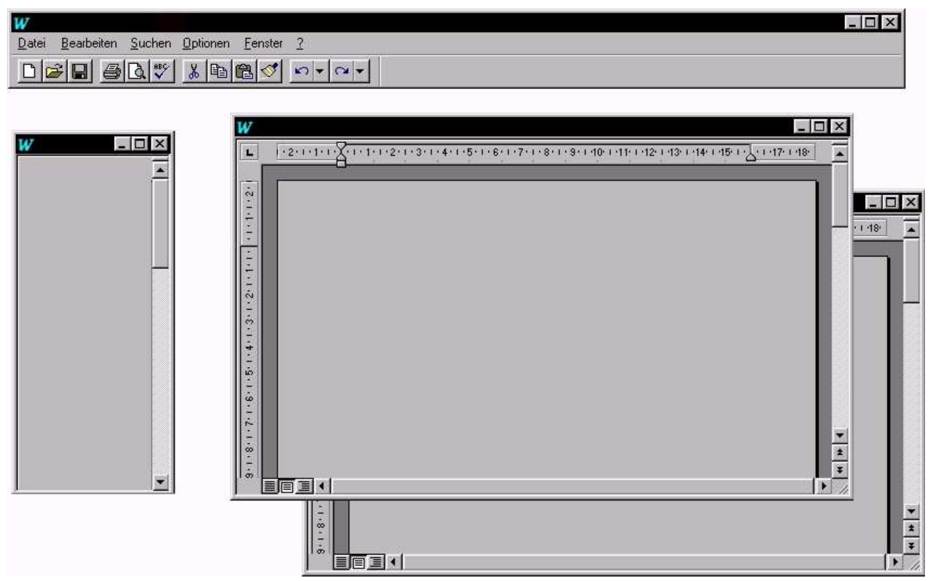


Abbildung 2.1: Getrennte Komponenten

Vorteile:

- individuelle Komposition der Arbeitsoberfläche
- mehrere Projektfenster sind gleichzeitig überschaubar

- Drag-and-Drop zwischen den Projektfenstern
- Komponenten können automatisch angeordnet werden

Nachteile:

- je nach Anzahl der Fenster oder Auflösung kann es unübersichtlich werden
- bei mehreren Projekten ist die Zuordnung der Funktionen (Menü/Icons) nicht eindeutig

Teilweise getrennte Komponenten

Diese Art der Anordnung ist eine Zusammenführung der Menüs/Icons mit den jeweiligen Projektfenstern. Dadurch ist eine Zuordnung aufgerufener Funktionen mit einem Projektfenster wieder gesichert, allerdings werden dadurch wertvolle Ressourcen verschwendet, da jedes Projektfenster ein eigenes Menü und Iconbar besitzen muß (vergleicheAbbildung 2.2).

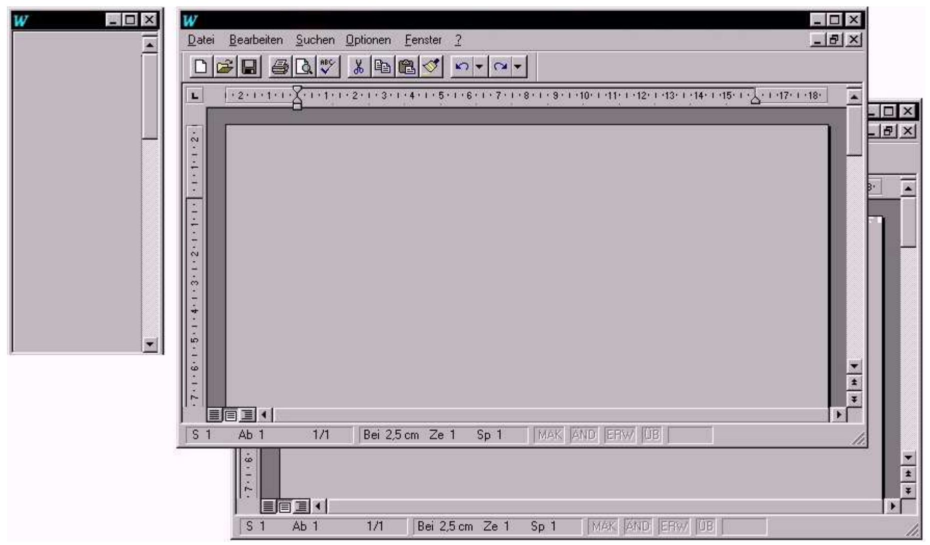


Abbildung 2.2: Teilweise getrennte Komponenten

Andere Möglichkeiten der Zusammenstellung von Komponenten sind z.B. die Zusammenführung von Toolbar und Projektfenster, oder die von Toolbar und Menü/Icons. Allerdings erscheinen auch diese in gleichem Maße ungeeignet, wie die oben beschriebene Zusammenstellung.

Vorteile:

- direkte Zuordnung der Funktion (Menü/Icons) zu den entsprechenden Projekten

Nachteile:

- Überladung der Fenster
- unnötige Verschwendung von Ressourcen

Zusammengefaßte Komponenten

Auch diese Art der Anordnung findet man in vielen Programmen wieder, alle Komponenten sind unter einem Fenster zusammengefaßt (vergleiche Abbildung 2.3). Auch wenn man sich die Fenster nicht selber anordnen kann (nur in einem geringen Maße), ist doch eine Steigerung der Übersicht erreicht. Allerdings kann in diesem Fall immer nur ein Projektfenster gleichzeitig bearbeitet werden.

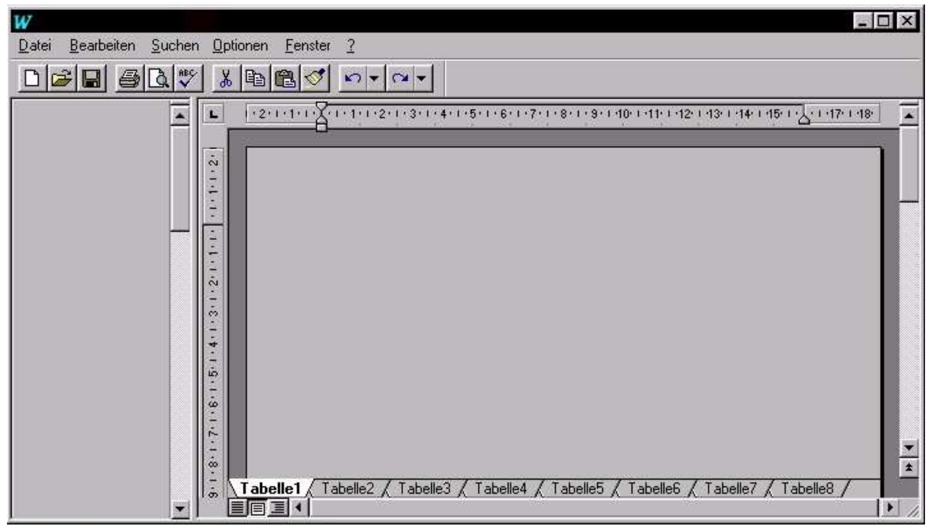


Abbildung 2.3: Zusammengefaßte Komponenten

Vorteile:

- klare Aufteilung der Komponenten innerhalb des Fensters

Nachteile:

- nur 'ein' Projektfenster zur Zeit sichtbar (kein Drag-and-Drop zwischen mehreren Projekten)
- keine freie Positionierung der Komponenten

Zusammenfassung

Die Anordnung mit den meisten Vorteilen sind vollständig getrennte Komponenten. Daher wird auch der deSIGN Editor diese Vorteile benutzen. Die Komposition der verschiedenen Fenster läßt dem Benutzer hierbei den größtmöglichen Spielraum für seine eigene, bevorzugte Anordnung. Allerdings sollte man bei dieser Anordnungsvariante davon ausgehen, daß eine ausreichend große Auflösung zur Verfügung steht.

2.2.2.2 Dialoge

Im deSIGN-System unterscheiden wir zwischen vier verschiedenen Arten von Dialogen, die hier auch einzeln betrachtet werden sollen:

Informationsdialoge Ein Informationsdialog besteht im wesentlichen lediglich aus einem Text, der dem Benutzer angezeigt werden soll, und einem Bestätigungsbutton (meistens 'OK'). Diese Art der Dialoge werden für Mitteilungen wie z.B. Fehler bei der Eingabe, Übersetzungen oder anderweitige Informationen benutzt.

Optionsdialoge Bei den Optionsdialogen hat der Benutzer verschiedene Markierungsmöglichkeiten sowie einen Bestätigungs- und einen Abbruchbutton. Die Markierungsmöglichkeiten sind vielfältig und mit ihnen lassen sich alle Arten von Optionen einstellen (z.B. Optionen für jedes Objekt des Editors, Grafikdetails an-/ausschalten, Zoomfaktor einstellen, Übersetzungsoptionen einstellen, etc.).

Abfragedialoge Aus einer Art der Erweiterung des Informationsdialogs entsteht der Abfragedialog. Auch in diesem Dialog wird ein Text angezeigt, dem Benutzer stehen allerdings zwei Buttons zur Antwortmöglichkeit zur Verfügung: 'Ja' und 'Nein'. Durch diesen Dialog lassen sich - wenn auch in geringem Umfang - Einstellungen durch den Benutzer durchführen.

Spezielle Dialoge Unter diese Kategorie fallen alle vom System verwalteten Dialoge (Datei- und Druckdialoge) sowie alle Sonderfälle, für die keine generellen Dialoge vorhanden ist.

2.3 OMT Vorträge

Im Rahmen der Sprachfindung wurde von uns unter anderem OMT untersucht. Dieses Kapitel stellt die verschiedenen Modelle (Objektmodell, funktionales und dynamisches Modell) vor und beleuchtet die jeweiligen Stärken und Schwächen.

2.3.1 Objektmodell

2.3.1.1 Einleitung (zum OMT-Buch)

Dieses Buch [RBP+91] präsentiert eine objekt-orientierte Software-Entwicklungsmethodologie, die Object Modeling Technique (OMT) genannt wird . Darin wird eine graphische Notation zur Beschreibung des objekt-orientierten Modells benutzt.

Was ist Objekt-Orientierung ?

Oberflächlich betrachtet, bedeutet objekt-orientiert, daß wir eine Software durch eine Sammlung von Objekten organisieren (gruppieren), die sowohl mit Datenstrukturen als auch mit dem Verhalten zusammenarbeiten.

Das ist der Gegensatz zur konventionellen Programmierung, bei der Datenstruktur und Verhalten keine feste Bindung haben.

Welche Charakteristik für eine objekt-orientierte Annäherung genau benötigt wird, ist noch streitig, aber generell wird sie vier Aspekte beinhalten:

- Identität
- Klassifikation
- Polymorphismus
- Vererbung

Was ist Objekt-Orientierte Entwicklung ?

Das ist eine neue Denkmethode für software-basierte Abstraktionen, die in der realen Welt existieren. In diesem Kontext verweist Entwicklung auf den vorderen Teil des Lebenszyklus einer Software:

- Analyse
- Design
- Implementation

Der Wesen von objekt-orientierter Entwicklung ist die Identifikation und Organisation von Konzepte einer Applikation, und weniger ihre letztliche Präsentation in einer Programmiersprache, objekt-orientiert oder nicht.

Eine objekt-orientierte Entwicklungsannäherung ermutigt die Softwareentwickler, während des Lebenszyklus einer Software-Engineering über den Applikationsbereich zu denken und zu arbeiten. Das geschieht nur, wenn die Konzepte einer Applikation identifiziert, organisiert und verstanden werden, so daß die Details einer Datenstruktur und die Funktionen effektiver angesprochen werden können.

Objekt-orientierte Entwicklung ist ein konzeptioneller Prozeß, unabhängig von einer Programmiersprache. Objekt-orientierte Entwicklung ist grundsätzlich eine neue Denkweise und keine Programmiertechnik. Diesen Vorteil können die Entwickler, Fachleute und die Kunden bei klarer Beschreibung der abstrakte Konzepte und deren Kommunikation miteinander nutzen.

Objekt-Orientierte Methodologie

Diese Methodologie besteht aus der Zusammensetzung des Modells eines Applikatiosbereiches und des Zufügens von Impementationsdetails, während das System konstruiert wird.

Diese Annäherung heißt OMT. Die Methodologie hat folgende Stadien:

- ANALYSE: Die Frage ist hier was muß ein erwünschtes System tun, und nicht wie muß es getan werden.
- SYSTEMDESIGN: Während der Systemkonstruktion wird das Zielsystem in Subsysteme gruppiert, die sowohl auf Analysestruktur als auch auf vorgeschlagene Architektur basiert sind.
- OBJEKTDESIGN: Der Objektdesigner bildet ein kostrukturiertes Modell, das wiederum auf Analysemodell basiert ist, aber die Implemetationdetails beinhaltet.
- IMPLEMENTATION: Die Objektklassen und entwickelte Beziehungen werden während der Objektkonstruktion letztendlich in eine Programmiersprache übersetzt

2.3.1.2 Modellierung als technisches Design

Der Begriff „Modell“

Eine Abstraktion von irgend etwas, mit dem Ziel dieses Etwas zu verstehen, bevor man mit seinem Bau beginnt.

Ein Modell läßt die unbrauchbaren Details aus.

Es ist einfacher, mit einem Modell zu arbeiten, als mit dem Original.

Modellierung

Die Designer bauen verschiedene Modelle für unterschiedliche Zwecke, bevor sie mit der eigentlichen Konstruktion anfangen, zum Beispiel, Flugzeugmodelle für Windtunneltests, Entwurf von Maschinenteilen usw.

Modelle dienen mehreren Zwecken:

- Das Testen eines physikalischen Objektes, bevor man es baut
- Kommunikation mit Kunden
- Visualisierung
- Reduzierung von Komplexitäten (vielleicht der Hauptgrund)

Abstraktion

Abstraktion ist eine selektierte Untersuchung bestimmter Aspekte eines Problems. Das Ziel der Abstraktion ist, diejenigen Aspekte, die wichtig für einige Zwecke sind, von den unwichtigen zu trennen.

Der Abstraktionszweck muß eindeutig sein, weil nämlich der Zweck bestimmt, was wichtig und was unerheblich ist. Viele verschiedene Abstraktionen kommen für eine Sache in Betracht. Dies hängt davon ab, für welche Zwecke sie gebaut werden. Abstraktionen sind ungenau und nicht komplett. Beim Bauen eines Modells muß man nicht nach der absoluten Wahrheit suchen, sondern Angemessenheit dieser Zwecke berücksichtigen. Es gibt kein einziges „korrektes“ Modell für eine Situation, sondern nur angemessene und unangemessene Modelle.

Object Modeling Technique (OMT)

Das Wort „Modell“ hat zwei Bedeutungen : Eine von der Sicht des Systems (Object Model, Dynamic Model oder Functional Model) gesehen und die andere vom Entwicklungsstadium (Analyse, Design oder Implementieren) aus betrachtet.

- OBJECT MODEL

Das Objektmodell beschreibt die Struktur von Objekten in einem System, ihre Identität, Beziehungen zu anderen Objekten, ihre Eigenschaft und Operationen. Es schafft den notwendigen Rahmen, in dem die dynamischen und funktionalen Modelle plaziert werden können. Objekte sind die Einheiten, in die wir die Welt einteilen, also die Moleküle unseres Modells. Das Ziel der Konstruktion eines Objektmodells besteht darin, daß man die Konzepte, die für eine Applikation wichtig sind , aus der realen Welt herausnimmt. Das Objektmodell wird durch graphische Darstellung, die die Objektklassen beinhaltet, repräsentiert. Klassen sind hierarchisch strukturiert und teilen die ganze Struktur und das Verhalten und sind ferner mit anderen Klassen verbunden. Klassen definieren die Werteattribute, die von jedem Objekt getragen werden und Operationen, die von jedem Objekt ausgeführt oder ausgelöst werden.

- DYNAMIC MODEL

Das dynamische Modell beschreibt die Aspekte eines Systems, die sich auf Zeit und Reihenfolge von Operationen konzentriert. Zum Beispiel: Ereignisse, die Ziele ändern ; Reihenfolge von Ereignissen ; Aussagen, die den Kontext von Ereignissen definieren ; die Organisation von Ereignissen und Aussagen. Das dynamische Modell wird durch eine graphische Darstellung, und zwar mit Zustandsdiagrammen, repräsentiert. Jedes Zustandsdiagramm zeigt die Zustands- und die Ereignisreihenfolge, die in einem System für eine Klasse von Objekten erlaubt ist. Zustandsdiagramme verweisen auch auf andere Modelle. Aktionen in einem Zustandsdiagramm korrespondieren mit Funktionen in einem funktionalen Modell. Ereignisse in einem Zustandsdiagramm stehen für Operationen von Objekten durch das Objektmodell.

- FUNCTIONAL MODEL

Das funktionales Modell beschreibt die Aspekte eines Systems, die sich auf Transformation von Werten konzentrieren. Dieses Modell faßt zusammen, was ein System tut, ohne Rücksicht auf das Wie und Wann zu nehmen. Dies wird durch ein Datenflußdiagramm repräsentiert. Datenflußdiagramme zeigen die Abhängigkeit zwischen Werten und die Berechnung von Ausgabewerten durch Eingabewerte und Funktionen, ohne Rücksicht darauf, ob und wann die Funktionen ausgeführt werden.

Beziehungen zwischen Modellen

Jedes Modell beschreibt einen Aspekt eines Systems, verweise jedoch auf andere Modelle. Das Objektmodell beschreibt die Datenstruktur, auf welche das dynamische und funktionale Modell, operieren können. Die Operationen in einem Objektmodell korrespondieren mit Ereignissen im dynamischen Modell und mit Funktionen im funktionalen Modell.

Das dynamische Modell beschreibt die Kontrollstruktur von Objekten. Es zeigt die Entscheidungen, die von Objektwerten abhängig sind und Aktionen verursachen, die Objektwerte ändern und Funktionen aufrufen.

Das funktionale Modell beschreibt die von Operationen ins Objektmodell aufgerufenen Funktionen und die ins dynamische Modell aufgerufenen Aktionen. Funktionen operieren auf vom Objektmodell spezifizierten Datenwerten. Das zeigt also die Begrenztheit von Objektwerten.

Es kann manchmal nicht festgestellt werden, welche Modelle welche Informationen beinhalten sollen. Dies ist natürlich, da jede Abstraktion nur ein grober Schnitt von der Realität ist. Gelegentlich werden die Grenzen unvermeidbar überschritten. Es ist auch normal, daß manche Eigenschaften von einem System möglicherweise durch Modelle schwach repräsentiert werden, da es keine Abstraktion gibt, die alles auffaßt. Das Ziel besteht darin die Beschreibung des Systems zu vereinfachen, ohne das Modell mit vielen Konstruktionen zu belasten, so daß es dann nicht behilflich ist. Für solche Dinge, die ein Modell nicht hinreichend auffassen kann, ist eine natürliche Sprache oder eine spezifische Notation noch ein akzeptables Werkzeug.

2.3.1.3 OMT: Das Objektmodell

Das Objektmodell stellt die Datensicht auf ein System dar. Es beschreibt also die statische Struktur eines Systems durch Darstellen von Objekten im System, Beziehungen zwischen Objekten, Attribute und Operationen, die jede Klasse von Objekten auszeichnen.

Das Objektmodell wird auch als das wichtigste unter den drei OMT-Modellen (Objektmodell, Dynamische-Modell und Funktionale-Modelle) angesehen. Als Grund hierfür dient in erster Linie die Tatsache, daß das Objektmodell in der Analysephase (erste Phase) der Modellierung eingesetzt wird, und somit alle folgenden Konzepte auf der Basis dieser Phase aufbauen.

Zum anderen stellt das Objektmodell einen Abriß der realen Welt dar (entspricht der realen Welt), und stellt außerdem eine grafische Notation eines Systems bereit, die wichtig für die Kommunikation mit Kunden ist.

2.3.1.4 Objekte

Der Zweck der Objektmodellierung ist das Beschreiben von Objekten. Beispiele hierfür sind Heinrich Meier , PID 7648 usw. Ein Objekt wird als ein Konzept, eine Abstraktion oder als ein Ding angesehen. Objekte dienen zwei Zwecken: zum einen fördern sie das Verstehen der realen Welt, zum anderen liefern sie eine praktische Grundlage für Programm-Implementierungen.

Alle Objekte haben eine Identität (Herkunft) und sind Unterscheidbar. Ihre Unterscheidbarkeit liegt aber nicht in deren beschreibbaren Eigenschaften, die sie haben, sondern vielmehr in ihrem Dasein (Existenz). Beispiele hierfür sind zwei Äpfel mit der gleichen Farbe, der gleichen Form usw. zwei individuell verschiedene Äpfel, weil eine Person erst den einen Apfel, dann den anderen essen kann.

Der Objektbegriff wird in der Literatur meist undeutig benutzt. Manchmal meint man mit „Objekt“ eine einzige Sache, manchmal weist es auf eine Gruppe von ähnlichen Dingen hin.

Nur der Zusammenhang (Kontext) läßt auf eine klare Semantik deuten. Daher wird bei einem Verweis auf exakt eine Sache der Begriff Objektinstanz verwendet. Bei einem Verweis auf eine Gruppe von ähnlichen Dingen sprechen wir dann von Objektklassen.

2.3.1.5 Klassen

Eine Objektklasse beschreibt eine Gruppe von Objekten mit ähnlichen Eigenschaften (gleichen Attributen), gleichem Verhalten und gleichen Beziehungen zu anderen Objekten. „Person“, „Tier“, „Prozeß“ sind alles Objektklassen. Jede Person hat ein Alter, IQ und viele haben einen Arbeitsplatz. Jeder Prozeß hat einen Eigentümer, Priorität usw. Der Begriff Klasse wird häufig anstelle des Objektklassen-Begriffs benutzt. Objekte in einer Klasse haben die gleichen Attribute und Verhaltensmuster. Die meisten Objekte gewinnen ihre Individualität durch Unterschiede in ihren Attributwerten und Beziehungen zu anderen

Objekten. Jedes Objekt kennt seine Klasse. Durch Zusammenfassen von Objekten in Klassen abstrahieren wir ein Problem.

2.3.1.6 Objekt Diagramme

Die Klassen- und Objektkonzepte haben wir bis dato mit Beispielen beschrieben. Weil diese Annäherung für mehrere komplexe Themen unklar ist, benötigen wir einen Formalismus, der verständlich, präzise und einfach zu formulieren ist, um Objekt-Modelle auszudrücken.

Dafür gibt es Objekt-Diagramme, die eine grafische Notation für die Modellierung von Objekten, Klassen und ihren Beziehungen zueinander liefern. Objekt-Diagramme sind also einfach zu verstehen und arbeiten gut in der Praxis (in der OMT von Rumbaugh werden sie über das ganze Buch hinweg eingesetzt).

Man unterscheidet zwei Typen von Objektdiagrammen :

- Klassendiagramme (beschreiben den allgemeinen Fall , d.h Klassen)
- Instanzendiagramme (beschreiben die Ausprägungen, Beispiele, d.h Objekte)

Es ist nicht sinnvoll Klassen und Instanzen in dem gleichen Objekt-Diagramm unterzubringen, obwohl das möglich ist.

2.3.1.7 Attribute

Ein Attribut ist ein Datenelement, das von Objekten in einer Klasse gehalten wird. Beispiele sind Name, Alter, Gewicht usw. Attribute von Personenobjekten. Jedes Attribut hat einen Wert für jede Instanz, z.B. hat Alter den Wert 24 im Objekt Joe Smith.

Verschiedene Instanzen können gleiche oder verschiedene Werte für ein gegebenes Attribut haben. Attribute sollen pure Datenelemente sein, d.h sie sollen keine Identität haben und somit keine Objekte sein.

2.3.1.8 Operationen und Methoden

Eine Operation ist eine Funktion, die von Objekten in einer Klasse benutzt werden kann. Die gleiche Operation kann auf mehrere verschiedene Klassen übertragen werden. Dabei spricht man von polymorphen Operationen(die gleiche Operation tritt in verschiedenen Klassen in verschiedenen Gestalten auf). Eine Methode ist die Implementierung einer Operation für eine Klasse. Z.B. hat die Klasse „File“ eine Operation „print“. Es können verschiedene Methoden implementiert werden, um ASCII-Dateien, digitalisierte Bilderdateien usw. auszudrucken. Diese Methoden würden alle die gleiche Aufgabe ausführen, das Drucken einer Datei; somit kann man auf sie durch die Operation „print“ verweisen. Jedoch kann jede Methode verschieden implementiert werden.

2.3.1.9 Zusammenfassung der Klassennotation

Eine Klasse wird durch eine Box darsgestellt und in drei Bereiche unterteilt. Von oben nach unten betrachtet beinhaltet diese: Klassennamen, Liste von Attributen und Liste von Operationen. Im zweiten Bereich (Attribute) wird der Attributname vom Datentyp durch einen Doppelpunkt getrennt, gefolgt (wenn vorhanden) durch ein Anfangswert (ebenfalls durch einen Doppelpunkt getrennt). In dem dritten Bereich (Operationen) folgt dem Operationsnamen, wenn vorhanden eine Liste von Argumenten, die durch runde Klammern eingeschlossen wird. Durch einen Doppelpunkt getrennt folgt dann das Ergebnistyp der Operation. Abbildung 2.4 verdeutlicht dies noch einmal visuell.

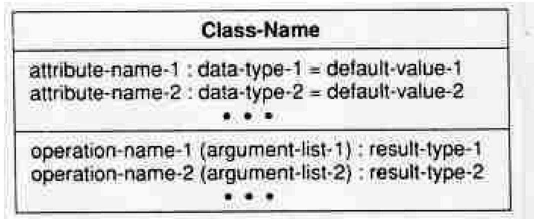


Figure 3.5 Summary of object modeling notation for classes

Abbildung 2.4: Klasse

2.3.1.10 Verbindungen und Assoziationen

Links(Verbindungen) und Assoziationen (Beziehungen) sind Konzepte, um Beziehungen zwischen Objekten und Klassen zu gründen.

Eine Link ist eine physikalische oder konzeptuelle Verbindung zwischen Objektinstanzen.

Mathematisch gesehen ist ein Link als ein Tupel definiert, das eine geordnete Liste von Objektinstanzen ist.

Eine Assoziation(Beziehung) beschreibt eine Gruppe von Links mit gleicher Struktur und gleicher Bedeutung. Alle Links in einer Assoziation verbinden Objekte von der gleichen Klasse.

Assoziationen und Links erscheinen häufig als „Verben“ in einer Problemaussage. Eine Assoziation beschreibt eine Menge von potentiellen Links, in der gleichen Weise, wie Klassen eine Menge von Objekten beschreiben. Assoziationen sind bidirektional.

Assoziation werden in Programmiersprachen häufig als Zeiger (Pointer) von einem Objekt zum anderen implementiert. Ein Zeiger ist ein Attribut in einem Objekt, das eine Referenz auf ein anderes Objekt beinhaltet.

In Abbildung 2.5 wird eine einfache 1:1 Beziehung (one-to-one) zwischen Klassen in einem Klassendiagramm dargestellt. Außerdem werden dazu die entsprechenden Verbindungen (Links) in einem Instanzendiagramm aufgezeigt.

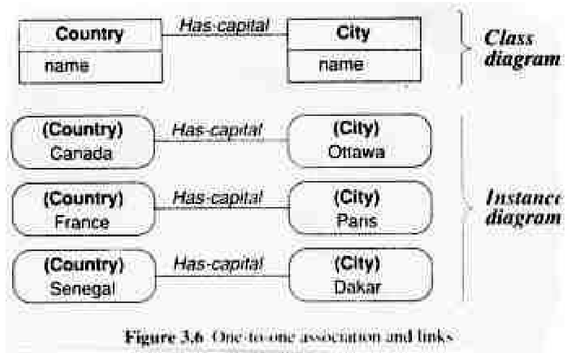


Abbildung 2.5: 1:1-Assoziation

Die Assoziation besagt, daß jedes Land eine Hauptstadt hat, das Instanzendiagramm stellt dann konkrete Ausprägungen (= Links) dieser Assoziation dar.

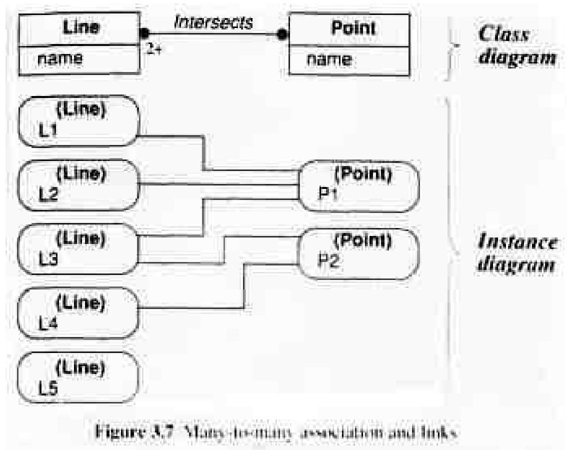


Abbildung 2.6: m:n-Assoziation

Abbildung 2.6 zeigt eine mehr-zu-mehr (many-to-many) Assoziation, in der zwei Klassen „Linie“ und „Punkt“ miteinander assoziieren durch ihre „Schnittpunkte“. Der Sachverhalt stellt dar, daß jede Instanz der Klasse Linie einen Schnittpunkt zu keinem oder mehreren Instanzen der Klasse Punkt haben kann. Umgekehrt wird jeder Punkt durch zwei oder mehreren Instanzen der Klasse Linie geschnitten.

Kardinalitäten (Multiplicity)

Kardinalitäten geben an, wieviele Instanzen einer Klasse Beziehung zu einer Instanz einer anderen Klasse haben können. Kardinalitäten werden mit einem einfachen Intervall am Ende der Assoziationslinie angegeben.

Die Bedeutungen von Kardinalitäten erläutert Abbildung 2.7:

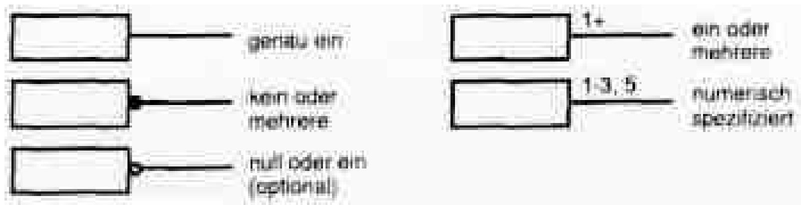


Abbildung 2.7: Kardinalität

Assoziationsattribute

Ein Attribut ist eine Eigenschaft von Objekten in einer Klasse. Ähnlich dazu ist ein Assoziationsattribut eine Eigenschaft von Links in einer Asssoziation. Jedes Assoziationsattribut hat einen Wert für jeden Link.

Ein Assoziationsattribut wird durch eine Box, die durch eine Schleife an die Assoziation gebunden ist, dargestellt. Mehrere Assoziationsattribute können in dem zweiten Bereich der Box erscheinen.

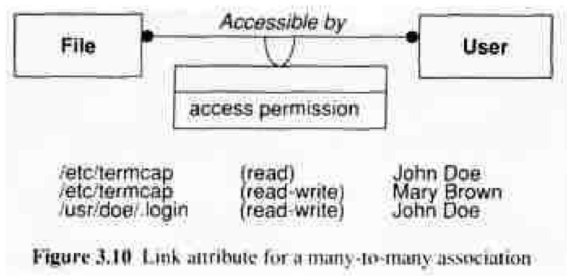


Abbildung 2.8: Assoziationsattribut

In Abbildung 2.8 wird ein Assoziationsattribut für eine many-to-many Assoziation gezeigt.

Hierbei haben wir zwei Klassen „File“ und „User“, die miteinander, aufgrund der Zugangsmöglichkeiten auf Dateien, in Beziehung stehen. Die Zugangsmöglichkeiten können durch ein Attribut Zugriffsrechte von „Accessible by“ eingeschränkt werden.

Rollennamen Das Ende einer Assoziation wird Rolle genannt. Eine binäre Assoziation hat somit zwei Rollen. Jedes davon kann einen Rollennamen haben. Ein Rollenname ist ein Name zum Bezeichnen eines Endes einer Assoziation. Jede Rolle einer binären Assoziation bezeichnet ein Objekt oder eine Menge von Objekten, das (die) mit einem anderen Objekt auf der anderen Seite in Beziehung steht (stehen). Rollennamen sind nötig für Assoziationen zwischen zwei Objekten der gleichen Klasse. Rollennamen sind somit wichtig, um zwei Assoziationen zwischen dem gleichen Paar von Klassen zu unterscheiden.

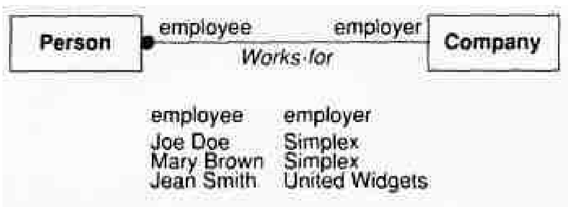


Figure 3.15 Role names for an association

Abbildung 2.9: Rollenname

Abbildung 2.9 zeigt die Verwendung von Rollennamen für Assoziationen an. Hierbei wird dargestellt, wie „Person“ und „Company“ über „Works“ for in einer Assoziation stehen.

Eine Person nimmt dadurch die Rolle eines Angestellten an und eine Company wird in die Rolle des Arbeitgebers versetzt, wobei eine Person für nur eine Company arbeitet, eine Company aber mehrere Personen einstellen kann.

Geordnete Assoziationen

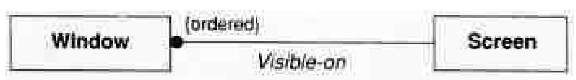


Figure 3.17 Ordered sets in an association

Abbildung 2.10: Geordnete Assoziation

In Abbildung 2.10 wird die Sichtbarkeit von Fenstern (Windows) auf einer Bildschirmfläche (Screen) geordnet. Hierbei wird durch„{ordered}“ ausgedrückt, daß eine Bildschirmfläche mehrere Fenster haben kann, aber nur ein Fenster in einem bestimmten Punkt der Bildschirmfläche sichtbar ist, die anderen Fenster dann verdeckt sind.

Qualifizierte Assoziationen Ein Qualifikator ist ein spezielles Attribut, das die Kardinalität einer 1:n bzw. m:n Assoziation reduziert. Eine Qulifizierte Assoziation bezieht zwei Klassen und einen Qualifikator ein.

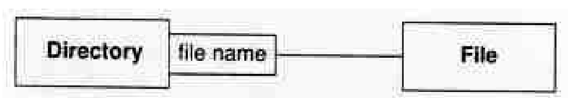


Figure 3.18 A qualified association

Abbildung 2.11: Qualifizierte Assoziation

In Abbildung 2.11 wird eine "qualifizierte Assoziation" dargestellt. Da Qualifikatoren die Kardinalität einer Assoziation reduzieren, erhalten wir in diesem

Beispiel aus einer one-to-many Assoziation eine one-to-one Assoziation. D.h im Grunde kann ein Verzeichnis mehrere Dateien haben. Aber durch die Angabe eines Qualifikators „file name“ am Ende der Assoziation wird daraus der Sachverhalt, daß ein Verzeichnis nur eine Datei mit einem bestimmten Namen haben kann.

2.3.1.11 Aggregation

Eine Aggregation ist eine gekoppelte Form von Assoziationen mit einigen extra Bedingungen. Die bedeutenste Eigenschaft der Aggregation ist die Transivität, die besagt: wenn A ein Teil von B ist, und B ein Teil von C ist , dann ist A auch ein Teil von C. Ferner gilt die Eigenschaft der Antisymmetrie, die besagt: wenn A ein Teil von B ist, dann ist B kein teil von A.

Ein Aggregation beschreibt also eine „Teil-Ganzes-Beziehung“, wobei ein einzelnes Objekt nur in einer direkten Aggregation partizipiert werden kann und „parallele“ Aggregationen zusammengefaßt werden, um Hierarchieebenen zu betonen.

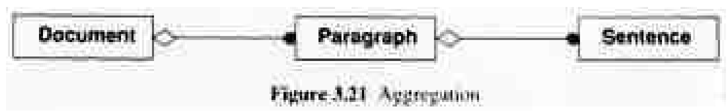


Abbildung 2.12: Aggregation

In Abbildung 2.12 wird eine Aggregation von Objekten dargestellt. Ein Satz ist ein Teil eines Paragraphen. Ein Paragraph wiederum ist ein Teil eines Dokuments. Es gilt jedoch nicht die Umkehrung, daß ein Dokument z.B. ein Paragraph ist (Antisymmetrie).

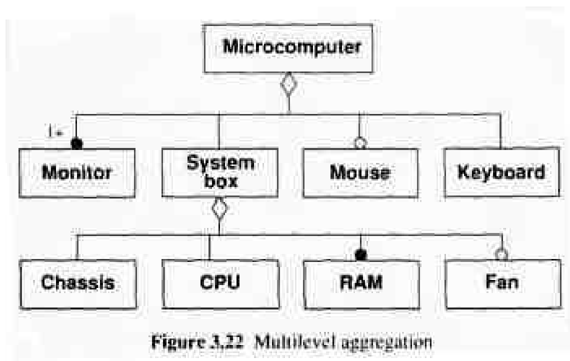


Abbildung 2.13: Multilevel Aggregation

Abbildung 2.13 zeigt im Gegensatz dazu eine Hierarchieebene, in der „parallele“ Aggregationen zusammengefaßt sind (Multilevel Aggregation). Ein Computersystem besteht aus mehreren verschiedenen Komponenten, wie Monitor, Systemeinheit, Maus, Tastatur usw. die alle Teil des Computersystems sind. Diese

Teile können wiederum weitere Komponenten haben, wie CPU, RAM usw. einer Systemeinheit. Somit wird eine Hierarchie aufgebaut, in der die Komponenten, die in der Hierarchieebene unten liegen Teile von denen sind, die über ihnen sind (falls sie denn miteinander aggregieren, z.B. ist die CPU kein Teil von einem Monitor).

2.3.1.12 Generalisierung (Vererbung)

Eine Generalisierung stellt eine Beziehung zwischen einer Klasse und ihren verfeinerten Versionen dar. Es geht hier also um die Vererbung von Attributen, Methoden und Assoziationen, wobei die zu vererbende Klasse als„Superklasse", und die vererbte Klasse als „Subklasse" betrachtet wird. Jede Klasse erbt die Merkmale ihrer Superklasse. Man spricht hierbei auch von disjunkter Spezialisierung. Die Generalisierung wird manchmal auch „is-a" Beziehung genannt, weil jede Instanz einer Subklasse eine Instanz einer Superklasse ist.

Jede Subklasse erbt aber nicht nur alle Merkmale seines Vorfahren. Es kann auch seine eigenen Attribute und Operationen haben.

Die Vererbung wird in objektorientierten Programmiersprachen streng unterstützt. Wichtigstes Nutzen der Vererbung ist die konzeptuelle Vereinfachung durch Reduzierung der Anzahl der abhängigen Merkmale eines Systems.

Die Vererbung bietet auch die Möglichkeit zum Überschreiben von ererbten Attributen und Methoden. Eine Subklasse kann ein Merkmal einer Superklasse überschreiben, indem es ein Merkmal mit dem gleichen Namen definiert. Die Subklassen-Eigenschaft verfeinert somit die überschriebene Eigenschaft der Superklasse. Es gibt normalerweise zwei Gründe, warum man ein Merkmal überschreiben sollte:

- Um die Definition des Originalmerkmals völlig zu ersetzen
- Um (bei Operationen) die Originalmethode um ein zusätzliches Verhalten zu erweitern

Es sei noch erwähnt, daß es außer der disjunkten Spezialisierung auch eine nicht-disjunkte Spezialisierung gibt, die auch Mehrfachvererbung genannt wird. Hierbei kann jede Subklasse zwei oder mehrere Superklassen haben.

Die Vererbung wird durch ein leeres Dreieck zwischen einer Superklasse und ihren Subklassen dargestellt.

Die Mehrfachvererbung wird durch ein leeres Dreieck zwischen den Superklassen und ihren Subklassen dargestellt.

Das Schema in Abbildung 2.14 zeigt eine Vererbungshierarchie auf, in der alle abgeleiteten Klassen die Merkmale einer ihr höherstehenden Klasse erben. In dem Beispiel werden abgeleitete Klassen (Subklassen) Kreis, Rechteck und Polygon dargestellt, die von einer gemeinsamen Basisklasse erben, wobei die Basisklasse (Superklasse) abstrakt ist. Dadurch wird die Superklasse in den Subklassen verfeinert = disjunkte Spezialisierung.

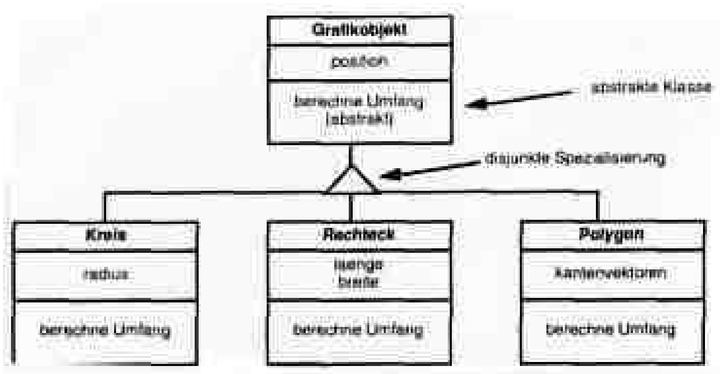


Abbildung 2.14: Vererbung

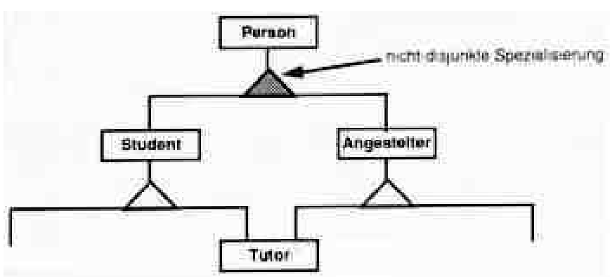


Abbildung 2.15: Mehrfachvererbung

Abbildung 2.15 zeigt eine Mehrfachvererbung, in eine Subklasse zwei oder mehrere Vorfahren (Superklassen) hat. Im Beispiel erbt ein Tutor die Merkmale eines Studenten und eines Angestellten. Diese Form der Vererbung wird deshalb auch „nicht-disjunkte Spezialiserung" genannt.

2.3.1.13 Module

Module sind Konstrukte, um Klassen, Assoziationen und Generalisierungen zu gruppieren. Sie dienen dazu, ein Objekt-Modell in handliche Stücke zu spalten. Ein Modul behandelt eine Sicht einer Situation. Ein Objekt-Modell beinhaltet ein oder mehrere Module.

2.3.1.14 Arbeitsblätter (Sheets)

Ein Sheet ist ein Mechanismus, um ein großes Objekt-Modell in Folgen von Seiten zu verteilen. Sheets sind ausgedruckbare Seiten, und sind daher nur notationeller Komfort, kein logisches Gebilde. Jedes Modul beinhaltet ein oder mehre Sheets. Als Regel sollten nicht mehr als ein Modul auf eine Seite sein.

Jedes Sheet hat einen Titel und einen Namen oder eine Nummer.

Jede Assoziation und Generalisierung erscheint auf einem einzelnen Sheet. Klassen können auf mehreren Sheets erscheinen.

Das Objektmodell vom Windowing-System Abbildung 2.16 zeigt ein Objektmodell von einer Workstation mit dem Window-Mangement-System, wie X Window-System oder SunView. Diese Modell ist sehr vereinfacht - ein wirkliches Modell eines Windowing-Systems würde mehrere Seite benötigen - aber es veranschaulicht viele Konstruktionen einer Objektmodellierung und zeigt, wie sie zu einem großen Modell zusammengesetzt werden.

Die Klasse „Window" definiert gemeinsame Parameter jeder Art von Windows, inbegriffen einer rechteckigen Eingrenzung durch die Attribute x1, y1, x2, y2 und Operationen „to display" und „undisplay" eines Fensters und „to raise" Hervorhebung nach Vorne (Vordergrund) oder „lower" Herunterdrücken nach hinten (Hintergrund) von einer ganzen Menge von Fenstern. „Panel", „Canvas" und „Textwindow" sind verschiedene Fenster. Ein Canvas ist ein Bereich einer Graphikzeichnung. Canvas erbt die Window-Grenze von Window und fügt die Dimensionen von dem zugrundeliegenden Canvas-Bereich ein, der durch Attribute cx1, cy1, cx2, cy2, definiert ist. Ein Canvas beinhaltet ein Menge von Elementen, gezeigt durch Assoziation zur Klasse „Shape". Alle „Shapes" (Formen) haben Farbe und Linienbreite. „Shapes" können Lineare, Ellipse oder Polygonen mit ihrer eigenen Parameter sein. Ellipse und Polygonen sind beide geschlossene Formen, welche eine Farbfülle und eine Musterfülle haben. Die Lineare sind eindimensional und können nicht gefüllt werden. Canvas-Fernster haben Operationen „add elements" und „delete elements". Text window ist eine Form von „Scrolling window" (Schriftroller), die ein zweidimensionales Schriftroll-offset innerhalb ihres Fensters hat, die durch „x-offset" und „y-offset" gekennzeichnet ist, genau wie eine Operation „scroll" zur Änderung der Schriftroll-Werte.

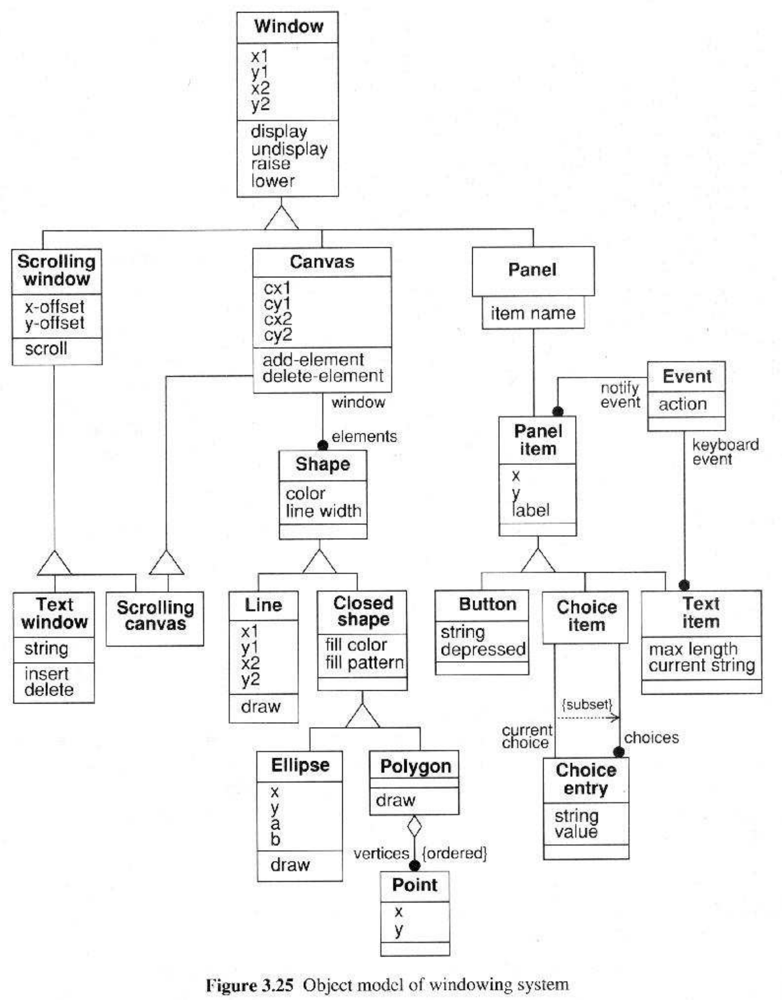


Abbildung 2.16: Das Objektmodell vom Windowing System

Ein Text-Fenster beinhaltet ein „string" und hat Operationen zum Einfügen und Löschen von den Charakteren. „Scrolling canvas" ist eine spezielle Form von Canvas, die Schriftrollen unterstützt; es ist ein Canvas und ein Schriftroll-Fenster zugleich. Dies ist ein Beispiel von Mehrfachvererbung.

Panel beinhaltet eine Menge von „Panel item" Objekten. Identifiziert wird jedes „Panel item" durch einen bestimmten „item name" innerhalb vom gegebenen Panel. Das wird durch „qualified association" gezeigt. Jedes „Panel item" gehört zu einem einzigen Panel. Ein Panel item ist ein vordefiniertes Icon, mit der ein Benutzer auf dem Bildschirm interagieren kann. Panel item kommen in drei Sorten vor : „Buttons" (Knöpfe), Auswahl-Item und Text-Item. Zu einem Knopf gehört ein String (Zeichenkette), der auf dem Bildschirm erscheint; der Knopf kann von einem Benutzer gedrückt werden. Er hat eine Attribute „depressed" (nichtgedrückt). Ein Auswahl-Item erlaubt dem Benutzer, der vordefinierten Mengen von Auswahlen auszusuchen, wobei jede von ihnen einen Auswahleintrag hat, der wiederum einen String zur Anschauung beinhaltet und einen Wert zurückgibt, falls der Eintrag ausgewählt wurde.

Es gibt zwei Assoziationen zwischen Auswahl-Item und Auswahl-Eintrag, eine eins-zu-mehrere Assoziation, die die Menge von erlaubten Auswahlen definiert, während die eins-zu-eins Assoziation die gängigen Auswahlen definiert. Die gängige Auswahl muß eine von erlaubten Auswahlen sein, so daß eine Assoziation eine Untermenge von anderen Auswahl ist, die durch einen Pfeil zwischen den beiden versehen ist. Damit wird die Begrenztheit von Auswahlen deutlich.

Wenn ein Panel-Item vom Benutzer ausgewählt wird, wird ein ein Ereignis in Gang gesetzt, welches ein Zeichen dafür ist, daß das Panel aktiviert wurde. Alle Sorten von Panel-Items haben eine „notify event" (Benachrichtigungsereignis) Assoziation. Jedes „Panel-Item" hat ein einziges Ereignis, aber ein Ereignis kann durch viele „Panel-Item" geteilt werden. „Text-Items" haben eine zweite Sorte von Ereignissen, welche dann geschehen, wenn eine Buchstabentaste getippt wird, während das „Text-Item" gewählt wurde. Die Assoziation „keyboard event" zeigt die Ereignisse. Also erbt „Text-Item" „notify event" von der Superklasse „Panel-Item" ; „notify event" wird erzeugt, wenn das ganze „Text-Item" mit einer Maus gewählt wurde.

In einem Modell gibt es viele Unterschiede, die zum Beispiel darin bestehen, ob wir vielleicht ein Typ „Rectangle" definieren sollen, welche für die Window- und Canvas-Eingrenzungen und nicht etwa für zwei ähnliche Gruppen von vier Attributspositionen benutzt werden kann. Ein Linear könnte ein spezialisierter Zustand von Polylinearen (eine verbundene Serie von Linearenstücke) sein, deren Form sowohl Polylineare als auch Polygon sein könnte. Diese beiden würden dann als Unterklasse der neuen gemeinsamen Superklasse werden, die eine geordnete Liste von Punkten definiert. Viele Attributen, Operationen und Klassen fehlen in der Beschreibung eines realistischen Windowing-Systems. Sicherlich haben die Fenster Assoziationen in sich, solche wie das Überlappen von einander. Wie auch immer gibt dieses einfache Modell ein Vorbild der Anwendung einer Objektmodellierung. Wir können ihre Details kritisieren, weil sie präzise Aussagen hat. Sie würde als die Basis eines vollständigeren Modells dienen.

2.3.2 Erweitertes Objektmodell

2.3.2.1 Einleitung

Dies ist der zweite Beitrag zur Vorstellung der Modellierungskonzepte von OMT. Es behandelt das Kapitel 4: „Advanced Object Modeling“ aus dem Buch „Object-Oriented Modeling and Design“. Als Fortsetzung der im Referat über Kapitel 3 des gleichen Buches, „Object Modeling“ vorgestellten Modellierungskonzepte dient es der vertiefenden Behandlung spezieller Aspekte und der Vorstellung einiger weitergehender Konzepte.

2.3.2.2 Aggregation

Die Aggregation ist eine stärkere Form der Assoziation. Das aggregierte Objekt wird aus Komponenten gebildet. Die Notation mit dem Diamant-Symbol wurde bereits in Kapitel 3 vorgestellt. Die Aggregationsbeziehung wird Teil-Ganzes-Beziehung, Teil-von-Beziehung oder whole-part-Struktur genannt.

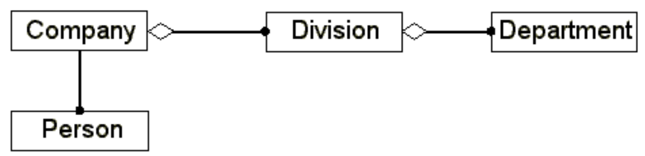


Abbildung 2.17:

Abbildung 2.17 zeigt einen Modellausschnitt mit Assoziation und Aggregation. Zwischen Person und Company gilt die Assoziation: eine Person arbeitet für genau ein Unternehmen. Die Arbeitslosigkeit scheint damit überwunden - es herrscht Vollbeschäftigung. Ein Unternehmen beschäftigt 0 bis n Personen. Das Unternehmen (Company) besteht aus Divisionen, diese wiederum aus Departments.

Die Aggregation ist transitiv und antisymmetrisch. Transivität bedeutet: Wenn a mit b in Beziehung steht und b mit c, dann steht auch a mit c in Beziehung.

Bezogen auf das Beispiel: weil Company aus Divisions besteht und Divisions aus Departments, bestehen Companies auch aus Departments.

Die Antisymmetrie besagt, daß wenn a mit b in Beziehung steht, steht b nicht mit a in Beziehung: eine Division besteht nicht aus Companies. Es wird deutlich, daß die Aggregation eine Unterordnungsbeziehung der Teile zum Ganzen beinhaltet.

Zum Identifizieren eine Aggregationsstruktur und zur Unterscheidung von der Assoziation soll ferner berücksichtigt werden, daß oftmals Eigenschaften, die für das Ganze (das Aggregat) gelten, auch für die Teile (Komponenten des Aggregats) gelten. Dies gilt für Attribute, deren Bedeutung oft vom Ganzen auf die Teile übertragbar sind, wie auch für Operationen die sich oft auf die Komponenten fortsetzen. Eine mögliche Eigenschaft der Company aus dem Beispiel wäre die Branchenzugehörigkeit, etwa EDV-Branche. Natürlich ist anzunehmen, daß dieselbe Branchenzugehörigkeit dann auch für die Komponenten, also die Divisions gilt. Gibt es für die Klasse Company eine Operation „Inventur“, so könnte sie sich logisch auf die Divisions und Departments fortsetzen: wenn das Unternehmen Inventur macht, tun dies auch ihre Abteilungen.

2.3.2.3 Aggregation und Generalisierung

Einige typische Fälle, bei denen Aggregation auftreten, sind die Beziehungen zwischen einer Baugruppe und ihren Bauteilen, zwischen einem Behälter und seinem Inhalt und zwischen einer Sammlung und ihren Elementen.

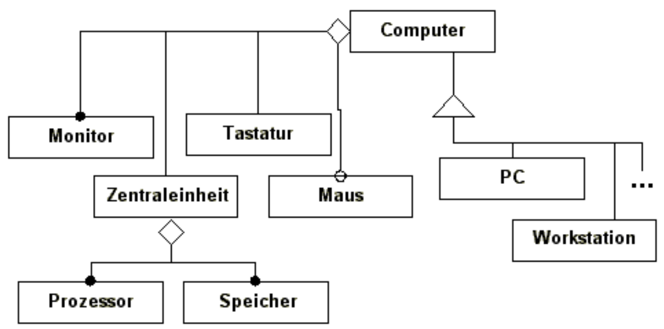


Abbildung 2.18:

Ein Beispiel für einen Modellausschnitt mit Aggregation und Generalisierung zeigt Abbildung 2.18. Die Aggregation ähnelt einer UND-Beziehung: ein Computer besteht aus Monitor und Zentraleinheit und Maus und Tastatur. Generalisierung ähnelt einer ODER-Beziehung: ein Computer ist ein PC oder eine Workstation oder ...

Bei Aggregations- und Generalisierungsstrukturen besteht Verwechslungsgefahr, insbesondere da auch die Generalisierungsbeziehung transitiv und antisymmetrisch ist. Der Generalisierungsbaum enthält Klassen, die ein Objekt beschreiben. Eine Instanz einer Unterklasse wird durch die Eigenschaften von Unterklasse und Oberklasse beschrieben. Der Aggregationsbaum enthält hingegen Klassen, deren Instanzen Teil eines aggregierten Objekts sind.

2.3.2.4 Aggregationsstrukturen

Die Aggregation kann in einer festen Struktur auftreten. Feste Struktur bedeutet, daß die Komponenten des Aggregats nach Typ und Anzahl festgelegt sind. Dies gilt zum Beispiel für die Beziehung zwischen Computer und Zentraleinheit in Abbildung 2.18. In dem Modell enthält jeder Computer genau eine Zentraleinheit. Da das Modell erlaubt, das eine unbestimmte Anzahl von Monitoren sowie eine optionale Maus Bestandteil des Computers sein können, handelt es sich hier jedoch um eine variable Struktur. In der variablen Struktur ist die Anzahl der Komponenten unbestimmt. Die Anzahl der Ebenen des Aggregationsbaums ist hingegen feststehend.

Rekursive Aggregationsstrukturen mit beliebiger Tiefe des Aggregationsbaums sind möglich.

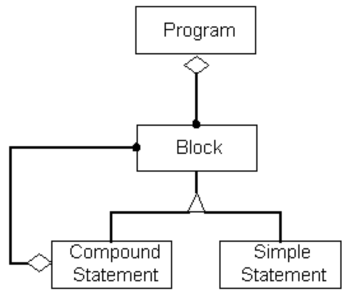


Abbildung 2.19:

Abbildung 2.19 modelliert den Aufbau eines Computerprogramms aus„Blocks“ und „Statements“. Das Programm besteht aus „Blocks“ Jeder „Block“ ist entweder ein „Simple Statement“, wobei die Rekursion hier endet, oder ein „Compound Statement“. Diese sind wiederum aus „Blocks“ zusammengesetzt.

Die hier gezeigte Struktur ist typisch für rekursive Aggregation. Das Aggregat enthält (direkt oder indirekt) eine Instanz seiner eigenen Klasse. Die Anzahl der Aggregationsstufen ist nicht begrenzt. Das Aggregat ist eine (abstrakte) Oberklasse mit zwei Unterklassen. Eine der Unterklassen dient als Zwischenknoten im rekursiven Aggregationsbaum. Sie ist aus Instanzen der Oberklasse (bzw. ihrer Unterklassen) zusammengesetzt. Die Rekursion endet bei einer Instanz der anderen Unterklasse (terminale Unterklasse).

Natürlich muß die Aggregation azyklisch sein: eine Instanz des Aggregats kann nicht sich selbst beinhalten - sonst würde die Rekursion nie enden.

2.3.2.5 Fortpflanzung von Operationen

Die automatische Fortpflanzung von Operationen über assoziierte Objekte wird Propagation genannt. Propagation basiert auf der Vorstellung, daß eine Operation bei einem Objekt startet (aufgerufen wird) und sich über die Links (Instanzen der Beziehungen zu anderen Klassen) automatisch fortsetzt.

Mit Hilfe des Konzepts der Propagation von Operationen wird das Ausbreitungsverhalten einer Operation beschrieben. Die Notation für Propagation ist ein Pfeil, der mit dem Namen der Operation beschriftet wird. Der Pfeil ist stets auf eine Beziehung (Assoziation, Aggregation) bezogen. Die progagierte Operation, der Ausbreitungsweg (Beziehung) sowie die Richtung der Ausbreitung sind daher im Modell ersichtlich.

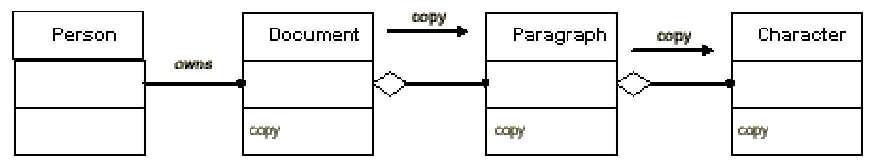


Abbildung 2.20:

Das Beispiel in Abbildung 2.20 zeigt einen Modellausschnitt mit Anwendung der Porpagations-Notation. Eine Person besitzt 0 bis n Dokumente. Ein „Document“ besteht aus mehreren „Paragraphs“, welche wiederum aus mehreren „Characters“ bestehen. Die Operation „copy“ breitet sich vom „Document“ über „Paragraphs“ und „Characters“ aus. Beim Kopieren des Dokuments werden alle darin enthaltenen Absätze und weiter die darin enthaltenen Zeichen kopiert. Die copy-Operation breitet sich aber nur in eine Richtung aus. Wenn ein Zeichen kopiert wird, wird nicht automatisch der Absatz kopiert, in dem das Zeichen enthalten ist. Die Operation breitet sich nur über die Beziehungen aus, an denen dies mit dem Pfeil gekennzeichnet ist: beim Kopieren des Dokuments wird nicht gleichzeitig ein Duplikat des Eigentümers erzeugt.

2.3.2.6 Abstrakte und konkrete Klassen

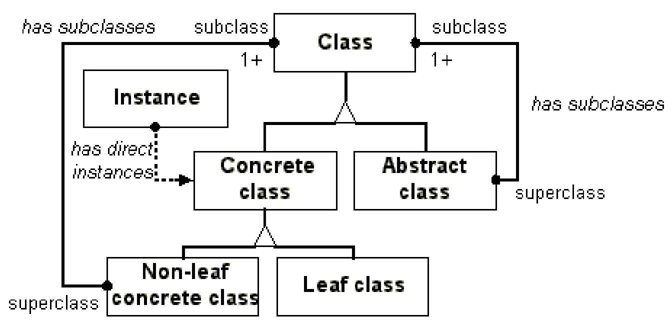


Abbildung 2.21:

In Abbildung 2.21 werden Klassen nach ihrer Funktion im Modell unterschieden: konkrete Klassen sind solche, die direkte Instanzen haben können, abstrakte Klassen dienen ausschließlich als Oberklasse für weitere Unterklassen. Eine abstrakte Klasse hat mindestens eine Unterklasse. Konkrete Klassen können als Blätter (Leaf class) im Generalisierungsbaum auftreten. Eine konkrete Klasse mit direkten Instanzen kann auch weitere Unterklassen haben.

Das Modell zeigt, daß abstrakte Klassen nicht instantiiert werden können und mindestens eine Unterklasse haben.

2.3.2.7 Abstrakte Klassen

Eine abstrakte Klasse ist daran zu erkennen, daß sie Operationen hat, für die sie keine Methode bereitstellt. Eine abstrakte Operation wird mit {abstract} markiert. Wenn in einer Klasse eine oder mehrere Operationen abstrakt sind, gilt die Klasse als abstrakte Klasse.

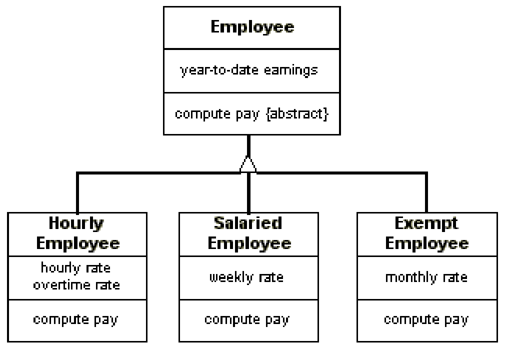


Abbildung 2.22:

Im Beispiel der Abbildung 2.22 ist die Klasse „Employee“ abstrakt. Sie hat eine Operation „compute pay {abstract}“, für die sie keine Methode (Implementation der Operation) bereitstellt.

Die Unterklassen „Hourly Employee“, „Salaried Employee“ und „Exempt Employee“ müssen diese abstrakte Operation Redefinieren und eine Methode bereitstellen, wenn sie instantiierbar sein sollen.

Die abstrakte Klasse legt ein Protokoll für das gemeinsame Verhaltensmuster von Unterklassen fest. Z.B. kann man für alle Unterklassen von Employee mittels „compute pay“ das Gehalt berechnen. Das Verfahren zur konkreten Gehaltsberechnung ist jedoch für die Unterklassen spezifisch.

2.3.2.8 Generalisierung und Erweiterung

Die Generalisierungsstruktur kann eingesetzt werden, um die Eigenschaften der Oberklasse zu erweitern. In Abbildung 2.22 fügen die Unterklassen von Employee jeweils weitere Attribute hinzu. Eine Instanz der Unterklasse ist Instanz aller Vorgänger (Oberklassen) im Vererbungsbaum. Die Attribute und Operationen aller (direkten und indirekten) Oberklassen stehen daher in der Instanz einer Unterklasse zur Verfügung.

Die Erweiterung ist möglich bezüglich der

- Attribute (Abbildung 2.22, Abbildung 2.23)
- Beziehungen (Abbildung 2.19)
- Operationen

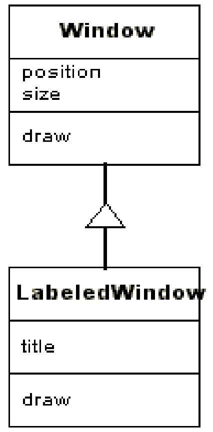


Abbildung 2.23:

Die Unterklasse kann neue Operationen beinhalten. Sie kann auch Operationen der Oberklassen überschreiben. Das Protokoll, das durch die Oberklassen für Operationen festgelegt ist, wird hierbei nicht geändert.

In Abbildung 2.23 ist ein Beispiel für das Überschreiben von Operationen zur Erweiterung angegeben. Die Klasse „Window“ hat Attribute, die Position und Größe des Fensters auf dem Bildschirm angeben. Die Operation draw kann einen Rahmen für das Fenster auf dem Bildschirm zeichnen. Die Unterklasse „LabeledWindow“ enthält zusätzlich ein Attribut für den Titel des Fensters. Um den Titel beim Neuzeichnen des Fensters auszugeben, redefiniert die Klasse die draw-Operation. Hier findet also ein Überschreiben der Operation zum Zweck der Erweiterung des Verhaltens statt.

2.3.2.9 Generalisierung und Begrenzung

Generalisierungsstrukturen können eingesetzt werden, um Eigenschaften der Oberklasse einzuschränken. Dies ist sinnvoll, wenn die spezialisierte Unterklasse einen eingeschränkten Wertebereich für Attribute hat.

So ist zum Beispiel der Kreis in Abbildung 2.24 ein Spezialfall einer Ellipse, bei der die beiden Achsen (major axis, minor axis) gleich lang sind.

Eine Klasse „Circle“ wie in Abbildung 2.25, muß sicherstellen, daß die Attribute „major axis“ und „minor axis“ der Oberklasse stets den eingeschränkten Wertebereich der Unterklasse beachten.

Daraus kann sich auch die Notwendigkeit zur Einschränkung des Protokolls ergeben. Die Operation „scale“ der Klasse Ellipse könnte es ermöglichen, den Wert der beiden Achsen getrennt zu verändern. Die scale-Operation der Klasse Circle darf nur eine gleichmäßige Veränderung der beiden Achsen zulassen.

Geerbte Attribute und Operationen können in der Unterklasse umbenannt werden. Da die beiden Achsen (major axis, minor axis) der Klasse Ellipse in der

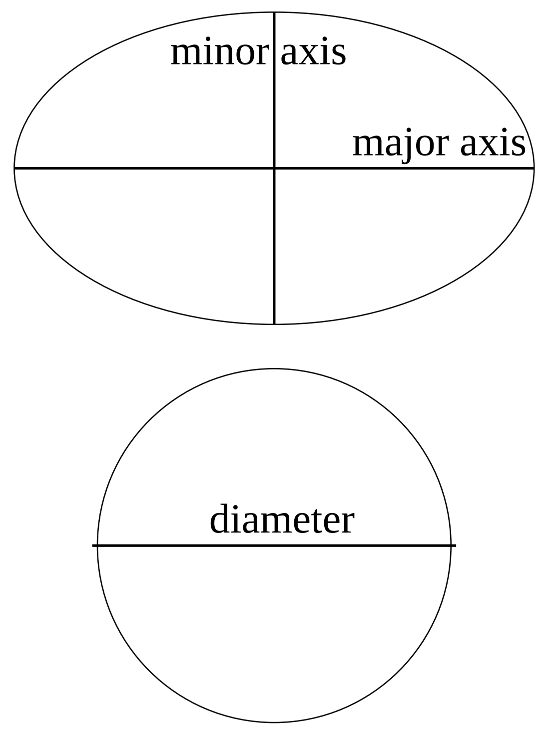


Abbildung 2.24:

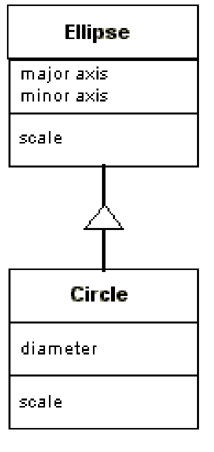


Abbildung 2.25:

Unterklasse Circle stets denselben Wert haben müssen, werden sie durch diameter (Durchmesser) ersetzt.

Die Begrenzung kann sich auf den Wertebereich der Attribute der Oberklasse, auf das Verhalten von Operationen oder auch auf den Wertebereich der Argumente und Rückgabewerte beziehen. Z.B. könnte eine Containerklasse für Ellipsen, eine Liste von Ellipsen, eine Operation „first“ anbieten, die ein Ellipsen-Objekt zurückliefert. Eine spezialisierte Containerklasse für Objekte der Klasse „Circle“ würde stattdessen eine „first-Operation“ anbieten, die „Circle-Objekte“ zurückliefert.

Wenn die Generalisierung mit „Begrenzung der Eigenschaften der Oberklasse“ eingesetzt wird, kann sich die Notwendigkeit zum Überschreiben (Redefinieren) von Operationen zum Zweck der Begrenzung ergeben.

Ein anderer Zweck ist das Überschreiben zur Optimierung. Wenn der Wertebereich von Argumenten und / oder Attributen eingeschränkt ist, kann dadurch evtl. ein geeigneterer (effizienterer, sicherer, schönerer...) Algorithmus verwendet werden. Als Beispiel sei hier die „rotate-Operation“ angeführt. Während das Rotieren einer Ellipse zu sichtbaren Veränderungen führt, wirkt sich das Rotieren eines Kreises nicht sichtbar aus. Eine rotate-Operation der Klasse Kreis kann deshalb als NULL-Operation implementiert werden.

2.3.2.10 Mehrfachvererbung

Es ist möglich, daß eine Unterklasse mehrere direkte Oberklassen hat. Eine solche Unterklasse wird Vereinigungsklasse genannt. Sie erbt Eigenschaften aus mehreren Generalisierungsbeziehungen. Ein Beispiel ist die Klasse „Amphibienfahrzeug“. In einer Generalisierungsstruktur für Fahrzeuge, die auf dem Kriterium Einsatzort bzw. Einsatzumgebung (discriminator) beruht, könnte diese Klasse eine Vereinigung der Klassen „Landfahrzeug“ und „Wasserfahrzeug“ sein.

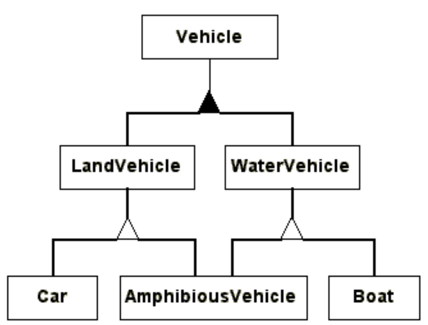


Abbildung 2.26:

Falls die direkten Oberklassen einer Mehrfachvererbungsstruktur aus demselben Generalisierungsbaum stammen, können sie nicht disjunkt sein. Im Beispiel (Abbildung 2.26) ist an dem ausgefüllten Dreieck der Generalisierungsstruktur zu erkennen, daß „LandVehicle“ und „WaterVehicle“ nicht-disjunkte Unterklassen von „Vehicle“ sind.

Bei Mehrfachvererbung ist Mehrdeutigkeit möglich, wenn mehrere Oberklassen dieselbe Operation oder dasselbe Attribut (bzw. eine gleichnamige Eigenschaft haben).

Eigenschaften einer indirekten Oberklasse werden nur einmal vererbt. Dadurch, daß im Beispiel (Abbildung 2.26) beide Oberklassen der Mehrfachvererbungsstruktur von derselben Klasse Vehicle abgeleitet sind, entsteht also keine Mehrdeutigkeit.

2.3.2.11 Klassenbegriff

Eine Klasse beschreibt eine Kollektion von Objekten mit gleichen Eigenschaften (Attribute, Operationen, Beziehungen) und gemeinsamer Semantik. Objekte einer Klasse haben ein gemeinsames Muster bzgl. der Attribute und des Verhaltens.

Individuelle Objekte einer Klasse unterscheiden sich meistens in Attributwerten - aber nicht immer. Die Identität ist eine inhärente Objekteigenschaft: Objekte einer Klasse werden unterschieden, auch wenn sie dieselben Attributwerte haben - allein aufgrund ihrer Existenz.

Neben der gemeinsamen Semantik haben Objekte einer Klasse auch gemeinsame pragmatische Eigenschaften. Sie dienen demselben Zweck.

Objekte verschiedener Klassen unterscheiden sich meistens in ihrem Muster von Eigenschaften (Attribute, Verhalten) - aber nicht immer. Klassen werden aufgrund ihres individuellen Zwecks (Pragmatik) unterschieden. Objekte haben eine explizite Klassenzugehörigkeit.

Im Gegensatz hierzu wird beim mathematischen Klassenbegriff eine regelbasierte Klassenzugehörigkeit verwendet. Eine Regel definiert die Bedingung der Klassenzugehörigkeit und ein Objekt gehört der Klasse an, wenn sein Wert (bzw. die Attributwerte) die Regel erfüllt. Die Klassenzugehörigkeit folgt somit aus den Attributwerten.

Ein mathematischer Klassenbegriff ließe sich gut anwenden, wenn die Attribute von Objekten unveränderliche Werte hätten. Tatsächlich ändern sich die Attributwerte eines Objekts während seiner „Lebenszeit“ - das Objekt bleibt dennoch dasselbe.

Bei dem hier verwendeten Begriff expliziter Klassenzugehörigkeit ist ein Objekt eine diskrete Einheit mit Identität und der impliziten Eigenschaft einer ausdrücklichen Klassenzugehörigkeit. Ein Objekt „kennt“ seine Klasse. Die Attribute des Objekts folgen hierbei aus der Klassenzugehörigkeit.

Im Unterschied zum mathematischen Klassenbegriff wird hier also ein programmiersprachlicher Klassenbegriff verwendet.

Analog zum Verhältnis zwischen Entity und Entitytyp ist die Beziehung zwischen Objekt und Klasse zu sehen. Eine Klasse gibt den Typ der ihr zugeordneten Objekte an. Aufgrund der expliziten Klassenzugehörigkeit sei hier eine namentliche Typäquivalenz (im Gegensatz zu struktureller Typäquivalenz) vorausgesetzt.

Ferner wird eine Klasse als „Kollektion“ im Sinne einer abzählbaren, endlichen Menge aufgefaßt.

2.3.2.12 Delegation

Delegation ist ein weiteres Modellierungsmittel, mit dem sich Sachverhalte der realen Welt darstellen lassen. OMT kennt Delegation nicht als eigenständiges, notierbares Konzept. Demzufolge widmen Rumbaugh et al. der Delegation auch keinen eigenen Abschnitt, sondern führen sie als ein Mittel zur Ersetzung von Mehrfachvererbung ein.

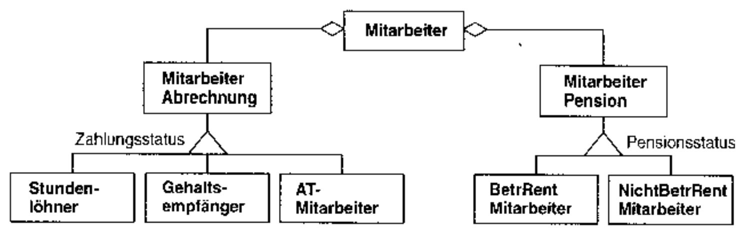


Abbildung 2.27:

Delegation kann aber auch mit OMT als Aggregation modelliert werden. Dabei werden die Eigenschaften mehrerer Klassen in einer aggregierten Klasse zusammengeführt. Es besteht zwischen den Instanzen der delegierenden Klasse und den aggregierten Klassen keine „Instanz-von-Beziehung“ mehr. Demzufolge werden auch die Attribute und Operationen dieser Klassen nicht vererbt. Anstelle dessen definiert die delegierende Klasse eigene Operationen, die implementiert sind, indem die zu erledigende Tätigkeit an die aggregierten Objekte delegiert wird.

2.3.2.13 Metadaten: Klassenattribute und -operationen

Metadaten sind Daten, die andere Daten beschreiben. Systeme, die Metadaten enthalten, modellieren eine zusätzliche Abstraktionsebene, in der über Objekte „gesprochen“ werden kann. Metadaten sind häufig verwirrend, weil sie die gewöhnlich vorhandene Grenze zwischen dem Modell und der realen Welt verwischen.

Klassen können selber Objekte sein. Dann sind sie Metaobjekte und Instanzen einer Metaklasse "Klasse". Klassenobjekte können in ihrer Eigenschaft als Objekt auch Attribute und Operationen besitzen, die dann Klassenoperationen und -attribute genannt werden. Mit OMT werden solche Elemente durch ein Dollarzeichen gekennzeichnet.

Programmiersprachen und Datenbanksysteme bieten diese Konzepte an:

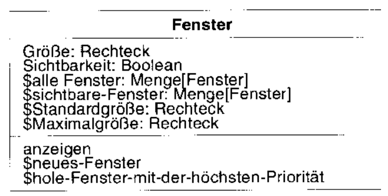


Abbildung 2.28:

- In C++ und Java gibt es Metaattribute und -operationen (static).
- In Smalltalk gibt es Metaklassen.

In Datenbanksystemen gibt es neben den Anwendungsdaten auch Daten, die Tabellenstrukturen und Attribute beschreiben (Data Dictionary).

2.3.2.14 Metadaten: Muster

Auch in der realen Welt gibt es Metadaten, z.B. beschreibt ein Architekturplan ein Haus. Solche Zusammenhänge führen zu „Instanz-von-Beziehungen" zwischen Objekten im Modell. Mit OMT können solche Beziehungen mit einer Verallgemeinerung der „Instanz-von-Beziehung" auf Klassen beschrieben werden. Rumbaugh et al. sprechen dann von einem Muster. Diese Form von Beziehung ist eine spezielle Assoziation bei der jeweils N Objekte der Instanzenklasse mit genau einem Objekt der Musterklasse assoziiert sind.

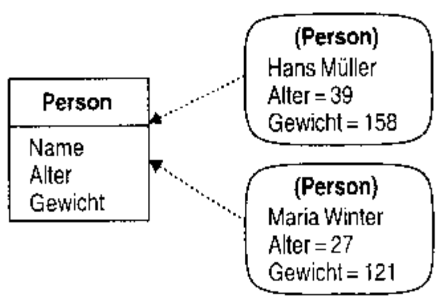


Abbildung 2.29:

2.3.2.15 Kandidatenschlüssel

OMT bietet mit Kandidatenschlüsseln eine andere Möglichkeit, um die Multiplizität (Kardinalität) von Beziehungen anzugeben. Kandidatenschlüssel werden in geschweiften Klammern„in der Nähe" der Beziehung angegeben.

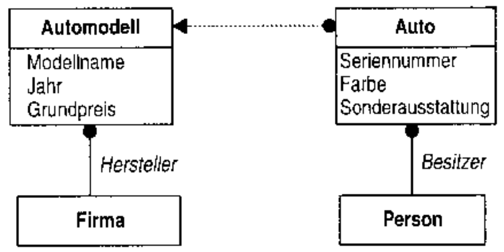
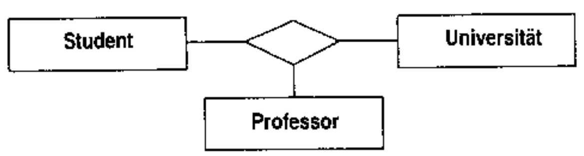


Abbildung 2.30:



{Kandidatenschlüssel: {Student, Universität}}

Student	Professor	Universität
Mary	Prof Weaver	Cambridge
Mary	Prof Rumrow	MIT
Susan	Prof Weaver	MIT
Susan	Prof Weaver	Cambridge
Bob	Prof Shapiro	Oxford

Abbildung 2.31:

Bei zweistelligen Beziehungen gibt es keine bedeutenden Unterschiede zur Punktnotation, bei mehrstelligen Beziehungen sind Kandidatenschlüssel jedoch die einzige Möglichkeit, die Multiplizität eindeutig anzugeben.

Ein Kandidatenschlüssel ist eine minimale identifizierende Menge von Objekten einer Beziehung (d.h. es kann auch mehrere Kandidatenschlüssel geben). Das Konzept ist eng verwandt mit den Schlüsselkonzepten der Theorie relationaler Datenbanken.

2.3.2.16 Assoziationen als Klassen

Neben einfachen Attributen können Assoziationen auch mit Objekten attributtiert werden. Dies ist insbesondere dann wünschenswert, wenn auch Operationen angegeben werden sollen, oder die Attributklasse selbst Beziehungen zu anderen Klassen ermöglichen soll.

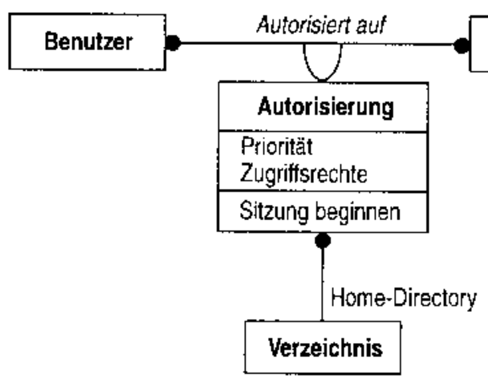


Abbildung 2.32:

2.3.2.17 Einschränkungen

OMT verwendet für nicht grafisch notierbare Sachverhalte Anmerkungen in geschweiften Klammen, z.B. bei:

- Kandidatenschlüsseln
- Ordnungseigenschaften von Assoziationen
- Abstraktion von Operationen

Damit können auch allgemeine Einschränkungen auf Objekten angegeben werden.

Einschränkungen, die mehrere Modellentitäten umfassen, verweisen auf diese mit gestrichelten Linien (siehe Abbildung 2.34).

Abbildung 2.33:

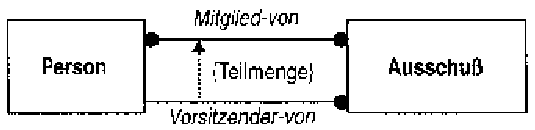


Abbildung 2.34:

Einschränkungen sind Invarianten eines Objektmodells. Einschränkungen können sich auf alle Entitäten des Modells beziehen.

Wie schon erwähnt verwendet OMT geschweifte Klammern für Einschränkungen und alles, für das keine grafische Notation vorgesehen ist; im Prinzip sind die Klammern Kommentarzeichen, da keine Syntax festgelegt ist. Daran wird auch deutlich, worin die Schwäche einer grafischen Notation besteht.

Ein System, das OMT-Diagramme auswerten, und bespielsweise in Code umsetzen wollte, müßte Anmerkungen in geschweiften Klammern entweder ignorieren, oder dafür Syntax und Semantik formal angeben.

2.3.2.18 Homomorphismus als spez. Einschränkung

Homomorphismus bedeutet „von gleicher Gestalt“. In Verbindung mit Assoziationsbeziehungen zwischen Klassen und darauf definierten Abbildungen, bedeutet das „Strukturgleichheit“. Homomorphismus ist keine besondere Einschränkung, aber Homomorphismen kommen in wirklichen Modellen häufig vor.

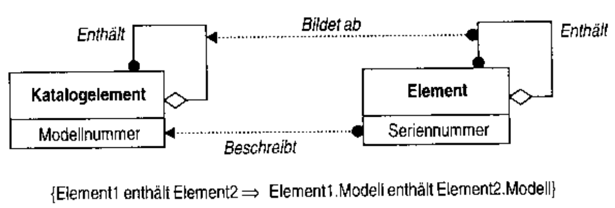


Abbildung 2.35:

Allgemein bietet OMT (genauso wie andere Methoden) für die Modellierung

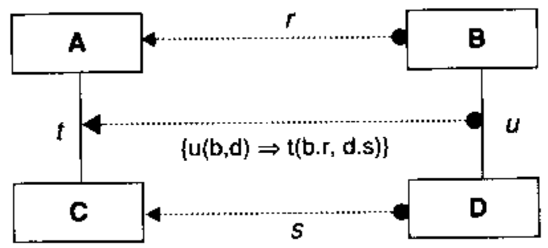


Abbildung 2.36:

solcher komplexen Modelleigenschaften keine Notation an, man behilft sich mit einschränkenden Anmerkungen in der {}-Notation.

2.3.2.19 Abgeleitete Objekte, Assoziationen und Attribute

Abgeleitete Objekte und Entitäten sind solche, die sich vollständig aus anderen Entitäten eines Objektmodells ergeben und sind damit redundant. Nach Rumbaugh et al. ist jedoch manchmal sinnvoll solche Entitäten aus Verständlichkeitsgründen in ein Objektmodell aufzunehmen.

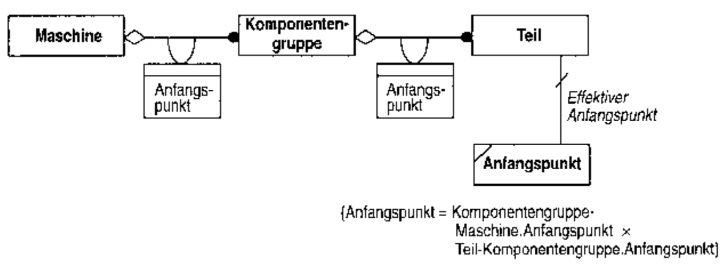


Abbildung 2.37:

Bei OMT werden abgeleitete Entitäten durch einen Schrägstrich „/“ gekennzeichnet.

Häufig repräsentiert ein abgeleitetes Objekt ein sinnvolles Konzept der realen Welt.

2.3.2.20 Darstellung komplexer Modelle

Problematisch ist bei allen Objekt-oreintierten Methoden die Darstellung komplexer und großer Modelle.

OMT schlägt die Aufteilung in Module vor, die jeweils eine Seite umfassen. Der Nachteil hierbei ist, daß Klassen an Modulgrenzen auf mehreren Seiten auftauchen.

Andere Methoden schlagen eine hierarchische Strukturierung des Modells vor. Zum Beispiel in Cluster, Subjects oder ähnliches. Dabei ist allerdings unklar, wie Assoziationen zum Cluster oder Subject gehandhabt werden.

Interessant wäre auch, die Komponenten (PDC , HIC, TMC, DMC nach Coad/Yourdon) eines Systems in verschiedenen Dimensionen oder auf verschiedenen Layern darzustellen.

In der OOA/OOD-Methode nach Coad/Yourdon wird die Layer-Technik jedoch stattdessen für die verschiedenen Klassen von Modellentitäten (Subjects, Klassen&Objekte, Attribute, Operationen, Beziehungen) verwendet.

Eine andere Alternative wäre die Unterdrückung von Modellentitäten (nach Typ oder nach Bedeutung) in verschiedenen Sichten, z.B. Klassen ohne Operationen/Attribute darstellen oder nur besonders wichtige Assoziationen anzeigen.

Die Darstellbarkeit komplexer Modelle ist ein wichtiges Thema für jeden, der solche Modelle erstellt, und jedes System, das für die Bearbeitung solcher Modelle konzipiert wird (denn jeder Bildschirm ist bekanntlich zu klein).

OMT bietet für dieses Problem letztlich nur genauso wenig befriedigende Patentrezepte an, wie andere Methoden.

2.3.3 Dynamisches Modell

2.3.3.1 Einleitung

Objekte werden erzeugt und zerstört, Objekte senden Nachrichten an andere Objekte in einer festgelegten Art und Weise. Daher ist es offensichtlich, daß wir die dynamischen Ereignisse beschreiben müssen.

In der objektorientierten Entwicklung mit OMT drückt man die dynamische Semantik eines Problems oder seine Implementierung durch Diagramme aus. Jede Klasse kann ein Diagramm besitzen, das das ereignisgesteuerte Verhalten der Klasseninstanzen angibt. Das Dynamikmodell beschreibt zeitliche Verhaltensfolgen durch Ereignisse, Zustände, Zustandsdiagramme, Bedingungen und Operationen. Das Objektmodell repräsentierte die statische Struktur eines Systems, wobei das dynamische Modell die Kontrollstruktur eines Systems repräsentiert. Die wichtigsten Konzepte der dynamischen Modellierung sind Ereignisse (events), welche den externen Antrieb darstellen und Zustände (states), welche die Werte der Objekte repräsentieren.

2.3.3.2 Ereignis (Event)

Ein Ereignis ist etwas, daß zu einem bestimmten Zeitpunkt eintritt. Beispiele hierzu:

- Der Flug 123 hebt am 01.10.97 um 9:00 Uhr in Bremen ab.
- Die Veranstaltung Datenbanken ist um 15:00 Uhr

Ein Ereignis verbindet ein Sendeobjekt mit einem Empfängerobjekt, gegebenenfalls werden Informationen von einem Objekt zum anderen Objekt gesendet. Dieses Senden geschieht unidirektional, es erfolgt somit keine Ereignisrückgabe. Im Idealfall sollen Ereignisse Zeitpunkte repräsentieren, also ohne Zeitausdehnung. Das Ereignis ist ein Geschehen, dessen Auftreten eine Veränderung eines Systemstatus bewirken kann.

Nebenläufige Ereignisse (concurrent events) Falls kein kausaler Zusammenhang zwischen dem Auftreten der Ereignisse vorliegt, nennt man dies nebenläufige Ereignisse. Zum Beispiel:

- Flug 123 muß in Chicago abheben, bevor er in San Francisco landen kann. Die beiden Ereignisse haben einen kausalen Zusammenhang.
- Flug 123 kann schon abfliegen, before oder nachdem Flug 456 in Rom abfliegt. Die beiden Ereignisse haben keinen kausalen Zusammenhang.

Ereignisklassen (event classes) Ereignisse werden zu Ereignisklassen gruppiert und geben jeder Ereignisklasse einen Namen, um eine allgemeine Struktur anzuzeigen. Zum Beispiel:

- Flug 123 fliegt von Chicago ab und Flug 456 fliegt von Rom ab sind beide Instanzen der Ereignisklasse Abflug.

Ereignisattribute

Ereignisklassen haben Attribute, die auf die Informationen, die zu übermitteln sind, hinweisen. Zum Beispiel:

- Abflug(Flugnummer, Stadt)
- Ziffer wählen(Ziffer)

Der Zeitpunkt ist ein implizites Attribut für jedes Ereignis.

2.3.3.3 Scenarios and Event Traces

Szenario Ein Szenario ist eine Folge von Ereignissen innerhalb einer speziellen Systemkonfiguration, wobei ein Ereignis Informationen von einem Objekt zum andern sendet. Es soll hilfreich bei der Analyse als Vorstufe zur Ereignisklassifikation sein. Zum Beispiel: Szenario für den Gebrauch einer Telefonleitung

- caller lifts receiver
- dial tone begins
- caller dials digit(5)

- dialtone ends
- caller dials digit (5)
- caller dials digit (5)
- caller dials digit (1)
- caller dials digit (2)
- caller dials digit (3)
- caller dials digit (4)
- called phone beings ringing
- ringing tone appears in calling phone
- called party answers
- called phone stops ringing
- ringing tone appears in calling phone
- phones are connected
- called party hang up
- phones are disconnected
- caller hangs up

Ereignisfolge (event traces) Eine Ereignisfolge ist eine Verfeinerung eines Szenarios, hier werden die Sender und Empfänger von Ereignissen eindeutig identifiziert. Es gehen horizontale Pfeile, die mit Ereignissen markiert sind, vom Sendeobjekt zum Empfängerobjekt. Die Objekte werden durch die vertikale Markierung angezeigt. Ein Beispiel hierzu zeigt Abbildung 2.38.

2.3.3.4 States

Zustand (states) Zustände kann man als Menge der Attributwerte eines Objektes zu einem Zeitpunkt verstehen. Der Zustand eines Objektes stellt die kumulierten Ereignisse seines Verhaltens dar. Wenn zum Beispiel ein Telefon zum ersten Mal installiert wird, befindet es sich im Status unbeschäftigt, d.h. daß das vorhergehende Verhalten nicht von großem Interesse ist, und daß das Telefon nun in der Lage ist, Anrufe durchzuführen oder anzunehmen. Wenn jemand den Hörer abnimmt, sagen wir, das Telefon ist jetzt abgehoben, und befindet sich im Wählstatus. Ereignisse repräsentieren einen Zeitpunkt, Zustände repräsentieren Zeitintervalle.

Zum Beispiel: Nachdem der Hörer abgehoben wurde und bevor eine Ziffer gewählt wurde, ist die Verbindung im Zustand Wahlton. Ein Ereignis trennt zwei Zustände, und ein Zustand trennt zwei Ereignisse.

Zustände lassen sich auch durch Formulare beschreiben (siehe auch Abbildung 2.39).

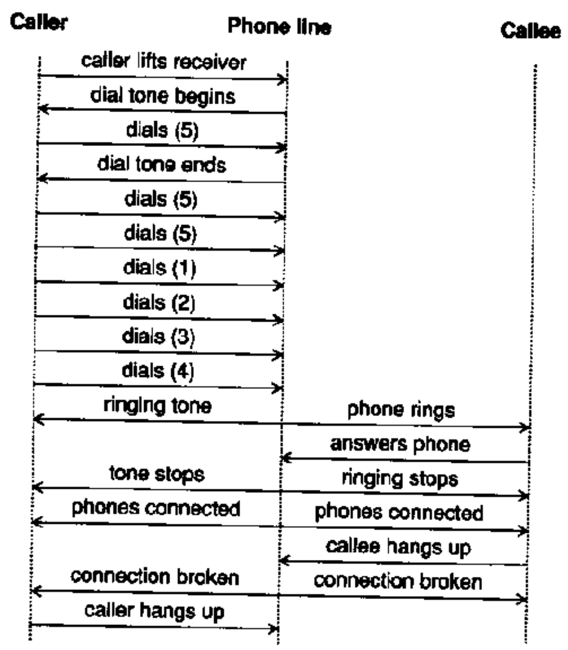


Figure 5.3 Event trace for phone call

Abbildung 2.38: Ereignisse beim Telefonanruf

State: *Alarm ringing*

Description: alarm on watch is ringing to indicate target time

Event sequence that produces the state:

set alarm (target time)

any sequence not including *clear alarm*

current time = target time

Condition that characterizes the state:

alarm = on, and target time ≤ current time ≤ target time + 20 seconds, and no button has not been pushed since target time

Events accepted in the state:

event	action	next state
current time = target time + 20	reset alarm	*normal*
button pushed (any button)	reset alarm	*normal*

Figure 5.4 Various characterizations of a state

Abbildung 2.39: Zustandsformular

2.3.3.5 State Diagram

Zustandsdiagramm (state diagram) Man kann Zustandsdiagramme für jede Klasse mit nennenswertem Verhalten konstruieren. Ein Zustandsdiagramm bringt Ereignisse und Zustände in Beziehung. Eine Zustandsänderung durch ein Ereignis wird Transition genannt. Transitionen verbinden zwei Zustände und werden durch eine Ereignisklasse markiert. Wir können Attributwerte in einem Zustand zusammenbündeln, welche die gleichen Reaktionen auf Ereignisse haben (zum Beispiel digit(n)).Abbildung 2.40 zeigt ein Beispiel hierfür.

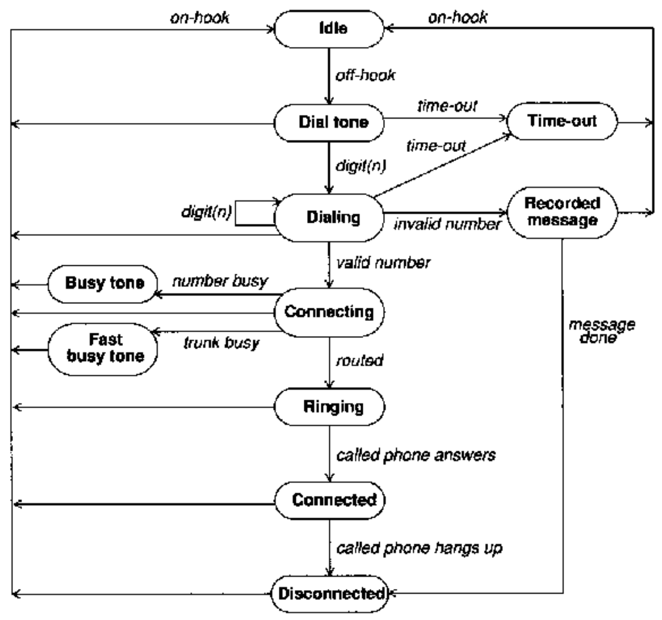


Figure 5.5 State diagram for phone line

Abbildung 2.40: Zustandsdigramm eines Telefons

2.3.3.6 Conditions

Bedingungen (conditions) Bedingungen sind Boolesche Funktionen auf Objektwerten. Sie beziehen sich auf Objektwerte, die typischerweise von der Zeit oder anderen Ereignissen abhängig sind. Bedingungen stehen als Text in eckigen Klammern auf einer Transition. Sie können auch als Wächter für Transitionen benutzt werden, da es zu Zustandsübergängen kommt, wenn die Bedingung erfüllt wird.

2.3.3.7 Operationen

Auslösen von Operationen Objekte reagieren auf Ereignisse durch das Ausführen von Operationen.

Aktivität Eine Aktivität ist eine Operation, deren Ausführung eine gewisse Zeit in Anspruch nimmt. Eine Aktivität ist einem Ereignis zugeordnet und sie läuft bis ihr Zustandsende erreicht ist.

Aktion Eine Aktion ist eine Operation, die für alle praktischen Anwendungen mit der Zeitdauer Null definiert wird. Eine Aktion stellt normalerweise den Aufruf einer Methode, das Auslösen eines anderen Ereignisses oder das Starten oder Beenden einer anderen Aktivität dar. Die Aktion wird an das Ereignis gebunden.

Zusammenfassung der Notationen zum Zustandsdiagramm in Abbildung 2.41:

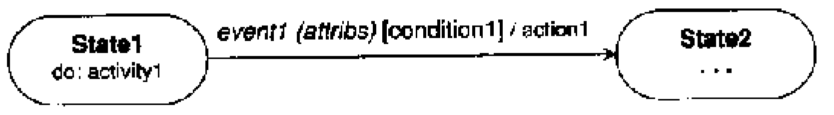


Figure 5.9 Summary of notation for unstructured state diagrams

Abbildung 2.41: Notationszusammenfassung für Zustandsdiagramme

Zusammenfassendes Zustandsdiagramm in Abbildung 2.42:

2.3.3.8 Nested State Diagram

Nested State Diagram Zustandsdiagramme können strukturiert werden, um komplexe Systeme einfacher zu beschreiben. Diese Strukturierung erfolgt durch die Verwendung von Aggregation und Generalisierung. Die Generalisierung ermöglicht es, verschachtelte Zustandsdiagramme (= Nested State Diagram) zu erweitern, wobei Zustände und Ereignisse in einer Hierarchie angeordnet werden. die Aggregation ermöglicht es, einen Zustand in mehrere verschiedene Teile zu zerlegen, wobei diese Zustände nebenläufig sind.

Beim ersten Entwurf von Zustandsdiagrammen treten häufig Probleme auf. Diese Diagramme erweisen sich als unstrukturiert, nicht sehr ausdrucksvoll und sind meistens unpraktisch für große Probleme. Als Beispiel dient ein Zustandsdiagramm mit fünf Zuständen, wobei zwischen jedem Zustand zwei Transitionen verlaufen und somit im gesamten System zwanzig Transitionen auftreten. Diese große Anzahl von möglichen Transition ist das Hauptproblem, da es fast unmöglich ist, jeder möglichen Transition eventuelle Ereignisse zuzuordnen. Ein erster Verbesserungsschritt ist die Streichung von redundanten Elementen, die häufig in komplexen Systemen auftreten.

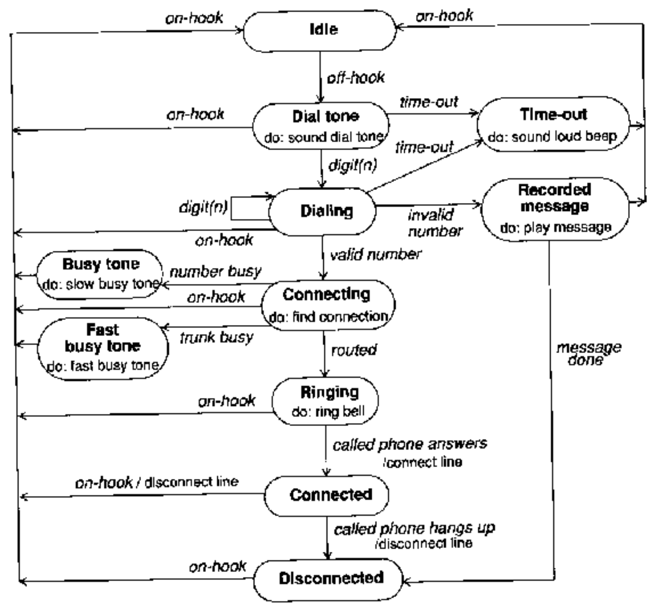


Figure 5.10 State diagram for phone line

Abbildung 2.42: Zustandsdigramm eines Telefons

Um aber Zustandsdiagramme deutlich zu vereinfachen, bedient man sich des Konzeptes der verschachtelten Zustandsdiagramme. Dabei werden Aktivitäten in einem Zustand auf ein Low Level Diagramm erweitert, wobei in diesem Low Level Diagramm jeder neue Zustand ein Schritt der Aktivität innerhalb des übergeordneten Zustands ist.

Als Beispiel für dieses Prinzip der verschachtelten Zustandsdiagramme dienen Abbildung 2.40 und Abbildung 2.44.

2.3.3.9 State Generalization

Ein verschachteltes Zustandsdiagramm ist immer eine Generalisierung von Zuständen, wobei die Zustände eines verschachtelten Diagramms immer die Verfeinerungen eines Zustands aus einem High Level Diagramm sind. Die Generalisierung ist eine „oder“- Beziehung, was bedeutet, daß ein Objekt in einem High Level Diagramm genau in einen Zustand des verschachtelten Diagramms abgebildet werden muß: Es muß sich immer in einem der Zustände befinden.

Zustände haben Unterzustände, die die Transitionen ihrer Oberzustände erben (genau wie die Vererbung von Attributen und Operationen bei Klassen und Unterklassen). Das Prinzip der Zustandsgeneralisierung verdeutlicht Abbildung 2.46, welches das Zustandsdiagramm einer Autogangschaltung zeigt. In diesem Diagramm gibt es drei Zustände: Forward, Neutral und Reverse, wobei der Oberzustand Forward die Unterzustände First, Second und Third beinhaltet. Dabei werden die Transitionen des Oberzustands von den Unterzuständen

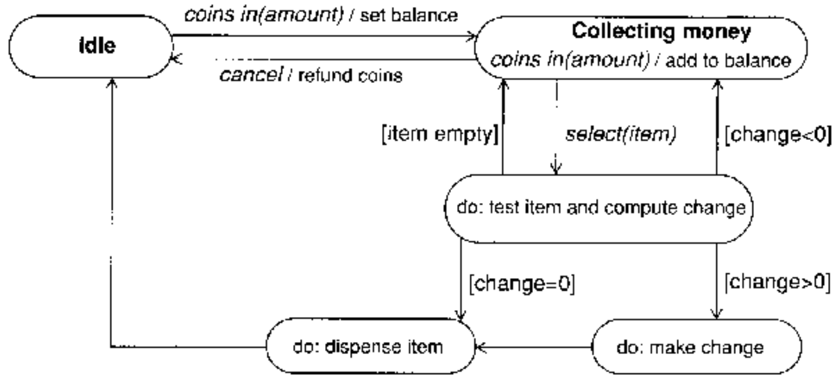


Figure 5.12 Vending machine model

Abbildung 2.43: High Level Diagramm eines Verkaufsautomaten

Figure 5.13 *Dispense item* activity of vending machine

Abbildung 2.44: Aktivität eines Verkaufsautomaten

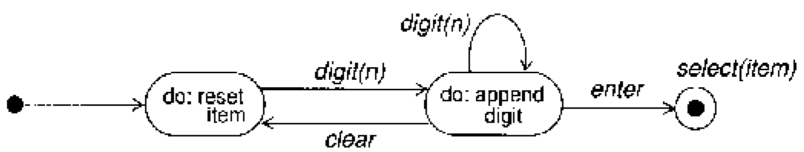


Figure 5.14 *Select item* transition of vending machine

Abbildung 2.45: Transition eines Verkaufsautomaten geerbt, so daß das Drücken von „N“ in allen Vorwärtsgängen bewirkt, daß in den Zustand „Neutral“ gewechselt wird. Innerhalb des Oberzustands ist der Unterzustand „Forward“ der Ausgangszustand (verdeutlicht durch den schwarzen Punkt). Wenn ein Ereignis „stop“ (der Wagen hält) eintritt, dann ist der nächste Zustand immer „First“.

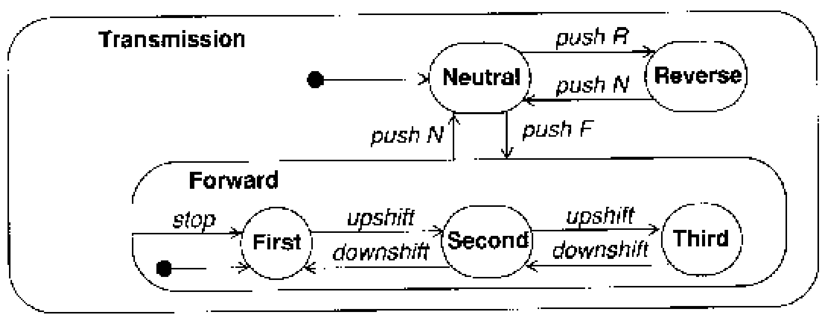


Figure 5.15 State diagram of car transmission with generalization

Abbildung 2.46: Zustandsdiagramm einer Gangschaltung mit Generalisierung

Auch Ereignisse können in einer Generalisierungshierarchie mit Vererbung von Ereignisattributen angeordnet werden. Die Ereignis mouse button down und keyboard character sind zwei Arten von Benutzereingabe. Beide Ereignisse erben das Attribut time vom Ereignis event und das Attribut device vom Event user input. Mouse button down und mouse button up erben location vom mouse button.

2.3.3.10 Nebenläufigkeit (Concurrency)

Es gibt zwei Arten von Concurrency/Nebenläufigkeit: Aggregations-Nebenläufigkeit und Nebenläufigkeit innerhalb eines Objekts.

Aggregations-Nebenläufigkeit

Das dynamische Modell beschreibt eine Menge von nebenläufigen Objekten, wobei jedes Objekt einen Zustand und ein Zustandsdiagramm besitzt. Der Zustand des Gesamtsystems wird dabei nicht durch den Zustand eines einzelnen Objekts repräsentiert, sondern er ist immer das Produkt der Zustände aller Objekte. Das gesamte Zustandsdiagramm ist dabei immer eine Zusammensetzung von vielen einzelnen Zustands Diagrammen der einzelnen Komponenten eines Systems. Somit beinhaltet Aggregation immer Nebenläufigkeit, so daß Aggregation eine „und“-Beziehung darstellt. Der aggregierte Zustand ist ein Zustand aus dem ersten Zustandsdiagramm und ein Zustand aus einem zweiten Diagramm und ein Zustand aus weiteren Diagrammen.

Abbildung 2.47 zeigt dabei den Zustand eines Autos als Aggregation der verschiedenen Zustände Ignition, Transmission, Accelerator und Brake, wobei einige Zustände noch Unterzustände besitzen.

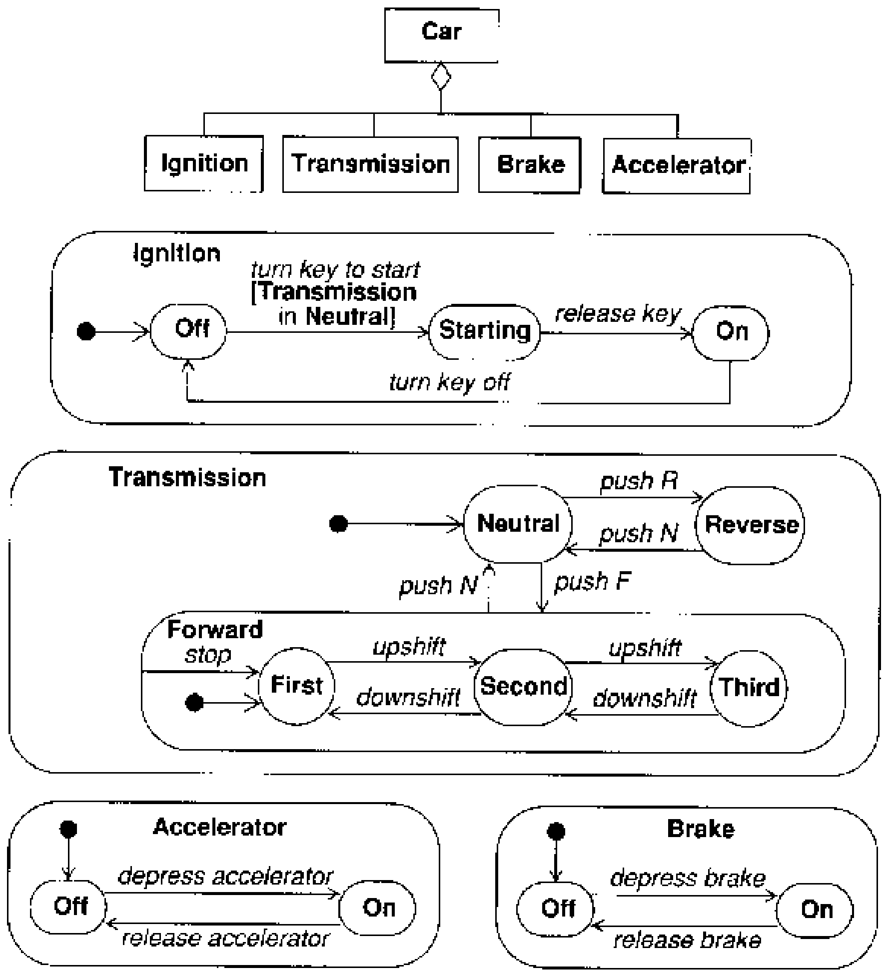


Figure 5.17 An aggregation and its concurrent state diagrams

Abbildung 2.47: Nebenläufige Zustandsdiagramm

Nebenläufigkeit innerhalb von Objekten

Nebenläufigkeit innerhalb eines Zustands eines Objekten entsteht, wenn ein Objekt in Untergruppen von Attributen geteilt werden kann. Jede Untergruppe hat dann ein eigenes Unterdiagramm . Der Zustand eines Objekts wird dabei aus den jeweiligen Zuständen der verschiedenenUnterdiagramme gebildet. Nebenläufigkeit innerhalb eines Objektes wird grafisch verdeutlicht, indem man den Zustand in zwei oder mehr Suddiagramm durch gepunktete Linien teilt.

2.3.3.11 Advanced Dynamic Modell

In diesem Abschnitt wird das erweiterte dynamische Modell vorgestellt, welches eine Verfeinerung des dynamischen Modells darstellt.

2.3.3.12 Entry/ Exit

Eingangs- und Ausgangsaktionen sind eine alternative Notation zu den Aktionen bei Transitionen. Dabei ist es möglich, dass Aktionen mit dem Betreten oder Verlassen eines Objekts assoziiert werden. Abbildung 2.48 zeigt ein Zustandsdiagramm eines Garagentoröffners, in dem Aktionen verwendet werden, wenn ein Zustand betreten wird. Wenn ein Zustand durch eine Transition betreten wird, erfolgt die Ausführung einer Entry Aktion (z.B. entry/ Motor off). Exit Aktionen sind seltener, aber das Prinzip ist das selbe. Wenn ein Zustand durch ein Transition verlassen wird, so wird die Exit-Aktion ausgeführt.

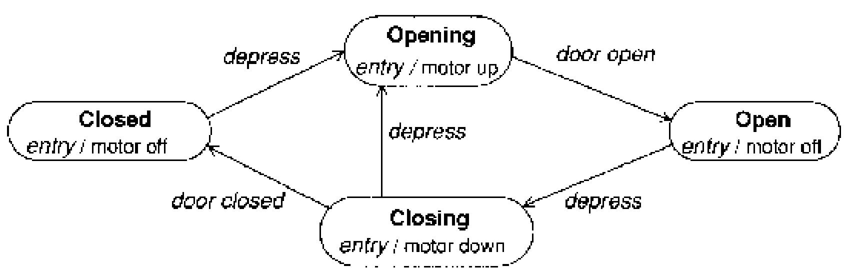


Figure 5.20 Actions on entry to states

Abbildung 2.48: Entry-Aktion eines Zustands

2.3.3.13 Internal Transition

Interne Aktionen treten auf, wenn ein Ereignis eintritt, ohne das ein Zustandswechsel erfolgt. Der Name eines Ereignis wird dabei nicht einer Transition zugeordnet, sondern er wird in den Zustandskasten eingetragen, um zu verdeutlichen, daß dieser Zustand durch dieses Ereignis nicht verlassen wird. Die Notation innerhalb eines Zustandes ist: event/ action. Ein Beispiel für solch eine interne Aktion ist in Abbildung 2.43 beim Zustand collecting money erkennbar.

2.3.3.14 Automatic Transition

Transitionen ohne einen Ereignisnamen werden automatische Transition genannt, die dann ausgeführt werden, wenn eine assoziierte Aktivität beendet ist. Wenn keine Aktivität angegeben ist, so wird die automatische Transition ausgeführt, wenn ein Zustand betreten wird (Lambda Tranistion).

2.3.3.15 Sending Events

Ein Objekt kann Ereignisse an andere Objekte versenden. Dabei interagiert ein Objektsystem durch den Austausch von Ereignissen. Notation: send Event (attributes).

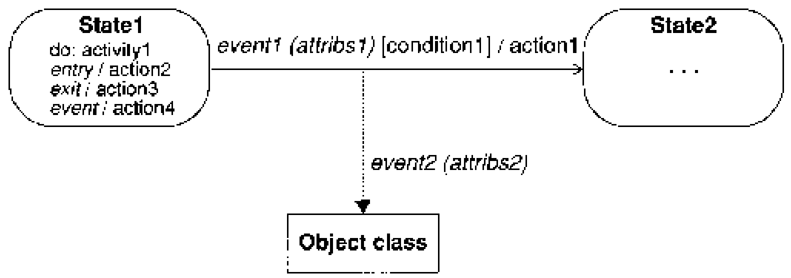


Figure 5.21 Summary of extended notation for state diagrams

Abbildung 2.49: Zusammenfassung der erweiterten Notation von Zustandsdiagramme

2.3.3.16 Synchronisation of concurrent Activities

Synchronisation von nebenläufigen Aktivitäten tritt auf, wenn ein Objekt zwei oder mehr Aktivitäten nebenläufig ausführen möchte. Die interne Abarbeitung der Aktivitäten erfolgt nicht unbedingt synchron, es ist aber notwendig, daß beide interne Aktivitäten beendet sein müssen, damit das Objekt in den nächsten Zustand gelangen kann.

Abbildung 2.50 zeigt die Ausgabefunktion eines Geldautomats. Der Zustand „Emitting" beinhaltet zwei nebenläufige Aktivitäten, die durch gepunktete Linien getrennt sind. Damit der Zustand Emitting beendet wird und ein Zustandswechsel erfolgen kann, müssen die beiden Aktivitäten dispense cash und eject card abgeschlossen sein. Diese Synchronisation wird durch Transitionspfeile verdeutlicht, die sich teilen (ready) und wieder zusammengehen („cash taken", „card taken").

2.3.3.17 Musterbeispiel

Um zu sehen, wie die verschiedenen Modellkonstrukte zusammenpassen, folgt jetzt ein Musterbeispiel für das dynamische Modell. Es geht hierbei um ein

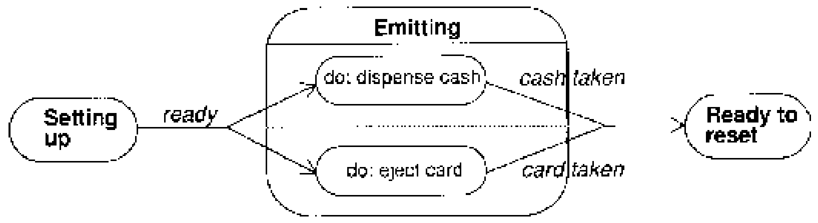


Figure 5.22 Synchronization of control

Abbildung 2.50: Synchronisation

Modell eines programmierbaren Thermostats. Diese Vorrichtung kontrolliert eine Heizanlage bzw. eine Klimaanlage entsprechend der zeitabhängigen Attribute. Die Attribute kann der Benutzer über eine Benutzungsoberfläche mit Knöpfen beeinflussen. Die Hauptaufgabe des Thermostats besteht darin, die gegenwärtige Temperatur an die Zieltemperatur anzugleichen. Für die Zieltemperatur gibt es eine Tabelle mit Programmwerten. Aus dieser Tabelle wird die zu erreichende Temperatur entnommen. Die Werte bestimmt der Benutzer selbst. Die Zieltemperatur gibt es für acht verschiedene Zeitperioden. Die Startzeiten werden wiederum vom Benutzer eingestellt.

Zu Beginn jeder Programmperiode sind die Werte der Tabelle neu zu setzen. Der Benutzer kann die Zieltemperatur innerhalb einer Periode auch überschreiben oder er kann sie unverändert lassen. Jedesmal wenn die Anlage in Betrieb ist, leuchtet eine entsprechende Anzeige auf. Außerdem hat der Thermostat einen Themperatur-Sensor, der die Lufttemperatur abliest.

Der Benutzer programmiert den Thermostat folgendermaßen: Er benutzt eine Unterlage mit zehn Knöpfen und drei Schaltern. Wenn ein Knopf gedrückt ist, bewirkt er jedesmal ein Ereignis. Hierbei wird jedem Knopf eine Eingabefunktion zugewiesen:

Knopf	Funktion
TEMP UP	steigert die Ziel- oder die Programmtemperatur
TEMP DOWN	senkt die Ziel- oder Programmtemperatur
TIME FWD	stellt die Uhr- oder Programmzeit vor
TIME BACK	stellt die Uhr- oder Programmzeit zurück
SET CLOCK	legt die gegenwärtige Zeit des Tages fest
SET DAY	legt den gegenwärtigen Tag der Woche fest
RUN PRGM	verläßt den setup- oder Programmodus und läßt das Programm laufen
VIEW PRGM	betritt den Programmodus
HOLD TEMP	stoppt die gegenwärtige Zieltemperatur (trotz des Programms)
F-C BUTTON	wechselt die Temperaturanzeige zwischen Fahrenheit und Celsius

Es werden die acht Festlegungen für die Programmzeit und die Programmtemperatur geprüft und modifiziert. Jeder Schalter liefert einen Parameterwert, der von zwei oder drei Möglichkeiten ausgewählt wird: Schalter Funktion / Werte Light switch: Beleuchtet die alphanumerische Anzeige. Werte: Licht aus, Licht an. Season switch: Spezifiziert die Thermostatkontrollen. Werte: Heizen (Heizanlage), Kühlen (Klimaanlage), aus (nichts). Fan switch: Spezifiziert, wenn der Lüfter in Betrieb ist. Werte: Lüfter an (Lüfter läuft kontinuierlich), Lüfter auto (Lüfter läuft nur, wenn Heizanlage oder Klimaanlage in Betrieb ist).

Wir modellieren die Kontrollen der Anlage mit den Aktivitäten "Heizanlage in Betrieb", "Klimaanlage in Betrieb", und Lüfter in Betrieb". Für das Modellieren des dynamischen Modells ist es wichtig, nur Zustände einzubeziehen, die den Kontrollfluß beeinträchtigen. Die anderen Informationen werden als Parameter oder Variablen dargestellt.

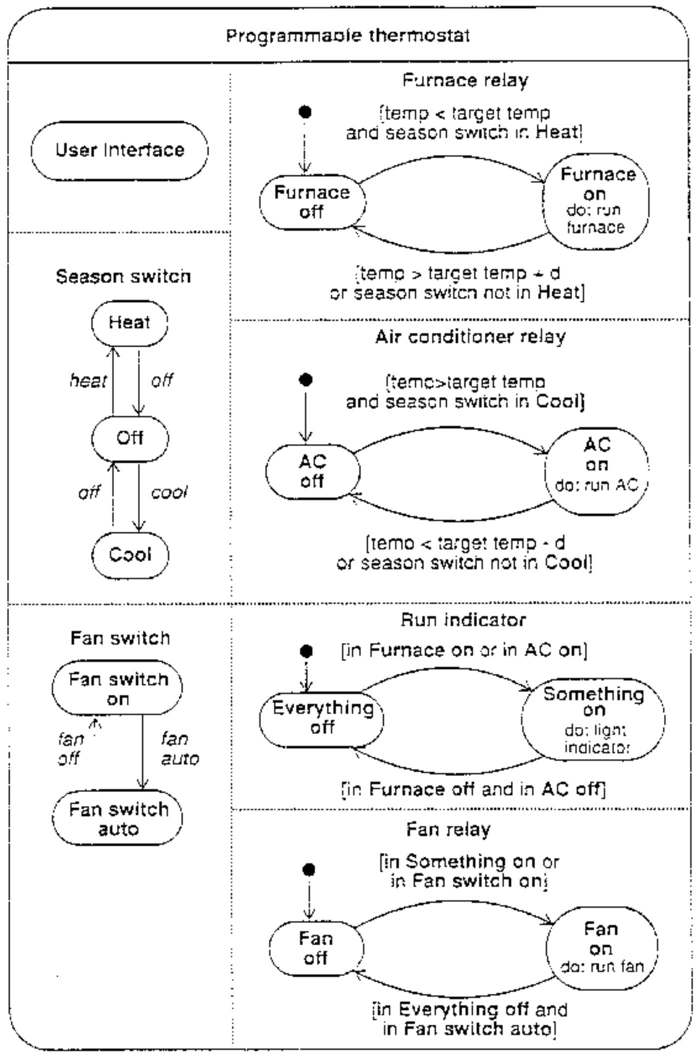
Die Abbildung Abbildung 2.51 zeigt die höchste Stufe im Zustandsdiagramm des Thermostats.

Sie enthält sieben nebenläufige Unterdiagramme. Die Benutzeroberfläche ist zu groß, um gezeigt zu werden. Wie sie im einzelnen aussieht, wird auf der nächsten Abbildung dargestellt. Jeder Schalter wird als ein unabhängiges nebenläufiges Unterdiagramm mit einem Zustand pro Schalterposition dargestellt. Wir weisen Ereignisnamen einer Zustandsänderung zu. Aber letztlich ist es der Zustand jedes Schalters, der wichtig ist. Die anderen vier Unterdiagramme zeigen die Ausgabe des Thermostats: Heizanlage, Klimaanlage, Lüftungsübertragungen, und das Licht für die Betriebsanzeige. Jedes dieser Unterdiagramme enthält einen Aus- und einen An- Unterzustand. Der Zustand von jedem Unterdiagramm wird genau festgesetzt durch die Bedingungen der Eingabeparameter und den Zustand der anderen Unterdiagramme. Der Zustand der vier Unterdiagramme rechts wird komplett abgeleitet und enthält keine zusätzlichen Informationen. Abbildung 2.52 zeigt das Unterdiagramm für die Benutzeroberfläche.

Das Diagramm enthält drei nebenläufige Unterdiagramme, eins für die interaktive Anzeige, eins für den Temperaturmodus und eins für das Nachtlicht. Das Nachtlicht wird von einem physikalischen Schalter kontrolliert, so daß der fehlende Startzustand unwichtig ist; der Wert kann direkt festgesetzt werden. Der Temperaturanzeige-Modus wird von einem einzigen Knopf kontrolliert, der die Temperatureinheiten zwischen Fahrenheit und Celsius verbindet. Der Startzustand ist notwendig; wenn die Vorrichtung angemacht wird, ist der Anfangstemperatur-Modus Fahrenheit.

Das Unterdiagramm für die interaktive Anzeige ist interessanter. Die Vorrichtung ist entweder in Betrieb oder das Programm wird festgelegt. Der Unterzustand Operate beinhaltet zwei Unterzustände, Run und Hold, zusätzlich zu zwei nebenläufigen Unterzuständen, einer, der die Anzeige für die Zieltemperatur kontrolliert, und einer, der die gegenwärtige Zeit und die Temperaturanzeige kontrolliert. Alle zwei Sekunden wechselt die Anzeige zwischen der gegenwärtigen Zeit und der gegenwärtigen Temperatur. Die Zieltemperatur wird kontinuierlich angezeigt und von den „temp up"- und „temp down"-Knöpfen modifiziert, genau wie das „set target"-Ereignis, das nur im Run-Zustand erzeugt wird.

Zu beachten: Der target temp-Parameter, der in diesem Unterdiagramm festgelegt wird, ist derselbe Parameter, der die Ausgabeübertragung kontrolliert. Im Operate-Zustand ist die Vorrichtung entweder in den Run- oder Hold-Unterzuständen. Jede Sekunde wird im Run-Zustand die gegenwärtige Zeit mit der gesamten Programmzeit in der Programmtabelle verglichen; falls sie gleich sind, geht das Programm zur nächsten Programmperiode vor, und es



State diagram for programmable thermostat

Abbildung 2.51: Zustandsdiagramm eines Thermostats

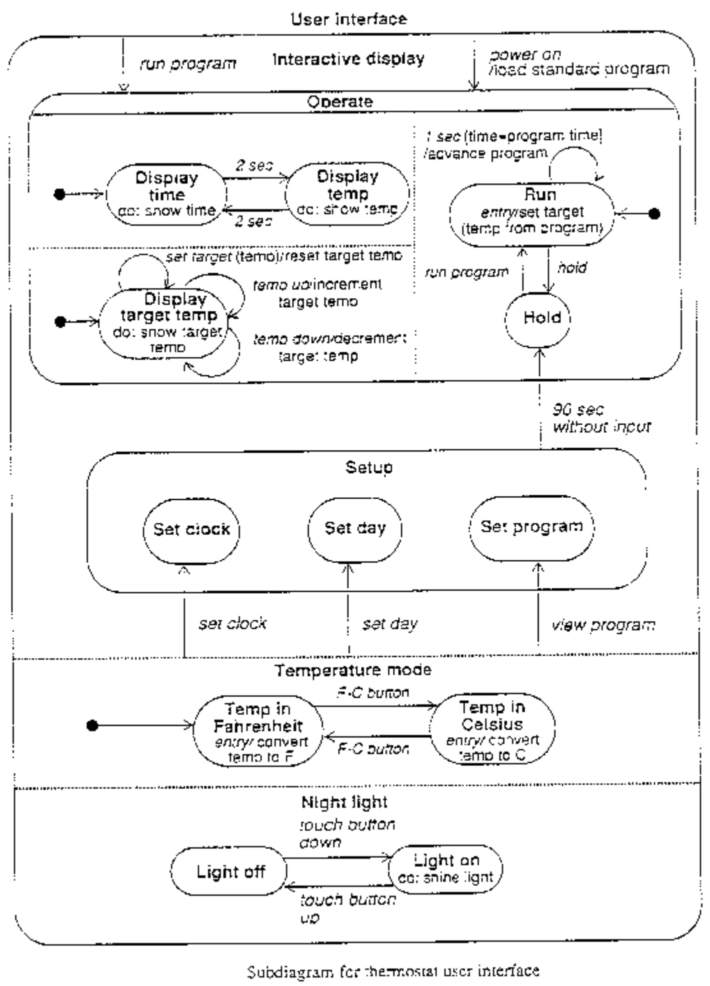
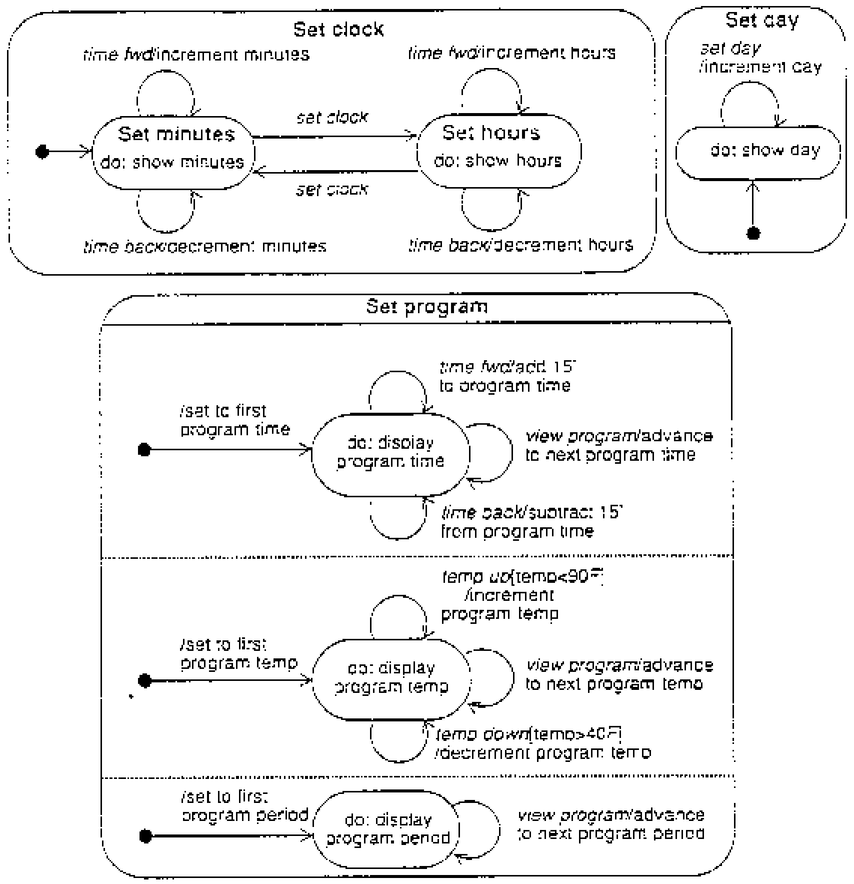


Abbildung 2.52: Unterdiagramm eines Thermostats

wird wieder in den Run-Zustand gegangen. Immer, wenn der Run-Zustand eingenommen wird, wird die Zieltemperatur auf der Programmtabelle neu gesetzt. Wenn die Oberfläche in einem der setup-Zustände 90 Sekunden lang ohne Eingabe ist, dann betritt das System den Hold-Zustand. Der Eintritt in den Hold-Unterzustand bewirkt auch einen Eintritt in den Startzustand der anderen beiden gleichlaufenden Unterdiagramme von Operate. Die drei setup-Unterdiagramme sind in Abbildung 2.53 zu erkennen.



Subdiagrams for thermostat user interface setup

Abbildung 2.53: Unterdiagramm eines Thermostats

Beim Drücken von set clock wird der Set minutes-Unterzustand als Start eingenommen. Bei folgendem Drücken stellt set clock die Verbindung zwischen den set hours- und set minutes-Unterzuständen her. Die time fwd- und time back-Knöpfe stellen die Programmzeit ein. Drücken von set day bewirkt den Set day-Unterzustand und zeigt den Wochentag an. Bei folgendem Drücken werden die nächsten Tage direkt angegeben.

Durch Drücken von view program kommt man in den Set program-Unterzustand, der drei nebenläufige Unterdiagramme hat. Jeweils die Anzeige der Programmzeit, Programmtemperatur und Programm-periode.

Der set program-Zustand startet immer mit der ersten Programmperiode, während die folgenden view program-Ereignisse die acht Programmperioden durchlaufen. Das view program-Ereignis wird auf allen drei Unterdiagrammen gezeigt; jedes Diagramm stellt die Festlegung nach vorne, die es kontrolliert.

Zu beachten: Die time fwd- und time back-Ereignisse steuern time im 15 Minuten-Zuwachs, anders als die gleichen Ereignisse im set clock-Zustand. Auch zu beachten: Die temp up- und temp down-Übergänge haben Überwachungsbedingungen, um die Temperatur in einem fixierten Bereich zu halten. Zum Schluß bleibt festzustellen: Keiner der Interactive display-Unterzustände hat einen expliziten Ausgangsübergang. Jeder Unterzustand wird implizit von einem Übergang in einen anderen Unterzustand beendet.

2.3.3.18 Übergang zwischen Objektmodell und dynamischem Modell

Das dynamische Modell spezifiziert zulässige Abfolgen der Attributwerte von Objekten im Objektmodell. Im einzelnen heißt das: Es gibt ein Zustandsdiagramm. Dieses beschreibt alles, oder ein Teil vom Verhalten eines Objekts in einer gegebenen Klasse. Die Zustände sind Äquivalenzklassen von Attributen und link-Werten für das Objekt. Ereignisse können als Operationen im Objektmodell dargestellt werden. Die Struktur des dynamischen Modells ist eng verbunden mit der Struktur des Objektmodells. Ein Unterzustand verfeinert die Attribute und link-Werte, die das Objekt haben kann. D.h. jeder Unterzustand schränkt diese Werte, die das Objekt haben kann, ein. Diese Verfeinerung von Objektwerten ist eine genaue Generalisierung und Einschränkung (siehe Kapitel 3.3.1 Abschnitt Generalisierung).

Für eine Hierarchie von Zuständen eines Objekts gilt: sie ist äquivalent zur beschränkten Hierarchie der Objekt-Klasse. Objektorientierte Modelle und Sprachen unterstützen normalerweise diese Einschränkung in der (Generalisierungs-)Hierarchie nicht. Daher ist das dynamische Modell der richtige Platz, um dieses darzustellen. Sowohl Generalisierung von Klassen, als auch Generalisierung von Zuständen unterteilen die Menge von möglichen Objektwerten. Ein einzelnes Objekt kann verschiedene Zustände haben, wobei das Objekt seine Identität behält, aber es kann nicht verschiedene Klassen haben. Vererbte Unterschiede zwischen Objekten werden daher als verschiedene Klassen modelliert, während vorübergehende Unterschiede als verschiedene Zustände derselben Klasse modelliert werden. Ein zusammengesetzter Zustand ist die Aggregation von mehr als einem nebenläufigen Unterzustand.

Es gibt drei Quellen von Nebenläufigkeit im Objektmodell: Die erste ist Aggregation von Objekten: Jeder Bestandteil einer Aggregation hat seinen eigenen unabhängigen Zustand, so dass die Zusammensetzung als ein Zustand gesehen werden kann, der sich von all diesen Teilen zusammensetzt.

Die zweite Quelle ist Aggregation mit einem Objekt: Die Attribute und Links eines Objekts sind seine Teile, und zusammengesetzte Gruppen davon definieren nebenläufige Unterzustände (dieses zusammengesetzten Objektzustands.)

Die dritte Quelle ist nebenläufiges Verhalten eines Objekts (Siehe Synchronisations-Abb.) Diese drei Quellen der Nebenläufigkeit sind normalerweise austauschbar.

Ein Beispiel: Ein Objekt kann ein Attribut enthalten, um anzuzeigen, daß es eine bestimmte Aktivität ausführt. Das dynamische Modell einer Klasse wird an seine Unterklassen vererbt. Die Unterklassen können dabei ihre eigenen Zustandsdiagramme haben. Aber wie wirken die Zustandsdiagramme der Oberklasse und die Unterklasse zusammen? Es wurde beschrieben, daß Zustände äquivalent zur Einschränkung von Klassen sind. Es gibt keine Probleme, falls die Zustansdiagramme der Oberklasse und die der Unterklasse sich mit getrennten Mengen der Attribute beschäftigen. Falls jedoch das Zustandsdiagramm der Unterklasse einige der gleichen Attribute enthält, wie das der Oberklasse, existiert ein möglicher Konflikt.

Das Zustandsdiagramm der Unterklasse muß also eine Verfeinerung von dem der Oberklasse sein. (Dabei hat die Unterklasse einen zusammengesetzten Zustand, und zwar zusammengesetzt von nebenläufigen Zustandsdiagrammen.) Es sollte auch ein unabhängiger und nebenläufiger Zusatz zum Zustandsdiagramm der Oberklasse sein. Und zwar definiert mit einer unterschiedlichen Menge an Attributen.

2.3.4 Funktionales Modell

2.3.4.1 Einordnung des funktionalen Modells in das OMT

Die Syntax und die Semantik der einzelnen Bestandteile des funktionalen Modells sollen erläutert und ihre Kombinationsmöglichkeiten an einigen Beispielen demonstriert werden. Beginnen wir aber zunächst mit einer genaueren Betrachtung dessen, was das funktionale Modell leisten soll.

2.3.4.2 Das funktionale Modell

Das funktionale Modell zeigt den Fluß von Ein- zu Ausgabewerten durch ein oder mehrere Datenflußdiagramme (Data Flow Diagrams). Dabei werden von den Datenflußdiagrammen immer Eingaben von einem oder mehreren Objekten von außen entgegengenommen und am Ende auch wieder an ein oder mehrere Objekte ausgegeben, die im Objektmodell spezifiziert wurden. Diese Objekte produzieren Daten als Eingaben für oder konsumieren Daten aus dem Datenflußdiagramm.

Das funktionale Modell spezifiziert sowohl Operationen des Objektmodells als auch Aktionen des dynamischen Modells. Die Operationen, die im Objektmodell bisher nur Namen erhalten haben und Objekten zugeordnet wurden, werden im funktionalen Modell mit Funktionalität versehen.

Das Datenflußdiagramm (Data Flow Diagram)

Das Datenflußdiagramm ist das Konstrukt zur Darstellung der Funktionen. Es zeigt Transformationen von Eingaben durch Prozesse zu Zielobjekten. Ein Datenflußdiagramm setzt sich zusammen aus vier möglichen Bestandteilen, die wir im folgenden näher betrachten wollen.

Der Datenfluß (Data Flow)

Abbildung 2.54:

Die einfachste Form eines Datenflusses ist ein einfacher Wert, er wird im funktionalen Modell als Pfeil dargestellt, der mit einer Bezeichnung für den Wert versehen sein kann (Abbildung 2.54).

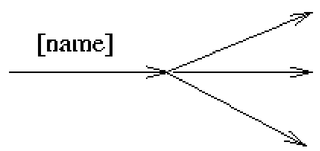


Abbildung 2.55:

Soll ein Datenwert an mehrere Stellen verteilt werden, also mehreren Prozessen als Eingabewert dienen, so kann dies durch einen sich aufspaltenden Pfeil symbolisiert werden, der nur mit einem Namen versehen sein kann (Abbildung 2.55).

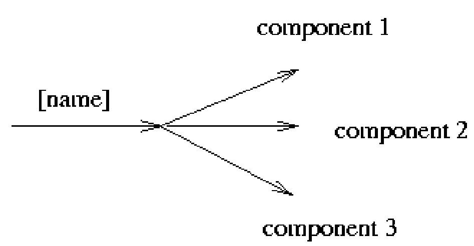


Abbildung 2.56:

Wenn es sich bei dem Datum um einen zusammengesetzten Typ handelt, können die einzelnen Komponenten an unterschiedlichen Prozessen als Eingabe dienen. Hierbei werden die einzelnen Komponentennamen an die sich aufspaltenden Pfeile geschrieben und an den Pfeil, von dem sie ausgehen, der Name des zusammengesetzten Datums (Abbildung 2.56).

Die Vereinigung von mehreren einzelnen Werten zu einem zusammengesetzten Datum geschieht genau andersherum (Abbildung 2.57).

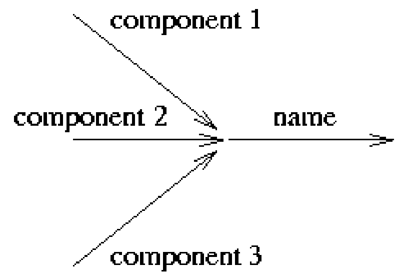


Abbildung 2.57:

Der Prozeß (Process)

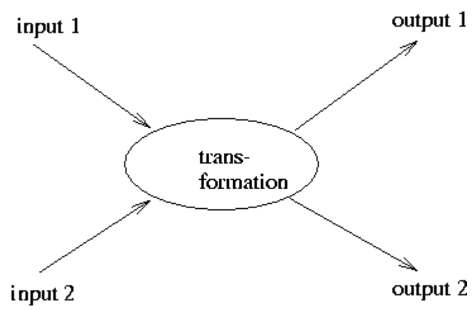


Abbildung 2.58:

Prozesse sind Transformationen, sie führen Funktionen auf Eingabewerten durch und berechnen Ausgaben.

Die Eingaben werden durch mit Namen versehene Pfeile symbolisiert, die auf die Ellipse zeigen, die wiederum den Prozeß darstellt.

In der Ellipse sollte ein aussagekräftiger Name für die Transformation stehen. Die Ausgaben des Prozesses sind mit Namen versehene Pfeile von der Ellipse wegzeigend (Abbildung 2.58).

Actors

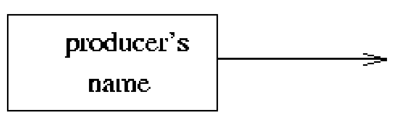


Abbildung 2.59:

Es wird zwischen produzierenden (Abbildung 2.59) und konsumierenden (Abbildung 2.60) Actors unterschieden. Produzierende Actors sind Objekte aus dem

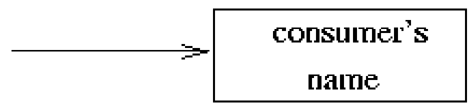


Abbildung 2.60:

Objektmodell die Werte in das Datenflußdiagramm liefern, sie stehen immer an den Rändern des Datenflußdiagramms.

Das Gegenteil sind konsumierende Actors, sie entnehmen Daten aus dem Datenflußdiagramm.

Datenspeicher (Data Store)

Abbildung 2.61:

Datenspeicher sind passive Elemente im funktionalen Modell. In ihnen können Daten gespeichert werden, auf die später zugegriffen werden kann. Der Pfeil in den Datenspeicher zeigt Operationen zum Verändern des Inhalts an. Dies kann das Einfügen, das Verändern oder das Löschen von Daten im Datenspeicher sein. Der Pfeil, der vom Datenspeicher wegzeigt, symbolisiert Zugriffsoperationen, die Daten aus dem Datenspeicher liefern (Abbildung 2.61).

Abbildung 2.62:

Um einen Datenspeicher zu initialisieren benutzt man als Eingangssymbol ein leeres Dreieck statt einer Pfeilspitze (Abbildung 2.62).

Der Kontrollfluß (Control Flow)

Der Kontrollfluß ist redundant zu Angaben im dynamischen Modell. Er ist ein gepunkteter mit einem Namen versehener Pfeil, der für einen booleschen Wert steht, der zunächst True sein muß, damit der Prozeß, der an der Pfeilspitze steht, ausgeführt werden kann (Abbildung 2.63).

2.3.4.3 Spezifikation der Operationen

Prozesse im Datenflußdiagramm müssen eventuell als Operationen auf Objekte implementiert werden. Jeder Prozeß, der elementar, d.h. nicht geschichtet, ist,

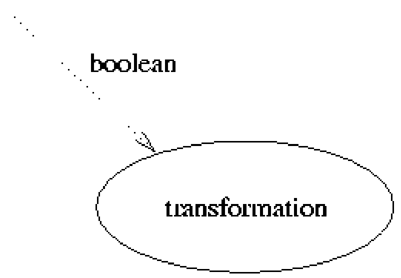


Abbildung 2.63:

ist eine Operation. Jede Operation muß angemessen spezifiziert werden, was auf verschiedene Arten möglich ist. Durch:

- mathematische Funktionen;

- Tabellen von Ein- und Ausgabewerten für kleine festgelegte Fälle;

- Gleichungen, die die Ausgabe in Abhängigkeit der Eingaben spezifizieren;

- Vor- und Nachbedingungen (Axiomatische Definition);

- Entscheidungstabellen;

- Pseudocode oder durch

- natürliche Sprache.

Eine Spezifikation einer Operation beinhaltet eine Signatur und eine Transformation. Die Signatur definiert die Schnittstelle von Operationen: die Argumente, die benötigt werden, sowie die Ausgabewerte (Anzahl, Reihenfolge und Typ). Gewöhnlich werden die Operationen im Objektmodell aufgeführt, um deren Abstammung zu verdeutlichen; die Signatur von allen Methoden, die Operationen implementieren, müssen dementsprechend zusammenpassen. Die Transformation definiert den Effekt der Operation, d.h. die Ausgabewerte als Funktion der Eingabewerte und deren Seiteneffekte.

Diese externe Spezifikation einer Operation beschreibt nur die Veränderungen, die von außen her sichtbar sind. Während der Implementierung von Operationen können interne, weitere Werte verwendet werden, die eine Programmierung bzw. eine Optimierung vereinfachen bzw. ermöglichen, die nur innerhalb der Operation selber zur Verfügung stehen oder möglicherweise innerhalb einer Objektklasse. Der Zweck einer Spezifikation ist die Festlegung was eine Operation logisch machen soll, nicht wie es getan wird. Demnach muß der Zustand eines Objektes zwischen der externen, sichtbaren Information und der privaten Information unterschieden werden. Veränderungen im internen Zustand eines Objektes, die von außen nicht sichtbar sind, haben keine Auswirkungen auf den Wert des Objektes.

Zugangsoperationen (access operations) sind Operationen, die Attribute lesen oder schreiben oder Links von Objekten sind. Eine genaue Spezifikation ist in der Regel nicht notwendig, da diese Zugangsoperationen im allgemeinen klar sind. Für den Entwurf ist es jedoch hilfreich diese Operationen zwischen public und private zu spezifizieren, um Klassen zu kapseln und diese gegen unerwünschte Seiteneffekte zu schützen, und um Veränderungen in der Implementierung zu ermöglichen. Zugangsoperationen sind direkt von den Attributen von Klassen aus dem Objektmodell abgeleitet.

Komplexere Operationen können in drei Kategorien eingeteilt werden: queries, actions und activities. Ein query ist eine Operation, die keinerlei Seiteneffekte aus dem externen, sichtbaren Zustand eines Objektes zeigt; es ist eine reine Funktion. Ein query ohne Parameter (ausgenommen des Zielobjektes) nennt man abgeleitetes Attribut (derived attributes), da es die Form (aber nicht notwendigerweise in der Implementierung) eines Attributes hat. Im Objektmodell können query-Operationen zusammen mit Attributen gruppiert werden, jedoch sollte der abgeleitete Zustand gekennzeichnet sein, da diese dem Zustand des Objektes keine zusätzlichen Informationen beitragen. Da query-Operationen keine externen Seiteneffekte haben, sind diese für die Analyse und den Entwurf von kleinerer Bedeutung als zum Beispiel Basisattribute oder actions. Sie können oft als Gleichungen mit Termen von anderen Attributen spezifiziert werden und benötigen keine Kontrollkomponenten.

Eine action ist eine Transformation, die Seiteneffekte auf das Zielobjekt oder auf andere Objekte innerhalb des erreichbaren Systems vom Zielobjekt hat. Eine action hat keine Zeitausdehnung (obwohl in Wirklichkeit jede Implementation einige Zeit benötigt). Da der Zustand eines Objektes durch deren Basisattribute und Links definiert ist, müssen alle actions als Modifizierungen von Basisattributen und Links definierbar sein. Da actions bei ihrem Auftreten durch den Zustand des Systems vor und nach der Aktion beschrieben werden können, ist eine Kontrollkomponente nicht notwendig. Für eine Aktion ist lediglich das Ergebnis, die es liefert, von Bedeutung, während der Zeitpunkt der Ausführung oder die Reihenfolge nicht relevant sind. Aktionen werden normalerweise von Prozessen im Funktionalen Modell abgeleitet.

Eine activity ist eine Operation zu oder bei einem Objekt, die eine Zeitausdehnung hat, im Gegensatz zu queries und actions, die als augenblicklich angesehen werden. Eine action hat unvermeidlich Seiteneffekte, da es allein schon eine Zeitausdehnung hat. Activities haben generell nur einen Sinn für actors, die Objekte von allein generieren können. Die genauen Informationen einer activity werden sowohl im Dynamischen als auch im Funktionalen Modell spezifiziert, daher kann eine activity nicht nur als eine Transformation betrachtet werden.

Eine Spezifikation besteht aus mehreren Elementen: dem Funktionsnamen, den Eingaben und den Ausgaben, den Transformationen zu bestimmten Werten und constraints. Constraints zeigen die Beziehungen zwischen zwei Objekten, die während der gleichen Zeit arbeiten, oder zwischen verschiedenen Werten vom gleichem Objekt zu unterschiedlichen Zeiten. Constraints können sowohl als reine Funktion als auch partielle Funktion ausgedrückt werden. Constraints können in allen drei Teilmodellen auftreten. Objektconstraints spezifizieren partielle oder auch totale Abhängigkeiten zu anderen Objekten. Dynamische constraints spezifizieren Beziehungen zwischen Zuständen oder events von verschiedenen Objekten. Funktionale constraints spezifizieren Einschränkungen auf verschiedene Operationen. Ein zeitweiliges constraint zwischen Werten eines Objektes wird oft als Invariante bezeichnet.

2.4 Fallstudien

Im folgenden hat sich das Projekt mit der Erarbeitung von Fallstudien auseinandergesetzt. Wie bereits erwähnt dienen diese Szenarien der Vertiefung des angeeigneten Wissens und zusätzlich dem späteren Testen des fertigen Programms deSIGN.

2.4.1 Autovermietung

2.4.1.1 Einleitung

Um sich mit der Beschreibungsweise und der Notation des Modellierungskonzeptes von OMT (Object Modeling Technique) vertraut zu machen, wurde das Beispielmodell „Autovermietung“ mit Hilfe eben dieser Beschreibungsmethoden modelliert und formuliert. Die Kenntnis der Notation wird als gegeben vorausgesetzt. Das Beispielmodell „Autovermietung“ soll die Abläufe und Zusammenhänge, die bei der Vermietung eines Autos auftreten, veranschaulichen. Dabei soll das Verhalten der einzelnen beteiligten Objekte (Kunde, Auto, etc.) durch die verschiedenen Modelle exakt beschrieben werden.

2.4.1.2 Objektmodell

Das Objektmodell beschreibt die Vermietung mit ihren Komponenten: Personal, Kunde, Konto, Vertrag, Fahrzeug aus statischer Sicht, die die Struktur und den Aufbau des Modells erkennen läßt und die Beziehungen zwischen den einzelnen Komponenten aufdeckt (Abbildung 2.64).

Personal und Kunde haben zum Teil gleiche Attribute. Diese wurden im Objekt Person (Generalisierung) zusammengefast und können von hier „vererbt“ werden. Der Kunde steht mit dem Personal in einem Dialog, in dem der Kunde seine Wünsche mitteilt. Außerdem hat der Kunde ein Konto, welches anschließend mit den Leihkosten belastet wird. Das Personal sucht anhand der Wünsche des Kunden ein Fahrzeug aus dem Fuhrpark aus (Methode SucheFahrzeugNachWünschen). Der Vertrag gilt speziell für ein Fahrzeug und einen Kunden. Jedoch kann ein Kunde mehrere Verträge abgeschlossen haben.

2.4.1.3 Dynamisches Modell

Das dynamische Modell beschreibt das zeitliche Verhalten der einzelnen beteiligten Komponenten mit Hilfe eines Zustandsdiagramm, wobei die Zustandsüberführungen mit verschiedenen Ereignissen, Bedingungen und Nachfolgeereignisse bei Bedarf genauer spezifiziert werden können (Abbildung 2.65).

Möchte ein Kunde ein Fahrzeug ausleihen, geht er in eine Filiale einer Autovermietung und nimmt den Dialog mit dem Personal auf. (Bis zu diesem Zeitpunkt befindet sich unsere Fallstudie im Idle-Zustand). Der Kunde äußert so lange seine Wünsche, bis ein entsprechendes Fahrzeug gefunden wurde und der Kunde damit einverstanden ist. Anschließend wird der Vertrag aufgesetzt und dem

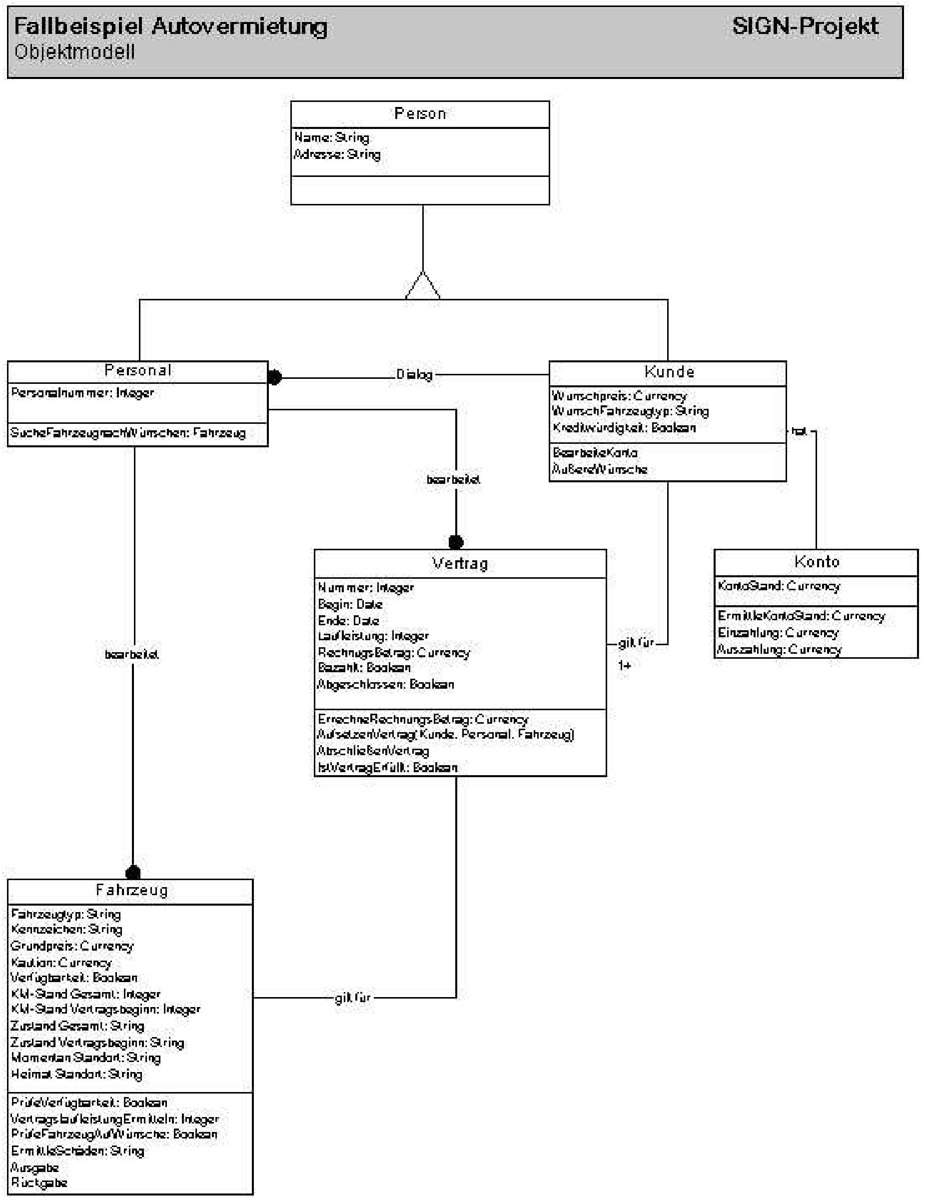


Abbildung 2.64:

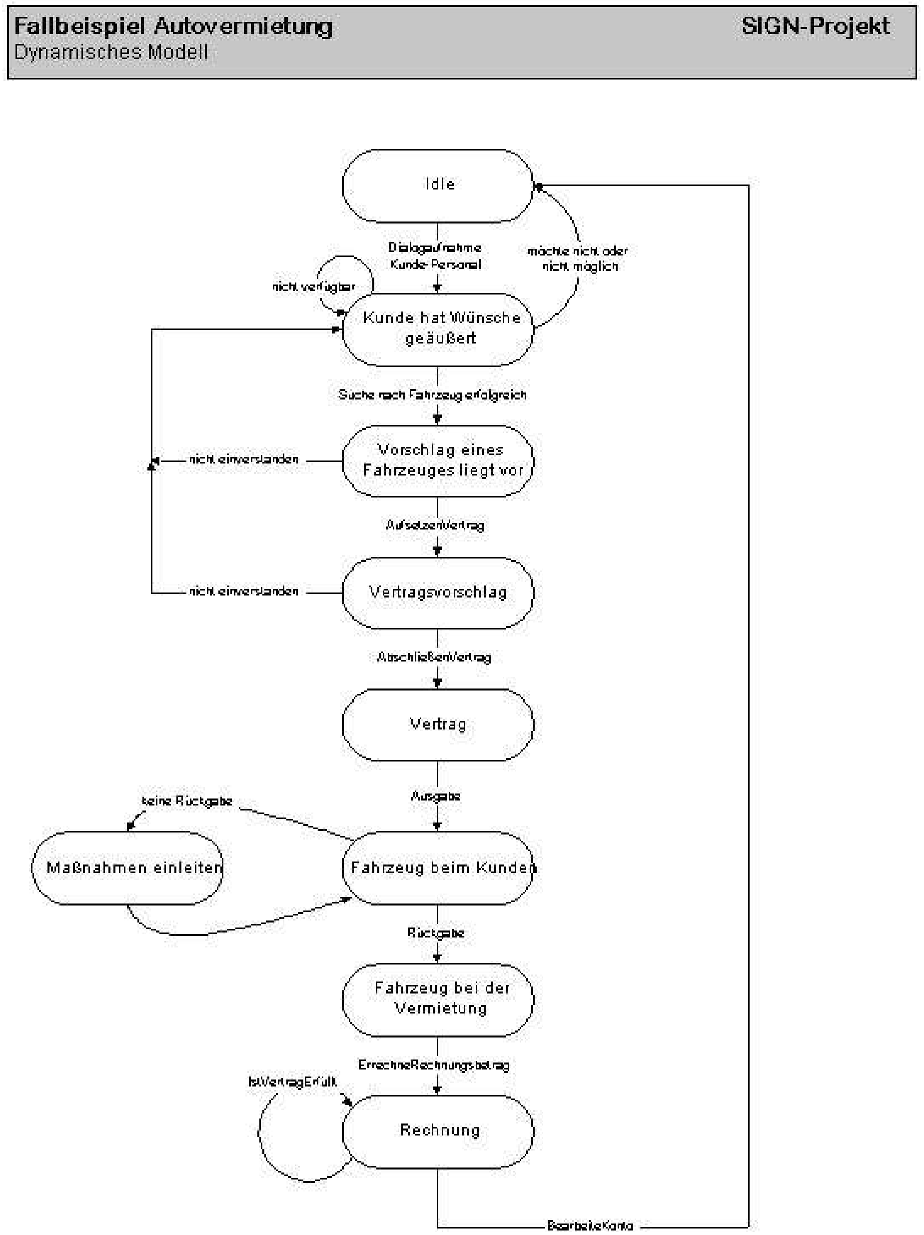


Abbildung 2.65:

Kunden das Auto ausgehändigt. In diesem Zustand wird so lange verblieben, bis eines der folgenden Ereignisse eintritt: entweder erfolgt die Rückgabe des Fahrzeuges fristgerecht, oder es werden nach Ablauf der Vertragsdauer Maßnahmen, die hier nicht näher erwähnt werden sollen, eingeleitet. Ist das Fahrzeug wieder vor Ort, wird die Rechnung erstellt und das Konto des Kunden belastet. Anschließend wird wieder in den Idle-Zustand übergegangen.

2.4.1.4 Funktionales Modell

Das Funktionale Modell beschreibt die Datenwerttransformation und somit den Datenfluß von und zu verschiedenen Prozessen. Anhand dieses Diagramms läßt sich der Ursprung der Eingabedaten zurückverfolgen bzw. die Erzeugung bestimmter Ausgabedaten von Prozessen erkennen. Eine zeitliche Aussage der Datenerzeugung und des Flusses macht das Funktionale Modell jedoch nicht (xref target=fimgAuto3" type=figure").

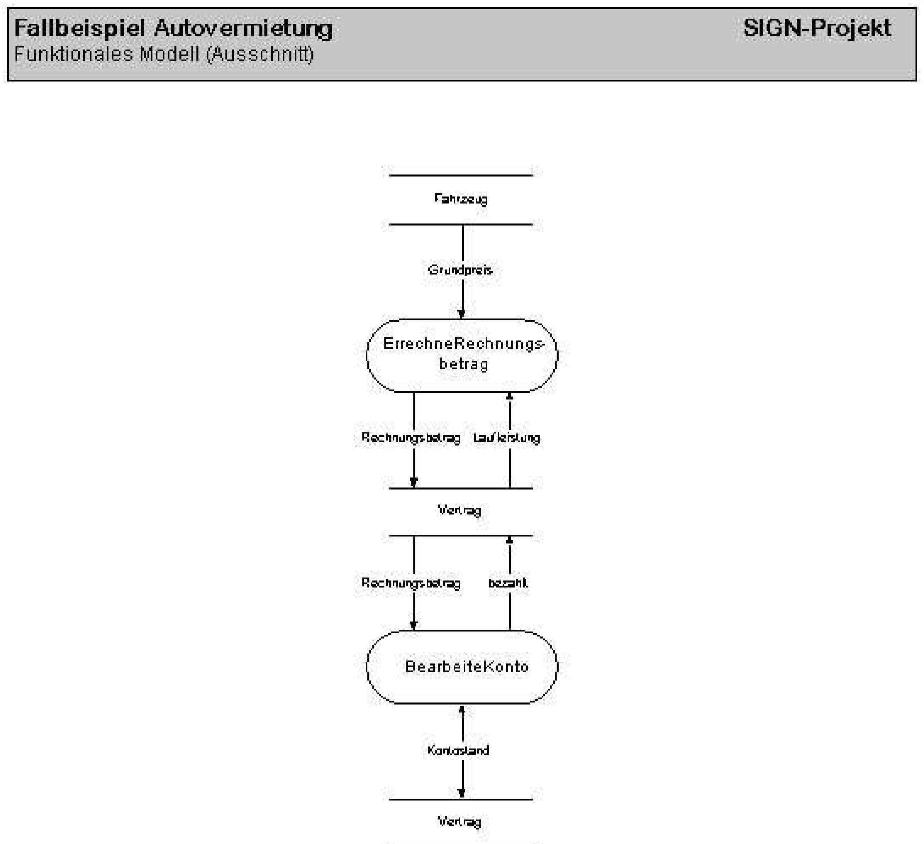


Abbildung 2.66:

Da das komplette Fallbeispiel zu umfangreich ist, wird hier nur ein Ausschnitt der Rechnungslegung und Bezahlung vorgestellt. Der Prozeß ErrechneRechnungsBetrag erhält als Eingabe den Grundpreis des Fahrzeuges (als Datenspeicher). Als Resultat erhält man den Rechnungsbetrag, der in den Vertrag übernommen wird. Wiederum wird aus dem Datenspeicher Vertrag die Laufleistung entnommen, die ihrerseits den Rechnungsbetrag beeinflußt. Der Prozeß BearbeiteKonto holt sich seine Eingabedaten (Rechnungsbetrag) aus dem Datenspeicher Vertrag. Nachdem das Konto erfolgreich manipuliert wurde, wird der Status bezahlt im Vertrag gesetzt und der Kontostand abgeglichen.

2.4.1.5 Fazit

Wie auch in den anderen Fallbeispielen gab es mehrere Möglichkeiten der Modellierung. Die mögliche Umsetzung hängt sehr stark von der Betrachtungsweise des Entwicklers ab. Je komplexer die Aufgabe, um so mehr mögliche Varianten gibt es. Somit steht diese Modellierungsvariante nur als Beispiel für eine mögliche Umsetzung der Autovermietung nach OMT. Außerdem hat das dynamische Modell der Autovermietung einen sehr starken sequentiellen Charakter und schöpft somit die Stärken von OMT kaum aus. Am schwierigsten gestaltete sich die Umsetzung des funktionalen Modells, da dieser Teil der OMT-Spezifikation etwas unklar erscheint.

2.4.2 Garagentor

2.4.2.1 Einleitung

Das Beispielmodell „Garagentor" soll ein automatisches Garagentor, das aus einem Motor, dem Tor und einem einfachen Schalter besteht, beschreiben. Dabei soll das automatische Verhalten der einzelnen Komponenten durch das Modell exakt beschrieben werden.

2.4.2.2 Objektmodell

Das Objektmodell beschreibt das Garagentor mit dessen drei Komponenten (dem Schalter, dem Tor und dem Motor) aus dem das System besteht in einer statischen Sicht, die die Struktur und den Aufbau des Modells erkennen läßt (Abbildung 2.67).

Der Motor hat im Modell zwei Attribute: „Richtung" mit den möglichen Werten 1 oder -1 und MState mit den Werten an oder aus. Diese sollen den Motor „einstellen", so daß der Motor richtig arbeiten kann. Das Attribut Richtung soll den Motor einstellen, ob dieser das Tor öffnen oder schließen soll. „MState" gibt den aktuellen Zustand des Motors, ob dieser arbeitet oder nicht, wieder. Die Prozedur „working" gibt an, daß der Motor „handeln" kann und zwar arbeiten (mehr dazu im FM (Funktionales Modell)).

Das Tor besitzt lediglich Attribute, die die Torposition beschreiben. Während „Zustand" nur wiedergeben soll, ob das Tor vollständig geöffnet oder geschlossen ist, soll „TPositon" die aktuelle und „wirkliche" Torposition für das Modell simulieren.

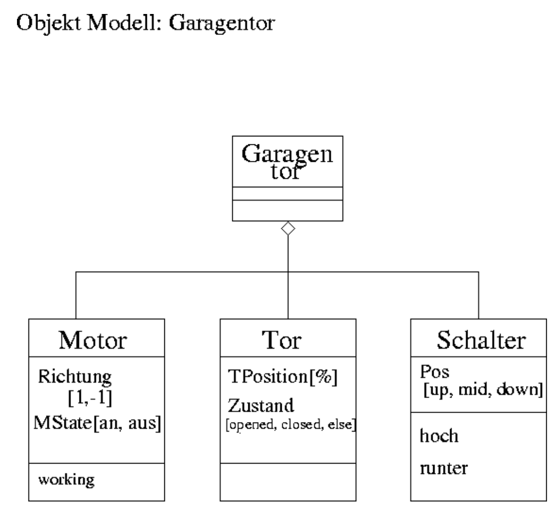


Abbildung 2.67:

Der Schalter besitzt das Attribut „Pos“ mit den Werten up, mid und down, die für dieses Bespielmodell die wirkliche Schalterposition simulieren soll. Die Prozeduren „hoch“ und „runter“ sollen das beabsichtigte Handeln der anderen Komponenten einstellen und signalisieren.

Der Benutzer des Garagentors, der in diesem Modell nur den Schalter bedient, ist in diesem Beispiel nicht modelliert. Seine Einwirkung auf das System, also das Betätigen des Schalters, wird mit dem Setzen des richtigen Wertes vom Attribut „Pos“ vorausgesetzt.

2.4.2.3 Dynamisches Modell

Das dynamische Modell beschreibt das zeitliche Verhalten der einzelnen Komponenten mit Hilfe eines Zustandsdiagramm wobei die Zustandsüberführungen mit verschiedenen Ereignissen, Bedingungen und Nachfolgeereignissen genauer spezialisiert werden können.

Schalter

Der Schalter hat drei mögliche Zustände; für die gültigen Zustandsüberführungen ist lediglich die Bedingung abhängig von dem Wert des Attributs „Pos“. In jedem der Zustände werden bestimmte Aktivitäten initialisiert. Der Schalter setzt je nach Zustand das Motorattribut „MState“ und ruft die eigenen Prozeduren „hoch“ und „runter“ auf (Abbildung 2.68).

Obwohl bis jetzt nicht ersichtlich ist es durchaus legitim, daß Objekte auf Attribute von anderen Objekten zugreifen und sogar manipulieren können. Dies wird im Funktionalen Modell im Datenflußdiagramm besser deutlich.

Dynamisches Modell: Schalter

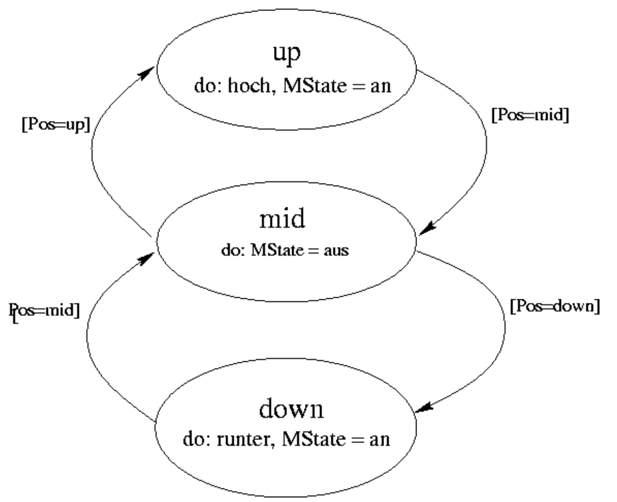


Abbildung 2.68:

Dynamisches Modell: Tor

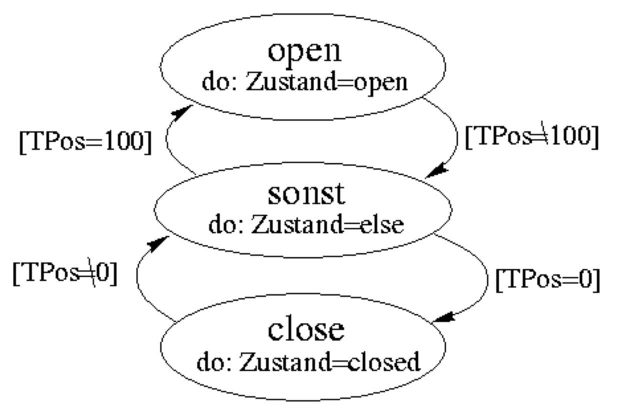


Abbildung 2.69:

Tor

Das Tor besitzt drei Zustände, die die Torposition wiedergeben. Durch die Bedingungen des Wertes vom Attribut „TPos“ werden die gültigen Zustandsüberführungen angegeben. Die einzige Aktivität, die das Tor macht, ist die Aktualisierung des eigenen Attributs „Zustand“ (Abbildung 2.69).

Motor

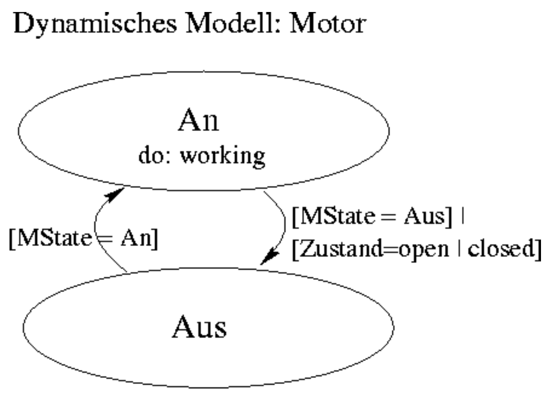


Abbildung 2.70:

Der Motor besitzt zwei Zustände: den Ruhezustand Aus und den arbeitenden Zustand An, in diesem die eigene Prozedur „working“ aufgerufen werden soll. Für die Zustandsüberführungen sind die Bedingungen abhängig vom Wert der Attribute „MState“, die vom Schalter gesetzt und geändert werden, und vom Attribute „Zustand“, des Tores (Abbildung 2.70).

Der Motor „beobachtet“ also seine wichtigen Attribute, die der Funktion eines Events entsprechen sollen, und reagiert dementsprechend auf die „Meldungen“ (eigentlich Bedingungen).

2.4.2.4 Funktionales Modell

Das Funktionale Modell beschreibt den Datenfluß von verschiedenen Prozessen. Anhand dieses Diagrammes läßt sich der Ursprung der Eingabedaten zurückverfolgen bzw. die Erzeugung bestimmter Ausgabedaten von Prozessen erkennen. Eine zeitliche Aussage der Datenerzeugung und des Flusses macht das Funktionale Modell jedoch nicht.

Schalter

Der Schalter besitzt zwei Prozeduren „hoch“ und „runter“, die dem Motor die Daten (in Form von Attributen) Richtung und MState für die Arbeitsanweisungen manipuliert. Damit der Schalter diese Attribute überhaupt zur Verfügung hat (Richtung und MState sind Attribute von Motor und nicht von Schalter) werden diese vom Motor an den Schalter übergeben (Abbildung 2.71).

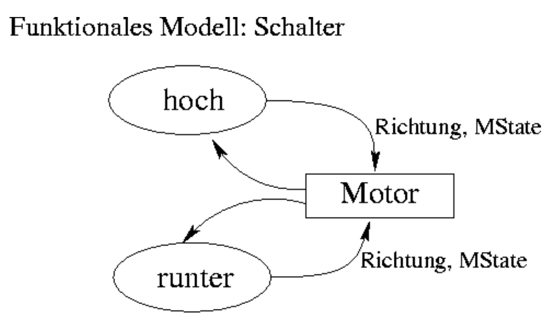


Abbildung 2.71:

Die genaue Beschreibung der Prozeduren von „hoch“ und „runter“ werden später näher erläutert.

Motor

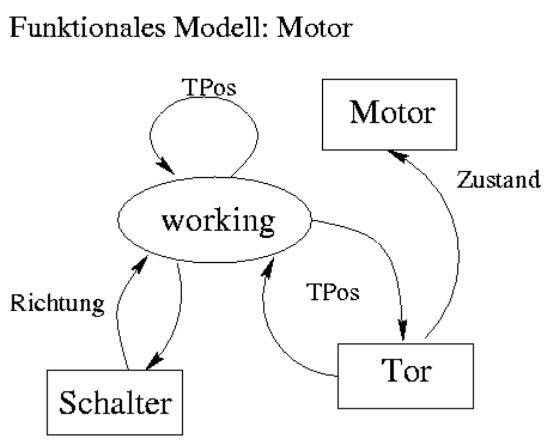


Abbildung 2.72:

Die Prozedur „working“ ist die einzige Prozedur, die der Motor hat. Die Prozedur „working“ übergibt das Attribut Richtung dem Schalter und bekommt diesen evtl. geändert zurück. Das Attribut „TPos(ition)“ wird von Tor bereitgestellt und von working wiedergegeben. Außerdem übergibt die Prozedur working TPos an sich selbst wieder (Abbildung 2.72).

Das Attribut „Zustand“ von Tor wird ebenfalls dem Motor übergeben (d.h. alle Daten, die von außen der Prozedur working gegeben werden auch). Da jedoch keine Prozedur mit diesem Attribut arbeitet, wird in diesem Datenflußiagramm der Motor extra als Konsument dargestellt.

Beschreibung der Prozesse

Zur Beschreibung der Prozesse ist jede „geeignete“ (nach OMT) Sprache zu verwenden. Da in diesem Beispielmodell keine größeren und komplexeren Prozesse auftauchen, werden für diesen Zweck einfache und hoffentlich intuitiv verständliche Gleichungen als Berechungsbeschreibung verwendet.

working: TPos = TPos + Richtung;

hoch: Richtung = 1 (MState = an);

runter: Richtung = -1 (MState = aus);

Obwohl streng genommen im OMT im dynamischen Modell für do-Anweisungen benannte Aktivitäten (activities) folgen, wird stattdessen in manchen Fällen die direkte Beschreibung der Handlung in dieser Modellierung in den Zuständen geschrieben. Diese Beschreibung könnte oder besser sollte jedoch auch völlig gekapselt in der Prozeßbeschreibung auftauchen. Da in diesem Fall viele Attribute (MState, Zustand...) eine Ereignis- und Meldungsfunktion für das zeitliche Verhalten besitzten, erschien es angebrachter, diese direkt im dynamischen Modell mit einzubeziehen.

2.4.2.5 Fazit

Die erste Feststellung in dieser „Mini“-Fallstudie war, obwohl das Szenario und das Modell genau beschrieben war und für jeden Beteiligten die Funktionalität und Arbeitsweise klar war, ergaben sich viele Modellierungsmöglichkeiten die dieses Bespielmodell angemessen in der OMT-Notation beschrieben hätten.

Dieses hier vorgeschlagene Modell soll nur als Anregung und Modellierungvorschlag gelten. Es soll auch keine „elegante“ Modellierungsmethode demonstrieren.

In diesem Modell ist lediglich anzumerken, daß die komplette Steuerung des dynamischen Verhaltens nicht durch die von OMT vorgegebenen Events abgehandelt wird, sondern durch Attribute verwirklicht werden, die durch „Beobachten“ von den angesprochenden Objekten, bei bestimmten Bedingungen (durch Werte der Attribute) ein entsprechendes Handeln steuern sollen. Damit wird in diesem Modellierungskonzept die Kommunikation der einzelnen Komponenten deutlicher, was bei einem solch kleinem und relativ leicht überschaubarem Beispielmodell noch seinen Sinn und Zweck erfüllt, jedoch bei größeren und komplexeren Modellen eventuell nicht angesbracht ist.

2.4.3 Konferenzorganisation

2.4.3.1 Einleitung

Die Fallstudie zu diesem Beispielmodell behandelt ein wissensbasiertes Informationssystem zur Organisation von Tagungen. Es dient zur Unterstützung für die Planung und Durchführung einer Tagung. Im Rahmen unserer Beschreibungen der Beispielmodelle ist diese Tagungsorganisation und der Ablauf einer Tagung die Grundlage für unsere Untersuchungen und Studien.

Die Modelle der OMT-Modellierungstechnik wurden konkret auf das Beispiel der Konferenzorganisation angewandt. Sie werden erläutert und ihre Funktionen, Konstrukte und Bedeutungen beschrieben. Auf die Syntax der Notation wird nicht im einzelnen eingegangen, da dies bereits behandelt wurde und bekannt ist. Die Grundlage für die Entwicklung der Modelle bietet das Buch „Object-Oriented Modeling and Design“ von Rumbaugh, Blaha, Premerlani, Eddy, Lorensen.

Das Objektmodell stellt hierbei praktisch einen Ausschnitt aus der realen Welt dar. Es ist nur bedingt ein Anwendungssystem und entspricht einem Klassendiagramm, da es keine Operationen enthält. Daher ist es problematisch auf dieser Grundlage das dynamische und funktionale Modell mit entsprechenden Zuständen und Prozessen zu entwerfen. Demnach beinhaltet das Objektmodell die umfassende Tagungsorganisation mit den zugehörigen Komponenten und Abläufen. Das dynamische und funktionale Modell enthalten einen Ausschnitt aus dem Objektmodell und beschreiben damit einen Teil des Tagungsablaufs. Es wird hierbei ein Szenario dargestellt, das die konkreten Anforderungen modelliert. Das Szenario behandelt die Teilnehmerverwaltung der Tagungsbesucher und beinhaltet die Vorgänge von den Einladungen des Organisationskomitees bis zur Abmeldung der Tagungsbesucher.

2.4.3.2 Das Objektmodell

In diesem Modell werden die statischen Strukturen der Objekte in einem System, sowie deren Beziehungen beschrieben. Zunächst gibt es eine Tagung, die einen zugehörigen Titel hat und an einem bestimmten Ort stattfindet.(Dort befindet sich ein Büro mit Telefon.) Zur Tagung gehören Gebühren, die zu zahlen sind. Deren Verbuchung wird vom Organisationskomitee übernommen. Außerdem gibt es einen Tagungsleiter. Mehrere Tagungen können von einer oder mehreren Organisationen veranstaltet werden. Eine Tagung hat außerdem einen Tagungsband, der nach Beendigung der Tagung zusammengestellt wird. Der Band enthält die Manuskripte der Autoren, die zu einem bestimmten Termin die Endfassung zusammengestellt haben müssen. Zudem gibt es Herausgeber und Verlage (xref target=fimgKonf1” type=figure»).

Das Organisations- und Programmkomitee sind Teil der Tagung. Das Organisationskomitee lädt Personen ein, die sich zur Tagungsteilnahme anmelden, und einen Beitrag zahlen müssen. Außerdem wird für sie ein Hotel reserviert. Nach Beendigung der Tagung müssen sie sich abmelden. Diese Vorgänge machen sie zu Tagungsteilnehmern, die aus dem In- oder Ausland kommen können. Zur Tagungsorganisation gehören immer bestimmte Personen, die Leiter oder Mitglieder sind und als Unterklassen durch die Generalisierung beschrieben werden.

Zum Organisationskomitee gehören Organisationskomiteemitglieder, genauso wie zum Programmkomitee Programmkomiteemitglieder (Aggregationen) gehören. Das Programmkomitee, das das Autorenprogramm zusammenstellt, hat auch einen Leiter. Daher wurde ein Constraint zwischen den Assoziationen modelliert. Die ternäre Assoziation besagt, daß die Programmkomiteemitglieder vom Tagungs- und Programmkomiteeleiter gewählt werden. Der Programmkomiteeleiter beauftragt ein Gutachten, das von den Mitgliedern erstellt werden soll. Es wird Fachgebieten zugeordnet und soll autorenunabhängig sein. Ein

Autor kann 0...n Manuskripte einreichen. Er bekommt eine Zu- oder Absage, muß die Manuskripte zu einem vorgegebenen Termin mit 3 Kopien einreichen. Außerdem behandelt ein Manuskript ein Fachgebiet und muß als Endfassung für den Tagungsband vorliegen.

Die Programmkomiteesitzung findet zu einem bestimmten Termin statt. Dort findet die Vortragauswahl statt. Es wird ein Programm erstellt, das mehrere Tagungssitzungen enthält (Aggregation). Jede Sitzung hat einen Sitzungsleiter, der ein Mitglied des Programmkomitees ist. Die Tagungssitzungen bestehen aus mehreren eingereichten Vorträgen, und jeweils einem eingeladenen Vortrag (von Wissenschaftlern gehalten). Sie sind Unterklassen der Oberklasse Vortrag. Ein eingereichter Vortrag basiert auf einem Manuskript eines Autors.

2.4.3.3 Dynamisches Modell

Das dynamische Modell beschreibt die Aspekte eines Systems, das sich zeitlich ändert. Das dynamische Modell wird benutzt, um die Kontrollaspekte eines Systems zu spezifizieren und zu implementieren. Innerhalb unserer Fallstudie wird nun das Szenario der Teilnehmerverwaltung beschrieben. Es bildet einen Ausschnitt aus dem Objektmodell, der einige ausgewählte Klassen beinhaltet und deren Assoziationen behandelt. Die folgenden Klassen sind in dem Modell eingebaut: Organisationskomitee, Tagungsteilnahme, Tagungsteilnehmer. Das Szenario soll beschreiben, wie die Verwaltung bzw. der Ablauf einer Tagungsteilnahme durchgeführt wird. Es beginnt mit der Einladung zur Tagung, die vom Organisationskomitee verschickt wird und endet mit der Abmeldung des Tagungsteilnehmers nach Beendigung der Tagung (xref target=fimgKonf2" type=figure»). Im einzelnen sind folgende Punkte enthalten:

Szenario Teilnehmerverwaltung:

- Organisationskomitee lädt Teilnehmer ein (Versand von Einladungen)
- Teilnehmer zahlt Teilnahmegebühr (falls nicht: Mahndatum, Mahnstufe)
- Verbuchung der eingegangenen Zahlungsgebühr
- Anmeldung
- Hotelreservierung
- Abmeldung (nach Beendigung der Tagung).

Auf dieser Grundlage wird das dynamische Modell entwickelt: Den Oberzustand bildet die Tagungsteilnahme. Als Startzustand wird der Zustand eingeladen erreicht, denn dadurch wird eine Tagungsteilnahme erst ermöglicht. Als nächstes erfolgt der Übergang in den Zustand angemeldet. Dieser wird durch das Event anmelden erreicht und eingenommen. Außerdem erfolgt das Event, das das Verbuchen des Betrages enthält (buchen (betrag)). Es wird also der Zahlbetrag bezahlt, und dadurch die Anmeldung ermöglicht.

Danach befinden wir uns im Unterzustand angemeldet. Zu diesem Zustand gehört die Hotelreservierung. Als Startzustand befinden wir uns zunächst im

OBJEKT-MODELL (KLASSENDIAGRAMM)

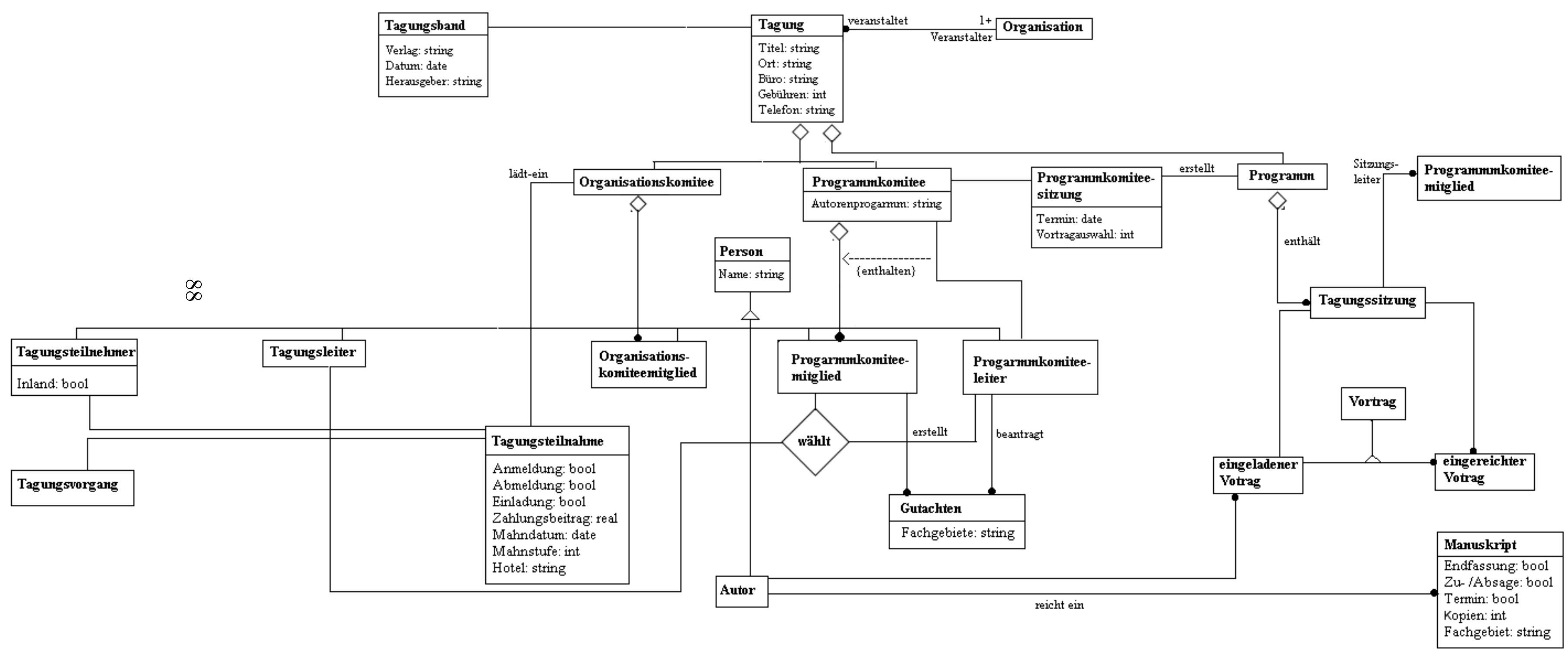


Abbildung 2.73:

Zustand nicht_reserviert. Durch das Event hotel_reservieren erreicht man den Zustand reserviert. Durch hotel_abmelden kommt man in den Ausgangszustand zurück. Es ist auch möglich, daß man sich in angemeldetem Zustand für weitere Tagungen anmelden kann, und damit an mehreren Tagungen teilnehmen kann. Dies wird durch die Transition von angemeldet auf sich selbst modelliert. Durch das Event abmelden kommt man schließlich in den Zustand abgemeldet und hat dann nach Beendigung der Tagung wieder die Möglichkeit zu einer anderen eingeladen zu werden.

DYNAMISCHES MODELL (Teilnehmerverwaltung)

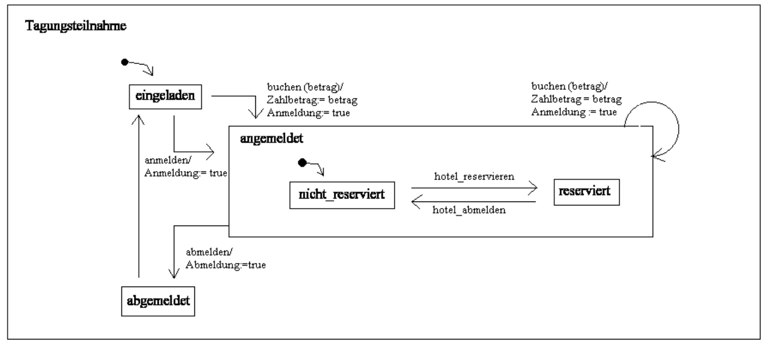


Abbildung 2.74:

2.4.3.4 Funktionales Modell

Das funktionale Modell beschreibt die Datenwerttransformationen in einem System. Das Szenario, das wir im dynamischen Modell verwendet haben, übertragen wir nun auf das funktionale Modell. Statt Zustände und Events benutzen wir jetzt unter anderem Prozesse und Datenflüsse. Das Prinzip bleibt jedoch das gleiche. Person als Data Store „fließt“ grundsätzlich zum Prozeß Organisationskomitee:: einladen, da es sich um Personen mit Namen und entsprechenden Eigenschaften handelt. Von dort resultiert ein Datenfluß in den Data Store Tagungsteilnehmer. Dies wird durch die :Tagungsteilnahme erreicht, das heißt eine Person wird durch die Tagungsteilnahme zum Tagungsteilnehmer. Es gilt hierbei noch auszudrücken, wann eine Tagungsteilnahme gewährleistet werden kann. Daher erfolgt der Datenfluß von Organisationskomitee:: einladen zum Data Store Tagungsteilnahme. Als nächstes werden die Prozesse für die Tagungsteilnahme beschrieben. Die folgenden vier Prozesse sind dazu modelliert worden:

Tagungsteilnahme:: anmelden ist die Voraussetzung, um an der Tagung teilzunehmen. Sie leitet die Teilnahme ein. Entsprechend verhält es sich mit Tagungsteilnahme:: abmelden. Der Prozeß bildet den Abschluß des Teilnehmerverwaltungsvorgangs. Eingeladen, angemeldet und abgemeldet bilden die „Statusgrundlage“ für die Teilnehmerverwaltung.

Tagungsteilnahme:: buchen bedeutet das Verbuchen des Zahlungsbeitrages. Der Datenfluß enthält den Betrag in Höhe eines Gleitkomma-Wertes. Tagungsteilnahme:: hotel_reservieren ist ein weiterer Prozeß. Als Datenfluß ist ein konkretes Hotel beschrieben, das für die Dauer der Tagung zur Verfügung gestellt wird. Mit Hilfe dieser Komponenten wird die Tagungsteilnahme ermöglicht.

Abschließend bleibt festzustellen, daß die drei Modelle orthogonale Teile der Beschreibung eines kompletten Systems sind und Querverbindungen aufzeigen. Das Objektmodell ist dabei das fundamentalste, weil es erforderlich ist, zu beschreiben, was abläuft und sich ändert bzw. transformiert wird bevor man beschreibt, wann oder wie es sich ändert (xref target=fimgKonf3" type=figure»).

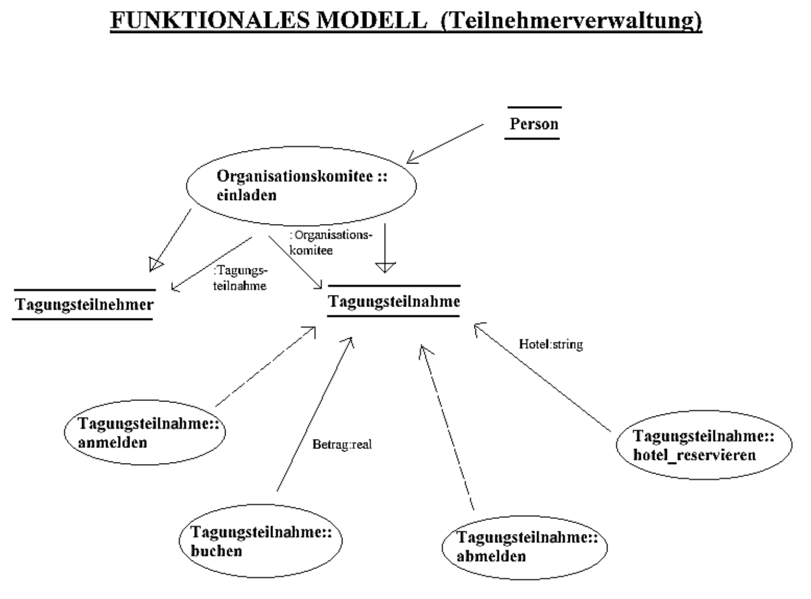


Abbildung 2.75:

2.4.4 Netztopologie

2.4.4.1 Einleitung

Im Rahmen unseres Projektes „SIGN“ (Software Implementation through Pictures) haben wir zum Umgang mit dem SIGN-Editor eine Fallstudie durchgeführt. Hierbei haben wir als Beispielmodell die grafische Darstellung der Netztopologie [SSC$^+$96] erarbeitet. Dieses ermöglicht dem Benutzer, im Gegensatz zur textuellen Darstellung, sich schnell einen Überblick über die aktuelle Konfiguration eines Netzes zu verschaffen. Auf der Basis von bereits vorhandenen

Informationen (der Host-Tabelle und Netzwerk-Tabelle) soll eine statische Map des lokalen Netzes als Grafik dargestellt werden. Die Dateneingabe kann hierbei auf zwei verschiedenen Wegen erfolgen. Man kann entweder die Host-und Netzwerkeinträge des lokalen Netzes auslesen, oder aus gegebenen ASCII-Dateien lesen. Für die grafische Darstellung wird das gegebene Netz so visualisiert, daß die Subnetze mit ihren Namen, sowie deren Verbindungen durch Gateways und die Namen dieser Gateways dargestellt werden können. In enger Anlehnung an OMT (Object Modeling Technique) wurden für das Beispielmodell alle drei Modellierungskonzepte (Objektmodellierung, Dynamische Modellierung, Funktionale Modellierung) von OMT erarbeitet. Für die Darstellung der einzelnen Modellkonzepte wurden Diagramme herangezogen.

2.4.4.2 Das Objektmodell

Das erste Modell „Objektmodell" beschreibt die statische Struktur der Objekte in einem System und ihre Relationen. Das Objektmodell enthält Objektdiagramme (Graphen von Objektklassen als Knoten und Relationen zwischen Klassen als Kanten). Das folgende Objektmodell für die Netztopologie besteht aus fünf Modulen. Das erste Modul „Data" beinhaltet die Datenklassen des Programms und besteht aus den Klassen CTableEntry, CHostEntry und CSubnetEntry, das Modul „Graphics" beinhaltet die Grafikklassen des Programms zur grafischen Ausgabe und besteht aus den Klassen CGraphObj, CGraphHostEntry, CGraphSubnetEntry und CCanvas. Das Modul „GTools" ist eine allgemeine Schnittstelle zum benutzten grafischen Toolkit. Alle speziellen Grafikbefehle werden in diesem Modul gekapselt. Dieses Modul beinhaltet die Klasse CWindow. Das Modul „Application" beinhaltet eine Klasse CNetViewApp zur Steuerung und zum Beenden der Anwendung. Es stellt eine zentrale Schnittstelle dar. Das Modul „Errors" beinhaltet die Klasse CErrors in der die Fehlermeldungen des Programms gekapselt werden (Abbildung 2.76).

Modul Data

Die Klasse CTableEntry des Moduls „Data" ist die Basisklasse zur Speicherung der einzulesenden Host- und Subnetzeinträge. Hier werden die grundlegenden Eigenschaften dieser Einträge definiert, wobei als Datenelemente die IP-Adresse (IP_Address), der einer IP-Adresse zugeordnete Name (Name), „inoffizielle Namen" (Nicknames) dieser IP-Adresse, sowie ein Datenelement Visible, das bei einer Ausgabeoperation die Sichtbarkeit eines solchen Eintrages bestimmt definiert sind. Um einen direkten Zugriff auf die Datenelemente zu unterbinden, können diese nur über entsprechende Methoden manipuliert oder ausgelesen werden. Das Datenelement IP_Address kann z.ß. mit den Methoden getIP_Address und getIP_AddressStr ausgelesen werden. Im ersten Fall wird die IP-Adresse als 32-Bit Integer, im zweiten Fall als String zurückgegeben. Für das Auslesen des Datenelementes Name wird die Methode getName verwendet, während die Anzahl der inoffiziellen Namen (Nicknames) mit der Methode getNrOfNickname ermittelt werden kann. Zum Auslesen des Sichtbarkeitsstatus der Klasse wird die Methode isVisible verwendet, die den Wahrheitswert true (false) liefert, wenn eine Instanz dieser Klasse (nicht) als sichtbar markiert ist.

Zur Modifikation der Datenelemente existieren die Methoden setIP_Address, setName, setVisibility und addNickname.

Die Klassen CHostEntry und CSubnetEntry sind Unterklassen der Klasse CTableEntry und wurden direkt von dieser abgeleitet. Die Klasse CHostEntry dient zum Aufnehmen von Einträgen, die aus einer Host-Tabellen- Datei oder direkt aus der Host-Tabelle des lokalen Netzes ausgelesen wurden. Die Klasse CSubnetEntry dient analog dazu zum Aufnehmen von Einträgen, die aus einer Netzwerk-Tabellen-Datei oder direkt aus der Netzwerk-Tabelle des lokalen Netzes ausgelesen wurden. Das Datenelement Gateway der Klasse CHostEntry dient dazu, den Hosteintrag, der in einer Instanz dieser Klasse abgelegt wird, als Gateway zu kennzeichnen, sofern der Host über diese Eigenschaft verfügt. Das Datenelement SubnetConnections stellt die Verbindungen des Hosts mit den Subnetzen dar, die er erreicht. Demnach dient die Methode isGateway zum Auslesen des Gateway- Status eines Hosteintrages, während man über die Methode setGateway den Gateway-Status eines Hosteintrages modifizieren kann. Mit addSubnetConnection kann bei einem als Gateway fungierenden Host die Verbindung zu einem mit diesem Gateway verbundenen Subnetz im Datenelement SubnetConnections hinzugefügt werden. Die Datenelemente NrOfHosts und NrOfGateways der Klasse CSubnetEntry beinhalten die Anzahl der Hosts und Gateways in dem von einer Instanz dieser Klasse repräsentierten Subnetz. Das Datenelement HostConnections verwaltet sämtliche dem Subnetz zugeordneten Hosteinträge. Zum Auslesen der Datenelemente NrOfHosts und NrOfGateways werden die Methoden getNrOfHosts und getNrOfGateways verwendet. Das Datenelement addHostConnection dient dazu, dem Datenelement HostConnections eine weitere Verbindung hinzuzufügen.

Modul Graphics

Klasse CGraphObj des Moduls „Graphics“ ist eine abstrakte Basisklasse, die einige für ein grafisches Objekt notwendige Eigenschaften definiert. Hierbei stehen die Datenelemente X, Y, W und H für Koordinaten der linken oberen Ecke, sowie Breite und Höhe eines grafischen Objektes, das durch die Klasse CGraphObj beschrieben wird. Zur Bildschirmausgabe dieses Grafikobjektes dient die Methode draw, während die Methode isInside testet, ob die als Parameter übergebene Koordinate sich innerhalb der Grenzen dieses Objektes befindet oder nicht. Die Methode setBounds setzt die Werte für obere linke Ecke, Breite und Höhe von CGraphObj und ist wie draw und isInside zunächst einmal nur abstrakt definiert. Die Klasse CGraphHostEntry vereinigt die Datenklasse CHostEntry und die abstrakte Grafikklasse CGraphObj zu einer Datenklasse mit Grafikfunktionalitäten. Dies ermöglicht eine einfachere Darstellung von Hosts auf dem Bildschirm. Hierbei werden alle Datenelemente von CHostEntry mit den Datenelementen von CGraphObj vereinigt. Weiterhin werden die Methoden draw und setBounds, die bereits als abstakte Methoden in CGraphObj vorgestellt wurden, mit Funktionalität versehen. Diese Methoden sorgen dann für die Darstellung der Klasse CGraphHostEntry auf dem Bildschirm, bzw. für das Setzen der Abmessungen dieser Klasse. Analog dazu vereinigt die Klasse CGraphSubnetEntry die Datenklasse CSubnetEntry und die abstrakte Grafikklasse CGraphObj zu einer Datenklasse mit Grafikfunktionalitäten. Dieses ermöglicht eine einfachere Darstellung von Subnetzen auf dem Bildschirm.

Hierbei werden die Datenelemente von CSubnetEntry mit den Datenelementen von CGraphObj vereinigt. Analog zu den Methoden der Klasse CGraphHostEntry werden auch hier die Methoden draw und setBounds, die bereits als abstrakte Methoden in CGraphObj vorgestellt wurden, mit Funktionalität versehen. Außerdem wird im Modul „Graphics“ die Zeichenfläche in der Klasse CCanvas gekapselt. CCanvas kümmert sich um den Aufbau , das Arrangieren und Zeichnen der grafischen Darstellung der Netztopologie. Die Klassen CGraphHostEntry und CGraphSubnetEntry stehen mit der Klasse CCanvas in einer „Teil-Ganzes-Beziehung“, wobei CCanvas sich hier die Host- und Subnetzeinträge für die grafische Darstellung holt. Die Methode arrange sorgt für den Aufbau und das Arrangieren der in der Netztopologie enthaltenen Daten. Die Methode draw veranlaßt die Ausgabe der Grafik im Grafikfenster.

Modul GTools

Die Klasse CWindow des Moduls „GTools“ ist die Motif-Schnittstelle der Anwendung. Die Motif- Routinen zur Erzeugung der notwendigen Grafikelemente werden in die Methoden von CWindow gekapselt. Die Datenelemente Kontext, Disp, GrWidget und WndGC werden von Motif zur Erzeugung der Applikations-Umgebung benötigt. Kontext bezeichnet die Applikations-Umgebung, während Disp verschiedene Informationen über das X-Windows-Display enthält. GrWidget steht für das Hauptfenster mit der zugehörigen Semantik. WndGC steht für den Grafik-Kontext des Hauptfensters. Jede Ausgabe geschieht mit Hilfe dieses Kontextes. Die Methode createMainMenu erzeugt das Hauptmenü der Anwendung, openPopupMenu erzeugt das Popup-Menu, das bei einem Klick mit der rechten Maustaste auf die Grafikumgebung angezeigt wird. openFileDialog sorgt für den Aufbau des Dialoges mit einer Dateilistbox, um die Netzwerk- und Host-Tabellen-Dateien einzulesen. Der ausgewählte Dateiname wird im Parameter Filename abgelegt. Die Methode getFilterString erzeugt das Dialog zum herausfiltern von Subnetzen. Zur Ausgabe von allgemeinen Informationen bzw. Fehlermeldungen auf dem Bildschirm dienen die Methoden showMessage und showErrorMessage. Die Methode eventLoop verleitet Motif zur zentralen Ereignisbehandlungsroutine. outText, line und rectangle können Text, Linienelemente und gefüllte Rechtecke auf dem Grafikbildschirm ausgeben, die zur Darstellung der Netztopologie benötigt werden.

Modul Application

Die Klasse CNetViewerApp des Moduls „Application“ ist die zentrale Klasse und erzeugt, steuert und beendet die Anwendung. Außerdem ist sie die Schnittstelle, die zwischen allen anderen Modulen vermittelt. Die Klasse CNetViewerApp muß als zentrale Schnittstelle Zugriff auf die wichtigsten Klasseninstanzen haben. So z.B. auf den Canvas (zum Zugriff auf die Zeichenfläche), Wnd (zum Zugriff auf das Grafikfenster), Hosts (zum Zugriff auf die Liste aller Hosteinträge) und Subnets (zum Zugriff auf die Liste alle Subnetzeinträge). Die Anwendung wird mit der Methode run gestartet. run wird unter anderem einen Aufruf von draw erzwingen, damit das Hauptfenster der Anwendung auf dem Bildschirm ausgegeben wird. Nun befindet sich die Anwendung in einer Endlosschleife, in der sie alle auftretenden Ereignisse verarbeitet. Dazu dient die Methode handleEvent. Die folgenden Methoden leiten Reaktionen auf die wichtigsten in der Anwendung beschriebenen Ereignisse ein. Die Methode openHostFile reagiert damit auf die Menüauswahl Datei - Lade Hostdatei und liest eine Datei mit Hosteinträgen ein. Die Methode openNetFile reagiert auf die Menüauswahl Datei - Lade Subnetzdatei und liest eine Datei mit Subnetzeinträgen ein. readLocalNet reagiert auf die Menüauswahl Datei - Lade Netzwerkinfos und liest die Host- und Subnetzinformationen aus dem lokalen Netz aus. closeApp beendet die Anwendung, nachdem der Menüpunkt Datei - Beenden gewählt wurde. Die Methode filter reagiert auf die Auswahl des Menüeintrages Bearbeiten - Filter und leitet eine Filterung der Bildschirmausgabe ein. Die Methode info reagiert auf den Menüeintrag Hilfe - Info und erzeugt ein Dialogfenster, in dem die Programminformationen angezeigt werden.

Errors Modul

In der Klasse CErrors des Moduls Errors werden sämtlichen Fehlercodes der dazugehörige Fehlertext zugeordnet. Mittels der Methode getErrorMessage wird dann zu dem entsprechenden Fehlercode die entsprechende Fehlermeldung ermittelt.

2.4.4.3 Dynamisches Modell

Das zweite Modell „Dynamisches Modell“ beschreibt die Aspekte eines Systems, die sich im Laufe der Zeit verändern. Das dynamische Modell wird zur Spezifizierung und Implementierung der Steuerungsaspekte eines Systems verwendet. Das dynamische Modell enthält Zustandsdiagramme („Ein Zustandsdiagramm ist ein Graph, dessen Knoten Zustände und dessen Linien Transitionen zwischen Zuständen sind, die durch Ereignisse ausgelöst werden“). Das folgende dynamische Modell zeigt ein Zustandsdiagramm für die Klasse CNetViewerApp des Objektmodells (Abbildung 2.77).

Nach dem Start des Programms befindet sich die Anwendung in einem Zustand „Idle“. Von hier aus kann sie alle auftretenden Ereignisse verarbeiten. Wenn nun das Ereignis (Menüauswahl) Datei - Lade Hostdatei ausgelöst wird, so wird die Aktion Wnd.openFileDialog ausgeführt. Die Anwendung befindet sich dann im Zustand „Filedialog wird angezeigt“, in dem ein Filedialog erscheint, um eine gewünschte Hostdatei zu laden. Man hat dann hier die Möglichkeit durch das Ereignis Cancel wieder in den Zustand Idle zu kommen, wobei sich bei der Grafikausgabe nichts tut (da man ja keine Hostdatei ausgewählt hat). Im Falle, daß man eine gültige Hostdatei ausgewählt hat (Bedingung) und auf OK klickt (Ereignis), so wird eine Aktion openHostFile ausgeführt, die dann dafür sorgt, daß die ausgewählte Hostdatei eingelesen, sortiert, arrangiert und auf dem Bildschirm ausgegeben wird. Man befindet sich dann wieder im Zustand „Idle“ und kann eine neue Menüauswahl treffen. Im Falle, daß man keine gültige Hostdatei ausgewählt hat bekommt man einen Fehlerdialog auf dem Bildschirm angezeigt (Transition in Zustand „Fehlerdialog wird angezeigt“). Durch ein OK-Ereignis kommt man dann wieder in den Zustand „Idle“, von wo aus man dann eine neue Menüauswahl treffen kann. Das Ereignis (Menüauswahl) Datei - Lade Subnetzdatei wird analog dazu behandelt, wobei hier für das Einlesen,

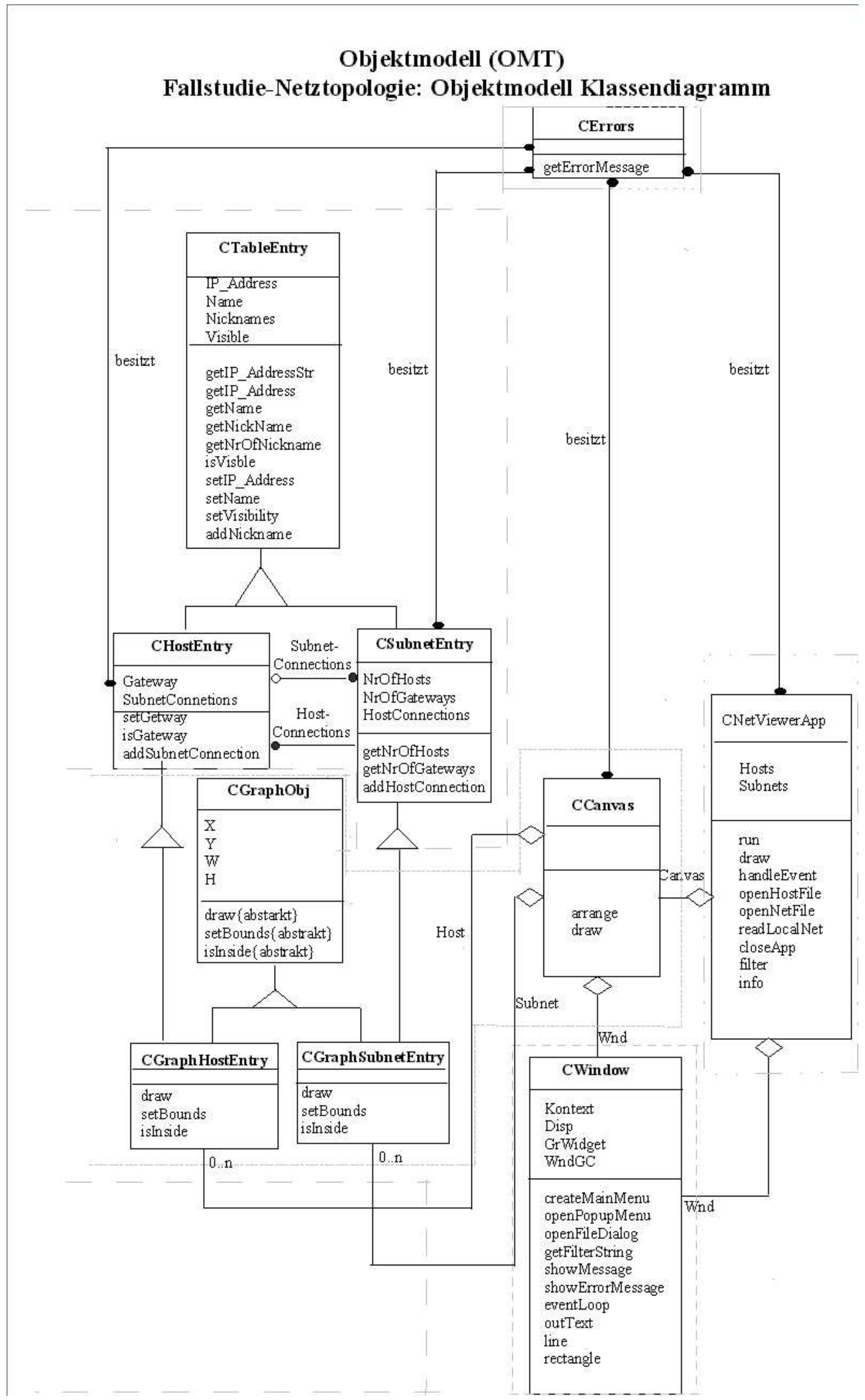


Abbildung 2.76: Fallstudie-Netztopologie: Objektmodell

Sortieren, Arrangieren der Netzdatei und ihre Ausgabe auf dem Bildschirm als Unterschied die Aktion openNetFile sorgt.

Bei dem Ereignis Datei - Lade Netzwerkinfos wird kein Filedialog angezeigt, so daß durch Auswahl dieses Menüeintrages (im Falle, daß die Netzwerkdatei korrekt ist) die Aktion readLocalNet direkt ausgeführt wird und nach den Routinevorgängen die Ausgabe auf dem Bildschirm erscheint. Man kommt dann wieder in den Zustand „Idle“ und kann neue Ereignisse auslösen. Wenn die Netzwerkinformationen nicht verfügbar sind, so erscheint wie gewohnt ein Fehlerdialog mit der entsprechenden Fehlermeldung. Ein Klick auf OK führt dann wieder in den Ausgangszustand („Idle“). Um nur bestimmte Host- bzw. Subneteinträge auf dem Bildschirm anzuzeigen gibt es die Möglichkeit des Filters. Hierbei bekommt man durch das Auslösen des Ereignisses (Menüauswahl) Bearbeiten - Filter einen Filterdialog angezeigt (durch die Ausführung der Aktion Wnd.getFilterString erfolgt eine Transition in den Zustand „Filterdialog wird angezeigt“). Man hat nun die Möglichkeit im Filterdialog die gewünschten Einträge zu selektieren und durch OK (Ereignis) dieses zu bestätigen. Es wird dann die Aktion filter ausgelöst, die dann die modifizierte grafische Ausgabe einleitet. Man kommt wie gewohnt in den Ausgangszustand. Ein Ereignis Cancel führt sofort in den Ausgangszustand, ohne daß die Selektion wirksam wird.

Im Programm ist dann noch ein Menüeintrag Hilfe - Info eingebaut. Ein Auslösen dieses Ereignisses leitet dann die Ausführung der Aktion info ein und man bekommt einen Infodialog angezeigt (neuer Zustand „Infodialog wird angezeigt“). Hier hat man nur die Möglichkeit durch einen OK-Klick (Ereignis) wieder in den Ausgangszustand („Idle“) zu wechseln. Wenn man das Programm dann letztlich beenden möchte, so reicht das Betätigen der Menüauswahl Datei - Beenden (Ereignis), die dann durch Aufruf der Aktion closeApp die Anwendung beendet.

2.4.4.4 Funktionales Modell

Das dritte Modell „Funktionales Modell“ beschreibt die Datenwert-Transformationen innerhalb des Systems. Das funktionale Modell enthält Datenflußdiagramme. Ein Datenflußdiagramm repräsentiert eine Berechnung. Ein Datenflußdiagramm ist ein Graph, dessen Knoten Prozesse und dessen Linien Datenflüsse sind. Das folgende Datenflußdiagramm zeigt das Laden von Hostdateien (Abbildung 2.78).

Hierbei löst ein User (Handlungsobjekt=Actor) ein Ereignis (z.B. aus dem Programmenü die Menüauswahl Datei - Lade Hostdatei) aus. Demnach wird der Prozeß (im Objektmodell „Methode“) handleEvent der Klasse CNetViewerApp ausgeführt. Als Resultat dessen wird ein Filedialog angezeigt (Prozeß openFileDialog der Klasse CWindow). Bei ausgewählter Datei wird der filename (im Falle, daß man den Filedialog mit OK abschließt und die Datei gültig ist) an den Prozeß handleEvent übergeben. Dieser gibt den filenamen weiter (Datenfluß) an den Prozeß openHostFile der Klasse CNetViewerApp. Wenn die Datei korrekt ist, läuft dann durch openHostFile der Einlesevorgang der ausgewählten Hostdatei (diese befindet sich im Datenspeicher=Data Store). In dem Datenspeicher Liste wird dann der Hosteintrag erst einmal sortiert (durch Prozeß sort) und weiter in dem Datenspeicher Liste gehalten. Als nächstes wird der

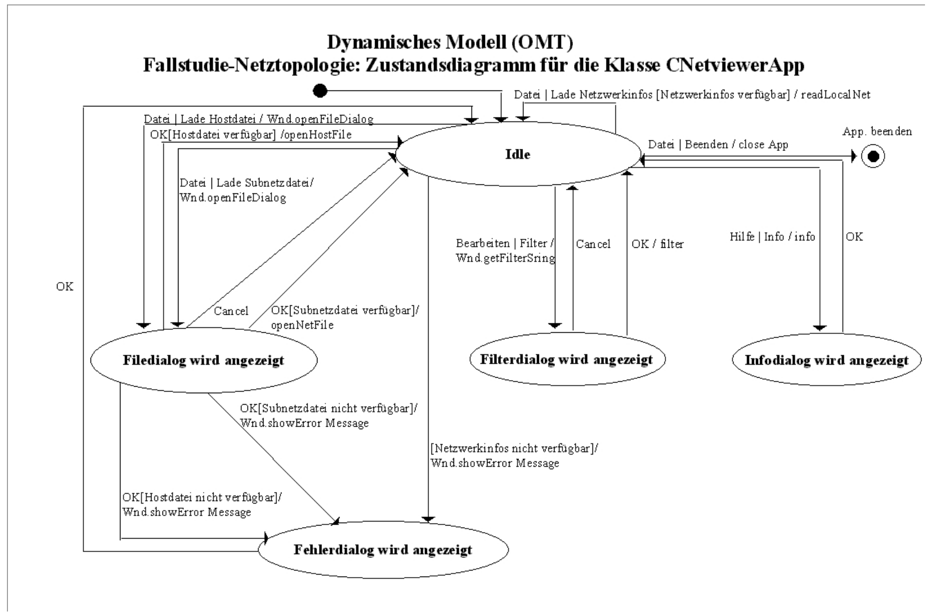


Abbildung 2.77: Fallstudie-Netztopologie: Dynamisches Modell

Hosteintrag arrangiert. Hierfür wird der Prozeß arrange der Klasse CCanvas mit den entsprechenden Abmessungen des Hosteintrages (x,y,w,h) ausgeführt (Datenfluß) und im Datenspeicher (Klasse CGraphHostEntry) abgelegt. Nun muß die sortierte und arrangierte Hostdatei noch auf dem Bildschirm dargestellt werden. Hierfür wird erst einmal der Prozeß draw der Klasse CNetViewerApp aufgerufen. Dieser ruft dann den Prozeß draw der Klasse CCanvas auf. Von hier aus wird dann letztlich der Prozeß draw der Klasse CGraphHostEntry aufgerufen und IP-Adresse, Name, Nicknamen des Hosteintrages zusammen mit seinen ermittelten Abmessungen auf dem Bildschirm (Handlungsobjekt Screen) ausgegeben (Datenfluß Output). Im Falle, daß die ausgewählte Datei aber nicht gelesen werden kann (durch Prozeß openHostFile nicht ausgeführt werden kann), wird eine Fehlermeldung auf dem Bildschirm angezeigt (die Nr. der Fehlermeldung wird hierbei dem Prozeß showErrorMessage der Klasse CWindow als Input übergeben. Dieser übergibt dem Prozeß getErrorMessage der Klasse CErrors dann diese Fehlernr. und bekommt als Gegenleistung die entsprechende Fehlermeldung als String übergeben, bevor diese dann letztlich auf dem Bildschirm ausgegeben wird).

2.4.5 Scotland Yard

2.4.5.1 Einleitung

Das Fallbeispiel Scotland Yard war ursprünglich Teil des Softwarepraktikums [KRCP96] im vierten Semester des Grundstudiums. Die Idee war hierbei eine Client-Server Architektur zu programmieren, um ein verteiltes Spielen des

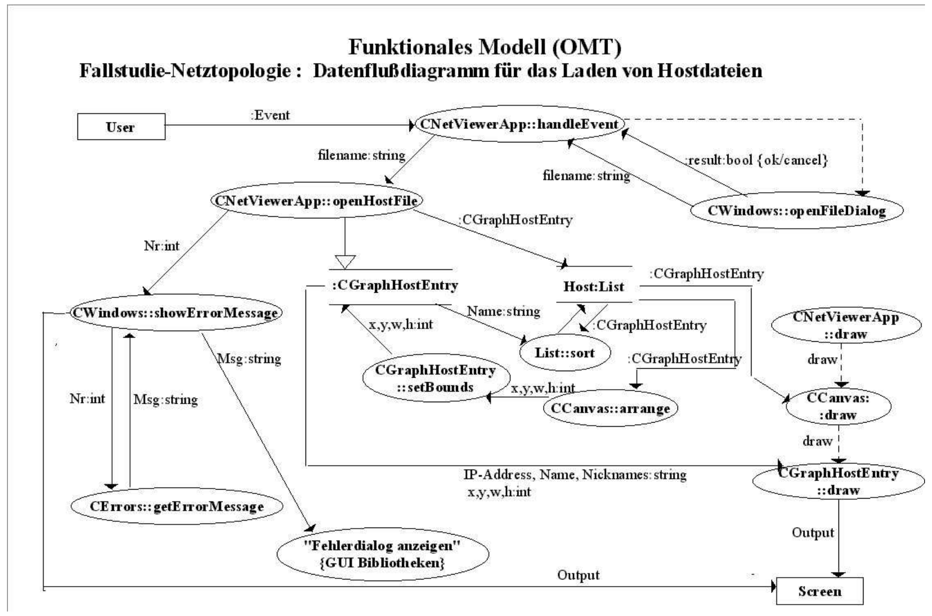


Abbildung 2.78: Fallstudie-Netztopologie: Funktionales Modell

bekannten Brettspiels auf mehreren Rechnern zu ermöglichen. Schon damals orientierte sich der Entwurf an der OOA/ OOD - Methode nach Coad/ Yourdon. Ferner wurde das System in einzelne Komponenten unterteilt:

- HIC: Human Interface Component - Benutzungsoberfläche
- PDC: Problem Domain Component - Anwendungsgebietskomponente
- TMC: Task Management Component - Koordinationskomponente

Diese logische Unterteilung vereinfachte die Implementation u.a. dadurch, daß große Teile in verschiedenen Modulen (Client/ Server) Wiederverwendung fanden.

2.4.5.2 Das Objektmodell

Das folgende Objektmodell bildet den Server der Applikation ab. Alle Spiele und ihre einzelnen Komponenten werden hier verwaltet (leider ist hierfür keine Abbildung vorhanden). Folgende Klassen machen das Modell aus:

HIC: Human Interface Component Benutzungsoberfläche

Der Server verfügt über eine Monitor-Funktion mit Textausgabe. Die Monitorfunktion dient der Übersicht über die angemeldeten bzw. verbundenen Clientrechner sowie der Kommunikationsüberwachung. Zwischen zwei verschiedenen

Bildschirmen bzw. Darstellungsoptionen des Monitors kann durch Eingabe von definierten Tasten gewechselt werden. Die beiden Masken CMConnections und CMMessages werden von der Klasse Mask abgeleitet, welche Methoden zur Abfrage des Tastaturbuffers sowie zum Zeichnen und Loeschens des Bildschirmes zur Verfügung stellt. Einzelne Elemente einer Maske sind u.a. Header und Messageline. Diese werden als Instanzen der Klasse Maskitem erzeugt. Ein weiteres Maskenobjekt, welches jedoch nicht unter Maskitem fällt sind die Tabellen. So besitzt jede Maske CM* eine Tabelle CT*, instanziiert aus der Klasse Table. Diese verwaltet die besonders häufig wechselnden Grafikkomponenten wie z.B. das Updaten aus dem Messagesbuffer.

PDC: Problem Domain Component Anwendungsgebietskomponente

Die Anwendungsgebietskomponente besteht aus der Party-Klasse. Diese verwaltet das Spiel, bestehend aus Karte, Spielern, Tickets sowie Zügen. Party greift hierbei auf die Klassen Gameboard, Move, Player und Ticket zu. Party selbst stellt die Möglichkeiten bereit, neue Partien zu starten oder zu beenden, ihren Status zu erfragen sowie in einer vorhandenen Partie zu ziehen. Diese Aktionen werden durch Interaktionen mit den Clients gestartet, hier auf der Partyklasse unter Zuhilfenahme der oben genannten Klassen verifiziert und als Information über den Server an alle Teilnehmer verteilt.

TMC: Task Management Component Koordinationskomponente

Die Klasse Server koordiniert die komplette Applikation SY_SERVER. Nach ihrem Aufruf erhält sie Anfragen, leitet diese an die Party oder die CMConnections- bzw. CMMessagesmaske weiter, erhält die entsprechenden Informationen und schickt sie zum jeweiligen Client bzw. an alle Clients zurück.

2.4.5.3 Das Dynamische Modell

Das dynamische Modell betrachtet die Auswahlmaske für Partien auf Clientseite. Den Einstiegspunkt bildet die Partiesmaske, Endpunkt wird nach Aufruf der Quit-Maske erreicht. Aktionen (Events) werden mit definierten Tasten (hier in spitzen Klammern dargestellt) gestartet. User- sowie Messages-Maske sind zwei der Partiesmake ähnliche Screens und mittels der Funktiontasten wechselseitig aufrufbar. Sie stellen zusammen mit der Quit- und der Help-Maske neue Oberflächen dar. Innerhalb der Parties-Maske können drei Aktionen gestartet werden:

- Eine bestehende Partie kannn ausgewählt werden. Danach kann man teilnehmen, einen Autoplayer starten, eine Partie verfolgen und sich die Tickets oder Moves zu einer Partie anzeigen lassen.
- Ein neues Spiel kann gestartet werden. Mit dem Tastenkürzel [S] wird ein Requester gestartet, der nach GRM (Game Resource Model), der Spielerrolle, die man gerne übernehmen möchte sowie einer eindeutigen Namensbezeichnung für das neue Spiel fragt. Als Condition wird vorrausgesetzt, daß der Benutzer sich zuvor mit Namen beim Server angemeldet hat (dies geschieht automatisch beim Starten des Clients).

- Eine Nachricht, die an alle Mitglieder der gerade ausgewählten Partie bestimmt ist kann abgeschickt werden. Mit dem Tastenkürzel [S] wird ein Requester gestartet, in den man nun seine Message eingeben kann. Alle Requester können durch Drücken der Escapetaste abgebrochen werden, der alte Zustand wird wiederhergestellt.

Abbildung 2.79 stellt diese Zusammenhänge noch einmal grafisch dar.

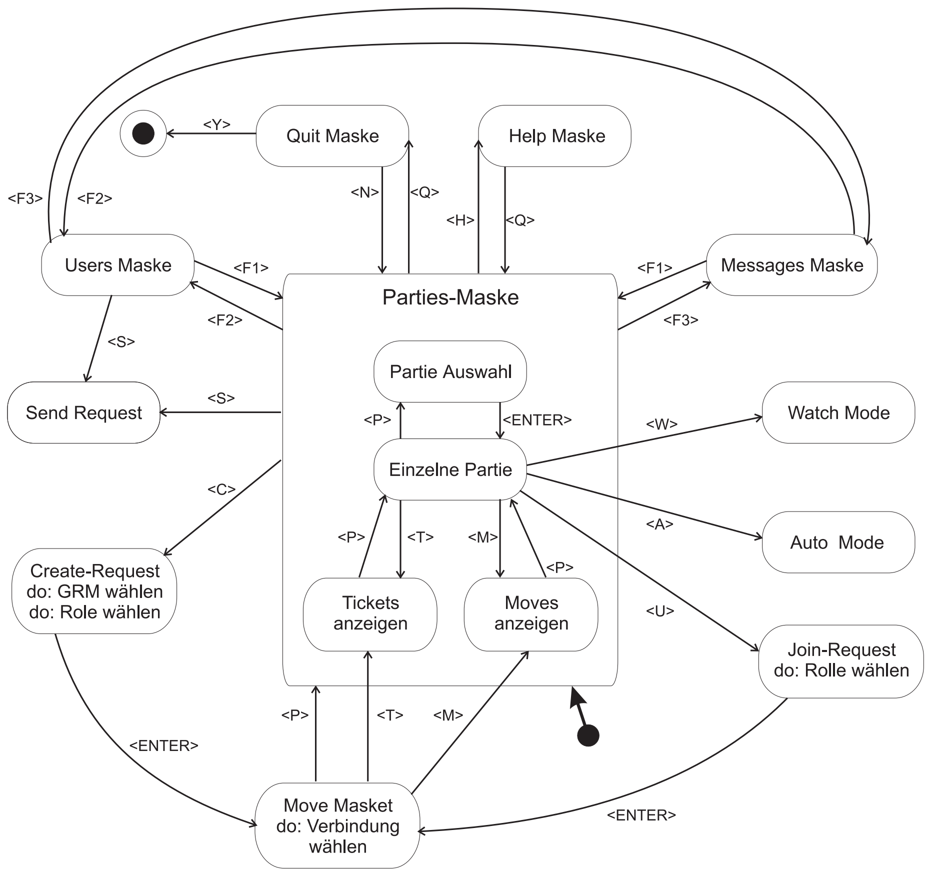


Abbildung 2.79: Dynamic Model - Scotland Yard

Kapitel 3

Projekt-Infrastruktur

Da zu Beginn noch keine Projekt-Räumlichkeiten zur Verfügung standen, mußten technische Vorhaben in den allgemein zugänglichen Praktikumsräumen erledigt werden. Zu Anfang des zweiten Semesters konnte das Projekt schließlich von einem komplett ausgerüsteten Raum in der fünften Ebene des MZH profitieren, so daß in der entscheidenden Phase die nötigen Randbedingungen erfolgreicher Programmierung gegeben waren. Entwickelt wurde auf drei Sun Ultra Sparc Workstations.

Fast ebenbürtig der internen Kommunikation zwischen den Projektmitgliedern, ist die Reflexion von erarbeitetem Wissen und Informationen nach außen. Es galt Wege zu finden, den Projektfortschritt angemessen zu dokumentieren. Aus diesem Grunde entschied sich das Projekt, gleich zu Beginn die Kommunikationswege der neuen Medien zu nutzen.

Die Internetdienste Email, Newsgroup und WorldWideWeb stellen sowohl ein interaktives Medium dar, das sich zum Informationsaustausch in Gruppen prädestiniert, als auch ein leicht zu pflegendes Hilfsmittel, um gezielt Aufgaben zu delegieren. Während sich nach den ersten Monaten zeigte, daß die eigens für das Projekt SIGN eingerichtete Newsgroup nur sporadisch von den Mitgliedern frequentiert wurde und der Kommunikationsfluß somit nicht gesichert war, bemühte sich das Management einen Kompromiß zu finden, indem persönliche Meetings in Arbeitsgruppen arrangiert und vermehrt Emails versandt wurden. Mit diesen Maßnahmen wurde gewährleistet, daß Projektmitglieder persönlich motiviert wurden und nicht in der Anonymität der Newsgroup verloren gingen.

Die Website erwies sich im Gegensatz zu der Newsgroup von vornherein als nützliches Tool. Nicht nur daß hier die Protokolle vergangener Plenarsitzungen jeder Zeit abrufbar waren; vielmehr sollten alle wissenschaftlichen Arbeiten, zu denen Referate, Fallstudien, technische Dokumentationen etc. zählen, an dieser Stelle zur Verfügung gestellt werden. Ohne einen festgelegten Dokumentenstandard, der ausnahmslos für alle schriftlichen Ausarbeitungen über die gesamte Projektdauer Bestand hat, wäre dies nur schwerlich zu realisieren gewesen.

Die Entscheidung die deskriptive Auszeichnungssprache SGML einzuführen erwies sich jedoch nicht nur für die Website als richtig. Da SGML-Code beliebig interpretierbar ist, können alle gewünschten Ausgabeformate erzeugt werden. So existiert neben einem HTML- Export die Möglichkeit, SGML nach LaTeX zu übersetzen. Gerade bei größeren Dokumenten und Projekten, an denen mehrere Menschen an unterschiedlichen Systemen arbeiten, kommen die Vorteile von SGML zum tragen. Insbesondere der Abschlußbericht profitiert von der Einheitlichkeit der Teildokumente.

Aber nicht nur bei der Erstellung eines Dokumentes können mehrere Personen beteiligt sein. Vor allem bei der Programmierung sind spezielle Mechanismen erforderlich, die Inkonsistenzen und Synchronisationsprobleme umgehen. Zu diesem Zweck benutzt das Projekt in beiden Fällen das Concurrent Versions System (CVS).

3.1 Corporate Website

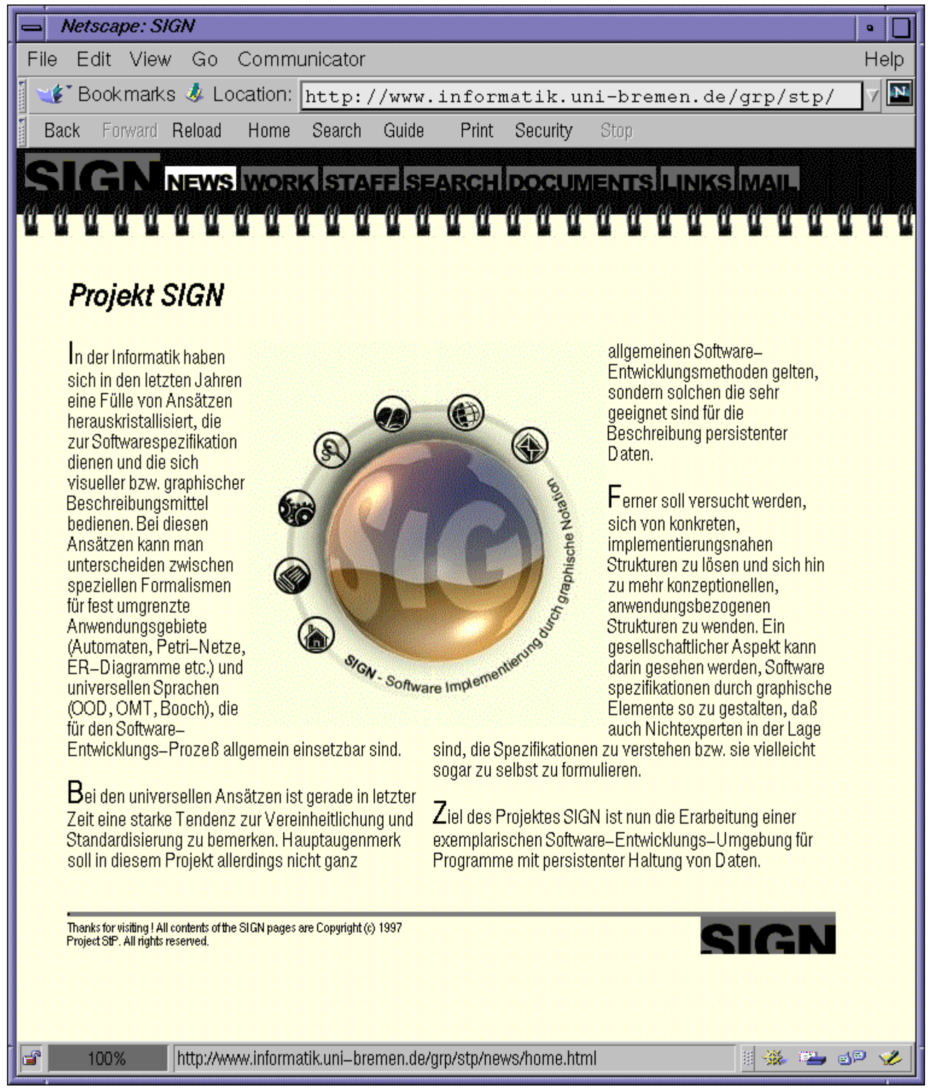


Abbildung 3.1: Projekt Homepage: www.informatik.uni-bremen.de/grp/stp

Das Projekt SIGN bediente sich von Beginn an den Möglichkeiten des World-WideWeb, um Informationen möglichst aktuell an die Projektmitglieder und interessierte Dritte weitergeben zu können. Die erste projekteigene Website war in der Gestaltung schlicht und vermochte mit ihrem Informationsgehalt lediglich den Mitgliedern eine Hilfe zu sein. Konzepte von Corporate ID und der Wunsch sich nach außen zu präsentieren, konnten erst in der zweiten und bisher endgültigen Version vereinigt werden.

Die folgenden Abschnitte sollen einen Einblick in den aktuellen Stand (August 98) der SIGN Website geben (Abbildung 3.1):

3.1.1 Sitemap

Die Sitemap oder Sitestructure wurde bewußt einfach gehalten und beschreibt bereits auf oberster Ebene, welche Informationen der Besucher auf der Website findet und auf welchem Weg er sein Ziel am schnellsten erreicht. Zu diesem Zweck existiert ein Frame am oberen Rand des Fensters, der, in Anlehnung an eine Reiterübersicht, die weitere Einteilung in Unterseiten darstellt. Im folgenden werden diese Seiten mit ihren Inhalten kurz skizziert:

- *NEWS*

  Ursprünglich war für die News-Page eine chronologische Liste mit Meilensteinen und Projekt-Vortschritten vorgesehen. Der redaktionelle Aufwand stand jedoch in keinem Verhältnis zum Nutzen. Daher wurde diese Form der Kommunikation bald eingestellt.

- *WORK*

  Diese Seite enthält neben dem offiziellen Phasenplan des Projekts ein Java-Applet, das über ein TreeControl alle Arbeitspakete kategorisiert darstellt. Für die Projektmitglieder sollte sich durch eine automatisierte Erfassung der APs der Verwaltungsaufwand mindern und eine höhere Transparenz bezüglich Arbeitsgruppen und Fortschritt ergeben. Ferner findet der Interessierte eine kurze Einführung in die Mechanismen der Projektverwaltung.

- *STAFF*

  Damit sich jeder Besucher der Seite einen Eindruck verschaffen kann mit wem er oder sie es zu tun hat, werden die Projektmitglieder, d.h. Betreuer und Studenten, hier kurz vorgestellt. Neben einem optionalen Photo, Namen, Email und Homepage werden die Arbeitsgruppen aufgelistet an denen die entsprechende Person zur Zeit arbeitet. Der eigentliche Nutzen dieser Seite soll aber darin liegen, die Motivation im Projekt zu erhöhen, indem sich jeder durch eine namentliche Nennung mit seinen Aufgaben stärker identifiziert.

- *SEARCH*

  Als nützliches Hilfsmittel hätte sich eine Suchfunktion erwiesen, mit deren Unterstützung Dokumente aufgespürt oder Referenzen gefunden worden wären. Es kam jedoch nie zu einer Implementierung, da keine freien Arbeitskapazitäten zur Verfügung standen und der Bedarf nach einem solchen Tool innerhalb des Projektes nicht geäußert wurde.

- *DOCUMENTS*

  Auf dieser Seite werden diverse schriftliche Ausarbeitungen gesammelt und die Protokolle jedes Plenums bereitgestellt. Da Dokumente innerhalb des Projektes in SGML verfaßt werden, konnte ein zu diesem Zweck entwickelter Postprocessor HTML- Dateien erzeugen, die dem Look and Feel der gesamten Website entsprechen. D.h. es wurde bei der Erstellung von HTML- Dokumenten kein Mehraufwand getrieben, da dieser Prozeß automatisiert werden konnte.

- *LINKS*

  Hier werden interessante Links zu projektverwandten Themen oder relevanten technischen Beschreibungen gesammelt. Leider konnte während des Projektzeitraumes keine angemessene Pflege stattfinden, so daß nur ein kleiner Teil der genutzten Internetresourcen abgebildet werden konnte.

- *MAIL*

  Adressen der Projektbetreuer, der Forschungsgruppe und der Weg zum Projektraum ermöglichen es dem Geneigten schnell mit dem Projekt in Kontakt zu treten. Die zusätzlich aufgeführte Email-Adresse: *stp@informatik.uni-bremen.de* leitet über einen Verteiler die Nachricht an alle Mitglieder von SIGN weiter.

3.2 SGML

3.2.1 Konzeption

Ihren Ursprung findet SGML (Standard Generalized Markup Language) in dem 1969 von IBM entwickeltem GML [Gol94]. Jene Entwicklung geht auf eine Grundkonzeption zurück, Formatierungen eines Textes von seinem Inhalt zu trennen. Schließlich wurde SGML 1986 als OSI (Open System Interchange) von der ISO festgeschrieben. Dadurch wurde die deskriptive Markup-Sprache ihrem ersten Anspruch, der internationalen Standardisierung, gerecht. Der Vorteil dieser Normung ist die System- und Verarbeitungsunabhängigkeit.

Ein SGML-Dokument beinhaltet neben dem Text eine Klassifizierung des selbigen. So lassen sich bestimmte Textbereiche z.B. als Kapitel definieren, ohne dabei jedoch festzulegen, wie sich diese Zuweisungen auf den Output (z.B. Druckbild) auswirken. Diese Klassifizierungen sind folglich keine Formatierungsanweisungen. Jene logische Strukturierung wird innerhalb des Dokumentes als Markup mit Hilfe von Entitäten, Elementen und Attributen vorgenommen. Eine entsprechende Deklaration findet in der DTD (Document Type Definition) statt [Bra96]. Da dieses Markup frei definiert werden kann, ergibt sich ein weiterer Vorteil gegenüber gewöhnlichen Taggingsystemen. Ferner wird in der DTD eine "Grammatik" festgelegt, welche eine Überprüfung der Gültigkeit eines Textabschnittes in einem bestimmten Kontext erlaubt.

Um die beschriebene Funktionalität von SGML zu realisieren, sind drei Komponenten nötig, die sowohl bei der Erstellung von SGML-Dokumenten, als auch in den DTD von entscheidender Rolle sind:

3.2.1.1 Elemente

Elemente stellen die eigentlichen Tags bzw. das deskriptive Markup dar und sind durch einen eindeutigen Namen, dem Generic Identifier (GI) spezifiziert. Sie treten i. A. paarweise auf - umschließen einen Textbereich. Es handelt sich entsprechend um ein Start- und ein End-Tag. Eine solche Konstellation wird auch als Container bezeichnet. Das Auftreten von Containern ohne Inhalt und/oder mit nur einem Tag ist in SGML jedoch nicht ausgeschlossen. Es ist möglich Container zu schachteln.

Im Inhaltsmodell wird die logische Struktur der Container festgelegt. So wird formuliert, welche Elemente geschachtelt werden dürfen, wieviele und in welcher Reihenfolge sie auftreten.

Beispiel:

```
<TITEL> Hier steht der Titel </TITEL>
```

3.2.1.2 Attribute

Attribute sind Eigenschaften von Elementen. Sie stellen zusätzliche Informationen dar, ohne im eigentlichen Inhalt des Elementes zu erscheinen. Mehrere Attribute sind erlaubt - Elemente kommen aber auch ohne sie aus.

Beispiel:

```
<TITEL ABSCHNITT="1.1.2"> Hier steht der Titel </TITEL>
```

3.2.1.3 Entitäten

Entitäten sind konstante Referenzen eines SGML-Dokumentes. Ihre Bedeutung wird zuvor in der DTD explizit definiert. Wird ihr Inhalt geändert, so sind automatisch alle Referenzen betroffen.

Beispiel: ü entspricht ü

3.2.2 Motivation

Die Verfügbarkeit und Nutzung verschiedener EDV-Systeme schafft Probleme hinsichtlich der Portabilität bzw. Kompatibilität gebundener Textverarbeitungssysteme. Der Versuch, den Datenimport und -export durch entsprechende Filter zu ermöglichen mag, als Lösung dieser Schwierigkeiten erscheinen, schafft mittelfristig jedoch neue Probleme. Mit dem Release eines Updates oder höheren Versionsnummern verlieren die Filter auf anderen Systemen in der Regel ihre Gültigkeit. Eine Anpassung der Software ist somit unumgänglich.

SGML hingegen befindet sich auf einer weitaus höheren Abstraktionsebene; hat mit solchen Problemen also nicht zu kämpfen. Die Standardisierung durch die ISO war lediglich der letzte Schritt ein einmalig geschriebenes SGML-Dokument auf allen erdenklichen Systemen verfügbar zu machen. Grundsätzlich ist dafür die Tatsache verantwortlich, daß es sich bei SGML nicht um eine Applikation handelt, sondern um eine Dokumentenbeschreibungssprache, die folglich mit jedem beliebigen Texteditor verarbeitet werden kann.

Während konventionelle Textverarbeitungssysteme durch prozeduralen Markup bestimmten Textpassagen explizite Layoutinformationen zuweisen - also das Aussehen des Dokumentes im gedruckten Zustand festlegen, nimmt SGML durch deskriptives Markup lediglich eine Klassifizierung vor. Der Möglichkeit einer Weiterverarbeitung sind somit keine Grenzen gesetzt. In der Praxis bedeutet das, Ausgabeformate in HyperText (Internet), PostScript (Drucker) etc. lassen sich aus nur einer Datei erzeugen.

Ferner besitzt SGML die positiven Fähigkeiten, mehrere Dateien zur Erstellung eines zusamenhängenden Dokumentes zusammenzubinden, Tabellen und ähnliches rein textuell zu beschreiben, Kommentare und Referenzen zu ergänzen, sowie die Dokumenten-Typ-Definition speziell auf die geforderten Spezifikationen anzupassen.

Schließlich zeigt das populäre Beispiel von HTML (Hyper Text Markup Language), einer artverwandten Sprache, daß SGML nicht nur Hochschulprojekten oder spezialisierten Firmen vorenthalten bleibt.

3.2.3 SGML Tools

Wie ein neues Dokument erzeugt und letztlich in das gewünschte Ausgabeformat konvertiert wird, sei im folgenden beschrieben. Damit alle Werkzeuge reibungslos funktionieren sind noch geringfügige Grundeinstellungen vorzunehemen:

```
[.bash_profile] PATH= .........;/home/stp/bin/
[.bash_profile] export PERL5LIB = /home/stp/lib/perl5/
```

3.2.3.1 emacs

Wie bereits erwähnt, stellt SGML lediglich eine Dokumentenbeschreibung dar. Ein SGML-Dokument ist demnach ein reines Textfile. Die Entscheidung, mit welchem Hilfsmittel diese Datei editiert wird, bleibt jedem selbst überlassen. Der Emacs bietet sich jedoch aufgrund seines SGML-Modes besonders an.

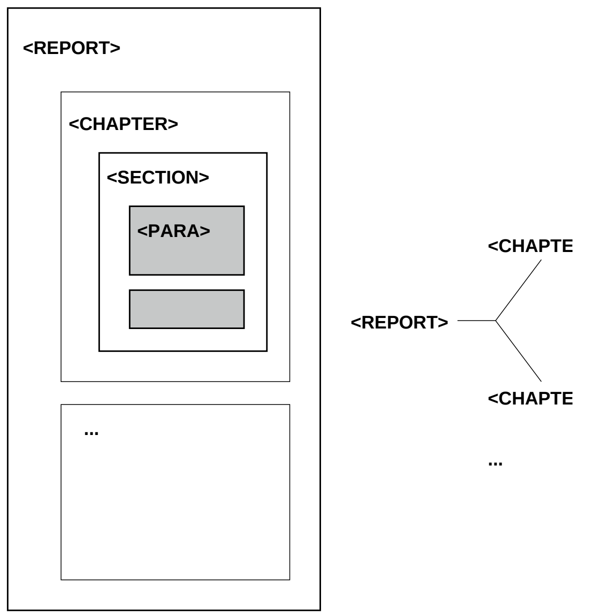


Abbildung 3.2: schematische Darstellung der Dokumentstruktur

Hierzu müssen die beiden Dateien psgml.el und psgml.elc in einem Verzeichnis abgelegt werden, das in der emacs load-path Variable festgelegt ist. Zusätzlich muß folgende Zeile in die Datei „~/.emacs“ eingetragen werden:

```
[.emacs] (autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

Eine Auswahl an Makros, ausgeführt durch spezielle Tastenkombinationen, macht ein komfortables Editieren möglich. So prüft der emacs schon während der Bearbeitung das Markup hinsichtlich seiner Korrektheit in dem gegebenen

Kontext. Markierte Abschnitte lassen sich von einem gewählten Tag automatisch einfassen, ebenso wie fehlende End-Tags durch Tastendruck hinzugefügt werden können. Weitere Informationen zur Nutzung des SGML-Modes in emacs findet man unter http://www.fak14.uni-muenchen.de/infos/psgml_toc.html im WWW.

TIP: Es empfiehlt sich, zur Erstellung eines neuen SGML-Dokumentes auf die vorhandenen Templates oder Beispiele zurückzugreifen. Der eigene Text muß nun lediglich in die entsprechenden Abschnitte getippt/kopiert werden. Ein Arbeiten nach der Methode copy'n paste führt am schnellsten zu den gewünschten Resultaten. Vorlagen zu allen Dokumenttypen befinden sich in dem Verzeichnis: /home/stp/doc/templates/

3.2.3.2 nsgmls

Der nsgmls ist ein Parser, der ein SGML-Dokument auf die in der DTD festgelegte Grammatik überprüft. Falls ein unzulässiger Gebrauch von Elementen, Attributen oder Entitäten gemacht wird, gibt der Parser eine Fehlermeldung aus.

Der explitzite Aufruf des nsgmls beim Portieren des SGML-Dokumentes in ein gewünschtes Zielformat ist nicht gefordert, da dieser Schritt automatisch beim gebrauch von stpmake (siehe unten) geschieht. Geht es aber darum, lediglich die syntaktische Korrektheit des Dokumentes zu kontrollieren, bietet sich dieses Tool an.

Der Aufruf geschieht über commandline und erfordert eine SGML-Datei als Argument. Die ausführbare Datei nsgmls befindet sich in dem Verzeichnis: /home/stp/bin/

3.2.3.3 stpmake

stpmake ist der eigentliche „Compiler“ für SGML-Dokumente. Damit es zu keinen Problemen kommt, sollte der Aufruf lediglich auf Rechnern mit Solaris erfolgen. Das spezialisierte Makefile ist vielfältig anwendbar:

Argument	*Bedeutung*
stpmake	alle sich in dem aktuellen Ordner befindlichen SGML-Dokumente werden sowohl nach HTML, als auch in das PS-Format konvertiert.
stpmake [format]	alle sich in dem aktuellen Ordner befindlichen SGML-Dokumente werden in das Zielformat format” (HTML, PS) konvertiert.
stpmake file.sgml	das einzelne File file.sgml wird nach HTML und PS konvertiert
stpmake file.html	das einzelne File file.sgml wird nach HTML konvertiert
stpmake file.ps	das einzelne File file.sgml wird nach PS konvertiert

Die ausführbare Datei nsgmls befindet sich in dem Verzeichnis: /home/stp/bin/

3.2.3.4 mv2folder

mv2folder vereinfacht die Einbindung von neuen Protokollen in die HTML-Strukturen der Heimseiten von StP. Nachdem ein Protokoll erfolgreich mit stpmake nach HTML konvertiert wurde, wird es durch Aufruf des Perlscripts [WSCP96], [Sha94]automatisch in den dafür vorgesehenen Ordner verschoben. Zusätzlich findet eine Aktualisierung der HTML-Seiten statt, die nun einen Link auf das Protokoll bieten.

Das Perlscript wird mit der HTML-Datei als Argument aufgerufen. Mit dem Argument „-r“ wird angegeben, daß der Link der folgenden Datei aus der HTML-Seite gelöscht werden soll. Diese Option ist nötig, um falsche Einträge rückgängig zu machen.

Beispiel:

```
mv2folder [-r] <filename>
```

3.2.4 SGML in der Praxis

3.2.4.1 Whitespace Characters

In SGML werden Tastaturzeichen wie Space, Tab und Newline ignoriert. Eine freie Zeile wird bei der PS-Konvertierung jedoch als neuer Absatz gewertet. Dieses Verhalten ist nicht im Sinne von SGML und sollte daher nicht mißbraucht werden. Die Ursachen liegen in der Ausgabe des SGML-Parsers.

3.2.4.2 Einbinden von Bildern

Abbildungen sollten in jedem Fall als Encapsulated Postscript-Dateien vorliegen, damit eine möglichst problemfreie Konvertierung in die verschiedensten Ausgabeformate gewährleistet wird. Es ist ebenfalls notwendig, daß sich die Grafiken im selben Ordner befinden, wie das zugehörige SGML-Dokument. Bei der Konvertierung nach HTML werden alle EPS-Grafiken in das GIF-Format umgewandelt. Um erfolgreich Grafiken in seine Dokumente einzubinden, müssen die Attribute **id** und **file** gesetzt sein. Optional kann eine überschrift **caption** angegeben werden.

Beispiel:

```
<figure id="f2" file="Arnold.eps" caption="Urlaub '94">
```

TIP: Falls Grafiken bereits in anderen Formaten (z.B. GIF, TIF, JPEG etc.) vorliegen und eine Kovertierung nicht sinnvoll ist (zu großer Qualitätsverlust), bietet es sich an, jene Dateien als Links in die Dokumente einzubinden.

Abbildung 3.3: Urlaub '94

3.2.4.3 Einbinden von Dateien/Links

Es ist in der StP.dtd vorgesehen, für Hypertext-Dokumente durchaus Links zuzulassen. Dabei ist es unerheblich, ob es sich tatsächlich um einen Link auf ein weiteres Dokument handelt, oder um eine Datei (z.B. Text, Grafik). Links können wie z.B. (*hyperlink* ▹ bremen.online) direkt in den Textfluß integriert werden.

Beispiel:

```
<ulink url="http://www.bremen.de" type="hyperlink">bremen.online</ulink>
```

3.2.4.4 Darstellen von Umlauten

Unter SGML wird der iso-latin-1 Standard eingehalten, d.h. Umlaute wie auch Sonderzeichen können direkt in den Text geschrieben werden. Es ist nicht länger nötig, die in der Tabelle aufgeführten Entitäten zu benutzen. Auf amerikanischen Tastaturen lassen sich die Umlaute über Tastenkombinationen erzeugen (zumindest in emacs).

Umlaut	*Entität*	*emacs keybinding*
ä	ä	C-x 8 " a
ö	ö	C-x 8 " o
ü	ü	C-x 8 " u
Ä	Ä	C-x 8 " A
Ö	Ö	C-x 8 " O
Ü	Ü	C-x 8 " U
ß	ß	C-x 8 " s

3.2.4.5 Darstellen von SGML-Markup

Mit einem kleinen Trick ist es ebenfalls möglich, den SGML-Parser kurzweilig ausszuschalten, um SGML-eigene Tags in den Dokumenten zu verwenden. Dazu muß am Anfag des zu kapselnden Blocks das Kürzel *<![CDATA [* eingefügt werden; am Ende entsprechend *]]>*.

Beispiel:

```
<prglisting>
<![ CDATA [
<list>
  <item>Logische und deskriptive Auszeichnung (<em>markup</em>) von
  Textelementen</item>
</list>
]]>
</prglisting>
```

3.2.4.6 Kommentare in SGML

```
<!-- Kommentarbereich -->
```

3.3 CVS Versionskontrolle

3.3.1 Was ist CVS ?

CVS oder auch „Concurrent Versions System“ dient der Revisionskontrolle und setzt auf das Tool RCS (Revision Control System) auf. Ein Programmierprojekt, bestehend aus einer Ansammlung von Dateien, kann mit Hilfe dieses Tools von mehreren Entwicklern gleichzeitig bearbeitet werden, ohne daß ein Datenverlust aufgrund inkonsistenter Versionen befürchtet werden muß.

CVS bietet die Möglichkeit einer gemeinsamen Nutzung derselben Dateien zur gleichen Zeit. Dabei sichert das Tool Unterschiede zwischen verschiedenen Versionen eines Files und kann diese gegebenfalls auf einen älteren Stand rekonstruieren. So läßt sich auch einfach nachvollziehen, welche Änderungen zu welchem Zeitpunkt gemacht worden sind.

Die geschützten Dateien liegen in einem speziellen Referenz-Verzeichnis - dem „Repository“, einer Ansammlung von Files, auch „Module“ genannt, denen ein Name zugewiesen wird, der zur eindeutigen Identifikation dient. Unter diesem Namen lassen sich Module aus dem Repository herausholen. Dieser Vorgang wird als „checkout“ bezeichnet. Die gewählten Dateien befinden sich nun in einem vom Benutzer angegebenen Arbeitsverzeichnis, dem „Working Directory“. Die Files lassen sich jetzt wie gewohnt editieren. Schließlich werden sie in das Referenzverzeichnis zurückgeschrieben: „Committed“. Hier werden sie dann mit früheren Versionen verglichen. Ferner ist es möglich einzelne Dateien zusätzlich in das Repository zu schreiben, oder zu entfernen.

Vom technischen Standpunkt aus ist CVS eine Ansammlung von Shellkommandos, die auf den Dateien im Repository und in der lokalen Kopie arbeiten. Als Interface läßt sich aber auch Emacs benutzen, so daß während der Arbeit kaum CVS-Kommandos direkt von der Shell aufrufen werden müssen.

3.3.2 Wie wird CVS konfiguriert ?

Um ein reibungsloses Arbeiten mit CVS zu garantieren, sind folgende Erweiterungen in der „~/.bash_profile“ verbindlich. Außerdem sollte sicher gestellt werden, daß RCS und CVS durch die PATH Variable aufrufbar sind (dies ist jedoch im Allgemeinen der Fall).

[.bash_profile] export CVSROOT=/home/stp/cvs - CVS-Rootverzeichnis setzen [.bash_profile] umask 002 - umask zum Setzen der Gruppenrechte

3.3.3 Wie wird mit CVS gearbeitet ?

Die Arbeit mit CVS setzt voraus, daß sich der Entwickler zusätzlich mit der Revisionskontrolle auseinandersetzt. Dies bedeutet in der Praxis, daß vor dem Editieren, zunächst die entsprechenden Dateien aus dem Repository geholt werden müssen:

3.3.3.1 Wie können Files aus dem Repository gelesen werden ?

Um Files aus dem Repository auszuchecken, wird der folgende Aufruf getätigt. Wobei entweder ein Verzeichnis bzw. Datei angegeben wird oder ein Modul, das sich bereits in dem Repository befindet. Das CVS erzeugt in dem aktuellen Verzeichnis eine komplette Abbildung der angegebenen Files, wie sie aktuell in dem Repository existieren. Die einzelnen Dateien können nun bearbeitet werden.

- cvs co <module>

3.3.3.2 Wie werden neue Files dem Repository hinzugefügt ?

Das Hinzufügen von Dateien ist völlig unproblematisch - leider kann der Befehl nicht rekursiv verwendet werden, weshalb bei der Erzeugung eines neuen Verzeichnisbaumes jede Hierarchieebene einzeln generiert werden muß. Ähnlich wie bei anderen CVS-Befehlen gilt auch hier der Grundsatz, daß erst nach einem commit (ci) die Änderungen tatsächlich stattfinden. Eine Ausnahme bilden Verzeichnisse.

- cvs add <file/directory>

3.3.3.3 Wie können Files aus dem Repository gelöscht werden ?

Prinzipiell gilt, daß in dem Working Directory keinesfalls Dateioperationen mit den ordinären UNIX-Kommandos (rm, cp, mv,...) stattfinden sollten, die Einfluß auf die Struktur haben. Im Allgemeinen existieren CVS-Pendants mit denen die gewünschten Resultate erzielt werden.

Anmerkung: Die angegebenen Dateien bzw. Verzeichnisse werden nicht physikalisch aus dem Repository gelöscht, statt dessen werden diese in das sog. Attic-Verzeichnis verschoben, um das Wiederherstellen älteren Versionen auf jeden Fall zu garantieren.

- cvs remove file/directory

3.3.3.4 Wie werden Änderungen dem Repository mitgeteilt ?

Das Einchecken von lokalen Änderungen geht immer mit einem Vergleich der Files einher. Dabei wird überprüft, ob mögliche Konflikte mit anderen Revisionen vorliegen. Ist dies der Fall, so werden die entsprechenden Stellen in den Dateien markiert. Es obliegt nun dem Entwickler, die Files so zu modifizieren, daß keine weiteren Konflikte auftreten. Immer dann, wenn die Daten nach Ansicht des Entwicklers einen sicherungswerten Stand erreicht haben, sollte ein „checkin“ (ci) geschehen.

- cvs ci [option] <file/directory>

3.3.3.5 Wie werden Änderungen anderer Entwickler auf die Dateien in meinem Arbeitsverzeichnis übertragen ?

Um eventuelle Änderungen der Daten im Repository lokal zu übernehmen, wird der Befehl update verwendet. Unterschiede zwischen Repository und Arbeitsverzeichnis werden zusammengeführt. Wie auch beim Einchecken wird auf Konflikte geprüft. Mit der Option [-d] werden mögliche neue Verzeichnisse des Repositories lokal erzeugt.

- cvs update [option] <module/file/directory>

3.3.3.6 Wie wird ein neues Repository angelegt ?

Eine schnelle und bequeme Art im Vergleich zu dem aufwendigen add-Befehl (fehlende Rekursion) ist der Import. Alles was sich im aktuellen Verzeichnis befindet, wird rekursiv (mit Unterverzeichnissen) in das Repository unter dem relativen Verzeichnis, mit der Bezeichnung module, geschrieben.

- cvs import <module> <Vendor-Tag> <Release-Tag>

3.3.3.7 Wie werde ich das Arbeitsverzeichnis los ?

Das lokale Arbeitsverzeichnis sollte auf keinen Fall gelöscht werden, ohne CVS davon in Kenntnis zu setzen; noch besser ist es, gleich das entsprechende CVS-Kommando aufzurufen. Mit dem release-Befehl wird das angegebene Verzeichnis aus der Kontrolle des CVS entlassen. Wird die Option [-d] nicht angegeben, so wird das Verzeichnis lediglich bei dem System abgemeldet. Andernfalls wird das lokale Verzeichnis komplett gelöscht. Auf jeden Fall werden die betroffenen Files zuvor in das Repository eingecheckt.

- cvs release [-d] <directory>

3.3.3.8 Wann und wie oft soll ich einchecken?

Eine wichtige Frage, wann und wie oft ein „commit“ erfolgen sollte, ist nach Möglichkeit noch vor der Implementierung zu klären. Checkt man zu häufig ein, so wird man oftmals Quellen im Repository haben, die nicht übersetzbar sind und oft sogar gänzlich unbrauchbar. Checkt man andererseits zu selten ein, ist die Wahrscheinlichkeit Konflikte zu bekommen höher. Wichtig ist hier, daß alle an einem Projekt beteiligten Entwickler darin übereinstimmen, wie oft (oder selten) neue Quellen eingecheckt werden.

3.4 Java

3.4.1 Einleitung

3.4.1.1 Entstehung

- 1991 SUN Microsystems
- Codename „Oak“: Software-Forschungsprojekt für elektron. Haushaltsgeräte
- Ziel: kleine, schnelle, plattformunabhängige Sprache
- Abstammung:

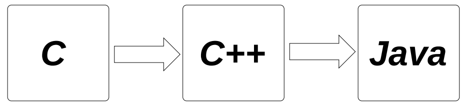


Abbildung 3.4:

umfaßt nur einen Teil von C++, konzentriert sich auf das Wesentliche, streng objektorientierter Aufbau

- Sprache mittlerweile relativ robust und stabil

3.4.1.2 Vorteile

Im Vergleich zu anderen Sprachen bietet Java [Fla98] viele Vorteile :

plattformunabhängig auf der Quelltextebene

- reiner ASCII-Text ? Benutzung diverser Editoren möglich
- C++ - ähnlich

plattformunabhängig auf der Binärebene

- es wird Byte-Code (auch Virtual Machine Code genannt) erstellt
- Byte-Code ähnelt nativem Code, jedoch nicht an einen speziellen Prozessortyp gebunden
- Ausführung des Programms auf der Anwenderseite unter verschiedenen Betriebssystemen mittels spezieller Interpreter
- ist in Browsern (Netscape, MS Explorer, HotJava,...) oder Appletviewer vorhanden
- Compilierung wahlweise auch als Anwendung (plattformabhängig)

umfangreiche Klassenbibliotheken

- java.lang: Object, Thread, Exception, System, Integer, Float, Math

streng objektorientiert

- stammt von C++ ab
- durch Nutzung der Klassenbibliotheken
- Entwicklung flexibler modularer Programme

leicht zu erlernen

- da kompakte Sprache
- durch Ähnlichkeit mit C++
- auf C++ -Komplexität wurde verzichtet
- Flexibilität dadurch kaum beeinträchtigt
- z.B. keine Pointer-Arithmetik: Strings und Arrays sind hier echte Objekte
- automatische Speicherverwaltung

Nachteile

- langsamere Ausführung als nativer Code
- hat Geschwindigkeit höhere Priorität ? Überführung in nativen Code während des Ladens empfohlen

3.4.1.3 Was benötige ich, um Java-Applets / -Anwendungen zu schreiben ?

- Java Developer's Kit (JDK) :
  - verfügbar für DOS, Windows, Unix, Solaris, Mac, ...
  - Compiler und Interpreter
- IDE:
  - Windows: Symantec Cafe, MS Visual J++, Borland Java, Mojo, Kawa
  - Unix: emacs im Java-Mode
- Browser:
  - Netscape Navigator ab V2.x
  - Microsoft Explorer ab 3.x (Javateil: bug-behaftet)
  - SUNs HotJava
  - Appletviewer aus dem JDK

3.4.1.4 Applets und Anwendungen

Entwicklung von zwei unterschiedlichen Programmtypen möglich:

	Applet (Haupteinsatzgebiet)	*Anwendung*
Einbettung:	in HTML-Seite	alleine lauffähig
benötigt:	javafähigen Browser (Java VM)	Java-Interpreter
Beispiel:	Kontoführung über Internet	HotJava-Browser

3.4.2 Einführungsbeispiel

- Dateien müssen immer so heißen, wie die Klassen, die sie beherbergen.
- Groß- und Kleinschreibung wird unterschieden

3.4.2.1 Anwendung

Quelltext schreiben

Erstellen einer Datei "HelloJava.java" mit folgendem Inhalt:

```
class HelloJava {
 public static void main (String args[]) {
 System.out.println("Hallo Java-Freunde!");
 }
}
```

- main-Methode wie in C: wird beim Programmstart ausgeführt

Quelltext compilieren

javac HelloJava.java

-> HelloJava.class (Byte-Code) wurde erstellt.

Ausführung

java HelloJava

- HelloJava.class wird ausgeführt und "Hallo Java-Freunde!" wird ausgegeben.

3.4.2.2 Applet

Quelltext schreiben

Erstellen einer Datei "HelloJavaApplet.java" mit folgendem Inhalt:

```
import java.awt.Graphics;

public class HelloJavaApplet extends java.applet.Applet {
        public void init() {
                resize(150,25);
        }
        public void paint(Graphics g) {
                g.drawString("Hallo Java-Freunde!", 50, 25);
        }
}
```

- import-Zeile entspricht #include in C und bindet benötigte Java-Klassen aus dem JDK ein
- paint-Methode dient zum Anzeigen des Applets auf dem Bildschirm
- main nicht erforderlich. Applets nutzen andere Standardmethoden: z.B. init, start, paint

Quelltext compilieren

javac HelloJavaApplet.java

- HelloJavaApplet.class (Byte-Code) wurde erstellt.

HTML-Datei schreiben

ab hier Unterschied zu den Anwendungen.

```
<HTML>
<HEAD>
<TITLE>Einfuehrungsbeispiel</TITLE>
</HEAD>
<BODY>
<P>Jetzt wird das Applet aufgerufen:<BR>
<APPLET CODE="HelloJavaApplet.class" WIDTH=200 HEIGHT=30>
Kein java-faehiger Browser vorhanden !
</APPLET>
</BODY>
</HTML>
```

Applet-Tag:

- der Text zwischen den „<APPLET>“ und „</APPLET>“-Tags wird nur von Browsern angezeigt, die nicht java-fähig sind.

Ergebnis anschauen

- mittels Appletviewer: appletviewer HelloJavaApplet.html
- mittels Browser: URL: helloJavaApplet.html

3.4.3 Sprachelemente

3.4.3.1 allgemeine Syntax

- C, C++

Anweisungen

- Anweisung enden mit ;
- Block: { Anweisung1; Anweisung2; ... } , auch geschachtelt

Bezeichner

- Bezeichner: { a...z — A...Z — _ — $ } [Unicode-Zeichen 1..n] z.B.: _testing $getit Test
- Unterscheidung von Groß- und Kleinschreibung
- Schlüsselwörter:

abstract	default	generic	operator	this
boolean	do	goto	outer	threadsafe
break	double	if	package	throw
byte	else	implements	private	throws
byvalue	extends	import	protected	transient
case	false	inner	public	true
cast	final	instanceof	rest	try
catch	finally	int	return	var
char	float	interface	short	void
class	for	long	static	volatile
const	future	native	super	while
continue		new	switch	
		null	synchronized	

Kommentare

- einzeilig: //
- mehrzeilig: /* ... */
- Dokumentationskommentare: /** ... */

Dokumentationskommentare

- haben die Form: /** ... */
- javadoc (im JDK enthalten) konvertiert diese Kommentare nach HTML
- in die Kommentare können HTML-Tags plaziert werden
- Vorgaben für javadoc beginnen mit @
- nur Kommentare für public oder protected werden übernommen
- Beispiel:

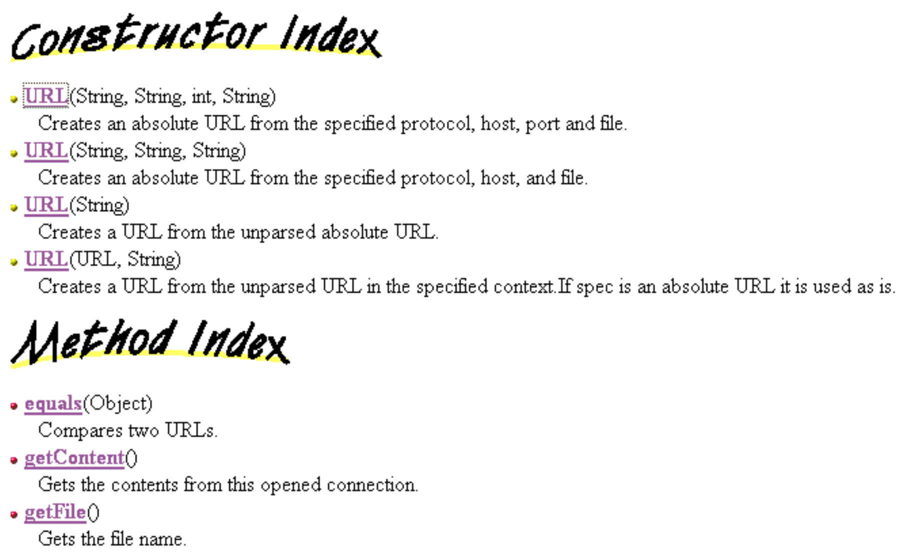
```
/**
* Dieser Text steht vor der Klassendefinition
* @version 0.0a
* @author wer auch immer
*/
public class ...
/**
* Dieser Text steht vor der Methode (oder einem Attribut).
* @see Klasse#Spezifizierer
* @param Parameter Beschreibung
* @exception Ausnahme Beschreibung
* @return Ergebnis Beschreibung
*/
public Ergebnis Methode(Parameter) throws Ausnahme
```

Ergebnis könnte so aussehen (HTML-File, welches von javadoc erstellt wird) :

Grunddatentypen:

Typ	*Länge (bit)*	*Bereich*	*Bemerkung*
byte	8	-127 ... 127	
short	16	-32 768 ... 32 767	
int	32	-2 147 483 648 ... 2 147 483 647	Standard
long	64	-9.22 hoch 18 ... 9.22 hoch 18	
float	32	10 hoch -38 ... 10 hoch 38 (7-8 Stellen)	
double	64	10 hoch -308 ... 10 hoch 308 (15-16 Stellen)	Standard
char	16	65536 versch. Zeichen (Unicode)	
boolean	8	true / false	

- boolean kann nicht wie in C in Zahlen konvertiert werden
- Escape-Code wie in C
- Type-Casting: valueNewType = (newTypeName) exprOldType)
- Unicode-Zeichen plattformabhängig



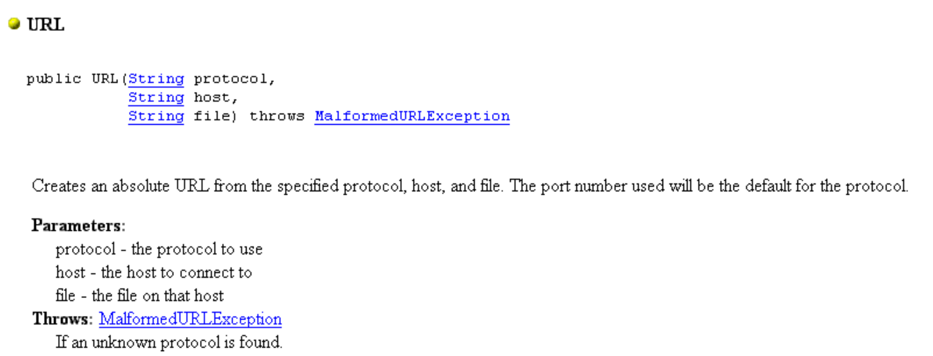
Constructor Index

- **URL**(String, String, int, String)
  Creates an absolute URL from the specified protocol, host, port and file.
- **URL**(String, String, String)
  Creates an absolute URL from the specified protocol, host, and file.
- **URL**(String)
  Creates a URL from the unparsed absolute URL.
- **URL**(URL, String)
  Creates a URL from the unparsed URL in the specified context.If spec is an absolute URL it is used as is.

Method Index

- **equals**(Object)
  Compares two URLs.
- **getContent**()
  Gets the contents from this opened connection.
- **getFile**()
  Gets the file name.

Abbildung 3.5: von javadoc erzeugtes HTML-File



- **URL**

```
public URL(String protocol,
           String host,
           String file) throws MalformedURLException
```

Creates an absolute URL from the specified protocol, host, and file. The port number used will be the default for the protocol.

Parameters:
protocol - the protocol to use
host - the host to connect to
file - the file on that host
Throws: MalformedURLException
If an unknown protocol is found.

Abbildung 3.6: von javadoc erzeugtes HTML-File

Beispiele:

```
int anzahl;
i = 10;
alter = (int) 25;
boolean abbruch = false;
int j = 3L;
a = b = c = 0x10;
char s = 'A';
char tab = '\t';
```

Arrays

- intern als Objekt
- Deklaration einer Array-Variablen; Erstellen des Array-Objektes; Initialisieren der Elemente
- unbalancierte Arrays möglich
- (unterschiedliche Dimensionen je Zeile möglich)

Beispiel:

```
int[] feld1;
int[] k = new int[5];
String[] wortListe = {"StP", "Java", "SUN", "DBS"};
k[3] = 3;
wortliste[0] = "Sign";
```

mehrdimensionale Arrays:

Beispiel:

```
int[][] arr;
arr = new int[3][4];
arr[1][2] = 10;
```

Klassen

Beispiele:

```
Button dieClearTaste;
```

Strings

- Strings sind Objekte (java.lang)
- automatische Erzeugung einer Instanz der Stringklasse bei Literalzuweisung
- Verketten von Strings mittels + bzw. +=
- Operator konvertiert automatisch „Nicht-Zeichenketten“ in einen Teilstring
- Vergleich zweier Strings mittels equals-Methode: if(farbe.equals(rot”))...;

Beispiele:

```
String testzeichenkette;
vorname = "Peter";
String wtag = "Montag";
String s1 = "Dieser Vortrag dauert " + 30*60 + " Sekunden.";
```

Konstanten

- keine Konstantenerstellung auf lokaler Ebene
- (innerhalb eines Methodenkörpers oder Blocks)
- Deklaration mittels Schlüsselwort final

Beispiele:

```
final double pi = 3.1415;
final int TOP = 100;
```

3.4.3.2 Operatoren

Kategorie	*Operator*	*Erklärung*	*Priorität*
Arithmetik	+	Addition (auch Strings)	5
	++	Inkrement	2
	-	Subtraktion (oder Verneinen)	5
	–	Dekrement	2
	*	Multiplikation	4
	/	Division	4
	%	Modulo	4
Zuweisung	=	einfach	11
	+=	inkl. Addition	11
	-=	inkl. Subtraktion	11
	*=	inkl. Multiplikation	11
	/=	inkl. Division	11
	%=	inkl. Modulo	11
	=++	inkl. Inkrement	11
	=–	inkl. Dekrement	11
	&=	inkl. UND	11
	— =	inkl. ODER	11
	<<=	inkl. Left-Shift	11
	>>=	inkl. Right-Shift (mit Vorzeichen)	11
	¿¿¿=	inkl. Right-Shift (ohne Vorzeichen)	11
Vergleich	= =	gleich	7
	!=	ungleich	7
	¡	kleiner	7
	>	größer	7
	¡=	kleiner oder gleich	7
	>=	größer oder gleich	7
log. Operator	!	Negation	3
	&&	UND	9
	——	ODER	9
bitw. Operator	&	UND	8
	—	ODER	8
		Komplement	3
		Exclusiv-Oder	8
	<<	Left-Shift (mit Vorzeichen)	6
	>>	Right-Shift (mit Vorzeichen)	6
	>>>	Right-Shift (ohne Vorzeichen)	6

Bemerkung:

- Tabelle enthält nur die wichtigsten Java-Operatoren
- kleinere Zahlen haben höhere Priorität
- gleiche Prioritäten werden von links nach rechts abgearbeitet

3.4.3.3 Verzweigungen

IF Anweisung

```
Methode 1:

if (boolean) {
        Anweisung1;
}
else {
        Anweisung2;
}

Methode 2:

if (boolean) {
        Anweisung1;
}
else if (boolean) {
        Anweisung2;
}
else {
        Anweisung3;
}
```

Bedingte Bewertung

```
(boolean) ? wennTrue : wennFalse;
```

- verkürzt für if ... else

SWITCH Anweisung

```
switch (test) {
        case wert1:
                Anweisungen;
                break;
        case wert2:
                Anweisungen;
                break;
        default:
                Anweisungen;
                break;
}
```

- Ersetzung von gehäuften if-Anweisungen
- Verzweigung zur entsprechenden case-Marke
- default (optional), wenn keine Übereinstimmung
- fehlt break, wird beim nächsten case fortgesetzt
- Typ der test-Variablen kann nur ein Grunddatentyp sein

3.4.3.4 Schleifen

FOR Schleife

```
for (initialization; test; increment) {
        Anweisungen;
}
```

- initialization: leitet Schleifenbeginn ein
- test: Abbruchkriterium (Schleifenende wenn false)
- increment: Manipulation des Schleifenzählers
- Abbruch durch break
- mittels continue zum Schleifenstart (nächster Schleifendurchlauf)

WHILE Schleife

```
while (test) {
        Anweisungen;
}
```

- test: Abbruchkriterium (Schleifenende wenn false)
- Abbruch durch break
- mittels continue zum Schleifenstart (nächster Schleifendurchlauf)

DO-WHILE Schleife

```
do {
        Anweisungen;
} while (test)
```

- test: Abbruchkriterium (Schleifenende wenn false)
- Abbruchbedindung test wird im Gegensatz zu for und while erst zum Schluß abgefragt
- Abbruch durch break
- mittels continue zum Schleifenstart (nächster Schleifendurchlauf)

3.4.4 Objektorientierte Programmierung

3.4.4.1 OOP in Java

- Klassenhierarchien als Packages (Pakete)

```
meineKlasse aClass;
aClass = new meineKlasse();
aClass.Print(5);
aClass.Offset = 0;
```

- new erzeugt neue Instanz einer Klasse (mittels eines Konstruktors)
- Manipulation von Objektvariablen bzw. Aufruf von Methoden mittels „.“
- Freigabe der nicht mehr benötigten Ressourcen erfolgt automatisch durch die Java-Runtime-Environment
- Umgebung (Garbage Collection); keine Destruktoren nötig
- finalize-Methode jedoch implementierbar (Destruktor)

3.4.4.2 Eigene Klassen schreiben

```
[public][final][static][abstract] class Klassenname
[extends Klassenname][implements Interface1,...] {
        Variablendeklarationen;
        Konstruktoren;
        Methoden deklarieren und implementieren;
        [Destruktor;]
}
```

Bezeichner	*Beschreibung*
public	erreichbar außerhalb des aktuellen Packages
final	kann nicht überschrieben werden
static	Klassenmethode (existieren einmal pro Klasse, gleich wieviele Instanzen)
abstract	ohne Implementierung; erfolgt in der Unterklasse
extends	explizite Angabe der Oberklasse (Superklasse) deren Variablen und Methoden geerbt werden
implements	deklariert einen Satz von Methoden und Konstanten ohne Spezifikation der Implementierung für eine der Methoden

- Konstruktor muß exakt den selben Namen haben wie die Klasse
- Konstruktoren erlauben Initialisierung während der Erzeugung
- Der Aufruf der Konstruktormethode erfolgt im Zusammenhang mit dem new-Operator
- Name-Overloading in Java erlaubt mehrere verschiedene Konstruktoren pro Klasse

3.4.4.3 Variablendeklarationen

[Zugriff-Spezifier][static][final][transient][volatile] type Name;

Spezifier	*Klasse*	*Unterklasse*	*Package*	*Welt*
private	X			
private protected	X	X		
protected	X	X	X	
public	X	X	X	X
friendly (standard)	X		X	

Bezeichner	*Beschreibung*
static	Klassenvariable (existieren einmal pro Klasse, gleich wieviele Instanzen) Standard: Instanzvariable (existiert je Instanz einmal)
final	Variable konstant
transient	nicht Teil eines beharrlichen Objektstatus
volatile	asynchrone Modifizierung

3.4.4.4 Methoden schreiben

```
returntype MethName([Datentyp1 Argumentname1[,Datentyp2 Argumentname2,...]]) {
        Methodenkörper
}
```

- Name-Overloading möglich (aber nicht mehr als eine mit derselben Signatur, alle den selben Returntype)
- innerhalb der Methode haben die Argumente die Bedeutung lokaler Variablen
- soll kein Ergebnis zurückgeliefert werden, dann void als Returntype
- Zuweisung des Rückgabewertes mittels return-Operator
- Datentypen von return und der Methodendeklaration müssen identisch sein
- wenn Objekttyp zurückgeliefert wird, muß das Objekt aus der deklarierten Klasse abgeleitet sein

- wenn Interfacetyp zurückgeliefert wird, muß das Objekt das spezifizierte Interface implementieren
- Argumente einfacher Datentypen (int, float, boolean, ...) und String werden als „pass by value“ (Kopie des Wertes wird übergeben; keine Änderungen des Argumentes möglich) übergeben
- Objekte als Argumente werden als „pass by reference“ (Referenz wird übergeben, Argumentänderungen wirken sich auch außerhalb der Methode aus) übergeben

3.4.4.5 Vererbung

- Vererbung mittels extends
- "Verstecken" von Mitgliedsvariablen der Superklasse durch Definition desselben Namens in der Unterklasse
- Zugriff auf diese "versteckte" Mitgliedsvariablen mittels super.Variable

```
class mySuper{
        int myX;
}

class myClass extends mySuper {
        double myX;             // versteckt myX aus mySuper
        super.myX = 5;  // greift auf int myX aus mySuper zu
}
```

- Überschreiben von Methoden zwecks Neuimplementation oder Erweiterung

```
class myClass extends mySuper {
        myClass() {
                super();                // Konstruktor der Superklasse
                // zusätzliche Initialisierung
        }
        myMethod() {            // überschreibt myMethod in mySuper
                super.myMethod();       // ruft myMethod von mySuper auf
        }
}
```

- Überschreiben nicht möglich, wenn als final oder static deklariert
- abstrakte Methoden der Superklasse müssen in der Unterklasse implementiert (überschrieben) werden bzw. Unterklasse muß als abstrakte Klasse deklariert werden
- Mehrfachvererbung nicht möglich; Interfaces als Alternative

3.4.4.6 Interfaces

Mechanismus, der es erlaubt, in hierarchisch nicht verwandten Klassen denselben Satz von Methoden zu implementieren und diese von außen anzusprechen. Sind frei von Abhängigkeiten einer speziellen Klasse oder Klassenhierarchie.

Wozu:

- Deklaration von Methoden, die eine oder mehrere Klassen implementieren werden
- gibt Aufschluß über das Programminterface eines Objektes, ohne Aufschluß über seine Klassen zu liefern (anonyme Objekte)
- Auffangen von Analogien nicht verwandter Klassen, ohne Klassenverwandtschaft zu erzwingen

```
[public] interface Interfacename [extends Interfacename1[,Interfacename2,...]] {
        Interfacekörper;
}
```

Bezeichner	*Beschreibung*
public	erreichbar außerhalb des aktuellen Packages
extends	explizite Angabe von Superinterfaces (mehrere möglich) deren Variablen und Methoden erweitert werden

- oft enden Interfaces mit einem "...able" (Runable,Printable)
- Interfacekörper enthält nur Methodendeklarationen und keine Methodenkörper
- zusätzlich Konstantendeklaration möglich
- eine Klasse, die Interfaces benutzt, muß alle Methoden implementieren, die im Interface deklariert sind (auch alle in Superinterfaces deklarierten Methoden)

Beispiel:

```
interface MyPrintable {
        static final double PAGEX_A4 = 21.0;
        getSeitenzahl();
        setSeitenzahl();
}
```

3.4.5 Klassenbibliotheken

3.4.5.1 Java-Packages

- Klassenbibliothek = Bündelung von verwandten Klassen, in Java Packages
- JDK besitzt gepackte Klassenbibliothek : classes.zip

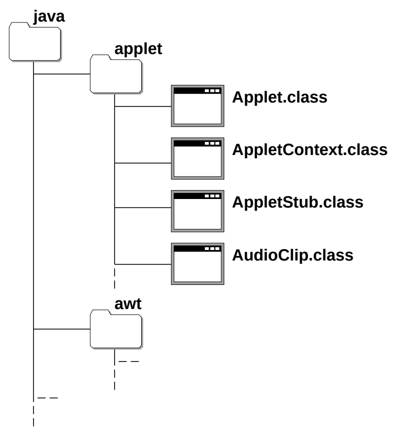


Abbildung 3.7: Java Klassenbibliothek

- Java liefert Sammelsurium von Packages:

Name	*Bezeichnung*	*Beschreibung*
Java Language Package	java.lang.*	- grundlegende Javasprachklassen Object, Thread, Exception, System, Integer, Float, Math, ... - wird automatisch durch den Compiler geladen
Java I/O Package	java.io.*	- Ein- und Ausgabestreams (Dateizugriffe)
Java Utility Package	java.util.*	- wichtige Datenstrukturen wie Stack, Vector, Dictionary, Hashtable, ... - Stringzerlegung - Bearbeiten von Datumswerten
Abstract Window Toolkit / Package	java.awt.*	- liefert GUI-Elemente (Fenster, Buttons, Panels, ...) - Unterpackage Image: Klassen und Interfaces für die Arbeit mit Bildern
Applet Package	java.applet.*	- enthält zu erbende Superklasse um ein Applet zu schreiben - AudioClip-Interface zur Abstraktion von Sounds
Java Networking Package	java.net.*	- Kapselung von URL (Uniform Ressource Locator) - Kapselung einer Internet IP-Adresse - Socket-Klasse für Client-/Server-Programme

3.4.5.2 Zugriff auf Packages

- mittels import
- Pfad und Klassenname des Packages sind anzugeben
- mehrere Klassennamen können über Wildcardzeichen ? erfaßt werden
- Default-Package: nur Angabe des Klassennamens ohne Pfad (Ablage ist im Verzeichnis, wo sich die Quelldatei befindet)

```
import java.applet.Applet;      // eine Klasse
import java.applet.*;           // alle Klasse
import myClass;                 // Default-Package
```

3.4.5.3 Eigene Packages

- package-Schlüsselwort erlaubt Gruppierung von Klassen und Interfaces zu einem Package
- Möglichkeit des Packens in ein Zip-Archiv

Vorgehensweise:

1. Hinzufügen folgender Zeile an den Beginn der Quelldatei: package gruppenname;

2. Compilieren jeder Quelldatei mit /d Option (Angabe der Plazierung in der Verzeichnisstruktur): javac /d myDir/classes *.java

-> Unterverzeichns myDir/classes/gruppenname wurde angelegt und enthält entsprechende *.class Dateien

3.4.6 Techniken der Programmentwicklung

3.4.6.1 Applets

siehe Kapitel Einführungsbeispiel

Für Programme, die über das WWW ausgeführt werden sollen, ist dies die bevorzugte Art. Hierbei wird die .class Datei vom (javafähigen) Browser ausgeführt.

3.4.6.2 Anwendungen

Da in diesem Fall einige Schwierigkeiten zu erwarten sind, wenn man ein solches Programm via WWW benutzen möchte, werden die meisten Java-Programme wohl nicht als Anwendung erstellt werden. In unserem Fall jedoch (schließlich wollen wir das Programm direkt am Rechner benutzen und nicht über das WWW benutzen), ist es „die“ Wahl.

3.4.6.3 Multithreading

Ein Thread ist ein einzelner sequentieller Fluß innerhalb eines Programms, wobei mehrere Threads nebenläufig abgehandelt werden können.

In Java sieht das wie folgt aus:

Alle Aktionen eines Threads werden in seiner run-Methode implementiert, die nach Erzeugen und Initialisieren des Threads vom Runtimesystem gestartet wird. Dabei hat man bei der Implementation zwei Varianten zur Auswahl. Zum einen kann man aus der Thread-Klasse eine Unterklasse ableiten und die run-Methode Überschreiben, oder man implementiert das Runable-Interface in seiner eigenen Klasse. Im Zweifelsfall sollte man auf die zweite Möglichkeit zurückgreifen, um ein weiteres Ableiten der eigenen Klasse zu ermöglichen. Dies kann dann wie folgt aussehen:

```
class ARun implements Runable {
        public void run () {
                // hier steht der Threadkörper...
}
```

```
}
Thread aThread;
ARun aRunI;
aRunI = new aRun();
aThread = new Thread(aRunI);
aThread.start();
```

Jeder Thread kann im Laufe seines Lebens folgende Zustände annehmen:

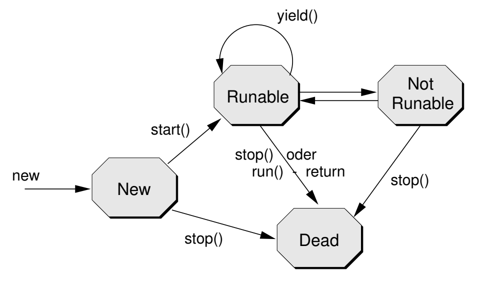


Abbildung 3.8: Zustände eines Threads

Dabei sind folgende Methoden für jeden Thread zuständig: new (zum Erzeugen), start (allokiert Systemresourcen und ruft run-Methode auf), stop (bewirkt daß ein Thread abgebrochen wird) und isAlive (liefert true, wenn der Thread gestartet aber noch nicht gestoppt wurde).

3.4.6.4 Fehlerbehandlung und Exceptions

Eine Exception ist ein Ereignis, das während der Ausführung des Programms auftritt und den normalen Fluß der Befehle unterbricht. Ursachen für solche Exceptions können diverser Natur sein, z.B. könnte versucht worden sein, auf ein Arrayelement außerhalb der Arraygrenzen zuzugreifen.

Ein sogenannter Exceptionhandler besteht im allgemeinen aus mehreren Blöcken. Am Anfang steht der try-Block, danach folgen ein oder mehrere catch-Blöcke oder ein finally-Block, wobei sich letzterer dann anbietet, wenn der Code in jedem Fall auszuführen ist.

Der try-Block beinhaltet die Anweisung(en), die eine Exception auslösen könnte(n). Direkt im Anschluß folgt der catch-Block, der auf die Exception eingeht. Diesen Blöcken kann der finally-Block folgen.

```
DateiLesen {
        try {
                openFile;
                Größe bestimmen;
        }
        catch (Größe nicht bestimmbar) {
```

```
                do_something;
        }
        finally {
                closeFile;
        }
}
```

```
Natürlich muß eine Exception auch noch definiert werden, dies geschieht mit:
class MyExcept extends Exception {}
Diese wird dann erzeugt durch:
throw new MyExcept ();
```

3.4.6.5 Netzwerke und URLs

Netzwerkfähigkeit / Kommunikationsprinzip:

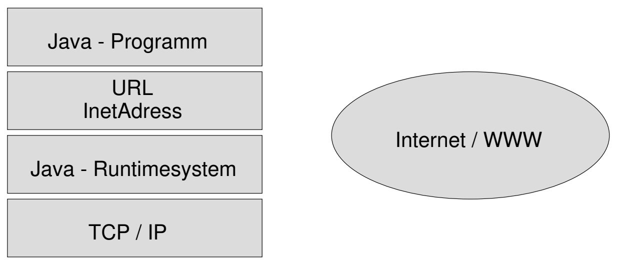


Abbildung 3.9:

- URL-Object erstellen:

```
URL gamelan = new URL("http://www.gamelan.com/");       // relative URL
URL gamelanNetw = new URL(gamelan,"Network.html");      // absolute URL
URL aURL = new URL("http:","//www.gamelan.com",80,"/Network.html");
```

- Verbindung herstellen (für erweiterte Verbindung):

```
URLConnection gamelanCon;
gamelanCon = gamelan.openConnection();
```

- lesen (und schreiben):

```
DataInputStream dis;
String inputLine;
dis = new DataInputStream(gamelan.openStream());
while ((inputLine = dis.readline()) != null) {
        System.out.println(inputLine);
}
```

3.4.6.6 Client/Server-Prinzip und Sockets

- Grundlage vieler moderner unternehmensweiter (oder größer) Anwendungen
- mittels Sockets - Kommunikationsverbindung zwischen zwei Programmen
- java.net stellt Klassen Socket und ServerSocket zur Verfügung
- garantiert plattformunabhängige Kommunikation
- TCP/IP-Kommunikation möglich: Übertragung von Datenpaketen mit Empfangsbestätigung
- Datagram-Verbindung: Datenübertragung ohne Empfangsbestätigung

```
// minimale Client/Server Anwendung : SERVER

import java.awt.*;
import java.net.*;
import java.io.*;
class aServer {
   public static void main( String args[ ] ){
      ServerSocket servsock = (ServerSocket) null;
      Socket sock1;
      String msg = "Hallo Welt !";
      OutputStream outStrm;
      try {
         // Port: 1234 und timeout 400 Sekunden
         servsock = new ServerSocket(1234, 400);
      } catch( IOException e ) { }
      while( true ) {
         try {
            // auf eine Verbindung warten
            sock1 = servsock.accept( );
            outStrm = sock1.getOutputStream( );
            // Senden
            for( int i = 0; i < msg.length();i++) {
               outStrm.write( (int) msg.charAt( i ) );
            }
            // den Socket schließen ( nicht ServerSockert )
            sock1.close( );
         } catch ( IOException e ) { }
      }
   }
}


// minimale Client/Server Anwendung : CLIENT
```

```
import java.awt.*;
import java.net.*;
import java.io.*;
class aClient {
   public static void main( String args[ ] ){
      Socket sock;
      int c;
      String msg = "Hallo Welt !";
      InputStream inStrm;
      // Öffnen an Port 1234
      sock = new Socket( "a_host", 1234 );
      inStrm = sock.getInputStream( );
      while( c = inStrm.read() != -1 ) {
         System.out.print( (char) c );
      }
      sock.close( );
   }
}
```

Kapitel 4

Konstruktionsphase I

In diesem Kapitel soll der Entwurf der statischen Sicht (Objektmodell) erläutert werden. Meilensteine der 1. Konstruktionsphase sind die SIGN Modeling Language, deren Konzepte bereits in der Vorbereitungsphase in Grundzügen beschrieben wurden, syntaktisch und semantisch aber erst hier explizit festgelegt werden - und der Systementwurf des Editors deSIGN.

4.1 Systementwurf I

4.1.1 Vorwort

4.1.1.1 Aufgabenstellung

Zur Erstellung eines Entwurfs wurde im Projekt SIGN ein Ausschuß gebildet. Die Aufgabe des Ausschusses ist die Ausarbeitung einer formalen Spezifikation der graphischen Entwicklungsumgebung.

Grundlage für den Entwurf sind insbesondere die Ergebnisse aus den folgenden Arbeitspaketen:

Nr.	Arbeitspaket	Beschreibung	Ergebnisse	h	Beginn	Ende	Verantwortlich
1320	Basisnotation	Entwurf der grundlegenden Notation für das Objektmodell der SIGN Modeling Language, Auswahl der Modellierungskonzepte anhand der Features List, Ausarbeiten einer Präsentation der Notation für das Basismodell, Vortrag am 24.01.97, 8:00h, MZH 7250, ca. 40 Min. Dauer	Bericht, Vortrag	30	10.01.97	14.02.97	Ersin
1330	Metamodell	Definition der Syntax für die Basisnotation der SIGN Modeling Language(SIGN ML). Erstellung eines Metamodells in SIGN ML. Entwurf der grundlegenden Notation für das Objektmodell, formelle Spezifikation (wie UML). Dient als Grundlage für die formelle Spezifikation der Anwendungsgebietskomponente des graphischen Editors (AP 232)	Bericht	40	24.01.97	07.02.97	Frank, Andreas
2200	Anforderungsdef. graphische Entwicklungsumgebung	informelle Spezifikation der Anforderungen an die graphische Entwicklungsumgebung			29.11.96	20.12.96	Holger, Martin, Verena
2210	Anforderungen an die allgemeine Funktionalität	informelle Spezifikation der Anforderungen an die Funktionalität des graphischen Editors, Vortrag am 31.01.97, 8:00h, MZH 7250, ca. 20 Min. Dauer	Bericht, Vortrag	30	29.11.96	10.01.97	Verena
2220	Anforderungen an die Benutzungsschnittstelle	informelle Spezifikation der Anforderungen an die Benutzungsschnittstelle des graphischen Editors, Vortrag am 31.01.97, 8:00h, MZH 7250, ca. 20 Min. Dauer	Bericht, Vortrag	20	29.11.96	10.01.97	Marc, Martin
2230	Anforderungen an die Dateischnittstelle	informelle Spezifikation der Anforderungen an die Dateischnittstelle des graphischen Editors, Vortrag am 31.01.97, 8:00h, MZH 7250, ca. 20 Min. Dauer	Bericht, Vortrag	20	29.11.96	10.01.97	

Ferner wurden im WS 96/97 im Projekt SIGN Referate zum Thema OMT, UML und Java gehalten und schriftlich zusammengefaßt. Die Inhalte dieser Referate sind wesentliche Hintergrundinformationen, die sowohl in den o.g. AP's als auch im Entwurf berücksichtigt werden.

Als Arbeitsergebnis legt der Entwurfsausschuß bis zum 30.05.97 die Version 1.0 des Entwurfs als Dokument vor. Ferner wird eine Präsentation für die Vorstellung im Plenum ausgearbeitet. Die im Ausschuß erarbeiteten Ergebnisse sollen dann in eine druckbare Form gebracht werden; hierfür ist jeweils eine Untergruppe verantwortlich.

Hier eine Übersicht über die dem Ausschuß übertragenen Arbeitspakete:

Nr.	Arbeitspaket	Beschreibung	Ergebnisse	h	Beginn	Ende	Verantwortlich
2300	Entwurf graphische Entwicklungsumgebung	formelle Spezifikation des graphischen Editors, insbesondere Objektmodell, evtl. auch Verhaltens- und Funktionssicht			19.04.97	30.05.97	
2310	Architektur	Skizzierung der Anwendungsarchitektur bzw. des modularen Aufbaus des graphischen Editors, Spezifikation des Framework der Anwendung, Berücksichtigung von UML	Vortrag, Bericht	20	19.04.97	30.05.97	Arne, Marcus
2320	Funktionalität	formelle Spezifikation der Anwendungsgebietskomponente des graphischen Editors nach der OMT-Methode, aufbauend auf AP 1330	Vortrag, Bericht	40	19.04.97	30.05.97	Frank, Andreas
2330	Datenverwaltung	formelle Spezifikation der Datenverwaltungskomponente des graphischen Editors mit OMT-Methode, Berücksichtigung von UML	Vortrag, Bericht	40	19.04.97	30.05.97	Holger, Hamid, Rainer
2340	Benutzerinterakt.	formelle Spezifikation der Benutzerinteraktionskomponente des graphischen Editors mit OMT-Methode, Berücksichtigung von UML	Vortrag, Bericht	40	19.04.97	30.05.97	Verena, Martin

4.1.1.2 Einordnung in den Projektablauf

Phasenplan zum Ablauf des Projekts SIGN:

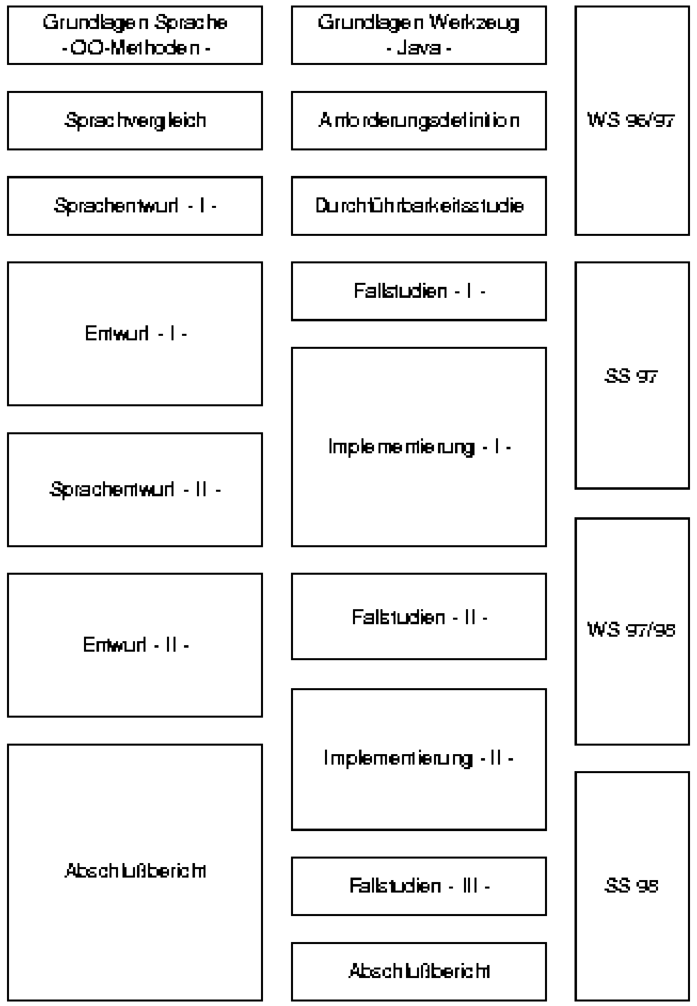


Abbildung 4.1: Phasenplan

Die vorliegende Version 1.0 des Entwurfs ist das Ergebnis der Phase Entwurf I im Phasenplan des Gesamtprojekts. Der Projektplan sieht vor, im direkten Anschluß mit dem Entwurf weitergehender Konzepte für die Notation SIGN Modeling Language (SIGN ML) fortzufahren (Phase Sprachentwurf II). Daran anschließend wird in der Phase Entwurf II der vorliegende Entwurf der graphischen Entwicklungsumgebung im Hinblick auf die Erweiterung der Notation überarbeitet.

4.1.2 Entwurf

Der Entwurf wurde in mehreren Phasen bzw. Arbeitspaketen erstellt, deren Ergebnisse hier in aufeinanderfolgenden Kapiteln präsentiert werden. Die Arbeitspakete wurden nacheinander im Entwurfsausschuß behandelt und anschließend in Untergruppen parallel ausgearbeitet.

Entwurfsprobleme

Als erstes werden einige grundlegende Probleme angesprochen, zu deren Lösung wichtige Entwurfsentscheidungen getroffen wurden.

Architektur

Im zweiten Schritt wurde die übergeordnete Architektur des Systems festgelgt. Hierzu wurden Subsysteme und Layer identifiziert und mittels Box-and-Line-Drawings illustriert.

Struktur- und Datensicht

Im nächsten Schritt haben wir uns mit der Struktur- und Datensicht auf das System beschäftigt. Dabei wurden Klassendiagramme erstellt und die Verantwortlichkeiten der Klassen festgelegt.

Funktionale Sicht

Die vierte Phase behandelt die funktionelle Sicht auf das System. Hierzu wurden Use Cases (Anwendungs- oder Benutzungsfälle) ermittelt und Datenflußdiagramme bzw. Activity Charts erstellt.

Dynamische Sicht

Zur Darstellung von Objektlebenszyklen und zur Spezifikation von Methoden wurden in Phase 5 State Charts bzw. Zustandsdiagramme verwendet. Dies ist die dynamische Sicht auf das System.

In der 4. und 5. Phase wurden nur einige interessante Funktionen bzw. Systemaufgaben detailliert behandelt. Gerade in diesen Phasen werden jedoch die benötigten Operationen und evtl. weitere Verantwortlichkeiten für Klassen identifiziert. Um möglichst schnell mit der Implementierung eines Prototypen beginnnen zu können, haben wir auf eine detailliertere Spezifikation im Entwurfsdokument verzichtet. Sind alle Operationen bekannt, könnte der Entwurf um ein weiteres Kapitel mit vollständigen Klassenbeschreibungen inklusive Spezifikation der Methoden (mittels Pseudocode oder State-Charts, o.ä.) ergänzt werden. Die Spezifikation der Methoden erfolgt direkt im Code der Programmiersprache Java während der prototypischen Implementierung des Systems. Die dabei gewonnenen Erkenntnisse werden in die erweiterte Version des Entwurfs einfließen.

4.1.2.1 Entwurfsprobleme

Modellstruktur

Wie soll die interne Repräsentation eines Modells aussehen? Die interne Darstellung des Modells soll auf dem Metamodell aufbauen. Das Metamodell beschreibt die Syntax der Basisnotation. Es definiert die logischen Modellbestandteile als Klassen und beschreibt ihre Beziehung zueinander. Die logischen Elemente des Modells haben eine festgelegte Syntax und Semantik. Sie sind die Grundlage für die Übersetzung eines Modells in den Code einer Programmiersprache. Darüber hinaus hat das Modell einen informellen Zweck. Wichtiger noch als die formelle Beschreibung und das Generieren von Programmcode scheint die Möglichkleit zur Strukturierung und übersichtlichen Darstellung eines Modells sowie die Ausgabe auf dem Papier zu sein. Im Hinblick auf die Modellstruktur ist zwischen logischen und anderen Modellbestandteilen zu Unterscheiden. Die logischen Modellelemente bilden zur Laufzeit des entstandenen Programms einen abstrakten Syntaxbaum - die Struktur ist durch das Metamodell vorgegeben.

Das gesamte System soll als graphische Entwicklungsumgebung der Spezifikation von Software dienen. Es soll ein Werkzeug zur Unterstützung des Entwurfs mit objektorientierten Methoden sein. Das Ergebnis der Spezifikation ist ein Modell. Das Modell enthält alle Aussagen über die Systemeigenschaften, die der Designer spezifiziert. Zur Spezifikation werden Diagramme verwendet. Zur Spezifikation verschiedener Aspekte des Systems werden verschiedene Diagrammtypen verwendet.

Klassendiagramme dienen der Spezifikation der Daten- und Struktursicht auf das Modell. Die erste Ausbaustufe der graphischen Entwicklungsumgebung enthält einen Editor für diesen Diagrammtyp.

Activity Charts werden verwendet, um die funktionelle Sicht auf das System zu modellieren. Sie sind Gegenstand der zweiten Version des Sprachentwurfs und sollen in einer weiteren Ausbaustufe der graphischen Entwicklungsumgebung enthalten sein.

State-Charts können zur Darstellung der dynamischen Sicht auf des Modell verwendet werden. Der erweiterte Sprachentwurf wird auch diese Sicht auf das System berücksichtigen.

Die verschiedenen Sichten auf das System, das modelliert wird, geben jeweils unterschiedliche Aspekte der Spezifikation wieder. Im Zentrum des zu unterstützenden Entwurfsprozesses steht jedoch die konsistente Spezifikation eines

Software-Systems durch Erstellung eines Modells. Der Designer betrachtet die verschiedenen Diagramme als zusammengehörige Bestandteile eines Modells.

Wie oben bereits angedeutet, wird daher zwischen der Modellebene bzw. logischen Elementen und der Diagrammebene bzw. Diagramm-Elementen unterschieden. Es ist hingegen nicht vorgesehen, die Modellebene direkt bearbeiten zu können. Eine direkte Bearbeitung der Modellebene könnte mittels eines Klassenbrowsers oder ähnlichem erfolgen. Stattdessen sieht unser Entwurf vor, daß auf die Modellebene nur indirekt über Diagramme Einfluß genommen wird. D.h. das Modell wird implizit über die Sichten bzw. Diagramme spezifiziert. Daher wird der Modell-Begriff an der Benutzeroberfläche kaum verwendet.

Wir gehen davon aus, daß bei einer objektorientierten Vorgehensweise beim Software-Entwurf das Klassendiagramm eine zentrale Rolle spielt. Ein typisches Vorgehen unter Benutzung unseres Werkzeugs legt nahe, zunächst ein Klassendiagramm zu erstellen. Anschließend können die bereits im Modell vorhandenen Elemente (Klassen, Attribute, Operationen, Assoziationen, ...) bei der Spezifikation der anderen Sichten auf das Modell verwendet bzw. referenziert werden. Nicht alle objektorientierten Methoden schlagen diese Reihenfolge beim Entwurf vor. So sind die objektorientierte Systemanalyse nach Heide Balzert [Bal96], die OOSE-Methode nach Rubin und Goldberg [RG92]und die OBS-Methode nach Jacobson [JCJO92] mehr an der funktionelle Sicht als Ausgangspunkt für die Modellbildung orientiert. Der erweiterte Entwurf für eine spätere Ausbaustufe der Entwicklungsumgebung sollte diesen Umstand berücksichtigen.

Layout

Wie werden die Modellelemente präsentiert? Der Benutzer soll die Möglichkeit haben, Modellelemente (z.B. eine Klasse) auf einem Blatt zu plazieren. Die Position und Größe der Modellelemente soll vom Benutzer direkt zu bearbeiten sein. Ein Modell soll zwecks Übersichtlichkeit auf verschiedene Blätter mit einstellbarer Größe zu verteilen sein. Die Layout-Information (etwa die gewollte Anordnung von Superklassen oberhalb von Subklassen, oder auch die Seiteneinteilung) ist zwar vom Standpunkt der Semantik unserer Basisnotation ohne Bedeutung, jedoch für die praktische Nutzbarkeit eines Modells nicht unerheblich. Die übersichtliche Strukturierung des Modells kann nicht automatisch durch den Rechner erfolgen. Sie muß explizit vom Benutzer vorgenommen werden. Bei der praktischen Anwendung des Systems wird sicher ein Großteil der Arbeit auf diese Aufgabe entfallen. Die Bedeutung der Layout-Information ist daher hoch einzuschätzen. Da die Modellelemente zum Teil auch auf mehreren Blättern eines Modells gleichzeitig erscheinen dürfen, muß die Layout-Information von der Repräsentation der Modellelemente getrennt gehalten werden. Dasselbe Modellement kann auf verschiedenen Blättern in einem anderen Kontext und mit anderem Aussehen und anderer Position erscheinen.

Die in der Anforderungsdefinition genannte Möglichkeit der automatischen Anordnung von Elementen in einem Diagramm kann lediglich als Hilfestellung dienen. Es kann ein Algorithmus implementiert werden, der Regeln wie die Anordnung von Klassensymbolen von links nach rechts und oben nach unten mit festem Abstand auf einer Seite, o.ä. respektiert. Die Auswahl der Klassen, die zusammen auf einer Seite darzustellen sind, kann hingegen kaum automatisch erfolgen.

Die Darstellung der Modellbestandteile ist durch Eigenschaften wie Linienart und -dicke, Fontart und -größe, etc. bestimmt. Diese Eigenschaften bestimmen den Stil der Darstellung. Verschiedene Elementarten (Klasse, Attribut, Operation, Assoziation,...) können einen spezifischen Stil bei der Darstellung haben. Der Stil für die verschiedenen Elementarten sollte zumindest für jedes Diagramm individuell einstellbar sein - evtl. sogar zusätzlich für einzelne Seiten des Diagramms.

Ferner sollte der Detailierungsgrad der Darstellung einstellbar sein. Es sollte möglich sein, für jede Elementart einen Detailierungsgrad anzugeben. Z.B. Klasse mit/ohne Attribute, Operationen mit/ohne Parameter, Assoziationen mit/ohne Namen, etc. Auch hier ist denkbar, diese Eigenschaften für ein ganzes Diagramm, eine ganze Seite oder ein einzelnes Element anzugeben.

Wir sehen im Entwurf vor, den Stil und Detailierungsgrad jedem einzelnen Element zuzuordnen. Ferner sollen Bearbeitungsoptionen vorgesehen werden, die es dem Benutzer ermöglichen, diese Eigenschaften für ganze Gruppen von Elementen gleichzeitig einzustellen.

Präsentation

Neben der SIGN-Notation (an UML angelehnt) sind viele weitere Notationen zur Zeit üblich: OMT, Booch, etc. Die in den Notationen verwendeten Konzepte in Klassendiagrammen gleichen sich weitgehend. Die graphische Repräsentation ist hingegen sehr unterschiedlich. Wünschenswert ist, das SIGN Modellierungswerkzeug an eine andere Notation anpassen zu können, ohne es völlig neu schreiben zu müssen. Die Anwendung des Entwurfsmusters Abstrakte Fabrik erlaubt es, die Objekterzeugung so zu Kapseln, daß die hierzu notwendigen Änderungen minimal sind. Die Modellelemente wie Klasse, Generalisierung, etc. werden nicht durch direkten Aufruf des new-Operators, sondern mit Hilfe einer abstrakten Fabrik konstruiert.

Verwendung vorhandener Komponenten

Das Ziel des Projekts ist nicht, eine neue GUI-Bibliothek für die Präsentation von graphischen Objekten zu erstellen. Sofern dies möglich ist, soll auf vorhandene Komponenten zurückgegriffen werden. Die Konzentration auf die Probleme des Anwendungsgebietes soll hohe Priorität haben. Das bedingt eine Vermeidung von zu großem Aufwand in Randbereichen - die Gefahr des Verzettelns soll vermieden werden. Konkret soll die Nutzung der im JDK vorhandenen Komponenten

GUI-Bibliothek

Objekt-Serialisierung / Persistenz

bereits im Entwurf vorgesehen werden. Gleichzeitig soll die Nutzung anderer Komponenten / Mechanismen nicht ausgeschlossen werden. Daher werden die Benutzungsschnittstelle und die Datenverwaltungsschnittstelle als Subsysteme betrachtet, die (wie Layer) oberhalb bzw. unterhalb des Subsystems Anwendungsgebietskomponente angeordnet sind. Die Subsysteme haben eine definierte öffentliche Schnittstelle. Sie sind daher austauschbar und eine Portierung des Systems mit Benutzung einer anderen GUI-Bibliothek oder eines anderen Datenspeichers wird erleichtert.

Benutzungsfunktionalität

Kommandos können mehrere Auslöser haben (z.B. Menü und Button). Jedes Kommando soll jedoch nur einmal implementiert werden. Erforderlich ist daher eine einheitliche Schnittstelle für das Aufrufen von Kommandos. Kommandos sollen (soweit möglich) auch Rückgängig gemacht werden können. Die Verwendung des Entwurfsmuster Kommando bietet sich daher an.

Übersetzung

Für eine spätere Version der graphischen Entwicklungsumgebung ist auch die Übersetzung nach Postscript, SQL und Java-Code vorgesehen. Hierzu ist eine Möglichkeit vorzusehen, den abstrakten Syntaxbaum zu traversieren. Da viele verschiedene Übersetzungen denkbar sind, sollte der Entwurf in dieser Hinsicht flexible Erweiterungen zulassen. Es empfiehlt sich, den Code-Erzeugungsalgorithmus in einem Objekt zu kapseln, um ihn so nach Bedarf austauschen zu können. Hierzu kann das Entwurfsmuster Besucher verwendet werden.

Zur Code-Erzeugung kann auch auf vorhandene SGML-Werkzeuge zurückgegriffen werden. Die Syntax der SIGN ML kann in einer SGML DTD repräsentiert werden. In der graphischen Entwicklungsumgebung würde dann ein Code-Erzeugungsalgorithmus implementiert, der ein SGML Dokument gemäß der in der DTD beschriebenen Syntax erzeugt. Um aus diesem Dokument z.B. eine Postscriptdatei zu erzeugen, ist eine weitere Übersetzung notwendig, für die jedoch Werkzeuge bereits vorhanden sind.

In der ersten Ausbaustufe der graphischen Entwicklungsumgebung ist noch keine Übersetzung vorgesehen - wohl aber die Möglichkeit der Speicherung von Modelldaten. Daher ist es ratsam, zunächst nur einfache Serialisierung der Objekte unter Ausnutzung der vom JDK bereitgestellten Verfahren zu realisieren.

Um die geplante Erweiterung der graphischen Entwicklungsumgebung zur Code-Erzeugung später ohne Verzögerung durchführen zu können, soll die Verwendung des Besuchermusters [GHJV96]in diesem Entwurf bereits vorgesehen werden.

Ferner soll der Entwurf ein eigenes Verfahren zur Handhabung der Objektidentität persistenter Objekte vorsehen. Damit soll ein evtl. späterer Übergang zur Speicherung der Daten in relationalen Datenbanksystemen oder im Textformat (bzw. SGML) erleichtert werden.

4.1.2.2 Architektur

In diesem Abschnitt wird die übergeordnete Sytemarchitektur beschrieben. Ziel ist es, das System in mehrere überschaubare und möglichst unabhängige Subsysteme zu unterteilen. Die von uns gewählte Systemarchitektur ist zweidimensional: Eine Dimension beschreibt die Gliederung nach funktionellen Gesichtspunkten des Anwendungsgebiets, die andere Dimension ist die Unterteilung in die Komponenten Benutzerschnittstelle, Anwendungsgebiet und Datenverwaltung.

Anwendungssubsysteme

In der ersten Ausbaustufe wird die graphische Entwicklungsumgebung eine IDE sowie einen graphischen Editor für Klassendiagramme enthalten. Weitere Ausbaustufen umfassen die im Diagramm Abbildung 4.2 gepunktet dargestellten Komponenten.

IDE

Die integrierte Entwicklungsumgebung unterteilt sich in den graphischen Editor und den Code Generator (siehe hierfür auch Abbildung 4.2). Sie soll im Hinblick auf die Einbindung weiterer Anwendungssubsysteme erweiterbar sein. Im folgenden werden die Teile graphischer Editor und Code Generator weiter ausgeführt.

- **Ausbaustufe grapischer Editor**

  Zunächst enthält das Subsystem graphischer Editor nur einen Editor zur Erstellung von Klassendiagrammen. In späteren Ausbaustufen sollen Editoren für Instanzdiagramme, funktionelle Sicht (Activity Charts) und dynamische Sicht (State Charts) hinzukommen.

- **Ausbaustufe Code Generator**

  Das Subsystem Code Generator wird erst in der zweiten Ausbaustufe der graphischen integrierten Entwicklungsumgebung verwirklicht werden. Im ersten Schritt ist ein Generator zur Ausgabe von SQL-Code (Script-Generator) für relationale DB geplant. Weiter ist ein Java-Code-Generator für Klassen-Gerüste bzw. Interfaces und Funktionsrümpfe sowie ein Generator zur Erzeugung Postscript für Druckausgabe geplant. Die mit der ersten Ausbaustufe der graphischen Entwicklungsumgebung erstellbaren Klassendiagramme ermöglichen bereits die Architektur und die Signatur der Komponenten objektorientierter Software informell zu spezifizieren. Zusammen mit textuellen Erläuterungen, insbesondere zur Verantwortlichkeit der Klassen, kann das Klassendiagramm als Entwurfsunterlage dienen. Die Möglichkeit der Druckausgabe ist daher von hoher praktischer Relevanz. Da der Anwendungsbereich für relationale Datenbanksysteme zur persistenten Speicherung von Objektdaten groß ist, wird - unabhängig von der zur Implementation der mit der Entwicklungsumgebung spezifizierten Software gewählten Programmiersprache - von einer Übersetzung in ein relationales Datenbankschema weiterer praktischer Nutzen erwartet.

optionale Erweiterungen:

- **Projektmanager / Modulverwaltung**

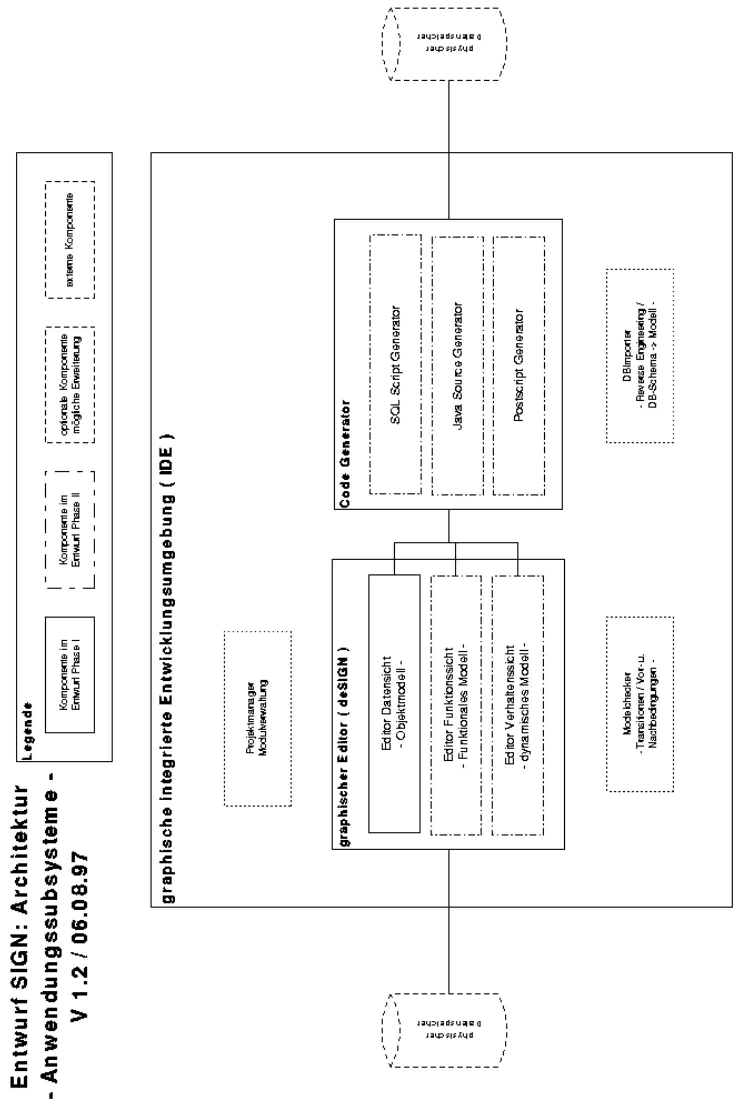


Abbildung 4.2: Architektur: Anwendungssubsysteme und IDE

Der Projektmanager könnte die Verwaltung mehrerer Modelle eines Projektes erlauben und dabei auch das Sharing von Modulen oder Klassen etc. zwischen verschiedenen Modellen unterstützen, eine Versions- und Benutzerverwaltung bieten und auch Multiuser-Fähigkeiten besitzen.

- **Modelchecker**

  Ein Modellchecker könnte die Prüfung eines Modells mit Spezifikation der dynamischen Sicht unter Verwendung von Invarianten sowie Vor- und Nachbedingungen übernehmen.

- **DBImporter**

  Mit Hilfe eines Programms zum Reverse Engineering wäre das das Erzeugen von Diagrammen aus einem bestehenden Datenbankschema möglich.

Komponenten / Layer

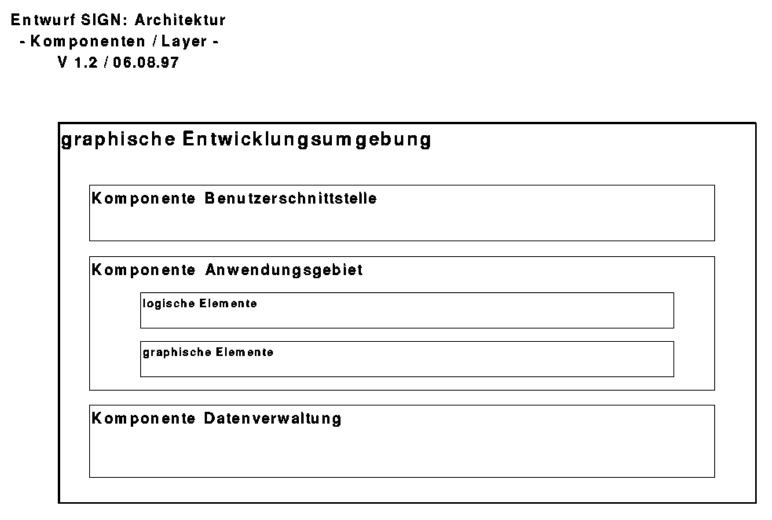


Abbildung 4.3: Architektur: Komponenten / Layer

Die Komponenten des Systems sind als Layer aufzufassen. Der mittlere Layer, die Komponente Anwendungsgebiet, wird durch den oberen Layer, die Benutzungsschnittstellenkomponente sowie den unteren Layer, die Datenverwaltungskomponente, von den äußeren Schnittstellen des Systems isoliert. Dieses soll durch Abbildung 4.3 verdeutlicht werden.

Komponente Benutzungsschnittstelle (HIC)

Die Benutzungsschnittstellenkomponente (= HIC = Human Interaction Component) stellt die Schnittstelle zur graphischen Benutzeroberfläche dar. Hierbei können zwei Strategien verfolgt werden.

Die eine Möglichkeit besteht in der Bereitstellung einer transparenten Schnittstelle zum GUI (Graphical User Interface, hier: AWT aus dem JDK 1.1.1), die von der PDC (PDC = Problem Domain Component = Komponente Anwendungsgebiet) verwendet werden. Die HIC stellt dabei als transparente Schnittstelle einen Satz elementarer Operationen zur Verfügung, die die Benutzerinteraktion unterstützen. Dies sind z.B. Operationen wie drawLine(POINT, POINT) zur Ausgabe von Grafik auf dem Bildschirm. Bei einer Portierung auf ein anderes GUI sind dann lediglich diese elementaren Operationen für das spezifische Interface neu zu implementieren.

Eine andere Möglichkeit ist die Erstellung einer vollständig GUI-spezifischen Benutzeroberfläche unter Verwendung von Komponenten, die bereits in der GUI-Bibliothek (hier: AWT) fertig vorhanden sind. In diesen Fall wird die HIC auf die Schnittstelle der PDC zugreifen, um z.B. Daten auszugeben. Die Zugriffsrichtung ist hierbei umgekehrt (im Vergleich zur o.g. ersten Möglichkeit). Der Vorteil dieser Variante ist die Nutzung der in der GUI-Bibliothek vorhandenen Komponenten z.B. zur graphischen Präsentation der Elemente aus der PDC.

Wir verwenden eine Kombination aus beidem. Die einzelnen Modellelemente zeichnen sich selber (mittels transparenter Schnittstelle zum GUI), aber die Eigenschaftsdialoge sind GUI-spezifische Module, die auf die Schnittstelle von Modellelementen zurückgreifen.

Komponente Anwendungsgebiet (PDC)

Die Grundlage für die Anwendungsgebietskomponente ist das Metamodell. Die im Metamodell vorliegenden Analyseergebnisse werden im Entwurf verfeinert und ergänzt. Das Metamodell beschreibt die Syntax der von uns gewählten Notation. Es repräsentiert die Struktur eines abstrakten Syntaxbaums. Nicht alles, was zur Erstellung eines Diagramms benötigt wird, ist jedoch durch die im Metamodell beschriebene Syntax schon erfaßt. Es fehlen noch Informationen zur graphischen Präsentation der Elemente in einem Diagramm sowie Möglichkeiten zur übersichtlichen Strukturierung eines Modells, z.B. Aufteilung auf mehrere Diagrammseiten. Die Bestandteile des Metamodells werden als logische Elemente bezeichnet, da sie die logische Struktur des Modells bzw. die Syntax der graphischen Notation beschreiben. Sie werden aufgrund obiger Überlegungen ergänzt durch die graphischen Elemente. Diese tragen im Gegensatz zu den logischen Elementen nichts zur Semantik des durch das Modell spezifizierten Systems bei.

Komponente Datenverwaltung (DMC)

Die Datenverwaltungskomponente (= DMC = Data Management Component) ist eine transparente Schnittstelle zur Handhabung persistenter Daten aus der Anwendungsgebietskomponente. Die einzelnen Methoden zur Speicherung der Daten sind hinter dieser Schnittstelle gekapselt. Techniken zur Implementation einer persistenten Datenspeicherung sind u.a.: das vom JDK unterstützte

Serialisierungsverfahren, SGML-Dateien und Ablegen in einer Datenbank. Die Vor- und Nachteile der einzelnen Möglichkeiten sind Grund für mehrere Diskussionen im Plenum gewesen. Wir haben beschlossen, zunächst eine Lösung mit einfacher Serialisierung der persistenten Objekte unter Ausnutzung der vom JDK gebotenen Möglichkeiten zu realisieren.

4.1.2.3 Struktur- und Datensicht

Komponente Anwendungsgebiet

Die Komponente Anwendungsgebiet wird beschrieben durch die Struktur der logischen und graphischen Modellelemente bzw. Diagrammelemente. Die logischen Elemente entstammen dem Metamodell. Für sie gilt die festgelegte Syntax und Semantik der Modellierungssprache (SIGN ML). Darüberhinaus umfaßt die Anwendungsgebietskomponente auch Bestandteile, die der Präsentation und Strukturierung von Modellen dienen. Dies sind die graphischen Elemente bzw. Diagrammelemente. Wir rechnen diese der Komponente Anwendungsgebiet zu, weil sie zur sinnvollen Verwendung der mit dem graphischen Editor erzeugten Spezifikation - also etwa zur Druckausgabe - benötigt werden. Die Anwendungsgebietskomponente umfaßt alle Informationen, die dauerhaft über ein Modell gespeichert werden müssen.

Die grobe Struktur der PDC und ihre Beziehung zur HIC wird in Abbildung 4.4 veranschaulicht.

Das Diagramm verdeutlicht, daß unsere Architektur in der PDC und HIC drei Ebenen (Layer) enthält. Ein Modell kann mehrere Diagramme haben, die jeweils unterschiedliche Aspekte des Modells repräsentieren. Ein Diagramm ist dabei ein Dokument, das der Benutzer erstellt. Verschiedene Diagrammarten bzw. Dokumentarten haben unterschiedliche Bearbeitungsfunktionalität. Die über das Menü erreichbaren Kommandos sowie die Reaktion des Systems auf Ereignisse wie Mausklick sind i.A. bei verschiedenen Diagrammarten unterschiedlich. Das Diagramm ist in Seiten bzw. Blätter (Sheets) eingeteilt und wird dem Benutzer in einer Ansicht (View) präsentiert. Der View hat den Zugriff auf alle Seiten des Diagramms mittels des Diagramm-Objekts. Jeder View stellt jeweils eine Seite (ActiveSheet) des Diagramms dar und erlaubt so das Bearbeiten eines Ausschnitts des Diagramms. Zu jedem Diagramm kann es mehrere Views geben, so daß das gleichzeitige Bearbeiten mehrerer Seiten - oder auch derselben Seite - möglich ist.

In jeder Ansicht (View) können spezifische Einstellungen getroffen werden, die die graphische Darstellung beeinflussen und die Bearbeitung betreffen. Z.B. kann der Vergrößerungsfaktor gewählt und ein Bearbeitungsmodus eingestellt werden. Diese View-spezifischen Einstellungen können für jede Ansicht unabhängig gemacht werden.

Änderungen, die das Dokument (Diagramm) betreffen, wie etwa das Hinzufügen oder Löschen eines Diagramm-Elements, müssen dagegen jedem View, der diese Seite darstellt, mitgeteilt werden.

Ebenso ist möglich, daß ein logisches Element - zumindest Klassen - auf mehreren Seiten des Diagramms vorkommen. Werden Eigenschaften des logischen

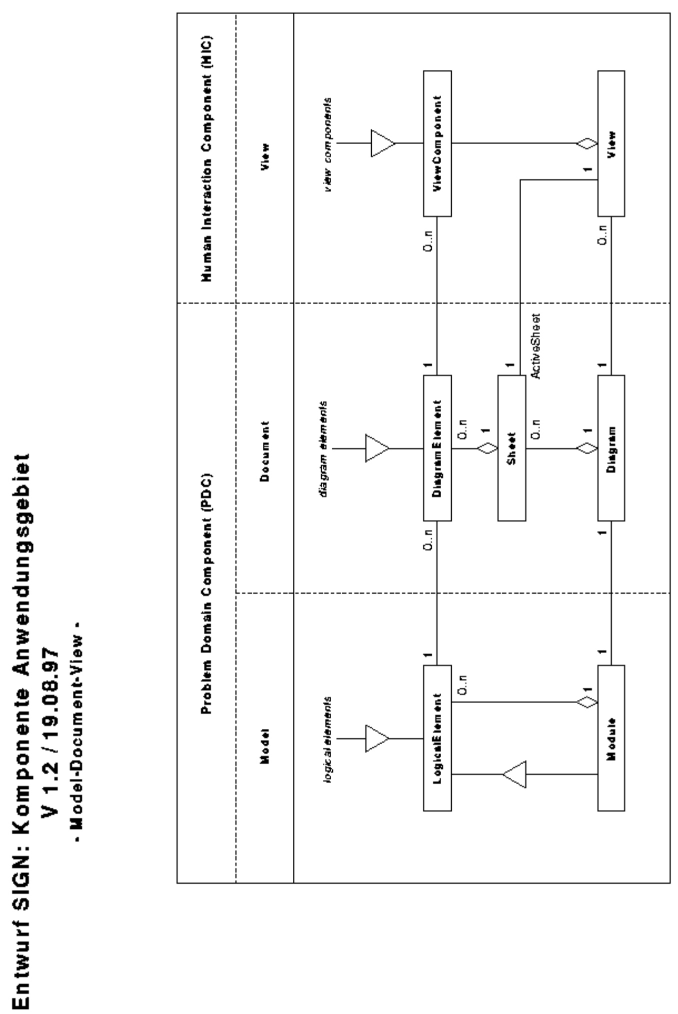


Abbildung 4.4:

Elements geändert - z.B. ein Attribut hinzugefügt - so muß dies allen Diagramm-Elementen, die es u.U. auf verschiedenen Seiten des Diagramms gibt, mitgeteilt werden. Gibt es mehrere Diagrammarten, so kann ein logisches Element auch in verschiedenen Diagrammen durch ein oder mehrere Diagrammelemente repräsentiert werden. Auch hier gilt dann die Benachrichtigungspflicht bei Änderungen.

Um die Aktualisierungen zu gewährleisten, sehen wir die Verwendung des Beobachtermusters [GHJV96] in unserem Entwurf vor. Ein beobachtbares Objekt stellt dabei die Observable-Schnittstelle zur verfügung. Der Beobachter implementiert eine Observer-Schnittstelle.

Hierarchie Logische Elemente I

Die Klassen der Hierarchie Logischer Elemente sind als Knoten eines abstrakten Syntaxbaums zu verstehen. Die Sprache, deren Syntax hier beschrieben wird (die Objektsprache), dient der Erstellung von Klassendiagrammen. Die Syntax der Objektsprache ist hier wiederum spezifiziert unter Verwendung einer Metasprache, die dieselben Sprachkonstrukte hat, wie die Objektsprache. Die Spezifikationssprache wird auf ihre eigene Spezifikation angewendet.

Die Aufgaben bzw. Verantwortlichkeiten, die den Klassen in diesem Diagramm zukommen, ergeben sich aus der Beschreibung der zugehörigen Sprachkonstrukte. Diese sind in [RBP+91], [BRJ97]sowie in der Ausarbeitung zu AP 1320 nachzulesen. Der Schwerpunkt der nachfolgenden textuellen Erläuterungen zu den Diagrammen wird daher in einer Interpretation des Diagramms (siehe Abbildung 4.5), nicht aber in einer Wiederholung der Beschreibung der Sprachkonstrukte liegen.

Klassen, Assoziationen, Aggregationen, Rollen, Attribute und Operationen können benannt werden. Die Angabe eines Namens ist jedoch in vielen Fällen (Assoziationen, Aggregationen, Rollen) optional. Generalisierungen, Qualifier und Candidate Keys haben keine Namen.

Üblicherweise wird für die Benennung von Klassen, Operationen und Attributen die Einhaltung von Eindeutigkeitsregeln wie in einer Programmiersprache mit Blockstruktur verlangt. D.h. innerhalb eines Moduls bzw. Diagramms sollen Klassennamen eindeutig sein. Innerhalb einer Klasse darf derselbe Name für eine Operation oder ein Attribut höchstens einmal vorkommen. Eine Ausnahme ist das Überladen von Operationen. Hierbei ist zulässig, daß zwei Operationen denselben Namen und denselben Typ als Rückgabewert haben, sofern sie sich in Typ und / oder Anzahl ihrer Parameter unterscheiden.

Class

Die Klasse „Class“ repräsentiert eine Klasse im Klassendiagramm. Eine Klasse ist ein Aggregat aus Attributen und Operationen. Sie steht über Generalisierungen und Assoziationen sowie Aggregationen in Beziehung zu anderen Klassen.

Generalization

Entwurf SIGN: Komponente Anwendungsgebiet
V 1.2 / 19.08.97
- Hierarchie Logischer Elemente I -

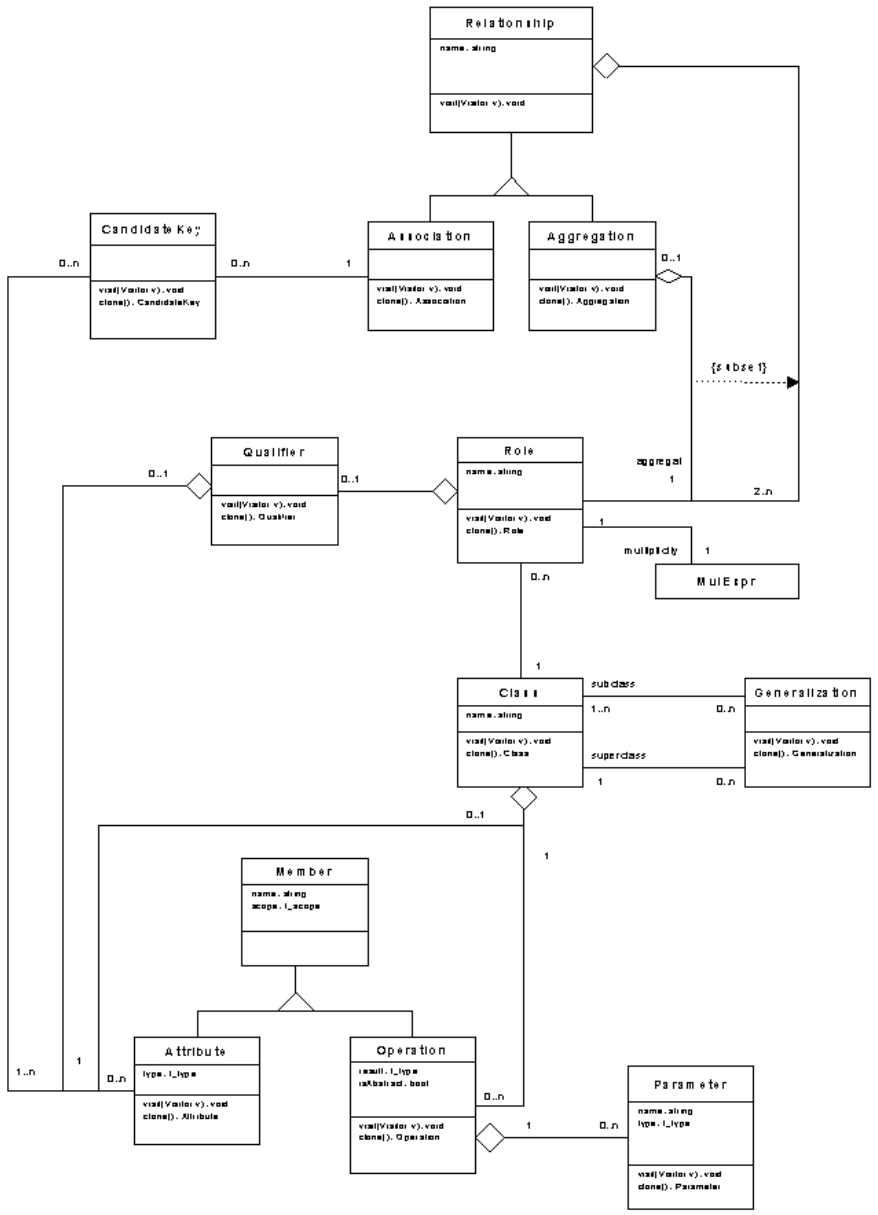


Abbildung 4.5:

Klassen können in Generalisierungsbeziehungen zu anderen Klassen stehen (Vererbung) und dabei die Rolle einer subclass (Unterklasse) oder superclass (Oberklasse) annehmen. Mehrfachvererbung wird durch mehrere Instanzen von Generalisierungen (Generalization) mit jeweils einer anderen Klasse in der Rolle der Oberklasse modelliert. Jede einzelne Generalisierungsbeziehung hat genau eine Oberklasse und mindestens eine Unterklasse.

Member

Sowohl Attribute wie auch Operationen sind Komponenten von Klassen. Sie haben insbesondere beide die Eigenschaft, daß ihr Gültigkeitsbereich auf eine Instanz der Klasse beschränkt sein kann oder für alle Instanzen der Klasse gilt. Daher enthält die Klasse Member als gemeinsame Oberklasse von Attributen und Operationen ein Attribut scope zur Angabe des Gültigkeitsbereichs (Instanz oder Klasse).

Attribute

Attribute werden durch Angabe eines Namens (siehe Klasse Member) und Typs spezifiziert. Sie sind i.d.R. Komponente einer Klasse bzw. Klassensignatur. Eine Ausnahme sind Qualifier-Attribute (siehe Qualifier).

Operation

Neben einem Namen und einem Typ für den Rückgabewert kann für eine Operation angegeben werden, ob die Klasse für sie eine Implementation (Methode) bereitstellt, oder nicht. Im letzteren Fall gilt sie als abstrakte Operation und muß von jeder konkreten Unterklasse redefiniert werden. Operationen haben optional eine Anzahl von Parametern zugeordnet.

Parameter

Die Parameter werden durch Name und Typ spezifiziert. Bisher ist nicht vorgesehen auch im Klassendiagramm bereits erkennbar zu machen, ob ein Parameter nach Ausführung der Operation verändert sein kann (Referenzparameter, Variablenparameter). Hierzu wird erst eine erweiterte Version des Entwurfs unter Einbeziehung von Datenflüssen bzw. Vor- und Nachbedingungen für Operationen Aussagen enthalten.

Relationship

Objekte von Klassen stehen untereinander mittels Beziehungstypen (Relationship) in Verbindung. Die Klasse Relationship dient der Beschreibung des Typs der Beziehung. Die Beschreibung besteht aus der Angabe eines Namens und der beteiligten Rollen. An jeder Relationship sind mindestens zwei Rollen beteiligt.

Association

Assoziationen sind eine spezielle Form von Relationship, an der auch mehr als nur zwei Rollen beteiligt sein können.

Aggregation

An jeder Aggregation sind genau zwei Rollen beteiligt: das Aggregat und die Komponentenrolle. In einer Aggregation ist eine der beteiligten Rollen als Aggregat ausgezeichnet. Dies wird im Modell durch die zusätzliche Assoziation zwischen der Klasse Aggregation und der Klasse Role dargestellt.

Role

Bei der Teilnahme an Beziehungen zu anderen Klassen (Relationship) nimmt jede Klasse eine Rolle ein. Die Rolle kann explizit durch Angabe eines Namens verdeutlicht werden. Durch Angabe einer Multiplizität wird festgelegt, wieviele Instanzen der Klasse an der Beziehung teilnehmen. Wir übernehmen die bei OMT übliche Notation, bei der die Rolle die Sicht der anderen Beziehungspartner auf die mit dieser Rolle verbundene Klasse angeben. Sind die Klassen A,B über die Assoziation R verbunden, und ist A dabei die Rolle a mit dem Namen „a_role “ und der Multiplizität 3 zugeordnet, so bedeutet das, daß jedem Objekt der Klasse B genau 3 Objekte der Klasse A zugeordnet werden, die aus der Sicht von B die Rolle eines „a_role “ haben. Man erkennt, daß dies bei ternären Assoziationen zu Schwierigkeiten führt. Ist an der obigen Assoziation auch die Klasse C beteiligt, würde die angegebene Multiplizität automatisch auch für die Beziehungen der Objekte dieser Klasse zu Objekten der Klasse A gelten. Die in ER-Modellen übliche Notation vermeidet das Problem - wird hier aber nicht verwendet.

MulExpr

Jeder Rolle wird eine Multiplizität zugeordnet, deren Bedeutung oben erläutert wurde. Die Multiplizität wird als regulärer Ausdruck in Textform angegeben. Dabei gilt die folgende Syntax (in EBNF):

```
mulexpr := multerm {"," multerm }
multerm := mulinterval | mulfactor
mulinterval := mulfactor ".." mulfactor
mulfactor := nat | "*"
nat := digit { digit }
digit := "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Im Allgemeinen ist die Multiplizität eine (möglicherweise unendliche) Untermenge der natürlichen Zahlen. Meistens kann sie jedoch mit einer Menge von Intervallen oder Zahlen angegeben werden. Um eine beliebige Anzahl auszudrücken, wird in unserer Syntax das Zeichen "*" verwendet. Als Kontextbedingung gilt, daß bei der Angabe eines Intervalls n..m die untere Grenze n kleiner als die obere Grenze m sein muß und daß sich die angegebenen Intervalle nicht überschneiden dürfen.

```
Beispiele:
0..*
0..1,3..*
```

```
0,1,2
0..2
1,2,6..9,12..*
*
```

Zur Erzeugung von Objekten der Klasse MulExpr wird die Klassenoperation createFromText(...) verwendet. Die Operation übernimmt einen Text, prüft die Syntax und erzeugt den nachfolgend abgebildeten Baum. Der Text wird Zwecks Ausgabe im Diagramm (siehe Abbildung 4.6) in dem Attribut source des neu erzeugten MulExpr-Objekts abgelegt.

MulTerm, MulInterval und MulFaktor

Ein Multiplizitätsausdruck (MulExpr) setzt sich zusammen aus einer Liste von Multiplizitätstermen. Jeder Term ist entweder ein Intervall, das mit oberer und unterer Grenze angegeben wird, oder er ist ein einfacher Faktor. Ein Faktor steht für einen Zahlenwert oder für das Unendlich-Symbol "*". Da gültige Werte für Multiplizitätsfaktoren stets nicht negativ sind, kann das Attribut value mit -1 belegt werden, um "*" zu symbolisieren.

Qualifier

Binären Assoziationen kann ein Qualifier zugeordnet werden, indem ein Attribut angegeben wird, das den Zugriff auf ein einzelnes Element oder auf eine Untermenge von Elementen der Assoziation ermöglicht.

Das Qualifier-Attribut ist als Bestandteil der Assoziation aufzufassen. Es ist daher keiner Klasse, sondern direkt dem Qualifier zugeordnet. Da ein Qualifier gemäß OMT-Notation - wie eine Rolle - an einem Ende der Assoziation dargestellt wird, ist der Qualifier der Rolle zugeordnet.

Beispiel: Klasse A und B sind über Association R verbunden. A hat in B die Rolle a, B hat in A die Rolle b. Das Attribut x wird als Qualifier-Attribut gewält und der Rolle a zugeordnet. Dann ist der Zugriff aus einem Objekt der Klasse A auf ein assoziiertes Objekt der Klasse B (oder auf eine Untermenge von assoziierten Objekten der Klasse B) über die Rolle b mit dem Qualifier x möglich. D.h. die Rolle b könnte durch ein assoziatives Array realisiert sein. Da x als Attribut der Assoziation gilt, ist der Zugriff unter Angabe des qualifizierenden Attributs x auch in umgekehrter Richtung möglich. D.h. die Rolle a kann auch als assoziatives Array realisiert sein. Im Diagramm wird der Qualifier jedoch nur an dem einen Ende der Assoziation R dargestellt. Im Beispiel bei der Rolle a auf der Seite von Klasse A.

Candidate Key

Candidate Keys dienen der Einschränkung der Multiplizität bei Assoziationen mit mehr als zwei beteiligten Klassen.

Entwurf SIGN: Komponente Anwendungsgebiet
V 1.2 / 19.08.97
- Multiplicity -

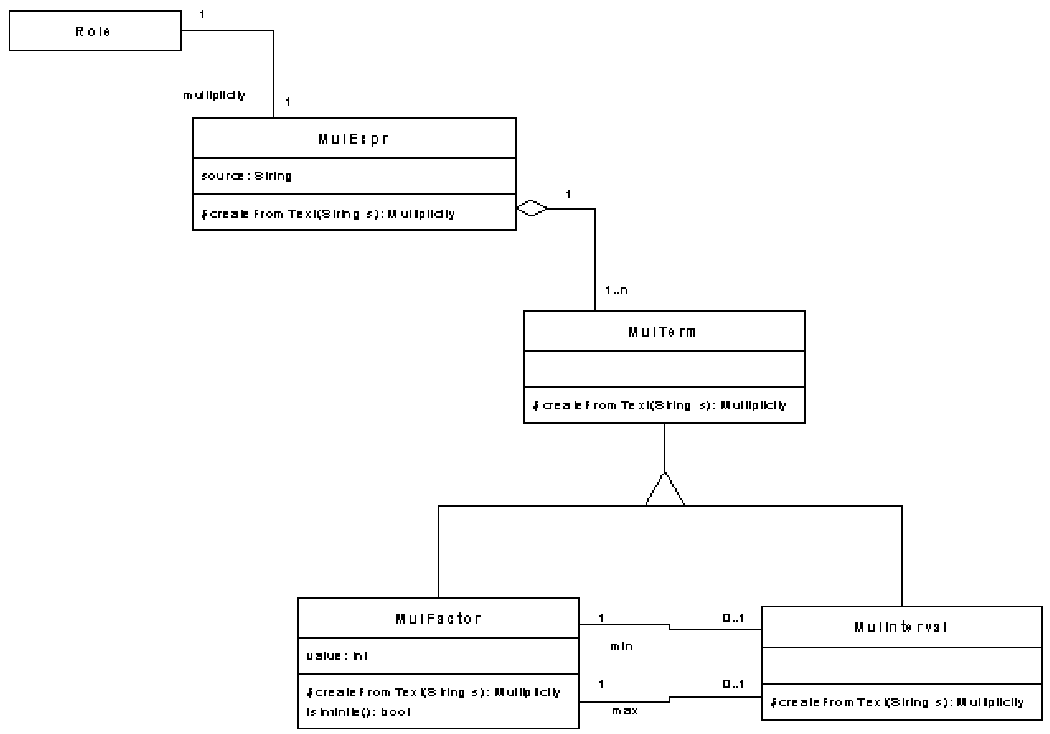


Abbildung 4.6:

elementare Datentypen

t_scope: Aufzählung instance_scope, class_scope

t_type: string zur Angabe des Datentyps z.B. in der Form integer — real — decimal — string — array[t_type] — name wobei name der Name einer Klasse, Accoziation oder Aggregation sein kann. Die Syntax des Texts wird zunächst nicht überprüft. Der Inhalt von Attributen dieses Typs wird zunächst unverändert bei der Übersetzung des Modells in eine Zielsprache verwendet. Später ist eine formelle Syntax für Typangaben zu spezifizieren und ein Parser einzusetzen. Als weitere Verbesserung wäre eine Dialogauswahl für die jeweils zulässigen Typangaben denkbar.

Hierarchie Logische Elemente II

LogicalElement

Die Klasse LogicalElement ist Oberklasse aller logischen Modellbestandteile. Jedes LogicalElement kann mit mehreren Diagrammelementen assoziiert sein. Die Diagrammelemente repräsentieren das LogicalElement auf den Seiten des Diagramms. Die Beziehung zwischen dem LogicalElement und seinen Diagrammelementen gleicht dem Beobachter-Muster [GHJV96]. Diese Beziehung wird bei der Beschreibung der funktionellen Sicht eingehend erläutert.

Module

Die logischen Elemente eines (Sub-) Systems werden in einem Modul zusammengefaßt. Prinzipiell können Module (rekursiv) auch weitere Submodule enthalten. Die Implementierung dieser Eigenschaft wird jedoch nicht verlangt, Eine einfache nicht-rekursive Implementation enthält genau ein Modul, das die Modellebene repräsentiert. Jedes Modell kann mit Diagrammen verschiedenen Typs assoziiert sein, die seine Bearbeitung durch den Benutzer ermöglichen. Dabei gilt, daß es zu jedem Modell nicht gleichzeitig mehrere Diagramme desselben Typs geben soll. Da es in der ersten Ausbaustufe nur Klassendiagramme gibt, wird jedes Modul während der Bearbeitung durch genau ein (Klassen-) Diagramm repräsentiert.

Für den Fall, daß auch Submodule erlaubt sein sollen, ist vorgesehen zwischen Module und Model zu differenzieren. Ein Model ist dann ein Module, das durch ein Diagramm dargestellt wird. Ein Module kann hingegen (als Submodul) durch ein Diagrammelement dargestellt sein.

Constraint

Jedes LogicalElement kann mit Zusatzangaben versehen werden. Diese können als informelle Anmerkung in Form von Text angegeben werden. Im erweiterten Sprachentwurf könnte auch eine Syntax zur Spezifikation von formellen Vor- und Nachbedingungen für Operationen oder allgemeine Invarianten bzw. Aussagen über zulässige Systemzustände enthalten sein. Die Klasse Constraint ist für

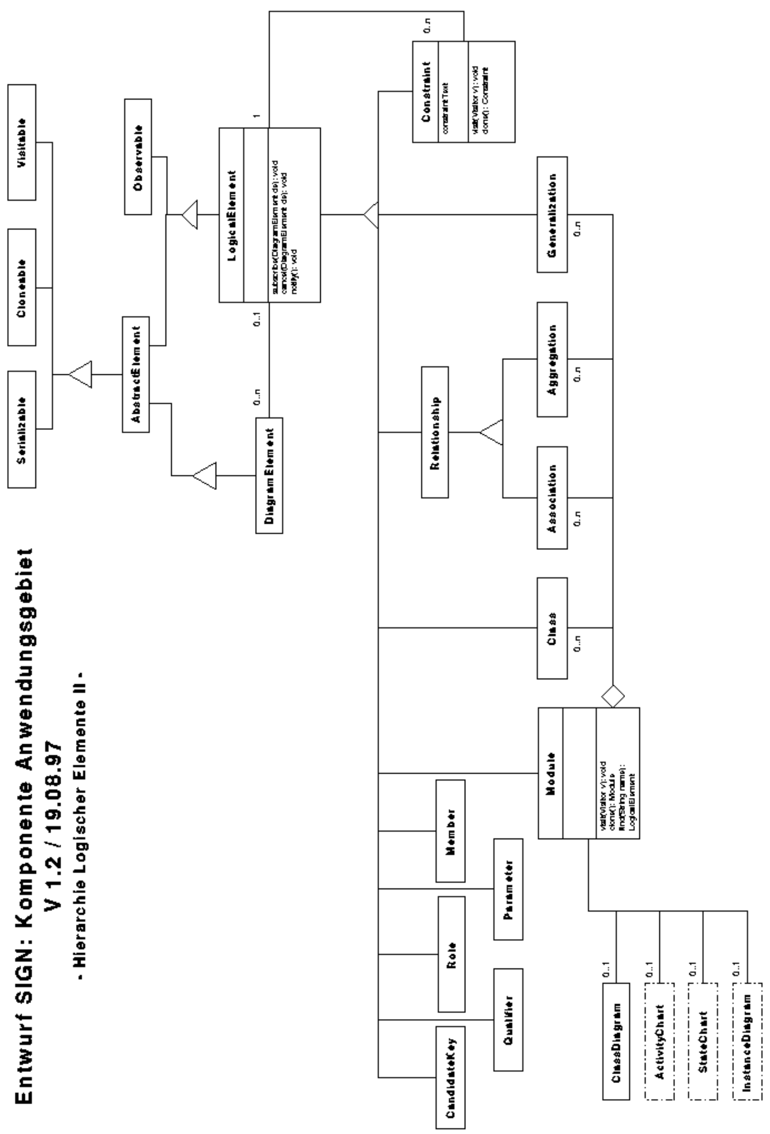


Abbildung 4.7:

solche Anmerkungen vorgesehen, die einer formellen Syntax folgen und evtl. zum Überprüfen des Modells (Verifikation) dienen können.

Informelle Anmerkungen können den Diagrammelementen zugeordnet werden (siehe Klasse Note, Hierarchie graphischer Elemente).

Hierarchie Graphische Elemente I

DiagramElement

Dies ist eine abstrakte Oberklasse aller Diagrammelemente. Neben den Eigenschaften der Interfaces Serializable, Cloneable und Visitable, wie sie auch für LogicalElement gelten, hat ein DiagramElement eine Observer-Schnittstelle. Die Beziehung zwischen DiagramElement und LogicalElement folgt dem Beobachter-Muster [GHJV96]. Jedem LogicalElement können mehrere Instanzen von DiagramElement zugeordnet sein.

Während die speziellen LogicalElement-Unterklassen jeweils nur die Klasse DiagramElement kennen müssen, sollen die Unterklassen von DiagramElement die Klasse des assoziierten logischen Elements kennen. Mit anderen Worten: Die Observable-Schnittstelle kann direkt in der Oberklasse LogicalElement implementiert sein.

Da die Diagrammelemente wiederum mit mehreren View-Komponenten verbunden sein können, muß in DiagramElement auch eine Observable-Schnittstelle implementiert werden.

Die Diagrammelemente beinhalten Daten und Code zur Darstellung von logischen Elementen.

Sheet

Diagramme können in Seiten unterteilt werden. Auf jeder Seite werden Diagrammelemente angeordnet. Dasselbe logische Element kann auch auf mehreren Seiten als ein Diagrammelement angeordnet werden.

Diagram

Die Diagramm-Klasse ist eine abstrakte Oberklasse für alle Diagrammarten, die es zu einem Modell gibt. Jedes Diagramm stellt ein Modul dar und erlaubt dessen Bearbeitung. Die Bearbeitungsfunktionalität hängt von der Diagrammart (Unterklasse von Diagram) ab. Damit auch die zur Verfügung stehenden Benutzerkommandos, Menüeinträge sowie die bei Mausbewegungen und Mausklicks ausgelösten Aktionen.

Da es verschiedene Diagrammarten geben kann, kann ein Modul auch gleichzeitig über verschiedene Diagramme bearbeitet werden. Hier findet wieder das Beobachter-Muster Anwendung. Trotzdem soll vermieden werden, ein Modul durch zwei Diagramme desselben Typs gleichzeitig zu repräsentieren.

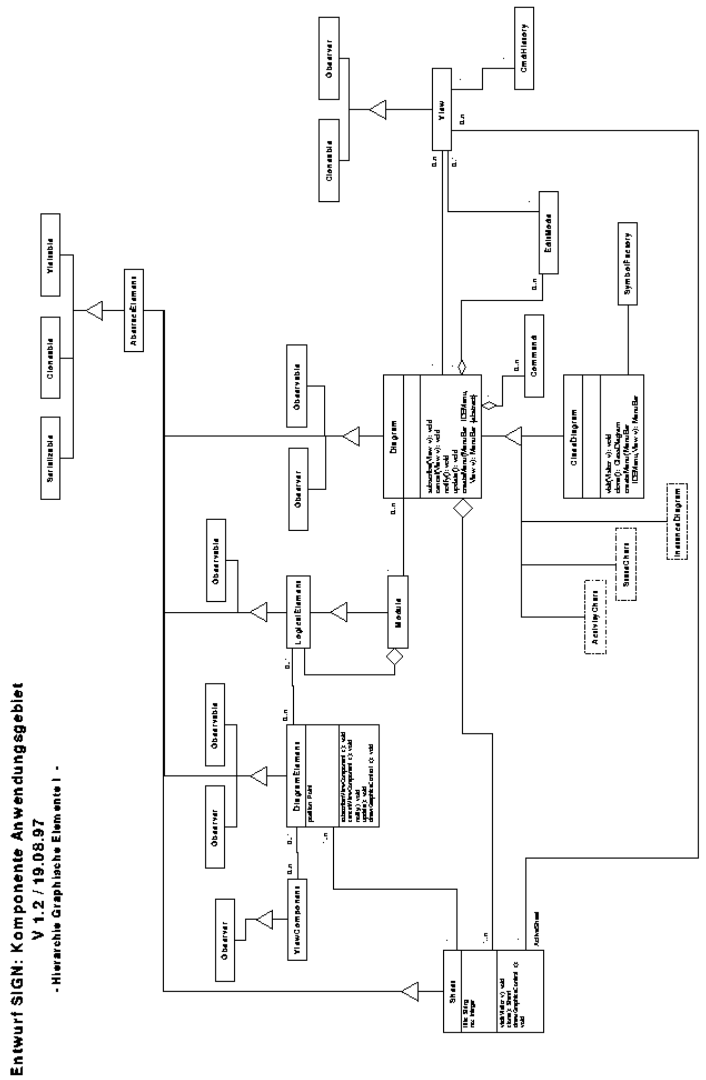


Abbildung 4.8:

ClassDiagram

Zunächst soll die graphische Entwicklungsumgebung nur den Diagrammtyp Klassendiagramm beinhalten. Alle im Folgenden gezeigten Unterklassen von DiagramElement, die auf den Sheets eines Diagramms angeordnet werden können, sind Klassendiagramm-Elemente.

Hierarchie Graphische Elemente II

DEClass

Diese Klasse entspricht der graphischen Repräsentation eines Klassensymbols. Objekte dieser Klasse sind mit einem logischen Element vom Typ Class assoziiert. Das Klassensymbol ist ein Rechteck und durch seine Position (location) sowie Höhe und Breite (height, width) gekennzeichnet. Die Höhe und Breite werden aus dem Platzbedarf für Attribute und Operationen berechnet. Es kann für jedes Objekt der Klasse DEClass individuell festgelegt werden, ob Attribute und/oder Operationen angezeigt werden sollen (show_attributes, show_operations). Höhe und Breite des Symbols werden entsprechend berechnet.

Am Rand des Klassensymbols befinden sich Ankerpunkte (DEBorderAnchor), die als Anknüpfungspunkt für Assoziationen etc. dienen. Zunächst werden die Ankerpunkte bei der Konstruktion des Objekts fest eingestellt. Später könnte man es dem Benutzer ermöglichen, diese zu verschieben, hinzuzufügen und zu löschen.

DEAbstractAnchor

Dies ist eine abstrakte Oberklasse für Ankerpunkte. An Ankerpunkten werden Verbinder (DEBinaryLink) oder Textboxen (DEAnchoredTextBox) befestigt. Allen Ankerpunkten ist ferner gemeinsam, daß ihre Position nicht feststehend, sondern von der Position und Größe anderer Diagrammelemente abhängig ist. Die Position kann aber berechnet werden. Der Algorithmus zur Berechnung der Position ist aber für jede konkrete Unterklasse (potentiell) verschieden.

DEBorderAnchor

Diese konkrete Unterklasse eines Ankerpunkts ist auf dem Rand eines Klassensymbols (DEClass) plaziert. Die Position ergibt sich aus der Seite des Rechtecks (border: t_border = top — bottom — left — right) sowie dem relativen Abstand vom Ursprung (relative_distance: t_distance = 0..1). Mit anderen Worten: die Position wird in Koordinaten relativ zum Ursprung des Klassensymbols angegeben. Dabei steht die eine Koordinate durch Wahl der Seite fest und die andere kann mittels des relativen Abstands ermittelt werden. (border=top => x=relative_distance*width, y=0, border=bottom => x=relative_distance*width, y=height, border=left => x=0, y=relative_distance*height, border=right => x=width, y=relative_distance*height)

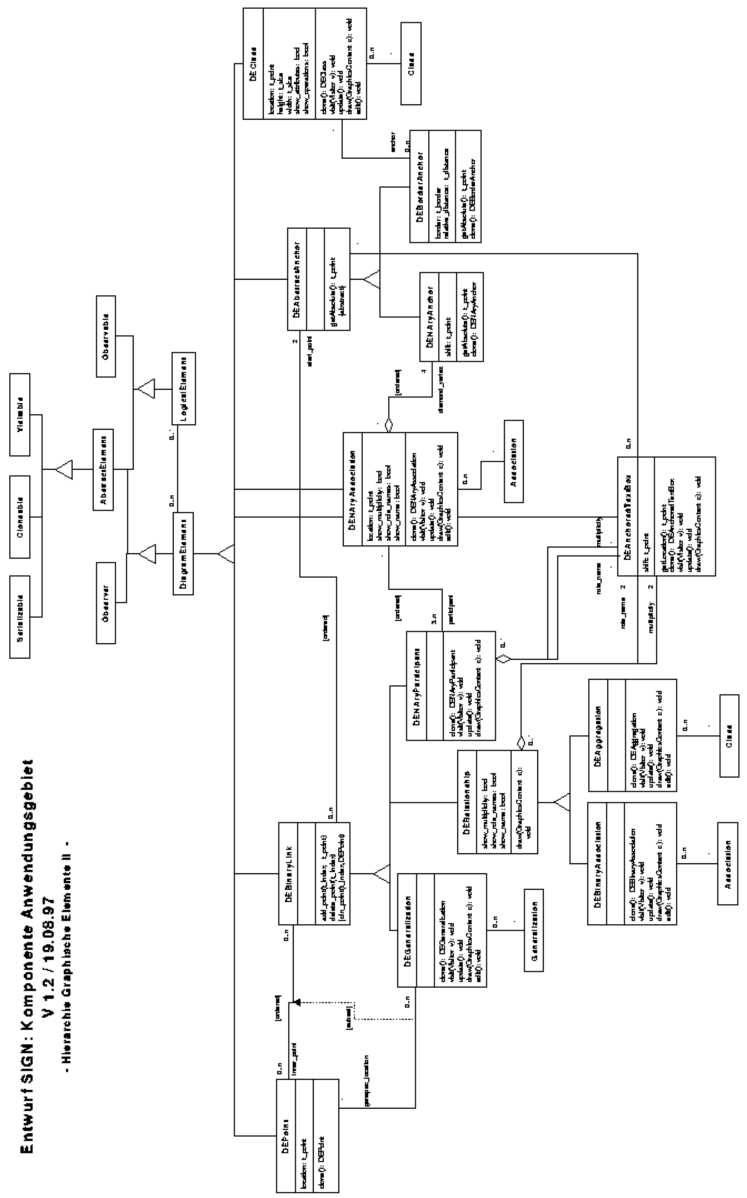


Abbildung 4.9:

DEPoint

Der Linienverlauf von Verbindern (DEBinaryLink) wird durch eine Anzahl von Zwischenpunkten bzw. Kontrollpunkten gesteuert. Der Benutzer kann beliebig Kontrollpunkte hinzufügen oder löschen und die Position von Kontrollpunkten ändern.

Derselbe Kontrollpunkt kann auch von mehreren Verbindern gleichzeitig genutzt werden. Damit soll erreicht werden, daß z.B. in Generalisierungsbeziehungen alle Verbinder der Oberklasse zu den Unterklassen scheinbar durch dasselbe Generalisierungssymbol laufen. Die Generalisierungssymbole werden dabei übereinander gelegt, indem sie denselben Punkt (DEPoint) als Kontrollpunkt referenzieren.

DEBinaryLink

Alle Verbinder sind von dieser Oberklasse abgeleitet. Die Klasse verwaltet die Kontrollpunkte (inner_point), die den Linienverlauf bestimmen. Ferner ist jeder Verbinder mit zwei Ankerpunkten (start_point) assoziiert. Der Linienverlauf ergibt sich aus Ankerpunkten und Kontrollpunkten. Seien die Ankerpunkte a0,a1 und die Kontrollpunkte k0,k1,k2 gegeben, so wird dadurch der Verlauf a0-k0-k1-k2-a1 bestimmt.

DERelationship

Die Verbinder werden weiter spezialisiert zu Generalisierungen und binären Relationships. Die binären Relationships umfassen binäre Assoziationen und Aggregationen. Die anzuzeigenden Attribute der korrenspondierenden logischen Elemente (Relationship, Role) sind der Name des Beziehungstyps, die Rollennamen und die Multiplizität. Die Klasse DERelationship enthält Attribute, um den Detaillierungsgrad der Anzeige einzustellen (show_multiplicity, show_role_names, show_name). Rollen und Multiplizitäten werden durch Instanzen der Klasse DEAnchoredTextBox repräsentiert.

DEBinaryAssociation

Die Klasse DEBinaryAssociation dient der graphischen Repräsentation eines logischen Elements Association mit zwei beteiligten Klassen (Rollen). Die in der Oberklasse DEBinaryLink referenzierten Ankerpunkte sind in diesem Fall vom Typ DEBorderAnchor und liegen am Rand der Klassensymbole der beteiligten Klassen.

DEAggregation

Analog zur Klasse DEBinaryAssociation repräsentiert ein Objekt der Klasse DEAggregation eine Instanz des logischen Elements Aggregation. Ein Unterschied in der graphischen Darstellung ist, daß hier am Ankerpunkt der Aggregat-Klasse bzw. am Rand Symbols der Aggregat-Klasse ein kleines Diamant-Zeichen als Aggregationssymbol gezeichnet wird.

DEAnchoredTextBox

Diese Klasse dient zur Ausgabe von Text für Rollennamen und Multiplizitäten. Ferner können informelle Anmerkungen (Notes) durch Objekte dieser Klasse dargestellt sein. Die Position des Textes im Diagramm ergibt sich aus der Position des assoziierten Ankerpunktes und einer relativen Verschiebung (shift) entlang der x- und y-Koordinatenachse. Der Benutzer soll die Möglichkeit haben, die Textposition relativ zu einem Ankerpunkt zu verschieben.

DEGeneralization

Das logische Element Generalization fast die Beziehungen einer Oberklasse zu ihren Unterklassen zusammen. Das Diagrammelement DEGeneralization beschreibt jeweils eine Beziehung Oberklasse-Unterklasse.

Auf der Modellebene (log.El.) kann eine Klasse auch in mehreren separaten Generalisierungen als Oberklasse erscheinen. Dies ist immer dann gewünscht, wenn Spezialisierungen oder Partitionierungen nach verschiedenen Aspekten vorgenommen werden sollen.

Der Benutzer soll die Möglichkeit haben, eine Generalisierungsbeziehung zwischen zwei Klassen durch Angabe von Ober- und Unterklasse festzulegen. Beschreiben zwei Generalisierungen (DEGeneralization) mit derselben Oberklasse eine Spezialisierung oder Partitionierung unter demselben Aspekt, so soll der Benutzer diese zusammenfassen können. Das Zusammenfassen geschieht durch Übereinanderlegen der Generalisierungssymbole. Die Generalisierungssymbole werden an dem ausgezeichneten Kontrollpunkt (DEPoint) mit der Rolle genspec_location plaziert. Zusammengefaßte Generalisierungen sollen dieselbe genspec_location referenzieren.

Plazierung des Generalisierungssymbols (siehe Abbildung 4.10)

DENAryAssociation

Assoziationen mit mehr als zwei beteiligten Klassen werden graphisch durch ein spezielles Symbol (großes Diamantsymbol) angezeigt. Jede der 3..n Teilnehmer an einer solchen Assoziation wird durch eine spezielle Form von Verbinder (DENAryParticipant) mit dem Assoziationssymbol verbunden. Das Assoziationssymbol kann vom Benutzer frei auf dem Diagramm positioniert werden. Ferner kann wie bei binären Assoziationen der Detaillierungsgrad der Anzeige eingestellt werden.

DENAryParticipant

Für mehrstellige (>2) Assoziationen gibt es diesen Typ von Verbinder. Die als start_point referenzierten Ankerpunkte der Oberklasse DEBinaryLink sind in diesem Fall ein Ankerpunkt am Klassensymbol (DEBorderAnchor) und ein Ankerpunkt am Assoziationssymbol (DENAryAnchor).

Entwurf SIGN: Komponente Anwendungsgebiet
V 1.2 / 19.08.97
- Platzierung des Generalisierungssymbols -

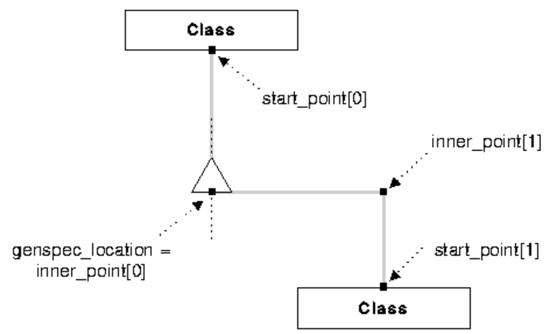


Abbildung 4.10:

DENAryAnchor

Jede NAryAssociation wird durch ein großes Diamantsymbol dargestellt. Die Ecken des Symbols (diamond_vertex) dienen als Ankerpunkte. Diese speziel-le Klasse von Ankerpunkten kann ihre absolute Position aus der Position des Assoziationssymbols und einer relativen Verschiebung (shift) um eine Strecke entlang der x- und y-Koordinatenachse berechnen.

Plazierung des Assoziationssymbols (siehe Abbildung 4.11)

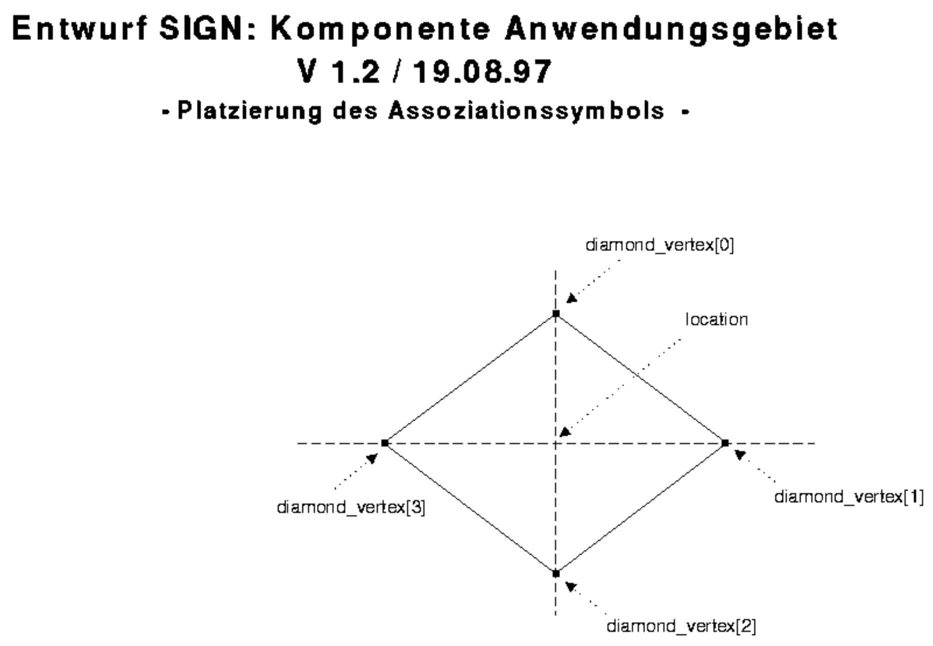


Abbildung 4.11:

Komponente Benutzungsschnittstelle

Der Entwurf der Benutzungsoberfläche behandelt die Themenbereiche Sichten, IDE, Fenster, Kommandos, Dialoge. Es wird die Architektur der Schnittstelle zwischen den Klassen des JDK und den Klassen der Komponente Anwendungs-gebiet beschrieben.

Diagramm-Ansicht (siehe Abbildung 4.12)

View

Der View ist als generische Schnittstelle zwischen dem Diagramm und dem Java-JDK konzipiert. Die verschiedenen Diagrammarten (Unterklassen von Diagram) arbeiten mit dieser Klasse zusammen.

Da die Bearbeitungsfunktionalität von der Diagrammart abhängt, übernimmt der View ein EditMode-Objekt vom Diagramm. Das EM-Objekt fungiert als Event-Handler. Der EditMode kann zur Laufzeit (per Benutzerkommando) umgeschaltet werden. Die EditMode-Klasse ist eine abstrakte Klasse. Sie legt lediglich ein Protokoll fest. Die eigentliche Bearbeitungsfunktionalität wird erst in den konkreten Unterklassen realisiert. Das Umschalten des EM erfolgt, in dem ein Objekt einer anderen Unterklasse von EditMode (z.B. EMSel, EMClass, EMGen, ...) an den View übergeben wird.

In dem View wird die aktive Seite des Diagramms (ActiveSheet) dem Benutzer zur Bearbeitung präsentiert. Die dargestellten Elemente sind Objekte vom Typ ViewComponent. Der Darstellungsbereich auf dem Bildschirm ist ein Teil des umgebenden Fensters (SIGNWindow) und vom Typ SIGNPanel.

ViewComponent

Die Klasse ViewComponent stellt die Verbindung zwischen den Klassen des Java AWT und den Diagrammelementen her. View-Komponenten sind sogenannte Lightweight Components, wie sie in der JDK 1.1-Dokumentation im Kapitel Java AWT: Lightweight UI Framework beschrieben sind.

Gleichzeitig implementiert die Klasse ViewComponent die MouseListener und MouseMotionListener-Interfaces gemäß dem Delegation Event Model des Java AWT 1.1. Die Maus-Ereignisse wie Anklicken von Elementen zur Selektion und ändern der Position durch Ziehen mit dem Mauszeiger können hier registriert und bearbeitet werden.

Möglicherweise empfiehlt es sich, View-Komponenten weiter in Unterklassen zu spezialisieren. Klassensymbole, die durch ein 2-dim. Rechteck dargestellt werden, können z.B. frei verschoben werden. Ebenso das Generalisierungssymbol, die Kontrollpunkte und das Symbol mehrstelliger Assoziationen. Dagegen können Verbinder, d.h. die Linienelemente in Assoziationen, Aggregationen und Generalisierungen nicht verschoben werden. Sie können aber markiert und z.B. gelöscht werden.

SIGNWindow

Jeder View soll in einem eigenen Fenster dargestellt werden. Die Klasse SIGNWindow hat die Aufgabe ein Rahmenfenster zur Verfügung zu stellen. Sie übernimmt ferner die Verwaltung des Menüs und die Reaktion auf die Ereignisse des Fenstersystems wie Aktivieren, Deaktivieren, etc. (siehe WindowAdapter).

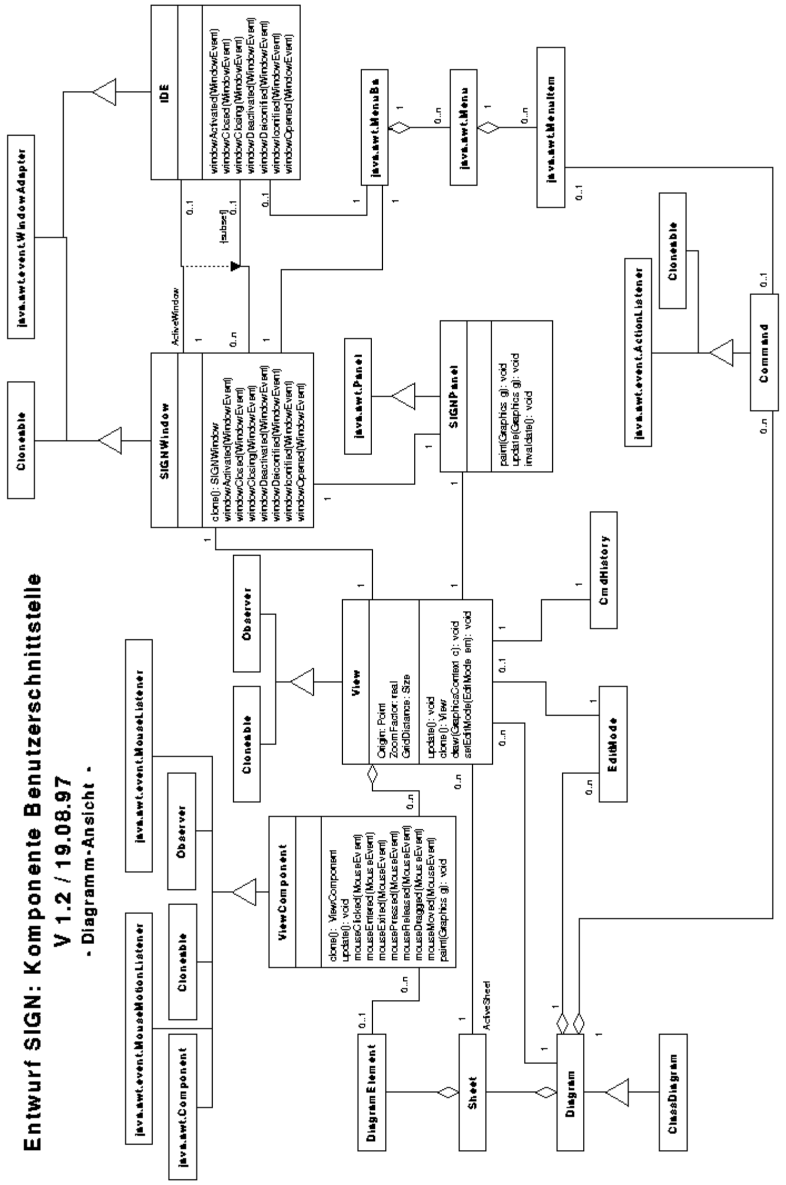


Abbildung 4.12:

SIGNPanel

Ein Teil des Fensters ist für die Darstellung des Views vorgesehen. Um das oben bereits genannte Lightweight UI Framework benutzen zu können, wird hierfür eine von der java.awt.Panel-Klasse angeleitete Klasse SIGNPanel verwendet. Sie erlaubt das Einfügen von Lightweight Components (java.awt.Component...).

Die Ereignisse für das Panel wie z.B. Mausklick, etc. werden von einem separaten Listener, einer Unterklasse von EditMode, bearbeitet. Der EditMode wird vom Diagramm zur Verfügung gestellt und vom View verwaltet. Der View erhält vom SIGNWindow den Zugriff auf den Darstellungsbereich SIGNPanel und ordnet diesem das Listener-Objekt EditMode zu.

IDE

Die integrierte Entwicklungsumgebung IDE stellt dem Benutzer eine zentrale Kommandoschnittstelle zur Verfügung. Hier sind in einem Rahmenfenster das Menü sowie eine Toolbar angeordnet. Das Menü enthält sowohl Einträge, die für alle Fenster und Diagrammarten gleich sind, als auch Einträge die für eine Diagrammart spezifisch sind und deren Ausführbarkeit vom jeweiligen Bearbeitungszustand des Views abhängig sind. Die allgemeinen Menüeinträge werden von der IDE erzeugt und mit einem Command-Objekt parametrisiert. Die spezifischen Menüeinträge werden vom Diagramm erzeugt und dem Fenster zugeordnet. Beim Aktivieren des Fensters sollen diese Einträge dann im Menü der IDE dargestellt werden. Der Bearbeitungszustand des Views wird dabei zunächst nicht berücksichtigt. Das Menü enthält stets eine Obermenge der im jeweiligen Zustand ausführbaren Kommandos. Das mit dem Menüeintrag verbundene Kommando-Objekt muß vor seiner eigentlichen Ausführung selbst prüfen, ob die Aktion zur Zeit möglich ist und dem Benutzer im Fehlerfall eine Rückmeldung geben. Später könnte ein Verfahren implementiert werden, bei dem die disable()/enable()-Operationen der java.awt.MenuItems benutzt werden, um das Menü kontextabhängig einzustellen.

Obwohl vorgesehen, ist die Toolbar bisher nicht im Entwurf modelliert. Sie enthält potentiell Auslöser (Buttons) für dieselben Kommandos wie das Menü. Ihre Konstruktion und Verwaltung kann daher in analoger Weise erfolgen.

java.awt.MenuBar

siehe Dokumentation zum JDK 1.1.

java.awt.Menu

siehe Dokumentation zum JDK 1.1.

java.awt.MenuItem

siehe Dokumentation zum JDK 1.1.

Entwurf SIGN: Komponente Benutzerschnittstelle
V 1.2 / 19.08.97
- Fenster, Listener und View-Komponenten -

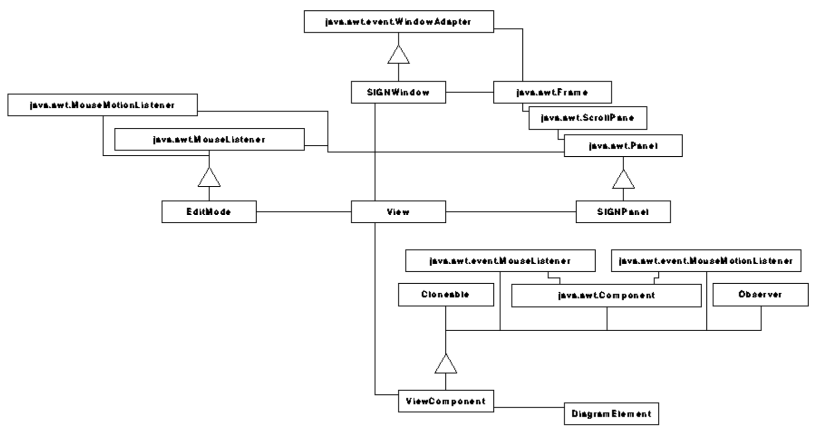


Abbildung 4.13:

Fenster, Listener und View-Komponenten

Diese Übersicht (Abbildung 4.13) zeigt den Zusammenhang der Fenster, Listener und View-Komponenten mit den Klassen des java.awt.

java.awt.Frame

siehe Dokumentation zum JDK 1.1.

java.awt.ScrollPane

siehe Dokumentation zum JDK 1.1.

java.awt.Panel

siehe Dokumentation zum JDK 1.1.

java.awt.Component

siehe Dokumentation zum JDK 1.1.

java.awt.event.MouseListener

siehe Dokumentation zum JDK 1.1.

java.awt.event.MouseMotionListener

siehe Dokumentation zum JDK 1.1.

java.awt.event.ActionListener

siehe Dokumentation zum JDK 1.1.

Bearbeitungsmodus und Selektion (siehe Abbildung 4.14)

EditMode

Jede Diagrammart stellt verschiedene Bearbeitungsmodi zur Verfügung. Ein Bearbeitungsmodus entspricht in etwa den Zeichenwerkzeugen (Selektion, Linie, Rechteck, Freihand, ...) wie sie aus diversen Mal- und Zeichenprogrammen bekannt sind. Abhängig vom eingestellten Modus erfolgen unterschiedliche Reaktionen auf Ereignisse wie Mausklick etc. und es sind unterschiedliche Kommandos aus dem Menü wählbar. Die wählbaren Editiermodi sind im Menü durch markierbare Menüeinträge (CheckboxMenuItem o.ä.) dargestellt. Sie sollten ferner durch Anklicken eines Buttons in der Toolbar einschaltbar sein.

Die Oberklasse der Editiermodi, EditMode, kann eine Reihe von Default-Aktionen für Ereignisse implementieren. Durch Anklicken eines Elements mit

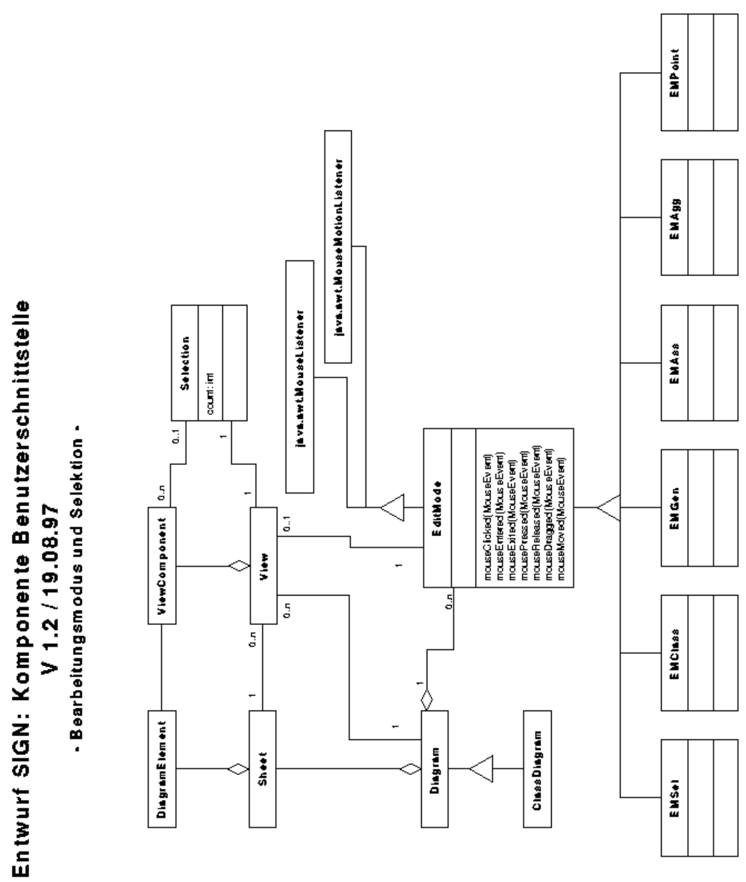


Abbildung 4.14:

der rechten Maustaste soll stets ein Kontextmenü geöffnet werden. Ein Doppelklick mit der linken Maustaste auf ein Element soll in der Regel den Eigenschaftsdialog zu dem Element aufrufen.

Das Kontextmeü ist im Entwurf bisher nicht modelliert. Es liegt jedoch nahe, hierfür ein PopupMenu aus dem Java AWT zu verwenden. Das PopupMenu-Objekt kann von einem DiagramElement erzeugt und mit einem ViewComponent-Objekt verbunden werden. Die Command-Objekte für die MenuItems werden ebenfalls von der speziellen Unterklasse von DiagramElement erzeugt und mit den notwendigen Parametern versorgt.

Jeder Bearbeitungsmodus sollte den Mauscursor so einstellen, daß der Benutzer erkennt, welche Reaktion auf die Mausereignisse zu erwarten sind.

EMSel

Werkzeug SSelektion": Elemente markieren mit Mausklick, Drag+Drop, Mehrfachselektion mit Rahmenziehen, Elemente zur Selektion hinzufügen mit Shift+Mausklick

EMClass

Werkzeug „Klasse“: Klasse hinzufügen mit linkem Mausklick

EMGen

Werkzeug „Generalisierung“: Ankerpunkt Oberklasse wählen, Zwischenpunkt(e) setzen, Ankerpunkt Unterklasse wählen, Generalisierung hinzufügen

EMAss

Werkzeug „Assoziation“: Ankerpunkt der ersten Klasse wählen, Zwischenpunkte(e) setzen, Ankerpunkt der zweiten Klasse wählen, Assoziation hinzufügen

EMAgg

Werkzeug „Aggregation“: Ankerpunkt der ersten Klasse wählen (diese ist die Aggregationsklasse), Zwischenpunkt(e) setzen, Ankerpunkt der zweiten Klasse wählen, Aggregation hinzufügen

EMPoint

Werkzeug ‘Kontrollpunkt“": Kontrollpunkte mit Mausklick selektieren, Verschieben durch Ziehen mit der Maus, Löschen durch Taste DEL, Hinzufügen mit Shift+Mausklick

Selection Viele Kommandos arbeiten auf den aktuell markierten Elementen eines Views. Jeder View kann daher eine Liste der markierten Elemente verwalten.

Kommandos

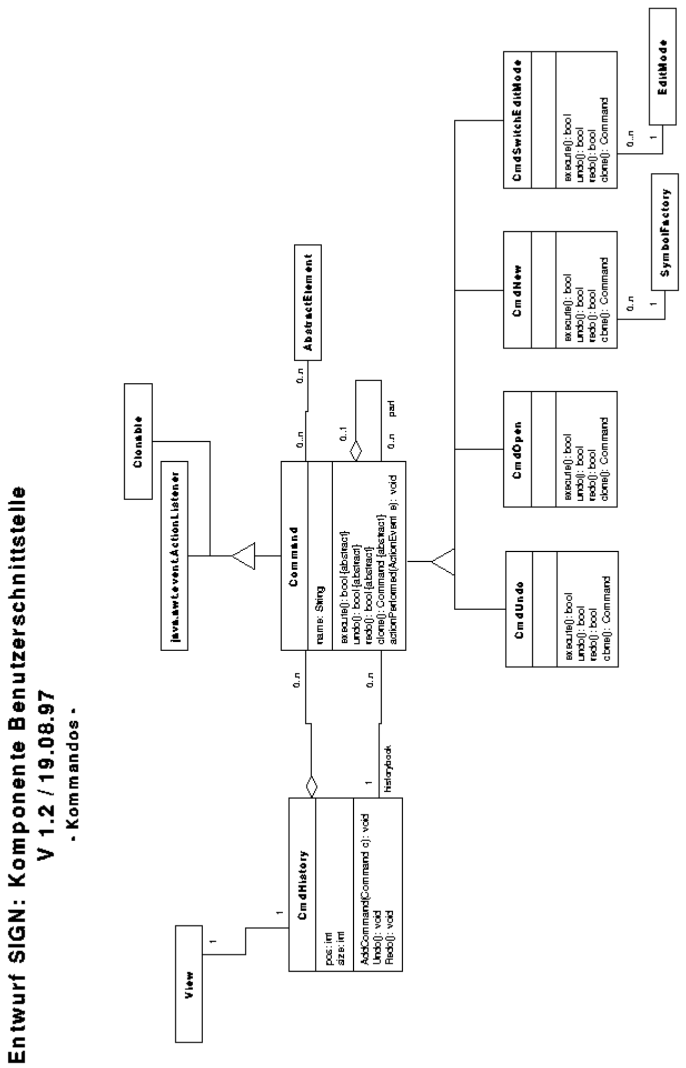


Abbildung 4.15:

Command

Die Klasse Command ist eine abstrakte Klasse zur Festlegung des Protokolls aller Kommandos (siehe Abbildung 4.15). Die Konkreten Unterklassen von Command müssen die execute(), undo(), redo() und clone()-Operation implementieren. Gleichzeitig implementiert die Command-Klasse eine ActionListener-Schnittstelle. Jedes Command-Objekt kann so als Event-Listener an ein java.awt.MenuItem-Objekt oder an ein java.awt.Button-Objekt gebunden werden.

Die Erzeugung von Benutzerkommandos als Reaktion auf Menüauswahl folgt dem Prototypen-Muster [GHJV96]. Die MenuItem-Objekte werden mit einem Objekt der Oberklasse Command parametrisiert, so daß dieses als Reaktion auf die Auswahl des MenuItems benachrichtigt wird. Die Benachrichtigung erfolgt durch Aufruf der Methode actionPerformed(ActionEvent e). Die Methode wird in der Oberklasse Command implementiert. Sie legt eine Kopie von sich selbst an durch Aufruf der clone()-Operation, die jede konkrete Unterklasse implementiert. Die Kopie wird ausgeführt durch Aufruf von execute(), während das Original weiterhin als Listener auf die Events des MenuItems reagieren kann, d.h. es bleibt als Prototyp erhalten.

CmdHistory

Ein Objekt der Klasse CmdHistory dient dazu, über die vom Benutzer ausgeführten Kommandos Buch zu führen. Jedes Kommando wird bei seiner Konstruktion mit einem CmdHistory-Objekt parametrisiert. Das Objekt nimmt für das Kommando die Rolle eines Geschichtsbuchs (historybook) ein. Wird das Kommando ausgeführt, so kann es sich mittels der Schnittstelle der CmdHistory-Klasse in das Geschichtsbuch eintragen.

CmdUndo

Das Kommando CmdUndo implementiert eine Undo-Funktion. Es verwendet das historybook-Objekt, um hier Undo() aufzurufen. Die Undo()-Operation der Klasse CmdHistory macht das zuletzt ausgeführte Kommando - also das zuletzt in das Geschichtsbuch eingefügte - wieder rückgängig. Kommandos können andere Teilkommandos beinhalten. In diesem Fall werden alle Teilkommandos in umgekehrter Reihenfolge ihrer vorherigen Ausführung rückgängig gemacht. Die korrekte Ausführung der zum Rückgängig machen erforderlichen Schritte ist für jedes spezielle Kommando in der Undo()-Methode zu implementieren.

Nach einem Undo() verbleiben die rückgängig gemachten Kommando-Objekte zunächst in der Historie. Solange nicht ein neues Kommando in die Historie eingefügt wird, ist es mittels der Redo()-Operation möglich, sie erneut auszuführen. Wenn ein neues Kommando eingefügt wird, werden die rückgängig gemachten Kommandos aus der Historie entfernt.

CmdNew

Ein neues Dokument wird eröffnet und initialisiert. Ein leeres Sheet wird erstellt, sowie ein View darauf.

CmdCut

Alle markierten Elemente werden ausgeschnitten und in das Clipboard kopiert. Gleiches gilt wie bei CmdDelete. Für ein Undo wird eine Kopie des Clipboards erstellt, und zwar bevor neue Objekte in das Clipboard kopiert werden. So läßt sich das alte Clipboard wiederherstellen, welches beim CCmdCut überschrieben wird.

CmdInsert

Alle logischen Elemente aus dem Clipboard werden in das aktuelle Sheet eingefügt. Die eingefügten Element aus dem Clipboard werden zwischengespeichert, um bei einem Undo diese wieder löschen zu können. Das Clipboard wird bei Redo/Undo nicht betroffen.

CmdDelete

Die markierten Elemente werden gelöscht (keine Kopie im Clipboard). Dazu wird über das Diagramm herausgefunden, welche logischen Elemente markiert sind, und diese werden entfernt (logische Elemente, Diagrammelemente sowie Graphische Elemente). Die gelöschten Attribute werden zwischengespeichert, um diese bei einem Undo wieder herstellen bzw. bei einem Redo wieder löschen zu können.

CmdEdit

Die markierten Elemente können editiert werden. Dazu wird entsprechend des Typs des Elements der jeweilige Dialog aufgerufen. Alle editierten Elemente werden jeweils vor und nach einer Änderung zwischengespeichert, um sie bei einem Undo bzw. Redo wiederherstellen zu können.

CmdArrange

Dies ist eine automatische Funktion zur Anordnung aller Objekte auf einem Sheet. Dabei werden die Relationen möglichst geradlinig gehalten und alle Klassen unter- und nebeneinander angeordnet. Alle Elemente des aktiven Sheets werden zwischengespeichert, um sie bei einem Undo wiederherstellen zu können. Bei einem Redo wird die Funktion einfach neu ausgeführt (Redo() entspricht Execute()).

CmdCopy

Alle markierten Elemente werden in das Clipboard kopiert.

CmdSearch

Mittels eines kleinen Dialoges werden die Attribute für eine Suche nach logischen Elementen ermittelt. Dieser Dialog bestimmt nach welchen Anforderungen gesucht werden soll (Name, Attribute, Methoden, alle/aktives Sheet etc).

CmdSave

Alle Objekte sollen abgespeichert werden. Ist noch keine Speicherung vorgenommen worden, wird das Kommando SaveAs aufgerufen; Ansonsten wird unter dem aktuellen Dokumentnamen das Dokument gespeichert.

CmdSaveAs

Wie in Save wird auch hier das Dokument gespeichert, aber vorher auf jeden Fall ein neuer Name abgefragt.

CmdPrint

Es können das aktuelle Sheet oder das gesamte Dokument gedruckt werden. Dazu wird auf den Standard-PrintDialog zurückgegriffen, der in Java definiert ist. Auch die Funktionialität des Druckens ist bereits in Java vordefiniert.

CmdExit

Das Programm wird nach einer Sicherheitsabfrage verlassen.

CmdClose

Sofern das Dokument nach der letzten Speicherung verändert wurde, erlaubt eine Sicherheitsabfrage (Speichern ja/nein) mittels Aufruf von CmdSave bzw CmdSaveAs eine Sicherung.

CmdGlobalOptions

Der "General Preferences Dialog" wird aufgerufen. Diverse Programmparameter wie z.B. Pfade, Umgebungseinstellungen etc. können eingestellt werden.

CmdCompileOptions

Der „Compiler Dialog“ zur Einstellung von Compiler Optionen wird aufgerufen. Es besteht die Möglichkeit Veränderungen vorzunehmen, die durch die Buttons Ok und Cancel bestätigt oder verworfen werden.

CmdCompile

Anhand der Optionen wird das gesamte Dokument in C++, Java oder SQL übersetzt. Dazu wird vorab ein Syntax-Check durchgeführt, der alle Objekte und ihre Beziehungen auf Korrektheit abfragt (siehe auch Kommando Syntaxcheck).

CmdSyntax

Ein Aufruf dieses Kommandos überprüft jede Klasse sowie ihre Beziehungen und erstellt gegebenenfalls einen MessageDialog, der auf Fehler hinweist.

Dialoge

Jeder SIGN-Dialog enthält einen OK-Button, der als Bestätigung für die Richtigkeit der Angaben dient. Wird der OK-Button angeklickt, so werden die entsprechenden Methoden aufgerufen, die die neu gewonnenen bzw. veränderten Informationen verarbeiten. Anschließend wird java.awt.Component.setVisible(false) aufgerufen.

Alle Dialoge außer dem Message-Dialog enthalten außerdem einen Cancel-Button, der nur dazu dient, java.awt.Component.setVisible(false) aufzurufen, also den Dialog zu schließen, ohne die Änderungen zu übernehmen.

Die folgende Grafik (Abbildung 4.16)soll verdeutlichen, wie diese einzelnen Dialoge in Verbindung zu einander stehen:

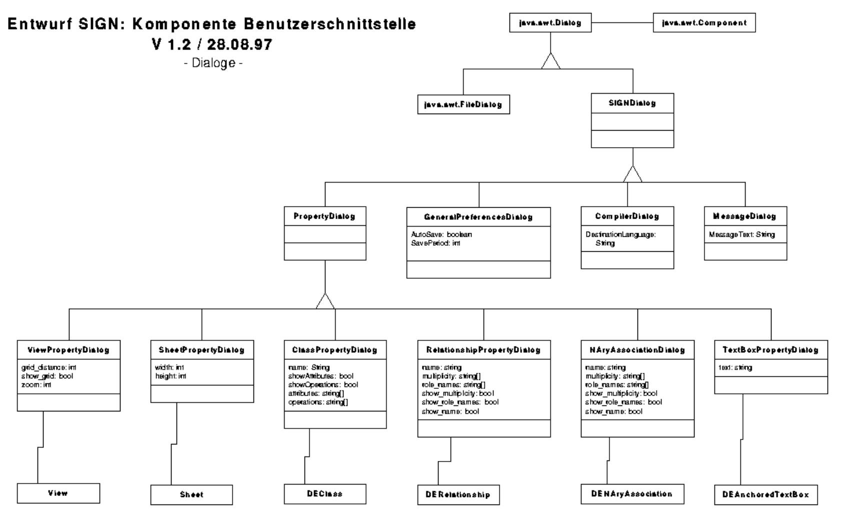


Abbildung 4.16:

SIGNDialog

Diese Klasse ist für das Erzeugen eines SIGNDialogs zuständig und enthält folgende von außen zugängliche Funktionen: Der Constructor der Klasse CSIGNDialog sorgt dafür, daß jeder Dialog den passenden Titel bekommt.

MessageDialog

Diese Klasse erzeugt einen Message-Dialog, also ein Fenster, daß einen Text beinhaltet und einen OK-Button um den Dialog zu schließen. Der Text wird dabei dem Constructor als Parameter übergeben.

GeneralPreferencesDialog

Diese Klasse erzeugt einen Dialog, der die allgemeinen Parameter des Editors anzeigt und deren Änderung ermöglicht. Als allgemeine Parameter wird in diesem Entwurf folgendes angesehen:

- Soll eine automatische Speicherung erfolgen und wie oft soll diese stattfinden? Die Antwort kann in Form von Radiobuttons ausgewählt werden.

CompilerDialog

Diese Klasse erzeugt einen Dialog, der die Einstellungen für das compilieren der geöffneten Datei anzeigt und deren Änderung ermöglicht. Die Zielsprache kann aus einer aufklappbaren Optionsliste gewählt werden.

PropertyDialog

Diese Klasse erzeugt einen Dialog, der die Eigenschaften eines Diagramm-Elements und des zugeordneten logischen Elements anzeigt und deren Änderung ermöglicht. Zum Anzeigen des aktuellen Status eines Elements werden die entsprechenden get*-Methoden der Klasse aufgerufen. Zum Setzen der Properties wird durch die set*-Methode des Property-Dialog's die entsprechende set*-Methode der Element-Klasse aufgerufen.

Hierbei steht set* für alle Methoden, die mit set beginnen. Entsprechendes gilt für get*.

ViewPropertyDialog

Diese Klasse erzeugt einen Dialog, der die Eigenschaften der Views anzeigt und deren Änderung ermöglicht. Die Eigenschaften eines Views sind das Gitter (grid) und der Zoom, mit dem die Graphik dargestellt werden soll. Dies läßt sich alles in Textfields eintragen, wobei der Zoom prozentual angegeben wird.

SheetPropertyDialog

Diese Klasse erzeugt einen Dialog, der die Eigenschaften der Sheets anzeigt und deren Änderung ermöglicht. Für ein Sheet werden bestimmte Standard-Größen bereitgestellt, zwischen denen man in einer aufklappbaren Optionsliste wählen kann.

ClassPropertyDialog

Diese Klasse erzeugt einen Dialog, der die Eigenschaften der Klassen anzeigt und deren Änderung ermöglicht. Hierbei kann der Name in einem Textfeld eingegeben werden. Die Operationen und Attribute der Klasse werden in einer Liste / Tabelle dargestellt und können hier editiert werden. Radiobuttons sind für die Option zuständig, ob Methoden und Attribute dargestellt werden sollen oder nicht.

RelationshipPropertyDialog

Diese Klasse erzeugt einen Dialog, der die Eigenschaften der Assoziationen anzeigt und deren Änderung ermöglicht. Die Eigenschaften einer Assoziation sind die Kardinalitäten und die zugehörigen Rollen. Sowohl die Kardinalitäten als auch die Rollennamen lassen sich in die dafür vorgesehenen Textfields eintragen. Dieser Dialogtyp wird sowohl für binäre Assoziationen wie auch für Aggregationen verwendet.

NAryAssociationPropertyDialog

Dieser Dialogtyp erlaubt das Bearbeiten der Eigenschaften mehrstelliger Assoziationen. Für jede beteiligte Klasse lassen sich Rollenname und Multiplizität eingeben.

NAryAssociationPropertyDialog

Diese Dialogklasse ermöglicht das Bearbeiten des Texts von Constraints / Notes, die auf der Diagrammebene durch ein Objekt der Klasse DEAnchoredTextBox repräsentiert werden.

Komponente Datenverwaltung

Zunächst soll lediglich das vom JDK unterstützte Serialisierungsverfahren zur Implementation der Verwaltung persistenter Daten eingesetzt werden. Hierbei ist das Verfahren zum Umgang mit der Objektidentität bereits vorgegeben. Im Hinblick auf eine spätere Erweiterung der graphischen Entwicklungsumgebung zum Import und Export eines Modells in SGML-Textdateien und / oder Verwendung von relationalen Datenbanken soll dieser Entwurf bereits ein eigenes Konzept zum Umgang mit Objektidentitäten vorsehen.

Ebenfalls im Hinblick auf spätere Erweiterungen haben wir für die Ausgabe von Code bzw. die Interpretation des Modells, z.B. zur Ausgabe von Postscript oder Java-Source-Code, die Verwendung des Besuchermusters vorgesehen (siehe hierfür auch Abbildung 4.17).

Allgemeine Services

Interfaces (siehe Abbildung 4.18und Abbildung 4.19)

Datenstrukturen

Vector

wird für Assoziationen und Aggregationen verwendet.

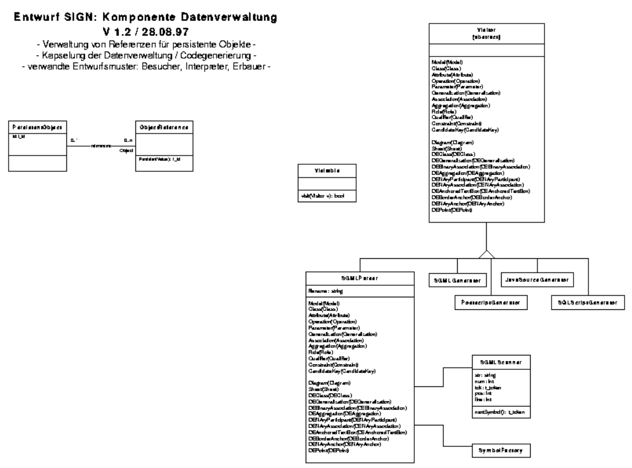


Abbildung 4.17:

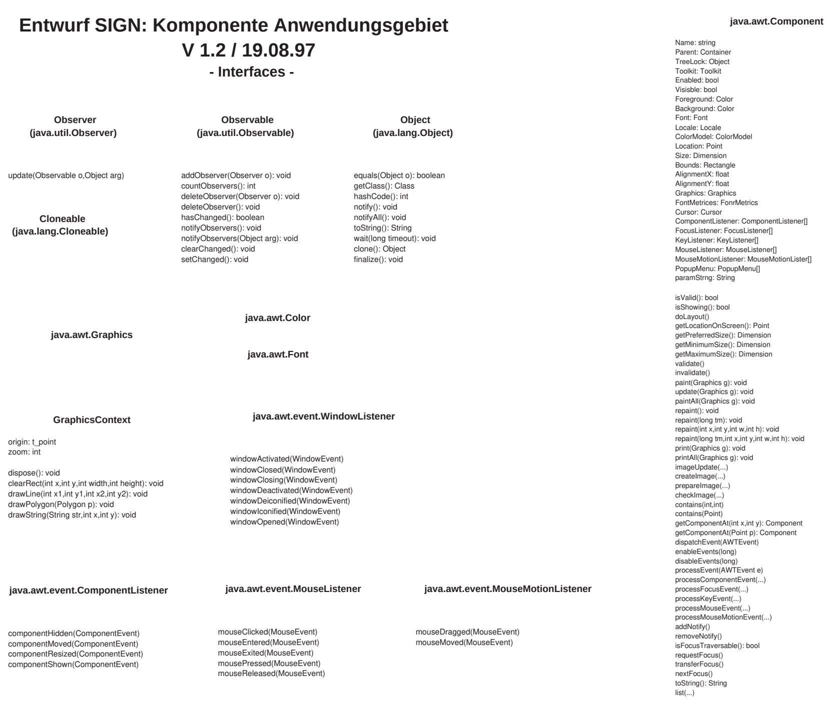


Abbildung 4.18:

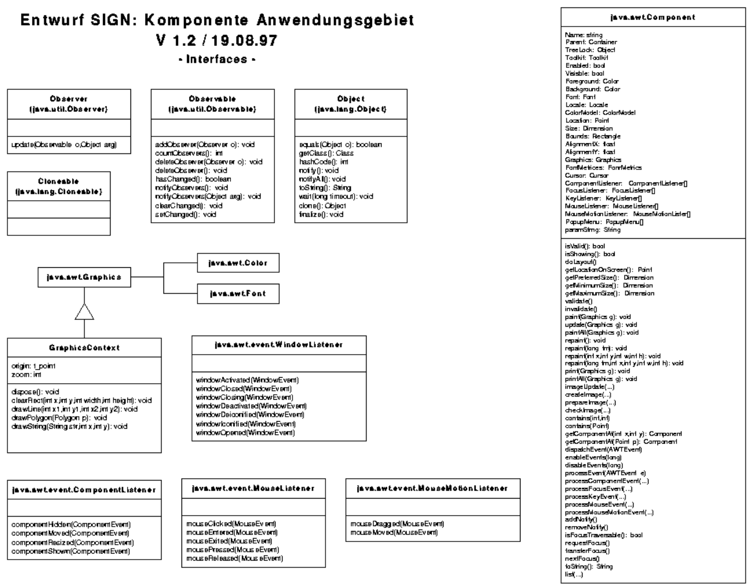


Abbildung 4.19:

Enumeration

wird für Assoziationen und Aggregationen verwendet, Rückgabewert der Operation getRoleName() einer Klasse mit Assoziation oder Aggregation ist vom Typ Enumeration (java.util.Enumeration).

4.1.2.4 Funktionale Sicht

Use Cases

Die nachfolgende Liste enthält Benutzungsfälle bzw. Use Cases (UML) für das Editieren von Klassendiagrammen. Auf eine Diagrammdarstellung wird hier verzichtet, da sie außer der Offenlegung einiger uses und extends Beziehungen wenig Erkenntnisse bringen würde. Die Listenform scheint besser geeignet, sich einen Überblick über die zu realisierenden Benutzungsfälle verschaffen zu können. Da den Projektteilnehmern der Anwendungskontext hinreichend bekannt ist, kann auf eine detaillierte Beschreibung der Benutzungsfälle verzichtet werden.

Use Case
Neues Klassen-Diagramm erstellen
Klassen-Diagramm laden
Klassen-Diagramm speichern
Klassen-Diagramm speichern unter neuem Namen
Klassen-Diagramm Eigenschaften/Layout/Anzeige-Optionen bearbeiten
Funktionelle Sicht bearbeiten
Dynamische Sicht bearbeiten
Klassen-Diagramm ausgeben (SGML, Postscript, SQL, Java, ...)
Klassen-Diagramm importieren (SGML, ...)
Klassen-Diagramm schliessen
Fenster duplizieren
Fenster schließen
Fenster aktivieren
Fenster Zoomfaktor einstellen
Fenster alle schliessen
Fenster anordnen (nebeneinander, überienander, ...)
Zeichenblatt hinzufügen
Zeichenblatt löschen
Zeichenblatt Eigenschaften/Layout/Anzeige-Optionen bearbeiten
Zeichenblatt auswählen
Zeichenblatt auswählen und in neuem Fenster bearbeiten
Zeichenblatt ausgeben (Postscript, ...)
Zeichenblätter Reihenfolge ändern
Modul hinzufügen
Modul löschen
Modul öffnen und bearbeiten
Klasse neu hinzufügen
Klasse auswählen und hinzufügen
Klasse löschen
Klasse verschieben
Klasse kopieren
Klasse ausschneiden
Klasse einfügen
Klasse Eigenschaften/Layout/Anzeige-Optionen bearbeiten
Klasse State-Chart bearbeiten
Klasse Ankerpunkt hinzufügen
Klasse Ankerpunkt Position verschieben
Klasse Ankerpunkt löschen
Klasse per Drag+Drop verschieben
Klasse per Drag+Drop kopieren
Attribut hinzufügen
Attribut Eigenschaften bearbeiten
Attribut löschen

Operation hinzufügen
Operation Eigenschaften bearbeiten
Operation löschen
Operation State-Chart bearbeiten
Aggregation hinzufügen
Aggregation löschen
Aggregation Eigenschaften/Layout/Anzeige-Optionen bearbeiten
Aggregation Kontrollpunkt hinzufügen
Aggregation Kontrollpunkt löschen
Aggregation in Assoziation umwandeln
Aggregation in Generalisierung umwandeln
Assoziation hinzufügen
Assoziation löschen
Assoziation Eigenschaften bearbeiten
Assoziation Kontrollpunkt hinzufügen
Assoziation Kontrollpunkt löschen
Assoziation in Aggregation umwandeln
Assoziation in Generalisierung umwandeln
Generalisierung hinzufügen
Generalisierung löschen
Generalisierung Kontrollpunkt hinzufügen
Generalisierung Kontrollpunkt löschen
Generalisierung Oberklasse wechseln
Generalisierung Unterklasse hinzufügen
Generalisierung Unterklasse entfernen
Generalisierung Unterklasse wechseln
Generalisierung in Assoziation umwandeln
Generalisierung in Aggregation umwandeln
Generalisierungen zusammenfassen
Generalisierungen trennen
Kontrollpunkt verschieben
Textposition verschieben (Rollenname,Multiplizität)
Rolle hinzufügen
Rolle löschen
Rolle Eigenschaften/Layout/Anzeige-Optionen bearbeiten
Rolle zu anderem Ankerpunkt verschieben
Rolle zu anderer Klasse verschieben
Element markieren / auswählen
Element zur Markierung hinzufügen
Element aus Markierung entfernen
markierte Elemente Eigenschaften/Layout/Anzeige-Optionen bearbeiten
markierte Elemente verschieben

markierte Elemente kopieren
markierte Elemente ausschneiden
markierte Elemente löschen
markierte Elemente per Drag+rop verschieben
markierte Elemente per Drag+rop kopieren
Element Markierung aufheben
Constraint zu Element hinzufügen
Constraint löschen
Constraint Position verschieben
Constraint zu anderem Element verschieben
Kommentar zu Element hinzufügen
Kommentar löschen
Kommentar Position verschieben
Kommentar zu anderem Element verschieben
Elemente suchen (und optional markieren)
Referenzen auf Elemente suchen
Kontextmenü anzeigen
Toolbar anzeigen
Befehl rückgängig machen
Befehl wiederherstellen

Kommando-Auslösung

Kommando Undo

Neues Klassen-Diagramm erstellen

Klassendiagramm laden

View-Komponenten erzeugen

Eigenschaften/Layout/Anzeige-Optionen bearbeiten

4.1.2.5 Implementationshinweise

Attribute

Zu jedem Attribut gibt es implizit Zugriffsfunktionen zum Lesen und Schreiben des Werts - diese werden im Modell nicht explizit aufgeführt.

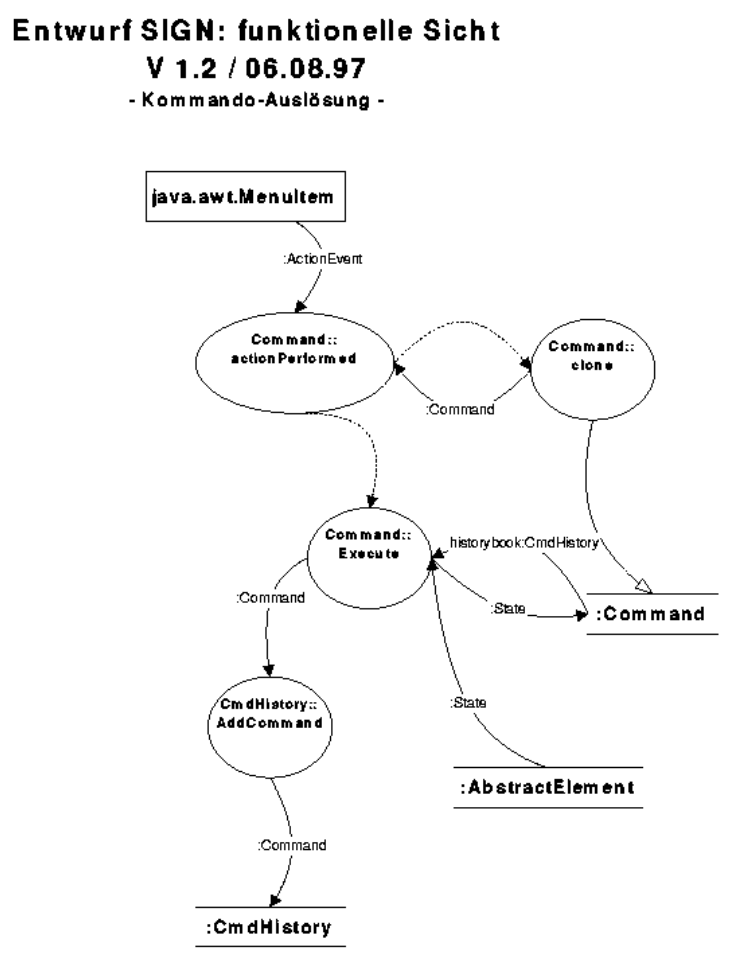


Abbildung 4.20:

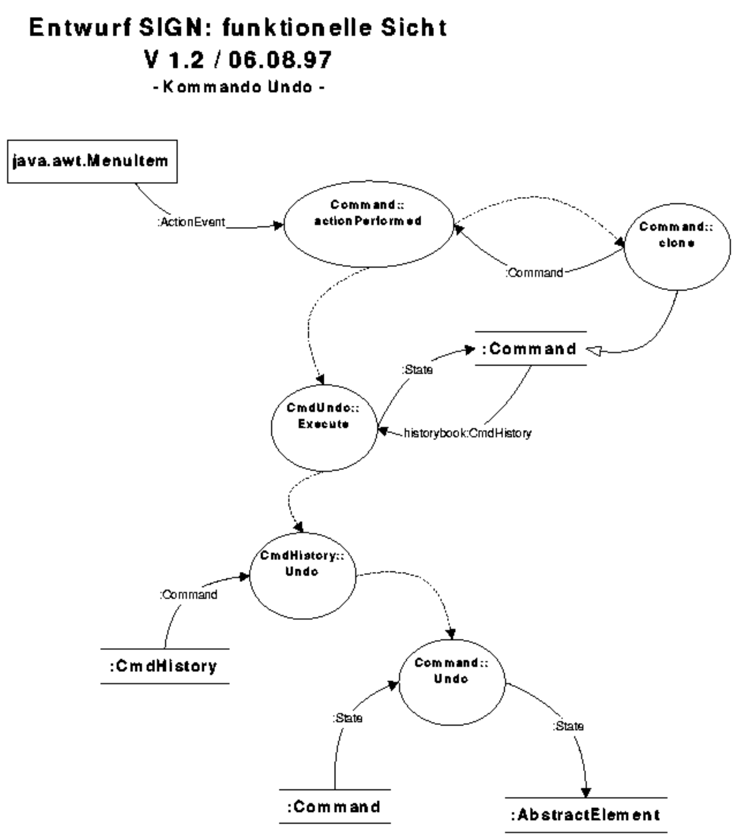


Abbildung 4.21:

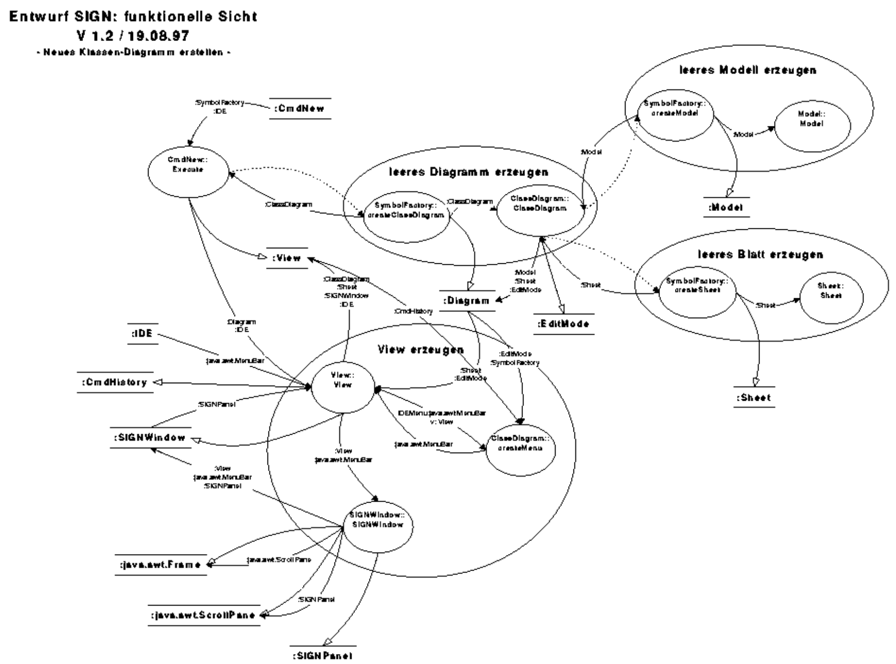


Abbildung 4.22:

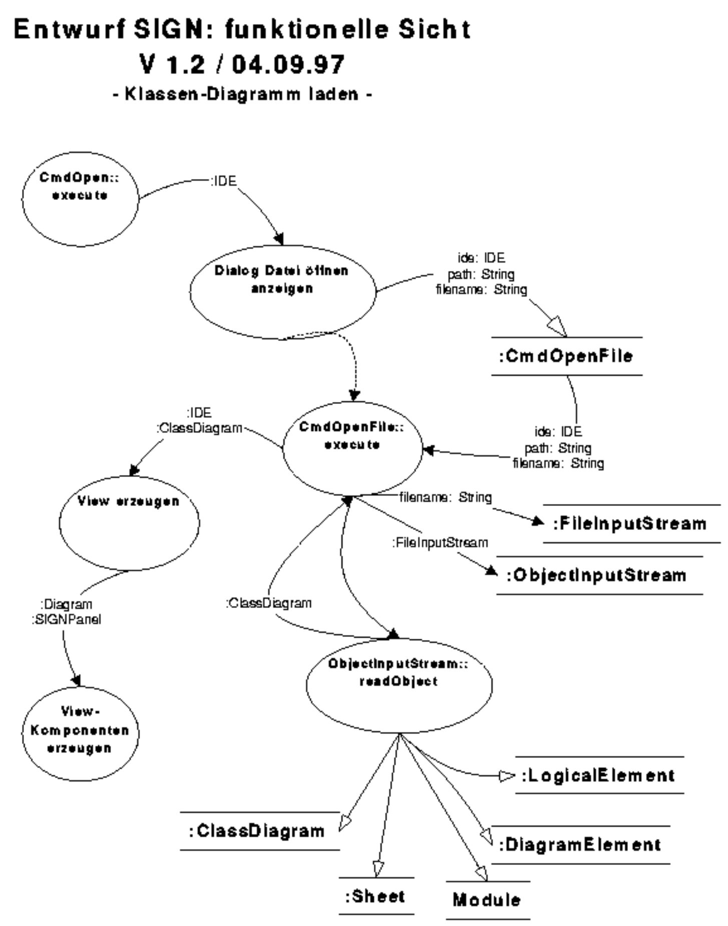


Abbildung 4.23:

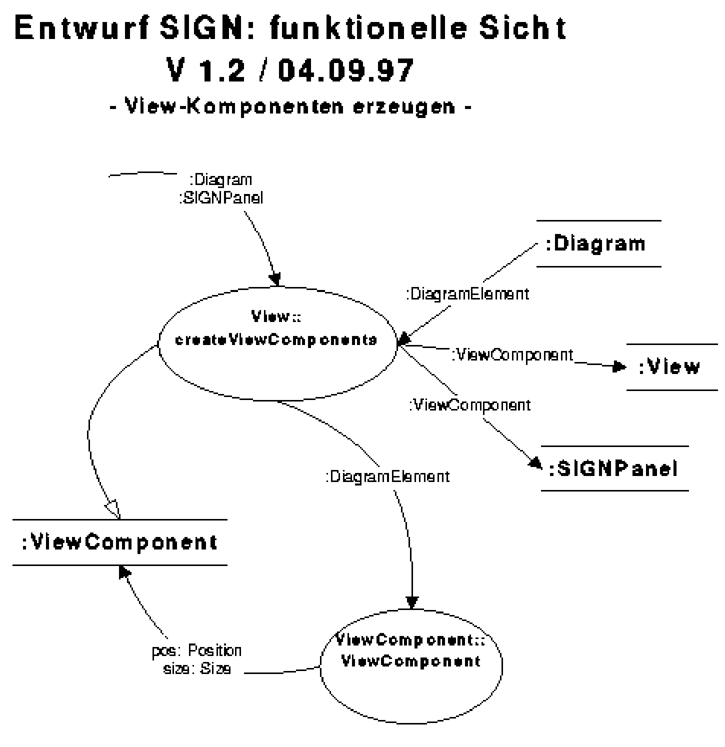


Abbildung 4.24:

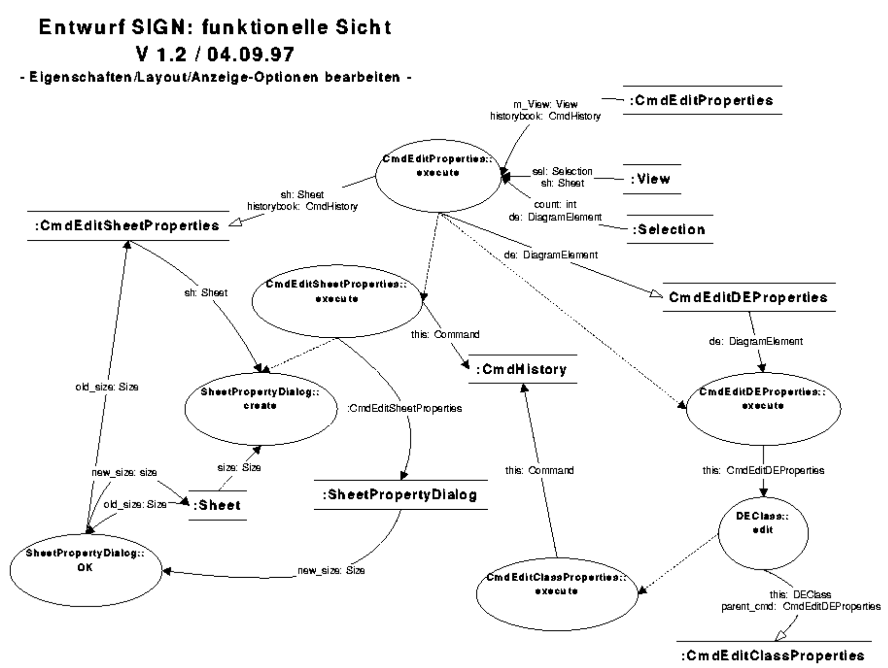


Abbildung 4.25:

Operationen

Es gibt Operationen, die nicht im Klassendiagramm aufgeführt sind, deren Existenz aber implizit vorrausgesetzt wird. Dies sind Operationen zum Lesen von Attributwerten (getAttributeName()), zum Schreiben von Attributwerten (setAttributeName(value)) sowie zum Erfragen von assoziierten oder aggregierten Objekten (getRolename()).

Ferner existieren Methoden zum Hinzufügen von Objekten zu Assoziationen oder Aggregationen (addRolename(object), wobei Rolename der Name der Rolle von object ist) und zum Entfernen von Objekten aus Assoziationen oder Aggregationen (removeRolename(object)). Anmerkung: Alternativ könnte vereinbart werden, daß beim Erfragen von Assoziationen und Aggregationen stets Referenzen als Ergebnis geliefert werden. Dann kann das Hinzufügen und Entfernen von Objekten mittels der für die Beziehungstypen vereinbarten Standardoperationen erfolgen. Dies Verfahren wird hier nicht gewählt. Es würde für alle Beziehungstypen eine gemeinsame Oberklasse zur Definition des Protokolls vorraussetzen. Wir wollen jedoch zur Implementation von Beziehungen zwischen Klassen die Benutzung von Konstrukten der Programmiersprache bzw. Datenstrukturen vorhandener Bibliotheken vorsehen.

Implizit gibt es zu jeder Klasse Konstruktoren (mit/ohne Parameter) und Destruktor.

Assoziationen

Wenn zwischen Objekten zweier Klassen A,B eine Assoziation besteht, so haben i.A. Objekte der Klasse A den Zugriff auf die assoziierten Objekte der Klasse B und umgekehrt. Der Zugriff wird realisiert durch eine private Member-Variable. Die Benennung orientiert sich am Rollennamen bzw., falls kein Rollenname vorhanden, am Namen der Klasse der assoziierten Objekte.

Zur internen Realisierung einer Assoziation ist prinzipiell jeder Datentyp (z.B. Vector) zulässig. Wenn ein Bezeichner für die Assoziation gegeben ist, sollte der Typ dieser Assoziation separat spezifiziert sein.

Für jede Assoziation muß die Klasse die impliziten Operationen getRolename(): Enumeration, addRolename(...): void, removeRolename(...): void implementieren. Im obigen Beispiel stellt die Klasse A die Operationen getB(): Enumeration, addB(B b): void, removeB(B b): void zur Verfügung. Die Klasse B stellt analog die Operationen getA(): Enumeration, addA(A a): void, removeA(A a): void zur Verfügung.

Eine Ausnahme kann gelten, wenn die Multiplizität der Assoziation max. 1 ist. Dann kann eine einfache Referenz (Pointer) zur internen Realisierung verwendet werden. In diesem Fall hat die Klasse ein zusätzliches implizites Attribut, das mit den Operationen getRolename() und setRolename(...) manipuliert wird. Sei ein Objekt der Klasse A mit höchstens einem Objekt der Klasse B assoziiert, so gibt es in A die Operationen getB(): B und setB(B b): void.

Dies sind die Minimalanforderungen zur Implementation von Assoziationen. Um etwa einen effizienteren Zugriff auf die assoziierten Objekte zu ermöglichen, kann eine Klasse auch weitere Operationen exportieren (z.B.: getBVec(): Vector, getBHash(): Hashtable, etc.).

4.1.3 Transformation nach SQL

Da man mit deSIGN nicht nur Zeichnen können soll, sondern die so entstandene Zeichnung auch direkt in ein geeignetes Format exportieren lassen kann, sind einige Vorüberlegungen zu diesem Thema nötig. Wie im Projektplan festgehalten, wollen wir uns in einer ersten Phase mit einer Übersetzung nach SQL begnügen. Hierzu werden wir nach und nach alle in der Basisnotation beschriebenen Elemente durchgehen und so aufzeigen, wie ein Exportformat nach SQL aussehen könnte.

4.1.3.1 Objekte der SIGN Grafik

Eine deSIGN Grafik kann nach der Basisnotation aus Klassen, (qualifizierten) Assoziationen, Aggregationen und Generalisierungen bestehen. Diese wollen wir in diesem Kapitel näher diskutieren.

Jede Tabelle läßt sich in SQL einem Schema zuordnen. Wir gehen dabei davon aus, daß alle in einem Modul befindlichen SIGN-Objekte einem Schema angehören sollen. Dieses erhält dann den Namen des Moduls.

Grundsätzlich läßt sich sagen, daß durch die Portierung nach SQL eine Menge von Tabellen erzeugt werden. Aus jedem Objekt wird mittels

```
CREATE TABLE ...
```

eine Tabelle generiert. Wie dies im einzelnen funktioniert, sollen die einzelnen Abschnitte zeigen. Im nächsten Kapitel wird anhand einer Beispielgrafik ein komplexerer Fall erläutert.

Klassen

Eine Klasse hat grundsätzlich einen Klassennamen. Optional kann eine Klasse auch noch über Attribute und Operationen verfügen. Letztere sind für die Generierung von SQL-Codesegmenten jedoch nicht wichtig und werden bei der Übersetzung so auch nicht beachtet.

Anders der Klassenname. Dieser ist Grundlage für die Erzeugung der Tabellen. Ebenso wichtig sind die Attribute. Da wir in unserer Basisnotation keine Möglichkeit haben, Schlüsselattribute zu benennen, wird für jede Klasse automatisch ein weiteres Attribut mit dem Namen *id* generiert (dieses ist vom Typ Integer). Wurde für ein angegebenes Attribut kein Typ vereinbart, so weisen wir diesem den Typ *char(99)* zu.

Grundsätzlich gilt für Typen: Befindet sich der angegebene Typ in der TranslationTable (Datei "./TranslationTable1.tt"), so wird dieser in den gewünschten Typ übersetzt. In der Basisversion werden hierbei Number in Integer und Text in String übersetzt. Die Datei läßt sich beliebig erweitern (nicht nur um Typen, sondern auch um Zielsprachenvorgaben).

Aus dem obigen Bild(Abbildung 4.26) ergibt sich folgende Anweisung:

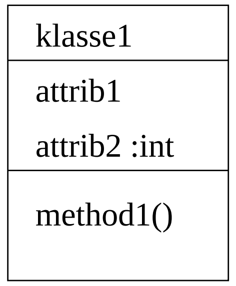


Abbildung 4.26:

```
CREATE TABLE klasse1(
  id int,
  attrib1 char(99),
  attrib2 int,
  PRIMARY KEY (id)
);
```

Assoziationen

Assoziationen werden grundsätzlich zwischen zwei oder mehr Klassen gezogen. Aus diesen entstehen Tabellen, die als Attribute die Schlüsselattribute (id) der beteiligten Klassen sowie die eigenen Attribute (Assoziationsattribute) enthalten. Assoziationen können ausserdem Kardinalitäten aufweisen. Diese werden jedoch nicht durch weitere Tabellen abgebildet, sondern durch Integritätsbedingungen. Ebenso kann eine Assoziation eine qualifizierte Assoziation sein. In diesem Fall fügen wir der Assoziation ein weiteres Attribut hinzu (das Qualifizierungsattribut), welches als Schlüssel fungiert. Wurde für die Assoziation kein Name angegeben, so wird dieser aus den Namen der beteiligten Klassen zusammengesetzt.

Aus dem obigen Bild(Abbildung 4.27) ergibt sich folgende Anweisung, wobei wir davon ausgehen, daß die einzelnen an den Assoziationen beteiligten Klassen schon definiert wurden:

```
//3-ary Association
CREATE TABLE asz1(
  klasse1_id int,
  klasse2_id int,
  klasse3_id int,
  id int,
  PRIMARY KEY (id),
  FOREIGN KEY (klasse1_id) REFERENCES klasse1(id) ON DELETE CASCADE,
  FOREIGN KEY (klasse2_id) REFERENCES klasse2(id) ON DELETE CASCADE,
  FOREIGN KEY (klasse3_id) REFERENCES klasse3(id) ON DELETE CASCADE
);

//Qualifizierte Assoziation
```

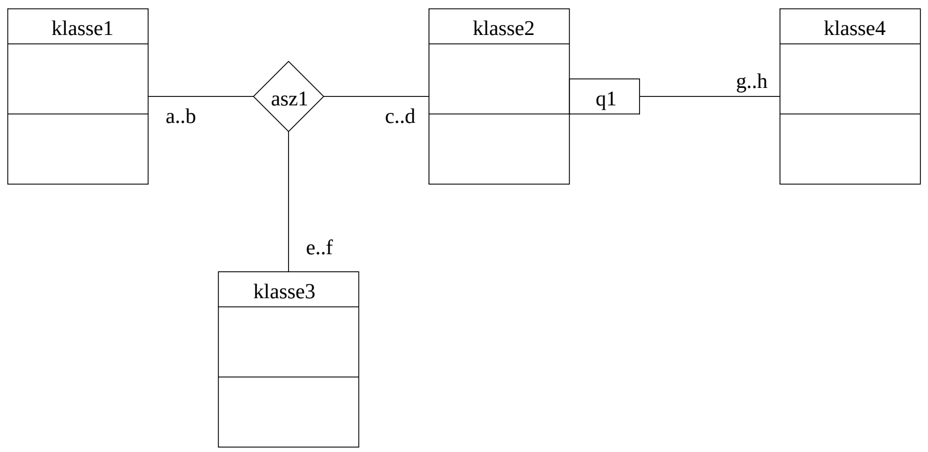


Abbildung 4.27:

```
CREATE TABLE klasse2_klasse4(
  klasse2_id int,
  klasse4_id int,
  q1 char(99),
  id int,
  PRIMARY KEY (id),
  FOREIGN KEY (klasse2_id) REFERENCES klasse2(id) ON DELETE CASCADE,
  FOREIGN KEY (klasse4_id) REFERENCES klasse4(id) ON DELETE CASCADE
);

//Kardinalitäten für 3-äre Assoziation
CREATE ASSERTION ass_asz1 CHECK(
  NOT EXISTS(
    SELECT COUNT(*)
    FROM asz1
    GROUP BY klasse1_id, klasse2_id
    HAVING (COUNT(*) < e OR COUNT(*) > f))
  AND
  NOT EXISTS(
    SELECT COUNT(*)
    FROM asz1
    GROUP BY klasse2_id, klasse3_id
    HAVING (COUNT(*) < a OR COUNT(*) > b))
  AND
  NOT EXISTS(
    SELECT COUNT(*)
    FROM asz1
    GROUP BY klasse3_id, klasse1_id
    HAVING (COUNT(*) < c OR COUNT(*) > d))
);
```

```
//Kardinalitäten für qualifizierte Assoziation
CREATE ASSERTION ass_klasse2_klasse4 CHECK(
  NOT EXISTS(
    SELECT COUNT(*)
    FROM klasse2_klasse4
    GROUP BY klasse2_id, klasse4_id
    HAVING (COUNT(*) < g OR COUNT(*) > h))
);
```

Aggregationen

Bei einer Aggregation verwendet die aggregierende Klasse Instanzen der aggregierten Klasse. Dies bilden wir ab, indem wir in der aggregierten Klasse auf die aggregierende verweisen.

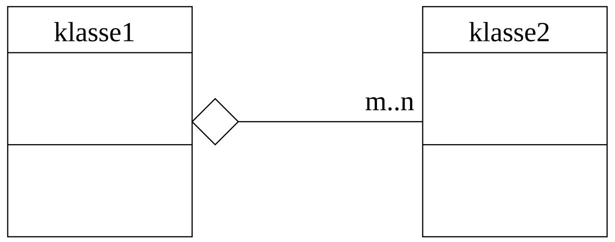


Abbildung 4.28:

Aus dem obigen Bild(Abbildung 4.28) ergibt sich folgende Anweisung:

```
CREATE TABLE klasse1(
  id int,
  PRIMARY KEY (id)
);

CREATE TABLE klasse2(
  id int,
  klasse1_id int,
  PRIMARY KEY (id),
  FOREIGN KEY (klasse1_id) REFERENCES klasse1(id) ON DELETE CASCADE
);

CREATE ASSERTION ass_klasse2 CHECK(
  NOT EXISTS(
    SELECT COUNT(*)
    FROM klasse2
    GROUP BY klasse1_id
    HAVING (COUNT(*) < m OR COUNT(*) > n))
);

CREATE ASSERTION ass_klasse1 CHECK(
```

```
  NOT EXISTS(
    SELECT COUNT(*)
    FROM klasse1
    GROUP BY klasse2_id
    HAVING (COUNT(*) < 0 OR COUNT(*) > 1))
);
```

Generalisierungen

Die Generalisierung wird verwendet, um Vererbungshierarchien zu schaffen. Hierbei vererbt die eine Klasse ihre Attribute (und Methoden) an die andere(n). Wir bilden dies ab, indem die erbende Klasse einen Verweis auf die geerbte(n) Klasse(n) enthält.

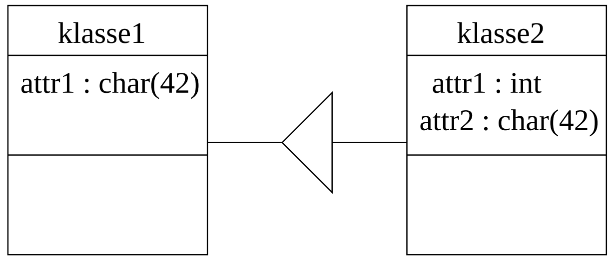


Abbildung 4.29:

Aus dem obigen Bild(Abbildung 4.29) ergibt sich folgende Anweisung:

```
CREATE TABLE klasse1(
  id int,
  attr1 char(42),
  PRIMARY KEY (id)
);

CREATE TABLE klasse2(
  id int,
  klasse1_id int,
  attr1 int,
  attr2 char(42),
  FOREIGN KEY (klasse1_id) REFERENCES klasse1(id) ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY (id)
);
```

Aus dem obigen Bild(Abbildung 4.30) ergibt sich folgende Anweisung:

```
CREATE TABLE klasse1(
  id int,
  attr1 char(42),
  PRIMARY KEY (id)
```

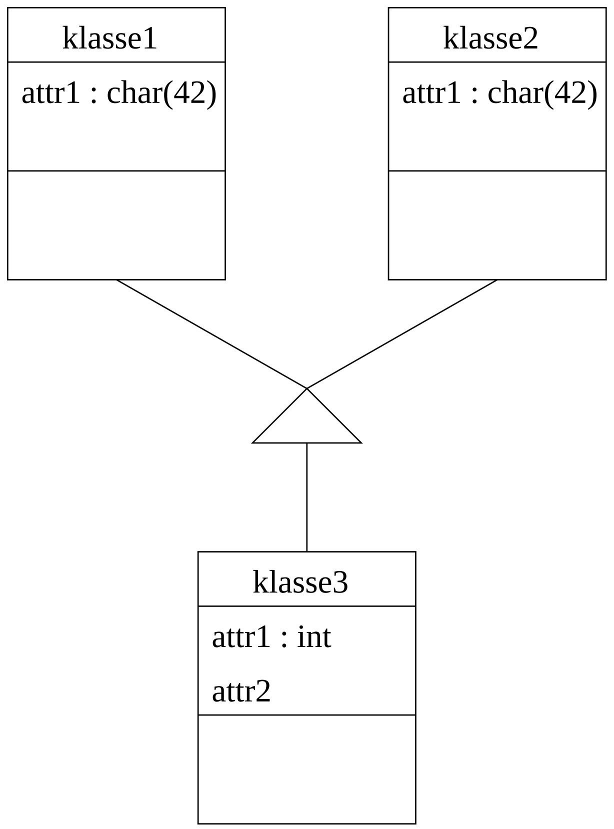


Abbildung 4.30:

```
);

CREATE TABLE klasse2(
  id int,
  attr1 char(42),
  PRIMARY KEY (id)
);

CREATE TABLE klasse3(
  id int,
  klasse1_id int,
  klasse2_id int,
  attr1 int,
  attr2 char(99),
  FOREIGN KEY (klasse1_id) REFERENCES klasse1(id) ON DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (klasse2_id) REFERENCES klasse2(id) ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY (id)
);
```

4.1.3.2 Beispielübersetzung

Aus dem obigen Bild(Abbildung 4.31) ergibt sich folgende Anweisung:

```
CREATE SCHEMA bsp;

CREATE TABLE bsp.kl1(
  id int,
  attr1 typ1,
  PRIMARY KEY (id)
);

CREATE TABLE bsp.kl2(
  id int,
  kl1_id int,
  attr1 typ2,
  attr2 char(99),
  FOREIGN KEY (kl1_id) REFERENCES bsp.kl1(id) ON DELETE CASCADE ON UPDATE CASCADE,
  PRIMARY KEY (id)
);

CREATE TABLE bsp.kl3(
  id int,
  attr1 char(99),
  PRIMARY KEY (id)
);

CREATE TABLE bsp.kl4(
  id int,
  attr1 char(99),
```

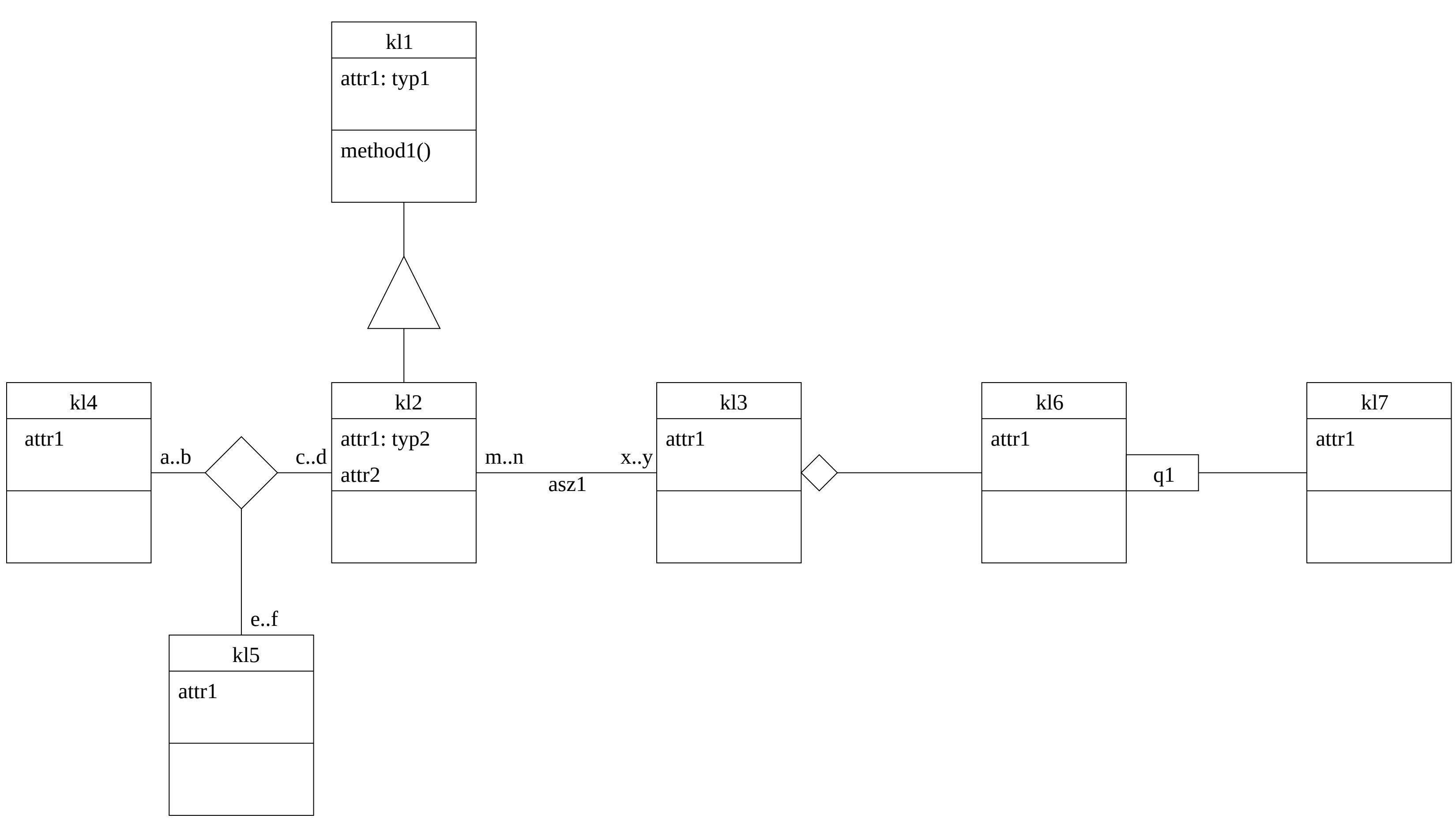


Abbildung 4.31:

```
  PRIMARY KEY (id)
);

CREATE TABLE bsp.kl5(
  id int,
  attr1 char(99),
  PRIMARY KEY (id)
);

CREATE TABLE bsp.kl6(
  id int,
  attr1 char(99),
  kl3_id int,
  PRIMARY KEY (id),
  FOREIGN KEY (kl3_id) REFERENCES bsp.kl3(id) ON DELETE CASCADE
);

CREATE TABLE bsp.kl7(
  id int,
  attr1 char(99),
  PRIMARY KEY (id)
);

CREATE TABLE bsp.kl2_kl4_kl5(
  kl2_id int,
  kl4_id int,
  kl5_id int,
  FOREIGN KEY (kl2_id) REFERENCES bsp.kl2(id) ON DELETE CASCADE,
  FOREIGN KEY (kl4_id) REFERENCES bsp.kl4(id) ON DELETE CASCADE,
  FOREIGN KEY (kl5_id) REFERENCES bsp.kl5(id) ON DELETE CASCADE
);

CREATE TABLE bsp.asz1(
  kl2_id int,
  kl3_id int,
  FOREIGN KEY (kl2_id) REFERENCES bsp.kl2(id) ON DELETE CASCADE,
  FOREIGN KEY (kl3_id) REFERENCES bsp.kl3(id) ON DELETE CASCADE
);

CREATE TABLE bsp.kl6_kl7(
  kl6_id int,
  kl7_id int,
  q1 int,
  PRIMARY KEY (q1),
  FOREIGN KEY (kl6_id) REFERENCES bsp.kl6(id) ON DELETE CASCADE,
  FOREIGN KEY (kl7_id) REFERENCES bsp.kl7(id) ON DELETE CASCADE
);

CREATE ASSERTION ass_kl3 CHECK(
  NOT EXISTS(
```

```
    SELECT COUNT(*)
    FROM bsp.kl3
    GROUP BY bsp.kl6_id
    HAVING (COUNT(*) < 0 OR COUNT(*) > 1))
);

CREATE ASSERTION ass_kl6 CHECK(
  NOT EXISTS(
    SELECT COUNT(*)
    FROM bsp.kl6
    GROUP BY bsp.kl3_id
    HAVING (COUNT(*) < g OR COUNT(*) > h))
);

CREATE ASSERTION bsp.ass_asz1 CHECK(
  NOT EXISTS(
    SELECT COUNT(*)
    FROM bsp.asz1
    GROUP BY bsp.kl3_id
    HAVING (COUNT(*) < m OR COUNT(*) > n))
  AND
  NOT EXISTS(
    SELECT COUNT(*)
    FROM bsp.asz1
    GROUP BY bsp.kl2_id
    HAVING (COUNT(*) < x OR COUNT(*) > y))
);

CREATE ASSERTION bsp.ass_kl2_kl4_kl5 CHECK(
  NOT EXISTS(
    SELECT COUNT(*)
    FROM bsp.kl2_kl4_kl5
    GROUP BY bsp.kl4_id, bsp.kl5_id
    HAVING (COUNT(*) < c OR COUNT(*) > d))
  AND
  NOT EXISTS(
    SELECT COUNT(*)
    FROM bsp.kl2_kl4_kl5
    GROUP BY bsp.kl2_id,bsp. kl5_id
    HAVING (COUNT(*) < a OR COUNT(*) > b))
  AND
  NOT EXISTS(
    SELECT COUNT(*)
    FROM bsp.kl2_kl4_kl5
    GROUP BY bsp.kl4_id, bsp.kl2_id
    HAVING (COUNT(*) < e OR COUNT(*) > f))
);
```

4.1.3.3 Implementierung/Ergänzungen

Bei der Implementierung gehen wir wie folgt vor: In der Klasse CSQLGen werden der Reihe nach alle Elemente (Klassen, Assoziationen, Aggregationen, Generalisierungen) durchlaufen und deren Visitoren aufgerufen. Dort werden mittels CSQLTable Tabellen erzeugt, Attribute und Schlüssel angehängt. Die Klasse SQLAssertion ist für die Erstellung und Modifikation von Integritätsbedingungen zuständig. Hier haben wir auch die Möglichkeit geschaffen, Klammern zu setzen, OR-Bedingungen einzufügen und Multiplizitäten zu verwenden. Diese müssen allerdings im „richtigen“ Format vorliegen (z.B. 1,7,9,11..20,30,..*). Für weitere Informationen verweisen wir an dieser Stelle auf die Programmdokumentation.

4.1.4 Transformation nach Java

Wie bei der Übersetzung nach SQL sind auch bei der Transformation von deSIGN-Grafiken nach Java (oder auch anderen objektorientierten Sprachen) einige Vorüberlegungen angebracht. Grundsätzlich kann man davon ausgehen, daß alle Exportformate, die eine objektorientierte Sprache zum Ziel haben, nach dem gleichen Muster erzeugt werden können. So hat man (hoffentlich) keinen allzu großen Arbeitsaufwand, wenn man statt Java eine andere Sprache (z.B. C++) verwenden möchte.

4.1.4.1 Objekte der SIGN Grafik

Eine deSIGN Grafik kann nach der Basisnotation aus Klassen, (qualifizierten) Assoziationen, Aggregationen und Generalisierungen bestehen. Diese wollen wir in diesem Kapitel näher diskutieren. Im nächsten Kapitel wollen wir dann anhand einer komplexeren Grafik eine Beispielübersetzung durchführen.

Klassen

Unter Java benötigt jede Klasse eine eigene Datei, die den gleichen Namen tragen muß. Dabei erfolgt der Aufbau einer Klasse immer nach dem gleichen Schema:

```
class KLASSENNAME
{
  KONSTRUKTOR;
  DESTRUKTOR;
  VARIABLEN;
  METHODEN;
}
```

Hierbei erhalten der Konstruktor und der Destruktor die durch den Klassennamen vorgegebenen Namen und die Variablen und Methoden den durch die deSIGN-Grafik bestimmten Namen. Man beachte, daß man den einzelnen „Objekten“ auch noch sogenannte „Spezifizierer“ zuordnen kann, wobei wir standardmäßig *protected* verwenden:

```
private
private protected
protected
public
friendly
final
static
abstract
transient
volatile
```

Jede Klasse kann unter Java einem *package* zugeordnet werden. Den Namen des Paketes erhalten wir dabei aus dem Modul; wurde kein Modulname angegeben (der Name ist also „untitled“), wird kein Paket verwendet. Nun wollen wir die einzelnen Punkte (Variablen, Konstruktor, Destruktor und Methoden) getrennt betrachten und erklären, wie der Code erzeugt werden soll.

Nach der SIGN-Basisnotation kann man Variablen auch einen Typ zuordnen. Sollte der Nutzer keinen Namen angegeben haben, so stehen wir hier vor einem Problem. In diesem Fall verwenden wir einfach den Typ *void*. Hat der User einen Typ angegeben, der in Java keine Entsprechung hat, so können wir leider keine Fehlerkorrektur durchführen, da der User eventuell eine Klasse gemeint hat.

Grundsätzlich gilt für Typen: Befindet sich der angegebene Typ in der TranslationTable (Datei „./TranslationTable1.tt“), so wird dieser in den gewünschten Typ übersetzt. In der Basisversion werden hierbei *Number* in *int* und *Text* in *String* übersetzt. Die Datei läßt sich beliebig erweitern (nicht nur um Typen, sondern auch um Zielsprachenvorgaben).

Der Konstruktor erhält grundsätzlich den gleichen Namen wie die Klasse. Zwar erlaubt Java auch die Verwendung von mehreren Kontruktoren, jedoch kann man nicht wissen, welche der User gerne hätte. Hat dieser jedoch in den Methoden weitere angegeben (dadurch zu erkennen, daß kein Rückgabewert angegeben wurde und der Methodenname gleich dem Klassennamen ist), so werden diese selbstverständlich auch übernommen.

Der Destruktor ist in Java zwar nicht unbedingt nötig (da Java über eine eigene Garbage-Collection verfügt), jedoch wollen wir diesen auf alle Fälle erzeugen. Dieser erhält dann den Namen *finalize*.

Unter deSIGN kann man natürlich auch Methoden benennen. Hier werden diese besprochen, die *keine* Konstruktoren sind. Wie auch bei den Variablen (Attribute) unterscheiden wir hier einige Fälle: Der User kann keinen Rückgabewert angegeben haben, dann weisen wir hier automatisch den Typ *void* zu. Hat der User einen unbekannten Typ angegeben, so können wir leider (wie oben beschrieben) keine Fehlerkorrektur durchführen. Die gleiche Behandlung erfolgt für die Parameter.

Zur Verdeutlichung hier schon mal eine Beispielübersetzung einer einzelnen Klasse:

```
class klasse1 {
```

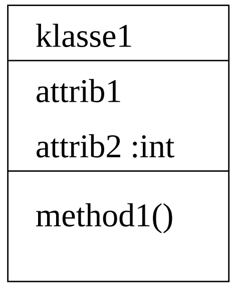


Abbildung 4.32:

```
  klasse1() {};
  finalize() {};
  void attrib1;
  int attrib2;
  void method1() {};
}
```

Assoziationen, Aggregationen und qualifizierte Assoziationen

Für alle diese Arten von Verbindungen müssen die entsprechenden Klassen mittels *import* eingebunden werden.

Wie auch schon für die Übersetzung nach SQL werden bei der Transformation nach Java separate Klassen für die Assoziationen benötigt. Würden wir statt dessen nur einfache Variablen anlegen, so hätten wir bei n-ären Assoziationen ein unlösbares Problem. In der Assoziationsklasse finden wir also Verweise auf die beteiligten Klassen inklusive der zu verwendenden Zugriffsoperationen. In den Klassen hingegen wird die Assoziationsklasse über Vectoren referenziert.

Handelt es sich um eine Aggregation, so wird in der aggregierenden Klasse eine Liste (hier aus Gründen der Handhabbarkeit *vector*) von Variablen vom Typ der aggregierten Klasse angelegt. Wurde für die Aggregation ein Rollenname angegeben, so wird dieser verwendet, ansonsten generieren wir diesen als aggr_xxx, wobei *xxx* für den Namen der aggregierten Klasse steht.

Der letzte Fall ist die qualifizierte Assoziation. In diesem Fall legen wir eine Hashtable an, wobei diese den Namen des Qualifiers erhält.

Generalisierungen

Im Fall der Einfachvererbung können wir an die erste Zeile der Klassendefinition *class* KLASSENNAME ein *extends* KLASSENNAME2 anhängen. Leider ist es unter Java nicht gestattet, fertig ausimplementierte Klassen mittels implements einzubinden; man ist hier auf die Definition von Interfaces angewiesen. Diese können dann auch im Rahmen einer Mehrfachvererbung verwendet werden. Allerdings können wir unter deSIGN leider kein Attribut implements oder abstract angeben und müssen so leider auf dieses Sprachfeature verzichten. Der Benutzer hat selber auf die korrekte Eingabe zu achten.

4.1.4.2 Beispielübersetzung

Hier soll anhand einer komplexeren Grafik die Transformation nach Java erläutert werden:

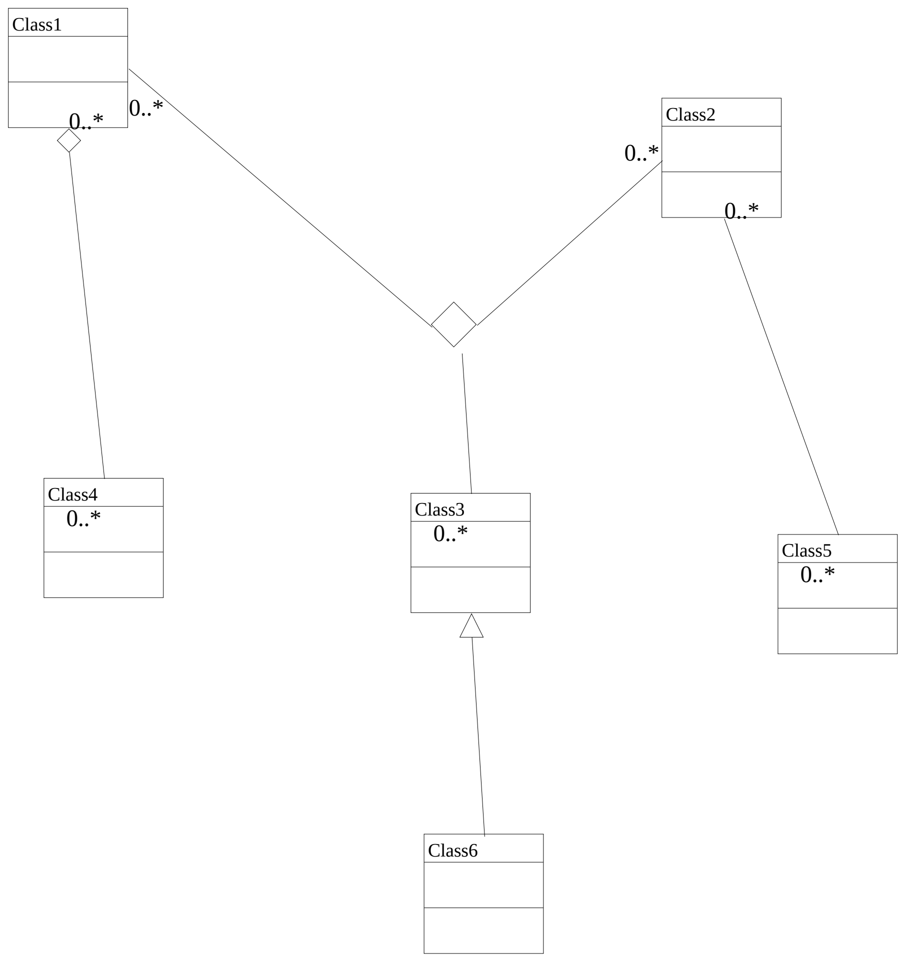


Abbildung 4.33:

```
//Anfang von Class1.java
package untitled;
// Imports ...
import java.util.Vector;

class Class1
{
// Attributes ...
private Vector Assoc_Class1_Class2_Class3 = new Vector();
private Vector Class4Elem = new Vector();
```

```
// Constructors ...
public Class1()
   {
   }

// Methodes
public void addClass4(Class4 aggregate)
    {
        // add one Aggregate.
        Class4Elem.addElement(aggregate);
    }

public void removeClass4(Class4 aggregate)
    {
        // remove one Aggregate.
        Class4Elem.removeElement(aggregate);
    }

public Enumeration getClass4s()
    {
        // get the Elements.
        return Class4Elem;
    }


}
//Ende von Class1.java

//Anfang von Class2.java
package untitled;
// Imports ...

class Class2
{
// Attributes ...
private Vector Assoc_Class1_Class2_Class3 = new Vector();
private Vector Assoc_Class2_Class5 = new Vector();

// Constructors ...
public Class2()
   {
   }

// Methodes

}
//Ende von Class2.java

//Anfang von Class3.java
```

```
package untitled;
// Imports ...

class Class3
{
// Attributes ...
private Vector Assoc_Class1_Class2_Class3 = new Vector();

// Constructors ...
public Class3()
   {
   }

// Methodes

}
//Ende von Class3.java

//Anfang von Class4.java
package untitled;
// Imports ...

class Class4
{
// Attributes ...

// Constructors ...
public Class4()
   {
   }

// Methodes

}
//Ende von Class4.java

//Anfang von Class5.java
package untitled;
// Imports ...

class Class5
{
// Attributes ...
private Vector Assoc_Class2_Class5 = new Vector();

// Constructors ...
public Class5()
   {
   }
```

```
// Methodes

}
//Ende von Class5.java

//Anfang von Class6.java
package untitled;
// Imports ...

class Class6 extends Class3
{
// Attributes ...

// Constructors ...
public Class6()
   {
   }

// Methodes

}
//Ende von Class6.java

//Anfang von Assoc_Class1_Class2_Class3.java
package untitled;
// Imports ...
import java.util.Vector;

class Assoc_Class1_Class2_Class3
{
// Attributes ...
private Class1 myRoleOfClass1;
private Class2 myRoleOfClass2;
private Class3 myRoleOfClass3;

// Constructors ...
public Assoc_Class1_Class2_Class3()
   {
   }

// Methodes
public Assoc_Class1_Class2_Class3(Class1 RoleOfClass1, Class2 RoleOfClass2, Class3 RoleOfClass3)
    {
        myRoleOfClass1 = RoleOfClass1;
        myRoleOfClass2 = RoleOfClass2;
        myRoleOfClass3 = RoleOfClass3;
    }

public Class3 getRoleOfClass3()
    {
```

```
        // get the Participant Classes of this Association.
        return myRoleOfClass3;
    }

public Class2 getRoleOfClass2()
    {
        // get the Participant Classes of this Association.
        return myRoleOfClass2;
    }

public Class1 getRoleOfClass1()
    {
        // get the Participant Classes of this Association.
        return myRoleOfClass1;
    }


}
//Ende von Assoc_Class1_Class2_Class3.java

//Anfang von Assoc_Class2_Class5.java
package untitled;
// Imports ...
import java.util.Vector;

class Assoc_Class2_Class5
{
// Attributes ...
private Class2 myRoleOfClass2;
private Class5 myRoleOfClass5;

// Constructors ...
public Assoc_Class2_Class5()
   {
   }

// Methodes
public Assoc_Class2_Class5(Class2 RoleOfClass2, Class5 RoleOfClass5)
    {
        myRoleOfClass2 = RoleOfClass2;
        myRoleOfClass5 = RoleOfClass5;
    }

public Class5 getRoleOfClass5()
    {
        // get the Participant Classes of this Association.
        return myRoleOfClass5;
    }

public Class2 getRoleOfClass2()
```

```
    {
        // get the Participant Classes of this Association.
        return myRoleOfClass2;
    }


}
//Ende von Assoc_Class2_Class5.java
```

4.1.4.3 Implementierung/Ergänzungen

Bei der Implementierung gehen wir wie folgt vor: In der Klasse *CJavaGen* werden der Reihe nach alle Elemente (Klassen, Assoziationen, Aggregationen, Generalisierungen) durchlaufen und deren Visitoren aufgerufen. Dort werden mittels *CJavaClass* Klassen erzeugt, Attribute und Methoden angehängt. Damit der Anwender nicht die alle Einzelheiten von Java, SQL, C++ und anderen Sprachen im Kopf haben muß, haben wir den Zugriff auf die VariablenTypen über eine Datei organisiert. Kardianlitäten werden zur Zeit noch nicht berücksichtigt, müssen dann allerdings im „richtigen“ Format vorliegen (z.B. 1,7,9,11..20,30,..*). Ein weiteres nicht-übersetztes Diagrammelement ist der Qualifier, für den wir leider kein wirklich passendes Konzept gefunden haben; vielleicht schaffen wir es in der nächsten Version (deSIGN II ?). Für weitere Informationen verweisen wir an dieser Stelle auf die Programmdokumentation.

4.1.5 Entwurfserweiterungen

4.1.5.1 Verbinderendpunkte

Mit dem folgenden Konzept sollen die Entwurfserweiterungen für Qualifier und Verbinderendpunkte erläutert werden.

Das Konzept für Qualifier stellt insofern eine Besonderheit dar, als das ein Qualifier Diagrammelement zugleich ein Verbinderendpunkt ist, der via Drag'n Drop an Ankerpunkte der selben als auch an Ankerpunkte anderer Klassen verschoben werden kann.

Qualifier

Binären Assoziationen kann ein Qualifier zugeordnet werden, indem ein Attribut angegeben wird, das den Zugriff auf ein einzelnes Element oder auf eine Untermenge von Elementen der Assoziation ermöglicht.

Das Qualifier-Attribut ist als Bestandteil der Assoziation aufzufassen. Es ist daher keiner Klasse, sondern direkt dem Qualifier zugeordnet. Da ein Qualifier gemäß OMT-Notation - wie eine Rolle - an einem Ende der Assoziation dargestellt wird, ist der Qualifier der Rolle zugeordnet.

Beispiel: *(siehe Abbildung 4.34) Klasse A und B sind über Assoziation R verbunden. A hat in B die Rolle a, b hat in A die Rolle b. Das Attribut x wird*

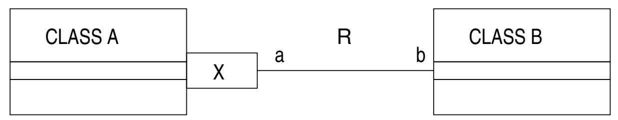


Abbildung 4.34: Beispiel für Qualifier

als Qualifier-Attribut gewählt und der Rolle a zugeordnet. Dann ist der Zugriff aus einem Objekt der Klasse A auf ein assoziiertes Objekt der Klasse B (oder auf eine Untermenge von assoziierten Objekten der Klasse B) über die Rolle b mit dem Qualifier x möglich. D.h. die Rolle b könnte durch ein assoziatives Array realisiert sein. Da x als Attribut der Assoziation gilt, ist der Zugriff unter Angabe des qualifizierenden Attributs x auch in umgekehrter Richtung möglich. D.h. die Rolle a kann auch als assoziatives Array realisiert sein. Im Diagramm wird der Qualifier jedoch nur an dem einen Ende der Assoziation R dargestellt. Im Beispiel bei der Rolle a auf der Seite von Klasse A.

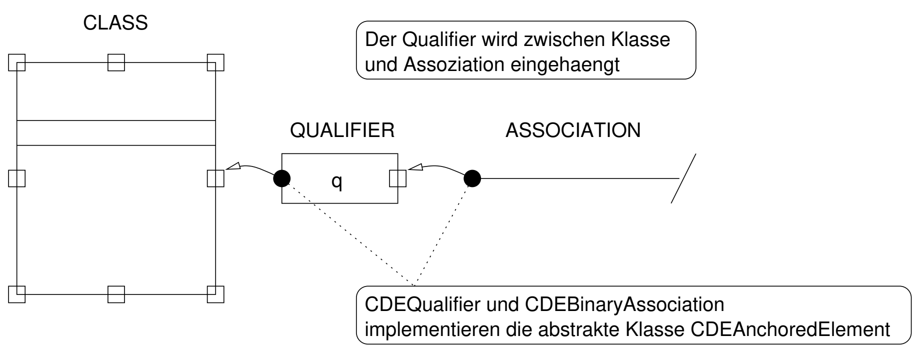


Abbildung 4.35: Schematische Darstellung

Bei genauerem Betrachten ist der Qualifier ein Diagrammelement, das sich zwischen BorderAnchor und AbstractAnchor einer bereits vorhandenen binären Assoziation bzw. eines NaryParticipants setzt (siehe Abbildung 4.35). Daher ist es erforderlich, bei der Erzeugung eines Qualifiers zunächst die Assoziation von der Klasse zu trennen, um dann den Qualifier an die Klasse zu hängen und schließlich die Assoziation wiederum an den Qualifier.

Diese Vorgehensweise erfordert, daß dem Qualifier der BorderAnchor der Klasse sowie das Diagrammelement der Assoziation bekannt sind. Zusätzlich bedarf es eines speziellen Ankers: dem QualifierAnchor (siehe Abbildung 4.36). Dieser wird wie auch NaryAnchor und BorderAnchor von AbstractAnchor abgeleitet. Der Qualifier selbst ist ein AnchoredElement.

Verbinderendpunkte

Der Qualifier ist auf Diagrammebene nichts weiter als ein Diagrammelement, das sich zwischen BorderAnchor einer Klasse und möglichen Relationen hängt.

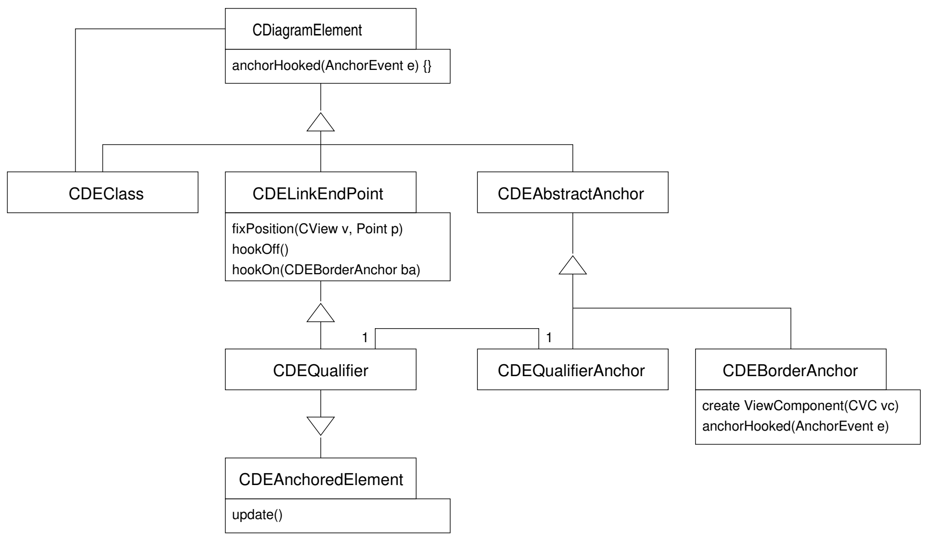


Abbildung 4.36: Klassendiagramm Verbinderendpunkte

Qualifier besitzen somit den selben Mechanismus wie Verbinderendpunkte - oder anders formuliert: Qualifier sind Verbinderendpunkte.

Der Unterschied liegt darin, daß Verbinderendpunkte kein logisches Element besitzen und ausschließlich dazu dienen, ein komfortables Arbeiten mit dem Editor zu ermöglichen. Sie sind dabei Teil eines Verbinders, d.h. einer binären-, nären Assoziation, Generalisierung oder Aggregation (siehe auch Konzept für rechtwinklige Verbinder Seite 220). Natürlich ist es ebenfalls nicht möglich Verbinderendpunkte mit Namen zu bezeichnen. Da sie lediglich eine Editierhilfe sind, fristen sie ein unsichtbares Dasein. Erst wenn mit der Mouse über die entsprechende ViewComponent gefahren wird, macht sich der Verbinderendpunkt mit einem Rechteck in Größe und Form der ViewCompoment bemerkbar.

Klassendiagramm

Aus dem Klassendiagramm (siehe Abbildung 4.36) geht hervor, daß Verbinderendpunkte CDELinkEndPoint Generalisierungen von Qualifiern CDEQualifier sind. Diese Vererbungshierachie ergibt sich aus der Sonderstellung der Qualifier.

Ferner besitzt der Qualifier genau einen Anchor, der eine Instanz der Klasse CDEQualifierAnchor ist.

Die Methode fixPosition der Klasse CDELinkEndPoint wird aufgerufen, wenn nach einem Drag der Verbinderendpunkt (oder Qualifier) 'losgelassen' wird. Es wird ein Event erzeugt (siehe Event Modell), den alle BorderAnchor empfangen. Tritt der Fall ein, daß tatsächlich verschoben werden muß, sorgen die Methoden hookOff und hookOn für das Aushängen bzw. Einhängen der Relation in den entsprechenden BorderAnchor.

Anwendungsbeispiel:

Das Arbeiten mit Verbinderendpunkten könnte sich wie folgt darstellen: Der Benutzer modelliert an einem Klassendiagramm mit mehreren Klassen, die durch verschiedene Relationen untereinander verknüpft sind. Er stellt fest, daß eine Relation falsch modelliert oder schlecht plaziert ist und möchte eine Seite der Relation an einen anderen BorderAnchor der selben oder einer anderen Klasse verschieben. Dazu bringt der Benutzer den Mauszeiger in die Nähe des BorderAnchors, an dem die zu verschiebende Seite der Relation liegt. Der Verbinderendpunkt wird gehighlighted in dem Moment, in dem der Mauszeiger über der zugehörigen ViewComponent steht. Nun kann der Endpunkt samt Relation durch Drücken der linken Maustaste aufgenommen und über den Bildschirm verschoben werden. Läßt der Benutzer die Maustaste über dem BorderAnchor einer Klasse los, so wird die Relation entsprechend verändert. Wird die Relation bzw. der Verbinderendpunkt über freien Raum 'abgelegt', schnellt der Endpunkt an seinen Ausgangspunkt zurück - es ändert sich nichts.

Erweiterungen Event Modell

Der BorderAnchor ruft letztlich die entscheidenden Methoden auf, um Relationen zu verschieben. Damit ein BorderAnchor weiß, wann er in Aktion treten soll und mit wem, ist ein Event-Mechanismus von Nöten. Leider ist es nicht möglich (vielleicht schon, aber dann nur mit erheblichem Aufwand) den MouseEvent durch die ViewComponent des Verbinderendpunktes (oder Qualifiers) durchzureichen. Daher wurde eine eigene Event-Verwaltung erforderlich.

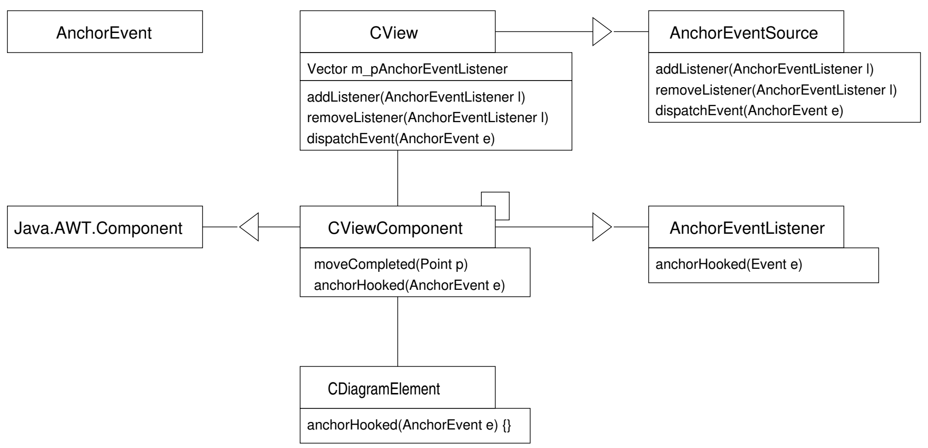


Abbildung 4.37: Klassen Diagramm Event Modell

Klassendiagramm:

Die Klasse CView ist eine 'Event-Quelle' und somit für die Verteilung der AnchorEvents verantwortlich (siehe auch Abbildung 4.37). Dazu implementiert sie die abstrakten Methoden der Klasse AnchorEventSource. Üblich sind die Methoden add/removeListener und dispatchEvent.

CViewComponent hingegen ist abgeleitet von der Klasse AnchorEventListener; verarbeitet somit neben den schon vorhandenen ActionEvents die neuen AnchorEvents. Eingehende Events können so an die zugehörigen Diagrammelemente weitergeleitet werden. Dort werden sie über die Methode anchorHooked aufgefangen.

Die Methode createViewComponent der Klasse CDEBorderAnchor trägt automatisch einen Listener in den Vector m_pAnchorEventListner der Klasse CView ein. Folglich werden AnchorEvents nur an Diagrammelemente vom Typ BorderAnchor geschickt.

4.1.5.2 Textboxen

Einleitung

Textboxen dienen dazu, im Diagramm verschiedene Arten von Texten darstellen zu können. So z.B. Beschriftungen von Assoziationen, Anzeigen von Kardinalitäten und Rollennamen und Kommentare.

Neue und veränderte Klassen

Um die Textboxen realisieren zu können, mußten einige neue Klassen erzeugt werden und schon existierenden Klassen wie CLogicalElement und Unterklassen wurden erweitert (zur Veranschaulichung siehe auch Abbildung 4.38).

CDETextBox

CDETextBox ist die Oberklasse für alle Textboxen. Hier wird die Breite der Textboxen berechnet.

CDEAssociationTextBox

Die Association-Textbox hat als Aufgabe, den Namen einer Assoziation anzuzeigen. Sie plaziert sich in der Mitte des ihr zugewiesenen Diagram-Elements. Es ist aber möglich die Textbox mittels der setOffset(int x, int y)-Methode entsprechend um "x,y" verschoben zu positionieren.

CDEAnchoredTextBox

Die Anchored-TextBox ist hauptsächlich zum Anzeigen von Kardinalitäten und Rollennamen zuständig. Ihr wird ein AbstractAnchor übergeben, an den sie sich dann ausrichtet.

CCmdAddAnchoredTextBox

CCmdAddAnchoredTextBox ist das Kommando zum Erzeugen von Anchored-Textboxen.

CCmdAddAssocTextBox

CCmdAddAssocTextBox ist das Kommando zum Erzeugen von Association-Textboxen.

CLogicalElement

Hier wurde noch die Methode getText() hinzugefügt, die einen String zurückliefert. Das Ergebnis dieser Methode wird den Textboxen als anzuzeigender Text übergeben.

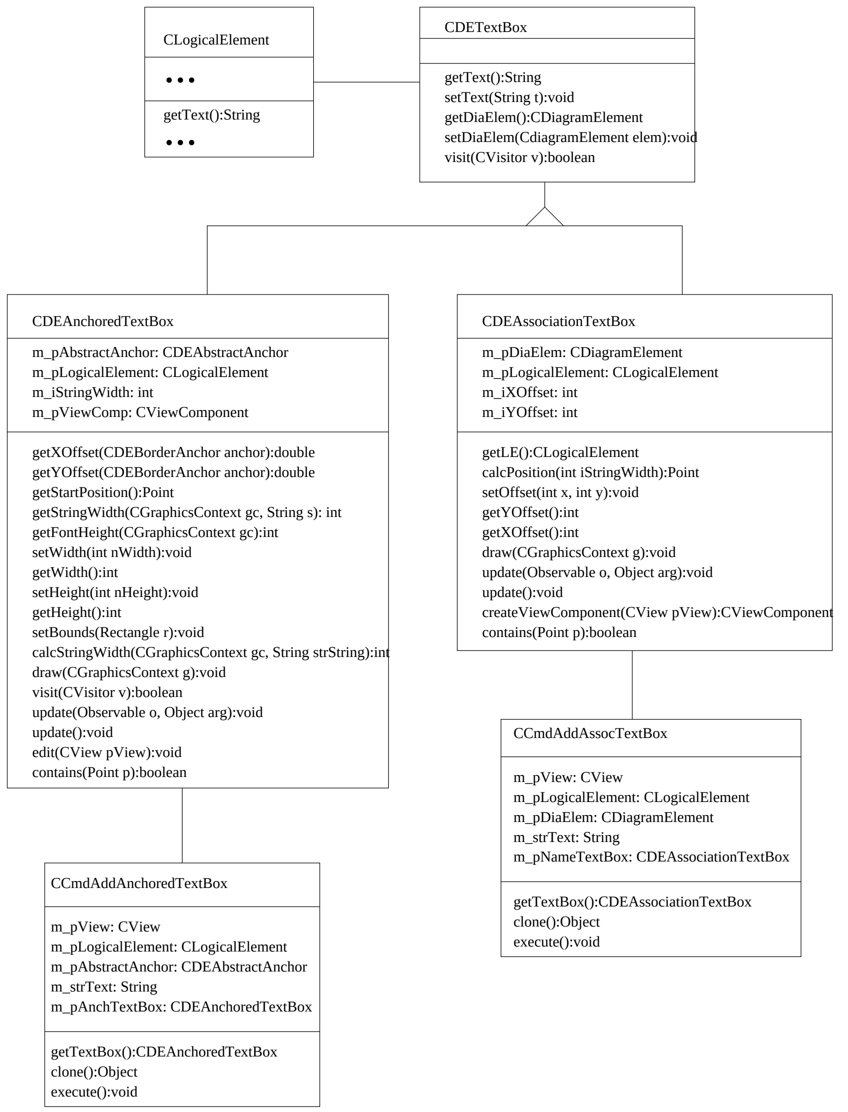


Abbildung 4.38:

4.1.5.3 Rechtwinklige Verbinder

Vorwort Der vorliegende Text ist eine Ergänzung zum Entwurf des Diagramm-Editors deSIGN. Es wird ein Konzept zur Realisiation von Verbindern entwickelt, die folgende Anforderungen erfüllen:

- Verbindung zweier Diagramm-Elemente, auch unterschiedlichen Typs
- benutzerdefinierte Kontrollpunkte bestimmen den Linienverlauf
- alle Teilstücke einer Linie verlaufen entweder waagerecht oder senkrecht

Zur Illustration des Konzepts werden im Folgenden verschiedene Diagrammarten verwendet.

Um eine intuitive Vorstellung zu bekommen, werden Bildausschnitte skizziert. Dabei wird ein Beispiel so visualisiert, wie es auch im Diagrammeditor oder in der Druckausgabe prinzipiell beabsichtigt ist. Es werden jedoch auch Metainformationen direkt in die Darstellung aufgenommen. D.h. die Bildausschnitte sind kein reines Beispiel, sondern mit Kommentaren, Beschriftungen und verdeutlichenden Symbolen angereichert. In der Bildausschnitt- Darstellung wird über das Layout bzw. über Diagramme geredet, wobei eine Metasprache verwendet wird, die nicht formell definiert ist. Die Bedeutung muß aus dem Kontext geschlossen werden - was hoffentlich gelingt.

Ferner werden Instanzdiagramme verwendet, um eine Beispielsituation durch Angabe der in einem möglichen Systemzustand vorhandenen Objekte und ihrer Beziehungen zu verdeutlichen. Hierbei sind nur Instanzbeziehungen, d.h. Assoziationen und Aggregationen, sichtbar. Klassenbeziehungen, d.h. Generalisierungen, sind dagegen den Klassendiagrammen zu entnehmen.

Die Systemstruktur wird mit Hilfe von Klassendiagrammen dargestellt. Die Notation von Instanz- und Klassendiagrammen ist an OMT angelehnt.

Begriffsdefinition

Verbinder Verbinder sind sichtbare Linien im Diagramm, die zwei Diagrammelemente miteinander verknüpfen. Alle Verbinder sind von der Oberklasse CDEBinaryLink abgeleitet. Konkrete Unterklassen bzw. konkrete Verbinderklassen sind:

- CDEBinaryAssociation (siehe Abb. 4.39)
- CDENAryParticipant (siehe Abb. 4.40)
- CDEAggregation (siehe Abb. 4.41)
- CDEGeneralization (siehe Abb. 4.42)

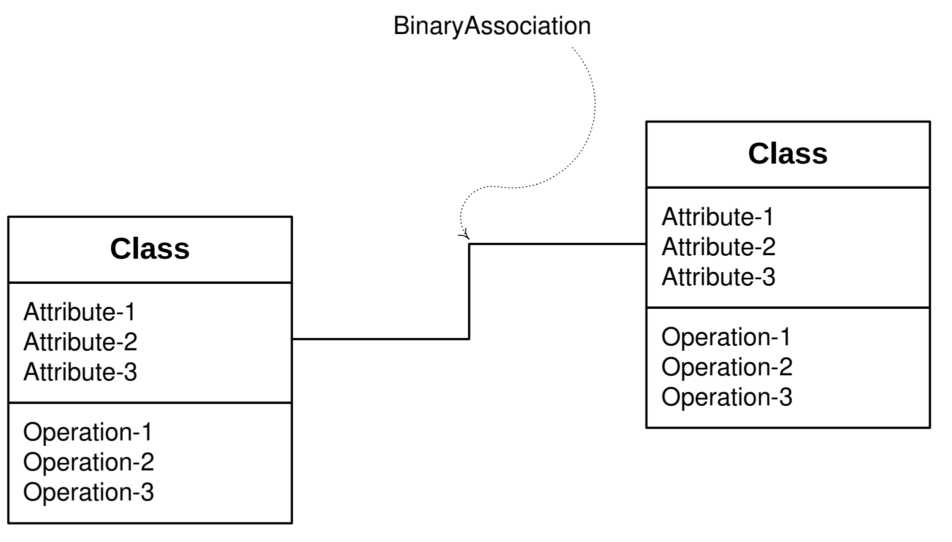


Abbildung 4.39: Bildausschnitt Binäre Assoziation

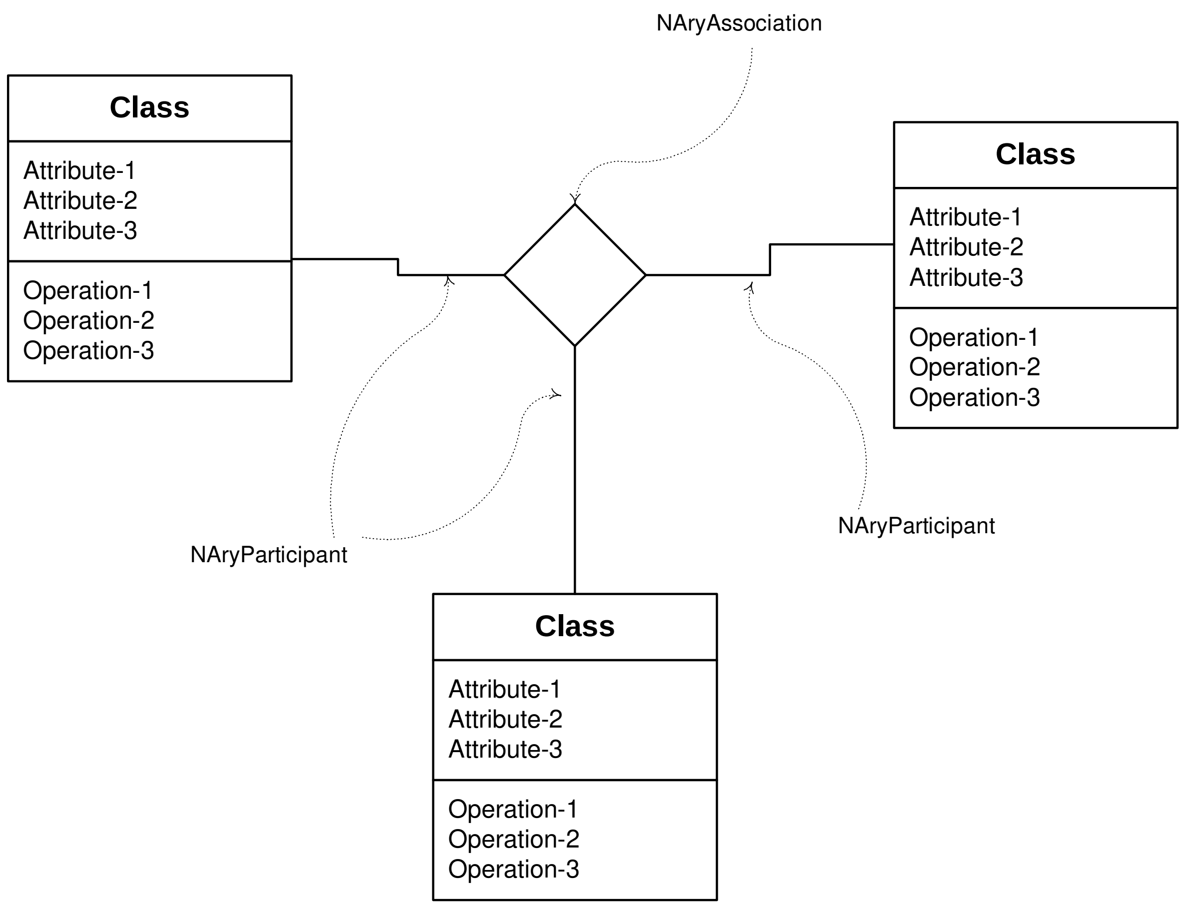


Abbildung 4.40: Bildausschnitt N-äre Assoziation

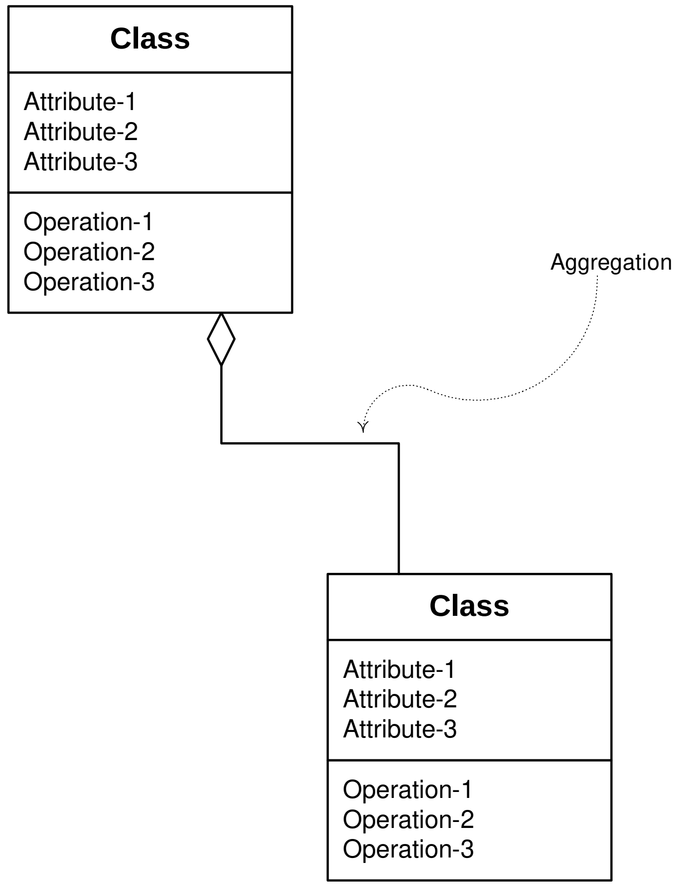


Abbildung 4.41: Bildausschnitt Aggregation

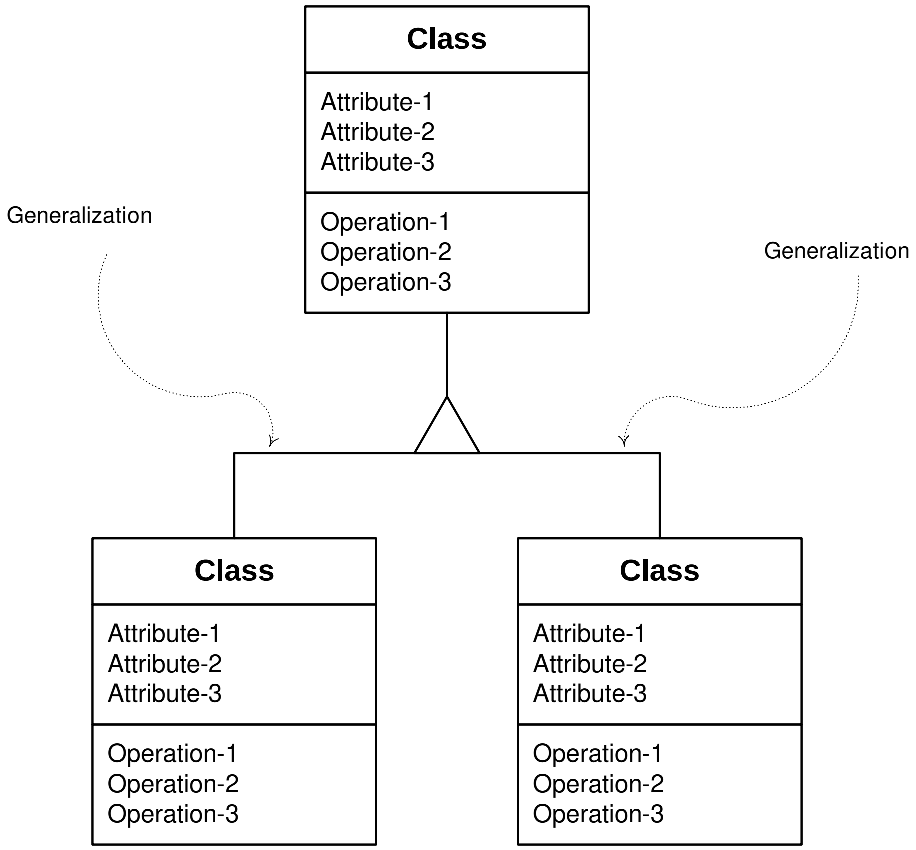


Abbildung 4.42: Bildausschnitt Generalisierung

Verbinderendpunkt Ein Verbinder hat genau zwei Verbinderendpunkte, die als Start- und Zielpunkt bezeichnet werden (Source, Sink). Bei manchen Verbindern ist die Richtung insofern relevant, als die Endpunkte verschiedene Eigenschaften haben. Bei Aggregationen wird der Startpunkt mit einem kleinen Diamantsymbol versehen. Bei Generalisierungen befindet sich zwischen den Endpunkten ein weiterer Punkt, der durch ein Dreieck dargestellt wird, dessen Spitze in Richtung der Oberklasse bzw. in Richtung des Startpunkt weist. Verbinderendpunkte haben nicht immer ein eigenes sichtbares Symbol. Sie sind nur dann sichtbar, wenn der Benutzer Mausaktionen darauf auslöst. Ein Verbinderendpunkt kann vom Benutzer durch Ziehen mit der Maus bei gedrückter linker Maustaste verschoben werden.

verankerbares Element Verbinderendpunkte sind verankerbare Elemente. Verankerbare Elemente können an Ankerpunkten befestigt werden. Sie implementieren hierzu das Interface CDEAnchoredElement.

Hybride Verbinder Ein Sonderfall ist das Diagrammelement CDEQualifier. Ein Qualifier- Diagrammelement kann an einen Ankerpunkt eines Klassensymbols angehängt werden. Hierzu besitzt es einen Verbinderendpunkt. Es weicht jedoch von dem Schema der Verbinder insofern ab, als es anstelle des zweiten Verbinderendpunktes einen Ankerpunkt zur Verfügung stellt, an dem ein Verbinder befestigt ist.

Ankerpunkt Verbinderendpunkte und andere verankerbare Elemente können an Ankerpunkte angehängt werden. Ankerpunkte fixieren somit Endpunkte von Verbindern. Diagrammelemente, die mit anderen Diagrammelementen mittels eines Verbinders verknüpft werden können, stellen einen oder mehrere Ankerpunkte zur Verfügung. Ankerpunkte werden durch Unterklassen von CDEAbstractAnchor realisiert. (siehe Abb. 4.44)

Kontrollpunkt Ein Kontrollpunkt markiert einen sichtbaren Punkt auf einem Verbinder, den der Benutzer mittels Mausaktionen verschieben kann. Kontroll- und Ankerpunkte sind nur sichtbar, wenn das Objekt (Diagramm-Element) selektiert ist bzw. Mausereignisse (MouseMove etc.) empfängt. Für Kontrollpunkte wird eine neue Klasse CDEControlPoint eingeführt. Jeder Kontrollpunkt kontrolliert die Position von zwei Zwischenpunkten, die an den Endpunkten eines Liniensegments liegen. Ein Kontrollpunkt ist stets nur in eine Richtung in der x,y-Ebene verschiebbar, d.h. er hat nur einen Freiheitsgrad. (siehe Abb. 4.43 und 4.44)

Zwischenpunkt Zwischenpunkte bestimmen den Linienverlauf eines Verbinders. Die Zwischenpunkte sind im Diagramm nicht gesondert dargestellt. Sie sind als Eckpunkte der Liniensegmente zu erkennen, haben aber kein Symbol und dienen nicht der Benutzerinteraktion. In den folgenden Abbildungen werden sie dennoch zur Verdeutlichung durch einen Kreis symbolisiert. Zwischenpunkte werden automatisch berechnet aus der Position der Anker- und Kontrollpunkte.

Liniensegment Eine Linie, die einen Verbinder symbolisiert, setzt sich aus mehreren Segmenten zusammen. Der Linienverlauf ist durch die Punktfolge $p_0 \dots p_n$ mit p_0 als Startpunkt und p_n als Endpunkt gegeben. Falls Zwischenpunkte vorhanden sind, ist $n > 1$ und $p_1 \dots p_{n-1}$ sind die Zwischenpunkte. Je zwei aufeinanderfolgende Punkte $p_i, p_{i+1} \quad i \geq 0$ ergeben ein Liniensegment. Die Liniensegmente sollen entweder horizontal (waagerecht) oder vertikal (senkrecht) verlaufen. In einer Verbinderlinie treten alternierend horizontale und vertikale Segmente auf.

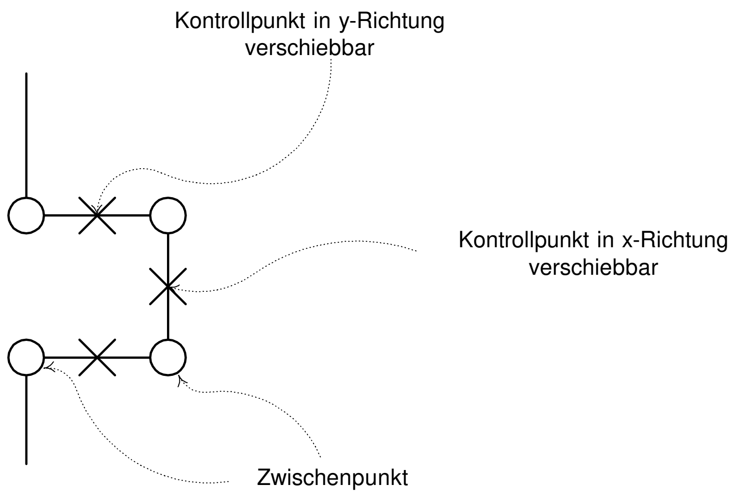


Abbildung 4.43: Kontrollpunkte und Zwischenpunkte

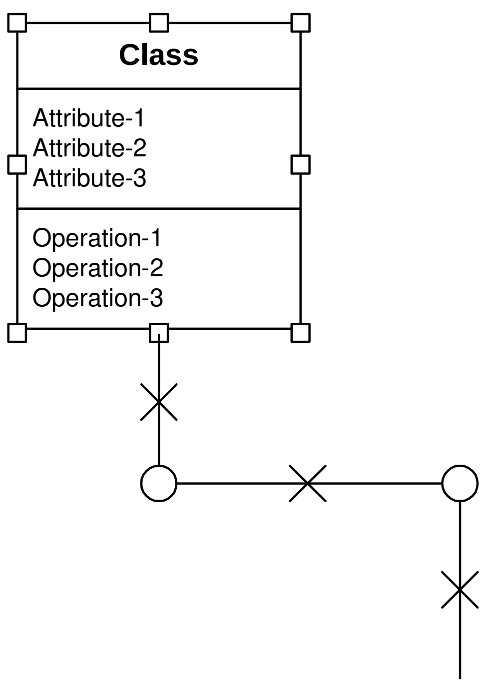


Abbildung 4.44: Ankerpunkte

Berechnung des Linienverlaufs

Eigenschaften von Kontrollpunkten Die Richtung des Liniensegments an einem Ankerpunkt ist durch die Position des Ankerpunktes gegeben. Jedes Liniensegment hat einen Kontrollpunkt. Der Kontrollpunkt kann vom Benutzer horizontal verschoben werden, wenn das Segment vertikal verläuft. Wenn die Richtung des Segments horizontal ist, ist nur ein Verschieben in vertikaler Richtung erlaubt. Das Start- und das Endsegment einer Linie kann nicht verschoben werden, da es durch den Ankerpunkt bzw. den Verbinderendpunkt fixiert ist.

In dem Bildausschnitt Abb. 4.45 wird eine solche Situation gezeigt. Die beteiligten Objekte und ihre Verbindungen sind in dem Instanzdiagramm Abb. 4.46 gezeigt.

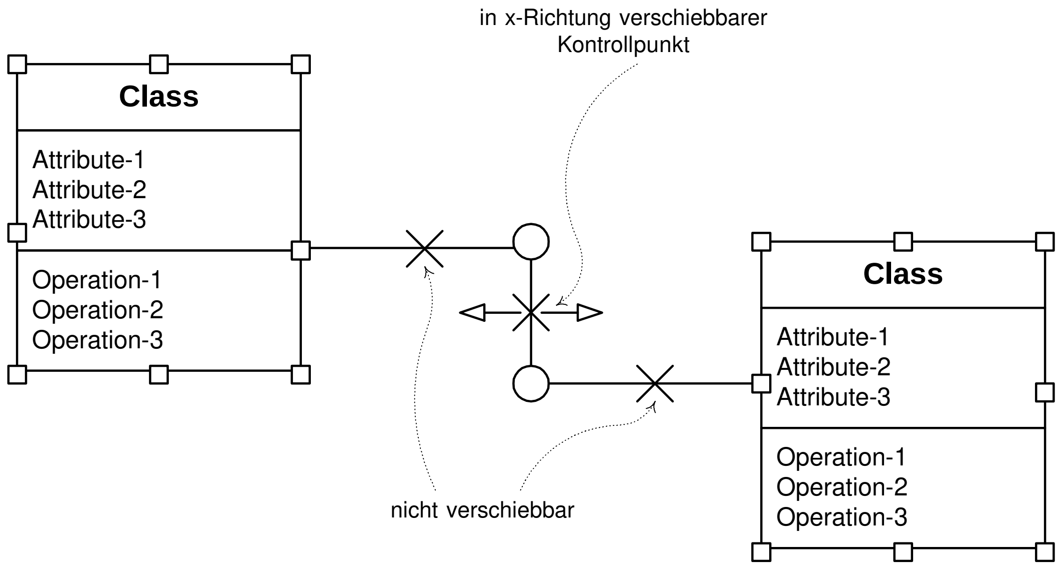


Abbildung 4.45: Bildausschnitt Linienverlauf Binäre Assoziation

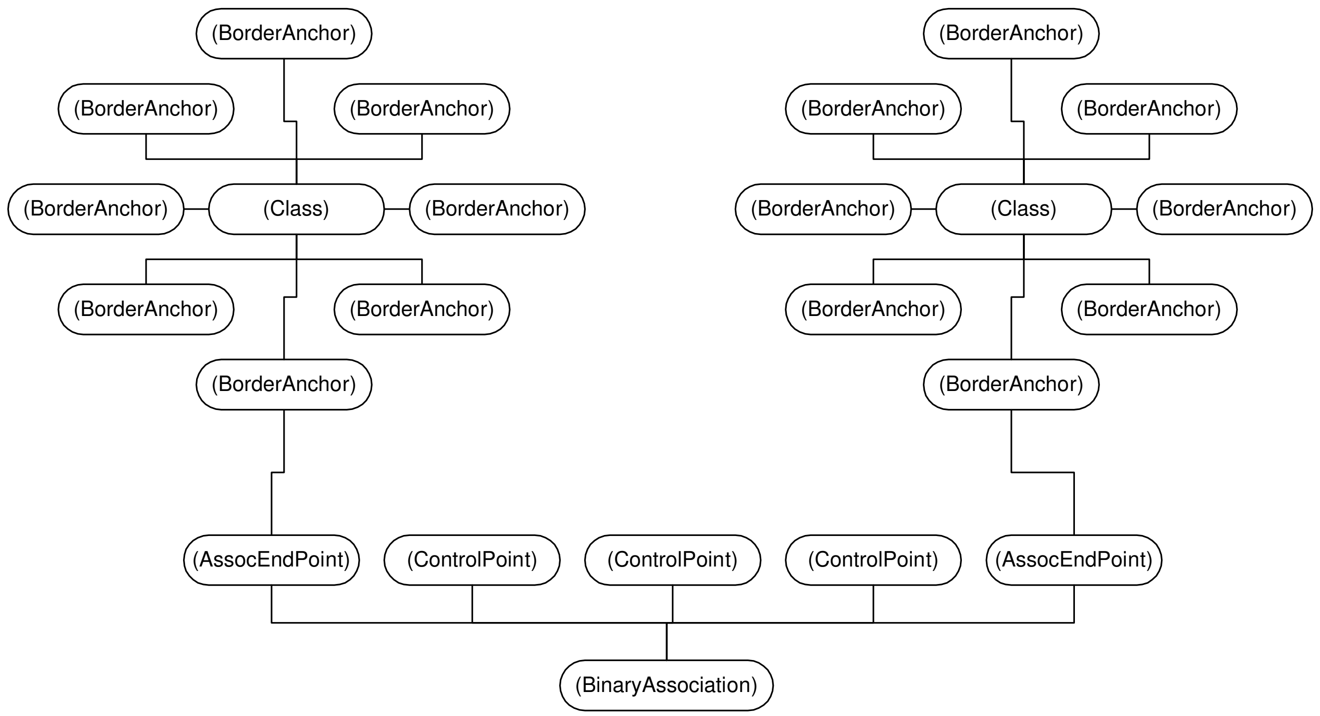


Abbildung 4.46: Instanzdiagramm Linienverlauf Binäre Assoziation

Die Liniensegemente werden als Geradenabschnitte beschrieben. Eine Gerade wird durch Angabe eines Bezugspunktes und einer Richtung identifiziert. Zusätzlich werden zwei Längenangaben benötigt, die den Abschnitt auf der Geraden begrenzen.

Die Richtung der Geraden ist stets senkrecht zur Bewegungsrichtung des Kontrollpunktes. Ein Verschieben des Kontrollpunktes bewirkt eine Parallelverschiebung der Geraden. Ein Kontrollpunkt hat die folgenden Eigenschaften:

$$\begin{aligned}
CONTROLPOINT &: \text{Menge der Kontrollpunkte} \\
movedirection &: CONTROLPOINT \to DIRECTION \\
distance &: CONTROLPOINT \to INT \\
move &: CONTROLPOINT \times INT \to CONTROLPOINT \\
line &: CONTROLPOINT \to LINE \\
linedirection &: CONTROLPOINT \to DIRECTION \\
distance(move(c,x)) &= distance(c) + x
\end{aligned}$$

Die Bewegungsrichtung des Kontrollpunktes, die mit der Funktion *movedirection* ermittelt werden kann, wird durch einen normalisierten Vektor in der Ebene angegeben. Wenn nur horizontale und vertikale Bewegungen erlaubt sein sollen, ist für ein $c \in CONTROLPOINT$

$$movedirection(c) \in \{(1,0),(0,1),(-1,0),(0,-1)\}$$

$$\begin{aligned}
DIRECTION &: \text{Menge der normalisierten Richtungsvektoren in der Ebene} \\
& DIRECTION \subset VINT2 \\
POINT &: \text{Menge der Punkte in der Ebene} \\
& POINT \subset VINT2 \\
VINT2 &: \text{Menge zweidimensionaler Vektoren} \\
& VINT2 \subseteq INT \times INT \\
createdirection &: VINT2 \to DIRECTION \\
createpoint &: VINT2 \to POINT \\
createvec &: INT \times INT \to VINT2 \\
x, y &: VINT2 \to INT \\
\cdot &: VINT2 \times INT \to VINT2 \\
\cdot &: INT \times VINT2 \to VINT2 \\
\cdot &: VINT2 \times VINT2 \to INT \\
x(createpoint(a,b)) &= a \\
y(createpoint(a,b)) &= b \\
x(createdirection(a,b)) &= a \\
y(createdirection(a,b)) &= b \\
v \cdot a &= createvec(x(v) \cdot a, y(v) \cdot a) \\
a \cdot v &= createvec(x(v) \cdot a, y(v) \cdot a) \\
v_1 \cdot v_2 &= x(v_1) \cdot x(v_2) + y(v_1) \cdot y(v_2)
\end{aligned}$$

Mit Hilfe des Skalarprodukts kann eine Funktion *movePosition* definiert werden, die Bewegungen der Mausposition in zulässige Bewegungen des Kontrollpunkts umsetzt.

$$\begin{aligned} movePosition &: CONTROLPOINT \times POINT \to CONTROLPOINT \\ movePosition(c,p) &= move(c, movedirection(c) \cdot p) \end{aligned}$$

Der Kontrollpunkt erlaubt das Berechnen einer Geraden mit der Funktion *line*. Eine Gerade hat die folgenden Eigenschaften.

$$\begin{aligned} LINE &: \text{Menge der Geraden in der Ebene} \\ p_0 &: LINE \to POINT \\ direction &: LINE \to DIRECTION \\ createline &: POINT \times DIRECTION \to LINE \end{aligned}$$

Für die Berechnung der Geraden zu einem Kontrollpunkt $c \in CONTROLPOINT$ gilt:

$$\begin{aligned} line(c) = createline(&createpoint(movedirection(c) \cdot distance(c)), \\ &linedirection(c)) \end{aligned}$$

Die Richtung der Geraden $linedirection(c)$ ist senkrecht zur Bewegungsrichtung des Kontrollpunktes. Mit der kanonischen Einbettung des R^2 in den R^3 gilt $(x,y) = (x,y,0)$ und $(x,y,0) \times (0,0,1) = (y,-x,0)$ ist orthogonal hierzu. Da in unserem Koordinatensystem die y-Achse nach unten zeigt, wählen wir die umgekehrte Richtung.

$$\begin{aligned} &linedirection(c) = \\ &createdirection(createvec(y(movedirection(c)), x(movedirection(c)))) \end{aligned}$$

Das Attribut *distance* eines Kontrollpunkts c kann als Länge der Verschiebung der Geraden entlang der Bewegungsrichtung aufgefaßt werden.
$distance(c) = 0$ bedeutet, dass der Bezugspunkt $p_0(line(c))$ der Geraden der Punkt $(0,0)$ bzw. $createpoint(createvec(0,0))$ ist.

Jedem Kontrollpunkt $c_i \in CONTROLPOINT$ wird somit eine Gerade zugeordnet:

$$\begin{aligned} g_i &= p_{0,i} + u_i \cdot v_i \\ &p_{0,i}, v_i \in R^2,\ u_i \in R \\ &p_{0,i} = movedirection(c_i) \cdot distance(c_i) \\ &v_i = linedirection(c_i) \end{aligned}$$

Der Bezugspunkt $p_{0,i}$ ergibt sich aus der Verschiebung *distance* des Kontrollpunktes und seiner Bewegungsrichtung *movedirection*. Diese Situation wird in Abb. 4.47 und 4.48 veranschaulicht.

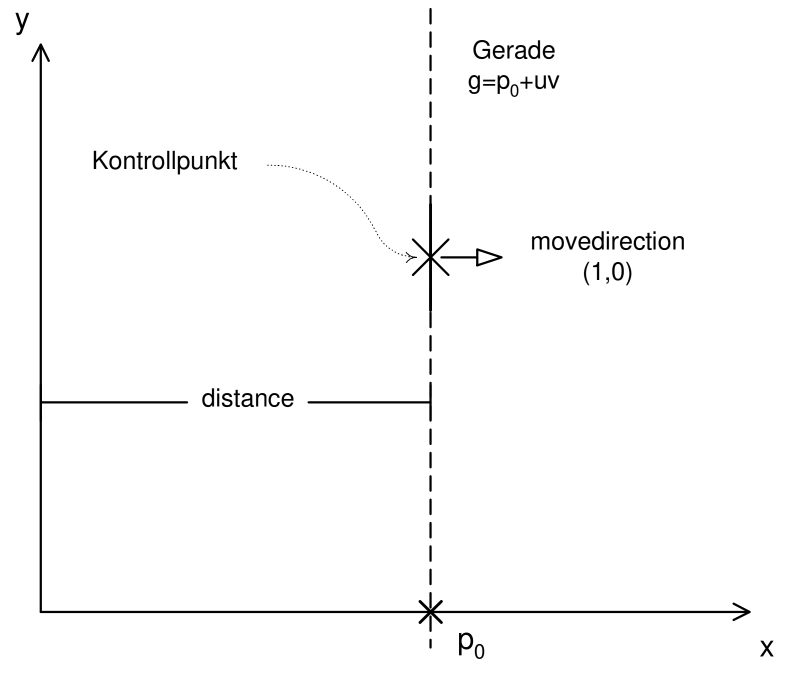


Abbildung 4.47: Kontrollpunkt und zugeordnete Gerade

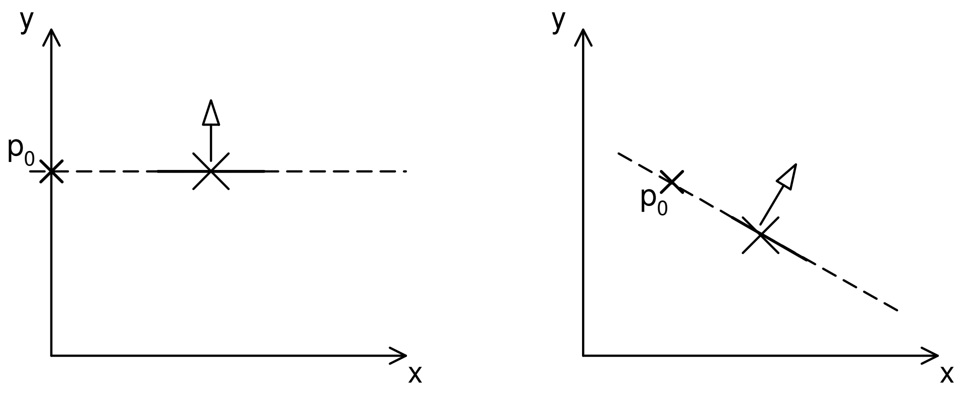


Abbildung 4.48: verschiedene Bezugspunkte

Berechnung der Zwischenpunkte Jedes Liniensegment ist ein Geradenabschnitt und wird durch zwei Punkte auf einer Geraden beschrieben, die den Abschnitt begrenzen. Die begrenzenden Punkte werden als Zwischenpunkte bezeichnet. Ein Liniensegment hat die folgenden Eigenschaften:

$$\begin{aligned}
LINESEG &: \text{Menge der Geradenabschnitte} \\
source &: LINESEG \to POINT \\
target &: LINESEG \to POINT \\
createlineseg &: POINT \times POINT \to LINESEG \\
source(createlineseg(s,t)) &= s \\
target(createlineseg(s,t)) &= t
\end{aligned}$$

Für eine Folge von Kontrollpunkten $C =< c_0, c_1, \ldots c_n >$ erhält man die Zwischenpunkte als $\{p \mid \exists\, i : c_i \in C \wedge (p = source(c_i) \vee p = target(c_i))\}$ Ferner gilt $source(c_{i+1}) = target(c_i)$ und $g_{i+1} = g_i$ hat eine Lösung, die den gesuchten Schnittpunkt bzw. Zwischenpunkt beschreibt.

Sind g_{i+1} und g_i nicht parallel, so gibt es Parameter $u_{i+1,s}$ und $u_{i,s}$ mit

$$\begin{aligned}
g_{i+1} = g_i \iff p_{0,i+1} + u_{i+1,s} v_{i+1} &= p_{0,i} + u_{i,s} v_i \\
\iff u_{i+1,s} v_{i+1} - u_{i,s} v_i &= p_{0,i} - p_{0,i+1}
\end{aligned}$$

In Komponentendarstellung erhält man ein Gleichungssystem mit zwei Unbekannten $u_{i+1,s}, u_{i,s}$ und zwei Gleichungen:

$$\begin{aligned}
u_{i+1,s} v_{i+1,x} - u_{i,s} v_{i,x} &= p_{0,i,x} - p_{0,i+1,x} \\
u_{i+1,s} v_{i+1,y} - u_{i,s} v_{i,y} &= p_{0,i,y} - p_{0,i+1,y}
\end{aligned}$$

Auflösen der ersten Gleichung nach $u_{i+1,s}$, Einsetzen des Ergebnisses in die zweite Gleichung und Auflösen nach $u_{i,s}$ ergibt

$$\begin{aligned}
u_{i+1,s} &= \frac{p_{0,i,x} - p_{0,i+1,x} + u_{i,s} v_{i,y}}{v_{i+1,x}} \\
u_{i,s} &= \frac{(p_{0,i,y} - p_{0,i+1,y}) v_{i+1,x} - (p_{0,i,x} - p_{0,i+1,x}) v_{i+1,y}}{v_{i,x} v_{i+1,y} - v_{i,y} v_{i+1,x}}
\end{aligned}$$

Die Koordinaten des Schnittpunkts erhält man dann mit

$$\begin{aligned}
x_s &= p_{0,i,x} + u_{i,s} v_{i,x} \\
&= \frac{v_{i+1,y} v_{i,x} p_{0,i+1,x} - v_{i,y} v_{i+1,x} p_{0,i,x} + (p_{0,i,y} - p_{0,i+1,y}) v_{i+1,x} v_{i,x}}{v_{i,x} v_{i+1,y} - v_{i,y} v_{i+1,x}} \\
y_s &= p_{0,i,y} + u_{i,s} v_{i,y} \\
&= \frac{v_{i+1,y} v_{i,x} p_{0,i,y} - v_{i,y} v_{i+1,x} p_{0,i+1,y} - (p_{0,i,x} - p_{0,i+1,x}) v_{i+1,y} v_{i,y}}{v_{i,x} v_{i+1,y} - v_{i,y} v_{i+1,x}}
\end{aligned}$$

Dies ist ein Algorithmus für die Funktion

$$intersect : LINE \times LINE \to POINT$$

Damit läßt sich ein Verbinder wie folgt beschreiben.

$$
\begin{aligned}
CONNECTOR &: \text{Menge der Verbinder}\\
source &: CONNECTOR \to ENDPOINT\\
sink &: CONNECTOR \to ENDPOINT\\
controlpoint &: CONNECTOR \times INT \to CONTROLPOINT\\
addpoint &: CONNECTOR \times CONTROLPOINT \to CONNECTOR\\
removepoint &: CONNECTOR \times CONTROLPOINT \to CONNECTOR\\
\sharp points &: CONNECTOR \to INT\\
createconnector &: ENDPOINT \times ENDPOINT \to CONNECTOR\\
lineseg &: CONNECTOR \times INT \to LINESEG
\end{aligned}
$$

$$
\begin{aligned}
&source(createconnector(s,t)) = s\\
&sink(createconnector(s,t)) = t\\
&removepoint(addpoint(l,c),c) = l\\
&\sharp points(createconnector(s,t)) = 2\\
&\sharp points(addpoint(l,c)) = \sharp points(l) + 1\\
&controlpoint(l,0) = s\\
&controlpoint(addpoint(createconnector(s,t),c),1) = c\\
&controlpoint(l,\sharp points(l)-1) = t\\
&lineseg(l,0) = createlineseg(position(source(l)),\\
&\qquad intersect(line(controlpoint(l,0)), line(controlpoint(l,1))))\\
&lineseg(l,i+1) = createlineseg(target(lineseg(l,i)),\\
&\qquad intersect(line(controlpoint(l,i)), line(controlpoint(l,i+1))))\\
&lineseg(l,\sharp points(l)-1) = createlineseg(target(lineseg(l,\sharp points(l)-2)),\\
&\qquad position(sink(l))))\\
&target(lineseg(l,i)) = source(lineseg(l,i+1))
\end{aligned}
$$

$$
\begin{aligned}
ENDPOINT &: \text{Menge der Verbinderendpunkte}\\
&ENDPOINT \subset CONTROLPOINT\\
anchor &: ENDPOINT \to ANCHOR\\
movedirection &: ENDPOINT \to POINT\\
position &: ENDPOINT \to POINT\\
movedirection(e) &= createdirection(createvec(0,0))\\
linedirection(e) &= direction(anchor(e))\\
distance(e) &= position(e) \cdot movedirection(e)\\
position(e) &= position(anchor(e))
\end{aligned}
$$

Die Verbinderendpunkte sind gleichzeitig auch Kontrollpunkte. Ein Verbinder hat immer einen Start- und einen End-Verbinderendpunkt. Somit hat jeder Verbinder auch mindestens zwei Kontrollpunkte. Die Verbinderendpunkte sind verankert, somit auch nicht beweglich. Daher liefert die Funktion *movedirection* für den Start- und Endkontrollpunkt stets $(0, 0)$ ab.

Die Funktion *position* liefert für einen Verbinderendpunkt die Position im Koordinatensystem des Diagramms bzw. Sheets, an der der Punkt verankert ist. D.h. die Position des Ankerpunktes bezüglich des Ursprungs der Diagramm-Seite.

Der Linienverlauf für einen Verbinder ist dann gegeben durch eine Folge von Liniensegmenten bzw. Geradenabschnitten, die mittels der Funktion *lineseg* ermittelt werden.

Die Kontrollpunkte haben keine feste Position. Nur die Verschiebung des Kontrollpunktes in Bewegungsrichtung *distance* ist stets bekannt. Zu einer vollständigen Positionsbestimmung fehlt dann noch eine weitere Koordinate. Die vollständige Position kann aber als Mittelpunkt des zugehörigen Liniensegments festgelegt werden.

Benutzungsfälle

Erstellen von Verbindern Verschiedene Arten von Verbindern können vom Benutzer interaktiv erstellt werden. Der Ablauf besteht immer im Markieren zweier Ankerpunkte und anschliessendem Auslösen eines Kommandos. Zur Zeit sind folgende Kommandos implementiert:

- CCmdLink: Erstellen von CDEBinaryAssociation oder CDENAryParticipant
- CCmdAgg: Erstellen von CDEAggregation
- CCmdGen: Erstellen von CDEGeneralization

Zur Verwendung der nach diesem Konzept erweiterten Verbinder sollte ein neuer Satz von Kommandos implementiert und dem Benutzer (evtl. über zusätzliche Menüeinträge und Buttons) angeboten werden. Die bisherige Funktionalität sollte weiter erhalten bleiben.

Die neuen Kommandos sind aufwendiger zu gestalten, da sie kontextabhängig mehrere Kontrollpunkte automatisch erzeugen sollen. Die Bewegungsrichtung der Kontrollpunkte eines Verbinders und damit auch die Richtung der Liniensegmente muß stets alternierend in x- und y-Richtung sein. Die Richtung des ersten und letzten Liniensegementes ist durch die Position der Ankerpunkte vorgegeben. Jeder Ankerpunkt soll die aufgrund seiner Position vorgegebene Linienrichtung angeben können, so daß die Funktion *linedirection* für Verbinderendpunkte (Verbinderendpunkte sind Kontrollpunkte...) als

$$linedirection(e) = direction(anchor(e))$$

definiert werden kann. Die möglichen Rückgabewerte sind in Abb. 4.49 gezeigt.

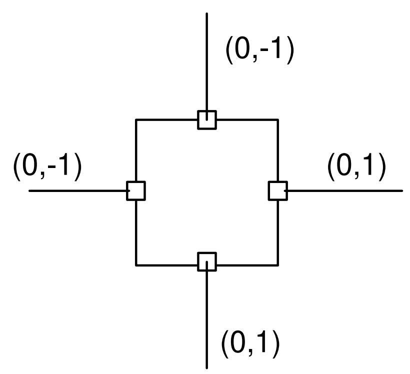


Abbildung 4.49: Richtung von Linien an Ankerpunkten

Bisher wurden nur acht Ankerpunkte für das Diagrammelement CDEClass bzw. für Klassensymbole erzeugt. Die Ankerpunkte auf den Ecken des Symbols sind dann je zwei der vier Seiten des Symbols zuzurechnen. Eine Richtung kann dann nicht eindeutig bestimmt werden. Um dieses Problem zu umgehen, können für jede Ecke des Symbols zukünftig zwei Ankerpunkte erzeugt werden, wie in Abb. 4.50 illustriert.

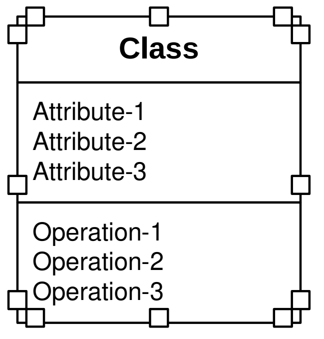


Abbildung 4.50: Vermeidung von Mehrdeutigkeit bei der Richtungszuordnung

In Abb. 4.51 sind verschiedene Beispielsituationen bei der Erzeugung von Verbindern gezeigt. Wegen der alternierenden Richtung der Liniensegmente müssen insbesondere die Richtungen von Start- und End-Ankerpunkt im Hinblick auf die Zahl und Position der zu erzeugenden Kontrollpunkte berücksichtigt werden.

Vermutlich reicht es, die folgenden Fälle zu unterscheiden:

- Richtung gleich, Vorzeichen gleich: 3 Kontrollpunkte hinzufügen
- Richtung gleich, Vorzeichen ungleich: 1 Kontrollpunkt hinzufügen
- Richtung ungleich, Vorzeichen gleich: 2 Kontrollpunkte hinzufügen
- Richtung ungleich, Vorzeichen ungleich: 2 Kontrollpunkte hinzufügen

Beim Hinzufügen von Kontrollpunkten ist die anfängliche Auslenkung *distance* bzw. die Verschiebung des Liniensegments in Bewegungsrichtung des Kontrollpunktes vorzugeben. Hierzu ist noch ein Algorithmus zu entwickeln, der einen akzeptablen anfänglichen Linienverlauf sicherstellt. Z.B. durch Einhaltung einer Mindestlänge für Liniensegmente oder sogar Vermeidung von Überschneidungen mit dem Diagrammelement zu dem Start- und End-Ankerpunkt gehören.

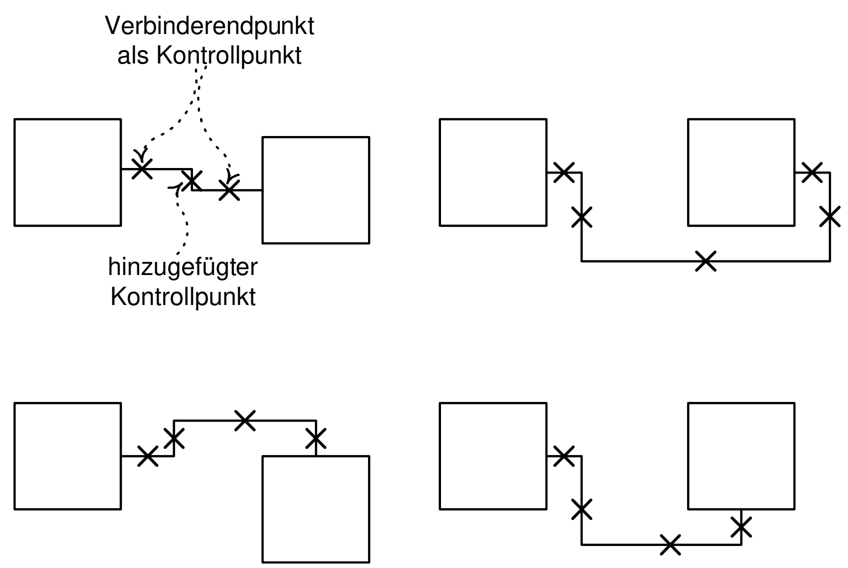


Abbildung 4.51: Beispiele für die Erzeugung von Verbindern

Hinzufügen und Löschen von Kontrollpunkten Damit die Bedingung alternierender Richtung von Liniensegmenten eingehalten wird, kann man Kontrollpunkte nur paarweise hinzufügen und löschen. Es muß immer mindestens ein Kontrollpunkt zusätzlich zu Start- und Endpunkt vorhanden sein.

Der Benutzungsablauf könnte so aussehen, daß zunächst ein Kontrollpunkt markiert wird und anschliessend das Kommando Kontrollpunkt hinzufügen bzw. löschen ausgelöst wird. Das Kommando fügt dann je einen Kontrollpunkt vor und einen hinter dem markierten Kontrollpunkt ein bzw. löscht die entsprechenden Kontrollpunkte. Hierzu wären dann weitere Funktionen nützlich:

$$\begin{aligned}
\mathit{addbefore} &: \mathit{CONNECTOR} \times \mathit{CONTROLPOINT} \\
&\quad \times \mathit{CONTROLPOINT} \to \mathit{CONNECTOR} \\
\mathit{addafter} &: \mathit{CONNECTOR} \times \mathit{CONTROLPOINT} \\
&\quad \times \mathit{CONTROLPOINT} \to \mathit{CONNECTOR} \\
\mathit{removebefore} &: \mathit{CONNECTOR} \times \mathit{CONTROLPOINT} \to \mathit{CONNECTOR} \\
\mathit{removeafter} &: \mathit{CONNECTOR} \times \mathit{CONTROLPOINT} \to \mathit{CONNECTOR}
\end{aligned}$$

Verschieben von Verbinderendpunkten Wenn der Endpunkt eines Verbinders an einen anderen Ankerpunkt verschoben wird, kann es notwendig sein, Kontrollpunkte hinzuzufügen. Hier gilt dasselbe wie beim Erstellen von Verbindern. Das Löschen von Kontrollpunkten sollte nicht automatisch erfolgen, sondern explizit durch den Benutzer.

Auswählen und Markieren von Verbindern Verbinder können vom Benutzer selektiert werden und sollen dann eine Markierung erhalten. Eine Reaktion auf Mausereignisse soll jedoch nur dann stattfinden, wenn die Mausposition nahe genug an einem Liniensegment ist. Dies kann dadurch erreicht werden, dass für jedes Liniensegment eine eigene ViewComponent erzeugt wird. Dann würde es sich anbieten, auch Liniensegmente durch Instanzen einer eigenen Klasse von Diagramm-Elementen zu repräsentieren.

Alternativ können die Liniensegmente aber auch immer bei Bedarf berechnet werden und nur die Kontrollpunkte werden durch Objekte repräsentiert.

Vermutlich ist ersteres leichter zu handhaben.

Entwurfsänderungen

Generalisierung konkreter Verbinderklassen Im bisherigen Entwurf V1.2 sind die Klassen DEGeneralization, DERelationship, DEBinaryAssociation, DEAggregation und DENAryParticipant direkt oder indirekt Unterklassen von DEBinaryLink.

Zu dem hier entwickelten Konzept paßt besser eine Sichtweise, in der es eine allgemeine Oberklasse für Verbinder, DEBinaryLink, gibt und zwei konkrete Unterklassen DESimpleBinaryLink und DEConnector. Die Klasse DESimpleBinaryLink implementiert dabei Verbinder in der Weise, wie sie jetzt bereits funktionieren: mit frei verschiebbaren Zwischenpunkten vom Typ DEPoint, einer rechteckigen Markierung in der Grösse der Bounding Box, etc.

Die Unterklasse DEConnector implementiert Verbinder so, wie in diesem Konzept spezifiziert.

Zwischen den Klassen DEGeneralization, DERelationship, DENAryParticipant und der (abstrakten) Klasse DEBinaryLink besteht eine ist-implementiert-mit Beziehung. Diese Beziehung wird nicht mittels Generalisierung, sondern durch eine Assoziation zur Klasse DEBinaryLink ausgedrückt.

Die Klasse DEBinaryLink sollte die Vereinigung der Schnittstellen von DE-SimpleBinaryLink und DEConnector umfassen. Operationen, die für eine der beiden Verbinderarten nicht zur Verfügung stehen, können als Default-Implementierung in der gemeinsamen Oberklasse eine Fehlermeldung produzieren. Prinzipiell sind dann beide Arten von Verbindern im Programm simultan nutzbar. Welche Art von Verbinder in einem konkreten Anwendungsfall benutzt wird, hängt nur von der Implementierung der Kommandos ab. Es könnte im Menü eine Option zur Umschaltung der benutzten Verbinderart geben. (Option - simple connector / smart connector)

Die geänderte Klassenhierarchie ist in Abb. 4.52 gezeigt.

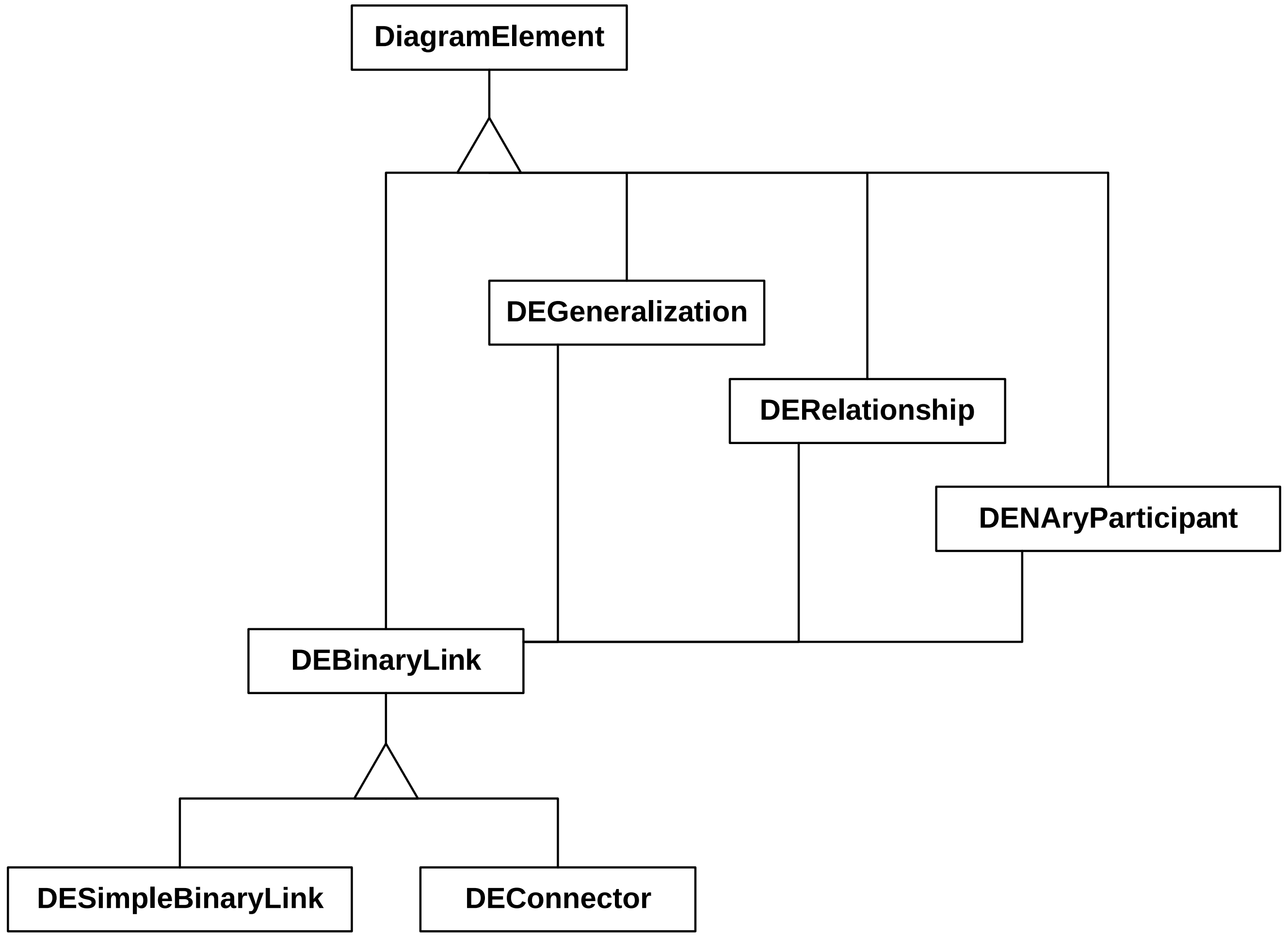


Abbildung 4.52: Klassenhierarchie Verbinder

Kontrollpunkte und Zwischenpunkte Zur Zeit gibt es eine Klasse DEPoint, die einen frei verschiebbaren Zwischenpunkt auf einem Verbinder (DEBinaryLink) darstellt. Der Zwischenpunkt kann zudem von mehreren Verbindern gleichzeitig genutzt werden, so daß die Klasse eine Assoziation zu mehreren Objekten vom Typ DEBinaryLink implementiert.

Kontrollpunkte gehören immer zu genau einem Verbinder bzw. zu einem Liniensegment des Verbinders. Sie haben ferner ein anderes Verhalten bei Benutzerinteraktion.

Die gemeinsamen Eigenschaften der beiden Klassen sind dann offenbar diejenigen, die in der gemeinsamen Oberklasse DiagramElement zusammengefasst sind (Position, Size, Interfaces Observer, Observable, Serializable, Cloneable, Visitable). Eine weitere Generalisierung scheint nicht gerechtfertigt.

Verbinderendpunkte Ein Konzept für Verbinderendpunkte ist inzwischen auch fertiggestellt. Aus dem vorliegenden Konzept geht hervor, dass Verbinderendpunkte gleichzeitig Kontrollpunkte sein sollen. D.h. die Klasse DELinkEndPoint ist Unterklasse von DEControlPoint.

Alternativ könnte man sich aber vorstellen, dass es eine Unterklasse von DEControlpoint z.B. DEEndControlPoint gibt, die eine ist-implementiert-mit Beziehung (d.h. Assoziation) zu einem Verbinderendpunkt (DELinkEndPoint) hat.

Rechtwinklige Verbinder Im Klassendiagramm Abb. 4.53 sind die genannten neuen Klassen und ihre Beziehungen gezeigt.

Damit die Berechnung der Liniensegmente bzw. der Zwischenpunkte (Schnittpunkte, s.o.) einfach möglich ist, werden die Kontrollpunkte in einer doppelt verketteten Liste angeordnet. Jeder Kontrollpunkt hat somit Zugriff auf Vorgänger (previous) und Nachfolger (next) in der Liste.

Jeder Kontrollpunkt hat ein zugeordnetes Liniensegment. Die Viewkomponente für den Kontrollpunkt kann so gewählt werden, daß sie ein Rechteck ist, welches das Liniensegment gerade umschließt. Eine Markierung des Verbinders würde dann eine Markierung aller Viewkomponenten von Liniensegmenten bedeuten. Dann umfaßt die Markierung auch nur den sensitiven Bereich, in dem Mausaktionen eine Wirkung haben.

Bestimmte Mausaktionen auf einem Liniensegment müssen über die gesamte Liste von Liniensegmenten weitergereicht werden. Z.B. soll beim Selektieren eines Segments automatisch der gesamte Verbinder makiert werden. Beim Verschieben eines Kontrollpunktes müssen die beiden begrenzenden Schnittpunkte des Liniensegments neu berechnet werden. Dann muß gegebenenfalls. auch der Nachfolger und der Vorgänger in der Liste der Kontrollpunkte sein Liniensegment aktualisieren.

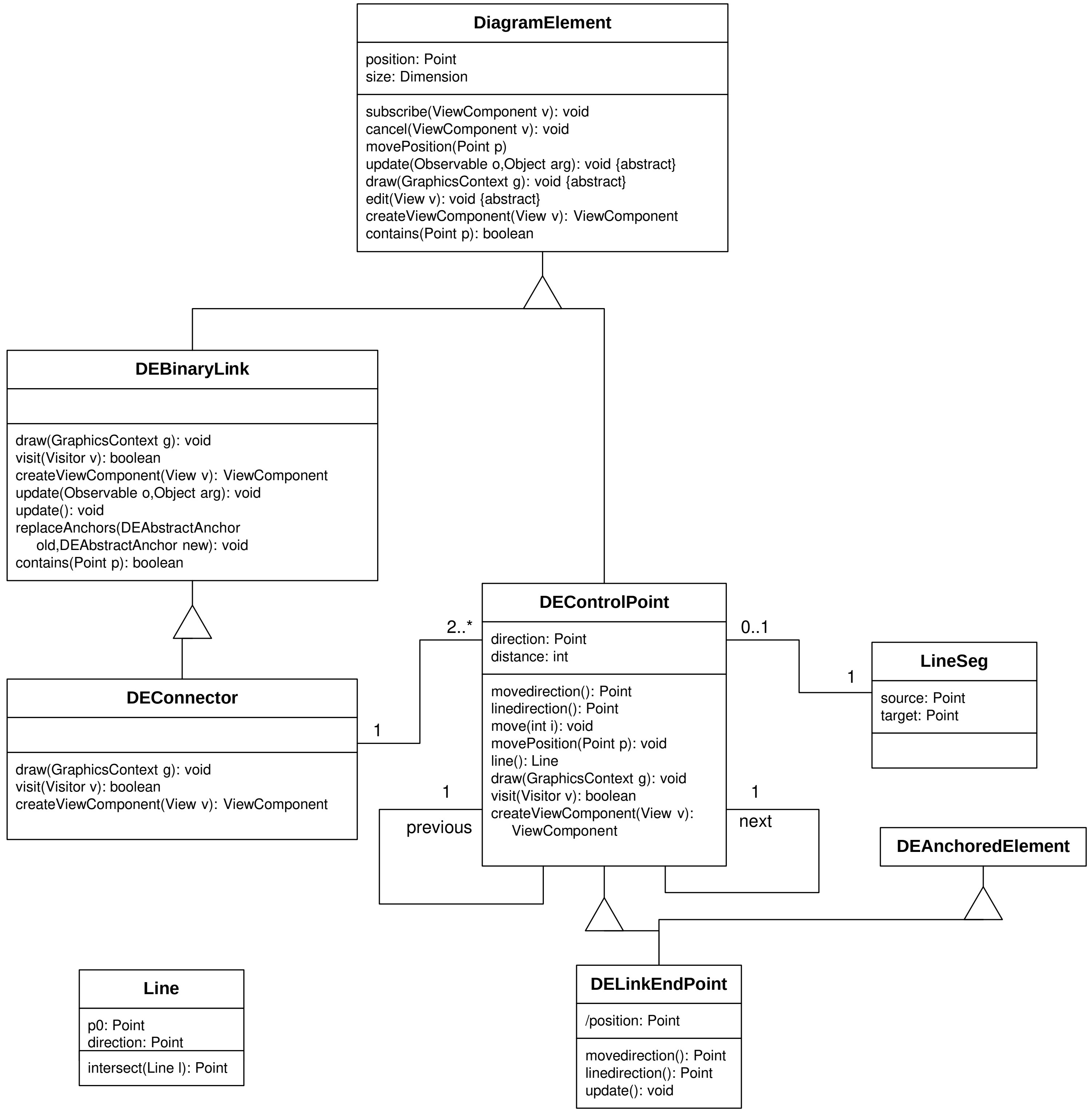


Abbildung 4.53: Diagrammelemente für Verbinder

4.2 Sprachentwurf I

In Anlehnung an die Modellierungssprache OMT, sowie die Vergleiche mit anderen Modellierungsmethoden (OOD, BOOCH, STATEMATE, UML, diese werden im Kapitel Anhang genauer erlätert) wurde die SIGN Modeling Language (SIGN ML) entwickelt. Diese legt im folgenden die Notation (Syntax) der wichtigsten Konzepte für die statische Sicht (Objektmodell) fest und wird daher auch „Basisnotation“ genannt.

4.2.1 Basisnotation

4.2.1.1 Einleitung

Das grundsätzliche Ziel des Projekts SIGN bestand darin, eine grafische Sprache für ein statisches Objektmodell zur Spezifikation der Datensicht von Programmen zu entwickeln. Des weiteren wurde auch ein grafischer Editor für diese Objektmodelle entwickelt, der eine Abbildung und strukturelle Implementierung des Schemas in der Zielsprache (diese waren Java, SQL und Postscript) ermöglicht. Die Tätigkeiten der Arbeitsgruppe „Sprachen“ beruhten deshalb im 1. Projektsemester hauptsächlich darauf, eine 1. Version der SIGN-Notation für eine grafische Software-Entwurfssprache festzuschreiben. Hierfür wurde OMT als Referenzmethode erarbeitet und mit anderen Methoden verglichen, woraus dann ein gemeinsamer Nenner festgelegt wurde. Bei dieser 1.Version der SIGN-Notation haben wir uns speziell auf das Objektmodell von OMT konzentriert, um schon mal ein Grundgerüst für den 1. Entwurf zu haben. In der weiteren Version des Entwurfs (Entwurf II) wurden die weiteren Teilmodelle (dynamisches Modell, funktionales Modell), sowie die wichtigsten Konzepte der Vergleichsmethoden (Booch, Coad/Yourdon, Statemate, UML) mit einbezogen.

Im folgenden werden die Konzepte des Objektmodells genauer erläutert und mit ihren grafischen Symbolen, wie sie in der OMT beschrieben sind, dargestellt. Es ist aber hier anzumerken, daß es sich nicht um das gesamte Objektmodell handelt. Es sind wirklich die wichtigsten Konzepte auserwählt worden, die uns auf dem Weg zum Hauptziel (siehe oben) des Projekts weiterbringen sollten. Angemerkt sei des weiteren, daß bei dieser Ausarbeitung nicht die grafischen Symbole (oder deren Notation) im Vordergrund stehen. Es geht hier vielmehr um die Konzepte, die verwendet wurden. Daher wurden die ausgewählten Konzepte nach einer umfangreichen Diskussion mit der gesamten Projektgruppe festgelegt und waren bindend. Im Gegensatz dazu wurden die grafischen Symbole der einzelnen Komponenten nach dem eigenen Ermessen der Arbeitsgruppe ”Werkzeuge” (diese beschäftigte sich mit der Implementierung) gestaltet. Zum besseren Verständnis der beschriebenen Konzepte werden hier die grafischen Symbole (nach OMT) mit dargestellt.

4.2.1.2 Die einzelnen Konzepten mit Beschreibung und grafischem Symbol

Klasse

Eine Objektklasse (häufig wird statt Objektklasse die Abkürzung Klasse verwendet) beschreibt eine Gruppe von Objekten mit ähnlichen Eigenschaften (Attributen), gemeinsamen Verhalten (Operationen), gemeinsamen Relationen zu anderen Objekten und einer gemeinsamen Semantik. „Person“, „Tier“, „Prozeß“ sind alles Objektklassen. Jede Person hat ein Alter, einen IQ und möglicherweise einen Beruf. Jeder Prozeß hat einen Besitzer, eine Priorität und eine Liste notwendiger Ressourcen. Objekte einer Klasse besitzen die gleichen Attribute und Verhaltensmuster. Die meisten Objekte gewinnen ihre Individualität durch Unterschiede in ihren Attributwerten und Relationen zu anderen Objekten. Aber auch Objekte mit identischen Attributwerten und Relationen sind möglich. Jedes Objekt kennt seine Klasse. Die meisten objektorientierten Programmiersprachen können die Klasse eines Objekts zur Laufzeit feststellen. Die Klasse eines Objekts ist eine implizite Eigenschaft des Objekts. Durch Gruppierung von Objekten zu Klassen abstrahieren wir ein Problem.

Das OMT-Symbol für eine Klasse ist ein Rechteck mit dem Klassennamen in fetter Schrift (siehe auch Abbildung 4.54).

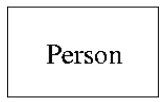


Abbildung 4.54: Klasse

Attribut

Ein Attribut ist ein Datenwert (in Programmiersprachen häufig auch Datenelement), den die Objekte in einer Klasse besitzen. Z.B. sind Name, Alter, Gewicht usw. Attribute der Objekte aus der Klasse Person. Jedes Attribut besitzt einen Wert für jede Instanz. Beispielsweise hat das Attribut Alter im Objekt Joe Smith den Wert "24" (d.h. Joe Smith ist 24 Jahre alt). Andere Instanzen können die gleichen oder andere Werte für ein gegebenes Attribut besitzen. Jeder Attributname ist innerhalb einer Klasse eindeutig (im Gegensatz zur Eindeutigkeit über alle Klassen hinweg). Deshalb können zwei verschiedene Klassen jeweils ein gleiches Attribut besitzen. Ein Attribut sollte ein reiner Datenwert sein, kein Objekt. Im Gegensatz zu Objekten besitzen reine Datenwerte keine Identität (deshalb ist z.B. das Auftreten einer Integerzahl nicht von einem anderen Auftreten der gleichen Integerzahl unterscheidbar).

Attribute werden im zweiten Teil der Klassenbox aufgeführt. Hinter jedem Attributnamen können optionale Details stehen (z.B. ein Typ oder ein Default-Wert). Vor jedem Typ steht ein Doppelpunkt. Vor jedem Default-Wert steht ein Ist-gleich-Zeichen (siehe Abbildung 4.55).

Operation und Methode

Eine Operation ist eine Funktion oder Transformation, die auf Objekte oder von Objekten einer Klasse angewendet werden kann. Alle Objekte in einer Klasse haben die gleichen Operationen. Die gleiche Operation kann für viele unterschiedliche Klassen gelten. Dabei spricht man von polymorphen Operationen

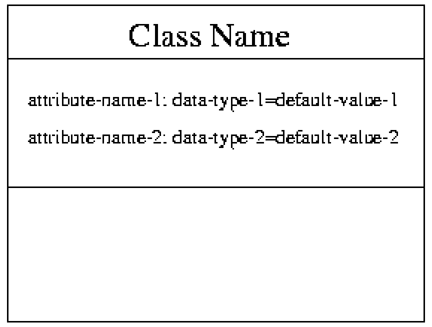


Abbildung 4.55: Attribut

(die gleiche Operation tritt in verschiedenen Klassen in verschiedenen Gestalten auf). Eine Methode ist die Implementierung einer Operation für eine Klasse. Beispielsweise kann die Klasse „File“ eine Operation „print“ besitzen. Es können verschiedene Methoden implementiert werden, um ASCII-Dateien, Binärdateien und digitalisierte Bilddateien zu drucken. Alle diese Methoden führen logisch die gleiche Aufgabe aus - das Drucken einer Datei. Sie können daher durch die allgemeine Operation „print“ auf die Methode verweisen. Jedoch kann jede Methode durch ein anderes Codestück implementiert sein.

Operationen werden im unteren Drittel der Klassenbox angegeben. Hinter jeder Operation können optionale Details stehen, z.B. eine Argumentliste oder ein Ergebnistyp. Eine Argumentliste steht hinter dem Namen in Klammern und die Argumente werden durch Kommas voneinander getrennt. Außerdem können der Name und Typ jedes Arguments angegeben werden. Vor jedem Ergebnistyp steht ein Doppelpunkt (siehe Abbildung 4.56).

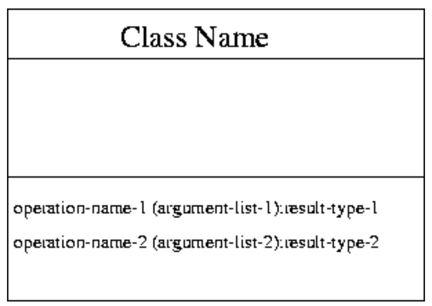


Abbildung 4.56: Operation

Assoziation

Assoziationen (Beziehungen) sind Konzepte, um Relationen zwischen Klassen herzustellen. Eine Assoziation beschreibt eine Gruppe von Links (Verknüpfungen) mit einer gemeinsamen Struktur und Semantik. Eine Person „works-for“ für eine Firma ist ein Beispiel für eine Assoziation. Alle Verknüpfungen in einer Assoziation verbinden Objekte der gleichen Klassen. Eine

Assoziation beschreibt eine Menge potentieller Verknüpfungen auf die gleiche Weise, in der eine Klasse eine Menge potentieller Objekte beschreibt. Assoziationen sind inhärent bidirektional. Assoziationen werden in Programmiersprachen häufig als Zeiger (Pointer) von einem Objekt zum anderen implementiert. Ein Zeiger ist ein Attribut in einem Objekt, das einen expliziten Verweis (Referenz) auf ein anderes Objekt enthält. Assoziationen können binär, ternär oder höherer Ordnung sein. In der Praxis sind fast alle Assoziationen binär oder qualifiziert (Sonderform von ternär, die später noch erörtert wird).

Das OMT-Symbol für eine binäre Assoziation ist eine Linie zwischen den beiden in Relation stehenden Klassen. Eine ternäre und n-äre Assoziation wird als Raute, von der Linien ausgehen, die die zueinander in Relation stehenden Klassen verbinden, dargestellt. Der Name der Assoziation steht an der Assoziationslinie (siehe Abbildung 4.57).

Abbildung 4.57: Assoziation

Kardinalität (Multiplicity)

Die Kardinalität spezifiziert, wieviele Instanzen einer Klasse mit einer Instanz einer assoziierten Klasse verknüpft sein können. Kardinalität begrenzt die Anzahl verknüpfter Objekte.

Im allgemeinsten Fall kann Kardinalität durch eine Zahl oder eine Intervallmenge angegeben sein, z.B. "1" (genau eins), "1+" (eins oder mehr), "3-5" (drei bis einschließlich fünf) und "2,4,18" (zwei, vier oder achtzehn). Bestimmte häufig vorkommende Kardinalitäten werden durch spezielle Linienenden gekennzeichnet. Ein ausgefüllter Kreis steht für „m“, d.h. null oder mehr. Ein unausgefüllter Punkt steht für „optional“, d.h. null oder eins. Eine Linie ohne Kardinalitätssymbole bezeichnet eine 1:1-Assoziation. Im Allgemeinen wird die Kardinalität am Linienende angegeben (siehe Abbildung 4.58).

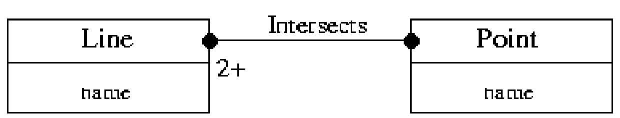


Abbildung 4.58: Kardinalität

Assoziationsattribut

Ein Attribut ist eine Eigenschaft der Objekte einer Klasse. Analog dazu ist ein Assoziationsattribut (Verknüpfungsattribut) eine Eigenschaft der Links (Verknüpfungen) einer Assoziation. Jedes Assoziationsattribut hat einen Wert für jeden Link.

Die OMT-Notation für ein Assoziationsattribut ist ein Rechteck, das durch eine Schleife an die Assoziation angehängt wird (siehe Abbildung 4.59). In dem zweiten Bereich des Rechtecks können ein oder mehrere Assoziationsattribute enthalten sein. m:n-Assoziationen liefern die überzeugendste Begründung für die Assoziationsattribute. Ein Assoziationsattribut ist unzweifelhaft eine Eigenschaft einer Verknüpfung und kann nicht einem der beiden Objekte zugeordnet werden.

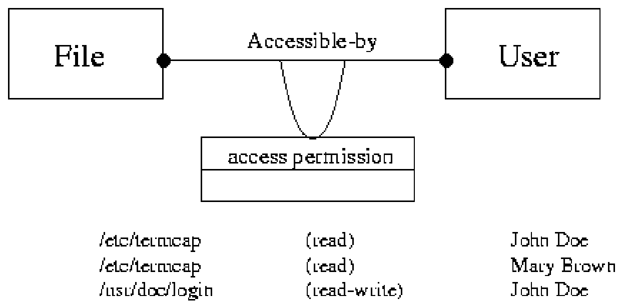


Abbildung 4.59: Assoziationsattribut

Rollenname

Das Ende einer Assoziation wird Rolle genannt. Eine binäre Assoziation besitzt somit zwei Rollen. Jede davon kann einen Rollennamen haben. Ein Rollenname ist ein Name, der ein Ende einer Assoziation eindeutig identifiziert. Jede Rolle einer binären Assoziation identifiziert ein Objekt oder eine Objektmenge, das (die) mit einem Objekt am anderen Ende assoziiert ist. Der Rollenname ist ein abgeleitetes Attribut, dessen Wert eine Menge von verknüpften Objekten ist. Durch die Verwendung von Rollennamen ist es möglich, Assoziationen von einem Objekt aus zu durchlaufen, ohne die Assoziation explizit zu benennen. Rollen haben in Problembeschreibungen oft die Form von Substantiven.

Ein Rollenname steht in OMT an der Assoziationslinie neben der Klasse, die die Rolle spielt (das heißt, der Rollenname erscheint am Zielende der durchlaufenen Assoziation, siehe Abbildung 4.60). Die Verwendung von Rollennamen ist optional, es ist aber oft einfacher und weniger verwirrend, Rollennamen anstelle von oder zusätzlich zu Assoziationsnamen zu verwenden.

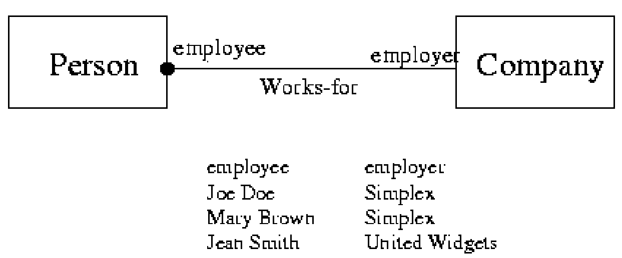


Abbildung 4.60: Rollenname

Qualifikation (Qualifizierte Assoziation)

Eine qualifizierte Assoziation verbindet zwei Klassen und eine Qualifikationsangabe. Die Qualifikationsangabe ist ein besonderes Attribut, das die effektive Kardinalität einer Assoziation herabsetzt. Qualifiziert werden können 1:m- und m:n-Assoziationen. Eine qualifizierte Assoziation kann auch als eine Form einer ternären Assoziation aufgefaßt werden. Qualifikation setzt in der Regel, aber nicht immer, die Kardinalität von m auf 1 herab. Qualifikation verbessert die semantische Genauigkeit. Es ist wesentlich informativer zu erfahren, daß z.B. ein Verzeichnis und ein Dateiname zusammen eine Datei identifizieren, als zu erfahren, daß ein Verzeichnis viele Dateien enthält.

Eine Qualifikationsangabe wird als kleines Rechteck am Ende der Assoziationslinie neben der Klasse, die sie qualifiziert, dargestellt (siehe Abbildung 4.61).

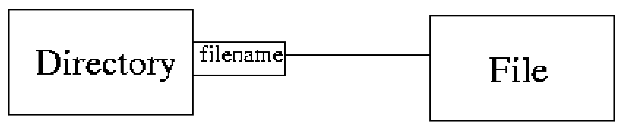


Abbildung 4.61: qualifizierte Assoziation

Aggregation

Aggregation bezeichnet die „Teil-Ganzes-“ oder „ist-Teil-von“-Relation. Hierbei sind die Objekte, die die Komponenten einer Sache repräsentieren, mit einem Objekt assoziiert, das die Komponentengruppe repräsentiert. Ein einzelnes Objekt kann nur in einer direkten Aggregation partizipiert werden. „Parallele“ Aggregationen werden zusammengefaßt, um Hierarchieebenen zu betonen. Aggregation ist eine eng gekoppelte Form der Assoziation mit einigen extra Bedingungen (zusätzlicher Semantik). Die wichtigste Eigenschaft der Aggregation ist die Transitivität, daß heißt: wenn A ein Teil von B ist, und B ein Teil von C ist, dann ist A auch ein Teil von C. Ferner gilt die Eigenschaft der Antisymmetrie, das heißt: wenn A ein Teil von B ist, dann ist B nicht Teil von A. Wir definieren eine Aggregationsrelation als Beziehung zwischen einer Komponentengruppenklasse mit einer Komponentenklasse. Eine Komponentengruppe mit vielen verschiedenen Komponenten hat dementsprechend viele Aggregationsrelationen. Wir definieren jede einzelne Paarrelation als Aggregation, damit wir die Kardinalität jeder Komponente innerhalb der Komponentengruppe spezifizieren können. Diese Definition unterstreicht, das die Aggregation eine Sonderform der Assoziation ist.

Eine Aggregation wird wie eine Assoziation dargestellt, wobei eine kleine Raute das Komponentengruppen-Ende der Relation kennzeichnet (sieheAbbildung 4.62).

Generalisierung und Vererbung

Generalisierung und Vererbung sind mächtige Abstraktionsmöglichkeiten, die dazu dienen, Ähnlichkeiten zwischen verschiedenen Klassen zu teilen und gleichzeitig ihre Unterschiede zu erhalten. Der Begriff Generalisierung bezieht sich

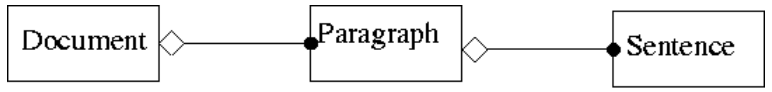


Abbildung 4.62: Aggregation

auf die Relation zwischen Klassen, der Begriff Vererbung bezieht sich auf den Mechanismus, Attribute und Operationen über eine Generalisierungsstruktur zu gewinnen. Generalisierung stellt die Relation zwischen einer Klasse und einer oder mehrerer verfeinerter Versionen davon dar. Es geht hier also um die Vererbung von Attributen, Methoden und Assoziationen, wobei die zu vererbende Klasse als Oberklasse, und die vererbte Klasse als Subklasse betrachtet wird. Jede Subklasse erbt die Merkmale ihrer Oberklasse. Man spricht hierbei auch von disjunkter Spezialisierung. Die Generalisierung wird manchmal auch „is-a“ Beziehung genannt, weil jede Instanz einer Subklasse eine Instanz der dazugehörigen Oberklasse ist. Jede Subklasse erbt aber nicht nur alle Merkmale seines Vorfahren. Es kann auch seine eigenen Attribute und Operationen haben. Die Vererbung wird auch in objektorientierten Programmiersprachen unterstützt. Wichtigster Nutzen der Vererbung ist die konzeptuelle Vereinfachung durch Reduzierung der Anzahl der abhängigen Merkmale eines Systems.

Die Vererbung bietet auch die Möglichkeit zum Überschreiben von ererbten Attributen und Methoden. Eine Subklasse kann ein Merkmal einer Oberklasse überschreiben, indem es ein Merkmal mit dem gleichen Namen definiert. Die Subklassen-Eigenschaft verfeinert somit die überschriebene Eigenschaft der Oberklasse. Es gibt normalerweise zwei Gründe, warum man ein Merkmal überschreiben sollte:

- Um die Definition des Originalmerkmals völlig zu ersetzen.
- Um (bei Operationen) die Originalmethode um ein zusätzliches Verhalten zu erweitern.

Die Notation für die Vererbung ist ein Dreieck, das eine Oberklasse mit ihren Subklassen verbindet. Die Oberklasse wird durch eine Linie mit der Spitze des Dreiecks verbunden. Die Subklassen werden durch Linien mit einem horizontalen Balken verbunden, der über die Grundlinie des Dreiecks verläuft (siehe Abbildung 4.63).

Abstrakte Klassen

Eine abstrakte Klasse ist eine Klasse, die selbst keine direkten Instanzen hat, deren Nachkommen aber direkte Instanzen besitzen (siehe Abbildung 4.64). Eine konkrete Klasse ist eine instanziierbare Klasse, d.h. sie kann direkte Instanzen besitzen. Eine konkrete Klasse kann abstrakte Unterklassen (die ihrerseits jedoch konkrete Nachkommen besitzen müssen) haben. Abstrakte Klassen organisieren Merkmale, die mehreren Klassen gemeinsam sind. Es ist oft nützlich eine abstrakte Oberklasse zu schaffen, um Klassen zusammenzufassen, die an der gleichen Assoziation oder Aggregation beteiligt sind. Einige abstrakte Klassen existieren bereits in der Anwendungsdomäne. Andere abstrakte Klassen

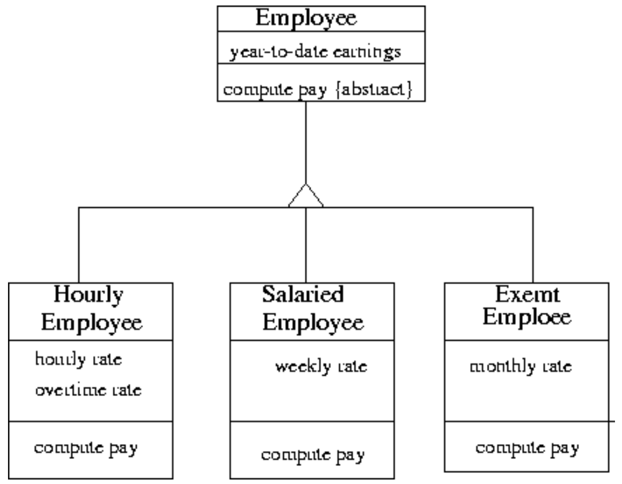


Abbildung 4.63: Vererbung

werden künstlich als Mechanismus zur Förderung von Codewiederverwendung eingeführt. Abstrakte Klassen werden häufig zur Definition von Methoden verwendet, die an Subklassen vererbt werden. Eine abstrakte Klasse kann aber auch nur das Protokoll für eine Operation definieren, ohne eine entsprechende Methode zur Verfügung zu stellen. In diesem Fall haben wir es mit einer abstrakten Operation zu tun. (Eine abstrakte Operation definiert die Form einer Operation, für die jede konkrete Subklasse ihre eigene Implementierung zur Verfügung stellen muß. Eine konkrete Klasse darf keine abstrakten Operationen enthalten, weil Objekte der konkreten Klasse sonst undefinierte Operationen enthalten würden. Eine abstrakte Operation wird als Kommentar in geschweiften Klammern angegeben)

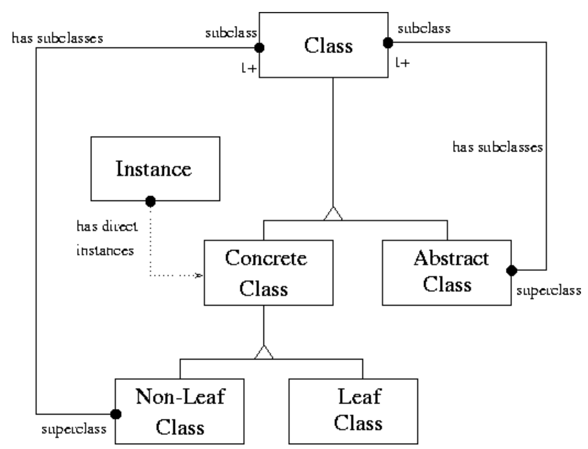


Abbildung 4.64: abstrakte Klasse

Spezialisierung mit Erweiterung

Die Generalisierungsstruktur kann eingesetzt werden, um die Eigenschaften der Oberklasse zu erweitern. Eine Instanz einer Klasse ist eine Instanz aller Vorfahren der Klasse. Dies ist ein Teil der Definition für die Generalisierung. Deshalb müssen alle Merkmale einer Vorfahrenklasse für die Instanzen einer Unterklasse gelten. Eine Nachkommenklasse kann ein Attribut einer Vorfahrenklasse nicht weglassen oder unterdrücken, da es sich sonst nicht um eine echte Vorfahreninstanz handeln würde. Ähnlich müssen Operationen auf einer Vorfahrenklasse für alle Nachkommenklassen gelten. Eine Subklasse kann eine Operation aus Effizienzgründen zwar anders implementieren, sie kann aber nicht das externe Protokoll ändern. Eine Subklasse kann neue Merkmale hinzufügen. Dies nennt man Erweiterung (siehe Abbildung 4.65). Die Erweiterung ist möglich bezüglich der

- Attribute
- Beziehungen
- Operationen

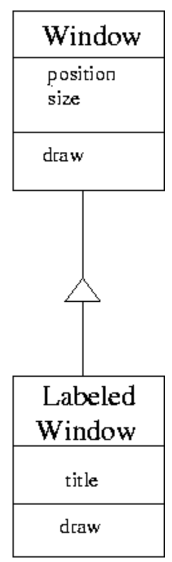


Abbildung 4.65: Spezialisierung mit Erweiterung

Spezialisierung mit Begrenzung

Eine Subklasse kann darüber hinaus Vorfahrenattribute auch einschränken. Dies heißt Begrenzung, weil die Werte, die Instanzen annehmen können, eingegrenzt werden.

Die Begrenzung kann sich auf den Wertebereich der Attribute der Oberklasse, auf das Verhalten von Operationen oder auch auf den Wertebereich der Argumente und Rückgabewerte beziehen. Wenn die Generalisierung mit Begrenzung der Eigenschaften der Oberklasse eingesetzt wird, kann sich die Notwendigkeit zum Überschreiben (Redefinieren) von Operationen zum Zweck der Begrenzung ergeben. Ein anderer Zweck ist das Überschreiben zur Optimierung. Wenn der Wertebereich von Argumenten und / oder Attributen eingeschränkt ist, kann dadurch evtl. ein geeigneterer (effizienterer, sicherer, schönerer...) Algorithmus verwendet werden (sieheAbbildung 4.66).

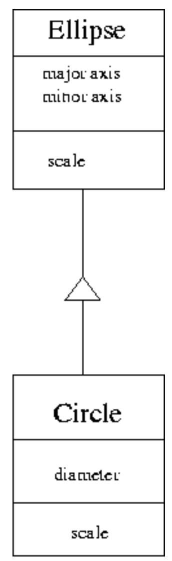


Abbildung 4.66: Spezialisierung mit Begrenzung

Mehrfachvererbung

Durch Mehrfachvererbung ist es möglich, daß eine Unterklasse mehrere direkte Oberklassen hat und Merkmale von allen Eltern erbt. Eine solche Unterklasse wird Vereinigungsklasse genannt. Sie erbt Eigenschaften aus mehreren Generalisierungsbeziehungen. Ein Beispiel ist die Klasse „Amphibious Vehicle“. In einer Generalisierungsstruktur für Fahrzeuge, die auf dem Kriterium Einsatzort bzw. Einsatzumgebung (discriminator) beruht, könnte diese Klasse eine Vereinigung der Klassen „Land Vehicle“ und „Water Vehicle“ sein.

Falls die direkten Oberklassen einer Mehrfachvererbungsstruktur aus demselben Generalisierungsbaum stammen, können sie nicht disjunkt sein. Bei Mehrfachvererbung ist Mehrdeutigkeit möglich, wenn mehrere Oberklassen dieselbe Operation oder dasselbe Attribut (bzw. eine gleichnamige Eigenschaft) haben. Eigenschaften einer indirekten Oberklasse werden nur einmal vererbt.

Die Mehrfachvererbung wird durch ein leeres Dreieck zwischen den Superklassen und ihren Subklassen dargestellt (siehe Abbildung 4.67) .

Metadaten: Klassenattribute und -operationen

Metadaten sind Daten, die andere Daten beschreiben.

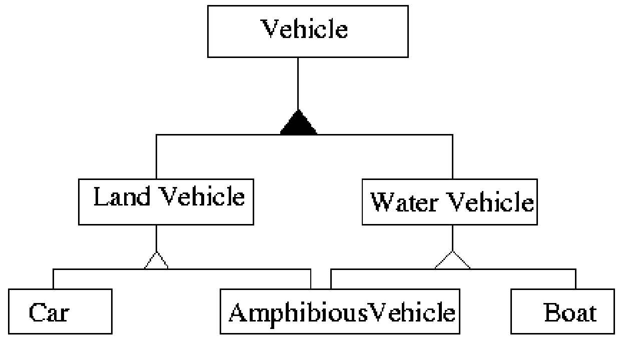


Abbildung 4.67: Mehrfachvererbung

Beispiel: Architekturplan beschreibt Haus

Klassenattribute sind Werte, die für eine ganze Klasse von Objekten gelten.

Klassenoperationen sind Operationen auf der Klasse selbst.

Systeme, die Metadaten enthalten, modellieren eine zusätzliche Abstraktionsebene, in der über Objekte „gesprochen" werden kann. Metadaten sind häufig verwirrend, weil sie die gewöhnlich vorhandene Grenze zwischen dem Modell und der realen Welt verwischen. Klassen können selber Objekte sein. Dann sind sie Metaobjekte und Instanzen einer Metaklasse „Klasse". Klassenobjekte können in ihrer Eigenschaft als Objekt auch Attribute und Operationen besitzen, die dann Klassenoperationen und -attribute genannt werden. In OMT werden solche Elemente durch ein Dollarzeichen gekennzeichnet (siehe Abbildung 4.68). Programmiersprachen und Datenbanksysteme bieten diese Konzepte an:

In C++ und Java gibt es Metaattribute und -operationen (static). In Smalltalk gibt es Metaklassen.

In Datenbanksystemen gibt es neben den Anwendungsdaten auch Daten, die Tabellenstrukturen und Attribute beschreiben (Data Dictionary).

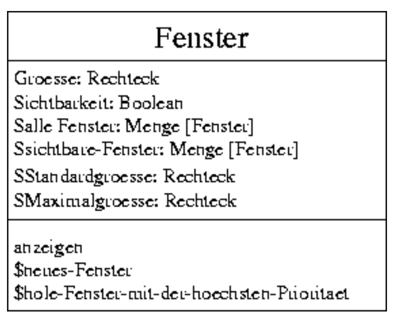


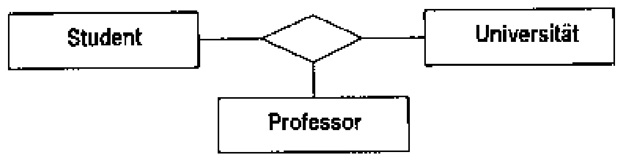
Abbildung 4.68: Metadaten: Klassenattribut / Klassenoperation

Kandidatenschlüssel

Kandidatenschlüssel ist ein anderes Konzept, um Multiplizität auszudrücken. Kandidatenschlüssel werden verwendet, um die Multiplizität n-ärer Beziehungen eindeutig anzugeben. OMT bietet mit Kandidatenschlüsseln eine andere Möglichkeit, um die Multiplizität (Kardinalität) von Beziehungen anzugeben. Kandidatenschlüssel werden in geschweiften Klammern „in der Nähe“ der Beziehung angegeben.

Bei zweistelligen Beziehungen gibt es keine bedeutenden Unterschiede zur Punktnotation, bei mehrstelligen Beziehungen sind Kandidatenschlüssel jedoch die einzige Möglichkeit die Multiplizität eindeutig anzugeben (siehe Abbildung 4.69).

Ein Kandidatenschlüssel ist eine minimale identifizierende Menge von Objekten einer Beziehung (d.h. es kann auch mehrere KS geben). Das Konzept ist eng verwandt mit den Schlüsselkonzepten der Theorie relationaler Datenbanken (siehe auch Kurs Datenbanksysteme 1).



{Kandidatenschlüssel: {Student, Universität}}

Student	Professor	Universität
Mary	Prof Weaver	Cambridge
Mary	Prof Rumrow	MIT
Susan	Prof Weaver	MIT
Susan	Prof Weaver	Cambridge
Bob	Prof Shapiro	Oxford

Abbildung 4.69: Kandidatenschlüssel

Einschränkungen

Einschränkungen sind funktionale Beziehungen zwischen Entitäten (umfaßt Objekte, Klassen, Attribute, Links und Assoziationen) eines Objektmodells. Eine Einschränkung begrenzt die Werte, die Entitäten annehmen können. Beispielsweise kann das Gehalt eines Mitarbeiters niemals das Gehalt des Chefs dieses Mitarbeiters überschreiten. Einschränkungen sind ein Kriterium, die Qualität eines Objektmodells zu messen: ein "gutes" Objektmodell erfaßt viele Einschränkungen durch seine Struktur.

OMT verwendet für nicht grafisch notierbare Sachverhalte Anmerkungen in geschweiften Klammern:

- bei Kandidatenschlüsseln
- für Ordnungseigenschaften von Assoziationen
- für Abstraktheit von Operationen

Damit können auch allgemeine Einschränkungen auf Objekten angegeben werden. OMT verwendet geschweifte Klammern für Einschränkungen und alles, für das keine grafische Notation vorgesehen ist; im Prinzip sind die Klammern Kommentarzeichen, da keine Syntax festgelegt ist. Daran wird auch deutlich, worin die Schwäche einer grafischen Notation besteht.

Die Objektmodellierungssyntax für Einschränkungen sieht also so aus, daß diese in geschweiften Klammern angegeben werden und neben der eingeschränkten Entität stehen (siehe Abbildung 4.70). Eine gepunktete Linie verbindet mehrere eingeschränkte Entitäten. Mit einem Pfeil kann eine eingeschränkte Entität mit einer Entität verbunden werden, von der sie abhängt.

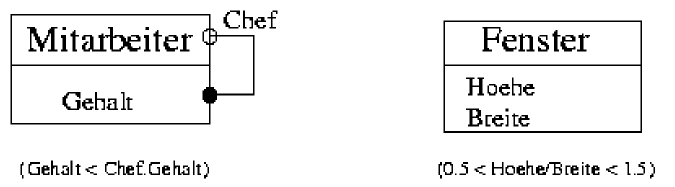


Abbildung 4.70: Einschränkung

4.3 Semantik I

4.3.1 Vorwort

Der vorliegende Text soll die Semantik der Basisnotation der graphischen Modellierungssprache im Projekt SIGN (SIGN ML) formell beschreiben.

Die Beschreibung folgt i.W. der mengentheoretischen Definition der Semantik des erweiterten ER-Modells, wie sie aus der Vorlesung „Datenbanksysteme II“im SS 1997 von Martin Gogolla bekannt ist. Als Referenz hierzu dienen die Folien zur Vorlesung [Go97] sowie der Text „Conceptual modelling of database applications using an extended ER model“von G.Engels, M.Gogolla, U.Hohenstein, K.Hülsmann, P.Löhr-Richter, G.Saake, H.Ehrich in Data & Knowledge Engineering 9 (1992/93), S.157-204 [En92].

Die zunächst zu beschreibende Basisnotation der SIGN ML enthält nur die Struktur- und Datensicht auf ein System in Form von Klassendiagrammen. Anders als das EER-Modell soll eine weiterentwickelte Version der Notation auch die Modellierung der funktionellen und dynamischen Sicht auf ein System erlauben.

Hierzu sollen State-Charts und Activity-Charts Verwendung finden. Diese dienen insbesondere der graphischen Modellierung der Zustandsänderungen sowie der Daten- und Kontrollflüsse.

4.3.2 Syntax der SIGN ML

Die Syntax der Basisnotation der SIGN ML ist beschrieben durch das Metamodell im Entwurf Endbericht (Kap. 3.3.1.1. Hierachie Logischer Elemente I). Für Beispiele und Erläuterungen der einzelnen Objekte ist die Basisnotation hilfreich.

4.3.3 Konventionen

4.3.3.1 Elementare Datentypen

- VAR: ist die Klasse aller bennanten Variablen von Daten
- $DATA$: ist die Klasse der elementaren, vordefinierten Datentypen wie zum Beispiel {int, decimal(n,m), real, string, ... }
- $\mid FISET \mid$: ist die Klasse der endlichen Mengen
- $\mid FUN \mid$: stellt die Klasse aller totalen Funktionen dar

4.3.3.2 Klassendiagramm

Gemäß der Syntax ist ein Klassendiagramm gegeben durch

- eine Menge von Klassen CLS
- eine Menge von Attributen ATT
- eine Menge von Operationen OPN mit Parametern
- eine Menge von Klassenbeziehungen REL mit Rollen für die beteiligten Klassen
- eine Menge von Generalisierungen GEN

sowie Einschränkungen CON, qualifizierte Attribute QUA, Kandidatenschlüssel CKE.

4.3.4 Klassen

Im Klassendiagramm wird eine Klasse durch Angabe einer Signatur der Form:

Klassenname

$$
\begin{aligned}
& Attribut_1 : Typ_1 \ldots Attribut_n : Typ_n, \\
& Operation_1(Parameter_{11} : Parameter_{Typ_{11}} \ldots \\
& \qquad Parameter_{1m} : Parameter_{Typ_{1m}}) : Ergebnis_{Typ_1} \ldots \\
& Operation_k(Parameter_{k1} : Parameter_{Typ_{k1}} \ldots \\
& \qquad Parameter_{km} : Parameter_{Typ_{km}}) : Ergebnis_{Typ_k}
\end{aligned}
$$

Entwurf SIGN: Komponente Anwendungsgebiet
V 0.8 / 29.05.97
- Hierarchie Logischer Elemente I -

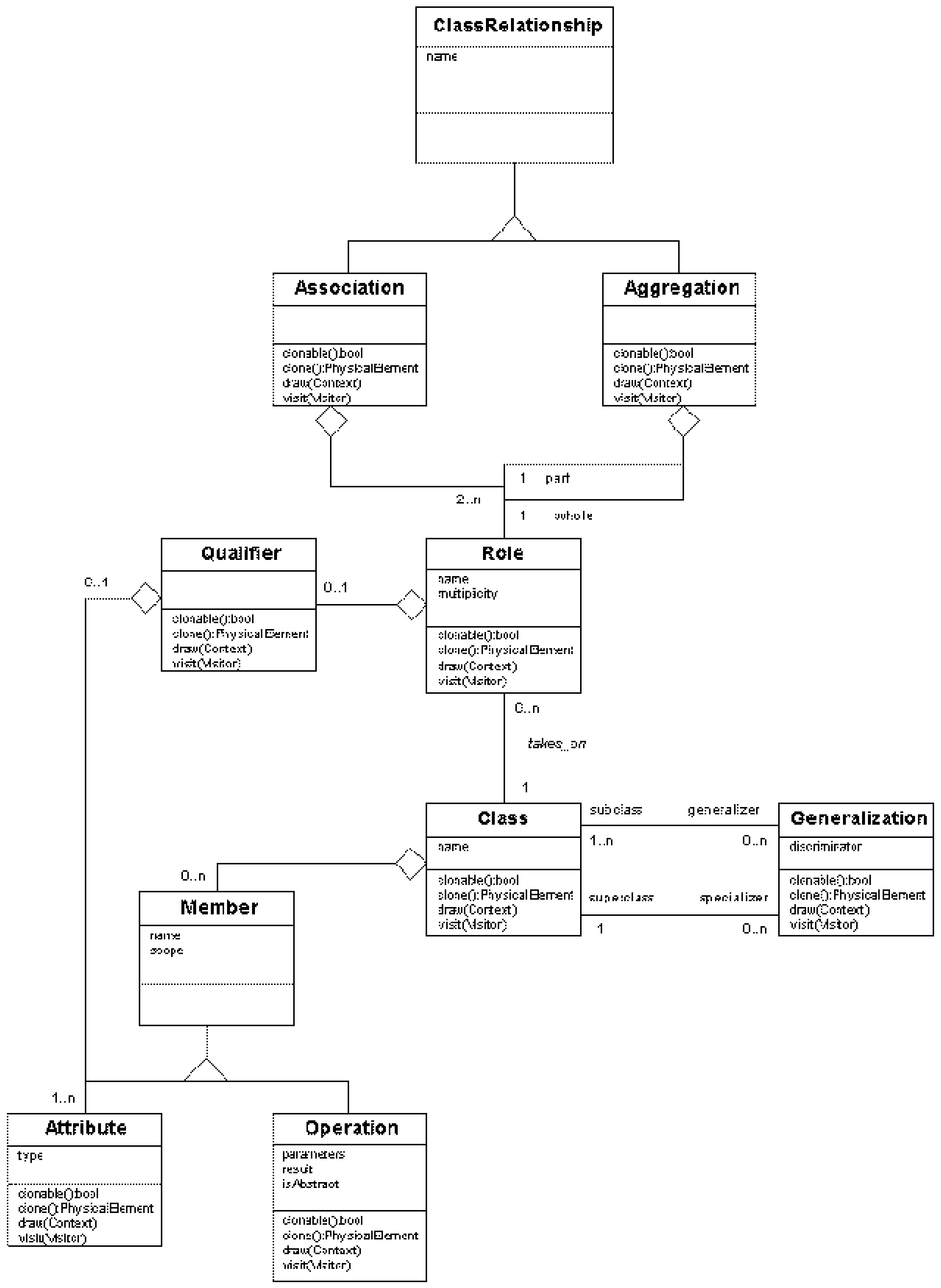


Abbildung 4.71: Hierachie Logischer Elemente

oder kurz:

$$C(a_1 : t_{a_1} ... a_n : t_{a_n}, op_1(p_{11} : t_{11} ... p_{1m} : t_{1m}) : t_{o1} ... op_k(p_{k1} : t_{k1} ... p_{km} : t_{km}) : t_{ok})$$

repräsentiert. Die Semantik einer Klasse C in einem Modell wird beschrieben durch die Semantikfunktion $S_c : CLS \rightarrow \mid FISET \mid$ mit $S_c(C) = \{c_1, ..., c_n\}$

Die Semantikfunktion S_c ordnet jeder Klasse eine endliche Menge von Instanzen (Objekten) zu. Mit anderen Worten: die Interpretation einer Klasse C ist gegeben durch die Menge der Instanzen der Klasse $\{c_1, ..., c_n\}$ im System zu einem Zeitpunkt. Dabei ist noch nicht bestimmt, ob diese Objekte benannt sind, d.h. über symbolische Namen zugreifbar sind.

Ferner ist nicht bestimmt, ob es sich um persistente Objekte in einer Datenbank oder transiente Objekte im Hauptspeicher eines Rechners handelt.

Im EER-Modell gelten Klassen als Zusammenfassung gleichartiger Objekte. Die Interpretation einer Klasse ist dann gleichbedeutend mit dem Zustand des Systems zu einem Zeitpunkt (Instanz). Demnach:

$$S_c(\sigma) : CLS \rightarrow \mid FISET \mid$$

Der Zustand ordnet jeder Klasse genau eine Menge von Instanzen zu, die den jeweiligen Sytemzustand beschreiben. Die programmiersprachliche Bedeutung des Zustandsbegriffs ist die Zuordnung von Variablen zu einer Belegung bzw. konkreten Werten. D.h.:

$$\sigma : \mid V_t \mid \rightarrow \mid D_t \mid$$

für alle $V_t \subseteq \mid VAR \mid$, $D_t \in$ DATA mit einem entsprechendem Typ t der Variablen V_t und dem zugeörigen Wertebereich D_t. Letztere Bedeutung wird wichtig, wenn die Semantik von Ausdrücken der dynamischen Sicht (State-Charts) zu erklären ist. Schwerpunkt dieser Systemmodellierungsperspektive sind die Zustandsänderungen des Systems mit deren erweiterten Beschreibungsmöglichkeiten (z.B. gemäß einer Action-Syntax der State-Charts) zu erklären, was die Bedeutung eines Ausdrucks in einem bestimmten Systemzustand σ ist.

In diesem Zusammenhang ist die Interpretation einer Klasse, die eines abstrakten Datentyps. Das Klassendiagramm gibt die Signatur des abstrakten Datentyps an. Innerhalb eines solchen Klassendiagramms wird ebenfalls nur die Signatur der vorhandenen Klassenoperationen deklariert. Die Spezifikation wird vervollständigt durch die Beschreibung der Operationen auf dem Datentyp in der dynamischen und funktionellen Sicht.

Das System, dessen Zustand hier betrachtet wird, ist als abstrakter Systemzustand innerhalb des Modells aufzufassen. Es ist der Träger einer konkreten Realisierung des Modells. Im Modell wird von vielen Eigenschaften eines konkreten Systems, wie etwa begrenztem Speicher, Zeitbedarf der Instruktionsausführung, etc. abstrahiert.

4.3.4.1 Attribute

Mit $C :: a : t$ wird ein Attribut a vom Typ t in der Klasse C bezeichnet. Ein Attribut $C :: a : t$ ist eine Abbildung von einer Klasse C in einen Datenbereich D_t vom Typ t.

$$a : C \rightarrow D_t$$

Der Definitionsbereich eines Attributs ist die Klasse, zu der das Attribut gehört. Der Wertebereich ist durch den Typ gegeben.

Anders als beim EER-Modell sind hier keine objektwertigen Attribute zu berücksichtigen, da in der SIGN ML stattdessen Klassenbeziehungen verwendet werden.

Die Bedeutung eines Attributs ist formell durch die Semantikfunktion S_a zu beschreiben.

$$S_a : ATT \rightarrow \mid FUN \mid$$

und für alle Attribute $a \in ATT$ mit $a : C \rightarrow D_t$ gilt:

$$S_a(a) : S_c(C) \rightarrow S_d(D_t)$$

oder mit $S_a[C]$, $S_c[C]$ als Einschränkung der Semantikfunktionen S_a, S_c auf die Klasse C:

$$S_a[C] : S_c[C] \rightarrow S_d(D_t)$$

wobei S_d die Semantikfunktion für den entsprechenden Datentyp sei, die den aktuellen Wert bzw. die Belegung gemäß dem Typ in einem Systemzustand liefert.

Im Hinblick auf die später zu erklärende Semantik von Ausdrücken der dynamischen Sicht, sind die Attributnamen Bezeichner für lokale, benannte Variablen.

4.3.4.2 Operation

Eine Operation op in der Klasse C mit Parametern $p_1, ..., p_n$ vom Typ $t_1, ..., t_n$ und einem Rückgabewert vom Typ t_0 wird angegeben als

$$C :: op(p_1 : t_1, ..., p_n : t_n) : t_0$$

Die Operation ist eine Abbildung $op : C \times D_{t1} \times ... \times D_{tn} \rightarrow D_{t0}$ mit $D_{ti} \in DATA$ $\cup$ CLS. Die Semantikfunktion S_o für Operationen ist eine Abbildung S_o, wobei OPN die Menge aller Operationen sei, wie folgt: S_o : OPN $\rightarrow \mid FUN \mid$ so daß für alle $op : C \times D_{t1} \times ... \times D_{tn} \rightarrow D_{t0}$ gilt: $S_o(op) : S(C \times D_{t1} \times ... \times D_{tn}) \rightarrow S_d(D_{t0})$ somit $S_o(op) : S_c(C) \times S(D_{t1}) \times ... \times S(D_{tn}) \rightarrow S(D_{t0})$ mit $S(D_{ti}) = S_c(D_{ti})$, falls $D_{ti} \in$ CLS und $S(D_{ti}) = S_d(D_{ti})$, falls $D_{ti} \in DATA$.

4.3.5 Klassenbeziehungen

4.3.5.1 Assoziationen

Eine Klassenbeziehung r (im EER-Modell in ähnlicher Form als Relation bekannt) mit $r \in REL$, wobei REL die Menge aller Relationen sei, und den beteiligten Klassen C_i, mit $1 \leq i \leq n$, $n \in N$ hat die Form: $r(C_1, ..., C_n)$. Dabei können zwischen den Klassen verschiedene Kardinalitäten (zwingend oder auch freistehend) existieren.

Auch sind Klassenbeziehungen zwischen Klassen des gleichen Typs (mit der identischen Signatur) denkbar, die jedoch erst in Verbindung von Beziehungsattributen (evtl. in der erweiterten SIGN-Notation) eine sinnvolle Modellierungsmethode anbieten.

Für die Assoziationen sind die beteiligten Klassen ungewichtet (abgesehen von der Bedeutung, die die Assoziation wiederspiegeln soll), so daß eine ungerichtete Verbindung zwischen diesen durch eine Assoziation ausgedrückt wird.

Die Semantikfunktion einer Klassenbeziehung S_r ist gegeben durch eine Abbildung $S_r : REL \rightarrow \mid FISET \mid$ und $\forall\ r(C_1, ..., C_n) \in REL$ gilt:

$$S_r(r) = S_c(C_1) \times ... \times S_c(C_n)$$

.

Die Interpretation einer Assoziation ist also zu verstehen, als eine mögliche Verbindung zwischen Instanzen vom Typ der beteiligten Klassen einer Beziehung. In einem konkreten Systemzustand σ stellen die Verbindungen Instanzen einer Assoziation dar.

4.3.5.2 Aggregationen

Eine Aggregation ist ein Assoziation für die Klassen C_i mit $i = 2$, d.h. von der Form $r(C_1, C_2)$. Für die Aggregation stellen die beteiligten Klassen eine „Teil von Ganzen"-Beziehung dar. Demnach ist eine Aggregation, anders als bei der Assoziation, eine asymmetrische Beziehung und ihre Semantik unterscheidet sich dementsprechend.

Für die Semantikfunktion S_r gelten für Aggregationen noch zusätzliche Bedingungen, die ihre unterschiedliche Bedeutung prägen.

Sei C_1 die Montage und C_2 die Komponente dann gilt:

- Existenz: $\exists(r(C_1, C_2) \wedge c \in S_c(C_2)) \Longrightarrow \exists(c' \in C_1 \wedge S_r(r(c', c)))$ in jedem Systemzustand σ
- Transitivität: $\exists(r_1(C_1, C_2) \wedge r_2(C_2, C_3)) \Longleftrightarrow \exists\, r_3(C_1, C_3)$

Für die Semantik einer Aggregation in der verkürzten graphischen Darstellung (Kurzschreibweise: Linien mit mehr als zwei Endungen) mit der Form $r(C_0, C_1, ..., C_n)$, wobei C_0 die Montage und $C_{i \in N}$ die Komponenten darstellt, gelten äquivalent dazu

$$r_1(C_0, C_1), ..., r_n(C_0, C_n)$$

Aggregationen.

4.3.5.3 Generalisierung und Vererbung

Eine Generalisierung ist in ihrer Semantikfunktion ähnlich der der Aggregation, jedoch in einer anderen Bedeutung. Sowie eine Aggregation ist eine Generaliserung eine gewichtete Beziehung. Die beteiligten Klassen nennt man Super-(oder auch Ober-) und Unterklassen. Ihre Bedeutung ist jedoch unterschiedlich (siehe SIGN-Basisnotation).

Für die Semantikfunktion S_r gelten die der Assoziationen mit den zusätzlichen Bedingungen der Aggregation für die Transitivität und ihrer hierachischen Anordnung.

Weitere Bedingungen, die für Generalisierungen gelten, trennt die Semantikfunktion der Generalisierung von der der Aggregation.

Sei C die Superklasse und C' die Unterklasse, op, a alle Operationen und Attribute einer Klasse dann gilt:

- Existenz(erweitert)): $\exists(r(C, C') \wedge c \in S_c(C')) \Longrightarrow \exists(c' \in S_c(C) \wedge r(c, c'))$ in jedem Systemzustand σ. Zusätzlich gilt: $\forall\, c, c' \in S_c(C), c'' \in S_c(C')(\exists\, r(c, c'') \wedge r(c', c'') \Longrightarrow c = c')$

- Vererbung: $\forall\, c \in C, c' \in C' (\exists\, r(c,c') \Longrightarrow (\forall (op \in c \wedge a \in c) \Longrightarrow (op, a \in c')))$

Da in der Basisnotation Operationen nur hinsichtlich ihrer Signatur deklariert werden, kann ein Überladen bzw. Überschreiben von Operationen also Spezialisierungen und Erweiterungen nicht adäquat ausgedrückt werden, und ist aus diesem Grund hier vernachlässigt worden.

4.3.6 Qualifikation

Ein Qualifier Q ist ein Attribut zu einer Assoziation. Nur zweiwertige (Tupel) Assoziationen mit den Kardinalitäten (1:n) oder (n:m) dürfen qualifiziert werden. Eine qualifizierte Assoziation hat die Form $r_q(C : Q, C')$, wobei C die qualifizierende Klasse ist.

Betrachtet man das Attribut Q als eine eigenständige Klasse mit ihren möglichen Werten ihres Datentyps als Instanzen dieses Qualifiers, kann man die Semantik eines Qualifiers mit einer ternären Assoziation beschreiben: $r_{q1}(C, Q, C')$ mit der zusätzlichen Bedingung $r_{q2}(Q, C')$, wobei r_{q1} eine Assoziation und r_{q2} eine Generalisierung mit Q als Superklasse ist d.h. alle Instanzen von C' werden genau einm Wert in Q zugeoordnet, wobei alle Werte des Attributs unterschiedlich sind, da diese (s.o Generalisierung) als „Instanzen“betrachtet werden.

Die Semantikfunktion eines Qualifiers folgt aus der Semantik der ternären Assoziation. Damit ergibt sich für die Semantikfunktion S_q folgende Signatur: $S_q : REL \times ATT \rightarrow \mid FISET \mid \times DATA$ mit $S_q(r_q) = S_{rq1}(C) \times S_{rq1}(C') \times S_a(q)$.

4.3.7 Candidate Key

Kanditatenschlüssel sind eine andere Möglichkeit Multiplizitäten zwischen beteiligten Klassen einer Assoziation anzugeben. Ein Kandidatenschlüssel $CandidateKeyc$ zu einer Assoziation $r(C_1, ..., C_n)$ hat die Form $c(C_1, ..., C_i)$ wobei $i \leq n$ gilt.

Für die betroffenen Klassen C_i gilt folgende Bedingung: $\forall\, c_n, c'_n \in C_n$ und $\forall\, r, r'$ wobei c instanziierte Klassen der Assoziation r und $c' \in C'$ sind, gilt: $\forall\, i : c_i = c'_i \Longleftrightarrow r = r'$.

4.3.8 Constraints

Ein Constraint ist eine Einschränkung zwischen zwei Entitäten, die mit einer funktionellen Beziehung ausgedrückt wird. Eine Einschränkung begrenzt die Werte, auch in Abhängigkeit von Werten anderer Entitäten, die die Entitäten annehmen können bzw. deren Anzahl von Instanzen.

Um eine genaue Semantik der Constraint formulieren zu können, werden nur ausreichend konkret formulierte Einschränkungen zugelassen. Das heißt Einschränkungen zwischen zwei Entitäten mit Werten werden, nur für alle entsprechend definierten Vergleichsoperatoren mit denen der gesamte Ausdruck für den gesamten möglichen Datenbereich ein Wahrheitswert ausgewertet werden kann, gültig.

Entitäten ohne Werte, werden die Anzahlen der Instanzen als Werte gelten, um eine einigermaßen geeignete Transparenz mit dem Datentyp int zu gewährleisten.

Eine Einschränkung *Constraint* hat die Form $r(E_1, ..., E_n)$ wobei E_i mit $i \in 1, ..., n$ die beteiligten Entitäten sind, mit einer Gleichung G, die in der noch folgenden deSIGN-Constraint Language (an OCL streng anlehnend), mit den Werten der Entitäten bzw. die Anzahl der Instanzen der Entitäten als Variablen.

Die Gleichung beschreibt die Abhängigkeiten der Enititäten bzw. deren Werten mit den erlaubten Vergleichsoperationen. Und ist für die Spezifizierung eines Constraints notwendig. Auch die Form der Gleichung bestimmt, ob ein Constraint gültig ist oder nicht bzw. unter welchen Vorraussetzungen ein Constraint gültig wäre.

Die Semantik einer Einschränkung als Bedingung formuliert ist: In jedem Systemzustand σ gilt die Gleichung G, für alle Datenwerte, und Instanzen der Entitäten.

Kapitel 5

Konstruktionsphase II

In diesem Kapitel soll der Entwurf der dynamischen (State-Charts) und funktionalen Sicht (Activity-Charts) beschrieben werden, die im Verbund mit der Object Constraints Language (OCL) eine Erweiterung der SIGN ML bilden. Der Systementwurf wurde hinsichtlich der neuen Konzepte der SIGN ML angepaßt und der Entwurf des Editors deSIGN somit um Funktionalität ergänzt.

Der vorliegende Entwurf zur Erweiterung des graphischen Editors auf die funktionale und dynamische Sicht soll lediglich eine moegliche Fortsetzung der Arbeit skizzieren. Da wir eine praktische Umsetzung des Entwurfs in einer weiteren Implementationsphase in der zur verfuegung stehenden Zeit nicht planen, beschraenken wir uns auf einen groben Entwurf der Struktur- und Datensicht. Hierzu werden im Folgenden hauptsaechlich Klassendiagramme verwendet.

Die Erweiterung des Entwurfs diente in unserem Projekt vornehmlich dem Zweck, die praktische Nutzbarkeit des Editors deSIGN an einem Beispiel realistischer Groesse zu ueberpruefen. Die nachfolgenden Klassendiagramme sind saemtlich mit dem im Projekt entwickelten Werkzeug - dem graphischen Editor deSIGN - erstellt worden. Die Durchfuehrung des Entwurfs II hat unter diesem Gesichtspunkt seinen Zweck als qualitaetssichernde Massnahme erfuellt.

5.1 Systementwurf II

5.1.1 Logische Elemente der State-Charts

Die Einbindung der „State Charts“ in die logischen Elemente erfolgt analog zu den der Klassen-Diagramme. Wir erweitern die von der Klasse „LogicalElement“ abstammenden Klassen der logischen Elemente um die erforderlichen Klassen: LEState, LETransition, LEEventExpression, LECondition und LEAction.

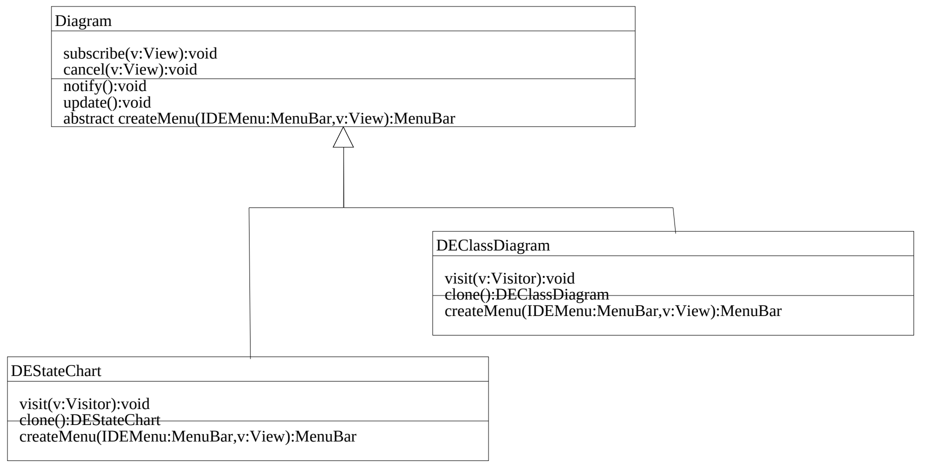


Abbildung 5.1:

Sie bilden das Grundgerüst des „State-Charts“ und werden analog zum bisherigen Klassendiagramm in der Klasse „Module“ durch Aggregationen zusammengefaßt (siehe Abbildung 5.1).

5.1.2 Logische Elemente: Activity Charts

Um Aktivitätsdiagramme darstellen zu können, benötigen wir vor allem Aktivitäten und Datenflüsse. Wie diese sich in die Hierarchie der Logischen Elemente einordnen, zeigt die Abbildung 5.2 (vgl. auch Sprachentwurf II):

Module beinhalten nicht nur Klassen, Generalisierungen, etc., sondern auch Activities und Flows, welche wiederum Unterklassen von Logical Element sind.

Wir unterscheiden fünf verschiedene Arten von Activities und drei Flows: PseudoActivities, ControlActivities, SimpleActivities, ActivityCharts und DataStores, sowie ComplexFlows, Dataflows und ContolFlows.

PseudoActivities beinhalten Aktivitäten und Flows. Hiermit können wir mehrere Aktivitäten gruppieren.

ControlActivities sind mit Activity Charts verknüpft, damit letztere auf (De-)Terminismus überprüft werden können.

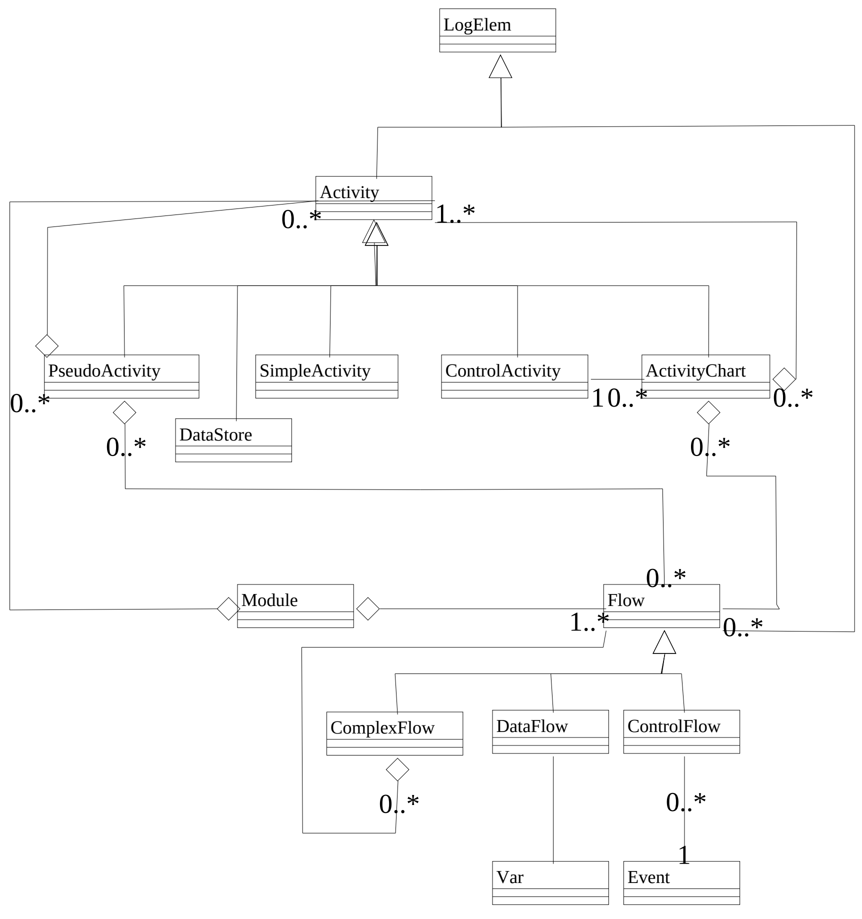


Abbildung 5.2:

SimpleActivities sind (wie der Name schon vermuten läßt) einfache Aktivitäten, die nicht weiter zerlegbar sind. Sie beinhalten keine weiteren Aktiväten und sind auch nicht als Kontrollaktivität verwendbar.

Aktivitäten können wiederum aus mehreren „Unter-“Aktivitäten bestehen. Aus diesem Grund haben wir ActivityCharts als Unterklasse von Activity eingeführt.

DataStores repräsentieren Datenspeicher und können genau wie SimpleActivities nicht weiter zerlegt werden. Sie beinhalten nur ein Datum mit dem sie über eine Assoziation verbunden sind.

ComplexFlows werden verwendet, um mehrere Flows zu einem zusammenfassen zu können. Sie aggregieren Flows.

DataFlows übermitteln Daten zwischen Aktivitäten. Sie sind über eine Assoziation mit ihrem Datum verbunden.

ControlFlows repräsentieren Signale, die die Aktivitäten steuern. Sie sind ebenfalls über Assoziationen mit den Ereignissen verbunden. Jedes Ereignis kann von mehreren ControlFlows verwendet werden.

5.1.3 Diagramm Elemente StateCharts

Um die States im State Chart darzustellen, erzeugen wir ein Diagramm Element Namens DEState, die sehr ähnlich wie die Klasse DEClass aus dem Klassendiagramm (siehe Entwurf I - DiagramElemente) funktioniert (siehe auchAbbildung 5.3). Unsere DEState Klasse besitzt wieder Ankerpunkte, an denen Verbindungen angehängt werden können. Die Klassen DEClass und DEState unterscheiden sich hauptsächlich nur in ihrer grafischen Darstellung als Symbol auf dem jeweiligem Diagramm und in ihrer Zuordnung zu ihrem Logischen Element Class bzw. State. Darum erzeugen wir noch zusätzlich eine Klasse DEBox, die von DiagramElement abgeleitet wird. Sie dient als abstrakte Klasse aller Knoten Elemente in unseren Diagrammen und verfügt über alle notwendigen Operationen und Attribute für Knoten. Diese DEBox beinhaltet z.B. die grafische Größe des Element, ihre Position, aber auch die Ankerpunkte an diesem Knoten. Unsere Klassen DEState und DEClass stammen von dieser DEBox ab und überschreiben die notwendigen Operationen, wie z.B. die Draw-Methode.

Im State Chart existieren nach dem Sprachentwurf (siehe Sprachentwurf II) verschiedene Arten von States. Für jede Art eines States fügen wir eine neue Klasse ein, die durch hierarchische Vererbung von der Klasse DEState abstammt. Diese States unterscheiden sich nicht nur in ihrer grafischen Darstellung, denn sie besitzen auch unterschiedliche Verwaltungsoperationen.

DESimpleState und DEComplexState werden direkt von DEState abgeleitet. DESimpleState entspricht dem SimpleState aus dem Sprachentwurf, und kann als ein atomares Element im State-Chart verstanden werden. DEComplexState bildet die Oberklasse von den AND-StateCharts und OR-StateCharts (siehe Abbildung 5.4).

Die And- und Or-StateCharts besitzen wiederum verschiedene Instanzen von States, die durch die Aggregationen dargestellt werden sollen. Ein AND-StateChart kann nur AND- oder OR-StateCharts besitzen, deswegen aggregieret

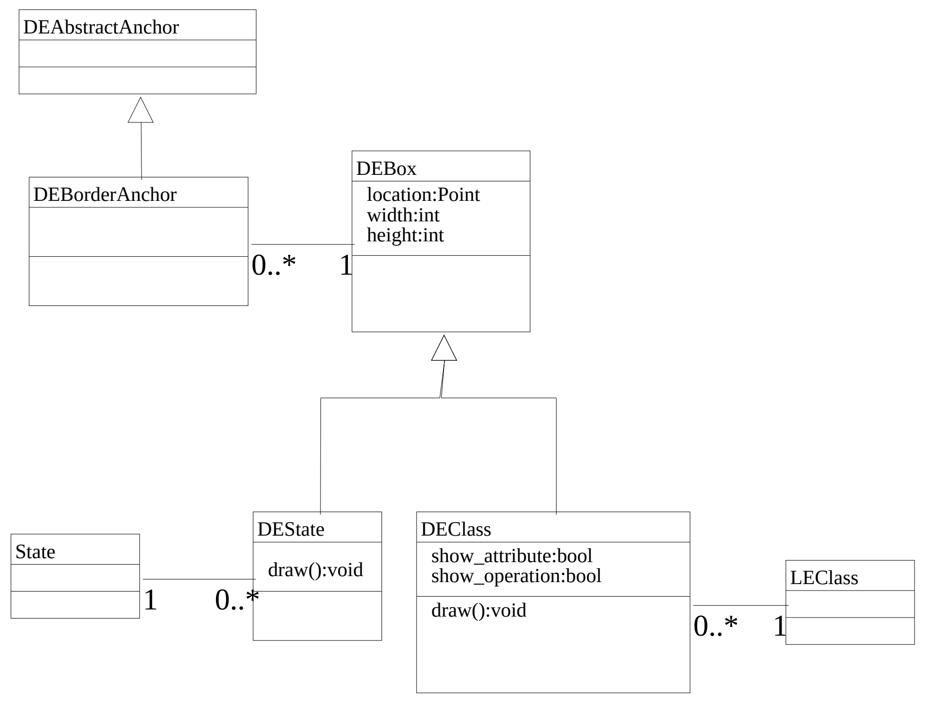


Abbildung 5.3:

DEAndStateChart die Klasse DEComplexStateChart. Da ein OR-StateChart aber dazu noch ein SimpleState besitzen kann, aggregiert dieses statt dessen DEState.

Nur Or-StateCharts können Transitionen besitzen, deswegen aggregieren wir auch nur hier die Tansitionen, d.h. die Klasse DETransition. Diese Transitionen verbinden verschiedene States.

Für den Initial-Transition erzeugen wir eine Klasse DEInitTransition, die von DETransition abgeleitet wird. Diese Transition verbindet nicht zwei States, sondern zeigt von einem Init-Symbol auf einen State.

Die Klasse DETransitionLabel wird von DEAnchoredTextbox abgeleitet und dient zur grafischen Beschreibung von Transitionen.

Die Darstellung einer Transition ähnelt hierbei einer N-ären Assoziation, daher werden alle für beide Sichten relevanten Daten in die Klasse "DEParticipant" verschoben. Die bisherige Klasse "DENAryParticipant" erbt nun von "DEParticipant", besitzt aber weiter alle für die Klassenansicht spezifischen Methoden und Attribute (z.B. eine geeignete Draw-Methode). Die neue Klasse "DETransisionParticipant" wiederum erbt auch alle gemeinsamen Methoden und Attribute, weisst allerdings auch Unterschiede auf (draw- Methode).

Wie bei einer Nären-Assoziation gibt es wieder eine Klasse, die alle internen Informationen speichert, die Participants sind lediglich Graphische „Zeiger“ zwischen Klassen/States. Dies übernimmt "DETransition". Da dieser Aufbau

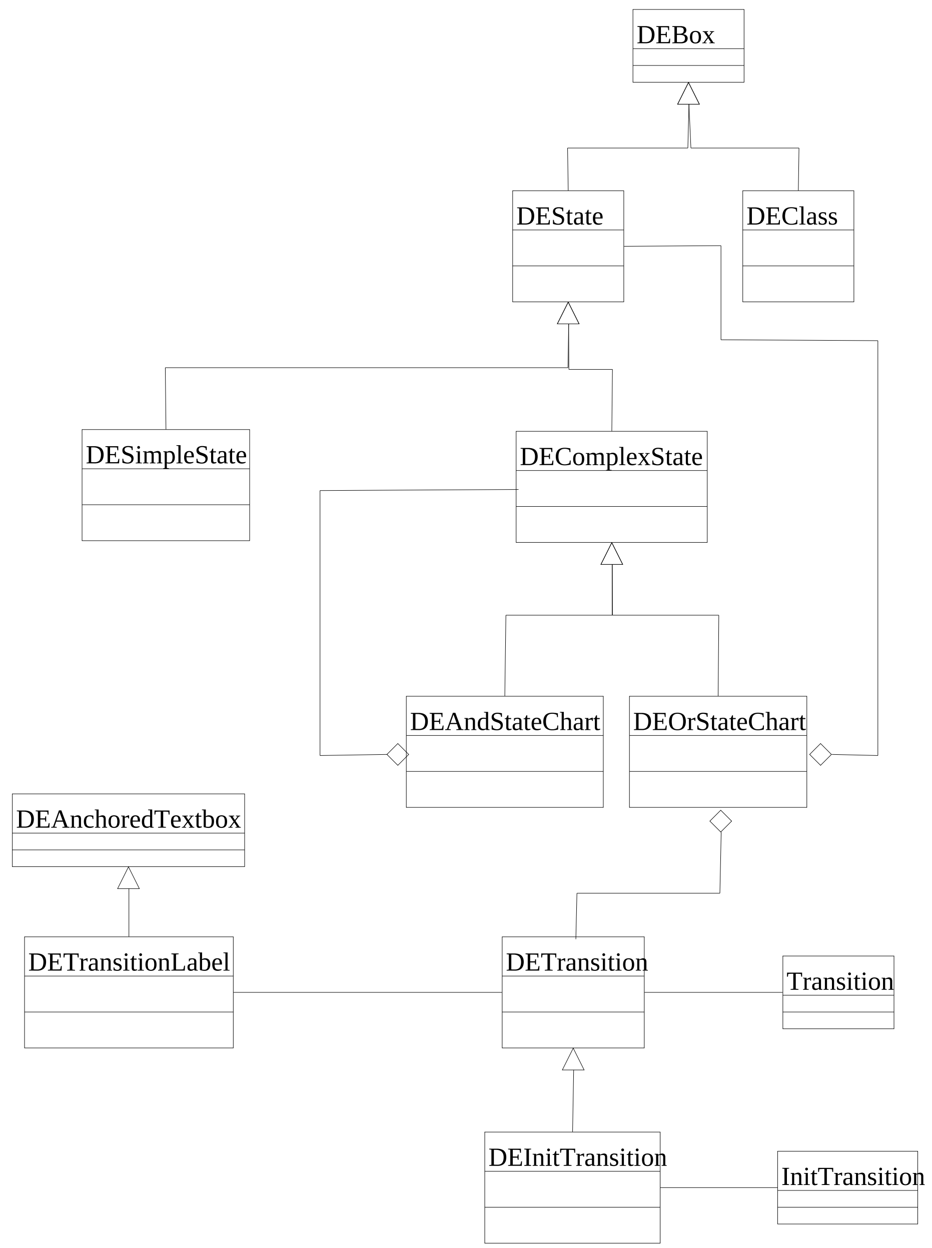


Abbildung 5.4:

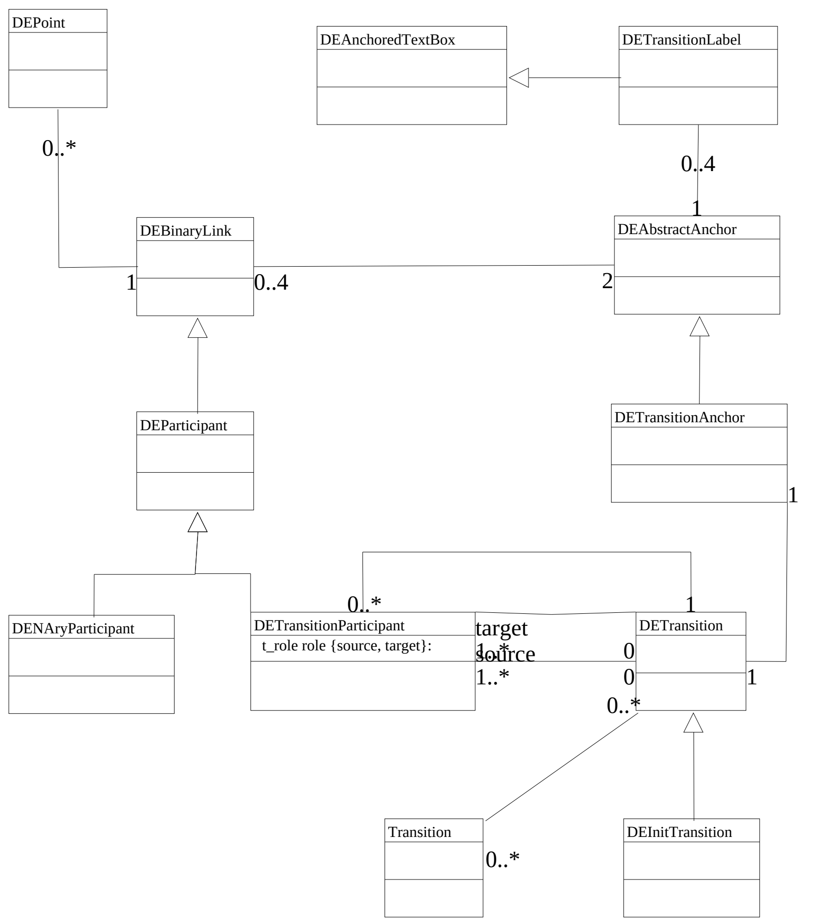


Abbildung 5.5:

nichts mit dem von "DENAryAssociation" zu tun hat, wird hier keine gemeinsame Oberklasse eingeschoben.

Das gleiche Prinzip wie bei den Participants wird bei den Anchors angewandt, hier wird wieder eine gemeinsame Oberklasse vorgeschoben und von "DENAryAnchor" und "DETransitionsAnchor" abgeleitet. Dieses wird in Abbildung 5.5 verdeutlicht.

Um den Zusammenhang zwischen dem Diagramm-Element einer Transition und ihren Quell- und Ziel-Diagramm-Elementen zu verdeutlichen, wird nachfolgend ein Instanzdiagramm (siehe Abbildung 5.6) gezeigt.

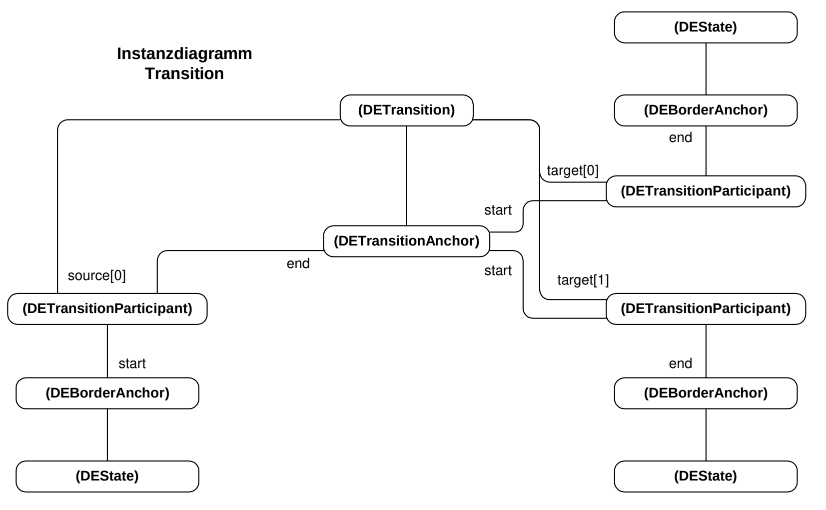


Abbildung 5.6:

Eine Transition kann mehrere Quell- und Ziel-Diagramm-Elemente haben. Ein Beispiel für diese Situation ist in dem nachfolgenden Diagrammausschnitt (siehe Abbildung 5.7) zu sehen.

Um diese n:m-Beziehung handhaben zu können, wird eine Transition durch mehrere Objekte vom Typ DETransitionParticipant repräsentiert, die an einer Instanz von DETransitionAnchor hängen. Der DETransitionAnchor wird von der Transition (Klasse DETransition) zur Verfügung gestellt. Die n:m-Beziehung wird so in eine 1:n-Beziehung der Transition zu ihren Quell-Diagramm-Elementen und eine 1:m-Beziehung zu den Ziel-Diagramm-Elementen zerlegt.

Die 1:n-Beziehung zu den Quell-Diagramm-Elementen wird dargestellt durch die Rolle source. Die Instanz der Klasse DETransition hat ueber die Rolle source den Zugriff auf n Instanzen der Klasse DETransitionParticipant. Jedes der Objekte dieser Klasse stellt eine 1:1-Beziehung zwischen Ankerpunkten dar. Die Klasse DETransitionParticipant ist von der Oberklasse DEBinaryLink abgeleitet. Im Kontext einer Transition sind dies der Ankerpunkt, der

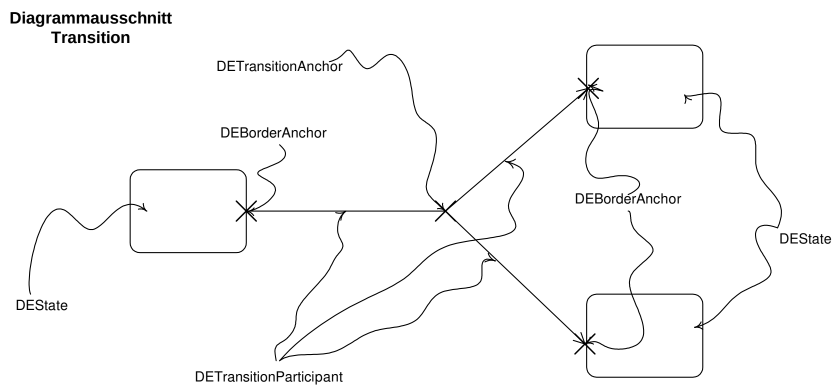


Abbildung 5.7:

von der Transition selber zur Verfügung gestellt wird und ein Ankerpunkt an einem Zustandssymbol (DEState). Analog wird die 1:m-Beziehung zu den Ziel-Diagramm-Elementen realisiert.

Die Transitionen der Statecharts werden prinzipiell genauso durch Diagramm-Elemente repräsentiert, wie die N-ären Assoziationen der Klassendiagramme. Lediglich ihr visuelles Erscheinungsbild ist anders, so daß sie durch neue spezialisierte Unterklassen von Diagramm-Elementen dargestellt werden.

Durch das aggregieren der Klassen DEState und DEComplexState müssen wir hier bei der Implementation bedenken, daß diese beiden Klassen nicht abstrakt definiert sein dürfen.

Die Klasse: „DEStateChart" wird analog zur Klasse „DEClassDiagram" von der Kontainerklasse aller grafischen Elemente, der Klasse „Diagram" abgeleitet, wie dieses bereits im Entwurf I optional vorgesehen war.

Hierdurch wird erreicht, daß „diagrammartunabhängige" Klassen wie z.B.: View, Window, Panel oder ViewComponent, welche von der Klasse „Diagram" verwaltet werden, von allen Diagrammarten (DEClassDiagram, DEStateChart,etc.) durch Vererbung erreichbar sind. Die Klasse „Diagram" enthält die abstrakte Methode „createMenu(...)" welche von den darunterliegenden diagrammabhängigen Oberklassen überschrieben wird, da diese Methode zwar von außen erreichbar sein soll, aber dennoch diagrammabhängig ist.

5.1.4 Diagramm Elemente: Activity Charts

Wie stellen wir nun die Elemente der Activity Charts dar? Um dieses Problem zu lösen existieren (genau wie bei den Klassendiagrammen) die Diagramm Elemente. Wie diese nun zusammenhängen, soll hier beschrieben werden. Einen groben Überblick darüber gibt die Abbildung 5.9 (vgl. auch Sprachentwurf II):

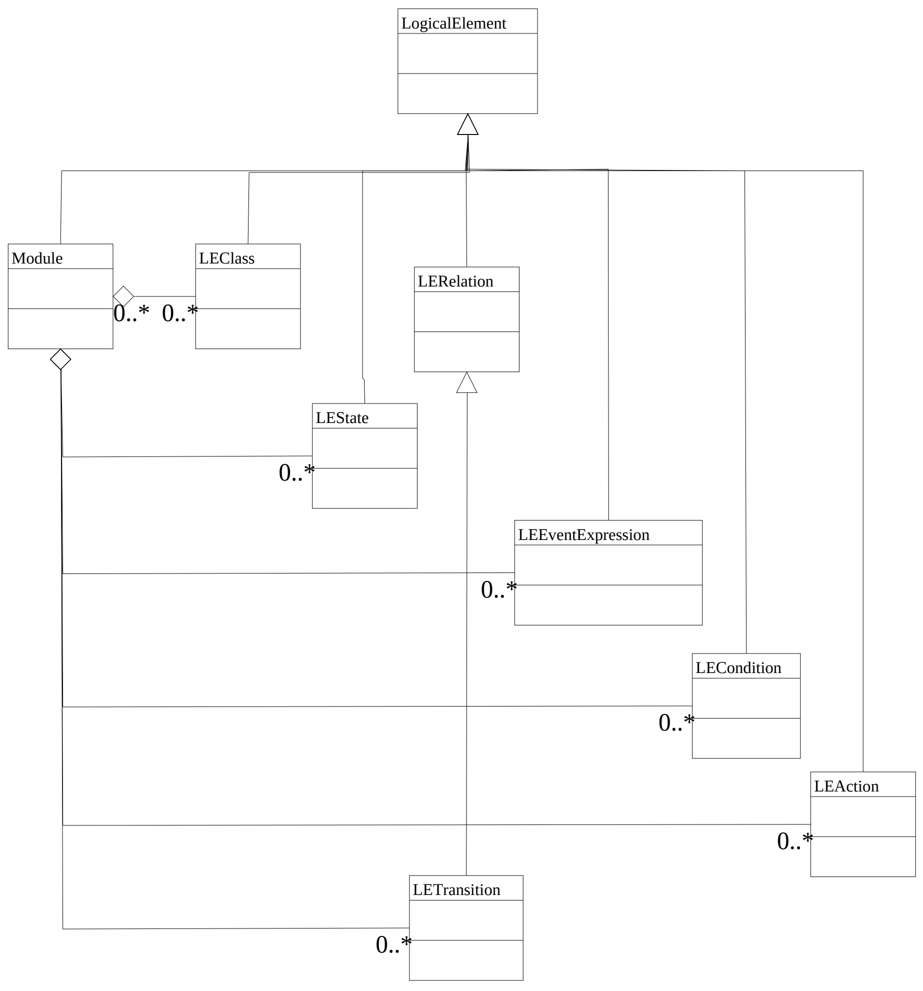


Abbildung 5.8:

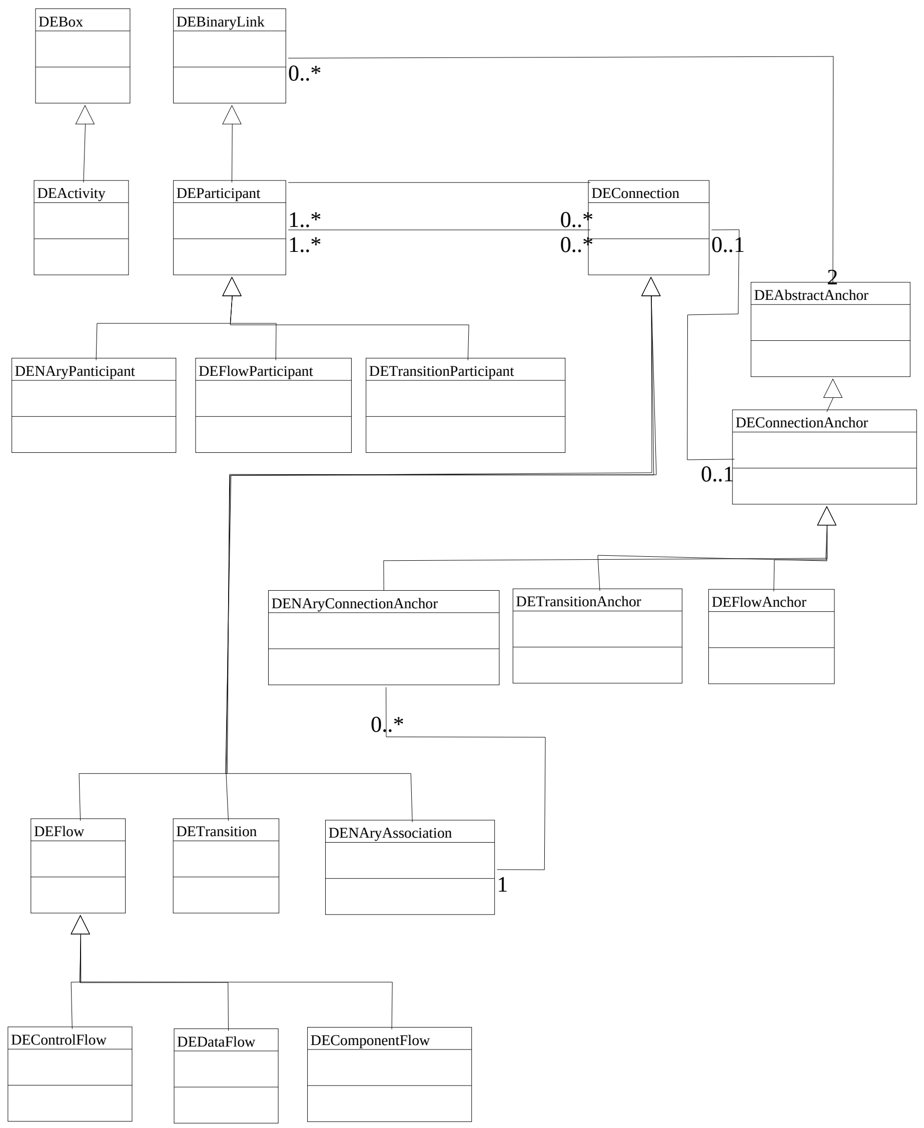


Abbildung 5.9:

DEBox wurde als übergeordnete Klasse für DEClass und DEActivity geschaffen. Hier werden die übergeordneten Attribute und Methoden gespeichert, während in den speziellen Klassen die Unterschiede festgehalten werden (z.B. draw()).

DEBinaryLink ist eine bereits bekannte Oberklasse (aus dem Entwurf I). Wir haben die Unterklassen um DEParticipant erweitert. Diese entspricht der Klasse DENAryAssociation - jedoch dient sie als Oberklasse für DENAryParticipant, DETransitionParticipant und DEFlowParticipant (ähnlich der neuen Oberklasse DEBox).

DEAbstractAnchor ist ebenfalls aus dem Entwurf I bekannt. Analog zu DEBinaryLink existiert hier DEConnectionAnchor als Oberklasse für DENAryAnchor, DETransitionAnchor und DEFlowAnchor.

Als Verbindung zwischen DEParticipant und DEConnectionAnchor haben wir die Klasse DEConnection eingesetzt. Diese ist eine Oberklasse für DENAryAssociation (schon aus Entwurf I bekannt) und den neuen Klassen DETransition und DEFlow. Bei letzterem unterscheiden wir wiederum zwischen DEControlFlow, DEDataFlow und DEComponentFlow.

5.2 Sprachentwurf II

In Anlehnung an die Modellierungssprachen OMT und STATEMATE [Pel95], sowie der Unified Modeling Language (UML) wurde die SIGN ML im folgenden konzeptionell erweitert.

5.2.1 Statecharts

5.2.1.1 Einleitung

Für den Sprachentwurf II haben wir als Alternative zu den bereits behandelten und untersuchten Modellen aus der OMT die Konzepte von STATEMATE untersucht und erarbeitet. STATEMATE ist ein Tool, was den Harelschen Ansatz[HPSS87] (hier die Statecharts) erfolgreich verwendet. Die einzelnen Modelle und deren Konzepte sind auch formeller beschrieben, als es in der OMT der Fall war, was für uns einmal mehr Grund dafür war, diesen Ansatz zu untersuchen. Im Gegensatz zur OMT ist STATEMATE aber nicht objektorientiert. Dieses ist aber für unsere Aktivitäten nicht von Bedeutung, da wir die einzelnen Modelle des Tools betrachten (die ja auf Harel aufsetzen und objektorientiert sind).

Die STATEMATE - Modellierung beinhaltet im wesentlichen die Teilabschnitte Statecharts und Activitycharts, sowie die Constraints.

Die folgenden Ausführungen beziehen sich speziell auf die Konzepte und den Aufbau der Statecharts [GH94], [Per95]. Die Verhaltenssicht (auch dynamisches Modell oder Statecharts genannt) ist in OMT und STATEMATE konzeptionell wesentlich gleich, da beide Sprachen auf Arbeiten von D. Harel aufsetzen. Die Verhaltenssicht beschreibt die dynamischen Ereignisse, d.h. das Erzeugen und Zerstören von Objekten, das Senden von Nachrichten zwischen Objekten usw. Sie beschreibt zeitliche Verhaltensfolgen durch Ereignisse, Zustände, Zustandsdiagramme, Bedingungen und Operationen. Die Verhaltenssicht repräsentiert also im Gegensatz zu dem Objektmodell (diese repäsentiert die statische Struktur eines Systems) die Kontrollstruktur eines Systems.

In STATEMATE werden die Statecharts benutzt, um die Kontrollaktivitäten in den Activitycharts zu beschreiben. Diese Aktivitäten bilden das Verhaltensgerüst eines STATEMATE Modells.

Bei unseren Untersuchungen geht es um die wesentlichen Merkmale der Statecharts. Hierbei wird die direkte Verbindung zu den Activitycharts außer acht gelassen, d.h. wir untersuchen nicht, wie die Aktivitäten durch die Statecharts kontrolliert werden, oder wie die Statecharts durch die Aktivitäten beeinflußt werden. Es werden hauptsächlich die internen Bestandteile selbst behandelt. Die Dokumentation beschreibt u.a wie die States organisiert sind, wie die einzelnen Verhaltensstufen dargestellt werden, und wie die einzelnen Elemente im Statechart - Konzept benutzt werden, insbesondere Events, Conditions, Actions. Die Darstellungsform der Statecharts ist ähnlich der State-Transition Diagramme (einfache Diagramme in STATEMATE, die ausschließlig aus Zuständen und Zustandsübergängen bestehen), enthält jedoch viele Bestandteile, wie z.B. Hierarchien und Ausdrücke. Wie bei den Activitycharts gibt es auch Darstellungsweisen für die Elemente, die in den Statecharts vorkommen. Die einzelnen Details werden im folgenden beschrieben.

Die Vorgehensweise ist wie folgt festgelegt worden:

Zuerst haben wir die einzelnen (wichtigsten) Konzepte, die Teile der STATEMATE - Sprache (und Teile der Statecharts) sind, ausgewählt (in Anlehnung an unsere Konzepte des dynamischen Modells aus der OMT-Feature-Liste).

Im folgenden werden die einzelnen Konzepte beschrieben und mit ihren grafischen Symbolen (aus STATEMATE bzw. OMT, da hier keine allzu großen Unterschiede in der Notation existieren) dargestellt. Es ist hier anzumerken, daß wirklich die wichtigsten Konzepte der Statecharts (bzw. des dynamischen Modells) auserwählt wurden. Des weiteren sei angemerkt, daß bei dieser Ausarbeitung nicht die grafischen Symbole (also deren Notation) im Vordergrund stehen. Es geht hier vielmehr um die Konzepte, die verwendet werden sollen. Daher sind die auserwählten Konzepte bindend. Im Gegensatz dazu sind die grafischen Symbole der einzelnen Komponenten nach dem eigenen Ermessen der Implementierungsgruppe gestaltbar, und daher nicht bindend. Die angegebenen grafischen Symbole sollen hier lediglich dazu dienen, daß man für das weitere Vorgehen (der Modellierung der Verhaltenssicht des SIGN-Editors deSIGN) schon mal eine Orientierung hat und die Beschreibungen der Konzepte auch besser verstehen kann.

5.2.1.2 Die einzelnen Konzepte mit Beschreibung und grafischem Symbol

State (Zustand) Zustände kann man als Menge der Attributwerte zu einem Zeitpunkt verstehen. Der Zustand eines Objektes stellt die kumulierten Ereignisse seines Verhaltens dar. Wenn zum Beispiel ein Telefon zum ersten mal installiert wird, befindet es sich im Status unbeschäftigt, d.h. daß das vorhergehende Verhalten nicht von großem Interesse ist, und daß das Telefon nun in der Lage ist, Anrufe durchzuführen oder anzunehmen. Wenn jemand den Hörer abnimmt, sagen wir, das Telefon ist jetzt abgehoben und befindet sich im Wahlstatus. Zustände repräsentieren Zeitintervalle. Ein Zustand trennt zwei Ereignisse (Events). Das Symbol für einen Zustand ist ein Oval mit dem Zustandsnamen darin (sieheAbbildung 5.10).

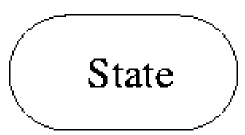


Abbildung 5.10: State

Event (Ereignis) Ein Ereignis ist etwas, daß zu einem bestimmten Zeitpunkt eintritt. Zum Beispiel, Flug 123 hebt am 01.10.97 um 9:00 Uhr in Bremen ab. Ein Ereignis verbindet ein Sendeobjekt mit einem Empfängerobjekt, gegebenfalls werden Informationen von einem Objekt zum anderen Objekt gesendet.

Dies geschieht unidirektional, d.h. ohne Ereignisrückgabe. Im Idealfall sollen Ereignisse Zeitpunkte repräsentieren, also ohne Zeitausdehnung. Das Ereignis ist ein Geschehen, dessen Auftreten eine Veränderung eines Systemstatus bewirken kann.

Falls kein kausaler Zusammenhang zwischen dem Auftreten der Ereignisse vorliegt, nennt man dies nebenläufige Ereignisse. Zum Beispiel, Flug 123 muß in Chicago abheben, bevor es in San Francisco landen kann. Ereignisse repräsentieren also einen Zeitpunkt. Ein Ereignis trennt zwei Zustände. Ereignisse werden an die Transitionspfeile geschrieben (sieheAbbildung 5.11). Mehrere Ereignisse können mittels „AND“ bzw. „OR“ verknüpft werden.

Abbildung 5.11: Event

Statechart (State Diagram = Zustandsdiagramm) Zustandsdiagramme werden einfach aus Zuständen und Ereignissen konstruiert (wie es in gewöhnlichen State-Transition-Diagrammen der Fall ist). Eine Zustandsänderung durch ein Ereignis wird Transition genannt. Transitionen verbinden zwei Zustände über einen Pfeil (Transitionspfeil). In dieser einfachen Form werden Statecharts auch Or-Statecharts (sieheAbbildung 5.12) genannt. Es ist daher auch möglich Statecharts zu verschachteln, hierarchisieren und nebenläufige Statecharts zu synchronisieren. Diese nebenläufigen Statecharts werden auch And-Statecharts (sieheAbbildung 5.13) genannt.

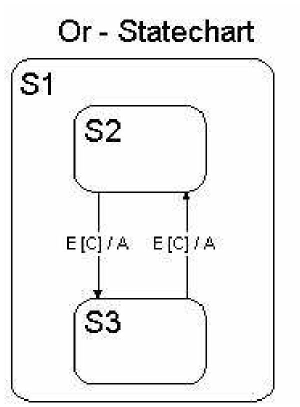


Abbildung 5.12: OR Statechart

Condition (Bedingung) Bedingungen sind boolesche Funktionen auf Objektwerten und Verbindungsattributen. Sie beziehen sich auf Objektwerte, die typischerweise von der Zeit oder anderen Ereignissen abhängig sind, und sind an

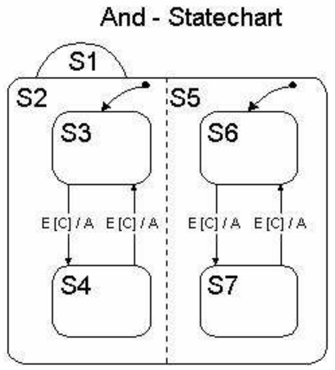


Abbildung 5.13: AND Statechart

Kanten im Zustandsdiagramm (oder Statechart) in eckigen Klammern gebunden. Bedingungen können als Wächter für Transitionen gebraucht werden, da es zu Zustandsübergängen kommt, wenn die Bedingung erfüllt wird. Bedingungen werden in "[]" angegeben und ebenfalls an die Transitionspfeile geschrieben, wobei sie erst nach den Ereignissen angegeben werden (sieheAbbildung 5.14). Mehrere Bedingungen können mittels „AND“, „OR“ verknüpft werden. Die Negation einer Condition kann mittels „NOT“ angegeben werden. Mittels „new(Objektname)“ (sieheAbbildung 5.15) kann die Erzeugung eines neuen Objektes als Bedingung angegeben werden.

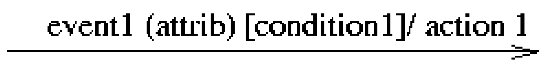


Abbildung 5.14: Condition

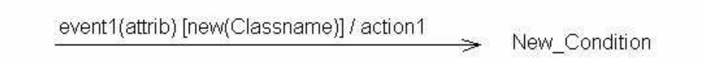


Abbildung 5.15: Condition (neues Objekt)

Action (Operation) Operationen reagieren auf Ereignisse durch die Ausführung von Operationen. Eine Aktion ist eine Operation, die für alle praktischen Anwendungen mit einer Zeitdauer Null definiert wird. Sie verbraucht also im Einsatz keine Zeiteinheit. Eine Aktion stellt normalerweise den Aufruf einer Methode dar, das Auslösen eines anderen Ereignisses oder das Starten oder Beenden einer anderen Aktivität. Die Aktion wird an das Ereignis gebunden. Actions werden mittels "/ action" nach den Ereignissen und Bedingungen angegeben (sieheAbbildung 5.16). Mehrere Actions können mittels ";" verknüpft werden (sieheAbbildung 5.17). Desweiteren kann ein Objekt einem anderen Objekt Nachrichten zusenden. Dieses erfolgt mittels „send“ (sieheAbbildung 5.20). Mit „delete(Objektname)“ kann das Objekt, das vorher mittels

„new(Objektname)“ erzeugt wurde, gelöscht werden (sieheAbbildung 5.18). Des weiteren gibt es eine Assign_Action, was im Klartext eine Zuweisung von Werten (Terme, Faktoren, Ausdrücke, also Conditions) an eine Variable ausdrückt (sieheAbbildung 5.19).

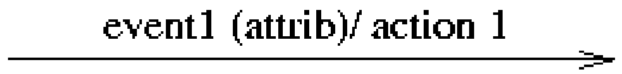


Abbildung 5.16: Action

Abbildung 5.17: Action Verknüpfung

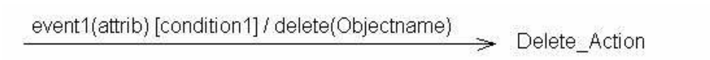


Abbildung 5.18: Action Delete

5.2.1.3 Notation in EBNF-Form

In enger Anlehnung an das Buch „Compilerbau“ von N. Wirth [Wir86]hat das Projekt eine Notation in EBNF - Form erstellt.

In der EBNF - Form werden die strukturellen Zusammenhänge der ausgewählten Konzepte dargestellt und die einzelnen Beziehungen zueinander verdeutlicht.

Sie legt die formelle Syntax für die einzelnen Komponenten fest, was insbesondere für Konzepte, die in unserer SIGN - Modellierung (dem grafischen Editor) eine textuelle Darstellung erfordern, von wichtiger Bedeutung ist (wie z. B. für Action).

Im folgenden wird die Notation in EBNF-Form dargestellt:

state = n_ident simple_state — sc.

simple_state = *Wird in Form einer Klasse symbolisiert*

sc = and_sc — or_sc.

and_sc = sc {sc}.

or_sc = init_transition {state} {transition}.

transition = {start_state} {end_state} [event_expr] [”[” condition ”]”] [”/” action].

start_state = state.

end_state = state.

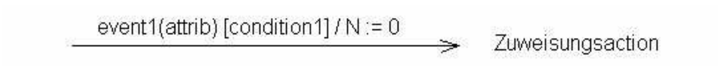


Abbildung 5.19: Action Assign

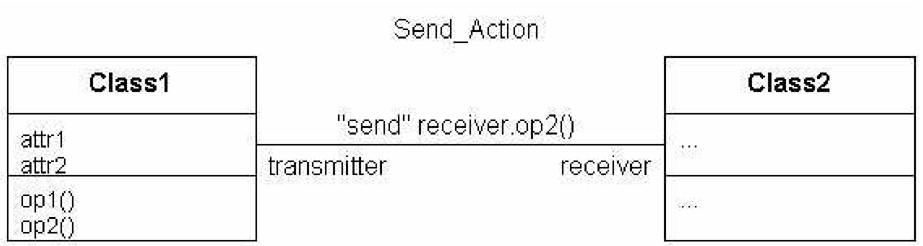


Abbildung 5.20: Action Send

event_expr = conj_event_expr [”OR” event_expr].

conj_event_expr = simple_event_expr [”AND” conj_event_expr].

simple_event_expr = n_ident [”(” v_ident { ”,” v_ident } ”)”].

condition = ”NOT” condition — conj_cond [”OR” condition].

conj_cond = comparison [”AND” conj_cond].

comparison = expr [(”=” — ”#” — ”<” — ”<=” — ”>” — ”>=”) conj_cond].

expr = [”+” — ”-”] term {(”+” — ”-”) term }.

term = factor {(”*” — ”/”) term }.

factor = v_ident — literal — ”(” condition ”)” — ”new (” n_ident ”)”.

action = simple_action [”;” action].

simple_action = assign_action — send_action — delete_action.

assign_action = [v_ident ”:=”] condition.

send_action = ”send” message.

message = link_expr ”.” simple_event_expr.

link_expr = link_ident.

delete_action = ”delete (” n_ident ”)”.

init_transition = end_state [”/” action].

5.2.1.4 Metamodell für Statecharts: Übertragung der EBNF-Form in einen Syntax-Baum

Im nächsten Schritt haben wir daraufhin die Festlegungen (formelle Syntax) der EBNF-Form in entsprechende Syntax-Bäume (Metamodell für Statecharts) übertragen.

Letztlich ergeben sich vier Syntax - Bäume, die in ihrem jeweiligen Abschnitt die Komponenten der Statecharts beinhalten.

Im einzelnen sind es die vier Bäume, in denen folgende Konzepte behandelt und erklärt werden:

- Syntax - Baum für States (+ Transitions)

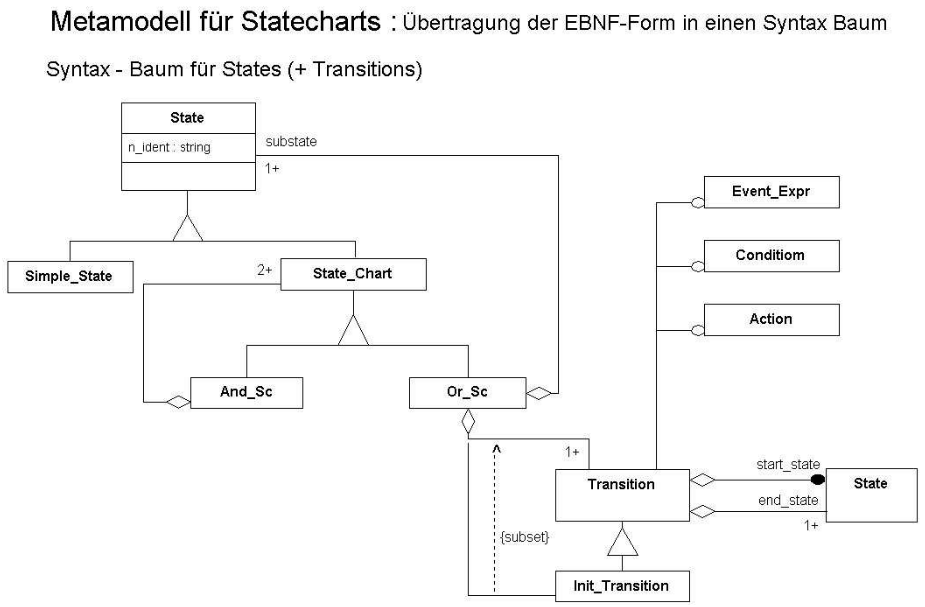


Abbildung 5.21: Syntaxbaum States

Interpretation des Syntax - Baumes (Abbildung 5.21):

Ein Zustand state ist zunächst ein allgemeiner Zustand, der einen Namen (n_ident) hat. Ein State kann zum einen Simple_State sein, also einen einfachen Zustand beschreiben. Zum anderen kann ein State auch State_Chart sein. Hierbei gibt es wiederum zwei Möglichkeiten: Einerseits kann es ein And_State_Chart sein. Dies bedeutet, daß zwei oder mehrere State_Charts vorhanden sind. Andererseits kann es Or_State_Chart sein. Es sind dann ein oder mehrere States vorhanden, wobei Or_State_Charts Unterzustände (substate) der States sind. States können demnach hierarchisch geschachtelt sein. Or_State_Charts enthalten ein oder mehrere Transitions. Eine Transition kann evtl. eine Init_Transition sein, was bedeutet, daß sie die Start-Transition ist und daher nur einmal vorkommt. Sie ist Untermenge (subset) der Transition. Transitionen können 0 (bei der Init_Transition) oder mehrere Start-States, sowie ein oder mehrere End-States haben. Außerdem haben sie 0 oder eine Event_Expr, 0 oder eine Condition und 0 oder eine Action.

- Syntax - Baum für Events (+ Event - Condition Beziehung)

Interpretation des Syntax - Baumes (Abbildung 5.22):

Eine Event_Expr kann ein einzelnes Event sein. Dies wird durch Event_Access beschrieben, wo ein Event enthalten ist. Die Composite_Event_Expr besteht

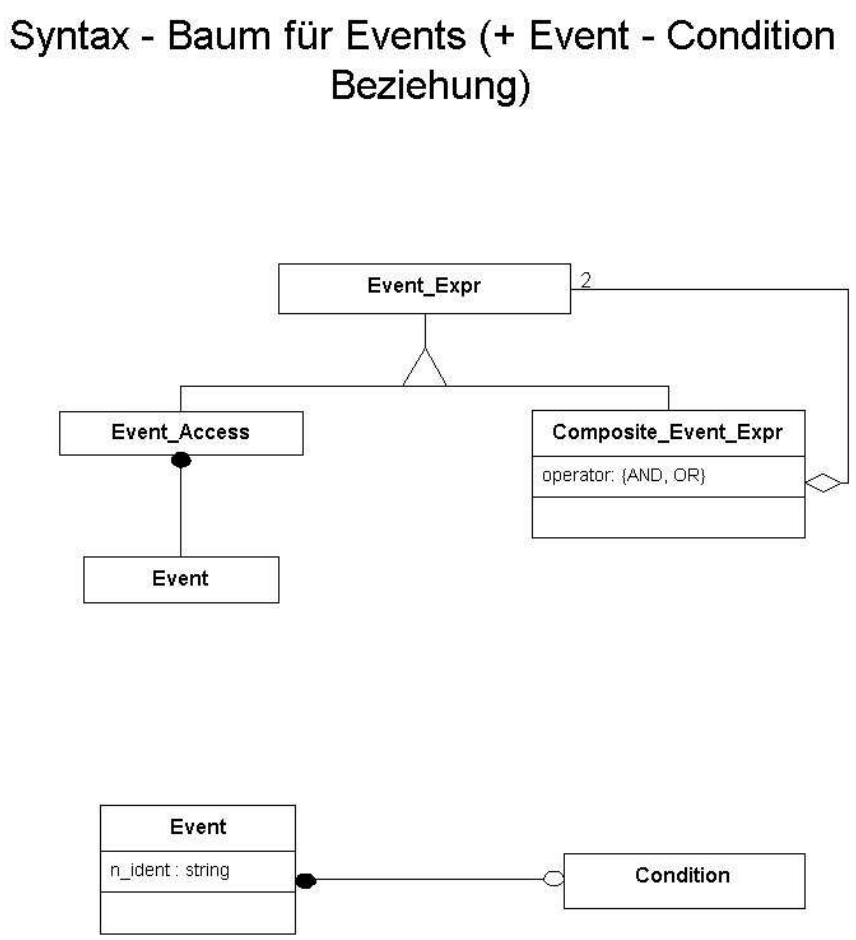


Abbildung 5.22: Syntaxbaum Events

aus zwei Event_Expr, sie enthält eine Verknüpfung für die Ereignisse mit dem Operator AND bzw. OR. Ein Ereignis Event hat einen Namen (n_ident) und kann ohne, oder verbunden mit einer Condition auftreten.

- Syntax - Baum für Conditions

Interpretation des Syntax - Baumes (Abbildung 5.23):

Für die Art einer Condition gibt es zwei Möglichkeiten, die operatorabhängig sind. Es gibt die Mo_Cond. Sie ist einstellig, da sie nur den NOT-Operator für die Bedingung enthält. Zum zweiten gibt es die Dy_Cond. Sie enthält die anderen, mehrstelligen Operatoren, die entsprechend angeordnet werden und mit denen Verknüpfungen möglich sind. Conditions enthalten entsprechende Ausdrücke Expr, die nach dem gleichen Prinzip strukturiert sind. Die Mo_Expr enthält die beiden Vorzeichenoperatoren, die Dy_Expr enthält die anderen Operatoren, die die Ausdrücke zu Termen machen. Die Ausdrücke enthalten wiederum Faktoren Factor. Man unterscheidet folgende Möglichkeiten: Es gibt Variablen Var, die über Var_Access festgelegt werden. Faktoren können Buchstaben (Literal), Paren-Factor für Bedingungen, In-Factor für Zustände oder New-Factor in einer Klasse, verbunden mit einer Variable, sein. Die Variable wird noch einmal differenziert betrachtet. Sie hat als Local_Var einen Typ, sowie Attribute und Parameter.

- Syntax - Baum für Actions (+ Vars)

Interpretation des Syntax - Baumes (Abbildung 5.24):

Eine Action kann aus zwei oder mehreren Actions zusammengesetzt sein. Dann ist sie eine Composite_Action. Falls sie eine einfache Action ist, nennt sie sich Simple_Action. In diesem Fall kann sie Assign_Action sein, und damit eine Zuweisung enthalten, in der Condition und Var_Access enthalten sind. Sie kann auch als Send_Action Nachrichten verschicken, die eine Link_Expr mit einem Rollennamen enthalten; zudem eine Event_Expr. Als letztes besteht auch die Möglichkeit zum Löschen eines n_ident mittels der Delete_Action, wiederum verbunden mit Var_Access.

Durch die Notation in EBNF - Form und die zugehörigen Syntax - Bäume ergibt sich also eine vollständige und übersichtliche Darstellungsform, die die Konzepte der Statecharts mit ihren einzelnen Bestandteilen gut darstellt.

5.2.2 Activity-Charts

5.2.2.1 Einleitung

Die Sprache von Activity-Charts in Statemate beschreibt die funktionale Sicht eines Systems. Die funktionale Beschreibung eines Systems gibt genau die Fähigkeiten dieses Systems an. Sie gibt detaliert die Funktionen oder Activities an, die das System zum Ausführen fähig ist, und die Informationen, die zwischen diese Funktionen fließen. Dies wird in Zusammenhang mit Systemumgebung durchgeführt und nicht etwa in physikalische und architekturelle Aspekte eines Systems, und noch weniger in der dynamische Verhaltensprobleme.

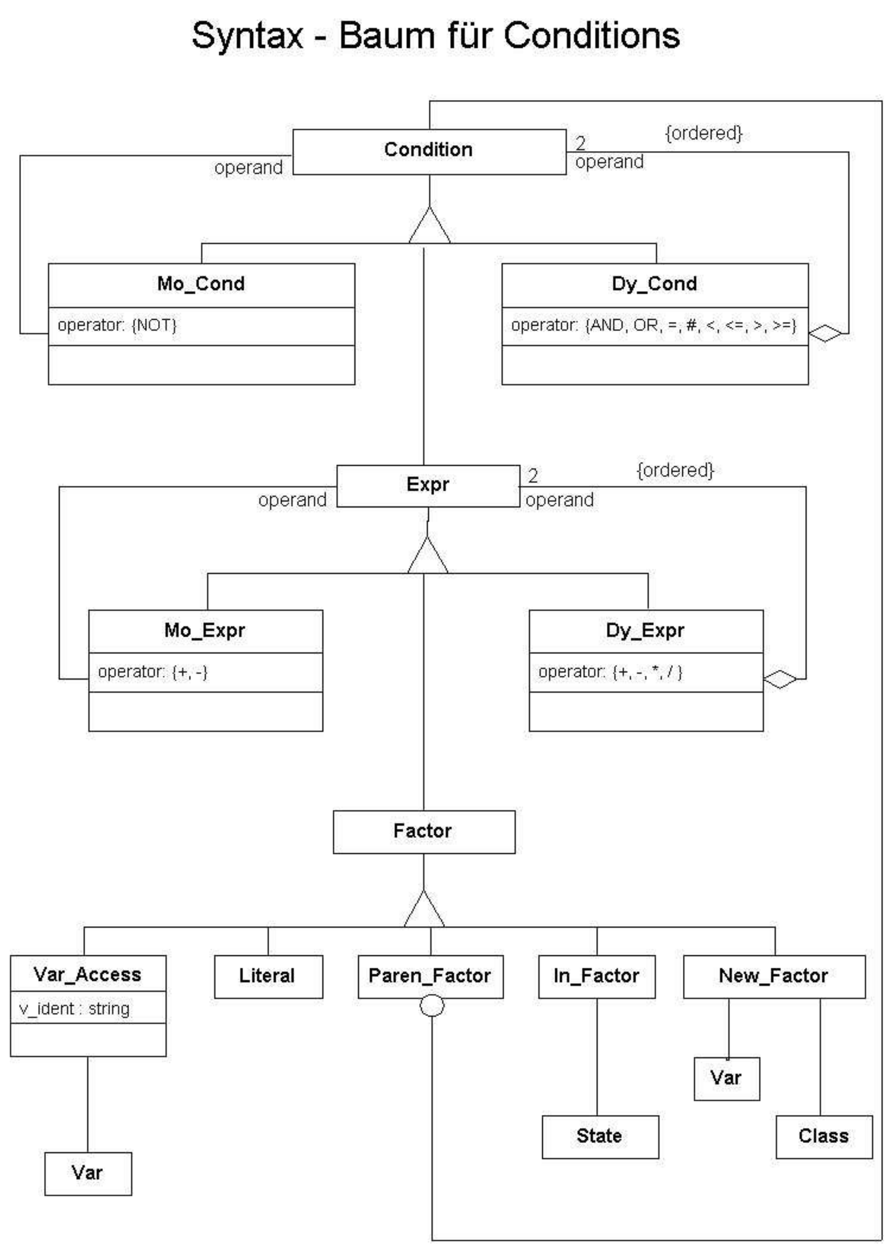


Abbildung 5.23: Syntaxbaum Conditions

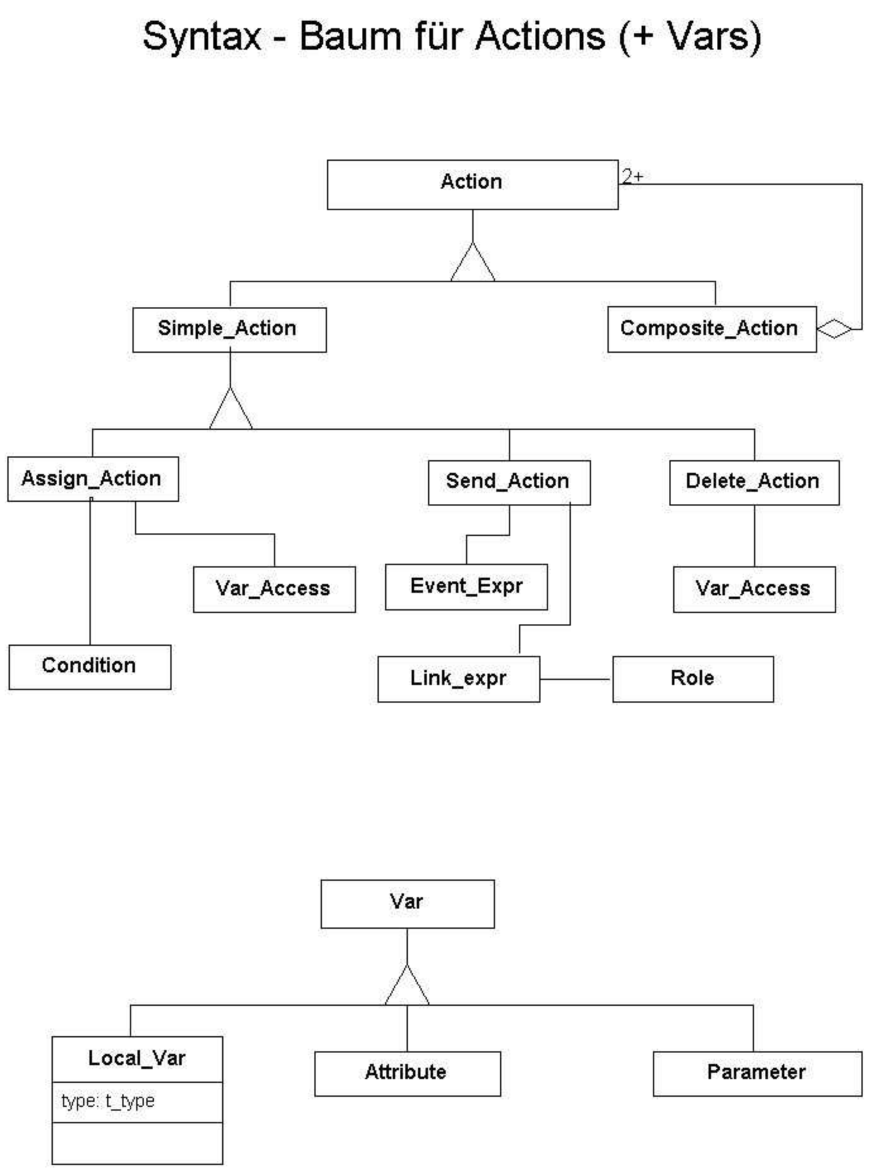


Abbildung 5.24: Syntaxbaum Actions

5.2.2.2 OMT-Konzepte

Wir werden hier die uns vertraute Featureliste von OMT mit den Konzepten von Statemate erweitern. Zur Erinnerung sind hier die Konzepte von OMT nochmal wiederholt:

Prozeß (Process):

Prozesse sind Transformationen, sie führen Funktionen auf Eingabewerten durch und berechnen Ausgaben.

Akteure (Actors):

Produzierende Actors sind Objekte aus dem Objektmodell,die die Werte in das Datenflußdiagramm liefern. Sie stehen immer an den Rändern des Datenfluß diagramms. Das Gegenstück sind die konsumierenden Actors. Sie entnehmen Daten aus dem Datenflußdiagramm.

Datenspeicher (Data Store):

wie bei STATEMATE

Datenfluß (Data Flow):

wie bei STATEMATE

Kontrollfluß (Control Flow):

wie bei STATEMATE

5.2.2.3 Die ausgewählten Konzepte fü Activity-Charts

Die Konzepte, die von den Activity-Charts übernommen werden, sind die folgenden:

Activity-Chart (Subactivity):

Um ein Multi-Level-Datenflußdiagramm zu ermöglichen, das über hierarchische Ebenen hinweg durcharbeitet.

Simple Activity:

Um eine Low-Level Basis zu garantieren.

Control Activity:

Eine Überprüfung der Terminierung oder Determinierung.

Pseudo Activity:

Um sich konkrete Ausschnitte aus Ebenen verdeutlichen zu können.

5.2.2.4 Das Meta-Modell grafisch dargestellt

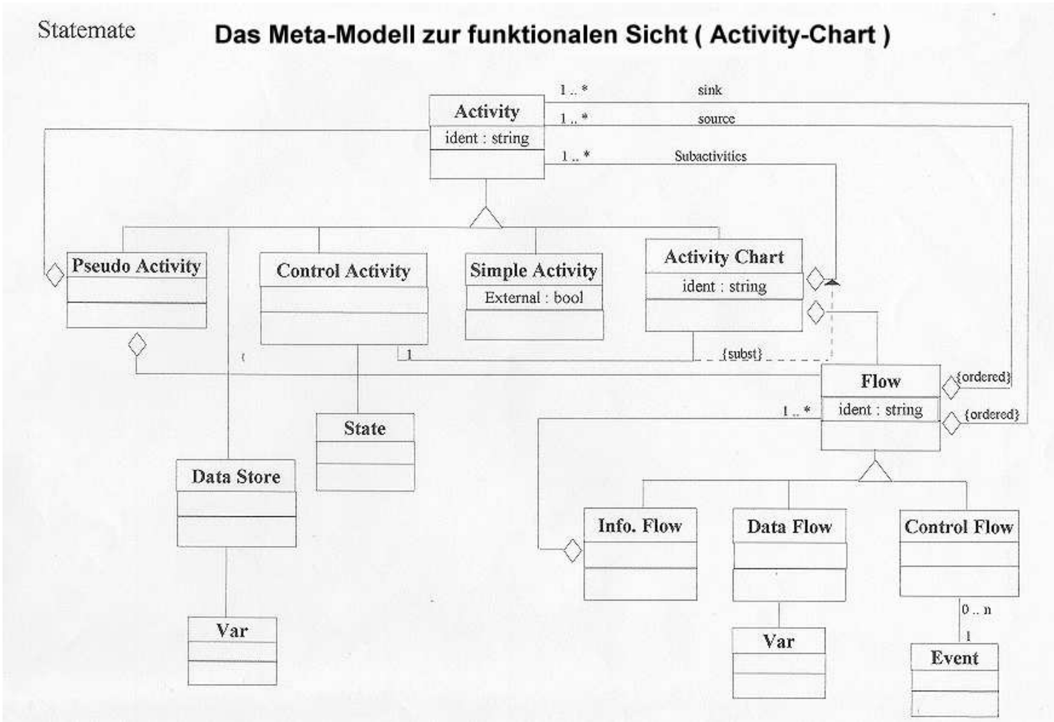


Abbildung 5.25: Metamodell Activity-Charts

5.2.2.5 Die einzelne Komponente der grafischen Darstellung des Meta-Modells

Activity

Das grundlegende Konzept in Statemate ist die Modellierung eines reaktiven Systems durch Activities, deren Verhalten durch verallgemeinerte endliche Automaten beschrieben wird. Activities sind Funktionen oder Prozesse, die das System beschreiben. Durch eine Hierarchie von Activities eingebettet in der Systemumgebung werden die Fähigkeiten einer Systemspezifikation beschrieben; z.B. welche Art von Informationen werden durch Activities manipuliert, wie diese Informationen fließen und wie sie gespeichert werden (sieheAbbildung 5.26).

Activity-Chart

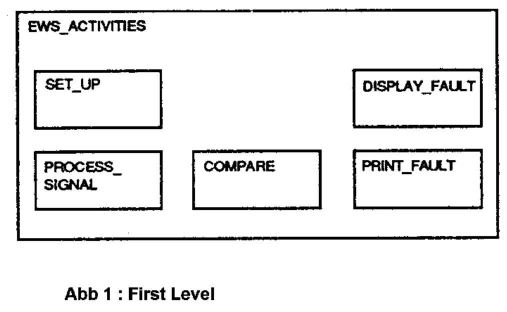


Abbildung 5.26: Activity

Activiy-Charts beschreiben die funktionelle Sicht auf dem System. Man kann sie als ein Multi-Level-Datenflußdiagramm verstehen, welches die Input und die Output der Daten beschreibt. In einem Activity-Chart kann keine Datenmanipulation durchgeführt werden, dies ist nur in State-Charts möglich. Ein Activity-Chart kann mehere Activities beinhalten, einige Activities können so lange in ihre Subactivities zerlegt werden, bis die Basis der Activities erreicht ist. Diese nennt man dann Simple-Activity. Die Activity-Charts werden durch Rechtecke oder Boxen dargestellt (sieheAbbildung 5.27).

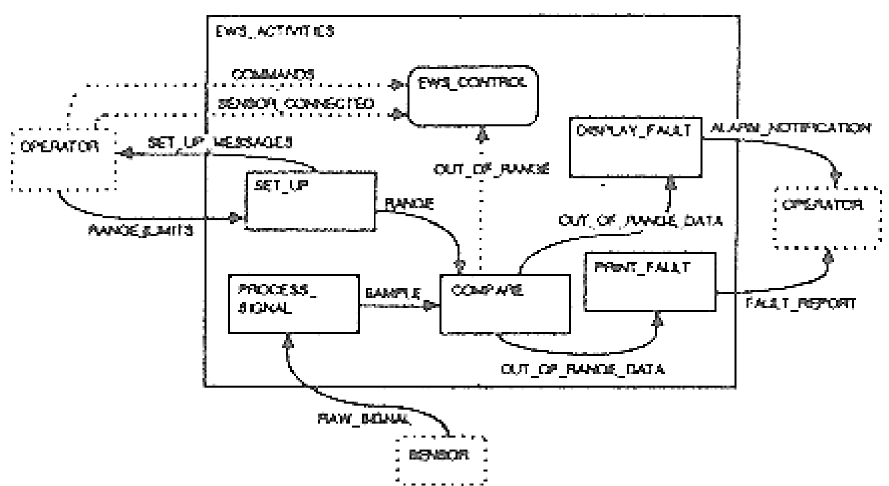


Abbildung 5.27: Activity-Chart

Simple-Activity

Wie erwähnt, die Simple-Activities sind solche Activities, die keine Subactivities enthalten. Sie sind die kleinste Komponente eines Activity-Chart. Die externen Activities, die die Umgebung eines Activity-Chart ausmachen, sind auch solche

Simple-Activities, da sie keine Subactivity beinhalten. Die externen Activities werden im Gegensatz zur (Simple-)Activities durch gestrichene Rechtecke dargestellt.

Control-Activity

Jedes Activity-Chart enthält ein Control-Activity und kann keine Subactivities haben. Ein Control-Activity kontrolliert die Funktionen anderer Activities und entscheidet über ihre Funktionalität. Ihre Spezifikation ist durch State-Charts beschrieben, die die grafische Sprache für die Modellierung des Verhalens eines Systems ist. Die Control-Activities werden durch ein abgerundetes Rechteck dargestellt (sieheAbbildung 5.28).

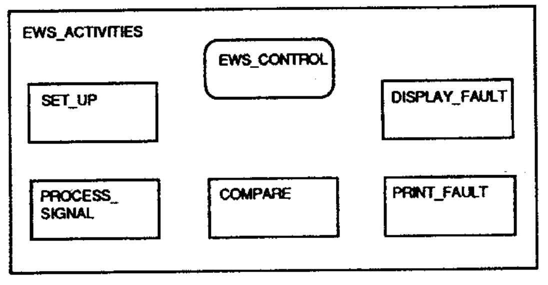


Abb 4 : Ein Contol Activity in einem Activity-Chart

Abbildung 5.28: Control-Activity

Pseudo-Activity

Das Pseudo-Activity läßt zu, daß verschiedene Gruppierungen von Activities möglich wird. Ohne Pseudo-Activity würden Activities sehr starr sein. Sie haben auch keine Beziehung zu State-Charts.

Data-Store

Es gibt keinen begrenzten Zeitraum, in dem die Informationen auf den Flusslinien bleiben können. Daten, die durch ein Quell-Activity produziert sind, bleiben immer für die Ziel-Activity vorhanden, wenn auch die Quell-Activity nicht mehr aktiv ist. In diesem Sinne kann eine Flusslinie als eine Art von Speicherplatz betrachtet werden. Dennoch ist es sinnvoller ein explizites Data-Store im Diagramm einzufügen, um Informationen für spätere Benutzung zu speichern. Ein Data-Store kann übrigens zur Angabe einer Aggregation von größeren Mengen von Daten, die über einen längeren Zeitraum dauernd hineinfliessen, benutzt werden.

Ein Data-Store ist also ein passives Activity, und zwar ein solches, das keine Informationen ändert oder produziert. Die Data-Stores können keine anderen Data-Stores oder Activities beinhalten. Ein Data-Store wird durch ein Rechteck mit gestrichelten vertikale Kanten repräsentiert (sieheAbbildung 5.29).

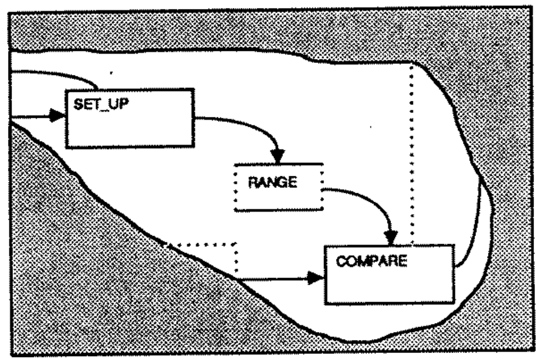


Abb 5 : Data Store

Abbildung 5.29: Data-Store

Flow

Wir benutzen das Wort „Fluß“ (eng. Flow) zur Übertragung der Informationen zwischen Activities. Die Klasse „Flow“ ordnet jedem Activity ein Fluß zu. Der Name eines Flusses kann ein einzelnes Informationselement sein (wie data-item, condition oder event), das durch Linien fließt, oder eine Gruppierung solcher Informationselemente sein, die wir „Information-flow“ nennen. Grundsätzlich kann man zwei unterschiedliche Typen von Flußlinien zulassen (sieheAbbildung 5.30):

Data-Flow-Linien, die durch einen durchgestrichenen Pfeil gekennzeichnet sind und Control-Flow-Linien, die durch einen gestrichenen Pfeil gemalt sind. Typischerweise beinhalten die Control-Flow-Linien Informationen oder Signale, die zu Kontrollentscheidungen führen, während Data-Flow-Linien Informationen zu Berechnungen und Datenprozessoperationen beinhalten.

Es gibt mehrere Typen von Informationselementen, die zwischen Activities fliessen können: Drei davon, events, conditions und data-items repräsentieren einzelne Informationsflüsse. Das vierte, information-flow, repräsentiert eine Anzahl von mehreren Flüssen, da sie durch den selben Pfeil fließen. Eine Information-flow kann als Komponente wiederum andere Flüsse beinhalten, die vielleicht wieder in Information-flow zerlegt werden, oder in bestimmte Informationselemente zersplittet werden (dataitems, conditions und events). Event und condition werden durch Kontrollflüsse fü Kontrollzwecke eingesetzt. Ein Data-Item ist hingegen eine Informationseinheit, die verschiedene Typen von Werten annehmen kann, also eine Variable.

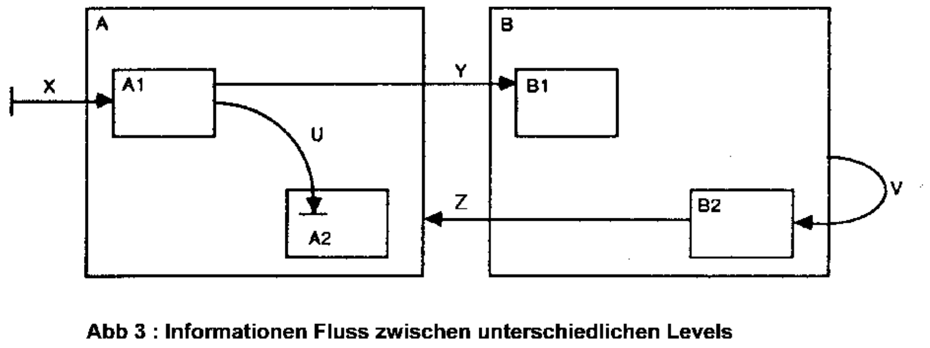


Abbildung 5.30: Flow

5.2.3 Constraints

5.2.3.1 Einleitung

Dieser Bericht stellt eine Object Constraint Language (OCL) für eine mögliche deSIGN-Erweiterung vor und definiert diese, basierend auf Object Constraint Language Specification (V.1.1) [BRJ97] für UML (Unified Modeling Language). Für das Modellieren von Systemen kann OCL verwendet werden, um Constraints und andere Ausdrücke zu spezifizieren. Darüber hinaus stellt dieser Bericht einen groben Entwurf einer OCL-Unterstützung im deSIGN-Projekt vor.

Was ist OCL ?

Beim objektorientierten Modellieren mit Hilfe einer graphischen Sprache (wie zum Beispiel UML) sind graphische Notationen oft nicht ausreichend, um eine vollständige, präzise und eindeutige Spezifikation von bestimmten Sachverhalten zu erhalten. Offensichtlich ist es notwendig, zusätzliche Bedingungen über Objekte in einem Modell zu beschreiben. Damit diese Beschreibungen eine einheitliche Notation und eine eindeutige Bedeutung bekommen, verwendet man OCL.

OCL ist eine reine Ausdruckssprache. Demnach garantieren OCL-Ausdrücke frei von Seiteneffekten zu sein, d.h. sie erzeugen auch keine. Für die Auswertung von OCL-Ausdrücken wird (theoretisch gesehen) nichts im System geändert. Im System werden keine Zustände, Werte noch andere Objekte verändert. OCL-Ausdrücke liefern bei der Auswertung lediglich einen Wert zurück.

OCL ist keine Programmiersprache. Man kann keine Prozesse oder Operationen und Methoden mit Seiteneffekten mit Hilfe der OCL-Ausdrücke aktivieren. Da OCL hauptsächlich eine Modellierungssprache ist, müssen nicht für alle Elemente des Ausdruckes eine direkte Auswertung möglich sein.

OCL ist eine getypte Sprache. Demnach hat jeder OCL-Ausdruck einen Typ. In einem korrektem OCL-Ausdruck sind alle Typen, die verwendet werden, wohl definiert. Generell sind alle Typen in OCL auch Klassifizierer innerhalb des UML-Modells.

Da OCL eine Modellierungssprache ist, wird jeder OCL-Ausdruck konzeptionell atomar betrachtet. Während der Auswertung kann kein Zustand irgend eines Objektes im System geändert werden.

5.2.3.2 Einführung

Notation

Eingerückte Zeilen sind OCL-Audrücke:

```
'Dies ist ein OCL-Ausdruck'
```

Die Begriffe, die vor einem Ausdruck als Überschrift eingerückt sind, bestimmen den Kontext des Ausdrucks:

```
TypName
  'dies ist ein OCL-Ausdruck im Kontext von TypName'
```

Generell werden OCL-Ausdrücke nur in ASCII-Zeichen geschrieben.

Das Diagramm inAbbildung 5.31 [BRJ97] wird in diesem Bericht als Beispiel dienen.

5.2.3.3 Verbindung mit dem UML Metamodell

Self

Jeder OCL-Ausdruck wird im Kontext einer Instanz von einem bestimmten Typ geschrieben. In einem OCL-Ausdruck wird der Name self benutzt, um sich auf die kontextuelle Instanz zu beziehen.

Invarianten

OCL-Ausdrücke können einen Teil von Invarianten bilden, welches die stereotypische Rolle von Constraints ist. Wenn eine Invariante mit einem geeigneten Klassifizierer des Systems assoziert ist, d.h. wenn ein wohl definierter Typ verwendet wird, muß der Ausdruck des entsprechenden Typs zu jeder Zeit true für jede Instanz ergeben.

Im Kontext von Company (siehe Beispieldiagramm) bezieht sich self auf eine Instanz Company. Im Ausdruck:

```
self.numberOfEmployees
```

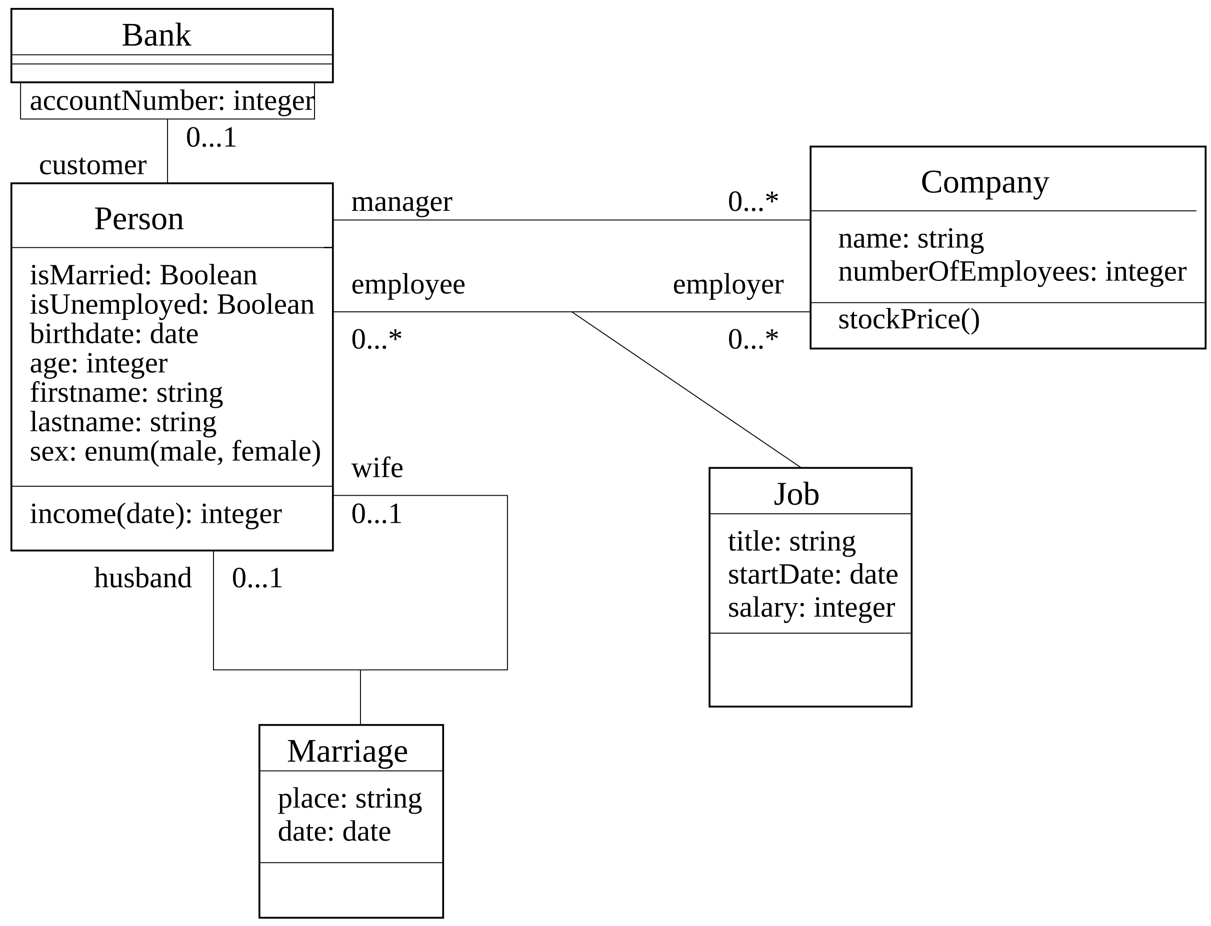


Abbildung 5.31: Beispiel eines Klassendiagramms

ist self eine Instanz vom Typ Company. Man kann self als Objekt betrachten, von dem aus der Ausdruck startet.

Ein Typ der kontextuellen Instanz eines OCL-Ausdruckes, welches auch eine Invariante ist, wird folgendermaßen formuliert:

```
Company
  self.numberOfEmployees
```

In vielen Fällen kann man self weglassen, da wie im obigen Beispiel der Kontext klar ist.

Als Alternative für self, kann ein anderer Name, der der Rolle von self entspricht, definiert werden, was sinnvoll bei Vergleichen von verschiedenen Instanzen des gleichen Typs ist:

```
c : Company
  c.numberOfEmployees
```

Dies ist identisch mit dem vorherigem Beispiel, das self benutzt.

Vor- und Nachbedingungen

OCL-Ausdrücke können Vor- und Nachbedingungen bilden, die stereotypisch zu den Constraints der Vor- und Nachbedingungen sind. Falls eine Vor- oder Nachbedingung auf einer Operation oder Methode formuliert wird, d.h. der Ausdruck ist eine Vor- oder Nachbedingung auf dieser Operation oder Methode, ist die kontextuelle Instanz self von dem Typ, welches die Operation oder Methode besitzt. Die Vor- und Nachbedingungen werden mit den Schlüsselwörten pre und post gekennzeichnet. Die betroffene Operation bildet den Kontext:

```
Typname::operationsName(parameter1 : Typ1, ...):RückgabeTyp
  pre : parameter1 = ...
  post: result = ...
```

Der Name self kann im Ausdruck benutzt werden, um sich auf das Objekt zu beziehen, von dem aus die Operation aufgerufen wird. Der Name result ist der Name für das Rückgabeobjekt, falls eines existiert. Die Namen der Parameter (parameter1, ...) können ebenfalls verwendet werden, wie in diesem Beispiel:

```
Person::income(d : Date) : Integer
  post: result = ...irgendeine Funktion von
                 self mit parameter1...
```

Allgemeine Ausdrücke

Jeder OCL-Ausdruck kann als Wert für ein Attribut der UML-Klassen dienen, da alle im UML-Modell definierten Klassen und Typen in OCL als Typen verwendet werden. Demnach können die OCL-Werte auch als Attribute von entsprechenden UML-Subklassen verwendet werden. In diesem Falle beschreibt die semantische Bedeutung des Modells diese Ausdrücke.

5.2.3.4 Typen

In OCL sind bereits einige Basistypen definiert, die zu jeder Zeit zur Verfügung stehen. Diese vordefinierten Wertetypen sind unabhängig von jedem Objektmodell und sind Teil der Definition von OCL.

Einige Basistypen werden in den Beispielen verwendet. Diese sind mit ihren zugehörigen Werten:

Typ	*Wert*
Boolean	true, false
Integer	1, 2 ,34, 26524, ...
Real	1.5, 3.14, ...
String	'Sein oder nicht sein...'

OCL definiert einige Operationen auf vordefinierten Typen. Die folgende Tabelle gibt einige Beispiele der Operationen auf den vordefinierten Typen.

Typ	*Operationen*
Integer	*, +, -, /, abs
Real	*, +, -, /, floor
Boolean	and, or, xor, not, implies, if-then-else
String	toUpper, concat

Collection, Set, Bag und Sequence sind ebenfalls Basistypen. Ihre Beschreibung wird in den darauffolgenden Kapiteln behandelt.

Typen vom UML-Modell

Jeder OCL-Ausdruck wird im Kontext eines UML-Modells oder vergleichbar (z.B. deSIGN-Modell) geschrieben, das aus einer Anzahl von Typen/Klassen, ihren Operationen und Assoziationen sowie Generalisierungen besteht. Alle Typen/Klassen vom UML-Modell sind Typen in OCL, das zu dem Modell zugehörig ist.

Enumerationstypen

Wie im Beispieldiagramm gezeigt, können neue Enumerationstypen im Modell definiert werden, indem man:

```
enum {value1, value2, value3}
```

benutzt. Die Werte der Enumeration (value1, ...) können innerhalb eines Ausdruckes verwendet werden.

Da es zu möglichen Namenskonflikten zwischen Attributnamen und Enumerationswerten kommen kann, werden die Enumerationswerte syntaktisch mit dem zusätzlichem Symbol # davor verwendet:

```
#value1
```

Typkonformität

OCL ist eine getypte Sprache, deren Basiswerttypen in einer Typhierachie geordnet werden. Diese Hierachie bestimmt die Konformität der verschiedenen Typen zueinander.

Ein OCL-Ausdruck, in dem alle vorkommenden Typen zueinander passen, ist ein zulässiger Ausdruck. Demnach bilden nichtpassende Typen einen ungültigen Ausdruck. Ein Typ typ1 paßt zu einem anderen Typ typ2, wenn eine Instanz von typ1 an jeder Stelle ersetzt werden kann, wo eine Instanz vom typ2 erwartet wird. Die Typkonformitätsregeln für die Typen in einem Klassendiagramm sind:

- Jeder Typ paßt zu seinem Supertyp (Obertyp)
- Typkonformität ist transitiv

Das heißt, dass alle Typen zu ihren Supertypen passen und allen anderen Subertypen darüber.

Die Typkonformitätsregeln für Wertetypen sind:

Typ	*Paßt zu / ist subtyp von*
Set	Collection
Sequence	Collection
Bag	Collection
Integer	Real

Die Relation der Konformität zwischen Collectiontypen hält nur dann, wenn die Typen der einzelnen Elemente der Collection zueinander passen.

Casting

Unter einigen Umständen ist es wünschenswert/erforderlich eine Eigenschaft eines Objektes, die in ihrem Subtyp des aktuell bekannten Typs des Objektes definiert ist, zu nutzen. Da diese Eigenschaft nicht im aktuell bekannten Typ definiert ist, resultiert ein Typkonformitätsfehler, der den Ausdruck ungültig macht.

Falls es sicher ist, daß der aktuelle Typ eines Objektes der Subtyp ist, kann das Objekt retypisiert werden unter der Verwendung der oclAsType(OclTyp) Operation. Die Operation resultiert in dem gleichem Objekt, jedoch ist der bekannte Typ das Argument OclTyp. Falls es ein Objekt object vom Typ typ1 gibt, und typ2 ist ein anderer Typ, so ist folgende Schreibweise erlaubt:

```
object.oclAsType(typ2)
```

Dieses wird zu einem Objekt mit dem Typ typ2 ausgewertet.

Ein Objekt kann nur zu einer seiner Subtypen retypisiert (gecastet) werden, also muß in diesem Beispiel typ2 ein Subtyp von typ1 sein. Falls nicht, d.h. beim Casting sind die beteiligten Typen nicht gleich, ist der Ausdruck undefiniert (siehe Kapitel 4.7).

Operationsprioritäten

in Ocl sind:

- Punkt- und Pfeiloperatoren haben die höchste Priorität
- Unäre ’not’ und das unäre Minus ’-’
- ’*’ und ’/’
- ’+’ und binäre ’-’
- ’and’, ’or’ und ’xor’
- ’if-then-else-endif’
- ’<’,’>’,’<=’,’>=’ und ’=’

Klammern wie ’(’ und ’)’ können verwendet werden, um diese Reihenfolge zu umgehen.

Kommentare

in OCL werden nach zwei Bindestrichen geschrieben. Alles bis zum Ende der Zeile sind dann Kommentare, wie zum Beispiel:

```
-- dies ist ein Kommentar
```

Undefinierte Werte

Wann immer ein OCL-Ausdruck ausgewertet wird, besteht die Möglichkeit, daß ein oder mehrere Elemente des Ausdruckes undefiniert sind. In diesem Fall ist der gesamte Ausdruck undefiniert.

Es gibt zwei Ausnahme zu dieser Regel für boolsche Operationen. Diese sind:

- True mit irgendetwas or-verknüpft ergibt True
- False mit irgendetwas and-verknüpft ergibt False

Die obigen zwei Regeln sind zulässig, gleichgültig in welcher Reihenfolge die Argumente sind und die obige Regel ist zulässig, gleichgültig ob der andere Wert bekannt ist oder nicht.

5.2.3.5 Objekte und Eigenschaften

OCL-Ausdrücke können sich auf Typen, Klassen, Verbindungen, Assoziationen (die als Typen agieren) und Datentypen beziehen. Außerdem können alle Attribute, Assoziationsenden, Methoden und Operationen ohne Seiteneffekte, die definiert sind auf diesen Typen etc., verwendet werden. In einem Klassenmodell ist eine Operation oder Methode definiert als seiteneffektfrei, wenn das isQuery-Attribut dieser Operation true ist. Für diese Zwecke in diesem Bericht bezieht man sich bei Attributen, Assoziationsenden und seiteneffektfreie Methoden auf Eigenschaften. Eine Eigenschaft ist demnach:

- ein Attribut
- ein Assoziationsende
- eine Operation, deren Eigenschaft true ist
- eine Methode, deren Eigenschaft true ist

Eigenschaften

Der Wert einer Eigenschaft von einem Objekt, das in einem Klassendiagramm definiert ist, wird mit einem Punkt gefolgt von dem Namen der Eigenschaft spezifiziert:

```
EinTyp
  self.eigenschaft
```

Wenn self eine Referenz zu einem Objekt ist, ist self.eigenschaft der Wert der Eigenschaft eigenschaft auf self.

Eigenschaften: Attribute

Als Beispiel wird das Alter einer Person wie folgt geschrieben:

```
Person
  self.age
```

Der Wert dieses Ausdruckes ist der Wert des age-Attributes auf der Person self. Der Typ dieses Ausdruckes ist der Typ des Attributes age, welches ein Basistyp Integer ist.

Mit Attributen und Operationen, die auf Basistypen definiert sind, lassen sich Berechnungen über das Klassenmodell ausdrücken. Als Beispiel kann die Regel gelten, daß alle Personen immer älter als null sein sollen. Dies läßt sich durch folgende Invariante formulieren:

```
Person
  self.age >= 0
```

Eigenschaften: Operationen

Operationen können Parameter haben. Wie im vorherigen Beispiel gezeigt hat ein Person-Objekt ein Einkommen als Funktion von dem Datum date. Um auf diese Operation zuzugreifen, für eine Person *einePerson* und ein Datum *einDatum*, schreibt man:

```
einePerson.income(einDatum)
```

Die Operation selber kann durch ein Nachbedingungsconstraint definiert werden. Dies ist ein Constraint, das als Nachbedingung (postcondition) stereotypisiert ist. Das Objekt, welches von der Operation zurückgegeben wird, kann mit dem Namen result referenziert werden. Demnach ergibt sich folgende Form:

```
Person::income (d: Date) : Integer
  post: result = -- irgendeine Funktion über d und anderen
                 -- Eigenschaften von Person
```

Die rechte Seite der Definition kann sich auf die Operation beziehen, die definiert wird. So kann zum Beispiel die Definition rekursiv sein, solange die Rekursion wohl definiert ist. Der Typ von result ist der Rückgabetyp, welcher in diesem Beispiel Integer ist.

Um auf eine Operation ohne Parameter zuzugreifen, werden leere Klammern verwendet:

```
Company
  self.stockPrice()
```

Eigenschaften: Assoziationsenden und Navigieren

Angefangen von einem spezifiziertem Objekt, kann man durch eine Assoziation in einem Klassendiagramm auf ein anderes Objekt und deren Eigenschaften zugreifen. Um dies zu erreichen, benutzt man das gegenüberliegende Assoziationsende:

```
objekt.rollenname
```

Der Wert dieses Ausdruckes ist ein Set von Objekten, die auf der anderen Seite der rollenname-Assoziation sind. Falls die Kardinalität des Assoziationsendes 0..1 oder 1 ist, ist der Wert dieses Ausdruckes ein Objekt. Als Beispiel:

```
Company
  self.manager  -- ist vom Typ Person
  self.employee -- ist vom Typ Set(Person)
```

Die Auswertung des ersten Ausdruckes wird in einem Objekt vom Typ Person resultieren, da die Kardinalität der Assoziation eins beträgt, während der zweite Ausdruck bei der Auswertung einen Set von Personen ergibt. Generell resultiert eine Navigation in einem Set von Objekten. Falls eine Assoziation einer Klasse geordnet ist, resultiert eine Navigation in einer Sequence (Reihe).

Collections wie Set, Bag und Sequences sind vordefinierte Typen in OCL. OCL stellt einige vordefinierte Operationen auf diesen bereit. Um auf eine Eigenschaft dieser Collection zuzugreifen, wird ein Pfeil -> gefolgt von dem Namen der Eigenschaft verwendet. Zum Beispiel:

```
Person
  self.employer->size
```

Dies verwendet die size-Eigenschaft des Set self.employer, welches in der Anzahl von employers von der Person self resultiert. Ein weiteres Beispiel:

```
Person
  self.employer->isEmpty
```

Dies verwendet die isEmpty-Eigenschaft des Set self.employer. Dies ergibt true, wenn der Set von employers leer ist, sonst false.

Fehlende Rollennamen

Wann immer ein Rollenname bei einem Assoziationsende fehlt, wird der Name des Typs des Assoziationsendes verwendet - mit dem ersten Buchstaben kleingeschrieben. Falls dies zu Zweideutigkeiten führt und unbenannte Rollennamen in reflexiven Assoziationen vorkommen, kann man einen Rollenname vergeben. Sonst können diese Rollennamen nicht verwendet werden.

Navigieren über optionale Assoziationen

Da die Kardinalität der Rolle manager eins ist, ergibt self.manager ein Objekt vom Typ Person. Solch ein einzelnes Objekt kann auch als Set verwendet werden. Es verhält sich wie ein Set, das ein einziges Objekt enthält. Die Verwendung als Set wird mit dem Pfeil, gefolgt von der Eigenschaft von Set, durchgeführt. Einige Beispiele:

```
Company
  self.manager->size -- 'self.manager' wird als Set
                      -- verwendet, da der Pfeil be-
                      -- nutzt wird, um auf die Set-
                      -- Eigenschaft 'size' zuzugreifen.
                      -- Der Ausdruck ergibt 1.
  self.manager.age -- 'self.manager' wird als Person
                   -- verwendet, da ein Punkt benutzt
```

```
                      -- wird, um auf die Eigenschaft 'age'
                      -- von Person zuzugreifen
```

Im Falle einer optionalen Assoziation (d.h. Kardinalität 0..1) ist es besonders hilfreich, um zu erfahren, ob ein Objekt exitiert oder nicht. Zum Beispiel:

```
Person
  self.wife->notEmpty implies self.wife.sex = #female
```

Kombinieren von Eigenschaften

Eigenschaften können kombiniert werden, um komplexere Ausdrücke zu konstruieren. Eine wichtige Regel ist dabei, daß ein OCL-Ausdruck immer zu einem bestimmten Objekt oder Typ evaluiert. Auf diesem Ergebnis lassen sich immer wieder weitere Eigenschaften darauf anwenden, so daß sich diese Ausdrücke von links nach rechts lesen und auswerten lassen.

Im folgenden einige Beispiele:

- Verheiratete Personen sind über 18

  ```
  self.wife->notEmpty implies self.wife.age >= 18 and
  self.husband->notEmpty implies self.husband.age >= 18
  ```

- Eine Firma hat maximal 50 Mitarbeiter

  ```
  self.employee->size <= 50
  ```

- Eine Heirat ist zwischen Mann und Frau

  ```
  self.wife.sex = #female and
  self.husband.sex = #male
  ```

- Eine Person kann nicht einen Mann und eine Frau haben

  ```
  not ((self.wife->size = 1) and (self.husband->size = 1))
  ```

Navigation zu Assoziationstypen

Um eine Navigation zu einer Assoziationsklasse zu beschreiben (in diesem Beispieldiagramm Job und Marriage), verwendet OCL einen Punkt und den Namen der Assoziationsklasse mit dem ersten Buchstaben kleingeschrieben:

```
Person
  self.job
```

Dies ergibt ein Set von allen Jobs, die eine Person mit der Company hat, der dort employer ist. In diesem Fall hat die Assoziationsklasse keinen expliziten Rollennamen und folgt den Konventionen wie in dem Paragraphen über Fehlende Rollennamen beschrieben.

Navigieren von Assoziationsklassen

Man kann von Assoziationsklassen aus zu den Objekten, die an dieser Assoziation beteiligt sind, navigieren. Dies wird mit der Verwendung der Punkt-Notation und der Rollennamen der Assoziationsenden erreicht.

```
Job
  self.employer
```

Die Navigation von einer Assoziationsklasse zu einem Objekt der Assoziation wird immer genau ein Objekt zurückliefern. Dieser Effekt resultiert aus der Definition der Assoziationsklasse. Obwohl das Ergebnis genau ein Objekt liefert, kann es trotzdem als Set mit Hilfe vom Pfeil -> verwendet werden.

Navigation durch qualifizierte Assoziationen

Qualifizierte Assoziationen verwenden ein oder mehrere qualifizierte Attribute um die Objekte am anderen Ende der Assoziation auswählen zu können. Um dies zu erreichen, kann der Wert des Qualifizierers in eckigen Klammern stehen. Es ist erlaubt den qualifizierenden Wert auszulassen. In dem Fall werden alle Objekte am anderem Ende der Assoziation zurückgegeben.

```
Bank
  self.customer          -- ergibt ein Set(Person), die
                         -- alle customers der Bank enthält
  self.customer[8764423] -- ergibt eine Person, die die KtoNr
                         -- 8764423 bei der Bank hat
```

Wenn mehrere qualifizierende Attribute existieren, werden diese durch Kommas getrennt. Es ist nicht erlaubt das qualifizierende Attribut nur teilweise anzugeben.

Collections

Ein weiterer vordefinierter Typ in OCL ist der Collection-Typ, der eine Anzahl vorgegebener Operationen besitzt, um dem Benutzer die Möglichkeit zu geben, durch OCL-Ausdrücke Collections zu verändern. Operationen auf Collections ändern niemals die Orginal Collection, sondern das Ergebnis wird in einer neuen Collection abgelegt.

Collection ist ein abstrakter Typ mit den Untertypen Set, Sequence und Bag. Ein Set ist dabei eine einfache Aufzählung von Elementen, wobei Elemente nicht doppelt vorhanden sein dürfen. Ein Bag ist gleich definiert wie ein Set, es darf jedoch doppelte Elemente enthalten (auch mehrmals). Eine Sequence gleicht wiederum einem Bag, die Elemente sind jedoch geordnet.

Hier einige Beispiele:

```
Set { 1 , 5 , 3 , 17 }
Bag { 1 , 5 , 3 , 5 , 1 }
Sequence { 1 , 3 , 5 , 7 , 9 }
```

Wie alle anderen Typen besitzen auch Collections eine Typ-Hierachie und Typ-Konformität.

5.2.3.6 Operationen für Collections

Für den Typ Collections sind in OCL einige Standard-Operationen definiert, um dem Benutzer die Möglichkeit zu geben, aus bestehenden Collections durch Verwendung dieser Operationen neue Collections zu bilden.

Select und Reject

Diese Operationen sind dazu geeignet, um aus einer Collection eine genaue Auswahl an Elementen in eine neue Collection zu überführen.

```
collection ->select ( boolean expression )
```

Als Parameter in der Klammer sind boolsche Ausdrücke erlaubt, so z.B.

```
Company
  self.employee ->select(age > 50)
```

Die Ergebnis-Collection enthält dabei alle Elemente aus der Ursprungs-Collection, für die der boolsche Ausdruck wahr wird. Dabei wird bei allen Elementen der Ursprungs-Collection geprüft, ob der boolsche Ausdruck wahr oder falsch ist. Nur wenn wahr festgestellt wird, wird ein Element in die Ergebnis Collection aufgenommen. Für das obere Beispiel ist self.employee vom Typ Set(Person) und select überprüft alle Personen aus self.employee, ob für diese der Ausdruck age $>$ 50 wahr wird.

Die Reject-Operation ist ähnlich zur Select-Operation, aber man erhält alle Elemente für die der boolsche Ausdruck falsch wird und diese werden in die Ergebnis-Collection aufgenommen. Die Reject-Syntax ist identisch zur Select-Syntax:

```
collection->reject( boolean expression)
```

Als Beispiel werden in diesem OCL-Ausdruck alle Employees ausgewählt, die nicht verheiratet sind:

```
Company
  self.employee->reject ( isMarried)
```

Collect Operation

Wie bei den vorherigen Operationen gezeigt, resultieren diese immer in einer Unter-Collection der Orginal-Collection. Wenn hingegen eine Collection entstehen soll, die aus einer anderen Collection abgeleitet wird, aber verschiedene Elemente aus der Orginal-Collection (jedoch keine Unter-Collection) enthalten soll, so benutzt man die Collect-Operation. Die Collect-Operation benutzt dieselbe Syntax wie select und reject:

```
collection->collect ( expression)
```

Als Beispiel:

```
Company
  self.employee->collect ( birthDate )
```

Als Hinweis ist zu erwähnen, das die Resultats Collection nicht ein Set, sondern ein Bag ist, da auch ein Geburtsdatum doppelt vorkommen kann.

ForAll Operation

Häufig wird ein Constraint für alle Elemente einer Collection benötigt. Hier benutzt man die forAll-Operation in Verbindung mit einem boolschen Ausdruck, der für alle Elemente der Collection gelten muß.

```
collect->forAll ( boolean expression )
```

Der forAll Ausdruck resultiert in einem Boolean, da das Ergebnis wahr wird, wenn der boolsche Ausdruck für alle Elemente der Collection wahr ist. Der gesamte Ausdruck wird falsch, wenn schon ein Element den boolschen Ausdruck nicht erfüllt.

Beispiel:

```
Company
  self.employee->forAll ( Forename='Jack' )
```

Dieser Ausdruck wird wahr, wenn jeder employee den Vornamen 'Jack' hat.

Exists Operation

Manchmal ist es notwendig zu wissen, ob wenigstens ein Element in einer Collection existiert, auf den das Constraint zutrifft. Die exists-Operation erlaubt es, einen boolschen Ausdruck zu definieren, der auf mindestens ein Element einer Collection zutrifft.

```
collect->exists ( boolean expression )
```

Auch die exists-Operation resultiert in einem Boolean, da das Ergebnis wahr wird, wenn der boolsche Ausdruck bei mindestens einem Element wahr wird. Der gesamte Ausdruck wird erst falsch, wenn der boolsche Ausdruck bei allen Elementen falsch wird.

Beispiel:

```
Company
  self.employee->exists ( forename = 'Jack' )
```

Dieser Ausdruck wird wahr, wenn der Vorname von mindestens einem employee 'Jack' lautet.

5.2.3.7 deSIGN-OCL-Erweiterung

Basistypen in deSIGN

Vordefinierte Basistypen, wie in OCL, werden in deSIGN nicht unterstützt, da mit der Verwendung von spracheigenen Typen es möglich wird, syntaktisch korrekte OCL-Ausdrücke zu formulieren, deren Auswertung fraglich ist, d.h. es ist nicht zu jeder Zeit möglich, diese Ausdrücke auszuwerten.

Um trotzdem mit den bequemen Basistypen arbeiten zu können, sollten in dem zugehörigen UML-Modell (oder deSIGN-Modell) diese Basistypen vom Modell aus in die Sprache als Typ eingebracht werden. Somit können auch kontextuelle und semantische Zweideutigkeiten beseitigt werden.

Innerhalb des deSIGN-Projektes wäre es denkbar, ein Standardtypenmodell zu erstellen, das als Kontextmodell permanent für OCL definiert ist. Somit würde das eigentliche UML- oder deSIGN-Modell jeweils um dieses Standardtypenmodell erweitert werden, so daß man bei der eigentlichen Modellierung durch dieses leicht eingeschränkt wird (Kollisionen von gleichnamigen Typen/Klassen etc.). Diese Einschränkung sollte jedoch für den Gewinn der Konformität vertretbar sein. Mit der Verwendung eines solchen evtl. selbst erstellten Standardtypenmodell könnnte man die vom Modell definierten Typen verwenden, die als Basistypen dienen.

Entwurf in deSIGN

Das Diagramm inAbbildung 5.32 stellt das OCL-Metamodell für den deSIGN-Entwurf vor.

Um die Typspiegelung vom Klassendiagramm zur OCL-Sprache herzustellen, erbt eine OCL-Interpreter-Klasse von der Klasse der DiagrammElemente. Jedes Diagrammelement des Klassendiagramms könnte also in der Lage sein, OCL-Ausdrücke zu interpretieren und zu jedem beliebigen Systemzustand in einer aktuellen Ausprägung auszuwerten. Dies wäre der generelle Ansatz einer Entwurfsmöglichkeit.

Falls in deSIGN dynamische und funktionale Modelle hinzukommen, wird der konzeptuelle Aspekt der atomaren Constraints problematisch, da keine reale Implementierung die Forderung der atomaren Auswertung erfüllen mag. Der Effekt läßt sich durch verschiedene Systemhaltepunkte jedoch durchaus simulieren.

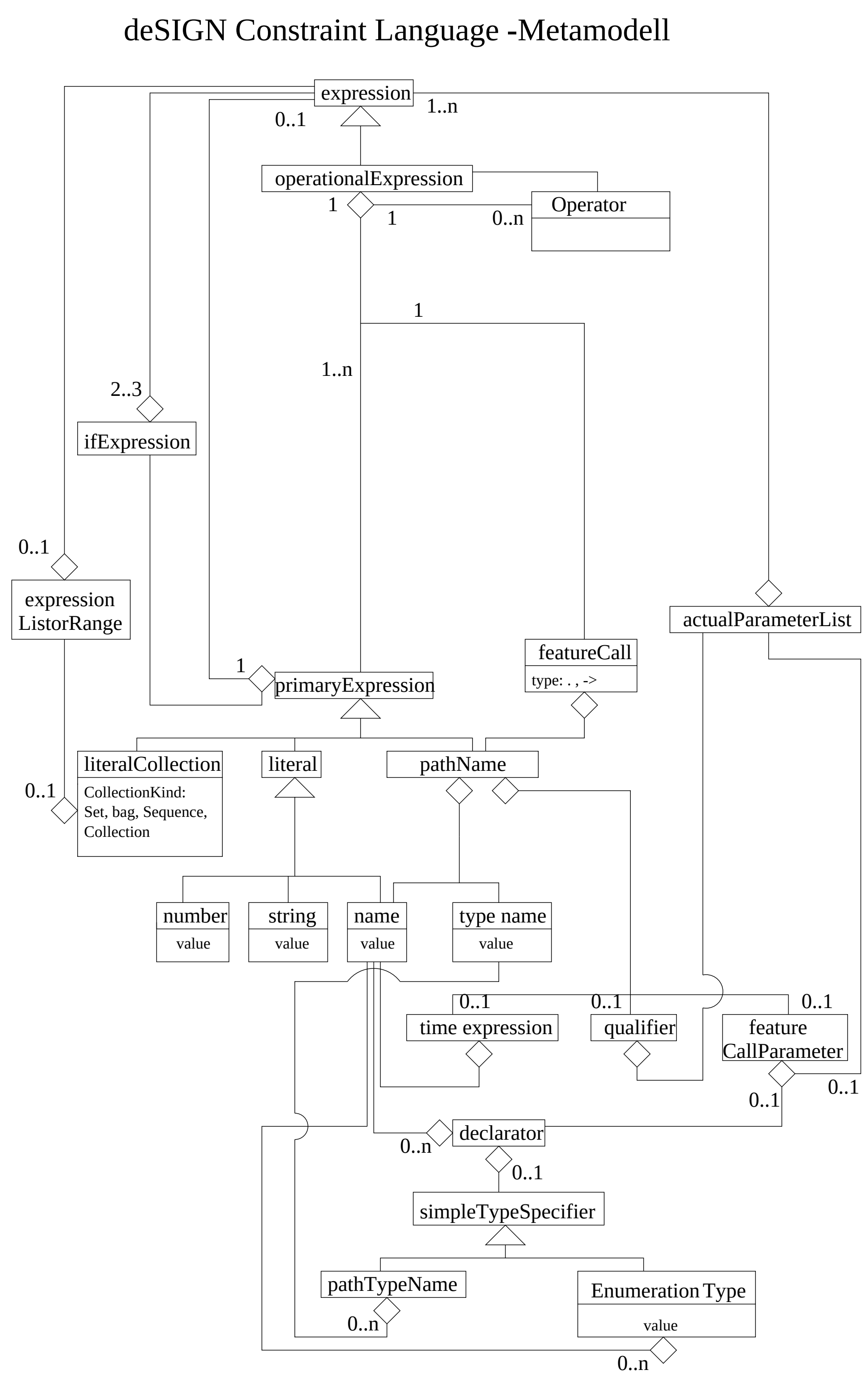


Abbildung 5.32: Metamodell deSIGN Constraint Language

5.3 Semantik II

5.3.1 Vorwort

Zunächst werden Statecharts isoliert, d.h. unabhängig von der Verwendung im Rahmen objektorientierter Spezifikationen, betrachtet. Statecharts werden als eigenständige Sprache präsentiert, die der Spezifikation von Verhaltensaspekten von IT-Systemen dienen soll. In einem zweiten Schritt kann dann untersucht werden, wie Statecharts sinnvoll mit Activitycharts und insbesondere mit Klassendiagrammen zur objektorientierten Beschreibung von Systemen kombiniert werden können.

Die isolierte Betrachtung von Statecharts erfolgt in mehreren Schritten. Dabei werden die Sprachkonstrukte der Statecharts nacheinander eingeführt und ihre Semantik sowie die Auswirkung auf die Interpretation der bereits vorhandenen Sprachkonstrukte erklärt. Der Rote Faden, der sich aufgrund der aufeinanderfolgenden Behandlung der Sprachkonstrukte durch diese Darstellung zieht, folgt i.W. nachstehender Aufzählung:

- sequentielle Automaten
- Nichtdeterminismus
- Interne Transitionen
- Termination
- Hierarchische Dekomposition
- Parallele Komposition
- Parametrisierte Statecharts
- Rekursion
- Boolsche Ausdrücke über Events
- Conditions
- Lokale Variablen
- Actions

Statechart-Struktur Statecharts können als algebraische Struktur aufgefaßt werden. Der Grundbereich einer solchen Struktur ist dann die Menge aller (syntaktisch korrekten) Statecharts bzw. die Sprache der Statecharts L_{SC}. In der Struktur gibt es die 0-stelligen Konstanten $\mathit{Stop}, \mathit{Skip}, \Omega$ sowie die 2-stelligen Konstanten $\longrightarrow$ (Transitionsrelation) und $\|$ (parallele Komposition).

Aus dieser Perspektive ist das Ziel der Definition der Semantik von Statecharts darin zu sehen, ein Axiomensystem für die algebraische Struktur $(L_{SC}, \mathit{Stop}, \mathit{Skip}, \Omega, \longrightarrow, \|)$ zu finden. Das Axiomensystem legt die Bedeutung der Konstanten fest.

Mit anderen Worten: Es soll eine Theorie der Statecharts entwickelt werden, so daß die Interpretation der Konstanten α_{SC} und der Grundbereich L_{SC} in der Struktur $S_{SC} = (L_{SC}, \alpha_{SC}) = (L_{SC}, \mathit{Stop}, \mathit{Skip}, \Omega, \longrightarrow, \|)$ es erlaubt, eine Interpretation anzugeben, die Modell des Axiomensystems ist.

Der in diesem Papier verfolgte Ansatz bei der Suche nach einem Axiomensystem, für das man mit der Struktur S_{SC} ein Modell erhält, besteht i.W. in einem Vergleich mit anderen Theorien. Es wird z.B. gefordert, daß sich Statecharts auf markierte gerichtete Graphen abbilden lassen. Ferner sollen einfache Statecharts auch als sequentielle Automaten aufgefaßt werden können. In Hinblick auf die Möglichkeit der schrittweisen Verfeinerung einer Spezifikation und der Beschreibung paralleler Prozesse drängt sich der Vergleich mit CSP auf. Durch die Definition von Abbildungen kann so das Wissen um die gut untersuchten Theorien von Graphen, sequentiellen Automaten, CSP, etc. genutzt werden, um zu einer formellen Semantik für Statecharts zu gelangen.

Folgende Theorien werden in der nachfolgenden Diskussion der Semantik von Statecharts berücksichtigt:

- Automatentheorie
- Graphentheorie
- Theorie Regelbasierter Systeme
- Communicating Sequential Processes (CSP)

5.3.2 Deterministische Sequentielle Automaten

Automaten Statecharts, die nur Transitionen mit einfachen Beschriftungen (Label) der Form $\sigma_1 \xrightarrow{e} \sigma_2$ haben, können als sequentielle Automaten aufgefaßt werden. Ein einfaches Statechart bzw. ein sequentieller Automat wird beschrieben durch ein Tupel

$$SC = (\Sigma, E, T, \sigma_0) \tag{5.1}$$

mit einer Menge von Zuständen (States) Σ, einer Menge elementarer Ereignisse (Events) E, einer Menge von Transitionen T und einem ausgezeichneten Anfangszustand $\sigma_0 \in \Sigma$. Jede Transition $t \in T$ hat die Form $\sigma_1 \xrightarrow{e} \sigma_2$, so daß $T \subseteq \Sigma \times E \times \Sigma$. Die Menge der elementaren Events E ist das Alphabet des Statecharts.

Intuitiv stellen wir uns vor, daß ein SC auf Ereignisse in seiner Umgebung reagiert und dabei von einem Zustand in einen anderen wechselt.

Wir verbieten zunächst nichtdeterministische Statecharts, so daß vorerst gilt:

$$\forall \sigma, \sigma_1, \sigma_2 \in \Sigma, e \in E : (\sigma, e, \sigma_1) \in T \wedge (\sigma, e, \sigma_2) \in T \Longrightarrow \sigma_1 = \sigma_2 \tag{5.2}$$

Durch die Transitionsmenge T ist die Zustandsübergangsfunktion $\delta : \Sigma \times E \to \Sigma$ eines deterministischen endlichen Automaten festgelegt. Die Menge T ist der Graph der Funktion δ.

$$\delta(\sigma, e) = \begin{cases} \sigma' & , \text{ falls } (\sigma, e, \sigma') \in T \\ \text{undef.} & , \text{ sonst} \end{cases} \tag{5.3}$$

Graphen Das SC läßt sich anschaulich als Graph darstellen:

$$G_{SC} = (V_{SC}, E_{SC}, s_{SC}, t_{SC}, l_{SC}, m_{SC}) \tag{5.4}$$

G_{SC} ist ein gerichteter Graph mit Knoten- und Kantenmarkierung, wie z.B. in [AH98] beschrieben. Im Allgemeinen besteht ein gerichteter markierter Graph $G = (V, E, s, t, l, m)$ aus einer Knotenmenge (Vertices) V, einer Kantenmenge (Edges) $E \subseteq V \times V$, einer Abbildung, die jeder Kante ihren Anfangsknoten (Source) zuordnet $s : E \to V$, einer Abbildung, die jeder Kante ihren Endknoten (Target) zuordnet $t : E \to V$, einer Knotenmarkierung (Label) $l : V \to C_V$, einer Kantenmarkierung (Marking) $m : V \to C_E$, sowie den Markierungalphabeten für Knoten C_V und Kanten C_E.

Der zugehörige Graph G_{SC} läßt sich für ein Statechart SC wie folgt berechnen:

$$\begin{array}{lcl} V_{SC} & = & \Sigma \\ E_{SC} & = & \{(s, s') \mid (s, e, s') \in T\} \\ s_{SC} & = & \{(e, v) \mid e = (v, v') \in E_{SC}\} \\ t_{SC} & = & \{(e, v') \mid e = (v, v') \in E_{SC}\} \\ m_{SC} & = & \{(e, c) \mid e = (v, v') \in E_{SC} \wedge (v, c, v') \in T\} \end{array} \tag{5.5}$$

Die Markierung der Kanten ergibt sich aus den Transitionen des SC. Das Markierungsalphabet ist die als Alphabet des SC-Automaten bezeichnete Ereignis-Menge E.

Die intuitive Vorstellung der Reaktion des SC auf Ereignisse in der Umgebung läßt sich dann mit Hilfe der Knotenmarkierung beschreiben. Die Markierung soll zu jedem Zeitpunkt genau einen Zustand als den gerade aktuellen Systemzustand auszeichnen. D.h. $l : V \to \{tt, ff\}$ leistet das gewünschte, wobei $l(v) = tt$ bedeutet, daß der Knoten v markiert ist und somit den zugehörigen Zustand als den aktuellen Systemzustand kennzeichnet. Durch den Anfangszustand $\sigma_0 \in \Sigma$ ist eine anfängliche Markierung $l_{SC,0}$ vorgegeben:

$$l_{SC,0}(\sigma) = \begin{cases} tt & , \text{ falls } \sigma = \sigma_0 \\ ff & , \text{ sonst} \end{cases} \tag{5.6}$$

Jetzt läßt sich das Verhalten des SC durch Angabe von Regeln definieren, die vorschreiben, wie sich die Markierung der Knoten ändert. Diese Regeln sind als Transitionen des SC bzw. als markierte Kanten des Graphen gegeben.

Regelbasierte Systeme Jedes SC induziert ein regelbasiertes System

$$rbS(SC) = (I_{SC}, P_{SC}, T_{SC}) \tag{5.7}$$

über der Regelbasis

$$\begin{aligned} B_{SC} &= (C, R, \Longrightarrow) \qquad (5.8)\\ &= (L_\Sigma, L_E, \longrightarrow) \qquad (5.9) \end{aligned}$$

Die Theorie Regelbasierter Systeme wird ausführlich in [Kre97] beschrieben. Eine umfassende Erläuterung der Theorie und der Begriffe Regelbasiertes System, Regelbasis, etc. führt über den Rahmen dieser Arbeit hinaus. Die Theorie wird hier als formaler Rahmen für eine Beschreibung von Statecharts verwendet. Dabei werden jedoch nur einzelne Aspekte der Theorie näher erläutert.

Die Regelbasis enthält

$$\begin{array}{lll} C & = L_\Sigma & \text{die Menge aller Konfigurationen für Systeme über} \\ & & \text{dieser Regelbasis, hier: die Sprache der Zustände} \\ R & = L_E & \text{die Menge aller Regeln für Statecharts,} \\ & & \text{hier: die Sprache der Ereignisse} \\ \Longrightarrow & = \longrightarrow & \text{dem Regelanwendungsoperator, hier: Transition} \end{array} \tag{5.10}$$

Das Regelbasierte System besteht aus

I_{SC}	$= \{\sigma_0\}$	eine Menge von Anfangskonfigurationen
P_{SC}	$= E$	eine Regelauswahl, hier: die im SC verwendeten Ereignisse
T_{SC}	$= \{\sigma \mid \forall \sigma' \in \Sigma, e \in E : (\sigma, e, \sigma') \notin T\}$	
	$= RED(P_{SC})$	eine Menge von Endkonfigurationen, hier: die bezüglich der Regelauswahl reduzierten Konfigurationen

(5.11)

Im Allgemeinen ist der Regelanwendungsoperator einer Regelbasis eine Relation auf den Zuständen:

$$\underset{r}{\Longrightarrow} \subseteq C \times C, \text{für } r \in R \tag{5.12}$$

Die Regelanwendung für den Operator $\underset{r}{\longrightarrow}$ wird definiert als

$$\sigma \underset{r}{\longrightarrow} \sigma' \quad \Longleftrightarrow \quad (\sigma, r, \sigma') \in T \tag{5.13}$$

$$\underset{r}{\longrightarrow} \quad = \quad \{(\sigma, \sigma') \mid (\sigma, r, \sigma') \in T\} \tag{5.14}$$

Jede direkte Ableitung $\sigma \underset{r}{\longrightarrow} \sigma'$ entspricht einer Transition des Statecharts. Ein Event wird als Regel interpretiert.

Anmerkung Ein Ereignis bezeichnet immer eine oder mehrere Transitionen zwischen den Zuständen eines SC. Statt der Ereignisse könnte man auch jede Transition als eine Regel auffassen. Die Regelmenge der Regelbasis für Statecharts wäre dann $R = L_\Sigma \times L_\Sigma$. Insbesondere im Hinblick auf die parallele Komposition von Statecharts ist es jedoch sinnvoll, Ereignisse als Regeln aufzufassen. Jede Regel repräsentiert dann eine oder mehrere Transitionen und die möglichen Ableitungen werden durch den Regelanwendungsoperator definiert. Im Falle paralleler Komposition von Statecharts kann eine Regel nur dann angewendet werden, wenn beide beteiligten Statecharts (rbS) dies zulassen. D.h., wenn eine Regel in der Regelauswahl beider an der parallelen Komposition beteiligten regelbasierten Systeme vorkommt, müssen beide Systeme in einer Konfiguration sein, für die der Regelanwendungsoperator die Anwendung zuläßt.

Regelbasiertes System auf Graphen Die Interpretation als Regelbasiertes System läßt sich problemlos auf den zugehörigen markierten Graphen übertragen. Sei $marking(\Sigma) = \{l : \Sigma \to \{tt, ff\}\}$ die Menge aller Markierungen der Knoten $V_{SC} = \Sigma$ des Graphen $G_{SC} = (V_{SC}, E_{SC}, s_{SC}, t_{SC}, l_{SC}, m_{SC})$. Dann induziert das SC ein regelbasiertes System

$$rbS(G_{SC}) = (I_G, P_G, T_G) \tag{5.15}$$

über der Regelbasis

$$B_G(\Sigma) = (marking(\Sigma), marking(\Sigma) \times marking(\Sigma), \longrightarrow) \tag{5.16}$$

Die Regelanwendung ist dann definiert durch

$$l \underset{r}{\longrightarrow} l' \iff \begin{array}{l} \exists\, v, v' \in V_{SC} : \\ \qquad \begin{array}{ll} (v, v') \in E & \\ m((v, v')) = r & \\ l(v) = tt & \\ l'(v') = tt & \\ l(v'') = \mathit{ff} & , \text{ f.a. } v'' \neq v \\ l'(v'') = \mathit{ff} & , \text{ f.a. } v'' \neq v' \end{array} \end{array} \tag{5.17}$$

Semantik Regelbasierter Systeme Die Semantik eines regelbasierten Systems S ist eine Relation zwischen Anfangs- und Endkonfigurationen.

$$SEM(S) \quad = \quad \overset{*}{\underset{P}{\Longrightarrow}} \cap\, I \times T \tag{5.18}$$

Die Semantik des Regelbasierten Systems für ein SC ist daher

$$SEM(S) \quad = \quad \{(\sigma_0, \sigma) \mid \sigma_0 \overset{*}{\underset{r \in P}{\longrightarrow}} \sigma \ \wedge\ \sigma \in RED(P)\} \tag{5.19}$$

Die Semantikrelation enthält alle Paare von Anfangs- und Endzuständen, für die gilt, daß man von dem Anfangszustand mittels wiederholter Regelanwendung zum Terminalzustand gelangt. Terminalzustände sind hier als solche Zustände definiert, die bezüglich der Regelauswahl reduziert sind. Dies sind die Konfigurationen, für die der Regelanwendungsoperator keinen Folgezustand liefert. Andere Alternativen zur Behandlung von Termination werden später in einem eigenen Abschnitt genannt.

Transformationelle vs. Reaktive Systeme Die Semantikrelation Regelbasierter Systeme genügt, wenn man das zu spezifizierende System als Transformationseinheit auffaßt. Eine Spezifikation, deren Semantik eine Relation zwischen Anfangs- und Endzustaänden liefert, behandelt den Transformationsprozeß bzw. das System als Black Box. Ein Programm P erfüllt die Spezifikation S, wenn gilt:

$$P \textbf{ sat } S \quad \iff \quad \forall\, \sigma \in I : \mathcal{M}[P](\sigma) \subseteq SEM(S)(\sigma) \tag{5.20}$$

Mit $\mathcal{M}[P] : \Sigma \to \wp(\Sigma)$ wird die Semantik des Programms P bezeichnet (genauer: Semantik partieller Korrektheit, da hier Divergenz nicht berücksichtigt wird, siehe [AO94], S.39).

Dies entspricht einer Spezifikation durch Vor- und Nachbedingungen bzw. Pre- und Postconditions ($Pre, Post$) wobei Pre und $Post$ Prädikate auf den Zuständen sind.

$$\sigma \models Pre \quad \Longrightarrow \quad \forall\, \sigma' \in SEM(S)(\sigma) : \sigma' \models Post \tag{5.21}$$
$$P \;\mathbf{sat}\; S \quad \Longleftrightarrow \quad \forall\, \sigma \in \Sigma : \sigma \models Pre \Longrightarrow \forall\, \sigma' \in \mathcal{M}[P](\sigma) : \sigma' \models Post \tag{5.22}$$

Die Spezifikation mittels Pre- und Postconditions wird auch als implizite Spezifikation bezeichnet, da hierbei nicht explizit ein Programm angegeben wird, das die gewünschten Eigenschaften hat, sondern implizit alle Programme zulässig sind, die die Spezifikation erfüllen.

Eine solche Spezifikation ist offenbar für Systeme geeignet,

- die ihre Eingaben auf einmal zur Verfügung haben, bzw. bei denen es nicht auf die Reihenfolge der Eingaben ankommt
- die aus den Eingaben Ausgabewerte berechnen und die Reihenfolge der produzierten Ausgabewerte keine Rolle spielt
- die Terminieren
- die mit der Umgebung nur über die Ein- und Ausgabewerte kommunizieren

Diese Eigenschaften charakterisieren transformationelle Systeme. Wir gehen jedoch davon aus, daß Statecharts insbesondere zur Beschreibung reaktiver Systeme eingesetzt werden. Für reaktive Systeme ist charakteristisch, daß

- sie kontinuierlich mit der Umgebung kommunizieren
- möglicherweise nie terminieren

Für reaktive Systeme ist es adäquater, das System als eine Folge von elementaren Ein- und Ausgabe-Ereignissen zu betrachten. Das Ziel der Spezifikation ist es dann, die Kommunikationsmöglichkeiten so einzuschränken, daß nur erwünschte Ereignissequenzen auftreten. Typische Beispiele reaktiver Systeme sind Systeme, die die Überwachung von Sensoren oder anderen Eingabegeräten beeinhalten.

Die Unterscheidung transformationeller und reaktiver Systeme ist aber pragmatischer Natur. Prinzipiell sind beide Sichtweisen äquivalent. Eine explizite Spezifikation in Form eines Programms, das bestimmte Ein- und Ausgabe-Ereignissequenzen ausführen kann, ist ersetzbar durch eine implizite Spezifikation in Form von Pre- und Postconditions über Ereignissequenzen. Eine implizite Spezifikation in Form einer Ein- und Ausgaberelation ist ersetzbar durch ein (nichtdeterministisches) Programm, daß entsprechende Eingabe-Ereignisse und Ausgabe-Ereignisse produziert. Die Entscheidung, ob eine Beschreibung als reaktives System oder als transformationelles System adäquat ist, hängt ab vom Spezifikationszweck bzw. den Aspekten des Systems, die in der Spezifikation berücksichtigt werden sollen.

Traces und Refusals Wir nehmen den Standpunkt ein, daß Statecharts verwendet werden, um eine explizite Spezifikation reaktiver Systeme anzugeben. Eine explizite Spezifikation soll dann die Menge der erlaubten Ereignissequenzen wiedergeben. Die Semantik von Statecharts soll es erlauben, die Menge aller möglichen Ableitungen zu berechnen. Für das zugeordnete Regelbasierte System ist dies die Menge

$$\begin{aligned} \mathit{derivations}(rbS(S)) &= \{(\sigma_0, \sigma) \mid \sigma_0 \xrightarrow[r \in P]{*} \sigma\} && (5.23) \\ &= \{\sigma_0 \xrightarrow[r_1]{} \sigma_1 \ldots \xrightarrow[r_n]{} \sigma_n \mid \sigma_0 \in I, \sigma_i \in \Sigma, r_i \in P\} && (5.24) \end{aligned}$$

Jede Ableitung $\sigma_0 \xrightarrow[r_1]{} \sigma_1 \ldots \xrightarrow[r_n]{} \sigma_n$ läßt sich auch durch ein Paar $(< r_1, r_2, \ldots, r_n >, R) = (s, R)$ beschreiben. Die Sequenz $s =< r_1, r_2, \ldots, r_n >$ ist ein Trace und $R = \{r \mid \forall \sigma' : (\sigma_n, \sigma') \notin \xrightarrow[r]{}\}$ eine Refusal-Menge, d.h. die Menge von Events, die das System nach Ausführung des Traces s ablehnen kann - oder: die Menge von Regeln, die in der nach Anwendung der Regeln aus s erreichten Konfiguration nicht angewendet werden können.

Das es zu jeder Ableitung ein äquivalentes Trace-Refusal-Paar gibt, kann unter Verwendung der starken Bisimulation als Äquivalenzbegriff für Statecharts (bzw. für die Konfigurationen des zugeordneten Regelbasierten Systems) begründet werden.

$$\sigma_0 \xrightarrow[r_1]{} \sigma_1 \ldots \xrightarrow[r_n]{} \sigma_n \iff (< r_1, r_2, \ldots, r_n >, \{r \mid \forall \sigma' : (\sigma_n, \sigma') \notin \xrightarrow[r]{}\}) \quad (5.25)$$

Die Behauptung ist, daß es eine bijektive Abbildung gibt, die jeder Ableitung ein Trace-Refusal-Paar zuordnet. Dazu zeigen wir, daß durch die genannte Beziehung eine Abbildung definiert wird, die injektiv und surjektiv ist. Nennen wir die Menge der Trace-Refusal-Paare $\mathit{Failures} = \mathit{Traces} \times \mathit{Refusals}$ Die durch 5.25 definerte Relation $f \subseteq \mathit{derivations} \times \mathit{Failures}$ soll die folgenden Eigenschaften haben:

$$\begin{array}{ll} \forall a \in \mathit{derivations} : \exists b \in \mathit{Failures} : & \\ (a, b) \in f & \text{Abbildungseigenschaft} \end{array} \quad (5.26)$$

$$\begin{array}{ll} \forall a \in \mathit{derivations} : \forall b_1, b_2 \in \mathit{Failures} : & \\ \{(a, b_1), (a, b_2)\} \subseteq f \Longrightarrow b_1 = b_2 & \text{Abbildungseigenschaft} \end{array} \quad (5.27)$$

$$\begin{array}{ll} \forall a_1, a_2 \in \mathit{derivations} : \forall b \in \mathit{Failures} : & \\ \{(a_1, b), (a_2, b)\} \subseteq f \Longrightarrow a_1 = a_2 & \text{Injektivität} \end{array} \quad (5.28)$$

$$\begin{array}{ll} \forall b \in \mathit{Failures} : \exists a \in \mathit{derivations} : & \\ (a, b) \in f & \text{Surjektivität} \end{array} \quad (5.29)$$

Die Eigenschaften 5.26 und 5.27 sind durch die in 5.25 definierte Relation sicher erfüllt. Somit ist $f : \mathit{derivations} \to \mathit{Failures}$ eine Abbildung.

Bisimulation Zum Nachweis der Injektivität von f müssen wir definieren, wann zwei Ableitungen a, a' äquivalent sind, d.h., wann $a = a'$ für $a, a' \in$ *derivations* gilt. Hierzu wird der Begriff der starken Bisimulation verwendet (siehe auch [Ros97], S.154ff). Eine Relation $R \subseteq \Sigma \times \Sigma$ auf den Zuständen bzw. den Konfigurationen des Regelbasierten Systems ist eine starke Bisimulation, g.d.w.

$$\begin{aligned}
&\forall \sigma_1, \sigma_2, \sigma_1' \in \Sigma : \forall r \in P : \\
&\qquad \sigma_1 \; R \; \sigma_1' \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad \Longrightarrow \quad \exists \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_2 \; R \; \sigma_2' && (5.30)\\
&\forall \sigma_1, \sigma_1', \sigma_2' \in \Sigma : \forall r \in P : \\
&\qquad \sigma_1 \; R \; \sigma_1' \wedge \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \quad \Longrightarrow \quad \exists \sigma_2 \in \Sigma : \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \wedge \sigma_2 \; R \; \sigma_2' && (5.31)
\end{aligned}$$

Wenn R eine Bisimulation ist und $\sigma \; R \; \sigma'$ gilt, dann sind σ und σ' bisimilar, d.h. äquivalent im Sinne der Bisimulation.

Zwei Konfigurationen σ und σ' sind bisimilar, wenn es eine Bisimulation R_{bisim} gibt, so daß $\sigma \; R_{bisim} \; \sigma'$ gilt. Die Relation

$$\begin{aligned}
R_{max} &= \{(\sigma, \sigma') \mid \sigma \text{ und } \sigma' \text{ sind bisimilar}\} && (5.32)\\
&= \{(\sigma, \sigma') \mid \exists \; R_{bisim} \colon \sigma \; R_{bisim} \; \sigma'\} \\
&= \bigcup R_{bisim}
\end{aligned}$$

ist selbst eine Bisimulation. Sie wird die maximale Bisimulation genannt. Die maximale Bisimulation ist eine Äquivalenzrelation, denn es gelten die folgenden Eigenschaften:

$$\begin{aligned}
&\forall \sigma \in \Sigma : \\
&\sigma \; R_{max} \; \sigma \quad \text{Reflexivität}
\end{aligned} \qquad (5.33)$$

$$\begin{aligned}
&\forall \sigma, \sigma' \in \Sigma : \\
&\sigma \; R_{max} \; \sigma' \Longrightarrow \sigma' \; R_{max} \; \sigma \quad \text{Symmetrie}
\end{aligned} \qquad (5.34)$$

$$\begin{aligned}
&\forall \sigma, \sigma', \sigma'' \in \Sigma : \\
&\sigma \; R_{max} \; \sigma' \wedge \sigma' \; R_{max} \; \sigma'' \Longrightarrow \sigma \; R_{max} \; \sigma'' \quad \text{Transitivität}
\end{aligned} \qquad (5.35)$$

Die Gleichheitsrelation $=$ ist eine Bisimulation. Dies ist offensichtlich, wenn man den Operator $=$ für R in die Gleichungen 5.30 und 5.31 einsetzt. Da $= \subseteq R_{max}$ gilt, folgt die Reflexivität von R_{max} und Gleichung 5.33 gilt.

Wenn R eine Bisimulation ist, dann auch die inverse Relation R^{-1}. Zum Beweis ist zu zeigen:

$$\begin{aligned}
&\forall \sigma_1, \sigma_2, \sigma_1' \in \Sigma : \forall r \in P : \\
&\qquad \sigma_1 \; R^{-1} \; \sigma_1' \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad \Longrightarrow \quad \exists \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_2 \; R^{-1} \; \sigma_2' && (5.36)\\
&\forall \sigma_1, \sigma_1', \sigma_2' \in \Sigma : \forall r \in P : \\
&\qquad \sigma_1 \; R^{-1} \; \sigma_1' \wedge \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \quad \Longrightarrow \quad \exists \sigma_2 \in \Sigma : \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \wedge \sigma_2 \; R^{-1} \; \sigma_2' && (5.37)
\end{aligned}$$

Wenn $\sigma_1 \; R^{-1} \; \sigma_1'$ gilt, dann gilt $\sigma_1' \; R \; \sigma_1$ und wenn R eine Bisimulation ist, dann gilt

$$\begin{aligned}
&\forall\, \sigma_1, \sigma_1', \sigma_2' \in \Sigma : \forall\, r \in P : \\
&\qquad \sigma_1' \; R \; \sigma_1 \wedge \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \quad \Longrightarrow \quad \exists\, \sigma_2 \in \Sigma : \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \wedge \sigma_2' \; R \; \sigma_2 \qquad (5.38)\\
&\forall\, \sigma_1, \sigma_1', \sigma_2 \in \Sigma : \forall\, r \in P : \\
&\qquad \sigma_1' \; R \; \sigma_1 \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad \Longrightarrow \quad \exists\, \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_2' \; R \; \sigma_2 \qquad (5.39)
\end{aligned}$$

durch äquivalente Umformung erhält man

$$\begin{aligned}
&\forall\, \sigma_1, \sigma_1', \sigma_2' \in \Sigma : \forall\, r \in P : \\
&\qquad \sigma_1 \; R^{-1} \; \sigma_1' \wedge \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \quad \Longrightarrow \quad \exists\, \sigma_2 \in \Sigma : \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \wedge \sigma_2 \; R^{-1} \; \sigma_2' \qquad (5.40)\\
&\forall\, \sigma_1, \sigma_1', \sigma_2 \in \Sigma : \forall\, r \in P : \\
&\qquad \sigma_1 \; R^{-1} \; \sigma_1' \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad \Longrightarrow \quad \exists\, \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_2 \; R^{-1} \; \sigma_2' \qquad (5.41)
\end{aligned}$$

Die Gleichungen 5.41 und 5.36 sind identisch. Ebenso die Gleichungen 5.40 und 5.37. Somit ist gezeigt, daß es für jede Bisimulation $R \subseteq R_{max}$ auch eine Bisimulation $R^{-1} \subseteq R_{max}$ gibt. Daraus folgt dann die Symmetrieeigenschaft von R_{max} und Gleichung 5.34 gilt.

Wenn R_1, R_2 Bisimulationen sind, dann auch das Relationenprodukt $R_1 \circ R_2$. Das Relationenprodukt ist wie folgt definiert:

$$R_1 \circ R_2 \quad = \quad \{(\sigma, \sigma'') \mid \exists\, \sigma' \in \Sigma : (\sigma \; R_1 \; \sigma') \wedge (\sigma' \; R_2 \; \sigma'')\} \qquad (5.42)$$

Zum Beweis, daß $R_1 \circ R_2$ eine Bisimulation ist, ist zu zeigen:

$$\begin{aligned}
&\forall\, \sigma_1, \sigma_2, \sigma_1' \in \Sigma : \forall\, r \in P : \\
&\quad \sigma_1 \; R_1 \circ R_2 \; \sigma_1' \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad \Longrightarrow \quad \exists\, \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_2 \; R_1 \circ R_2 \; \sigma_2' \qquad (5.43)\\
&\forall\, \sigma_1, \sigma_1', \sigma_2' \in \Sigma : \forall\, r \in P : \\
&\quad \sigma_1 \; R_1 \circ R_2 \; \sigma_1' \wedge \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \quad \Longrightarrow \quad \exists\, \sigma_2 \in \Sigma : \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \wedge \sigma_2 \; R_1 \circ R_2 \; \sigma_2' \qquad (5.44)
\end{aligned}$$

Aus der Definition des Relationenprodukts nach Gleichung 5.42 folgt

$$\begin{aligned}
&\forall\, \sigma_1, \sigma_1' \in \Sigma : \\
&\quad \sigma_1 \; R_1 \circ R_2 \; \sigma_1' \quad \Longrightarrow \quad \exists\, \sigma_a \in \Sigma : \sigma_1 \; R_1 \; \sigma_a \wedge \sigma_a \; R_2 \; \sigma_1' \qquad (5.45)
\end{aligned}$$

Da R_1, R_2 Bisimulationen sind, gilt

$$\begin{aligned}
\sigma_1 \; R_1 \; \sigma_a \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad &\Longrightarrow \quad \exists\, \sigma_b \in \Sigma : \sigma_a \underset{r}{\longrightarrow} \sigma_b \wedge \sigma_2 \; R_1 \; \sigma_b \qquad (5.46)\\
\sigma_a \; R_2 \; \sigma_1' \wedge \sigma_a \underset{r}{\longrightarrow} \sigma_b \quad &\Longrightarrow \quad \exists\, \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_b \; R_2 \; \sigma_2' \qquad (5.47)
\end{aligned}$$

Zusammen ergibt sich aufgrund der Eigenschaften des Relationenprodukts und der Eigenschaften der Bisimulation:

$$\begin{aligned}
&\forall\, \sigma_1, \sigma_2, \sigma_1' \in \Sigma : \forall\, r \in P : \\
&\quad \sigma_1 \; R_1 \circ R_2 \; \sigma_1' \wedge \sigma_1 \underset{r}{\longrightarrow} \sigma_2 \quad \Longrightarrow \quad \exists\, \sigma_a, \sigma_b, \sigma_2' \in \Sigma : \sigma_a \underset{r}{\longrightarrow} \sigma_b \wedge \sigma_2 \; R_1 \; \sigma_b \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_b \; R_2 \; \sigma_2' \\
&\qquad\qquad\qquad\qquad\qquad\qquad \Longrightarrow \quad \exists\, \sigma_2' \in \Sigma : \sigma_1' \underset{r}{\longrightarrow} \sigma_2' \wedge \sigma_2 \; R_1 \circ R_2 \; \sigma_2' \qquad (5.48)
\end{aligned}$$

Damit ist Gleichung 5.43 bewiesen. Der Sachverhalt wird in der Abbildung 5.33 veranschaulicht. Der Beweis für Gleichung 5.44 verläuft analog.

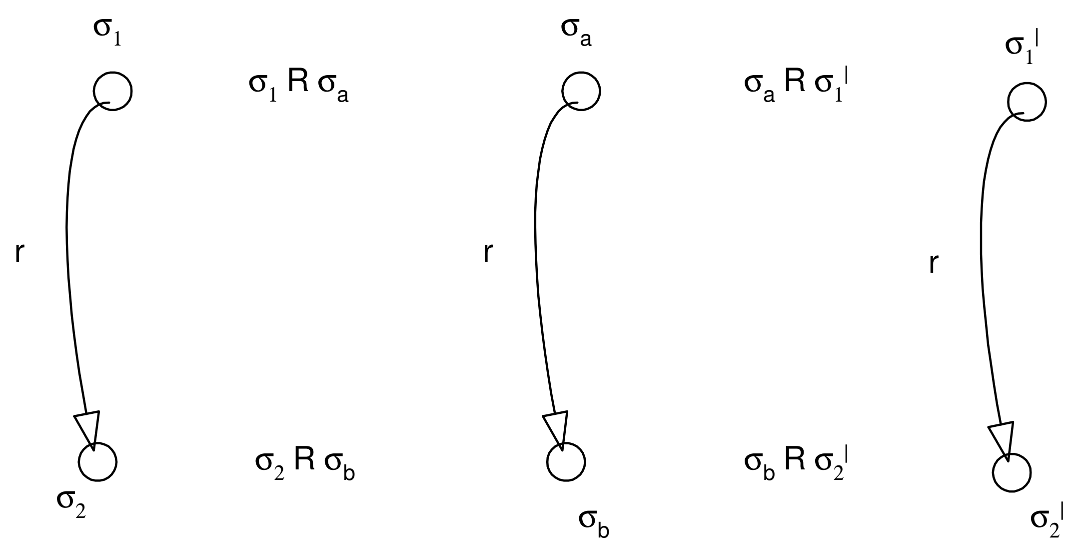


Abbildung 5.33: Kompositon von Bisimulationen

Die Transitivität von R_{max} ist nun offensichtlich, denn mit $R_1, R_2 \subseteq R_{max}$ gilt auch $R_1 \circ R_2 \subseteq R_{max}$.

Beweis der Injektivität Der Nachweis der Injektivität der in Gleichung 5.25 definierten Abbildung f : *derivations* $\rightarrow$ *Failures* gelingt, wenn man die Bisimulation zur Definition der Äquivalenz von Ableitungen verwendet. Zwei Ableitungen $a = \sigma_0 \underset{r_1}{\longrightarrow} \sigma_1 \ldots \underset{r_n}{\longrightarrow} \sigma_n$ und $a' = \sigma'_0 \underset{r'_1}{\longrightarrow} \sigma'_1 \ldots \underset{r'_n}{\longrightarrow} \sigma'_n$ werden als gleich angesehen, wenn $r_i = r'_i$ und $\sigma_j \; R_{max} \; \sigma'_j$ für $i = 1 \ldots n$ und $j = 0 \ldots n$ gilt. Wir verwenden die Schreibweise $a \overset{bisim}{=} a'$, um auszudrücken, daß a und a' äquivalente Ableitungen im Sinne der Bisimulation sind.

Dann kann man mit Induktion über die Länge der Ableitung $\sharp a$ zeigen, daß f injektiv ist. Zu zeigen ist:

$$f(a) = (s, R) \wedge f(a') = (s, R) \quad \Longrightarrow \quad a = a' \tag{5.49}$$

Es gilt sicher, daß $\sharp a = \sharp a' = \sharp s$ ist. Ferner gilt $s =< r_1, r_2, \ldots, r_n >=< r'_1, r'_2, \ldots, r'_n >$ somit $r_i = r'_i$ für $i = 1 \ldots n$. Also haben die Ableitungen a und a' die Form

$$a = \sigma_0 \underset{r_1}{\longrightarrow} \sigma_1 \ldots \underset{r_n}{\longrightarrow} \sigma_n \tag{5.50}$$

$$a' = \sigma'_0 \underset{r_1}{\longrightarrow} \sigma'_1 \ldots \underset{r_n}{\longrightarrow} \sigma'_n \tag{5.51}$$

Induktionsanfang: $\sharp a = 0$. Dann gilt $s =<>, R = R_{\sigma_0} = R_{\sigma'_0}$ und $a = \sigma_0 \overset{0}{\longrightarrow} \sigma_0$ und $a' = \sigma'_0 \overset{0}{\longrightarrow} \sigma'_0$ sind äquivalente Ableitungen im Sinne der Bisimulation, wenn σ_0 und σ'_0 bisimilar sind. Da Statecharts nach unserer jetzigen Definition nur einen Anfangszustand haben, beginnen alle Ableitungen mit demselben Zustand und es gilt $\sigma_0 = \sigma'_0$. Dann gilt jedenfalls $\sigma_0 \; R_{max} \; \sigma'_0$ und $a \overset{bisim}{=} a'$.

Induktionsvorraussetzung: Für eine Ableitung der Länge n gilt

$$f(a_n) = (s, R) \wedge f(a'_n) = (s, R) \quad \Longrightarrow \quad a_n = a'_n \tag{5.52}$$

D.h. es gilt auch $\sigma_n \; R_{max} \; \sigma'_n$.

Induktionsschritt: $\sharp a = n + 1$. Für eine Ableitung der Länge n+1 gilt dann $f(a_{n+1}) = f(a'_{n+1}) = (s, R)$ und $a_{n+1} = a_n \underset{r_{n+1}}{\longrightarrow} \sigma_{n+1}$ sowie $a'_{n+1} = a'_n \underset{r_{n+1}}{\longrightarrow} \sigma'_{n+1}$.

Nach Induktionsvorraussetzung gilt bereits $a_n \overset{bisim}{=} a'_n$. Es bleibt zu zeigen, daß auch $\sigma_{n+1} \; R_{max} \; \sigma'_{n+1}$ gilt.

Nach Induktionsvorraussetzung gilt auch $\sigma_n \; R_{max} \; \sigma'_n$. Daher gilt

$$\begin{aligned} & \forall \sigma_{n+1} \in \Sigma, r \in P : \\ & \quad \sigma_n \underset{r}{\longrightarrow} \sigma_{n+1} \quad \Longrightarrow \quad \exists \sigma'_{n+1} \in \Sigma : \\ & \qquad \sigma'_n \underset{r}{\longrightarrow} \sigma'_{n+1} \wedge \sigma_{n+1} \; R_{max} \; \sigma'_{n+1} \end{aligned} \tag{5.53}$$

Da wir zur Zeit nur deterministische Statecharts betrachten, liefert die Definition von f zusammen mit der Induktionsvorrausetzung genau einen Folgezustand σ'_{n+1} und für diesen gilt $\sigma_{n+1}\ R_{max}\ \sigma'_{n+1}$.

Anmerkung Die Situation wird offenbar komplizierter, wenn man Nichtdeterminismus zuläßt. Die Kernaussage, die sich hier aus der Existenz einer bijektiven Abbildung $f : \mathit{derivations} \rightarrow \mathit{Failures}$ ableiten läßt ist jedoch, daß $\mathit{Failures}(rbS(SC)) \overset{bisim}{=} \mathit{derivations}(rbS(SC))$ gilt. Dies gilt auch für nichtdeterministische Statecharts. Der Beweis muß dann jedoch mit Induktion über die Menge der in n Schritten ableitbaren Konfigurationen geführt werden.

Failures vs. derivations In der Menge $\mathit{derivations}(rbS(SC))$ werden Ableitungen unterschieden, die namentlich verschiedene Konfigurationen bzw. Zustände enthalten, die aber bzgl. der möglichen Regelanwendungen bzw. Ereignisse dasselbe Verhalten aufweisen. Die maximale Bisimulation ist eine Relation, die die Konfigurationen in solche Mengen unterteilt, deren Elemente dasselbe Verhalten aufweisen. In der Menge $\mathit{failures}(rbS(SC))$ sind alle Ableitungen enthalten, die sich bis auf Äquivalenz bzgl. Bisimulation unterscheiden. In diesem Sinne gilt $\mathit{Failures}(rbS(SC)) \overset{bisim}{=} \mathit{derivations}(rbS(SC))$. Bemerkenswert ist, daß bisimilare Zustände sich nicht hinsichtlich ihres beobachtbaren Verhaltens unterscheiden. Sollen Statecharts verwendet werden, um das Verhalten von (reaktiven) IT-Systemen zu beschreiben, so ist eine Semantik unter Verwendung des Failures-Modells offenbar besser geeignet.

Beweis der Surjektivität Zum Nachweis, daß durch Gleichung 5.25 eine bijektive Abblidung definert wird, ist noch die Surjektivität zu zeigen:

$$\forall (s,R) \in \mathit{Failures} : \exists\, a \in \mathit{derivations} : \qquad f(a) = (s,R) \tag{5.54}$$

Sei $s =< r_1, r_2, \ldots, r_n >$, dann gibt es nach Definition von f Zustände $\sigma_0, \ldots, \sigma_n$ und eine Ableitung $a = \sigma_0 \underset{r_1}{\longrightarrow} \sigma_1 \underset{r_2}{\longrightarrow} \ldots \underset{r_n}{\longrightarrow} \sigma_n$ und im Zustand σ_n können die Regeln der Menge R nicht angewendet werden. $R = \{r \mid \forall \sigma' \in \Sigma : (\sigma_n, \sigma') \notin \underset{r}{\longrightarrow}\}$.

Sprache als Regelbasiertes System Man kann die Sprache der Statecharts selber als ein Regelbasiertes System auffassen. Die operationelle Semantik von Statecharts kann dann durch Angabe von Regeln zur syntaktischen Transformation definiert werden. Bei diesem Vorgehen orientieren wir uns an der mathematischen Logik. Für die Sprache der mathematischen Logik 1. Stufe gibt es den Sequenzenkalkül (siehe [EFT94],[Gen34]), der es erlaubt, alle aus einer Formelmenge ableitbaren Formeln per syntaktischer Transformation zu gewinnen. Wie bereits eingangs angedeutet, betrachten wir jetzt die Sprache selber als den Grundbereich der Objekte und beschreiben die Semantik durch Angabe von Axiomen und abgeleiteten Regeln. Dabei verwenden wir eine Metasprache, bei der wir uns am Sequenzenkalkül sowie an der Theorie Regelbasierter Systeme orientieren.

Wenn man die Sprache der Statecharts als Regelbasiertes System auffaßt, so ist die Regelbasis

$$B_{Lang} = (\wp(\longrightarrow_{SC}), \wp(\longrightarrow_{SC}), \Longrightarrow_{Lang}) \quad (5.55)$$

$$\longrightarrow_{SC} \subseteq L_{SC} \times L_{Event} \times L_{SC} \quad (5.56)$$

Die Regelbasis der operationellen Semantik beschreibt eine Relation auf den Transitionen bzw. direkten Ableitungen von Statecharts.

Die Konfigurationen sind Transitionen von Statecharts. Die Regeln überführen ein Statechart mittels syntaktischer Transformation in ein anderes Statechart. Für elementare Statecharts ($Skip, Stop$) werden Axiome eingeführt, aus denen hervorgeht, welche Ableitungen für sie möglich sind. Für zusammengesetzte Statecharts werden Regeln angegeben, die es ermöglichen, aus den Ableitungen der Komponenten-Statecharts, die Ableitungen des zusammengesetzten Statecharts zu berechnen. Die Mengen $derivations(rbS(SC))$ und $Failures(rbS(SC))$ lassen sich durch wiederholte Anwendung des Regelanwendungsoperators ermitteln, wenn man sich zwischendurch die Zustände und Ereignisse bzw. Konfigurationen und Regeln merkt. Wie beim Sequenzenkalkül der mathematischen Logik werden die Regeln in der folgenden Form notiert:

$$\begin{aligned} &\frac{A}{S}\,[C] \\ &mit \quad A \quad = Antecedent, Premisse \\ &\qquad\quad\; S \quad = Succedent, Konklusion \\ &\qquad\quad\; C \quad = Condition, BedingungderAnwendbarkeit \end{aligned} \quad (5.57)$$

Im Antezedent A steht eine Sequenz von Ableitungen (der Komponenten-Stratecharts eines zusammengesetzten Statecharts) oder, falls der Antezedent leer ist, ist die Regel ein Axiom der operationellen Semantik. Im Sukzedent S steht ebenfalls eine Ableitung (für das aus den Komponenten des Antezedent zusammengesetzte Statechart). Zusätzlich kann eine in der Sprache der mathematischen Logik formulierte Bedingung C gefordert werden, die bei Anwendung der Regel erfüllt sein muß.

Ein Regelbasiertes System über dieser Regelbasis

$$S = (\wp(\longrightarrow_{SC}), P, RED(P)) \quad (5.58)$$

kann als ein Interpreter oder Modelchecker aufgefaßt werden. Das System beginnt mit eine initialen Menge von Transitionen, die durch ein Statechart beschrieben werden, als Anfangskonfiguration und wendet solange es geht die Regeln der operationellen Semantik P an. Im Folgenden definieren wir die operationelle Semantik von Statecharts durch Festlegung der Regelmenge P bzw. durch explizite Angabe der enthaltenen Regeln.

Im Allgemeinen ist es möglich, daß für eine Konfiguration mehrere Regeln zur Auswahl stehen, die angewendet werden könnten. Das System ist daher i.A. nichtdeterministisch. Zum Umgang mit Nichtdeterminismus gibt es verschiedene Möglichkeiten, die ausführlich in [Kre97] beschrieben sind. Ein Modelchecker wird beispielsweise die ausschöpfende Suche anwenden, um alle möglichen Ableitungen eines Statecharts zu ermitteln. Ein Interpreter könnte zur Regelauswahl die Interaktion mit der Umgebung bzw. dem Benutzer erfordern.

Transitionsregel Für die Transitionen des Statecharts läßt sich folgende Regeln angeben:

$$\frac{}{(\Sigma, E, T, \sigma) \xrightarrow{e} (\Sigma, E, T, \sigma')} \, [e \in E, \sigma, \sigma' \in \Sigma, (\sigma, e, \sigma') \in T] \quad \text{Transition} \tag{5.59}$$

Dabei ist σ der Anfangszustand des Statecharts vor Ausführung der Transition und σ' der Folgezustand nach dem Event e bzw. der Anfangszustand des Statecharts nach Ausführung der Transition. Das Ausführen der Transition entspricht dem Anwenden der mit e benannten Regel des $rbS(SC)$. Die Transitionsregel sagt aus, daß die Regel e angewendet werden kann, wenn es im Statechart eine mit dem Label e versehene Transition gibt, deren Quellzustand der Anfangszustand σ ist. Nach der Regelanwendung erhält man ein Statechart mit dem neuen Anfangszustand σ', wobei dies der Zielzustand der auszuführenden Transition ist.

5.3.3 Nichtdeterminismus

Verzichtet man auf die in Gleichung 5.2 angegebene Bedingung, so erhält man ein nichtdeterministisches Statechart. Bei diesem ist es möglich, daß mehrere Transitionen mit demselben Label aus einem Zustand herausführen.

Der Regelanwendungsoperator des zugeordneten Regelbasierten Systems ist dann nichtdeterministisch. In einer Konfiguration kann eine Regel gelegentlich mehrfach angewendet werden. Der lokale Suchraum $choice(\sigma) = \{\sigma' \mid \sigma \underset{E}{\Longrightarrow} \sigma'\}$ ist jedoch stets endlich. Das Regelbasierte System ist deshalb lokal endlich.

Ein nichtdeterministisches Statechart kann als nichtdeterministischer endlicher Automat aufgefaßt werden. Die Zustandsübergangsfunktion muß in diesem Fall jedoch abgewandelt werden: $\delta : \Sigma \times E \rightarrow \wp(\Sigma)$

$$\delta(\sigma, e) = \{\sigma' \mid (\sigma, e, \sigma') \in T\} \tag{5.60}$$

Für nichtdeterministische endliche Automaten ist bekannt, daß sie durch Konstruktion eines Potenzautomaten in einen äquivalenten deterministischen Automaten umgewandelt werden können (siehe auch [HU94]).

5.3.4 ϵ -Transitionen

Als Spezialfall einer Transition können unbeschriftete Transitionen betrachtet und zugelassen werden. Transitionen ohne Label haben die Form $(\sigma, \epsilon, \sigma')$ wobei ϵ für ein Ereignis steht, daß von außen nicht wahrgenommen werden kann. Eine solche Transition findet ohne Kommunikation mit der Umgebung des Statecharts statt. Ein Zustand, aus dem eine ϵ-Transition herausführt ist nicht stabil und führt ebenfalls zu nichtdeterministischem Verhalten.

5.3.5 Transformation nach CSP

Statecharts können als Prozeßbeschreibungen im Sinne von CSP aufgefaßt werden. Eine Transition kann in CSP mit dem Prefix-Operator ausgedrückt werden:

$$(\sigma_0, e, \sigma') \in T \quad \Longleftrightarrow \quad P = e \rightarrow P' \tag{5.61}$$

wobei P' der Prozeß ist, der dem Statechart zugeordnet ist, daß aus (Σ, E, T, σ_0) durch Anwendung der Transitionsregel hervorgeht.

Nichtdeterminismus kann in CSP durch Verwendung des internal choice Operators dargestellt werden.

$$\begin{array}{c} (\sigma_0, e, \sigma') \in T \\ \wedge \\ (\sigma_0, e, \sigma'') \in T \end{array} \quad \Longleftrightarrow \quad \begin{array}{rl} P = & e \rightarrow P' \\ & \sqcap \\ & e \rightarrow P'' \\ = & e \rightarrow (P' \sqcap P'') \end{array} \tag{5.62}$$

Für jede Transition, die das Statechart im Zustand σ ausführen kann, besitzt der zugeordnete CSP-Prozeß eine Alternative bzw. einen Subprozeß. Jede der Alternativen bietet der Umgebung die Möglichkeit, einen Event zu kommunizieren (bzw. auftreten zu lassen).

$$T \supseteq \{(\sigma_0, e_i, \sigma_j) \mid i = 1 \dots n, j = 1 \dots m\} \quad \Longleftrightarrow \quad \begin{array}{rl} P = & e_1 \rightarrow P_1 \\ & \Box\, e_2 \rightarrow P_2 \\ & \dots \\ & \Box\, e_n \rightarrow P_m \end{array} \tag{5.63}$$

Die Umgebung entscheidet, welcher der angebotenen Events auftritt. Der CSP-Operator $\Box$ wird deshalb auch external choice oder deterministic choice genannt. Gilt jedoch $e_i = e_j$ für ein $i \neq j$, so wird daraus eine nichtdeterministische Auswahl, denn es gilt:

$$(a \rightarrow P) \Box (a \rightarrow Q) \quad = \quad a \rightarrow (P \sqcap Q) \tag{5.64}$$

Wir haben als Terminalzustände von Statecharts solche Zustände identifiziert, die keine über Transitionen erreichbaren Folgezustände haben. Ein Statechart mit nur einem initialen Zustand und leerer Transitionsmenge kann hierfür als Repräsentant dienen.

$$(\{\sigma_0\}, \emptyset, \emptyset, \sigma_0) \quad \Longleftrightarrow \quad \mathit{Skip} \tag{5.65}$$

Der CSP-Prozeß *Skip* terminiert sofort. Um Termination anzuzeigen, besitzt CSP einen speziellen Event $\checkmark$, genannt tick. Der Event $\checkmark$ gehört zu keinem

Alphabet E eines Prozesses und kann von der Umgebung nicht wahrgenommen werden. Es gilt (mit gewissen Einschränkungen, s.u.)

$$Skip \quad = \quad \checkmark \rightarrow Stop \tag{5.66}$$

wobei *Stop* ein Prozeß ist, der keinerlei Kommunikation mit der Umgebung ausführt. Der Prozeß *Stop* verhält sich wie ein Prozeß, der sich in einer Deadlock-Situation befindet. Für die Behandlung von Termination gibt es in CSP spezielle Regeln (siehe [Ros97]). Insbesondere wird der tick-Event in den Traces sichtbar, die ein Prozeß ausführen kann. Die Semantik muß diesen speziellen Event berücksichtigen, wenn man zwischen Termination und Deadlock unterscheiden können will.

Für den STOP-Prozeß gibt es in der operationellen Semantik von CSP keine Regel. Dies ist gerade kennzeichnend für die Deadlock-Situation: es ist keine Regelanwendung mehr möglich, aber es ist auch kein Terminalzustand erreicht. Markiert man das Erreichen eines Terminalzustands mit dem speziellen Event $\checkmark$, so läßt sich die Deadlock-Situation von erfolgreicher Termination unterscheiden. Für den SKIP-Prozeß gibt es in der operationellen Semantik von CSP daher die Regel

$$\frac{}{Skip \xrightarrow{\checkmark} \Omega} \tag{5.67}$$

wobei Ω für einen Prozeß steht, der bereits erfolgreich terminiert hat. Aus technischen Gründen (insbesondere zur Handhabung verteilter Temination, siehe [Ros97]) will man Ω von *Stop* unterscheiden, so daß Gleichung 5.66 nur in einem gewissen Sinne gilt.

5.3.6 Termination

Bisher haben wir für Statecharts nicht zwischen Deadlock und Termination unterschieden. Wir können dies tun, indem wir die Endzustände explizit markieren. In der graphischen Repräsentation kann man diese Zustände mit einem zweiten Rahmen umgeben o.ä., wie dies in der Automatentheorie üblich ist. Zusätzlich oder alternativ kann man die Termination durch Angabe einer Transition in einen (Pseudo-) Zustand *Skip* darstellen. In der UML-/OMT-Variante von Statecharts wird dieser Zustand mit dem Symbol $\odot$ gekennzeichnet. Wir schließen uns der zweiten Variante an und definieren, daß es für jedes Statechart die speziellen Zustände $\mathit{Skip}, \Omega, \mathit{Stop}$ gibt. Unsere operationelle Semantik für Statecharts ergänzen wir um die Regel:

$$\frac{}{(\Sigma, E, T, \mathit{Skip}) \xrightarrow{\checkmark} (\Sigma, E, T, \Omega)} \quad \text{Skip} \tag{5.68}$$

Zusätzlich können automatisch alle Zustände als Terminalzustände gelten, für die es keine Folgezustände gibt. Die Äquivalenzbeziehung 5.65 kann daher weiter gelten.

Wie in der Automatentheorie üblich, wird ein Statechart nunmehr um die explizite Angabe der Menge terminaler Zustände ergänzt (F = Final States):

$$SC = (\Sigma, E, T, \sigma_0, F) \tag{5.69}$$

Für Statecharts ohne hierarchische Dekomposition gilt dann

$$\sigma \in F \quad \Longrightarrow \quad \sigma = \mathit{Skip} \tag{5.70}$$
$$\sigma \notin F \quad \Longrightarrow \quad \sigma = \mathit{Stop} \tag{5.71}$$

Die terminalen Zustände der Menge F sind die *Skip*-Zustände, auf die man die Regel 5.68 anwenden kann.

5.3.7 Hierarchische Dekomposition

Ein Zustand in einem Statechart ist entweder elementar oder er wird durch Angabe eines eigenen Statecharts näher spezifiziert. Es gilt dann:

$$L_{SC} = \wp(L_{St}) \times \wp(L_{Ev}) \times \wp(L_{Tr}) \times L_{St} \times \wp(L_{St}) \quad (5.72)$$

$$L_{St} = State + (State \times L_{SC}) \quad (5.73)$$

wobei mit $State : Exp$ die Menge aller elementarer Zustände (Zustandsbezeichner) und mit L_{St}, L_{Ev}, L_{Tr} die Sprache der Zustände, Ereignisse und Transitionen bezeichnet wird.

Man erhält so eine wechselseitig rekursive Beziehung zwischen Zuständen und Statecharts.

Ein hierarchisch strukturiertes Statechart kann mittels Transformationsregeln in ein äquivalentes einfaches Statechart ohne hierarchische Dekomposition übersetzt werden. Ein auf Graphtransformation basierendes Verfahren hierzu wird in [GP98] beschrieben.

Beobachtungen Transitionen, deren Quelle am Rand eines Statecharts liegt, haben eine besondere Bedeutung. Als Beispiel kann die mit dem Label i versehene Transition in den Abbildungen 5.34 und 5.35 dienen. In dem Beispiel führt das Auftreten des Events i stets zu einem Übergang in den Zustand Y. Dabei ist es egal, in welchem Zustand sich das Statecharts X gerade befindet. Die Übersetzung in ein äquivalentes Statechart ohne hierarchische Dekomposition behandelt diesen Fall daher so, als gäbe es eine mit dem Label i versehene Transition von jedem der Zustände in X zum Zustand Y.

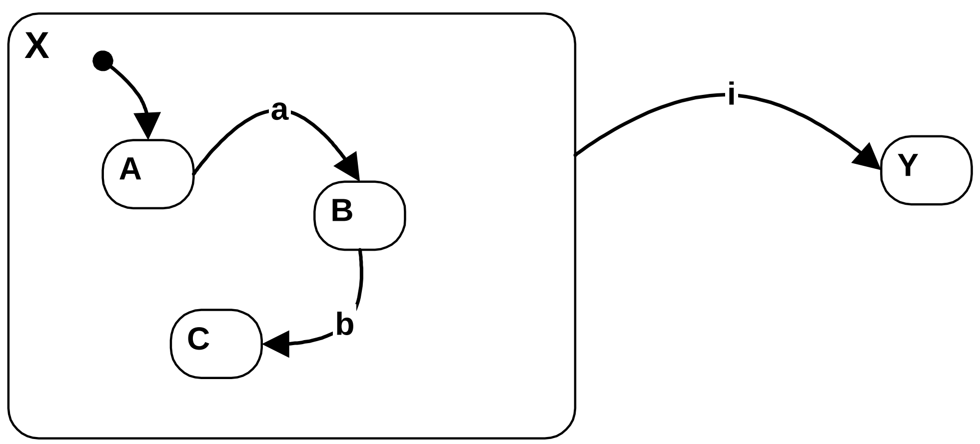


Abbildung 5.34: Interrupt

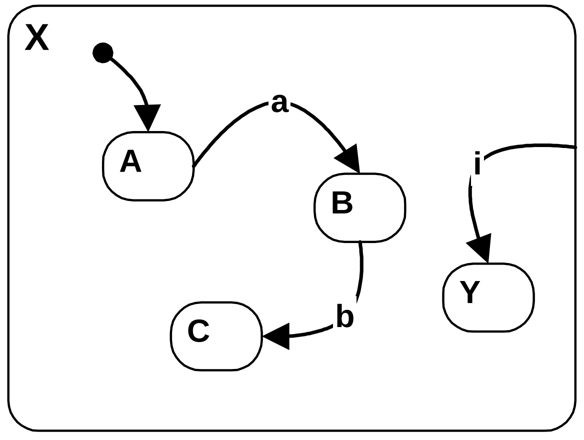


Abbildung 5.35: Interrupt

Bei der Übersetzung in einen CSP-Prozeß bietet sich die Verwendung des Interrupt-Operators an:

$$\begin{aligned} P &= X \triangle_i Y \\ X &= a \to b \to \mathit{Skip} \end{aligned} \tag{5.74}$$

Wir können Zustände mit CSP-Prozessen identifizieren. Ein elementarer Zustand entspricht dann dem Prozeß *Stop*, ein terminaler Zustand dem Prozeß *Skip* und Transitionen werden durch den Interrupt-Operator dargestellt. Das Beispiel-Statechart aus Abbildung 5.34 wird dann wie folgt übersetzt.

$$\begin{aligned} P &= X \triangle_i Y \\ X &= \mathit{Stop} \triangle_a (\mathit{Stop} \triangle_b \mathit{Skip}) \end{aligned} \tag{5.75}$$

In der Failures/Divergences-Semantik von CSP sind die Prozesse 5.74 und 5.75 äquivalent (siehe [Ros97], S.235). Es gilt:

$$\frac{}{P \triangle_i Q \xrightarrow{i} Q} \quad \text{Interrupt-1} \tag{5.76}$$

$$\frac{P \xrightarrow{e} P'}{P \triangle_i Q \xrightarrow{e} P' \triangle_i Q} [i \neq e] \quad \text{Interrupt-2} \tag{5.77}$$

Da es für den Prozeß *Stop* keine Regel gibt, kann auf $\mathit{Stop} \triangle_i Q$ die Regel 5.77 nie angewendet werden. Daher gilt

$$\mathit{Stop} \triangle_i Q = i \to Q \tag{5.78}$$

In der denotationellen Semantik von CSP wird der Interrupt-Operator folgendermaßen definiert (siehe [Ros97]):

$$\begin{aligned} \mathit{traces}(P \triangle_i Q) &= \mathit{traces}(P) && (5.79)\\ &\cup \{s \frown \langle i \rangle \frown t \mid \; s \in \mathit{traces}(P) \cap \Sigma^* \\ &\qquad\qquad\qquad\quad t \in \mathit{traces}(Q)\} \\ \mathit{failures}(P \triangle_i Q) &= \{(s, X) \mid \; (s, X) \in \mathit{failures}(P) \\ &\qquad\qquad\quad \wedge s \in \Sigma^* \wedge i \notin X\} && (5.80)\\ &\cup \{(s, X) \mid \; s \frown \langle \checkmark \rangle \in \mathit{traces}(P) \\ &\qquad\qquad\quad \wedge \checkmark \notin X\} && (5.81)\\ &\cup \{(s \frown \langle \checkmark \rangle, X) \mid \; s \frown \langle \checkmark \rangle \in \mathit{traces}(P)\} && (5.82)\\ &\cup \{(s \frown \langle i \rangle \frown t, X) \mid \; s \in \mathit{traces}(P) \cap \Sigma^* \\ &\qquad\qquad\quad \wedge (t, X) \in \mathit{failures}(Q)\} && (5.83) \end{aligned}$$

Aus der Definition der *failures* kann man ablesen:

5.80: solange P nicht terminiert hat ($s \in \Sigma^* \Longrightarrow \text{last}(s) \neq \checkmark$), kann $P \triangle_i Q$ den Ebent i nicht ablehnen ($i \notin X$).

5.81: wenn P terminiert, terminiert auch $P \triangle_i Q$ und $\checkmark$ kann nicht abgelehnt werden.

5.82: nach der Termination von P bzw. $P \triangle_i Q$ kann jeder Event abgelehnt werden - auch i.

5.83: solange P bzw. $P \triangle_i Q$ nicht terminiert ($s \in \mathit{traces}(P) \cap \Sigma^* \Longrightarrow \mathrm{last}(s) \neq \checkmark$), kann $P \triangle_i Q$ durch den Event i unterbrochen werden und verhält sich danach wie Q.

Die Regeln der operationellen Semantik von CSP für den Interrupt-Operator müssen um eine dritte Regel ergänzt werden, um das Terminationsverhalten korrekt (so wie in der denotationellen Semantik) wiederzugeben:

$$\frac{}{P \triangle_i Q \xrightarrow{i} Q} \quad \text{Interrupt-1} \tag{5.84}$$

$$\frac{P \xrightarrow{e} P'}{P \triangle_i Q \xrightarrow{e} P' \triangle_i Q} [i \neq e] \quad \text{Interrupt-2} \tag{5.85}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \triangle_i Q \xrightarrow{\checkmark} \Omega} \quad \text{Interrupt-3} \tag{5.86}$$

Auf den ersten Blick scheint der Vergleich der Transitionen von Statecharts mit den CSP-Interrupt-Operator zu hinken. In dem gezeigten Beispiel 5.34 gibt es in dem Prozeß X keine lokale Wahlmöglichkeit. Wäre z.B. auch $(A, c, B) \in T$ eine Transition im Statecharts von X, so reicht der hier beschriebene Interrupt-Operator nicht aus, um einen äquivalenten Prozeß in der Sprache CSP zu formulieren. Zumindest kann man nicht wie in Gleichung 5.75 alle Transitionen mit dem Interrupt-Operator darstellen. Die Formulierung nach Gleichung 5.74 wäre immerhin noch möglich. Gäbe es aber eine Transition (X, e, Y), so wäre die Übersetzung auf diesem Wege gänzlich unmöglich. Der Interrupt-Operator ist nicht assoziativ, so daß $(P \triangle_i Q) \triangle_e R \neq P \triangle_i (Q \triangle_e R)$ ist. Die Formulierung $(X \triangle_i Y) \triangle_e Y$ würde daher nicht das gewünschte Verhalten ausdrücken. Man benötigt eine Version des Interrupt-Operators, die es ermöglicht, eine Menge von Events anzugeben, die die Unterbrechung bewirken sollen. Glücklicherweise wird in [Sca95] ein CSP-Interrupt-Operator beschrieben, der diese Eigenschaft hat. Wir kommen darauf später zurück, wenn wir die CSP-Übersetzung von Statecharts mit hierarchischer Dekomposition behandeln. Im Moment dient uns der Interrupt-Operator lediglich als Denkanstoß, um die Semantik von Transitionen in Statecharts mit hierarchischer Dekomposition zu erfassen.

Ein weiterer Widerspruch könnte darin gesehen werden, daß $P \triangle_i Q$ terminiert, wenn P terminiert. In [BRJ97] werden sogenannte completion transitions beschrieben. Dies sind Transitionen, die ausgeführt werden, wenn der Quell-Zustand bzw. das Quell-Statecharts einen Endzustand erreicht. Das Erreichen eines Endzustands löst einen completion event aus, der im umgebenden Statecharts wahrgenommen wird und die completion transition aktiviert. Der completion event wird i.d.R. nicht explizit als Trigger-Event im Label einer completion transition angegeben. Ein Beispiel ist in der Abbildung 5.36 zu sehen.

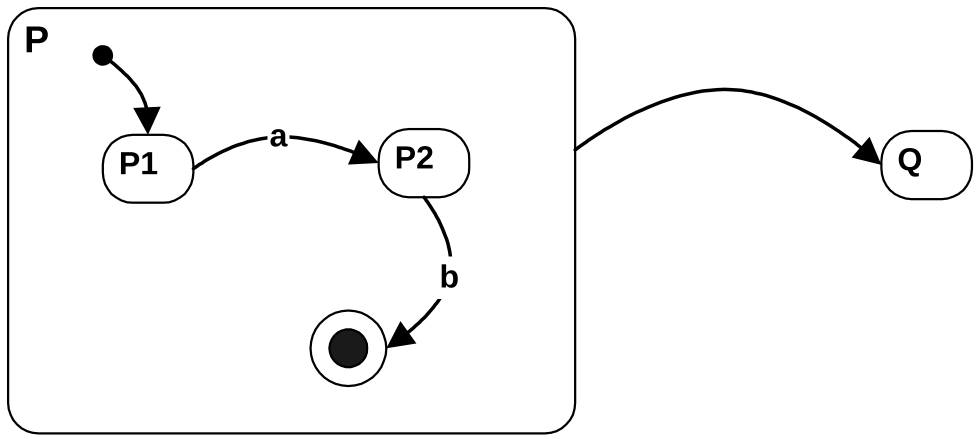


Abbildung 5.36: Completion transition

Eine bedeutungserhaltende Übersetzung nach CSP wäre in der sequentiellen Komposition $P;\ Q$ zu sehen.

Sequentielle Komposition Für die sequentielle Komposition von Prozessen git es in CSP einen expliziten Operator ; . Die operationelle Semantik für den Operator wird durch die folgenden Regeln beschrieben:

$$\frac{P \xrightarrow{x} P'}{P\ ;\ Q \xrightarrow{x} P'\ ;\ Q}[x \neq \checkmark] \quad \text{Sequence-1} \tag{5.87}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P\ ;\ Q \xrightarrow{\tau} Q} \quad \text{Sequence-2} \tag{5.88}$$

(5.89)

wobei τ einen internen Event anzeigt, der von der Umgebung nicht wahrgenommen wird.

Completion transitions sind offenbar gesondert zu behandeln, haben aber auch in CSP ein entsprechendes Gegenstück. Die Bedeutung entspricht nicht wie bei den übrigen Transitionen der des Interrupt-Operators, sondern der sequentiellen Komposition.

Definition Statecharts mit hierarchischer Dekomposition haben nun folgenden syntaktischen Aufbau:

$$\begin{aligned}
\mathit{Stop} &\in L_{SC} && (5.90)\\
\mathit{Skip} &\in L_{SC} && (5.91)\\
\Omega &\in L_{SC} && (5.92)\\
(\Sigma, E, T, \sigma_0, F) &\in L_{SC} && (5.93)\\
\mathit{mit}\ \ & \Sigma \in \wp(L_{SC}) = 2^{L_{SC}}\\
& E \in \wp(L_{Ev})\\
& T \in \wp(L_{Tr})\\
& \sigma_0 \in \Sigma\\
& \Sigma \in \wp(L_{SC})
\end{aligned}$$

Die Sprache der Statecharts ist dann definiert als die kleinste Lösung der rekursiven Mengengleichung:

$$\begin{aligned}
L_{SC} &= \mathit{Stop} + \mathit{Skip} + \Omega && (5.94)\\
&\quad +(\wp(L_{SC}) \times \wp(L_{Ev}) \times \wp(L_{Tr}) \times L_{SC} \times \wp(L_{SC}))
\end{aligned}$$

An die Stelle der Transitionsregel 5.59 treten jetzt zwei neue Regeln:

$$\frac{S \xrightarrow{e} S'}{(\Sigma, E, T, S, F) \xrightarrow{e} (\Sigma, E, T, S', F)}[\forall\, \sigma' \in \Sigma : (S, e, \sigma') \notin T] \quad \text{Transition-1} \tag{5.95}$$

$$\frac{}{(\Sigma, E, T, \sigma, F) \xrightarrow{e} (\Sigma, E, T, \sigma', F)}[(\sigma, e, \sigma') \in T] \quad \text{Transition-2} \tag{5.96}$$

(5.97)

Die Regel 5.95 beschreibt die Ausführung einer internen Transition im Zustand bzw. Statechart S. Falls S ein elementarer Zustand ist, dann wird er durch das Statechart *Stop* repräsentiert. Die Regel 5.95 kann dann nie angewendet werden, da *Stop* keine Transitionen ausführt. Wird S aber durch ein eigenes Statechart beschrieben, dann kann dieses Statechart eine Transition ausführen und geht dabei in ein anderes Statechart S' über. Die Bedingung für die Anwendbarkeit der Regel ist, daß der Zustand S nicht die Quelle einer Transition ist, die durch das Ereignis e ausgelöst wird.

Falls es im Statechart eine Transition gibt, deren Quelle der Anfangszustand ist und die durch e ausgelöst wird, muß Regel 5.96 angewendet werden. Das Statechart geht durch die Regelanwendung in ein neues Statechart mit dem Anfangszustand σ' über, wobei dies der Zielzustand der auszuführenden Transition ist.

Wir definieren ferner, daß mit $S \in \Sigma$ auch alle $S' \in \Sigma$ sind, für die gilt, daß $S \xrightarrow{*} S'$ eine Ableitung für S ist bzw. S läßt sich durch Regelanwendung in S' überführen. D.h.

$$\begin{aligned} & \forall\, S \in \Sigma : \\ \Sigma \quad & \supseteq \quad \{S' \mid S \xrightarrow{*} S'\} \qquad (5.98) \\ & \qquad (5.99) \end{aligned}$$

Jedes Statechart steht in diesem Sinne für eine Menge von Statecharts, nämlich alle Statecharts, die sich durch Anwendung der Regeln unserer operationellen Semantik aus dem Statechart gewinnen lassen.

Die Regel 5.96 ist noch nicht ganz korrekt, denn ein Zustand S kann nach Regel 5.95 in einen anderen Zustand (bzw. ein anderes Statechart) übergehen und die Transitionen des Statecharts, zu dem der ursprüngliche Zustand S gehört, beziehen sich immer noch auf S - nicht aber auf S'. Es gibt mindestens zwei Möglichkeiten, dies zu korrigieren. Man kann jedes Vorkommen von S bei Anwendung von Regel 5.95 durch S' substituieren. Alternativ erweitert man die Bedingung der Anwendbarkeit von Regel 5.96 wie folgt.

$$\frac{\sigma \xrightarrow{*} \sigma'}{(\Sigma, E, T, \sigma', F) \xrightarrow{e} (\Sigma, E, T, \sigma'', F)}\, [(\sigma, e, \sigma'') \in T] \quad \text{Transition-2} \qquad (5.100)$$

Dabei soll gelten $\sigma \xrightarrow{0} \sigma$ ist eine Ableitung der Länge 0 und $\sigma \xrightarrow{*} \sigma'$ ist eine Ableitung beliebiger Länge für das Statechart, daß dem Zustand σ zugeordnet ist.

Jetzt kann man auch sogenannte stubbed transitions korrekt handhaben. Es geht dabei um Transitionen, deren Quelle oder Ziel nicht ein elementarer Zustand im selben Statechart oder der Rand eines Statecharts ist, sondern ein interner Zustand in einem anderen Statechart. Ein Beispiel für diese Art von Transitionen ist in der Abbildung 5.37 gezeigt.

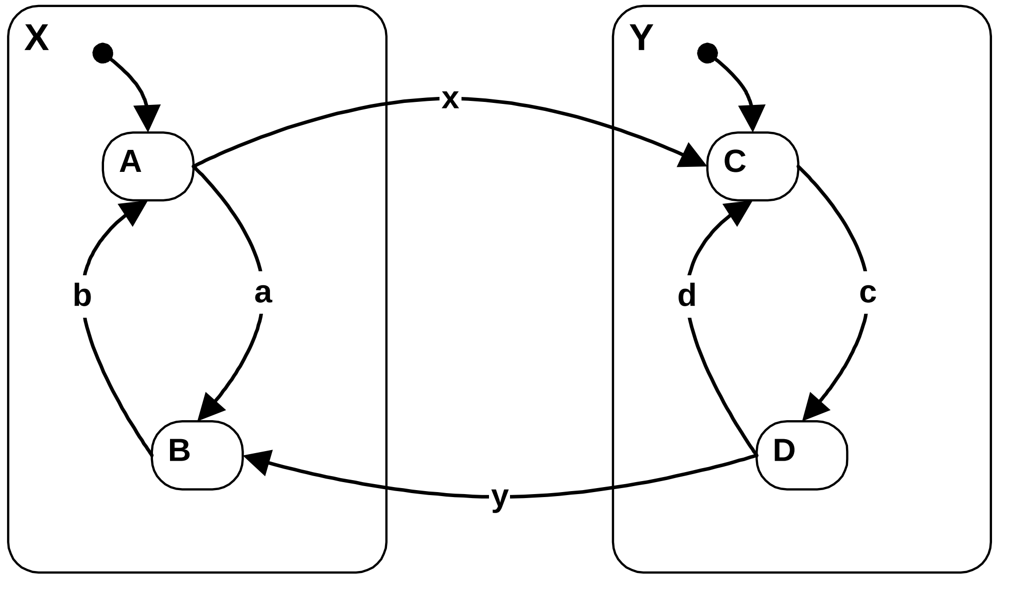


Abbildung 5.37: Stubbed Transitions

In dem Beispiel der Abbildung 5.37 gilt

$$X = (\{A,B\},\{a,b\},\{(A,a,B),(B,b,A)\},A,\emptyset) \quad (5.101)$$

$$Y = (\{C,D\},\{c,d\},\{(C,c,D),(D,d,C)\},C,\emptyset) \quad (5.102)$$

und für das umgebende Statechart gilt:

$$\begin{aligned}\Sigma = \{ & (\Sigma_X, E_X, T_X, A, \emptyset),\\ & (\Sigma_X, E_X, T_X, B, \emptyset),\\ & (\Sigma_Y, E_Y, T_Y, C, \emptyset),\\ & (\Sigma_Y, E_Y, T_Y, D, \emptyset)\}\end{aligned} \quad (5.103)$$

Die stubbed transitions lassen sich dann als $((\Sigma_X, E_X, T_X, A, \emptyset), x, (\Sigma_Y, E_Y, T_Y, C, \emptyset))$ und $((\Sigma_Y, E_Y, T_Y, D, \emptyset), y, (\Sigma_X, E_X, T_X, B, \emptyset))$ angeben.

Während Transitionen, die sich auf den Rand eines Statecharts S beziehen automatisch für alle aus diesem Statechart ableitbaren Statecharts S' gelten (vgl. Interrupt-Semantik der CSP-Transformation, Regel Transition-2), beziehen sich stubbed transitions auf das Statechart, das man aus S erhält, wenn man den Anfangszustand durch den Zustand ersetzt, auf den sich die Transition bezieht. Dabei ist nicht unbedingt vorrauszusetzen, daß dieses Statechart auch durch Regelanwendung aus S ableitbar ist. Wir definieren daher

$$\begin{aligned} & \forall(\Sigma', E', T', \sigma_0', F') \in \Sigma :\\ \Sigma \supseteq & \{(\Sigma', E', T', \sigma', F') \mid \sigma' \in \Sigma'\}\end{aligned} \quad (5.104)$$

und ändern die Regel 5.100 erneut zu

$$\frac{}{(\Sigma, E, T, \sigma_1, F) \xrightarrow{e} (\Sigma, E, T, \sigma', F)} \left[\begin{array}{l}((\sigma = \sigma_1)\vee\\ ((\sigma_1 = (\Sigma', E', T', \sigma'', F'))\wedge\\ (\sigma \in \{(\Sigma', E', T', \sigma_0', F') \mid \sigma_0' \in \Sigma'\})))\\ \wedge\\ (\sigma, e, \sigma') \in T\end{array}\right] \text{Transition-2} \quad (5.105)$$

Um auch die Möglichkeit sequentieller Komposition zu berücksichtigen, benötigen wir eine weitere Regel.

$$\frac{S_1 \xrightarrow{\checkmark} S_2}{(\Sigma, E, T, S_1, F) \xrightarrow{\tau} (\Sigma, E, T, S', F)} \left[\begin{array}{l}((S = S_1)\vee\\ ((S_1 = (\Sigma', E', T', \sigma', F'))\wedge\\ (S \in \{(\Sigma', E', T', \sigma_0', F') \mid \sigma_0' \in \Sigma'\})))\\ \wedge\\ (S, \checkmark, S') \in T\end{array}\right] \text{Transition-3} \quad (5.106)$$

Ferner soll das Statechart terminieren können, wenn es keine completion transition gibt.

$$\frac{S_1 \xrightarrow{\checkmark} S_2}{(\Sigma, E, T, S_1, F) \xrightarrow{\checkmark} \Omega} \left[\begin{array}{l} \nexists\, S, S' \in \Sigma : \\ ((S = S_1) \vee \\ ((S_1 = (\Sigma', E', T', \sigma', F')) \wedge \\ (S \in \{(\Sigma', E', T', \sigma'_0, F') \mid \sigma'_0 \in \Sigma'\}))) \\ \wedge \\ (S, \checkmark, S') \in T \end{array}\right] \quad \text{Transition-4} \tag{5.107}$$

Die ersten beiden Transitionsregeln müssen noch einmal geändert werden, um die Sonderrolle des ✓-Events zu berücksichtigen. Insgesamt haben wir jetzt die folgenden Regeln, die die operationelle Semantik von Statecharts mit hierarchischer Dekomposition, sequentieller Komposition und Termination beschreiben.

$$\text{Skip-Regel} \qquad \frac{}{\mathit{Skip} \xrightarrow{\checkmark} \Omega} \tag{5.108}$$

Interne Transition

$$\frac{S \xrightarrow{e} S'}{(\Sigma, E, T, S, F) \xrightarrow{e} (\Sigma, E, T, S', F)} [e \neq \checkmark \wedge \forall \sigma' \in \Sigma : (S, e, \sigma') \notin T] \tag{5.109}$$

Interrupt Transition

$$\frac{}{(\Sigma, E, T, \sigma_1, F) \xrightarrow{e} (\Sigma, E, T, \sigma', F)} \left[\begin{array}{l} e \neq \checkmark \\ \wedge \\ ((\sigma = \sigma_1) \vee \\ ((\sigma_1 = (\Sigma', E', T', \sigma'', F')) \wedge \\ (\sigma \in \{(\Sigma', E', T', \sigma'_0, F') \mid \sigma'_0 \in \Sigma'\}))) \\ \wedge \\ (\sigma, e, \sigma') \in T \end{array}\right] \tag{5.110}$$

Sequentielle Komposition

$$\frac{S_1 \xrightarrow{\checkmark} S_2}{(\Sigma, E, T, S_1, F) \xrightarrow{\tau} (\Sigma, E, T, S', F)} \left[\begin{array}{l} ((S = S_1) \vee \\ ((S_1 = (\Sigma', E', T', \sigma', F')) \wedge \\ (S \in \{(\Sigma', E', T', \sigma'_0, F') \mid \sigma'_0 \in \Sigma'\}))) \\ \wedge \\ (S, \checkmark, S') \in T \end{array}\right] \tag{5.111}$$

Termination

$$\frac{S_1 \xrightarrow{\checkmark} S_2}{(\Sigma, E, T, S_1, F) \xrightarrow{\checkmark} \Omega} \left[\begin{array}{l} \neg\, \exists S, S' \in \Sigma : \\ ((S = S_1) \vee \\ ((S_1 = (\Sigma', E', T', \sigma', F')) \wedge \\ (S \in \{(\Sigma', E', T', \sigma'_0, F') \mid \sigma'_0 \in \Sigma'\}))) \\ \wedge \\ (S, \checkmark, S') \in T \end{array}\right] \tag{5.112}$$

Übersetzung nach CSP Die Transitionen hierarchisch strukturierter Statecharts lassen sich unter Verwendung des in [Sca95] beschriebenen Interrupt-Operators in CSP formulieren. Der Interrupt-Operator ist wie folgt definiert.

$$\frac{P \xrightarrow{x} P'}{P \bigtriangleup Q \xrightarrow{x} P' \bigtriangleup Q} [x \neq \checkmark] \quad \text{Interrupt-1} \tag{5.113}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \bigtriangleup Q \xrightarrow{\checkmark} P'} \quad \text{Interrupt-2} \tag{5.114}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \bigtriangleup Q \xrightarrow{\tau} P \bigtriangleup Q'} \quad \text{Interrupt-3} \tag{5.115}$$

$$\frac{Q \xrightarrow{a} Q'}{P \bigtriangleup Q \xrightarrow{a} Q'} \quad \text{Interrupt-4} \tag{5.116}$$

Die Übersetzung eines Statecharts in einen äquivalenten CSP-Prozeß wird in Form einer Funktion $sc2csp : L_{SC} \rightarrow L_{CSP}$ angegeben.

$$\begin{aligned}
sc2csp(\mathit{Stop}) &= \mathit{Stop} && (5.117)\\
sc2csp(\mathit{Skip}) &= \mathit{Skip} && (5.118)\\
sc2csp((\Sigma, E, T, \sigma_0, F)) &= \mu\, P_{\sigma_0}.sc2csp(\sigma_0) && (5.119)\\
&\quad \bigtriangleup \\
&\quad (sc2csp((\Sigma, E, T, \sigma_0, F), \{(e, \sigma) \mid (\sigma_0, e, \sigma) \in T\})) \\
sc2csp((\Sigma, E, T, \sigma_0, F), \emptyset) &= \mathit{Stop} && (5.120)\\
sc2csp((\Sigma, E, T, \sigma_0, F), I \cup \{(e, \sigma)\}) &= sc2csp((\Sigma, E, T, \sigma_0, F), I) && (5.121)\\
&\quad \Box \\
&\quad e \rightarrow sc2csp((\Sigma, E, T, \sigma, F))
\end{aligned}$$

Ein Problem tritt auf, wenn das Statechart $(\Sigma, E, T, \sigma_0, F)$ auch eine completion transition mit σ_0 als Quellzustand hat. Die Situation ist in dem Beispiel 5.38 gezeigt.

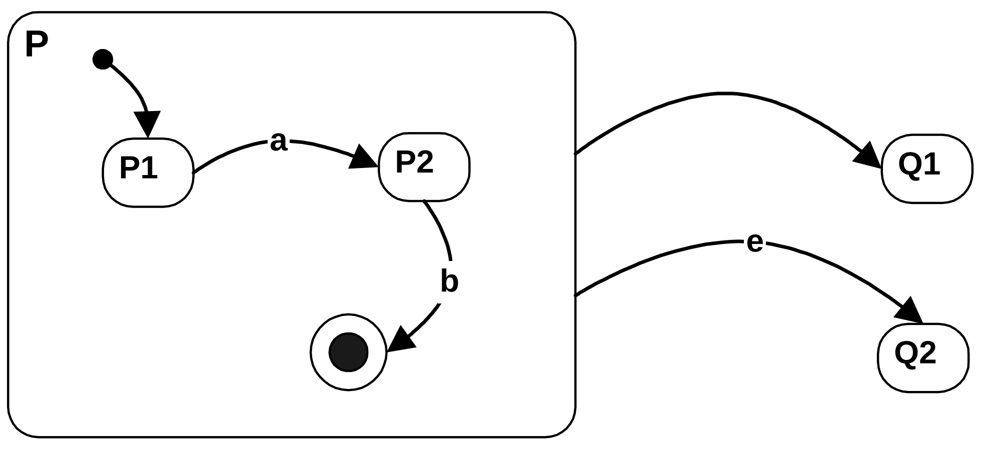


Abbildung 5.38: Completion transition

Eine CSP-Formulierung für das Statechart P ist:

$$\begin{aligned} P &= \mathit{Stop} \bigtriangleup (\mathit{Stop} \,\Box\, a \rightarrow (\mathit{Stop} \bigtriangleup (\mathit{Stop} \,\Box\, b \rightarrow \mathit{Skip})))) && (5.122)\\ &= \mathit{Stop} \bigtriangleup (a \rightarrow (\mathit{Stop} \bigtriangleup (b \rightarrow \mathit{Skip}) && (5.123)\\ &= a \rightarrow b \rightarrow \mathit{Skip} && (5.124) \end{aligned}$$

Die Vereinfachung wurde mit Hilfe der folgenden Regeln durchgeführt:

$$\begin{aligned} \mathit{Stop} \,\Box\, P &= P && (5.125)\\ \mathit{Stop} \bigtriangleup P &= P && (5.126) \end{aligned}$$

Das umgebende Statechart werde mit S bezeichnet. Der zugehörige CSP-Prozeß enthält dann die Transition mit dem Label e, die wir wie vorher mit dem Interrupt-Operator formulieren können: $P \bigtriangleup (e \rightarrow Q_2)$. Nach der Termination von P greift in dem Statechart S die completion transition und es findet ein Übergang in den Zustand Q_1 statt. Dies würde der sequentiellen Komposition $P\,;\,Q_1$ entsprechen. Die zusammengesetzte Formulierung $(P \bigtriangleup (e \rightarrow Q_2))\,;\,Q_1$ entspricht nicht der Statechart- Semantik, da nach Termination von Q_2 nicht automatisch Q_1 ausgeführt werden soll. Auch $(P\,;\,Q_1) \bigtriangleup (e \rightarrow Q_2)$ trifft nicht zu, denn es gibt keine Transition von Q_1 mit dem Event e nach Q_2 im Statechart. Eine mögliche Lösung ist:

$$(((P\,;\,x \rightarrow \mathit{Skip}) \bigtriangleup (e \rightarrow Q_2)) \,|[\,\{x\}\,]|\, (x \rightarrow Q_1)) \setminus \{x\} \qquad (5.127)$$

Dabei wurde der completion-event ($\checkmark$) des Statecharts ersetzt durch einen neuen Event x gefolgt vom Prozeß *Skip*. Der neue Event x dient ausschließlich der internen Synchronisation der parallelen Prozesse $(P\,;\,x \rightarrow \mathit{Skip}) \bigtriangleup (e \rightarrow Q_2)$ und $x \rightarrow Q_1$. Er wird vor der Umgebung mittels Verwendung des Hiding-Operators $\setminus \{x\}$ verborgen. Die Synchronisation der parallelen Prozesse in dem Konstrukt $P \,|[\,A\,]|\, Q$ erfolgt ausschließlich über die Events in der Menge A. D.h. in dem Beispiel kann der Event x nur dann auftreten, wenn beide parallelen Prozesse ihn gleichzeitig ausführen können. Die übrigen Events des Alphabets der Prozesse stehen für jeden der beiden Prozesse frei zur Kommunikation mit der Umgebung zur Verfügung. Der Event x tritt auf, wenn P terminiert hat. Das Auftreten des Events wird für die Umgebung nicht sichtbar und dient lediglich der internen Synchronisation - sozusagen als Startsignal für Q_1. Falls vor Termination von P der Event e auftritt, wird Q_2 ausgeführt und x tritt niemals auf, denn x soll eine neuer Event sein, der weder im Alphabet von S noch von P oder Q_1 oder Q_2 enthalten ist. Wenn P terminiert hat, tritt zunächst der interne Event x auf und dann terminiert die linke Hälfte der parallelen Komposition inklusive des Interrupt-Konstrukts: $\mathit{Skip} \bigtriangleup (e \rightarrow Q_2)$. Leider ist noch möglich, daß nach dem Event x der Event e auftitt und das ganze Konstrukt zu $Q_2 \,|[\,\{x\}\,]|\, Q_1$ mutiert. Das ist nicht erwünscht und muß korrigiert werden. Aber ansonsten gilt: wenn *Skip* terminiert hat, dann auch $\mathit{Skip} \bigtriangleup (e \rightarrow Q_2)$ und die parallele Komposition terminiert dann, wenn Q_1 terminiert hat. Andererseits terminiert das parallele Konstrukt nicht, wenn Q_2 terminiert, weil $x \rightarrow Q_1$ nie terminieren kann, wenn x nicht auftritt. Auch das muß noch korrigiert werden. Nun zur Korrektur der verbliebenen Fehler:

$$\begin{aligned} &((((P\,;\,x \rightarrow \mathit{Skip}) \bigtriangleup ((z \rightarrow e \rightarrow Q_2) \,\Box\, (y \rightarrow Q_1))) && (5.128)\\ &|[\,\{x, y, z\}\,]|\\ &((x \rightarrow y \rightarrow \mathit{Skip}) \,\Box\, (z \rightarrow \mathit{Skip}))) \setminus \{x, y, z\} \end{aligned}$$

Jetzt wird der Event x verwendet, um den Zweig $y \rightarrow Q_1$ in der deterministischen Auswahl auf der rechten Seite des Interrupt-Operators zu ermöglichen. Nach x kann kein anderer Event als y auftreten. Dies wird durch die Definition des rechten Operanden des Parallel-Operators sichergestellt. Wenn noch kein x aufgetreten ist, kann das parallele Konstrukt stets z ausf"hren und danach e. Die Synchronisation findet ausschließlich mit Hilfe der internen Events x, y statt. Da diese vor der Umgebung versteckt werden, sind sie in den Failures des Prozesses nicht sichtbar.

Die Übersetzungsfunktion wird nach o.g. Schema erweitert. Wir verwenden dabei die Funktion $choices(\sigma) = \{e \mid (\sigma, e, \sigma') \in T\}$, um alle Events zu erhalten, die im Zustand σ auftreten können.

$$sc2csp(Stop) = Stop \tag{5.129}$$

$$sc2csp(Skip) = Skip \tag{5.130}$$

$$\begin{aligned}
&sc2csp((\Sigma, E, T, \sigma_0, F)) = \\
&\quad \mu\, P_{\sigma_0}.(\\
&\qquad sc2csp(\sigma_0) \\
&\qquad \triangle \\
&\qquad sc2csp((\Sigma, E, T, \sigma_0, F), \{(e, \sigma) \mid (\sigma_0, e, \sigma) \in T\}) \\
&\quad) \qquad \text{, falls } \checkmark \notin choices(\sigma_0)
\end{aligned} \tag{5.131}$$

$$\begin{aligned}
&sc2csp((\Sigma, E, T, \sigma_0, F)) = \\
&\quad \mu\, P_{\sigma_0}.(\\
&\qquad (\\
&\qquad\quad (sc2csp(\sigma_0)\,;\ x \to Skip) \\
&\qquad\quad \triangle \\
&\qquad\quad (\quad (z \to sc2csp((\Sigma, E, T, \sigma_0, F), \{(e, \sigma) \mid (\sigma_0, e, \sigma) \in T \wedge e \neq \checkmark\})) \\
&\qquad\qquad \square \\
&\qquad\qquad (sc2csp((\Sigma, E, T, \sigma_0, F), \{(y, \sigma) \mid (\sigma_0, \checkmark, \sigma) \in T\}) \\
&\qquad\quad) \\
&\qquad) \\
&\qquad |[\, \{x, y, z\}\,]| \\
&\qquad (\\
&\qquad\quad (x \to y \to Skip) \,\square\, (z \to Skip) \\
&\qquad) \\
&\quad) \setminus \{x, y, z\}) \qquad \text{, falls } \checkmark \in choices(\sigma_0)
\end{aligned} \tag{5.132}$$

$$sc2csp((\Sigma, E, T, \sigma_0, F), \emptyset) = Stop \tag{5.133}$$

$$\begin{aligned}
&sc2csp((\Sigma, E, T, \sigma_0, F), I \cup \{(e, \sigma)\}) = \\
&\quad sc2csp((\Sigma, E, T, \sigma_0, F), I) \\
&\quad \square \\
&\quad e \to sc2csp((\Sigma, E, T, \sigma, F))
\end{aligned} \tag{5.134}$$

So wie hier definiert, terminiert der Übersetzungsprozeß nicht unbedingt, da ein Statechart auch rekursive Transitionen beinhalten kann. Es wird jedoch angedeutet, daß CSP-Prozesse benannt $\mu\, P_{\sigma_0}$ und ebenfalls rekursiv definiert werden können. P_{σ_0} soll ein eindeutiger Name für das Statechart im Zustand σ_0 sein.

Anmerkung: Der parallele Hilfprozeß kann als Slave dargestellt werden. Bezogen auf das Beispiel der Abbildung 5.38 kann ein äquivalenter CSP-Prozeß auch mittels Enslavement formuliert werden. Sei $P' = (P\,;\ x \to Skip) \triangle ((z \to e \to$

$Q_2) \square (y \rightarrow Q_1))$ und $Q' = (x \rightarrow y \rightarrow \mathit{Skip}) \square (z \rightarrow \mathit{Skip})$. Dann lautet der gesuchte Prozeß $(P' \,|[\, \{x, y, z\} \,]|\, Q') \setminus \{x, y, z\} = P' \,//_{\{x,y,z\}}\, Q'$.

5.3.8 Parallele Komposition

In der CSP-Transformation des vorangegangenen Abschnitts haben wir bereits die parallele Komposition von CSP-Prozessen eingesetzt. Für Statecharts gibt es ein ähnliches Konstrukt, die AND-Komposition. Ein AND-Statecharts besteht aus zwei Statecharts, die in der virsuellen Darstellung mit einer durchbrochenen Linie voneinander getrennt sind. Ein Beispiel wird in der Abbildung 5.39 gegeben.

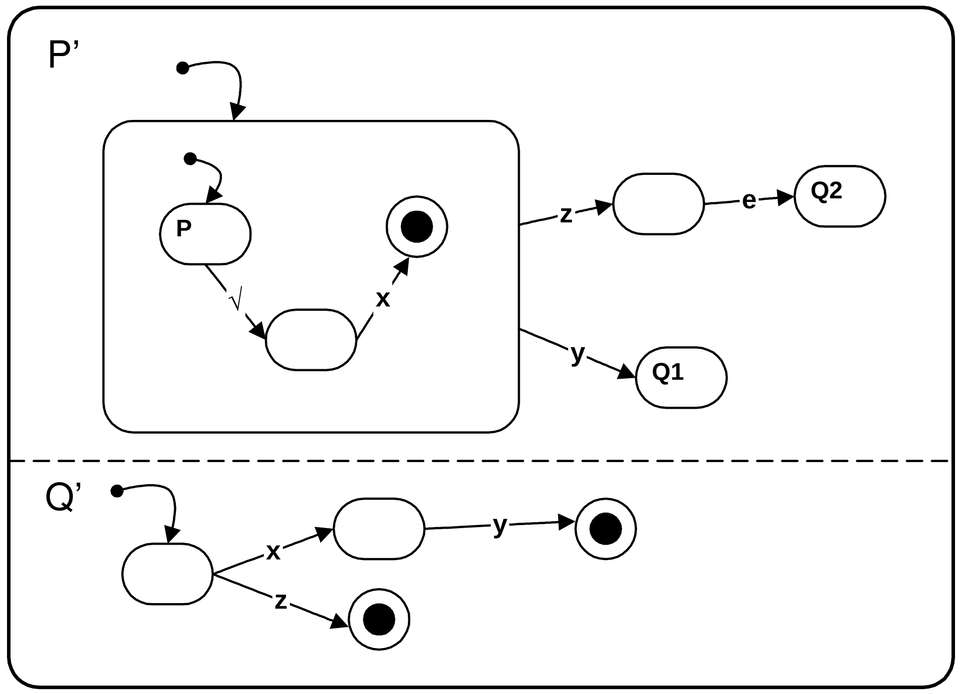


Abbildung 5.39: Parallele Komposition (AND-Statechart)

Bis auf das Auftreten der Events y, x, z ist das Statechart Äquivalent zu dem der Abbildung 5.38. Es repräsentiert die Rückübersetzung des CSP-Prozesses aus Gleichung 5.128.

Die Semantik der AND-Statecharts definieren wir wie die parallele Komposition von CSP-Prozessen. Für die textuelle Darstellung eines AND-Statecharts mit den Komponenten P, Q wird die für CSP gebräuchliche Schreibweise $P \parallel Q$ vereinbart.

Parallele Komposition - 1 (unsynchronisierter Event) (5.135)

$$\frac{(\Sigma_1, E_1, T_1, \sigma_1, F_1) \xrightarrow{e} (\Sigma_1, E_1, T_1, \sigma_1', F_1)}{\begin{array}{l}(\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2) \\ \xrightarrow{e} (\Sigma_1, E_1, T_1, \sigma_1', F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2)\end{array}} \left[\begin{array}{l} e \neq \checkmark \\ \wedge\ e \notin E_1 \cap E_2 \end{array}\right] \tag{5.135}$$

Parallele Komposition - 2 (unsynchronisierter Event) (5.136)

$$\frac{(\Sigma_2, E_2, T_2, \sigma_2, F_2) \xrightarrow{e} (\Sigma_2, E_2, T_2, \sigma_2', F_2)}{\begin{array}{l}(\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2) \\ \xrightarrow{e} (\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2', F_2)\end{array}} \left[\begin{array}{l} e \neq \checkmark \\ \wedge\ e \notin E_1 \cap E_2 \end{array}\right] \tag{5.136}$$

Parallele Komposition - 3 (Synchronisation) (5.137)

$$\frac{\begin{array}{c}(\Sigma_1, E_1, T_1, \sigma_1, F_1) \xrightarrow{e} (\Sigma_1, E_1, T_1, \sigma_1', F_1) \\ (\Sigma_2, E_2, T_2, \sigma_2, F_2) \xrightarrow{e} (\Sigma_2, E_2, T_2, \sigma_2', F_2)\end{array}}{\begin{array}{l}(\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2) \\ \xrightarrow{e} (\Sigma_1, E_1, T_1, \sigma_1', F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2', F_2)\end{array}} \left[\begin{array}{l} e \neq \checkmark \\ \wedge\ e \in E_1 \cap E_2 \end{array}\right] \tag{5.137}$$

Parallele Komposition - 4 (interner Event) (5.138)

$$\frac{(\Sigma_1, E_1, T_1, \sigma_1, F_1) \xrightarrow{\epsilon} (\Sigma_1, E_1, T_1, \sigma_1', F_1)}{\begin{array}{l}(\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2) \\ \xrightarrow{\epsilon} (\Sigma_1, E_1, T_1, \sigma_1', F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2)\end{array}} \tag{5.138}$$

Parallele Komposition - 5 (interner Event) (5.139)

$$\frac{(\Sigma_2, E_2, T_2, \sigma_2, F_2) \xrightarrow{\epsilon} (\Sigma_2, E_2, T_2, \sigma_2', F_2)}{\begin{array}{l}(\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2, F_2) \\ \xrightarrow{\epsilon} (\Sigma_1, E_1, T_1, \sigma_1, F_1) \parallel (\Sigma_2, E_2, T_2, \sigma_2', F_2)\end{array}} \tag{5.139}$$

Parallele Komposition - 6 (Termination) (5.140)

$$\frac{S_1 \xrightarrow{\checkmark} S_1'}{S_1 \parallel S_2 \xrightarrow{\epsilon} \Omega \parallel S_2} \tag{5.140}$$

Parallele Komposition - 7 (Termination) (5.141)

$$\frac{S_2 \xrightarrow{\checkmark} S_2'}{S_1 \parallel S_2 \xrightarrow{\epsilon} S_1 \parallel \Omega} \tag{5.141}$$

Parallele Komposition - 8 (Termination) (5.142)

$$\frac{}{\Omega \parallel \Omega \xrightarrow{\checkmark} \Omega}$$

Während es in CSP Varianten des Operators gibt, bei denen das zur Synchronisation verwendete Interface (die Menge der gemeinsamen Events) explizit angegeben werden kann, ist dies für Statecharts i.A. nicht üblich. Eine Ausnahme ist hierbei die Statechart-Variante Mini-Statecharts (siehe [DN96b, DN96c, DN96a, Sch96, JP97]). Ansonsten wird implizit die Schnittmenge der Alphabete der beteiligten Prozesse als synchronsiertes Interface angenommen.

Es erweist sich als Handicap, das es für Statecharts darüberhinaus wenig Möglichkeiten gibt, das externe Interface bzw. Alphabet eines Statecharts zu manipulieren. Auch hier stellen Mini-Statecharts eine Ausnahme dar. In CSP stehen hierfür der Hiding-Operator sowie die Möglichkeit der expliziten Erweiterung der zum Alphabet gehörenden Event-Menge zur Verfügung.

Interne Transitionen Bereits in der Regel 5.111 (Sequentielle Komposition) wurd τ als ein interner Event verwendet, der durch Regelanwendung produziert werden kann.

In Abschnitt 5.3.4 wurde erklärt, daß ϵ als ein spezieller Event für Transitionen verwendet werden kann. Diese Transitionen können dann spontan und ohne Kommunikation mit der Umgebung ausgeführt werden.

Beide Events sind gleichbedeutend. Während τ übicherweise in der Literatur zu CSP verwendet wird, findet man die Bezeichnung ϵ in der Automatentheorie und im Zusammenhang mit Statecharts.

Die ϵ-Events gehören nicht zum Alphabet eines Prozesses. Sie werden nicht in den Traces eines Statecharts sichtbar.

Kapitel 6

Implementierung

6.1 Voraussetzungen

Das Projekt SIGN hat sich bei der Wahl einer Programmiersprache für die neue objektorientierte Programmiersprache JAVA [LL96] entschieden. Einer der Hauptgründe hierfür war die Plattformunabhängigkeit von JAVA (JAVA ist sowohl auf der Quell- als auch auf der Binärebene plattformunabhängig), was einer der wichtigsten Vorteile ist, die JAVA gegenüber anderen Programmiersprachen zu bieten hat. Insbesondere für Umgebungen, in denen auf verschiedenen Plattformen gearbeitet wird (wie z.B. im Projekt SIGN, wo die einzelnen Projektmitlieder das Programm sowohl an der Uni, als auch zu Hause oder bei ihrer Arbeitsstelle weiterentwickelt haben).

Da die Programmiersprache JAVA momentan noch in einem Entwicklungsstadium ist, hat sich das Projekt ursprünglich als einheitliche Basis für das Übersetzen des Programms auf SUN's Entwicklungsumgebung JDK (Java Development Kit) Version 1.1.1 festgelegt.

Da SUN seine Entwicklungsumgebung aber zwischenzeitlich weiterentwickelt hat (und es weiterhin tut) und neue Versionen des JDK herausbrachte, hat sich das Projekt SIGN natürlich auch diese Neuerungen und Überarbeitungen angeschaut. Aus dem Grund, daß diese Weiterentwicklungen einige für das Projekt SIGN wichtige Erkenntnisse und Fortschritte mit sich brachten (zumindest gab es wichtige BUG-Fixes, die mit der Version 1.1.1. des JDK nicht bereinigt wurden), arbeitete das Projekt später (einheitlich) nur noch auf der Version 1.2 (beta2) des JDKs.

Das Programm wurde unter Sun Solaris (ab Version 2.4) auf Workstations der Ultra SPARC-Klasse (Projektrechner) implementiert und getestet. Die Implementierung erfolgte wie bereits erwähnt unter Benutzung der frei verfügbaren JAVA Entwicklungsumgebung (samt dem JAVA Compiler javac) JDK 1.1.1-1.2 (beta 2) von SUN.

Da JAVA plattformunabhängig ist, läßt sich das Programm auch auf andere Systeme portieren (übersetzen und ausführen). Voraussetzung hierfür ist lediglich das Vorhandensein der JAVA Entwicklungsumgebung (JDK) mit der geeigneten Version (hier zur Zeit JDK 1.2).

Die JAVA Entwicklungsumgebung (JDK) wird von SUN momentan für die weiteren Betriebssysteme Windows95, Windows NT, OS/2 und Apple Macintosh bereitgestellt.

Das Übersetzen des Programms erfolgt in der Regel durch :

```
javac PROGRAMMNAME.java
```

wobei javac der JAVA Compiler ist und dahinter der Name des Hauptprogramms stehen muß. Nach dem fehlerfreien Übersetzen des Programms generiert JAVA eine .class Datei mit dem gleichen Namen, wie das Hauptprogramm. Dieses ist die Bytecode-Datei. Diese kann nun mit dem JAVA-Interpreter ausgeführt werden. Der im JDK enthaltene JAVA-Interpreter heißt schlicht *java*. Das Ausführen des Programms erfolgt also durch:

```
java PROGRAMMNAME
```

ohne Angabe der Endung .class.

Um das Programm erfolgreich zu kompilieren, sowie ausführen zu können, kann es erforderlich sein, den Klassenpfad (Classpath) zu setzen. Hierfür hat sich das Projekt SIGN ein Makefile geschrieben (gmake), um sich diese lästige Arbeit zu sparen. Dieses Makefile ist allerdings nur auf Systemen verwendbar, die das gmake(gnu-make) enthalten.

Wie es nun mal bei Projektarbeiten üblich ist, mußte jeder Projektteilnehmer natürlich auch Programmteile verstehen und nachvollziehen können, die andere implementiert haben. Um dieses zu vereinfachen und die Übersichtlichkeit des gesammten Programmcodes beizubehalten, wurde ein Stylguide (Programmierrichtlinien) festgelegt. Dieser soll im folgenden dargestellt werden.

6.2 Styleguide (Programmierrichtlinien)

6.2.1 Allgemeines

Aus Gründen der Lesbarkeit sollten die einzelnen Zeilen eine Zeilenlänge von 79 Zeichen nicht überschreiten. Somit sollte auch jeder Ausdruck gut lesbar sein.

6.2.2 Programmstruktur

6.2.2.1 Ebenen

Damit verschiedene Ebenen in der Programmstruktur besser aus dem Code ersichtlich sind, ist es sinnvoll, die untergeordneten Befehle in den nächsten Zeilen einzurücken. Dies sollte um eine einheitliche Weite geschehen: 4 Zeichen. Hier ein Beispiel:

```
while(Bedingung)
{
    Anweisung;
    ...
    Anweisung;
}
```

6.2.2.2 Leerzeichen

Leerzeichen sind nur an solchen Stellen einzufügen, an denen diese die Lesbarkeit des Quelltextes erhöhen. Dies ist der Fall nach trennenden Kommata. Störend sind Leerzeichen jedoch vor und nach öffnenden oder schließenden Klammern. Hier ist auf Leerzeichen zu verzichten. Auch hier ein Beispiel zur Verdeutlichung:

```
falsch:
Funktionsname   (Variable1,Variable2 );
if    (  Bedingung )

richtig:
Funktionsname(Variable1, Variable2);
if(Bedingung)
```

6.2.2.3 Programmblöcke

Programmblöcke sind durch geschwungene Klammern eingegrenzt. Logisch zusammengehörige Klammern stehen dabei immer in der gleichen Spalte, d.h. übereinander. Außerdem ist vor und nach jeder öffnenden und schließenden Klammer ein Zeilenumbruch durchzuführen. Der so geklammerte Text wird wie oben beschrieben entsprechend eingerückt. Es folgt ein Beispiel:

```
falsch:
while(Bedingung){
Ausdruck;
...
Ausdruck;}

richtig:
while(Bedingung)
{
    Anweisung;
    ...
    Anweisung;
}
```

6.2.3 Bezeichner

Um ein einheitliches Bild im Programmcode zu erhalten, ist es unerläßlich, daß alle Bezeichner in englischer Sprache gewählt werden. Schließlich sind auch die von JAVA mitgelieferten Klassen in Englisch gehalten. Jeder Bezeichner sollte einen intuitiven Namen bekommen, also bitte kein X4711FG42 oder ähnliches. Zur besseren Unterscheidung sollten sich Variablen, Methoden-, Klassen- und Objektnamen auch schon in der Schreibweise unterscheiden. Variablen beginnen immer mit kleinen Buchstaben. Dabei unterscheiden wir zwischen Methoden-Variablen und Member-Variablen (Variablen, die fuer das ganze Objekt gelten). Variablen lassen sich in vier Typ-Gruppen aufteilen: Numerisch, Zeichenketten, Controls und Objektreferenzen. Numerische Variablen beginnen mit einem „n"; mit Ausnahme bei Floats, diese beginnen mit einem „f". Zeichenketten beginnen mit „str", Controls mit „ctl" und Objektreferenzen mit „p". Danach folgt der Name der Variable. Dazu ein Beispiel:

```
    [...]
    public int exampleMethode(CString strText)
    {
        int           nIndex;
        double        nDoubleIndex;
        float         fIndex;
        String        strText;

        List          ctlListbox;
        Button        ctlButton;
        CShapeEntity  pShape;

        nIndex = 0;
        [...]
    }
    [...]
```

Member-Variablen beginnen mit „m_" und folgend wie oben beschrieben. Auch hier ein Beispiel:

```
public class CExampleFrame extends java.awt.Frame
{
    public CExampleFrame()
    {
        int nListCounter;

        super();
        setTitle("Example Frame");
        m_ctlButton = new Button("OK");
        m_ctlLBProject = new List();
        for (nListCounter = 0; nListCounter < 10; nListCounter++)
        {
            [...]
        }
    }

    public int exampleMethode(CString strText)
    {
        [...]
    }
    [...]

    int           m_nIndex;

    Button        m_ctlButton;
    List          m_ctlLBProject;

    CShapeEntity  m_pRootShapeEntity;
}
```

Klassen- und Methodenvariablen sollen ebenfalls einen intuitiven Namen erhalten. Dabei beginnen Klassen stehts mit einem Großbuchstaben, Methoden hingegen mit Kleinbuchstaben. Es soll durch den Methodennamen die auszuführende Funktion ersichtlich sein. Die Klassennamen beginnen mit einem „C“ (für class), gefolgt vom eigentlichen Namen (siehe vorheriges Beispiel).

6.2.4 Kommentare

Wir unterscheiden hier zwischen Programmkommentaren (ein- und mehrzeilig) und Dokumentationskommentaren. Die erste Gruppe ist so einzusetzen, daß der Quellcode besser verständlich wird. Hier sind wichtige Algorithmen zu erklären (u.ä.). Die zweite Art von Kommentaren ist jedoch mindestens genau so wichtig, wenn nicht noch mehr. Eine Musterdatei, nach der solche Kommentare in die Dateien einzubinden sind, liegt unter:

`/home/stp/doc/richtlinien/Beispiel_JavaDoc.java.`

6.2.5 Dateien

Aus Gründen der Übersichtlichkeit ist es nötig, daß jede Datei nur eine Klasse enthält. Der Name der Datei hat dem Namen der Klasse zu entsprechen, wie auch schon von JAVA selbst gefordert wird.

6.3 Statistik

Das Projekt SIGN hat es schließlich auf einen Programm-Umfang von ca. 160 Klassen und ca. 33000 Zeilen Programm-Code gebracht.

Kapitel 7

Schlußwort

Nach zwei Jahren harter Arbeit ziehen wir nun einen Schlußstrich unter unsere gemeinsam verbrachte Zeit im Projekt.

Was ist für uns dabei herausgekommen? Nun, vor allen Dingen Erfahrung im Umgang miteinander und in der Organisation von (Software-) Projekten. Nicht immer war es leicht, Entscheidungen zu finden, mit denen alle einverstanden waren - so manches Plenum wurde durch Diskussionen in die Länge gezogen. Aber auch dies gehört zum Prozeß dazu, genauso wie unterschiedliche Charaktere das Team bereichert haben.

Des weiteren ist zu erwähnen, daß es immer Projektmitglieder gab, die für die anderen eingesprungen sind, wenn Hilfe benötigt wurde. So konnte das Projekt immer am Laufen gehalten werden. Wer auch immer aus welchem Grund ausfiel - jeder war für die anderen mit da, was die Teamarbeit erst ermöglicht hat.

Glücklicher Weise wurden wir von diversen Personen tatkräftig unterstützt - ohne deren Hilfe wäre so einiges nicht oder nur schwer möglich gewesen. Somit danken wir: Martin Gogolla, Mark Richters und Heino Gärtner, die als Betreuer fungiert haben, Herrn Samlowski für sein Projektmanagementseminar und Oliver Laumann, ohne den wir häufig ohne funktionsfähige Rechner dagesessen hätten.

Egal wie lange wir auf unserem Gebiet geforscht und gearbeitet haben, es bleiben immer noch Arbeiten offen, für die das Projekt entweder keine Zeit hatte oder die einfach die Projektziele überschritten. Aus diesem Grund wird es in Zukunft auch weitere studentische Projekte geben, die sich mit dem Softwareentwurf beschäftigen werden.

Aber nicht nur weitere studentische Projekte im Rahmen des Projektstudiums können hier genug Stoff für ihre Arbeit finden. Genauso wird sich die eine oder andere Diplomarbeit (von Projektmitgliedern) mit diesem Thema beschäftigen. Bleibt zu hoffen, daß diese genauso erfolgreich abgeschlossen werden, wie unser Projekt.

Leider ist die Zeit nach zwei Jahren um - genauso wie der Bericht am Ende angekommen ist. Auf alle Fälle hatten wir viel Spaß miteinander - nicht nur am Projekttag, und es ist wirklich schade, daß wir nicht mit unserer Arbeit in diesem Team weiterarbeiten können. Aber wer kann schon sagen, ob wir uns nicht später noch mal in der weiten Welt über den Weg laufen.

Anhang A

Anhang

A.1 Handbuch Graphischer Editor deSIGN

A.1.1 Einleitung

Herzlichen Glückwunsch, Ihre Wahl für einen graphischen Entwurfseditor fiel auf deSIGN aus dem Hause StP. Sie haben sich hiermit für ein Produkt entschieden, welches Sie bei der Entwicklung von objektorientierter Software und Datenbankschemata unterstützen wird.

Um Ihnen den Einstieg in dieses Programm zu erleichtern, ist dieses Handbuch geschrieben worden. In den folgenden Kapiteln werden Sie lesen, wie Sie deSIGN installieren, konfigurieren und ausführen können. Im dritten Kapitel geben wir Ihnen eine Übersicht über alle Programmfunktionen, die deSIGN Ihnen bietet, sowie eine kurze Einführung in die SIGN-Modelling-Language (SIGN-ML). Im vierten Kapitel werden wir Ihnen anhand eines Beispiels die Benutzung von deSIGN eingehend erklären. Kapitel fünf beschäftigt sich mit Problemen, die bei der Benutzung von deSIGN auftreten könnten. Des weiteren werden hier Fehlermeldungen beschrieben, die bei der Arbeit mit dem Programm auftreten können. Zu den Fehlermeldungen werden dann auch gleich entsprechende Behebungsmaßnahmen angegeben. Das Kapitel wird durch ein Stichwortverzeichnis abgeschlossen.

Wenn Sie dennoch Fragen zu SIGN oder deSIGN haben sollten, so wenden Sie sich bitte an das Programmierteam. Dieses ist per E-mail unter *stp@informatik.uni-bremen.de* zu erreichen.

Und nun wünscht Ihnen das komplette StP-Team viel Spaß mit deSIGN.

A.1.2 Technische Voraussetzungen

A.1.2.1 Installation und Systemanforderungen

Für die Benutzung von desIGN ist ein funktionstüchtiges Java Development Kit (JDK) erforderlich, da das Programm unter der objektorientierten Programmiersprache JAVA entwickelt wurde. In der aktuellen Version setzen wir Release 1.2beta2 voraus. Dies ist über das Internet direkt bei SUN erhältlich. Falls Sie nicht im Besitz dieser Software (dem Java-Compiler und dem Java-Interpreter) seien sollten, können Sie deSIGN leider nicht verwenden.

JDK ist unter anderem für folgende Plattformen erhältlich: Linux, Windows (95/NT/3.1x), Solaris 2.xx

Um Java auf Ihrem Rechner zu installieren, müssen Sie als erstes den Quellcode in den Bytecode übersetzen. Kopieren Sie sich dazu den deSIGN-Quellcode in ein Verzeichnis Ihrer Wahl (die Verzeichnisstruktur muß jedoch erhalten bleiben) und wechseln Sie dann in das Verzeichnis *hic/gui*. Um den Compiler zu starten geben Sie in der Eingabeaufforderung einfach *gmake ide* (Solaris/Linux) ein. Sollten Sie unter Windows arbeiten, ist dies ein wenig komplizierter: starten Sie ihren Java-Compiler mit der CLASSPATH-Option und tragen Sie dort den Pfad ein, unter dem Sie den Quellcode installiert haben. Diese Arbeit kann Ihnen abgenommen werden, wenn sie das *Makefile* verwenden können. Hier werden alle erforderlichen Parameter automatisch gesetzt. Sollten Sie dennoch Probleme bei der Installation haben, so wenden Sie sich bitte an Ihren Systemadministrator oder an das Projekt SIGN.

A.1.2.2 Programmausführung

Als erstes ist der Editor zu starten. Dies sollte am besten über die Verwendung des mitgelieferten Makefiles durchgeführt werden. Wechseln Sie dazu in der Eingabeaufforderung (Prompt) in Ihr deSIGN-Homeverzeichnis (wie bei der Installation) und geben dann dort folgende Zeile ein:

gmake run

deSIGN sollte nun starten. Haben Sie bitte jedoch etwas Geduld. Alle Java-Programme benötigen einen Interpreter und sind dadurch bedingt in ihrer Ausführungsgeschwindigkeit gebremst - das ist der Preis, den man für die Plattformunabhängigkeit zahlen muß.

A.1.3 Übersicht der Programmfunktionen

Bei der Arbeit mit dem Programm werden Sie grundsätzlich mit der folgenden IDE-Leiste beschäftigt sein (vgl. Abbildung A.1):

Hier sind in einem Rahmenfenster das Menü sowie die Toolbar angeordnet.

Bevor Sie mit den Funktionen des Programms (also der IDE-Leiste) vertraut werden, sollten Sie die Bedeutungen einiger einfacher Funktionen in der objektorientierten Modellierung kennen lernen.

A.1.3.1 Begriffserklärungen

Eine Klasse (vgl. Abbildung A.2) beschreibt eine Gruppe von Objekten mit ähnlichen Eigenschaften (Attributen), gemeinsamen Verhalten (Operationen),

Abbildung A.1: IDE-Leiste

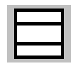


Abbildung A.2: Klasse

gemeinsamen Relationen zu anderen Objekten und einer gemeinsamen Semantik. Sie steht über Generalisierungen, Assoziationen und Aggregationen (hierzu später mehr) zu anderen Klassen in Beziehung.

Eine Klasse besitzt die Komponenten Attribute und Operationen.

Attribute sind Datenwerte (in Programmiersprachen häufig auch Datenelemente). Sie werden durch Angabe eines Namens und Typs spezifiziert.

Operationen sind Funktionen oder Transformationen. Eine Operation besitzt einen Namen und einen Typ für den Rückgabewert. Optional wird ihnen noch eine Anzahl von Parametern zugeordnet.

Assoziationen (auch Beziehungen genannt) sind Konzepte, um Relationen zwischen Klassen herzustellen. Assoziationen sind bidirektional. Sie werden in Programmiersprachen häufig als Zeiger (Pointer) von einem Objekt zum anderen implementiert. Assoziationen haben optional Namen, Rollennamen, Qualifizierer und Kardinalitäten.

Kardinalitäten (Multiplizitäten) spezifizieren, wie oft eine Klasse mit einer assoziierten Klasse in Beziehung stehen darf. Kardinalitäten stellen gewisse Einschränkungen für die Klassen dar. Sie können durch eine Zahl oder eine Intervallmenge beschrieben werden.

Bei der Teilnahme an Beziehungen zu anderen Klassen nimmt jede Klasse eine Rolle ein. Die Rolle kann explizit durch Angabe eines Namens verdeutlicht werden. Ein Rollenname ist ein Name, der ein Ende einer Assoziation eindeutig identifiziert.

Assoziationen können Qualifier zugeordnet werden, indem ein Attribut angegeben wird, das den Zugriff auf ein einzelnes Element oder auf eine Untermenge von Elementen der Assoziation ermöglicht. Die Qualfikationsangabe ist also ein besonderes Attribut, das die effektive Kardinalität einer Assoziation herabsetzt. Ein Qualifier verbessert die semantische Genauigkeit.

Bei Assoziationen unterscheidet man zwischen binären und n-ären Assoziationen.

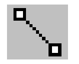


Abbildung A.3: Binäre Assoziation

Bei binären Assoziationen (vgl. Abbildung A.3) werden Relationen zwischen einer Klasse oder zwei Klassen hergestellt.

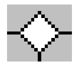


Abbildung A.4: N-äre Assoziation

Bei n-ären Assoziationen (vgl. Abbildung A.4) werden Relationen zwischen mehreren Klassen (mehr als zwei Klassen) hergestellt.

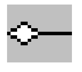


Abbildung A.5: Aggregation

Aggregationen (vgl. Abbildung A.5)bezeichnen die „Teil-Ganzes“ oder „ist-Teil-von“-Relation, an der genau zwei Rollen beteiligt sind: das Aggregat und die Komponente. Eine Aggregation ist eine eng gekoppelte Form der Assoziation mit einigen speziellen Bedingungen. Ihre wichtigsten Eigenschaften sind die Transitivität (d.h., wenn A Teil von B ist, und B ein Teil von C ist, dann ist A auch ein Teil von C) und die Antisymetrie (d.h., wenn A ein Teil von B ist, dann ist B nicht Teil von A).

Die Generalisierung (auch Vererbung genannt) ist eine mächtige Abstraktionsmöglichkeit, sowie auch ein wichtiges Konzept in objektorientierten Programmiersprachen. Sie dient dazu, Ähnlichkeiten zwischen verschiedenen Klassen zu teilen und gleichzeitig ihre Unterschiede zu erhalten. Die Generalisierung stellt die Relation zwischen einer Klasse und einer oder mehrerer verfeinerter Versionen davon dar. Es geht hier also um die Vererbung von Attributen, Methoden und Assoziationen, wobei die zu vererbende Klasse die Rolle einer Oberklasse (superclass) annimmt, und die vererbte Klasse als Unterklasse (subclass) betrachtet wird. Jede Unterklasse erbt die Merkmale ihrer Oberklasse. Das Vererbungskonzept wird in objektorientierten Programmiersprachen unterstützt. Im Falle, daß eine Unterklasse mehrere direkte Oberklassen besitzt und Merkmale von allen Eltern erbt, spricht man von Mehrfachvererbung.

A.1.3.2 Beschreibung der Funktionen

Nach dem Start des Programms erhalten Sie nun folgendes Bild (siehe Abbildung A.7):

Ganz oben befindet sich die Menüleiste. Direkt unter der Menüleiste finden Sie die Toolbar, welche alle verfügbaren Befehle für den schnelleren und einfacheren

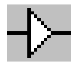


Abbildung A.6:

Abbildung A.7: Toolbar (einfach)

Zugriff als Icons (Bilder) zur Verfügung stellt. Die Beschreibungen der einzelnen Funktionen des Programms beruhen auf den Menüleisteneinträgen. Damit Sie jedoch auch die entsprechenden Einträge der Toolbar kennenlernen, sind diese in Abbildung A.8 mit Zahlen durchnummeriert. Dieses hat den Zweck, daß bei der Beschreibung der Menüeinträge auf die entsprechenden Toolbareinträge über diese Nummerierungen verwiesen wird (falls es für bestimmte Menüeinträge entsprechende Toolbareinträge gibt).

File Menü

In der Menüleiste haben Sie momentan nur die Möglichkeit das File Menü zu wählen (sieheAbbildung A.7). Dort finden Sie folgende Einträge:

New Diagram

Um ein neues Diagramm zu erzeugen, wählen Sie diesen Menüpunkt aus. Daraufhin öffnet sich ein neues Fenster. Dieses ist Ihre Zeichenfläche. Außerdem wird die IDE-Leiste erweitert. Zum Ausführen dieser Funktion über die Toolbar klicken Sie auf die Schaltfläche (Button) mit der Nummerierung „1“ (aus Abbildung A.8).

Open Diagram

Hiermit haben Sie zwei Möglichkeiten. Zum Laden eines zuvor mit der Serialisierungsmethode gespeicherten deSIGN Diagramms wählen Sie den Menüeintrag „Open Diagram / Open Serialized Diagram File...“. Bei der Serialisierungsmethode wird der Serialisierungsmechanismus von JAVA zum Abspeichern von Diagrammen genutzt. Zum Laden eines zuvor in Textform gespeicherten deSIGN Diagramms wählen Sie den Menüeintrag „Open Diagram / Open deSIGN Text Diagram File...“. Bei der Textformmethode wird eine ASCII-Datei erzeugt, die das Diagramm mittels Markups speichert. Beim Laden wird das Diagramm dann geparst und wieder in ein Diagramm umgewandelt. Zum Ausführen dieser Funktion über die Toolbar klicken Sie auf die Schaltfläche mit der Nummerierung „2“.

Quit

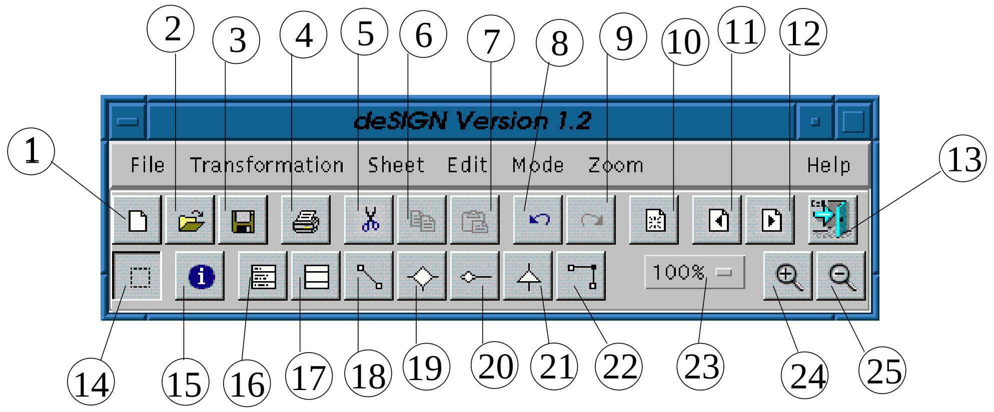


Abbildung A.8: Toolbar (erweitert)

Hiermit beenden Sie deSIGN. In der aktuellen Programmversion ist das Beenden des Programms aus Komfortgründen auch über die Toolbar möglich. Zum Ausführen dieser Funktion über die Toolbar klicken Sie auf die Schaltfläche mit der Nummerierung „13“.

Das Betätigen des Menüeintrages „New Diagram“ führt, wie bereits erwähnt, zur Erweiterung der IDE-Leiste (sieheAbbildung A.8). Neben den bereits beschriebenen Einträgen finden Sie in dem „File Menü“ zusätzlich folgendes:

Save

Für das Speichern von Diagrammen wählen Sie diesen Menüeintrag. Wenn man das Diagramm vorher noch nicht gespeichert hat, erscheint ein Filedialog, in dem man den Pfad (also Verzeichnis- und Dateinamen) des zu speichernden Diagramms angeben kann. Das Diagramm wird hierbei mit der Serialisierungsmethode gespeichert. Mit der Serialisierungsmethode gespeicherte deSIGN Diagramme erhalten grundsätzlich die Endung „.dsg“ (also Diagrammname.dsg). Für das Speichern über die Toolbar klicken Sie auf die Schaltfläche mit der Nummerierung „3“.

Save As

Hiermit hat man die Möglichkeit anzugeben, ob das Diagramm mit der Serialisierungsmethode oder in Textform gespeichert werden soll. Für die erste Methode wählen Sie den Menüeintrag „Save As / Save As Serialized Diagram File...“. Für das Speichern in Textform wählen Sie hingegen den Menüeintrag „Save As / Save As deSIGN Text Diagram File...“.

Ein weiterer Unterschied dieses Menüeintrags im Gegensatz zu der Funktion „Save“ liegt darin, daß man hierbei die Möglichkeit hat, bereits gespeicherte Diagramme unter einem anderen Namen und unter einem anderen Pfad abzuspeichern. Analog zu der „Save“-Funktion erscheint auch hier ein Filedialog zum Angeben des Pfads (Verzeichnis- und Dateiname). Beim Speichern mit der Serialisierungsmethode erhalten die Diagramme, wie bereits beschrieben, die Endung .dsg. Beim Speichern in Textform lautet die Endung hingegen „.dst“ (also Diagrammname.dst).

Print

Für das Drucken von Diagrammen wählen Sie den Menüeintrag („Print"). Daraufhin erscheint ein Setup-Dialog, in dem Sie vor dem Drucken gewisse Einstellungen vornehmen können.

Standardmäßig druckt das Programm Diagramme im Postscript-Format (PS). Es ist aber auch möglich, Diagramme im Encapsulated Postscript-Format (EPS) zu drucken. Hierfür reicht es aus, die Schaltfläche neben der Beschriftung „EPS" im Dialog zu aktivieren. Des weiteren ist es zusätzlich möglich, neben der Ausgabe auf dem Drucker, den Druck in eine Datei umzulenken. Hierfür ist es erforderlich, den Menüeintrag „File" aus der Auswahlbox neben der Beschriftung „Print to" auszuwählen. Daraufhin ändert sich der Dialog, und es ist nun möglich, in das Textfeld unter der Beschriftung „Filename" den Namen der Datei anzugeben, unter dem das Diagramm gespeichert werden soll (oder mittels der Schaltfläche „Browse" durch Verzeichnisse zu navigieren).

Außerdem können Sie bereits aus gängiger Software bekannte Druckereinstellungen, wie das Drucken von gesamten Diagrammen oder nur bestimmten Seiten hiervon, Angabe von Druckerkommando, Druckernamen und Druckparametern, Seiteneigenschaften (Blattausrichtung, Blattgröße), Druckqualität etc. angeben. Für den entsprechenden Menüeintrag aus der Toolbar klicken Sie auf die Schaltfläche mit der Beschriftung „4".

Transformation Menü

Transform to SQL

Hiermit können Sie für gezeichnete oder geladene Modelle (Diagramme) relationale Datenbankschemata mit SQL generieren. Die erzeugten Datenbankschemata werden dann in einer Datei (unter dem gleichen Namen, wie das zu transformierende Diagramm) im aktuellen Verzeichnis abgelegt und können mit einem Texteditor geladen und eingesehen werden. Die Datei wird mit der Endung „.sql" gespeichert (also Diagrammname.sql).

Transform to JAVA

Hiermit können Sie für gezeichnete oder geladene Modelle (Diagramme) JAVA Klassengerüste erzeugen. Auch hier werden die erzeugten JAVA-Codes in Dateien abgespeichert, wobei hierfür im aktuellen Verzeichnis ein Unterverzeichnis mit dem Namen des Diagramms angelegt wird. Darin werden dann die einzelnen (erzeugten) JAVA-Dateien gespeichert. Diese können auch hier mit einem Texteditor geladen und eingesehen werden.

Load Translation Table

Hier können Sie eine Übersetzungstabelle laden. Sorgen Sie dafür, daß es sich um eine gültige Tabelle handelt, ansonsten interpretiert deSIGN diese eventuell falsch und es kann so zu Fehlern führen. Standardmäßig liegt deSIGN eine Übersetzungstabelle bei (TranslationTable1.tt), welche Sie natürlich editieren können. Achten Sie dabei darauf, daß Sie das vorgegebene Format beibehalten: An der ersten Stelle ist das Element der SIGN-ML zu finden, an den weiteren die Einträge für die Zielsprachen. Alle Einträge sind dabei durch Leerzeichen getrennt.

Show Translation Table

Wenn Sie eine Übersetzungstabelle geladen haben (standardmäßig ist TranslationTable1.tt geladen), können Sie sich diese auch angucken. Wählen Sie hierzu diesen Menüpunkt. In der obersten Zeile sollten Sie den Eintrag SIGNML finden. Außerdem stehen hier sämtliche Zielsprachen (z.B. Java oder SQL). In den folgenden Zeilen stehen Quell-Ausdruck und entsprechende Zielausdrücke (z.B. Number Integer int). Sind Sie mit den Ergebnissen nicht einverstanden, so können Sie die Datei mit jedem beliebigen Texteditor (z.B. emacs, Notepad, ...) ändern.

Sheet Menü

Das Bearbeiten von komplexeren und größeren Diagrammen erfordert die Möglichkeit, Diagramme in kleinere Ausschnitte zu unterteilen, da diese sonst evtl. nicht auf ein DIN A4 Blatt passen (insbesondere wichtig beim Drucken). Hierfür gibt es in deSIGN das Konzept von Sheets. Ein Sheet ist ein Arbeitsblatt (ein DIN A4 Blatt). Demnach können Diagramme in Seiten (also Sheets) unterteilt werden. Das Konzept der Sheets dient zudem dazu, die Übersichtlichkeit von Diagrammen zu erhöhen.

New Sheet

Wenn Sie im aktuellen Diagramm eine neue Seite (also ein Sheet) erzeugen möchten, so wählen Sie diesen Menüeintrag. Das Ausführen dieser Funktion über die Toolbar erfolgt durch betätigen der Schaltfläche mit der Nummerierung „10“.

Next Sheet / Previous Sheet

Das Umblättern zwischen mehreren erzeugten Sheets erfolgt über diese beiden Einträge. Dabei können Sie mittels „Next Sheet“ zum nächsten Sheet und mittels „Previous Sheet“ zum vorherigen Sheet blättern. Das Ausführen der Funktion „Previous Sheet“ über die Toolbar erfolgt durch betätigen der Schaltfläche mit der Nummerierung „11“. Für die Funktion „Next Sheet“ lautet die entsprechende Schaltfläche Nr. „12“.

First Sheet / Last Sheet

Für das komfortable Navigieren durch die gesamte erzeugte Sheetliste gibt es diese beiden Einträge. Hierbei können Sie mittels „First Sheet“ zum ersten erzeugten Sheet und mittels „Last Sheet“ zum letzten erzeugten Sheet blättern.

Edit Menü

Im Edit Menü werden Funktionen zum Bearbeiten von Diagrammen bereitgestellt. Das Programm deSIGN bietet hierfür die folgenden Funktionen:

Delete

Genauso, wie es möglich ist, Objekte im Diagramm zu zeichnen, so muß es natürlich auch die Möglichkeit geben, diese aus dem Diagramm wieder löschen zu können. Dazu dient dieser Menüeintrag. Um ein Objekt aus dem Diagramm (für immer) zu löschen, muß es erst einmal selektiert werden (auf die Funktionalität der Selektion wird später detailliert eingegangen. Hier soll es lediglich aus Gründen der Vollständigkeit erwähnt werden). Nachdem das graphische Objekt selektiert wurde, wählen Sie nun diesen Menüeintrag aus. Das selektierte Objekt wird dann aus dem Diagramm entfernt (zusammen mit evtl. diesem Objekt in Verbindung stehenden anderen Objekten). Für das Ansteuern dieses Kommandos über die Toolbar betätigen Sie die Schaltfläche mit der Nr. „5“.

Properties

Für das bereits beschriebene Sheet-Konzept (siehe Beschreibung des „Sheet Menüs“) kann man über diesen Menüeintrag einige Einstellungen vornehmen. Nachdem Sie diesen Menüeintrag ausgewählt haben, öffnet sich daraufhin ein Eigenschaftsdialog, in dem Sie für das aktuelle Sheet auswählen können, in welchem DIN-Format (oder einer benutzerdefinierten Größe) es ausgedruckt werden soll.

Mode Menü

Bei der Arbeit mit dem Programm deSIGN werden Sie sicherlich am häufigsten dieses Menü benutzen müssen. Hier werden alle graphischen Objekte (Diagrammelemente) zur Verfügung gestellt, die Sie für das Zeichnen von Diagrammen benötigen. Das „Mode Menü“ (der Name „Mode“ ist übrigens aus dem Begriff „ Edit Modes“ abgeleitet, was soviel wie „Bearbeitungsmodi“ bedeutet und enhält mehrere „Bearbeitungsfunktionen“) ist als Auswahlbox dargestellt. Dieses hat den Effekt, daß zur gleichen Zeit immer nur eine Bearbeitungsfunktion aktiv sein kann, so daß die Auswahl eines neuen Menüeintrages den letzten gewählten Eintrag deaktiviert (analog dazu ist auch die Toolbar für die Bearbeitungsmodi aufgebaut). Im folgendem werden die einzelnen Bearbeitungsmodi des „Mode Menüs“ vorgestellt:

Selection

Dieser Bearbeitungsmodus ist voreingestellt, wenn man ein neues Diagramm öffnet. Er dient zum Selektieren von (gezeichneten oder geladenen) Diagrammelementen im Diagramm. Somit können Sie auf einzelne, selektierte Diagrammelemente die bereits vorgestellten Kommandos des „Edit Menüs“ (also Cut, Copy, Paste u.ä) anwenden. Des weiteren hat es auch den Sinn, daß für einzelne, selektierte Diagrammelemente entsprechende Eigenschaftsdialoge geöffnet und bearbeitet werden können. Um aus dem Diagramm ein Diagrammelement zu selektieren, aktivieren Sie diesen Bearbeitungsmodus („Selection“) und klicken dann mit der Maus auf das gewünschte Diagrammelement. Für das Selektieren von mehreren Diagrammelementen aktivieren Sie ebenfalls diesen Bearbeitunsmodus, halten dann die Shift-Taste der Tastatur gedrückt und klicken auf die zu selektierenden Diagrammelemente.

Ein alternativer Ansatz für das Selektieren von einem oder mehreren Diagrammelementen ist das Drücken der Maus an einer freien Stelle im Diagramm (d.h. eine Stelle, wo kein Diagrammelement positioniert ist) und anschließendes Ziehen der Maus, bei immer noch gedrückter Taste über die zu selektierenden Diagrammelemente. Es erscheint ein viereckiger Rahmen, der alle sich darin befindlichen Diagrammelemente automatisch selektiert. In der Toolbar aktivieren Sie hierfür die Schaltfläche mit der Nr.„14“.

Insert Class

Das Zeichnen von Diagrammen beginnt mit dem Anlegen einer Klasse. Aktivieren Sie hierfür diesen Eintrag im Menü „Mode“. Daraufhin klicken Sie mit der linken Maustaste an eine gewünschte Stelle im Diagramm. Sie haben soeben Ihre erste Klasse im Diagramm angelegt. Die Klasse erhält standardmäßig einen Klassennamen zugewiesen (beginnend mit „Class1“). Durch das Aktivieren des bereits beschriebenen Selektionsmechanismus (Bearbeitungsmodus „Selection“) können Sie die Klasse nun selektieren, wobei selektierte Objekte im Diagramm rot markiert werden.

Nachdem Sie Ihre Klasse selektiert haben, bekommen Sie durch das Betätigen der Schaltfläche mit der Nr. „15“ aus der Toolbar zu der selektierten Klasse einen Eigenschaftsdialog. Hier können Sie den Klassennamen verändern, sowie der Klasse Eigenschaften (Attribute) und Verhalten (Operationen) verleihen. Wenn Sie beispielsweise Attribute angeben möchten, so brauchen Sie diese nur der Reihe nach in das entsprechende Textfeld einzugeben und ihre Eingaben jeweils mit dem Betätigen der Schaltfläche „Add“ im Dialog abzuschließen. Die angegebenen Attribute erscheinen dann in dem benachbarten Textfeld. Durch einen Klick auf ein gewünschtes Attribut und dem anschließenden Betätigen eine der Schaltflächen „Delete“ oder „Modify“ können Sie dieses Attribut löschen bzw. ändern. Analog hierzu verfahren Sie dann auch für die Operationen der Klasse.

Durch das Aktivieren oder Deaktivieren der Schaltflächen „Show Attributes“ und „Show Operations“ können Sie die Attribute und Operationen der Klasse dann jederzeit ein- bzw. ausblenden. Bestätigen Sie Ihre vorgenommenen Einstellungen mittels des „OK“-Buttons, so erscheinen diese in der Klasse in dem Diagramm. Bei aktiviertem Menüeintrag „Insert Class“ können Sie dann im Diagramm soviele neue Klassen erzeugen und nach dem nun bekanntem Verfahren selektieren und bearbeiten, wie Sie möchten. Sie können die Klassen im Diagramm jederzeit auch bewegen (verschieben). Hierfür klicken Sie mit der Maus auf die zu bewegende Klasse und ziehen diese bei gedrückter Maustaste einfach an eine andere Stelle. Die entsprechende Funktionalität (dieses Bearbeitungsmodus) erreichen Sie in der Toolbar durch das Aktivieren der Schaltfläche mit der Nr. „17“.

Durch das Aktivieren der Schaltfläche mit der Nr. „16“ in der Toolbar können Sie eine bereits existierende Klasse im Diagramm „klonen“. Somit wird die gleiche Klasse mit dem gleichen Namen erneut dargestellt. Dieses macht insofern Sinn, als daß die Übersichtlichkeit des Diagramms erhalten bleibt. Nach dem Aktivieren dieser Schaltfläche, erhalten Sie einen Dialog, in dem Sie aus einer Auswahlbox, die von Ihnen zu „klonende“ Klasse auswählen können. Die „geklonte“ Klasse erscheint somit in dem Diagramm. Im allgemeinen werden „Klonklassen“ standardmäßig im Diagramm immer an der selben Position (linke obere Ecke) plaziert.

Wie Sie sicherlich auch bemerkt haben, erscheint die Klasse nur mit einem Namen, d.h. ohne jegliche Attribute und Operationen. Dieses ist beabsichtigt und kann auch später nicht verändert werden. Das Bearbeiten dieser „geklonten“

Klasse bzgl. Attribute und Operationen läßt diese vorgenommenen Einstellungen nur in der Originalklasse erscheinen.

Insert Association

Um Beziehungen (auch Relationen genannt), also Assoziationen zwischen Klassen darzustellen, wählen Sie diesen Bearbeitungsmodus aus, wobei binäre Assoziationen, also Beziehungen zwischen zwei Klassen gemeint sind, wenn von Assoziationen die Rede ist. Diese Unterscheidung ist wichtig, da es auch n-äre Assoziationen gibt, also Beziehungen zwischen mehr als zwei Klassen, wie später noch gezeigt wird.

Nachdem Sie also zwei Klassen auf dem Diagramm gezeichnet haben, wählen Sie nun diesen Bearbeitungsmodus aus. Dann klicken Sie nacheinander die beiden Klassen an, wobei die Reihenfolge, in der die Klassen angeklickt werden, nicht entscheidend ist. Sie haben nun eine einfache Assoziation zwischen zwei Klassen erzeugt. Wie Sie sehen, wird hierbei eine Linie zwischen den Klassen gezeichnet.

Vielleicht haben Sie bemerkt, daß die Assoziation von der linken oberen Ecke der einen Klasse zur linken oberen Ecke der anderen Klasse gezeichnet wird. Dieses hat seinen Grund. Jede Klasse besitzt sogenannte Ankerpunkte (acht Stück), um Verbindungen anzuhängen. Durch einen Klick auf die Klasse selbst wird die Verbindung standardmäßig an den Ankerpunkt in der linken oberen Ecke der Klasse angehängt. Sie können aber natürlich auch einen bestimmten Ankerpunkt anwählen. Hierfür gehen Sie mit der Maus einfach über die Kanten der Klasse (die Ankerpunkte werden dann sichtbar, wenn Sie mit der Maus in Ihrer Nähe sind) und klicken dann auf einen gewünschten Ankerpunkt. Durch das Konzept der Ankerpunkte werden gewisse Überschneidungen von Assoziationslinien verhindert, so daß man die optimalste (kürzeste) Route zwischen den zu verbindenden Klassen anwählen kann (dieses fördert auch die Übersichtlichkeit des Diagramms).

Das Prinzip der Ankerpunkte gilt auch für n-äre Assoziationen, Aggregationen und Generalisierungen (diese werden später beschrieben) in analoger Weise und wird daher später nicht noch mal in dieser Ausführlichkeit beschrieben, sondern lediglich erwähnt bzw. darauf verwiesen.

Des weiteren bietet das Programm die Möglichkeit, Beziehungen umzuhängen. Im Klartext bedeutet das, daß Sie durch einfaches Klicken auf den Ankerpunkt, an dem der Anfang bzw. das Ende der Assoziation hängt und anschließendes Ziehen der Assoziation, bei noch gedrückter Maustaste (Drag and Drop) an eine andere Klasse bzw. an andere Ankerpunkte, diese verschieben (also umhängen) können. Dieses Konzept ist gleichermaßen auf alle anderen Beziehungsarten (N-äre Assoziationen, Aggregationen und Generalisierungen) anwendbar.

Nachdem Sie eine Assoziation zwischen zwei Klassen gezogen haben, können Sie diese nun bearbeiten. Nach dem bereits bekannten Muster wählen Sie hierfür den Bearbeitungsmodus für Selektionen (oder die entsprechende Schaltfläche in der Toolbar) aus und klicken dann analog zu der Selektion von Klassen hier nun auf die Assoziationslinie. Danach wählen Sie die Schaltfläche (Nr. „15“) für den Eigenschaftsdialog aus. Daraufhin öffnet sich der Eigenschaftsdialog. Hier können Sie sowohl allgemeine, als auch speziell die an der Assoziation beteiligten Klassen betreffende Einstellungen vornehmen.

In dem Textfeld, das der Beschriftung „Name of the Association“ zugehörig ist, kann für die Assoziation ein Name vergeben werden. Für die zwei Klassen, die an der Assoziation beteiligt sind, gibt es (jeweils dem Klassennamen zugeordnete) Felder, in denen man Rollennamen, Multiplizitäten (auch Kardinalitäten genannt) und Qualifikationen angeben kann (machen Sie sich keine Sorgen, wenn Sie mit diesen Begriffen nichts anfangen können. Diese werden später noch ausführlich erläutert. Außerdem können Sie zum Kapitel „Begriffserklärungen“ zurückblättern, um die allgemeine Bedeutungen dieser Begriffe nachzulesen).

Die Multiplizitäten sind standardmäßg auf 0...* gesetzt (also keine Einschränkungen).

Wie Sie an den Schaltflächen für „Show Associationname“ und „Show Roles/Multiplicites“ in dem Dialog sehen können, ist der Assoziationsname standardmäßig ausgeblendet, wobei die Rollennamen, sowie die Multiplizitäten eingeblendet werden. Aber auch diese Einstellungen können ohne weiteres verändert werden. Für das erfolgreiche Abschließen der vorgenommenen Einstellungen klicken Sie auf den „OK“-Button.

Für das Ansteuern dieses Bearbeitungsmodus über die Toolbar aktivieren Sie die Schaltfläche mit der Nr. „18“.

Insert n-ary Association

Wie bereits erwähnt, gibt es neben der Möglichkeit binäre Assoziationen darzustellen, auch noch jene, um n-äre Assoziationen zu modellieren. Um mehr als zwei Klassen miteinander zu verbinden, wählen Sie aus dem Menü „Mode“ diesen Bearbeitungsmodus aus.

Nachdem Sie dieses geschafft haben und mindestens drei Klassen, die miteinander assoziieren sollen, in Ihrem Diagramm zur Verfügung haben, können Sie auf zweierlei Art diesen Bearbeitungsmodus nutzen. In der ersten und Ihnen bekannten Methode klicken Sie einfach an eine Stelle im Diagramm. Daraufhin erscheint ein Diamantsymbol (Rautensymbol). Dieses Diamantsymbol charakterisiert eine n-äre Assoziation.

Damit Sie sich nun nicht die Frage stellen, was Ihnen an dieser Methode bekannt sein soll, können Sie das Diamantsymbol mit einer Klasse gleichsetzen (natürlich nicht semantisch). Bei den binären Assoziationen haben Sie gelernt, wie man Beziehungen zwischen zwei Klassen darstellt. Hier ist es nun ähnlich. Mit dem Unterschied, daß Sie es mit einer Klasse und einem Diamantsymbol zu tun haben, können Sie aber dadurch, daß sich das Diamantsymbol wie eine Klasse verhält, auf bekannte Art und Weise Beziehungen zwischen der Klasse und dem Diamantsymbol erzeugen.

Hierfür aktivieren Sie nun den Bearbeitungsmodus für binäre Assoziationen, also „Insert Association“ (oder den entsprechenden Button aus der Toolbar). Dann klicken Sie nach bekanntem Muster mit der Maus der Reihe nach auf die Klasse und das Diamantsymbol (oder umgekehrt). Sie können natürlich auch einzelne Ankerpunkte anklicken (aber das wissen Sie ja schon). Übrigens hat das Diamantsymbol auch Ankerpunkte, nämlich vier Stück. Nachdem Sie das geschafft haben, sehen Sie, daß eine Ihrer Klassen mit dem Diamanten in Verbindung steht. Dieses ist aber nicht Sinn der Sache und würde auch keine logische Bedeutung haben.

Wenn Sie sich aber nun eine weitere Klasse nehmen und den gleichen Vorgang mit dem selben Diamantsymbol erneut durchspielen, erhalten Sie nun zwei Klassen, die über den Diamanten in Beziehung stehen. Hört sich schon nicht schlecht an. Wir hatten doch aber gesagt, daß die n-ären Assoziationen zur Modellierung von Beziehungen zwischen mehr als zwei Klassen dienen. Also spielen Sie den Vorgang ein weiteres mal durch, vielleicht sogar solange, bis Sie nicht mehr können.

Wenn Sie es dann doch noch dazu gebracht haben, mindestens drei Klassen über das Diamantsymbol zu verbinden, dann haben Sie endlich eine n-äre Assoziation modelliert. Eigentlich nicht, bedarf lediglich ein Paar Übungsvorgängen. Aber nichts desto trotz gibt es eine alternative und einfachere (zumindest schnellere) Methode, n-äre Assoziationen darzustellen. Hierfür müssen wir wieder zurück zu unserer Basissituation - Sie sollten mindestens drei Klassen in Ihrem Diagramm besitzen. Danach aktivieren Sie erneut den Bearbeitungsmodus für n-äre Assoziationen, also „Insert n-ary Association“. Anschließend halten Sie die Shift-Taste der Tastatur gedrückt und klicken nacheinander die Klassen (oder bestimmte Ankerpunkte derer), um sie zu selektieren. Dieses können Sie alternativ auch durch klicken der Maustaste (an irgendeiner Stelle im Diagramm) und anschließendes Ziehen dieser bei gedrückter Maustaste über die betroffenen Klassen erreichen.

Wenn Ihre Klassen selektiert sind, erhalten Sie jeweils eine rote Umrandung. Nun brauchen Sie nur noch an eine von Ihnen gewünschte Stelle im Diagramm zu klicken und die n-äre Assoziation ist fertig (es muß natürlich eine leere Stelle sein, also eine, an der noch kein Objekt plaziert ist).

Bei dieser Methode können Sie die Ankerpunkte des Diamanten, an denen die Klassen der n-ären Assoziation hängen sollen, nicht auswählen. Daher werden automatisch die richtigen Ankerpunkte des Diamanten ausgewählt. Richtig bedeutet hier, die kürzeste Route zwischen dem Diamanten und der zu verbindenden Klasse. (Wenn also beispielsweise eine Klasse links von dem Diamanten plaziert ist, so wird der linke Ankerpunkt des Diamanten ausgewählt, um eine Verbindung herzustellen).

Ein großer Nachteil dieser Methode liegt darin, daß nach dem Darstellen einer n-ären Assoziation diese nicht mehr verändert bzw. erweitert werden kann. Beispielsweise kann aus einer dreistelligen Assoziation später keine vierstellige oder andere Assoziationen gemacht werden.

Es ist dann wieder die vorher (als erste) vorgestellte Methode nötig. Auch bei n-ären Assoziationen haben Sie die Möglichkeit gewisse Einstellungen vorzunehmen. Hierzu verfahren Sie wieder nach dem teilweise bekanntem Muster. Als erstes aktivieren Sie den Selektionsmechanismus. Anschließend klicken Sie dann aber auf das Diamantsymbol. Mit einem Klick auf die Schaltfläche für den Eigenschaftsdialog öffnet sich dieser. Hierbei können Sie analog zu dem Eigenschaftsdialog für binäre Assoziationen den Namen der Assoziation, Rollennamen, Multiplizitäten und Qualifikationen, für die an der Assoziation beteiligten Klassen angeben, und diese Eigenschaften ein- und ausblenden.

Um diese Eigenschaften für eine bestimmte Klasse zu setzen, wählen Sie diese Klasse aus der Auswahlbox aus, und geben Sie dann für die genannten Eigenschaften Ihre Einstellungen in die entsprechenden Textboxen ein. Ihre Einstellungen für diese Klasse schließen Sie dann durch Betätigen des „Add“-Buttons ab. Nun können Sie weitere Klassen auswählen und analog verfahren. Mittels „Add“ werden die Einstellungen für die Klasssen gespeichert. Das anschließende Bestätigen der Einstellungen mittels des „OK“-Buttons stellt diese dann in dem Diagramm dar.

Das Ansteuern dieses Bearbeitungsmodus über die Toolbar erfolgt per Aktivierung der Schaltfläche mit der Nr. „19“.

Insert Aggregation

Zum Darstellen von Aggregationsbeziehungen aktivieren Sie diesen Bearbeitungsmodus. Ähnlich, wie bei der Modellierung von binären Assoziationen, können Sie hiermit eine Beziehung zwischen zwei Klassen darstellen. Hierfür klicken Sie wieder nacheinander die beiden Klassen (oder wie bereits bekannt, gezielte Ankerpunkte der Klassen) im Diagramm an, zwischen denen eine Aggregation dargestellt werden soll. Allerdings ist hierbei die Reihenfolge der angewählten Klassen wichtig. Denn die Klasse, die als erste angewählt wird, ist auch zugleich die Aggregierende. Die andere Klasse ist demnach die Komponenten- bzw. Teilbeziehung des Aggregats.

Für diese Art von Relationen können über den Eigenschaftsdialog bestimmte Einstellungen vorgenommen werden. Hierzu verfahren Sie analog zu der vorgehensweise für binären Assoziationen. Erst aktivieren Sie den Selektionsmechanismus, dann klicken Sie auf die Aggregationslinie. Diese ist daraufhin selektiert. Anschließend betätigen Sie die Schaltfläche für den Eigenschaftsdialog (dieser sollte mittlerweile bekannt sein). Daraufhin öffnet sich der Eigenschaftsdialog. Analog zu dem Eigenschaftsdialog für binäre Assoziationen können Sie der Aggregation einen Namen geben, sowie für die an der Aggregation beteiligten Klassen Rollennamen, sowie Multiplizitäten vergeben und alle Eigenschaften ein- und ausblenden.

Die Ansteuerung des Aggregationsmodus erreichen Sie über die Toolbar, indem Sie die Schaltfläche mit der Nr. „20“ aktivieren.

Insert Generalisation

Für die Modellierung von Vererbunsbeziehungen aktivieren Sie diesen Bearbeitungsmodus. Ähnlich dem Bearbeitungsmodus für Aggregationen verfahren Sie auch hier. Klicken Sie demnach auch hier der Reihe nach auf die zu modellierenden Klassen im Diagramm (oder auf ihre Ankerpunkte), wobei auch hier die Reihenfolge des Auswählens der Klassen wichtig ist. Die zuerst angewählte Klasse wird hier nämlich zur Oberklasse und die andere Klasse zu ihrer Unterklasse (subklasse bzw. abgeleitete Klasse).

Das ist wichtig insofern, als daß die Unterklasse die Eigenschaften der Oberklasse erbt. Vererbungsbeziehungen besitzen in der objektorientierten Modellierung grundsätzlich keinen Namen, oder zusätzliche Attribute, wie Rollennamen, Multiplizitäten o.ä. Daher existiert hierfür auch kein Eigenschaftsdialog.

Um diesen Bearbeitungsmodus über die Toolbar anzusteuern, aktivieren Sie die Schaltfläche mit der Nr. „21“.

Connector

Über diesen Bearbeitungsmodus können Sie zwischen Beziehungen jeglicher Art Kontrollpunkte setzen (dazu gehören binäre Assoziationen, n-äre Assoziationen, Aggregationen und Generalisierungen). Diese Kontrollpunkte sollen die Übersichtlichkeit im Diagramm fördern, da somit an den starren Verbindungslinien an jeder gewünschten Stelle „Knicke“ eingefügt werden und diese dann bewegt werden können.

Um diese Funktionalität verwenden zu können, ist es erforderlich, daß im Diagramm bereits irgendeine Art von Beziehung (siehe o.g) dargestellt ist. Wenn dies der Fall ist und Sie diesen Bearbeitungsmodus aktiviert haben, können Sie nun an irgendeiner Stelle an der Verbindungslinie der Beziehung mit der Maus klicken. Daraufhin erhalten Sie einen „Knick“ an der Verbindungslinie. Dieser charakterisiert einen Kontrollpunkt und ist nun im Diagramm beliebig verschiebbar. Wie bereits erwähnt, können Sie diese Funktionalität überall und so oft sie wollen, an der Verbindungslinie durchführen.

Für das Ansteuern dieser Funktion über die Toolbar aktivieren Sie die Schaltfläche mit der Nr. „22“.

Zoom Menü

Für das Vergrößern und Verkleinern von Diagrammen gibt es die Zoomfunktion. Durch Auswählen eines Wertes zwischen 25%-200% können Sie Diagramme vergrößern bzw. verkleinern. Standardmäßig ist der Wert auf 100% gesetzt. Zum vergrößern dienen die Werte größer 100% (also 150% und 200%). Zum verkleinern wählen Sie hingegen die Werte kleiner 100% (also 75%, 50% und 25%), wobei gilt: je kleiner der Wert, desto kleiner auch das gezoomte Diagramm.

Diese Funktionalität können Sie, wie auch fast alle beschriebenen Funktionalitäten des Programms über die Toolbar bedienen. Hierfür werden zwei Möglichkeiten angeboten. Die eine ist die Auswahlbox (Nr. „23“), aus der Sie einfach einen Wert (wie o. g.) auswählen. Die alternative Möglichkeit bieten die Schaltflächen mit den Nummern „24“ (+ Button) und „25“ (- Button) an, wobei der „+ Button“ zum vergrößern dient und der „- Button“ zum verkleinern. Mit jedem Klick auf einen dieser Buttons wird auf den nächsthöheren bzw. niedrigeren Wert gezoomt.

Help Menü

About

Durch das Anwählen dieses Menüeintrages wird ein Informations-Dialog geöffnet, in dem Sie Informationen über das Programmierteam des Programmes deSIGN (Copyright), Angaben zur Programmversion u.ä erhalten.

Online Help

Durch das Anwählen dieses Menüeintrages (oder per Tastatursteuerung über „Control (Ctrl) + H“) wird auf eine Online Hilfe verwiesen. Die Online Hilfe ist zur Zeit dieses Benutzerhandbuch. Das Benutzerhandbuch des Programmes deSIGN wird also über diesen Eintrag, als WWW-Dokument in einen Browser geladen.

A.1.4 Beispiel

In diesem Kapitel wollen wir uns einmal ein komplexes Beispiel vornehmen (sieheAbbildung A.9), damit Sie sehen, wie einfach man mit deSIGN die gewünschten Diagramme zurechtbasteln kann. Wenn Sie wie wir Programmierer schon mal in den Genuß eines echten Mensa-Essens gekommen sind, so werden Sie sicherlich nachvollziehen können, warum wir gerade dieses Beispiel gewählt haben.

Wichtig bei der Auswahl eines Beispiels war uns, daß sich aus dem Diagramm sowohl (halbwegs) sinnvoller Java-Code erstellen läßt, als auch eine genauso sinnvolle SQL-Datenbank. Hoffentlich haben Sie bei der Erstellung der Graphik genausoviel Spaß wie wir - und keine Angst - Sie brauchen das Erzeugte nicht essen!

Um Ihnen zuerst einen Überblick zu verschaffen, werfen Sie doch mal einen Blick auf ihr Ziel:

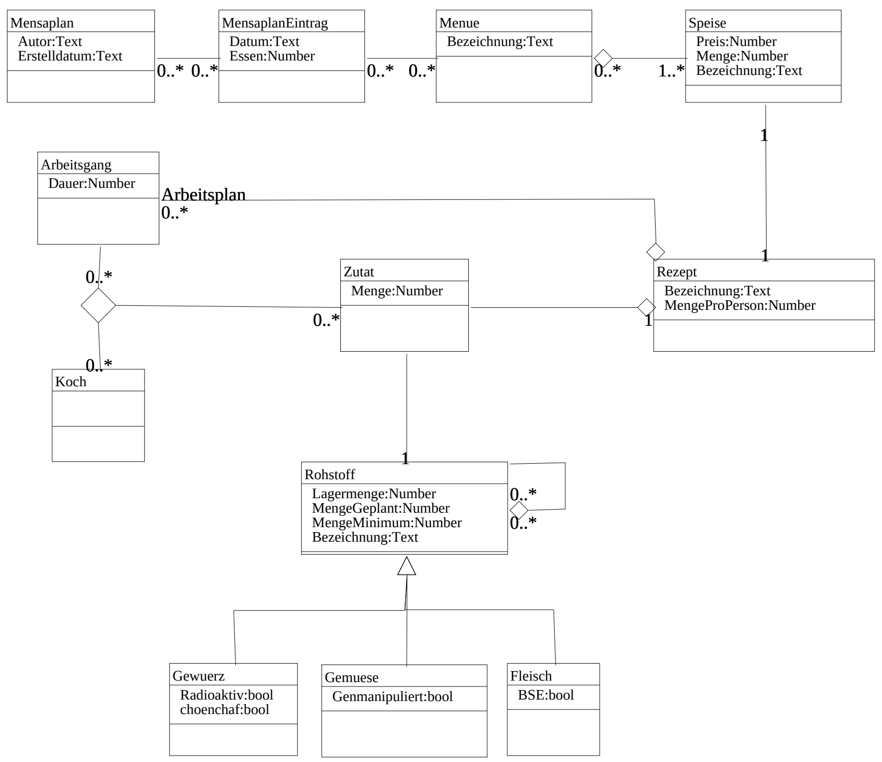


Abbildung A.9: deSIGN Beispiel Diagramm

Keine Bange, wir werden uns langsam an das Ziel herantasten.

Als erstes sollten Sie alle benötigten Klassen erzeugen. Dazu wechseln Sie in den Klassen-Modus, indem Sie den Button „17“ drücken oder aus dem Modus-Menü „Klassenmodus“ auswählen. Nun erzeugen Sie 12 Klassen: Klicken Sie 12-mal auf dem Zeichenblatt die Maustaste. Bei jedem Mausklick wird eine

Klasse erzeugt. Leider habe diese noch keinen sinnvollen Namen, Operationen und Attribute. Um diese einzutragen, verwenden Sie den Eigenschaftsdialog für Klassen. Drücken Sie hierzu den Button „14“ oder wählen Sie im Menü „Mode“ den Eintrag „Selection“. Sie befinden sich jetzt im Selektionsmodus.

Sie markieren einfach die Klasse, die Sie editieren möchten und klicken nun auf Button „15“ (bzw. „Properties“ aus dem „Edit“-Menü wählen). Damit öffnet sich der Eigenschaftdialog des gewählten Diagrammelements (in unserem Fall der Klassendialog). Hier können Sie alle Änderungen vornehmen, die Sie an dieser Klasse für nötig befinden. Nach und nach sollten Sie alle Klassen abarbeiten - immer eine nach der anderen. Tragen Sie hierzu die unten aufgeführten Werte in die jeweils dafür vorgesehenen Bereiche ein:

Klassenname	Attribute	Operationen
Arbeitsgang	Dauer: int	
Koch		
Zutat	Menge: int	
Rohstoff	Lagermengen_Bez: String	
	Menge_geplant: int	
	Menge_min: int	
Gewuerz	Radioaktiv: bool	
	Choenchaf: bool	
Gemuese	Gen_manipuliert: bool	
Fleisch	BSE: bool	
Rezept	Menge_pro_person: int	
	Bezeichnung: String	
Speise	Preis: int	
	Menge: int	
	Bezeichnung: String	
Menue	Bezeichnung: String	
Mensaplan	Autor: String	
	Erstelldatum: int	
MensaPlanEintrag	Datum: Text	
	Essen: Number	

Wenn Sie dies geschafft haben, haben Sie das gröbste hinter sich. Arangieren Sie die Klassen so, wie Sie sie auf dem Zeichenblatt verteilt haben möchten. Danach versuchen wir uns an den Beziehungen zwischen den Klassen.

Im ersten Schritt wollen wir uns die binären Assoziationen vornehmen: Wechseln Sie jetzt in den Assoziationsmodus, indem Sie Button „18“ drücken oder „Insert Association“ aus dem „Mode“-Menü wählen. Nun verbinden Sie die Klassen: Klicken Sie zuerst in die Klasse Mensaplan und danach in die Klasse MensaplanEintrag. Daraufhin erscheint die Assoziation zwischen den beiden Klassen, wobei diese an den Ankerpunkten links oben an den Klassen verankert ist. Verschieben Sie die Enden der Assoziationen (wechseln Sie in den Selektionsmodus) an die Ankerpunkte, die ihnen am besten passen. Diese Aktionen führen Sie nun für die Assoziationen zwischen MensaplanEintrag und Menü, Speise und Rezept, und Zutat und Rohstoff aus.

Sehen Sie, Sie haben es fast geschafft. Für die Assoziationen fehlen jetzt nur noch die Kardinalitäten, Rollen und Qualifier. Dazu wechseln Sie bitte in den

Selektionsmodus und rufen den Assoziationsdialog auf. Nun tragen Sie für die Assoziationen die Kardinalitäten ein, die Sie im Beispiel wiederfinden: Zwischen Speise und Produkt jeweils 1, usw.

Kommen wir zu der N-ären Assoziation zwischen Koch, Arbeitsgang und Zutat. Klicken Sie, um in den entsprechenden Bearbeitungsmodus zu gelangen, auf Button „19“ oder im Menü „Mode“ auf „Insert n-ary Association“. Jetzt klicken Sie nacheinander auf die oben genannten Klassen und anschließend auf eine freie Stelle im Diagramm. Dort wird nun das Symbol für N-äre Assoziationen, das Sie natürlich wieder verschieben können, gezeichnet.

Fast geschafft... Wenn Sie Ihr Diagramm mit dem obigen vergleichen, sollten jetzt noch drei Generalisierungen und vier Aggregationen fehlen. Um die Generalisierungen zu zeichnen, wechseln Sie in den Generalisierungsmodus (Button „21“ oder „Insert Generalization“ aus „Mode“) und klicken nun nacheinander auf Zutat, Gewuerz, Zutat, Gemuese, Zutat, Fleisch. Es existieren nun drei Generalisierungen, die jeweils von Zutat ausgehen und bei den Unterklassen enden. Verschieben Sie die Enden wie Sie dies schon mit den Assoziationen getan haben.

Um die Aggregationen zu zeichnen, wechseln Sie auch hier in den entsprechenden Modus (Button „20“ oder „Insert Aggregation“ aus „Mode“) und klicken wiederum abwechselnd auf Menue, Speise, Rezept, Arbeitsgang, Rezept, Zutat, Rohstoff und Rohstoff. Nach jeweils zwei Klicks sollte eine Aggregation zwischen den gewählten Klassen erscheinen. Wenn Sie auch diese wie gewünscht plaziert haben, bleibt noch eine Aufgabe: Wie schon bei den Assoziationen können auch Aggregationen Kardinalitäten aufweisen. Schauen Sie auf das Beispieldiagramm, und schreiben Sie diese einfach ab (wenn Sie möchten).

Sie werden es kaum glauben, aber die Modellierung ist geschafft. Sie können es nun speichern, um Ihre Freunde mit einem echten Uni-Mensa-Schmaus zu überraschen (manche würden erschrecken sagen...). Oder, wenn Sie vielleicht wirklich ein Programm zur Essensversorgung schreiben möchten, sich Java-Klassen- Rümpfe und SQL-Code erzeugen lassen. Wir wünschen Ihnen jedenfalls viel Spaß mit deSIGN (und der nächsten Essenszubereitung...).

A.1.5 Fehlermeldungen und Problembehebung

Die Fehlermeldungen des Programms erscheinen in Message-Dialogen. Alle diese Message-Dialoge haben gemein, daß der Benutzer nach dem Erscheinen des Dialoges, nur die Möglichkeit hat, diesen durch das Betätigen des „OK“-Buttons zu schließen. Bei der Arbeit mit dem Programm kann Ihnen dieses mit den folgenden Fehlermeldungen passieren:

- File 'Filename' not found !
- No accordant logical element could be found !
- It is not allowed to associate a diamond symbol with itself or with other diamond symbols !
- There are no properties available for the chosen diagram element !

- Filename not given, please choose a valid filename !
- File already exists, please choose another filename !

Die Fehlermeldung „File 'Filename' not found !“ erhalten Sie dann, wenn Sie versuchen, eine nicht existierende, oder eine sich im falschen Format befindliche Datei zu laden. Falsches Format bedeutet hier, daß wenn Sie eine Datei, die mit der Serialisierungsmethode gespeichert wurde, laden möchten (mittels „Open Diagram / Open Serialized Diagram File...“), die Endung der Datei immer .dsg sein muß. Entsprechendes gilt auch für das Laden eines im Textformat gespeicherten Diagramms (mittels „Open Diagram / Open deSIGN Text Diagram File...“). Hier muß die Endung der Datei .dtg sein.

Achten Sie hier also auf die Endungen der Dateien und versuchen Sie nur eine existierende Datei zu laden. Um hierbei Fehler zu vermeiden, ist es einfacher, daß Sie beim Laden einer Datei im Filedialog (mit der Maus) durch die Verzeichnisse navigieren und so die zu ladende Datei aufrufen. Somit können Sie Tippfehler vermeiden.

Bei der Fehlermeldung „No accordant logical element could be found !“ handelt es sich darum, daß der Versuch eine Klasse zu „klonen“, obwohl keine Klasse im Diagramm existiert, zurückgewiesen wird. Für eine nicht-existierende Klasse kann natürlich auch kein Duplikat erzeugt werden. Daher müssen Sie zuerst mindestens eine Klasse im Diagramm zeichnen, damit Sie Klassen duplizieren können.

Der Versuch, bei einer n-ären Assoziation, anstatt Klassen mit dem Diamantsymbol zu verbinden, ein Diamantsymbol mit sich selbst oder mit mehreren anderen Diamantsymbolen zu assoziieren, führt zu der Fehlermeldung „It is not allowed to associate a diamond symbol with its self or with other diamond symbols !“.

Dieser Versuch des Benutzers würde natürlich auch keinen Sinn machen, da das Diamantsymbol in der objektorientierten Modellierung lediglich eine n-äre Assoziation (also die Beziehung zwischen mehreren Klassen) charakterisiert und daher nur die Verbindung von Klassen mit einbeziehen muß.

Der Eigenschaftsdialog einer n-ären Assoziation wird, wie wir bereits wissen, über das Diamantsymbol aktiviert. Der Versuch diesen über die Assoziation selbst, also über die Verbindungslinie zu aktivieren (wie es von den binären Assoziationen her bekannt ist), führt zu der Fehlermeldung „There are no properties available for the chosen diagram element !“.

Die Fehlermeldungen „Filename not given, please choose a valid filename !“ und „File already exists, please choose another filename !“ werden durch unerlaubte Aktionen im Printer-Dialog erzeugt. Im Printer-Dialog (befindet sich im „Mode Menü“), der Einstellungen für die Druckausgabe vorsieht, erhalten Sie die Fehlermeldung „Filename not given, please choose a valid filename !“, wenn Sie die Option „Print / File“ wählen, ohne einen Dateinamen anzugeben. Das Umlenken einer Druckausgabe in eine Datei muß also immer mit der Angabe eines gültigen Dateinamens abgeschlossen werden.

Wenn Sie eine Druckausgabe bereits in eine Datei umgelenkt haben, können Sie beim erneuten Drucken in eine Datei, keinen bereits existierenden Dateinamen angeben. Dieses würde gegebenenfalls zur Überschreibung der entsprechenden Datei führen und wird aus Sicherheitsgründen vom Programm durch die Fehlermeldung „File already exists, please choose another filename !“ zurückgewiesen.

A.2 Sprachvergleiche

Im Rahmen der Suche nach einer geeigneten Sprache für unseren Editor deSIGN haben wir lange überlegt, welche der vorhandenen Sprachen wohl am besten geeignet sei, unseren Ansprüchen zu genügen. Aus diesem Grund wurden viele Entwurfs- und Beschreibungssprachen unter die Lupe genommen, von denen ein Teil im folgenden näher beschrieben wird. Zu den untersuchten Sprachen gehören die von Booch[Boo91], OMT [RBP$^+$91], OOD [CY91a] [CY91b], Statemate [Pel95] und UML [BRJ97]. Aus diesen Sprachen haben wir die wichtigsten Konzepte herausgezogen und letztendlich unsere eigene Sprache SIGN entwickelt. Hier kamen uns dann die Vergleiche der einzelnen Sprachen zu gute: Wir konnten sehen, welche Möglichkeiten von welcher Sprache geboten werden, wie sie modelliert werden und (last but not least) ob sie von SIGN unterstützt werden.

A.2.1 OMT

A.2.1.1 Object Model

Klasse Beschreibt Gruppe von Objekten mit ähnlichen Attributen, gleichem Verhalten und Beziehungen zu anderen Objekten. Abstraktion eines Problems (sieheAbbildung A.10).

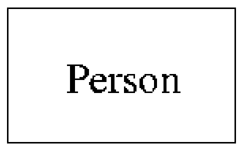


Abbildung A.10: Klasse

Instanz Konzept, Abstraktion oder Ding, fördert das Verstehen der realen Welt und liefert praktische Grundlage für Implementierungen (sieheAbbildung A.11).

Alle O. haben Identität und sind unterscheidbar.

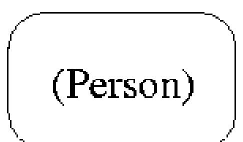


Abbildung A.11: Instanz

Attribut Datenenlement, das von Objekten in einer Klasse gehalten wird. Jedes A. hat einen Wert für jede Instanz (sieheAbbildung A.12).

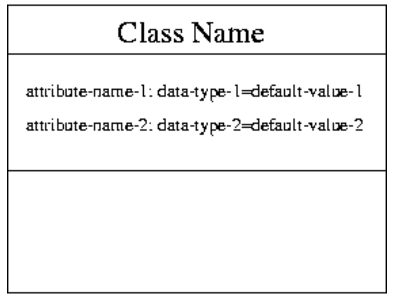


Abbildung A.12: Attribut

Operation Funktion, die von Objekten in einer Klasse benutzt werden kann. Gleiche O. kann auf mehrere verschiedene Klassen übertragen werden (sieheAbbildung A.13).

Eine Methode ist die Implementierung einer O. .

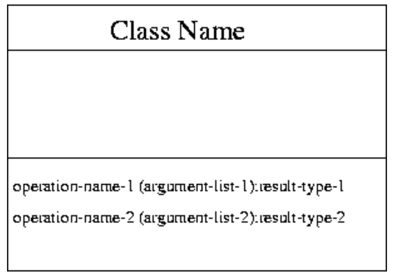


Abbildung A.13: Operation

Link Physikalische oder konzeptuelle Verbindung zwischen Objektinstanzen (sieheAbbildung A.14).

Mathematisch gesehen entspricht es einem Tupel, das eine geordnete Liste von Objektinstanzen ist.

Abbildung A.14: Link

Assoziation Beziehungen zwischen Klassen beschreibt Gruppe von Links mit gleicher Struktur und gleicher Bedeutung (sieheAbbildung A.15).

In Programmiersprachen häufig als Zeiger verwendet.

Abbildung A.15: Assoziation

Kardinalität Gibt an, wieviele Instanzen einer Klasse Beziehung zu einer Instanz einer anderen Klasse haben kann (sieheAbbildung A.16).

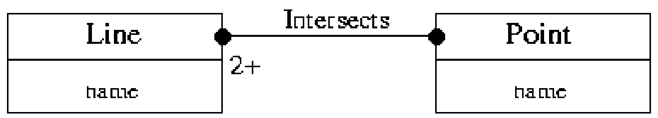


Abbildung A.16: Kardinalität

Assoziationsattribut Eigenschaft von Links in einer Assoziation (sieheAbbildung A.17).

Jedes A. hat einen Wert für jeden Link.

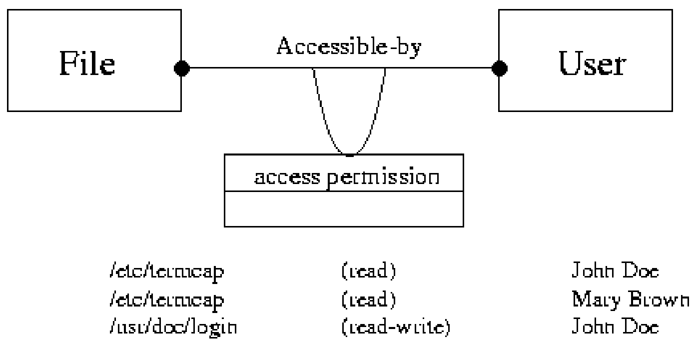


Abbildung A.17: Assoziationsattribut

Rollenname Name zum bezeichnen eines Endes einer Assoziation (sieheAbbildung A.18).

Nötig für Assoziationen zwischen zwei Objekten der gleichen Klasse.

Geordnete Assoziation Ordnung der Beziehungen zwischen Objekten durch Angabe von {ordered} (sieheAbbildung A.19).

Qualifizierte Assoziation Spezielles Attribut, das die Kardinalität einer 1:n bzw. n:m Assoziation reduziert (sieheAbbildung A.20).

Eine Qualifizierte Assoziation bezieht zwei Klassen und einen Qualifizierer.

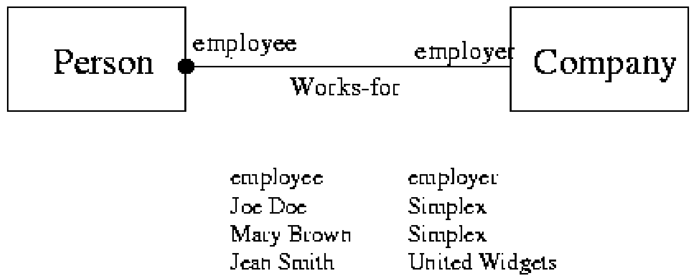


Abbildung A.18: Rollenname

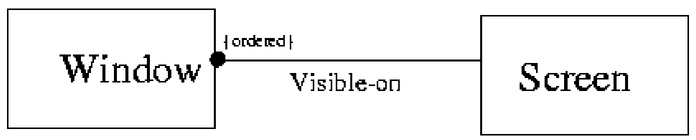


Abbildung A.19: Geordnete Assoziation

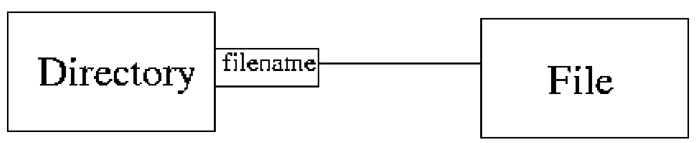


Abbildung A.20: Qualifizierte Assoziation

Aggregation Gekoppelte Form von Assoziationen mit einigen extra Bedingungen. Bedeutenste Eigenschaft ist die Transivität, ferner gilt auch die Antisymmetrie (sieheAbbildung A.21).

Eine Aggregation beschreibt eine Teil-Ganzes-Assoziation, wobei einzelnes Objekt nur in einer direkten Aggregation partizipiert werden kann und parallele Aggregationen zusammengefaßt werden, um Hierarchieebenen zu betonen.

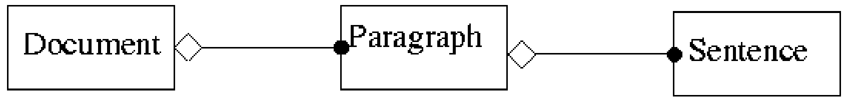


Abbildung A.21: Aggregation

A.2.1.2 Advanced Object Model

Message Kommunikation zwischen Objekten; gerichtete Beziehung (call oder event) (sieheAbbildung A.22).

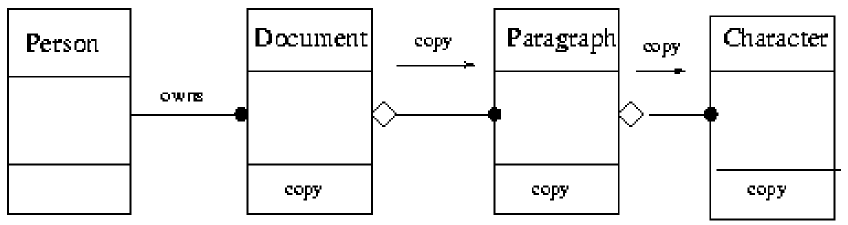


Abbildung A.22: Advanced Object Model

Propagation Fortpflanzung von Operationen entlang von Assoziationen (spez. Aggregationen), auch Aggregatoperation definiert durch Operationen der Bestandteile.

Vererbung gerichtete Beziehung zwischen Klassen, Unterklasse erbt Operationen und Attribute einer oder mehrerer Oberklassen (Mehrfachvererbung) (disjunkt) (sieheAbbildung A.23).

Abstrakte Klassen Klassen die keine direkten Instanzen haben (sieheAbbildung A.24).

Instanz-von-Beziehung

Beziehung zwischen Objekten und Klassen (gerichtet) (sieheAbbildung A.25).

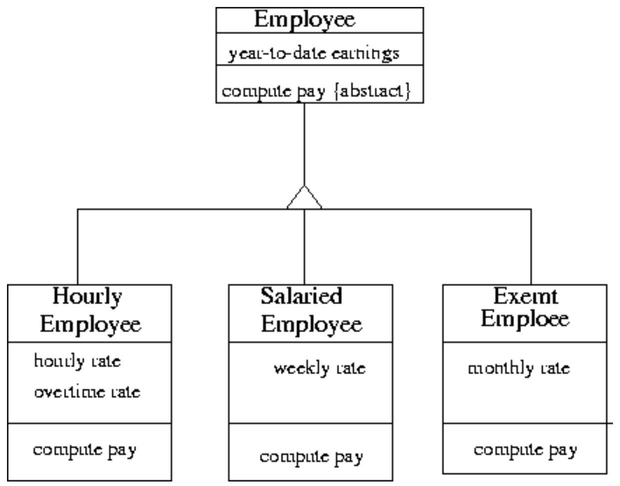


Abbildung A.23: Vererbung

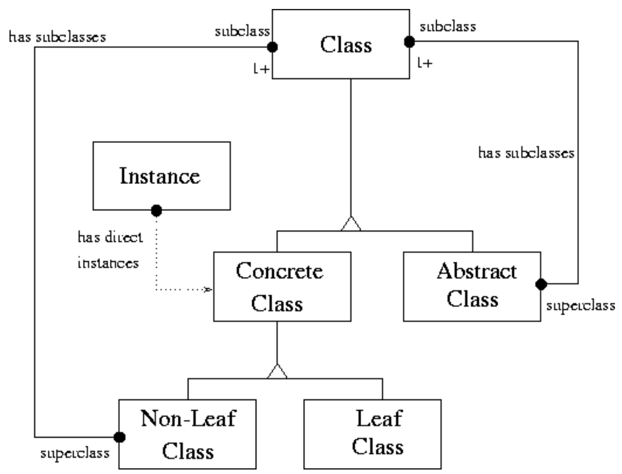


Abbildung A.24: Abstrakte Klassen

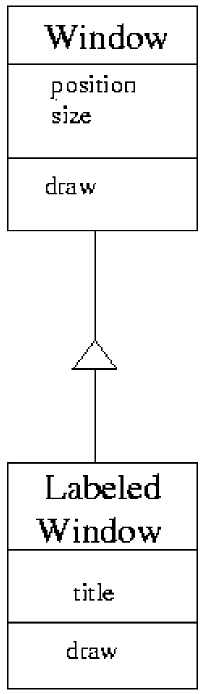


Abbildung A.25: Spezialisierung mit Erweiterung

Spezialisierung mit Erweiterung

Durch Hinzufügen von Operationen und Attributen.

Spezialisierung mit Begrenzung

Durch Einschränken zulässiger Attributwerte, des Verhaltens (sieheAbbildung A.26).

Nicht-disjunkte Vererbung

Indirekte Instanzen von Oberklasse können mehreren Unterklassen angehören.

Mehrfachvererbung Unterklasse erbt Eigenschaften mehrerer Oberklassen (sieheAbbildung A.27).

Delegation Aggregate implementieren Operationen durch delegieren an Komponenten (sieheAbbildung A.28).

Klassenattribute Attribute, die es nur einmal pro Klasse gibt (sind also für jedes Objekt identisch) (sieheAbbildung A.29).

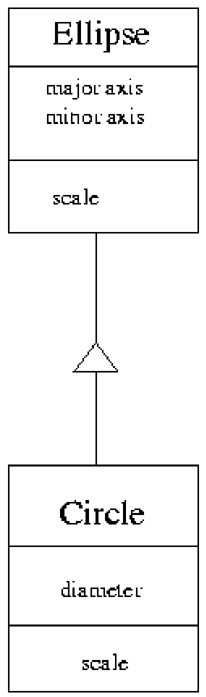


Abbildung A.26: Spezialisierung

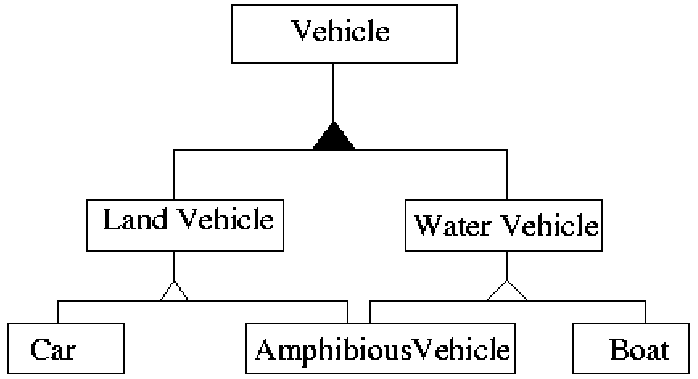


Abbildung A.27: Mehrfachvererbung

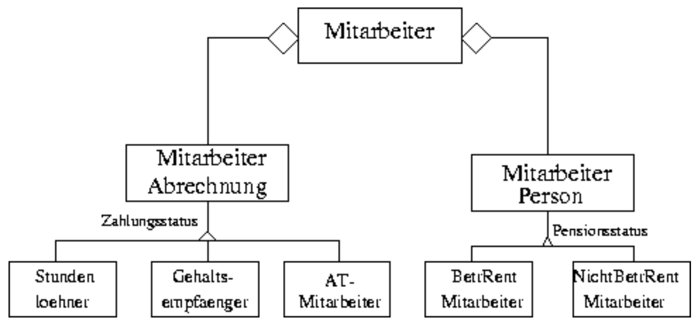


Abbildung A.28: Delegation

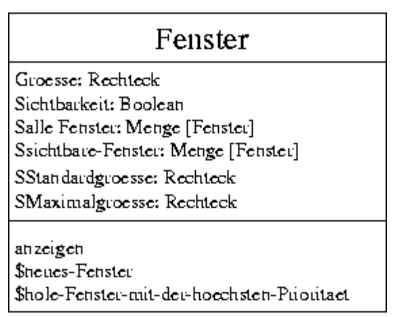


Abbildung A.29: Klassenattribute

Klassenoperationen Operationen die außerhalb eines Objektkontextes und ohne Zugriff auf Instanzeigenschaften auskommen.

Metadaten Instanz-von-Beziehungen zwischen Klassen, gerichtet (Instanzen der einen Klasse sind beschrieben durch Instanzen der Musterklasse (Metaklasse)) (sieheAbbildung A.30).

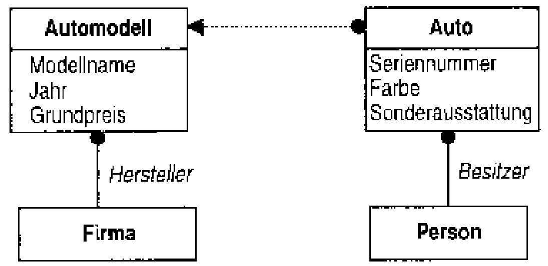
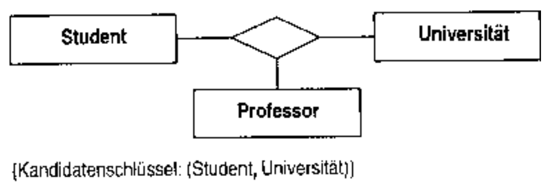


Abbildung A.30: Metadaten

Kandidatenschlüssel Angabe der Kardinalität (vorzugsweise mehrstelliger) Beziehungen durch die Beziehungsinstanz identifizierende Objektmengen (sieheAbbildung A.31).



Student	Professor	Universität
Mary	Prof Weaver	Cambridge
Mary	Prof Rumrow	MIT
Susan	Prof Weaver	MIT
Susan	Prof Weaver	Cambridge
Bob	Prof Shapiro	Oxford

Abbildung A.31: Kandidatenschlüssel

Assoziationen als Klassen Assoziation wird durch eine Klasse modelliert (sieheAbbildung A.32).

Einschränkungen (Text)

Invarianten für einzelne Modellentitäten (sieheAbbildung A.33).

Einschränkungen (Links)

Invarianten für mehrere Modellentitäten, Beziehung zwischen Modellentitäten (sieheAbbildung A.34).

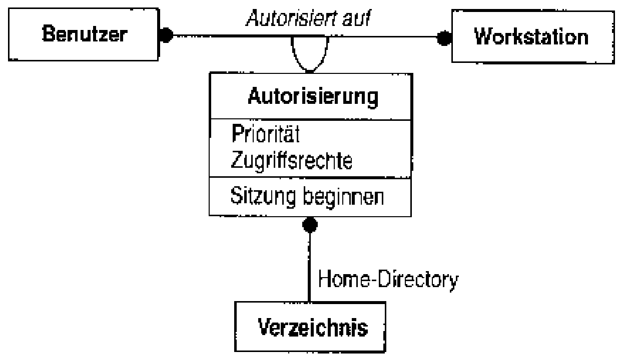


Abbildung A.32: Assoziationen als Klassen

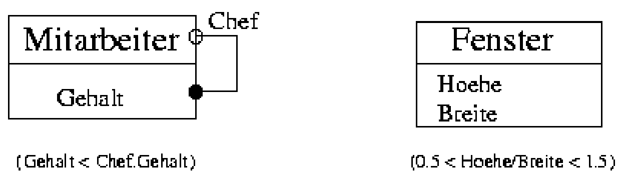


Abbildung A.33: Einschränkungen (Text)

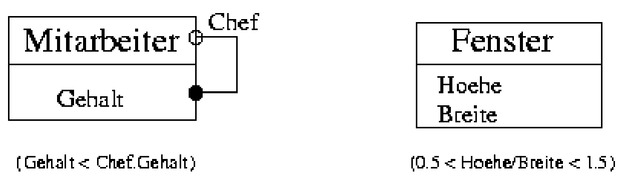


Abbildung A.34: Einschränkungen (Links)

Abgeleitete Modellentitäten

Kennzeichnung prinzipiell redundanter Modellentitäten, die aus anderen Berechnet werden können.

A.2.1.3 Dynamic Model

Zustand (state)

Ein Zustand ist eine Menge der Attribute eines Objektes zu einem Zeitpunkt (sieheAbbildung A.35).

Abbildung A.35: Zustand

Ereignis (event)

Ein Vorfall, der eine Zustandsänderung des Systems bewirken kann (sieheAbbildung A.36).

Abbildung A.36: Ereignis

Aktivität (activity)

Eine Operation, die bei ihrer Ausführung Zeit verbraucht (sieheAbbildung A.37).

Abbildung A.37: Aktivität

Aktion (action)

Eine Operation, die im Einsatz keine Zeiteinheit verbraucht (sieheAbbildung A.38).

Abbildung A.38: Aktion

Bedingungen (condition)

Operationen mit Booleschen Bedingungen (sieheAbbildung A.39).

Abbildung A.39: Bedingungen

Start

Kennzeichnet einen Anfangszustand (sieheAbbildung A.39).

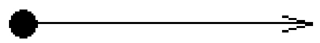


Abbildung A.40: Start

verschachteltes Zustandsdiagramm

Beinhaltet Generalisierung von Zuständen (mit Unter- und Überzuständen) (sieheAbbildung A.41).

Nebenläufigkeit innerhalb von Zuständen

Unterteilung eines Überzustands in nebenläufige Unterzuständen (sieheAbbildung A.42).

Sending Event

Versenden eines Ereignises an ein anderes Objekt.

Send Event (attrib)

Entry/Exit

Alternative zur Aktion, die das Starten oder Stoppen einer Aktivität bewirkt (sieheAbbildung A.43).

Internal Aktion

Ereignisse, die keinen Zustandswechsel bewirken (sieheAbbildung A.44).

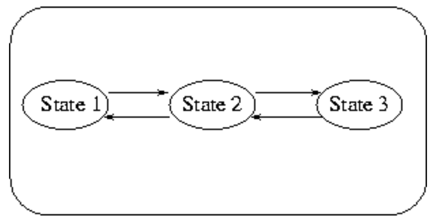


Abbildung A.41: verschachteltes Zustandsdiagramm

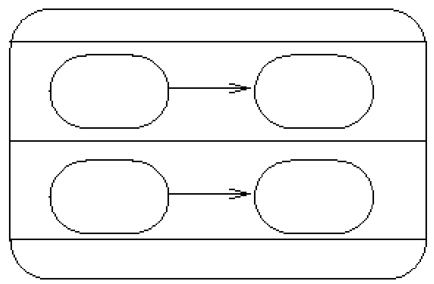


Abbildung A.42: Nebenläufigkeit innerhalb von Zuständen

Abbildung A.43: Entry/Exit

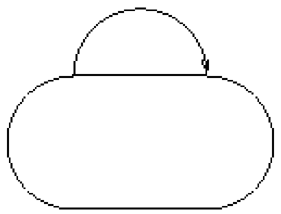


Abbildung A.44: Internal Aktion

Synchronisation von nebenläufigen Aktivitäten

Ensteht, wenn ein Objekt zwei oder mehr Aktivitäten nebenläufig ausführen möchte (sieheAbbildung A.45).

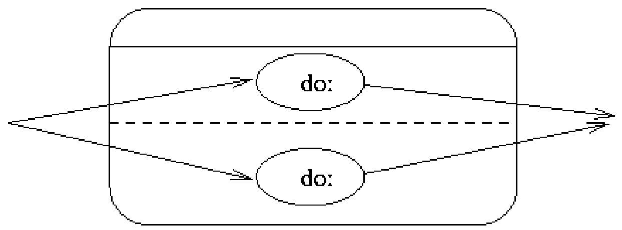


Abbildung A.45: Synchronisation von nebenläufigen Aktivitäten

A.2.1.4 Functional Model

Der Datenfluß (Data Flow)

Die einfachste Form eines Datenflusses ist ein einfacher Wert, er wird im funktionales Modell dargestellt als Pfeil der mit einer Bezeichnung für den Wert versehen sein kann (sieheAbbildung A.46).

Abbildung A.46: Datenfluß (einfacher)

Soll ein Datenwert an mehrere Stellen verteilt werden, also mehreren Prozessen als Eingabewert dienen, so kann dies durch einen sich aufspaltenden Pfeil symbolisiert werden, der nur mit einem Namen versehen sein kann (sieheAbbildung A.47).

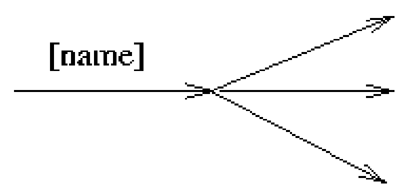


Abbildung A.47: Datenfluß (mehrere Prozesse)

Wenn es sich bei dem Datum um einen zusammengesetzten Typ handelt, können die einzelnen Komponenten an unterschiedlichen Prozessen als Eingabe dienen. Hierbei werden die einzelnen Komponentennamen an die sich aufspaltenden Pfeile geschrieben und an den Pfeil, von dem sie ausgehen der Name des zusammengesetzten Datums (sieheAbbildung A.48).

Die Vereinigung von mehreren einzelnen Werten zu einem zusammengesetzten Datum geschieht analog (sieheAbbildung A.49).

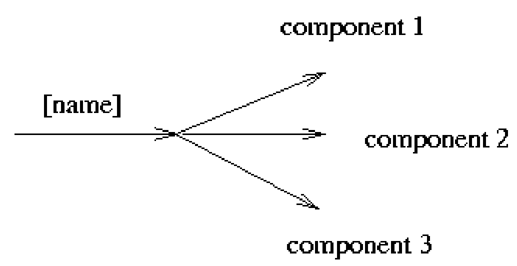


Abbildung A.48: Datenfluß (Komponenten verteilend)

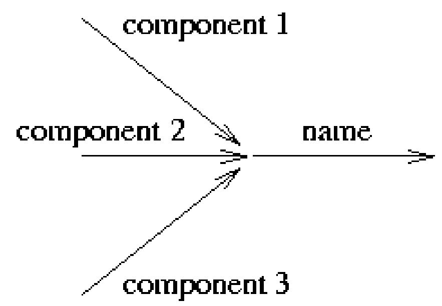


Abbildung A.49: Datenfluß (Komponenten vereinigend)

Der Prozeß (Process)

Prozesse sind Transformationen, sie führen Funktionen auf Eingabewerten durch und berechnen Ausgaben.Die Eingaben werden durch mit Namen versehene Pfeile symbolisiert, die auf die Ellipse zeigen, die wiederum den Prozeß darstellt. In der Ellipse sollte ein aussagekräftiger Name für die Transformation stehen. Die Ausgaben des Prozesses von der Ellipse wegzeigend (sieheAbbildung A.50).

Akteure (Actors)

Es wird zwischen produzierenden und konsumierenden Aktoren unterschieden. Produzierende Aktoren sind Objekte aus dem Objektmodell die Werte in das Datenflußdiagramm liefern, sie stehen immer an den Rändern des Datenflußdiagramms (sieheAbbildung A.51).

Das Gegenteil sind konsumierende Actors, sie entnehmen Daten aus dem Datenflußdiagramm (sieheAbbildung A.52).

Datenspeicher (Data Store)

Datenspeicher sind passive Elemente im funktionalen Modell. In ihnen können Daten gespeichert werden, auf die später zugegriffen werden kann. Der Pfeil in den Datenspeicher zeigt Operationen zum Verändern des Inhalts an. Dies kann das Einfügen, das Verändern oder das Löschen von Daten im Datenspeicher sein. Der Pfeil der vom Datenspeicher wegzeigt symbolisiert Zugriffsoperationen, die Daten aus dem Datenspeicher liefern (sieheAbbildung A.53).

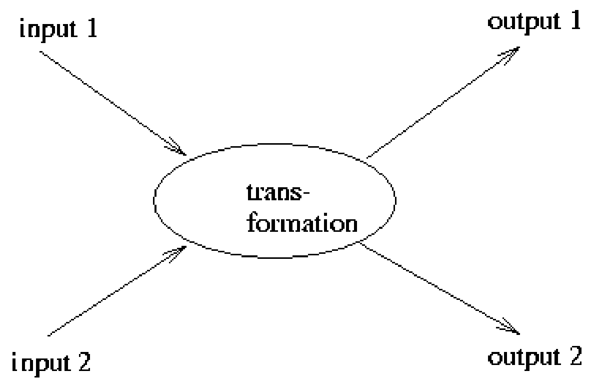


Abbildung A.50: Prozeß

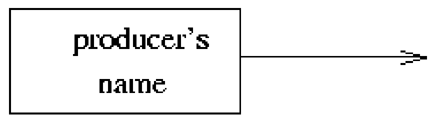


Abbildung A.51: Aktoren (produzierend)

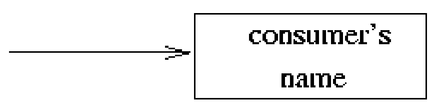


Abbildung A.52: Aktoren (konsumierend)

Abbildung A.53: Datenspeicher

Um einen Datenspeicher zu initialisieren, benutzt man als Eingangssymbol ein leeres Dreieck statt einer Pfeilspitze (sieheAbbildung A.54).

Abbildung A.54: Datenspeicher

Der Kontrollfluß (Control Flow)

Der Kontrollfluß ist redundant zu Angaben im dynamischen Modell. Er ist ein gepunkteter mit einem Namen versehener Pfeil, der für einen booleschen Wert steht, der zunächst der True sein muß, damit der Prozeß, der an der Pfeilspitze steht ausgeführt werden kann (sieheAbbildung A.55).

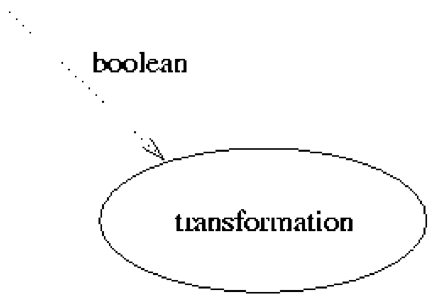


Abbildung A.55: Kontrollfluß

A.2.2 OOD

A.2.2.1 Object Model

Klasse & Objekt (Class & Object)

Objekt: Eine Abstraktion von etwas in einem Anwendungsgebiet oder seiner Implementierung, die die Fähigkeit des Systems wiedergibt, darüber Informationen zu halten, damit zu interagieren oder beides.

Klasse: Eine Beschreibung von einem oder mehreren Objekten, darstellbar durch einen einheitlichen Satz von Attributen und Services (sieheAbbildung A.56).

Klasse (Class)

Klasse: Eine Beschreibung von einem oder mehreren Objekten, darstellbar durch einen einheitlichen Satz von Attributen und Services; zusätzlich kann beschrieben sein, wie neue Objekte in dieser Klasse erzeugt werden (sieheAbbildung A.57).

Abbildung A.56: Klasse und Objekt

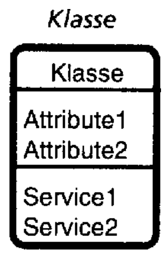


Abbildung A.57: Klasse

Instanzverbindung

Eine Instanzverbindung ist eine Form von Verbindung in einem Anwendungsgebiet, die ein Objekt mit einem anderen Objekt benötigt, um seine Verantwortlichkeiten zu erfüllen. Eine Instanzverbindung repräsentiert einen Teil der Statusinformation eines Objektes (sieheAbbildung A.58).

Abbildung A.58: Instanzverbindung

Teil/Ganzes Beziehung (Whole/Part)

Die Whole/Part Beziehung kann als "hat-ein" Struktur bezeichnet werden (sieheAbbildung A.59).

Generalisierung/Spezialisierung

Die Generalisierung/ Spezialisierung dagegen läßt sich als ïst eine oder ïst eine Art von Struktur bezeichnen. Eine Struktur ist dabei ein Ausdruck von Anwendungsgebiets- Komplexität und sie ist relevant für die Systemverantwortlichkeiten. Es gibt sowohl Gen/Spec-Strukturen als auch Whole/Part-Strukturen (sieheAbbildung A.60).

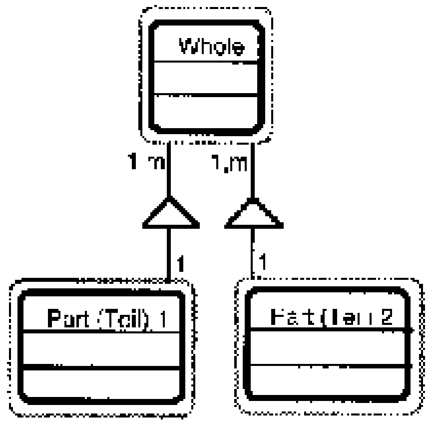


Abbildung A.59: Teil/Ganzes Beziehung

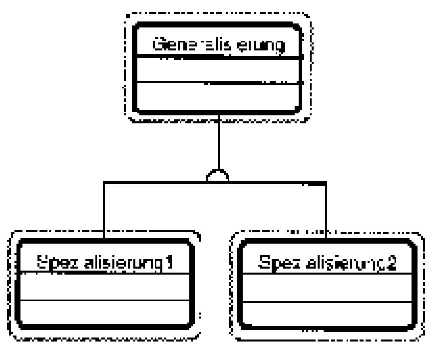


Abbildung A.60: Generalisierung/Spezialisierung

Nachrichtenverbindung

Eine Nachrichtenverbindung modelliert die Verarbeitungsabhängigkeit eines Objekts, das einen Service benötigt, um seine Verantwortlichkeiten zu erfüllen (sieheAbbildung A.61).

Abbildung A.61: Nachrichtenverbindung

A.2.3 BOOCH

A.2.3.1 Object Model

Klasse

Beschreibt Gruppe von Objekten mit ähnlichen Attributen, gleichem Verhalten und Beziehungen zu anderen Objekten. Abstraktion eines Problems (sieheAbbildung A.62).

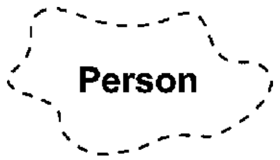


Abbildung A.62: Klasse

Attribut

Datenenlement, das von Objekten in einer Klasse gehalten wird. Jedes A. hat einen Wert für jede Instanz (sieheAbbildung A.63).

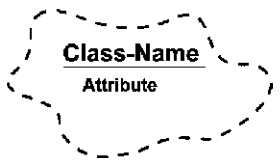


Abbildung A.63: Attribut

Operation

Funktion, die von Objekten in einer Klasse benutzt werden kann. Gleiche Operationen können auf mehrere verschiedene Klassen übertragen werden.

Eine Methode ist die Implementierung einer Operation (sieheAbbildung A.64).

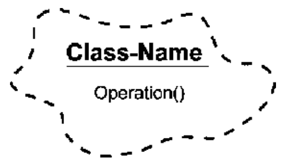


Abbildung A.64: Operation

abstrakte Klasse

Klasse die keine direkten Instanzen hat.

Eine abstrakte Klasse wird unter der Vorraussetzung erzeugt, daß ihre konkreten Unterklassen sie um Struktur und Verhalten ergänzen, eigentlich durch Implementierung von abstrakten Operationen (sieheAbbildung A.65).

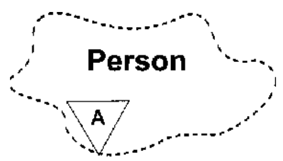


Abbildung A.65: abstrakte Klasse

Assoziation

Beziehungen zwischen Klassen beschreibt Gruppe von Links mit gleicher Struktur und gleicher Bedeutung (sieheAbbildung A.66).

In Programmiersprachen häufig als Zeiger verwendet.

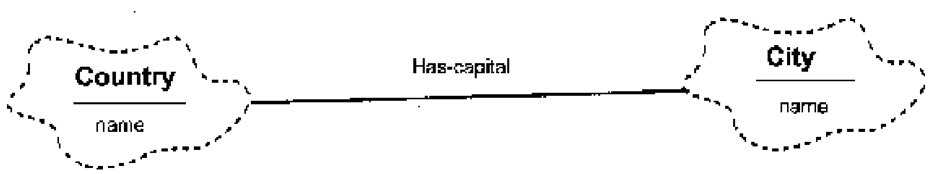


Abbildung A.66: Assoziation

Verwendungs-Beziehung

Verfeinerung einer Assoziation, wobei wir angeben, welche Klasse der Client und welche der Supplier für bestimmte Dienste ist.

Der Client ist auf irgendeine Art und Weise vom Supplier abhängig, um bestimmte Dienstleistungen zu realisieren (sieheAbbildung A.67).

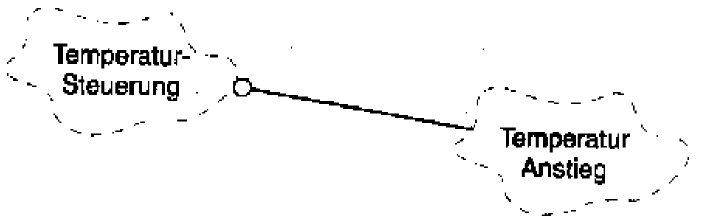


Abbildung A.67: Verwendungs-Beziehung

Kardinalität

Gibt an, wieviele Beziehungen eine Instanzen einer Klasse zu einer Instanz einer anderen Klasse haben kann (sieheAbbildung A.68).

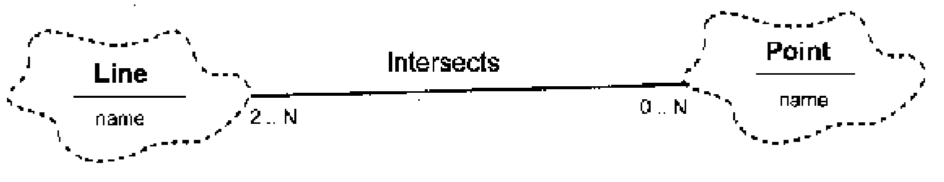


Abbildung A.68: Kardinalität

Vererbung

Gerichtete Beziehung zwischen Klassen, Unterklasse erbt Operationen und Attribute einer oder mehrerer Oberklassen (Mehrfachvererbung) (disjunkt) (sieheAbbildung A.69).

Klassenkategorie

Logische Ansammlung von Klassen, von denen einige für andere Klassen sichtbar sind, andere nicht.

Eine Klassenkategorie ist ein Aggregat, das Klassen und andere Klassenkategorien enthält (sieheAbbildung A.70).

Mehrfachvererbung

Unterklasse erbt Eigenschaften mehrerer Oberklassen (sieheAbbildung A.71).

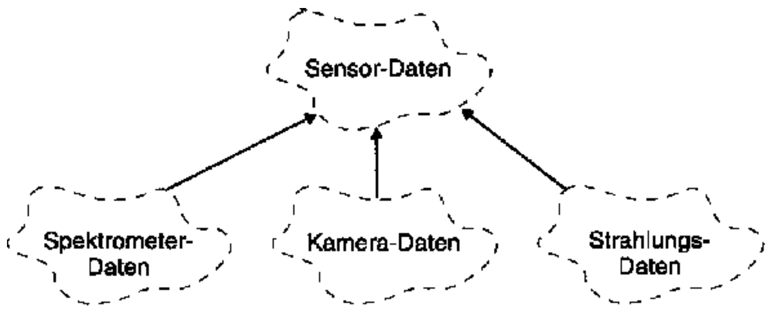


Abbildung A.69: Vererbung

Abbildung A.70: Klassenkategorie

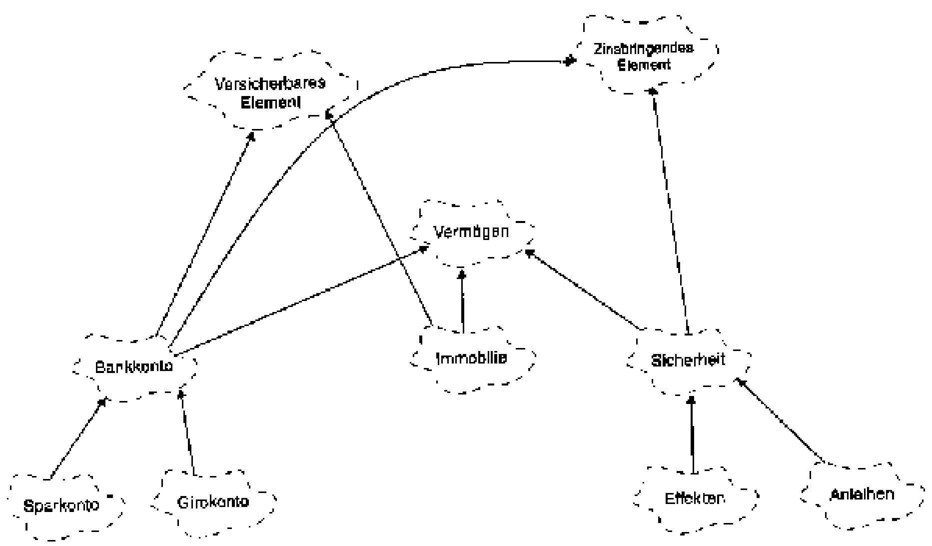


Abbildung A.71: Mehrfachvererbung

Generische Klasse

Klasse, die als Schablone für andere Klassen dient, in denen die Schablone von anderen Klassen, Objekten und/oder Operationen parametrisiert werden kann. Eine generische Klasse muß instantiiert werden (d.h. ihre Parameter müssen angegeben werden), bevor Objekte erzeugt werden können (sieheAbbildung A.72).

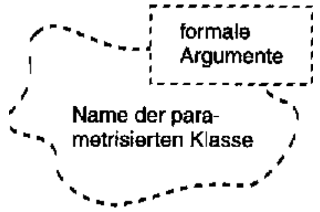


Abbildung A.72: Generische Klasse

Instantiierte Klasse

Eine instantiierte Klasse definiert eine neue Klasse, die sich von allen anderen konkreten Klassen derselben Familie unterscheidet, deren aktuelle Parameter unterschiedlich sind (sieheAbbildung A.73).

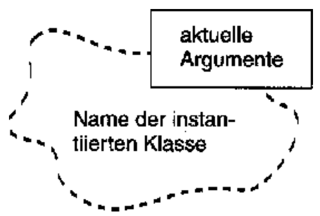


Abbildung A.73: Instantiierte Klasse

Metadaten

Die Klasse einer Klasse. Eine Klasse, deren Instanzen selbst wieder Klassen sind (sieheAbbildung A.74).

Abbildung A.74: Metadaten

Klassen-Utilities

Eine Ansammlung von freien Unterprogrammen bzw. in C++ eine Klasse, die nur statische Elemente und/oder statische Elementfunktionen enthält (sieheAbbildung A.75).

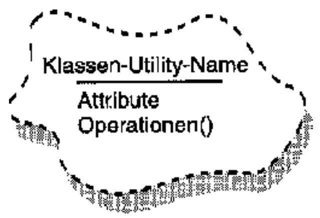


Abbildung A.75: Klassen-Utilities

Schachtelung

Klassen können in anderen Klassen geschachtelt werden, um Kontrolle über den Namensraum zu erhalten, meistens taktisch bedingte Designentscheidung (sieheAbbildung A.76).

Abbildung A.76: Schachtelung

Export-Steuerung

Dient dazu den Zugriff auf die Schnittstelle sehr detailliert zu spezifizieren (sieheAbbildung A.77).

Folgende Symbole:

keine Ergänzung, Zugriff ist public, Zugriff ist protected, Zugriff ist private, Nur die Implementierung hat Zugriff.

physikalisches Enthalten

Aggregation gibt Teil/Ganzes-Hierarchie an, diese muß nicht unbedingt physikalisches Beinhalten bedeuten (sieheAbbildung A.78):

Berufsgruppe hat mehrere Mitglieder, das heißt nicht, daß Gruppe ihre Elemente besitzt:

Wahl der Aggregation als physikalisches Enthalten ist eine taktische Frage.

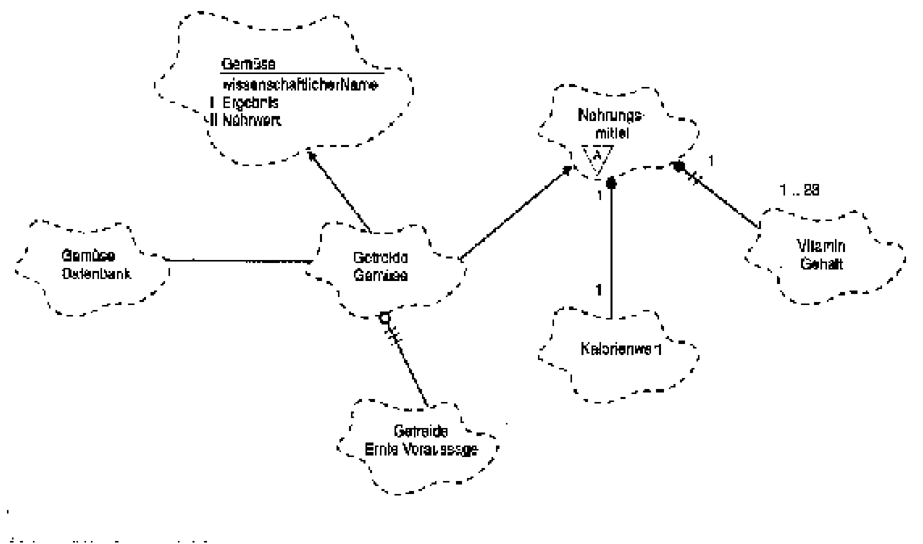


Abbildung A.77: Export-Steuerung

In hybriden Sprachen existieren zwei Arten von physikalisches Enthalten: über den Wert, über eine Referenz.

Eigenschaften

In manchen Sprachen sind einige Beziehungs-Qualifikationen, so verwoben und ihre Semantik so fundamental , daß sie die Verwendung von speziellen Symbolen einführen (sieheAbbildung A.79).

In C++ gibt es drei solcher Eigenschaften :

static: Beziehung eines Klassenelementobjekts oder einer Klassenelementfunktion.

virtual: Beziehung einer gemeinsam verwendeten Basisklasse in einem Vererbungsnetz mit der Gestalt eines Diamanten.

friend: Beziehung einer Klasse, die einer anderen Klasse Rechte erteilt, damit diese auf nicht-public Teile zugreifen kann.

Role Name (Rolle)

Name zum bezeichnen eines Endes einer Assoziation (sieheAbbildung A.80).

Nötig für Assoziationen zwischen zwei Objekten der gleichen Klasse.

Einschränkungen

Ausdruck einer semantischen Bedingung, die erfüllt werden muß.

Garantien, die nur zutreffen, wenn der Status des Systems stabil ist (sieheAbbildung A.81).

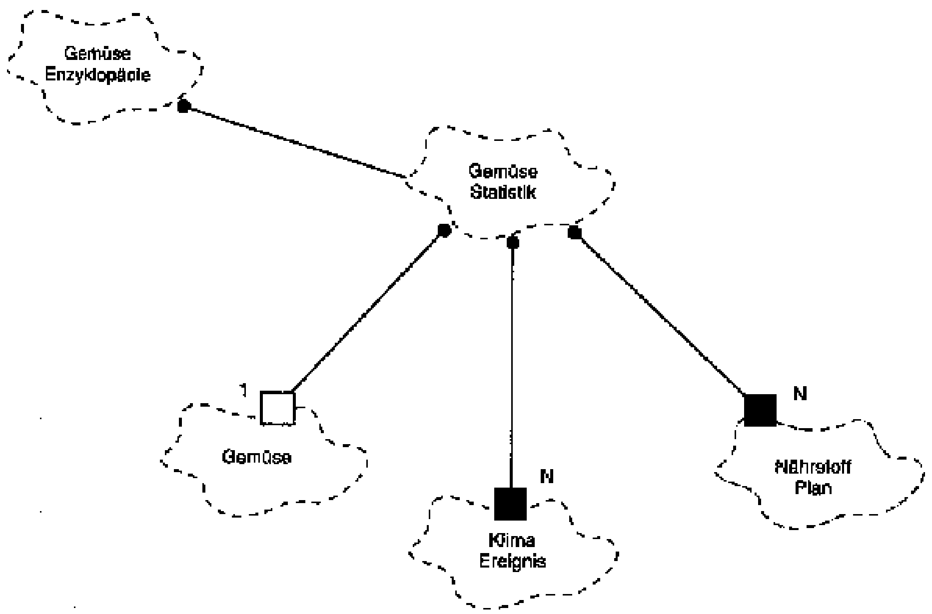


Abbildung A.78: physikalisches Enthalten

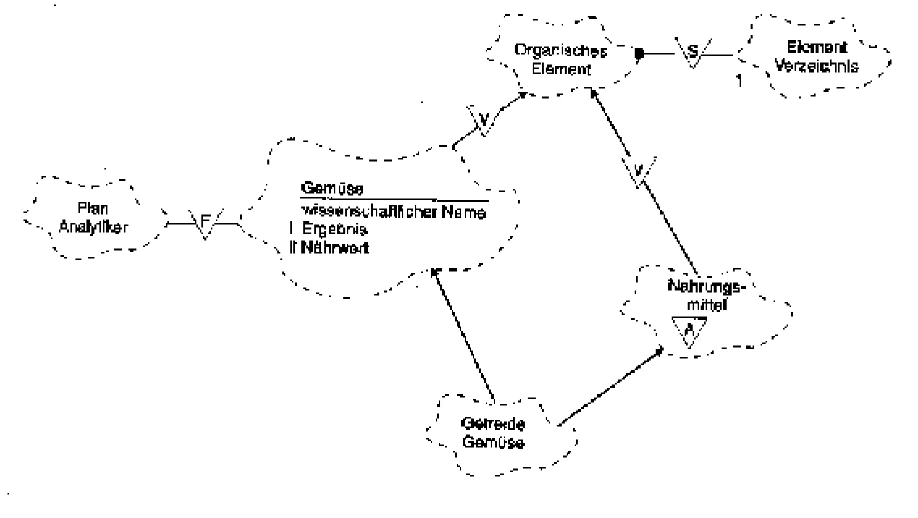


Abbildung A.79: Eigenschaften

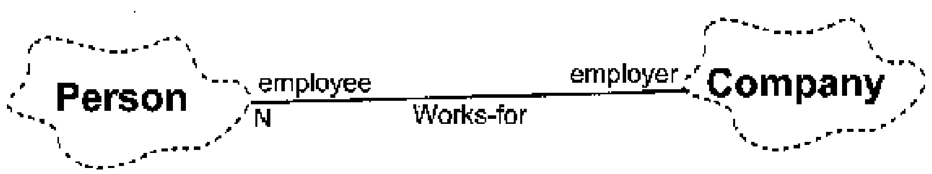


Abbildung A.80: Rollenname

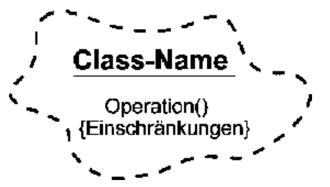


Abbildung A.81: Einschränkungen

Assoziationsattribut

Eigenschaft von Links in einer Assoziation.

Jedes Assoziationsattribut hat einen Wert für jeden Link (sieheAbbildung A.82).

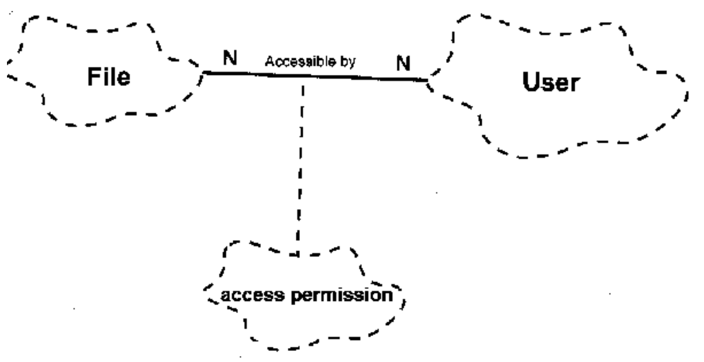


Abbildung A.82: Assoziationsattribut

Instanz

Konzept, Abstraktion oder Ding, fördert das Verstehen der realen Welt und liefert praktische Grundlage für Implementierungen.

Alle Objekte haben Identität und sind unterscheidbar (sieheAbbildung A.83).

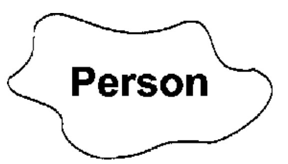


Abbildung A.83: Instanz

Link

Physikalische oder konzeptuelle Verbindung zwischen Objektinstanzen (sieheAbbildung A.84).

Mathematisch gesehen entspricht es einem Tupel, das eine geordnete Liste von Objektinstanzen ist.

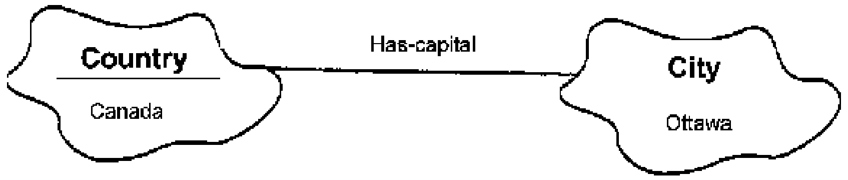


Abbildung A.84: Link

Aggregation

Gekoppelte Form von Assoziationen mit einigen extra Bedingungen. Bedeutenste Eigenschaft: Transitivität, ferner gilt auch Antisymmetrie.

Beschreibt Teil-Ganzes-Assoziation, wobei einzelnes Objekt nur in einer direkten Aggregation partizipiert werden kann und parallele Aggregationen zusammengefaßt werden, um Hierarchieebenen zu betonen (sieheAbbildung A.85).

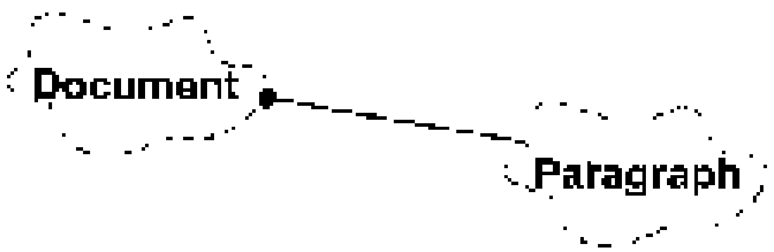


Abbildung A.85: Aggregation

Sichbarkeit

In manchen Szenarien ist es sinnvoll, zu verfolgen, in welcher Art und Weise ein Objekt sichtbar für anderes Objekt ist (sieheAbbildung A.86).

Für die Art der Sichtbarkeit gibt es folgende Zusätze :

G : Supplier-Objekt ist global für Client.

P : Supplier-Objekt ist Parameter einer Operation des Clients.

F : Supplier-Objekt ist Teil des Client-Objekts.

L : Supplier-Objekt ist lokal deklariertes Objekt im Gültigkeitsbereich des Objektdiagramms.

Aktive Objekte und Synchronisation

Wenn Objekt über Link Nachricht an anderes Objekt sendet, heißen beide Objekte synchronisiert (sieheAbbildung A.87).

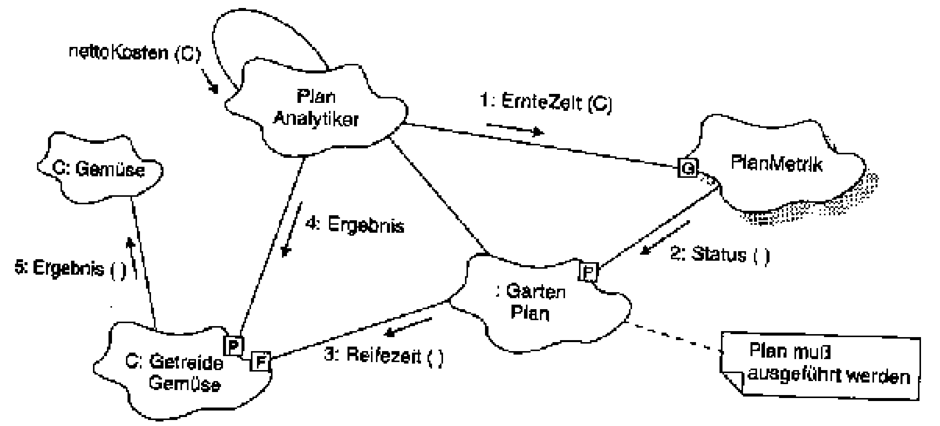


Abbildung A.86: Sichbarkeit

Aktive Objekte können eigenen Steuerungsprozeß beinhalten.

Bei Link zwischen aktiven und passiven Objekte, muß eins der folgenden Ansätze für die Synchronisation gewählt werden: *Sequentiell, Bewacht, Synchron.*

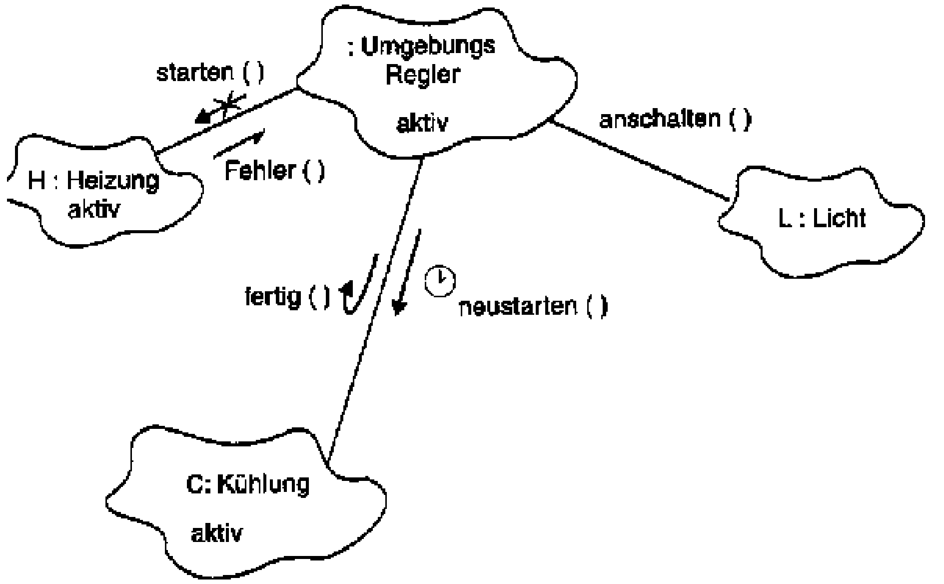


Abbildung A.87: Aktive Objekte und Synchronisation

A.2.4 STATEMATE

A.2.4.1 Activity-charts

Aktivität (Activity)

Eine Aktivität wird als rechteckige Box dargestellt (sieheAbbildung A.88). In der linken oberen Ecke wird der Name (sinnvollerweise Verben) der Aktivität angegeben, Unter-Aktivitäten werden als kleinere Boxen hineingezeichnet. Diese Unter-Aktivitäten können in der gleichen Darstellung noch weiter verfeinert werden, so daß auch Unter-Unter-Aktivitäten im Diagramm sichtbar werden.

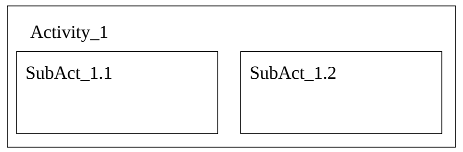


Abbildung A.88: Aktivität

Basis-Aktivität (Basic Activity)

Am Ende einer Aktivitäten-Hierarchie stehen als unterste Aufgliederung die Basis-Aktivitäten, die nicht weiter in Unter-Aktivitäten aufgespalten werden können (sieheAbbildung A.89).

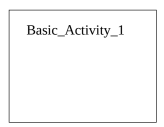


Abbildung A.89: Basis-Aktivität

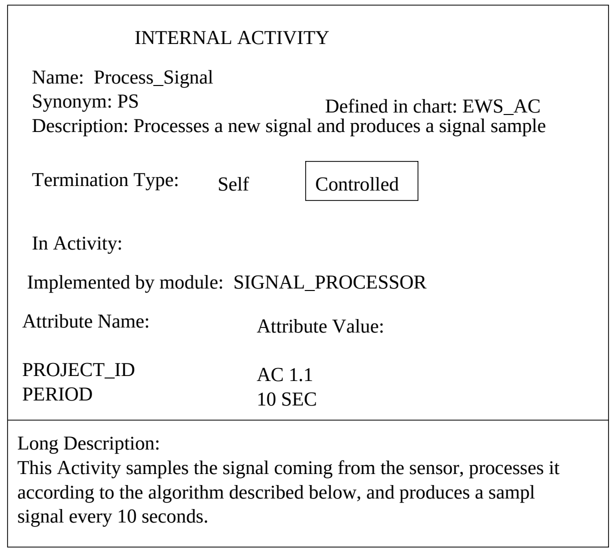
Formular (Form)

Zu jeder Aktivität gibt es ein Formular, das insbesondere die nicht graphisch darstellbaren Aspekte der Aktivität beinhaltet. Dieses Formular beinhaltet unter anderem textuelle Beschreibungen, Attribute und Beziehungen zu anderen Elementen (sieheAbbildung A.90).

Die Bezeichnungen der einzelnen Aspekte sind: *Title, Name, Synomym, Defined in Chart, Description, Termination, Is activity, Implemented by module, Attribute name, Attribute value, Long description.*

activity-chart flow-line

Stellt den Fluß einer einzelnen oder einer Gruppe von Informationen von einer Aktivität zu einer anderen dar. Die activity-charts flow-lines beginnen beim Produzenten der Informationen und enden beim Konsumenten dieser Information. Wenn die Enden der Activity-charts flow-line mit einem Balken gekennzeichnet sind, ist der Ursprung oder die Verwertung der Daten noch nicht bekannt. Eine activity-charts flow-line muß nicht in einer Basis-Aktivität enden, endet sie nicht in einer Basis-Aktivität, dann steht die Informationen allen Unter-Aktivitäten zur Verfügung. Sie stellen weder dar, wie die Information übermittelt wird, noch



INTERNAL ACTIVITY

Name: Process_Signal
Synonym: PS
Defined in chart: EWS_AC
Description: Processes a new signal and produces a signal sample

Termination Type: Self Controlled

In Activity:

Implemented by module: SIGNAL_PROCESSOR

Attribute Name:	Attribute Value:
PROJECT_ID	AC 1.1
PERIOD	10 SEC

Long Description:
This Activity samples the signal coming from the sensor, processes it according to the algorithm described below, and produces a sampl signal every 10 seconds.

Abbildung A.90: Formular

ob sie zur Verfügung steht. Sie enthalten auch keine zeitabhängige Komponente, wann die Information zur Verfügung steht (sieheAbbildung A.91).

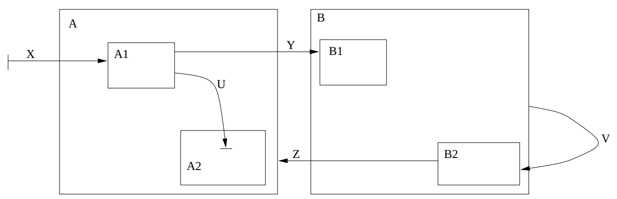


Abbildung A.91: activity-chart flow-line

data flow-line

Werden dargestellt als durchgezogene Linie. Beinhalten Daten für Berechnungen (sieheAbbildung A.92).

control flow-line

Dargestellt als gepunktete Linie. Sind Signale um die Aktivitäten zu steuern (sieheAbbildung A.93).

Steuerungs-Aktivitäten

Empfängt Signale um die umgebende Aktivität zu starten, laufen zu lassen oder zu stoppen (sieheAbbildung A.94).

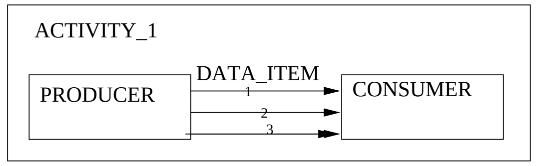


Abbildung A.92: data flow-line

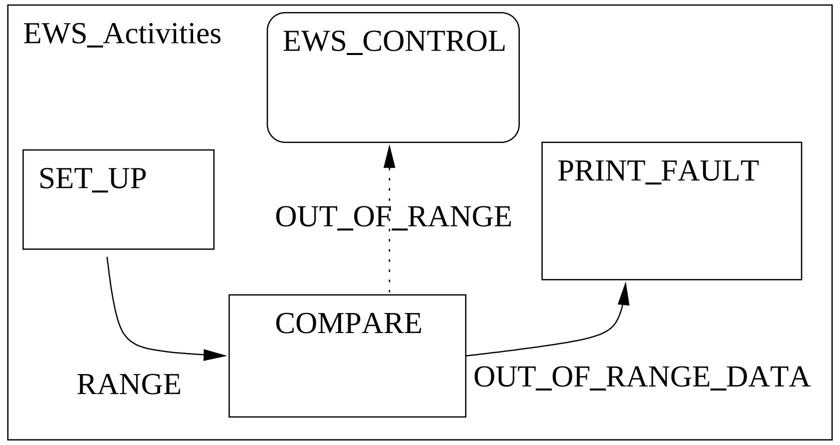


Abbildung A.93: control flow-line

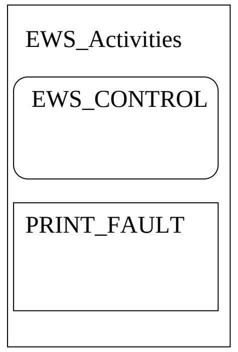


Abbildung A.94: Steuerungs-Aktivitäten

Ereignisse (Events)

Eine Form der Informationselemente, die Events überträgt (sieheAbbildung A.95).

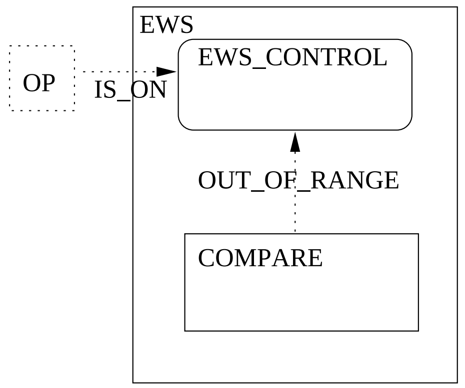
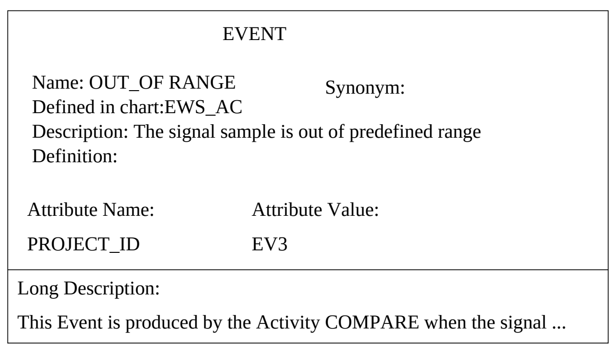


Abbildung A.95: Ereignisse

Ereignis-Formular (Event Form)

Textuelle Beschreibung eines Event-Flows (sieheAbbildung A.96).



EVENT

Name: OUT_OF RANGE Synonym:
Defined in chart:EWS_AC
Description: The signal sample is out of predefined range
Definition:

Attribute Name:	Attribute Value:
PROJECT_ID	EV3

Long Description:

This Event is produced by the Activity COMPARE when the signal ...

Abbildung A.96: Ereignis-Formular

Bedingungen (Conditions)

Form der Informationselemente die Bedingungen übertragen. Sie werden mittels eines Kontrollflusses übertragen und können die Werte TRUE oder FALSE annehmen (sieheAbbildung A.97).

Data-Items

Es können weiterhin Records, 'one-of's, groups und lists übertragen werden ihre textuelle Beschreibung erfolgt im Data-Item Form (sieheAbbildung A.98).

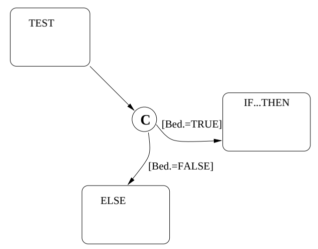


Abbildung A.97: Bedingungen

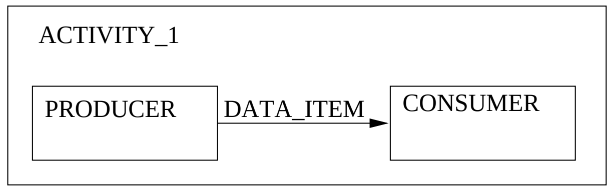


Abbildung A.98: Data-Items

Data-Item Form

Im Data Item Form wird der Typ des Data-Items bestimmt und es werden die textuellen Erklärungen zu dem Item aufgeschrieben (sieheAbbildung A.99).

Information-flows

Sie dienen der Bündelung einzelner Datenflüsse, um Pfeile in den Activity Charts zu minimieren (sieheAbbildung A.100).

Data-stores

Datenspeicher, in die Daten hineingeschrieben und später wieder ausgelesen werden können (sieheAbbildung A.101).

Data-stores Forms

Textuelle Beschreibung der Datenspeicher (sieheAbbildung A.102).

Forks and Joints

Teilen Informationen auf, wobei die gleiche Information an mehrere Empfänger geht, bzw. vereinigt zwei gleiche Informationen, die an einen Empfänger gehen (sieheAbbildung A.103).

Junction Connectors

Dienen dazu verschiedene Daten in einem Informationsfluß weiterzuleiten (sieheAbbildung A.104).

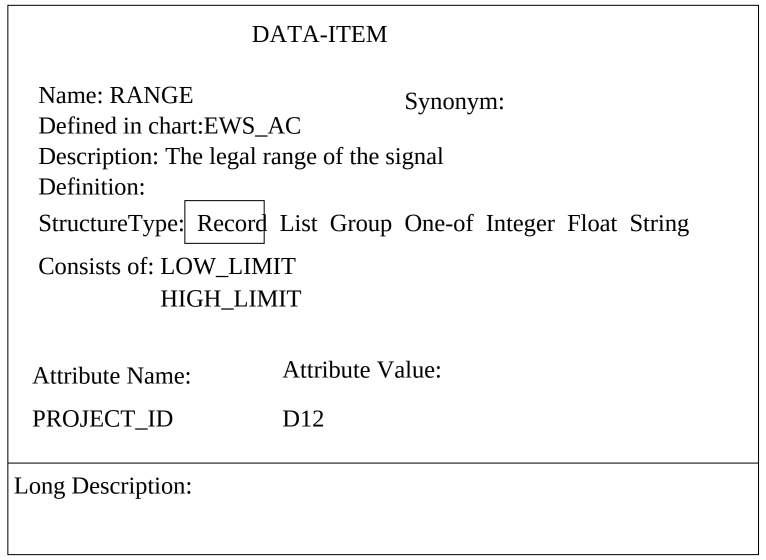


Abbildung A.99: Data-Item Form

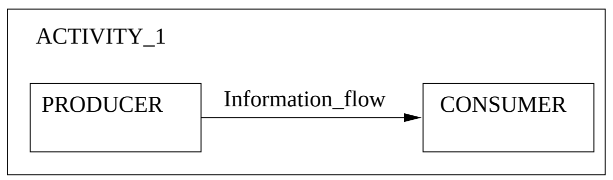


Abbildung A.100: Information-flows

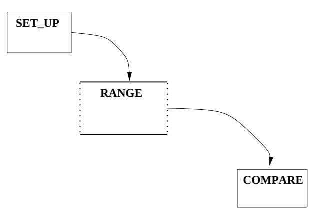
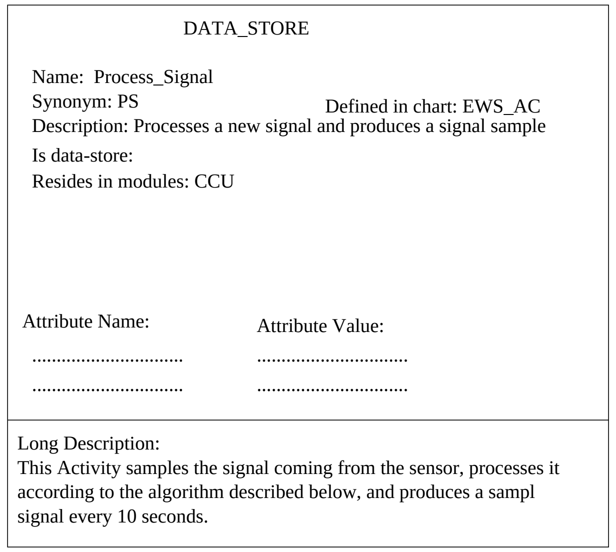


Abbildung A.101: Data-stores


DATA_STORE

Name: Process_Signal
Synonym: PS
Defined in chart: EWS_AC
Description: Processes a new signal and produces a signal sample
Is data-store:
Resides in modules: CCU

Attribute Name: Attribute Value:
....
....

Long Description:
This Activity samples the signal coming from the sensor, processes it according to the algorithm described below, and produces a sampl signal every 10 seconds.

Abbildung A.102: Data-stores Forms

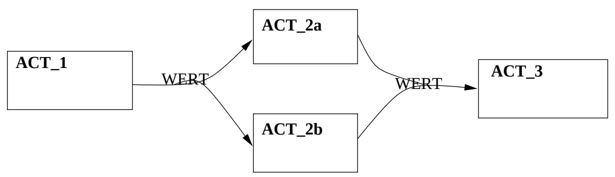


Abbildung A.103: Forks and Joints

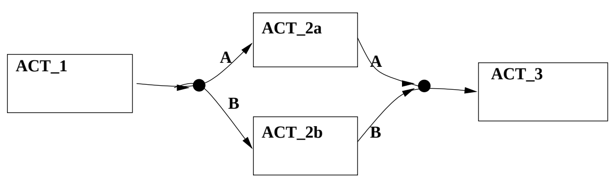


Abbildung A.104: Junction Connectors

Composition Connectors

Um einzelne Komponenten eines Records aufzuspalten und an unterschiedliche Empfänger weiterzuleiten (sieheAbbildung A.105).

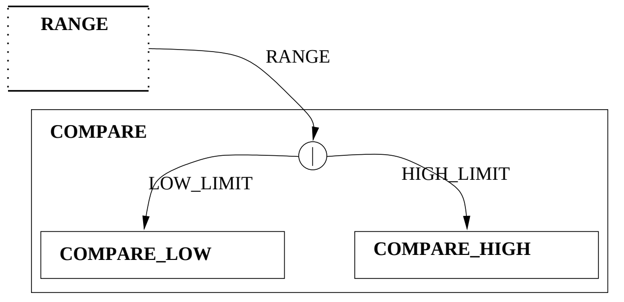


Abbildung A.105: Composition Connectors

Diagram Connectors

Abkürzung der Pfeile, ersparen zu lange Linien; sind mit eindeutigen Namen zu versehen (sieheAbbildung A.106).

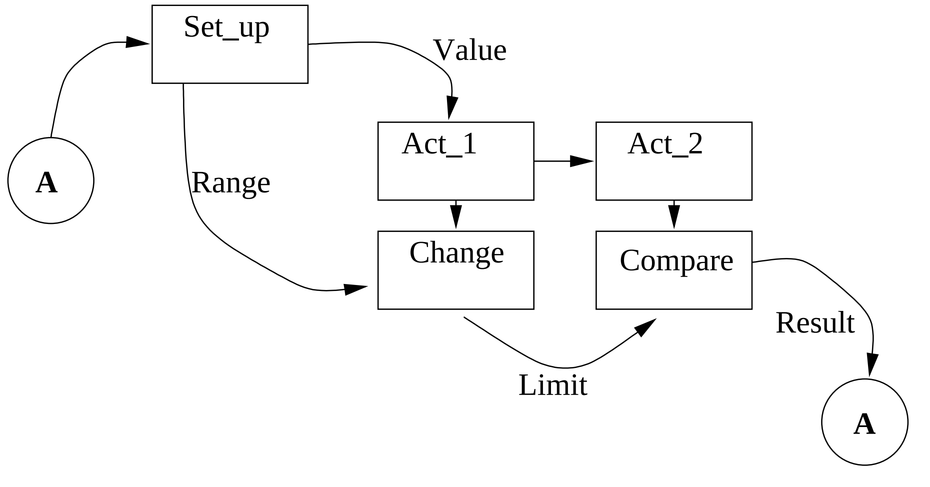


Abbildung A.106: Diagram Connectors

To-control Connectors

Siehe auch Diagram Connectors; aber für Controllflüsse (sieheAbbildung A.107).

A.2.4.2 Statecharts

States

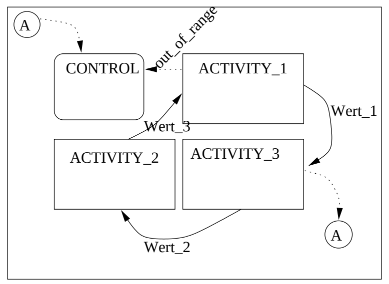


Abbildung A.107: To-control Connectors

Ein Zustand wird betitelt und als ein Rechteck mit abgerundeten Ecken dargestellt (sieheAbbildung A.108).

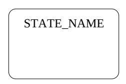


Abbildung A.108: States

State Form

Textuelle Beschreibung des Zustands (sieheAbbildung A.109).

AND-State

Transition

Überführung von einem Zustand in einen anderen mit einem Trigger, der mit Bedingungen verknüpft sein kann (sieheAbbildung A.111).

Joint C., Junction C. ...

Transitionen haben die gleichen Darstellungsmöglichkeiten wie Informationsflüsse (sieheAbbildung A.112).

Condition Connector

Ein einzelner Trigger kann je nach Wahrheitswert an verschiedene Empfänger versendet werden (sieheAbbildung A.113).

A.2.4.3 Module-charts

Modul-Forms

Textuelle Beschreibung der Module (sieheAbbildung A.114).

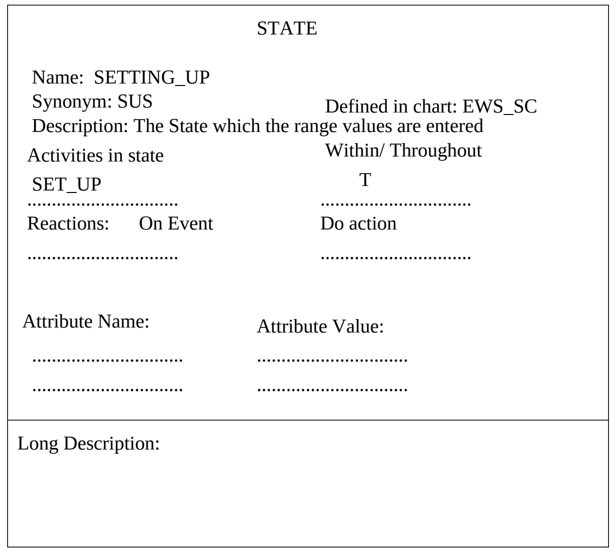


Abbildung A.109: State Form

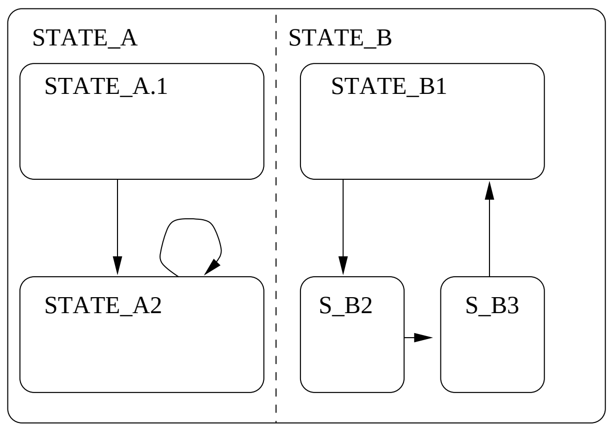


Abbildung A.110: AND-State

Abbildung A.111: Transition

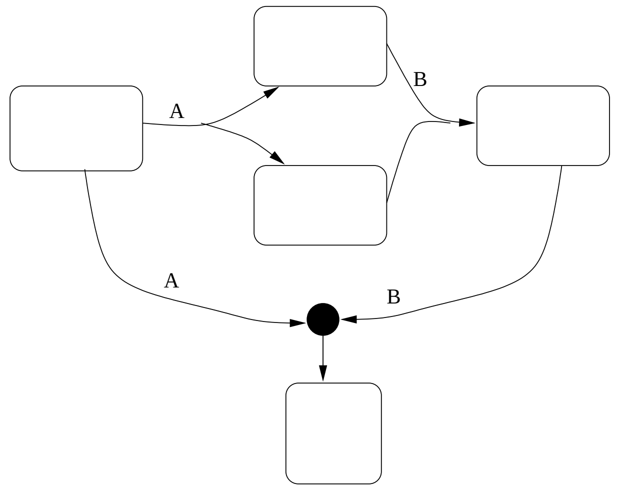


Abbildung A.112: Joint C., Junction C. ...

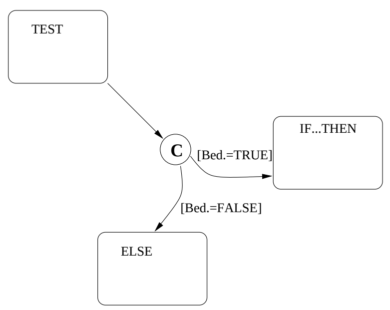
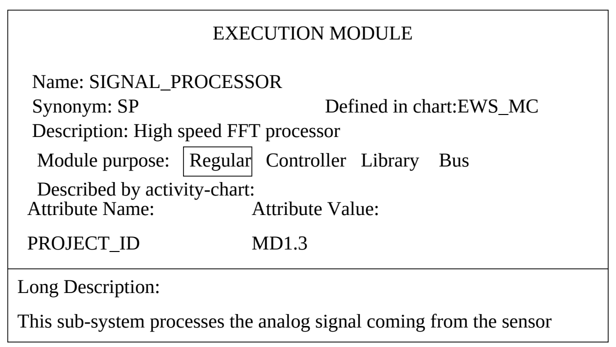


Abbildung A.113: Condition Connector



EXECUTION MODULE

Name: SIGNAL_PROCESSOR
Synonym: SP Defined in chart:EWS_MC
Description: High speed FFT processor
Module purpose: Regular Controller Library Bus
Described by activity-chart:

Attribute Name:	Attribute Value:
PROJECT_ID	MD1.3

Long Description:

This sub-system processes the analog signal coming from the sensor

Abbildung A.114: Modul-Forms

Communication Lines

Kommunikation zwischen den getrennten Modulen. Sehen aber genauso aus, wie normale Informationsflüsse.

A.2.5 UML

A.2.5.1 Warum wir modellieren

Die Entwicklung eines Modells vor der Implementierung ist genau so wichtig wie der Entwurf für ein Haus. Das Modell reduziert die Komplexität und ist für eine bessere Kommunikation zwischen den Mitarbeiter notwendig.

A.2.5.2 Unified Modeling Language (UML)

- UML ist eine Modellierungssprache zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen eines Software-Systems.

- UML vereint die Konzepte von Booch, OMT und OOSE.

- UML stellt ausreichende Semantik und Notation für ein breites Spektrum von Modellierungsproblemen in einer direkten und wirtschaftlichen Art zur Verfügung.

- UML stellt ferner eine ausreichende Semantik zur Verfügung, um bestimmte Modellierungsprobleme, die in Zukunft erwartet werden, lösen zu können. Insbesondere Probleme wie Component Technologie, Verteilte Systeme, Frameworks, Ausführbarkeit (eg. : executability).

- UML stellt Erweiterungsmechanismen für einzelne Projekte zur Verfügung, so daß das Metamodell für die Applikation mit niedrigen Kosten erweitert werden kann.

- UML stellt eine Semantik zur Verfügung, die den Wechsel von Modell zwischen verschiedenen Werkzeugen erleichtert.

A.2.5.3 Die Ziele von UML

1- Zur Verfügung stellen von einer für Benutzer Ready-To-Use, ausdrucksvollen, visuellen Modellierungssprache, so daß bedeutungsvolle Modelle entwickelt und ausgetauscht werden können.

2- Zur Verfügung stellen von Erweiterungs- und Spezialisierungsmechnismen um die Core-Konzept zu erweitern.

3- Zur Verfügung stellen von einer formalen Basis um die Modellierungssprache zu verstehen.

4- Unabhängig sein von bestimmen Programmiersprachen und Entwicklungsprozessen.

5- Ermutigung der Zunahme von OO-Tools auf dem Markt.

6- Unterstützung höherer Entwicklungskonzepte so wie Collaborations, Frameworks, Patterns und Components.

A.2.5.4 UML definiert folgende graphische Diagramme

- 1-Class Diagramm
- 2-Use Case Diagramm
- 3-Verhaltensdiagramme :
  - a. Statechart Diagramm
  - b. Activity Diagramm
  - c. Interaction Diagramme :
    - I. Sequence Diagramm
    - II. Collaboration Diagramm
- 4-Implementation Diagramme :
  - a. Component Diagramm
  - b. Deployment Diagramm

A.2.5.5 Die generellen Erweiterungsmechnismen in UML

1. Constraints und Kommentare :

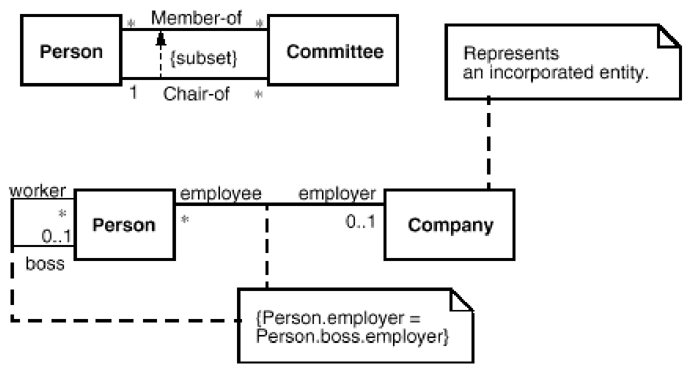


Abbildung A.115: Constraints und Kommentare

Die Constraints werden in geschweiften Klammern geschrieben "{ }". Wenn es nur ein graphisches Symbol gibt, wird das Constriant in die Nähe des Symboles geschrieben. Falls zwei graphische Symbole vorhanden sind, dann zeichnet man einen gestrichenen Pfeil von einem Element zum anderen, markiert durch den Constraint-String in der geschweiften Klammer (sieheAbbildung A.115).

Für drei oder mehrere graphische Symbole wird der Constriant-String in ein „Notiz-Symbol“ geschrieben und über eine gestrichelte Linie mit jedem Symbol verbunden.

{ author = "Joe Smith", deadline = 31-March-1997, status = analysis }

{ abstract }

Abbildung A.116:

2. Element-Eigenschaften :

Eigenschaft heißt: jeder Wert, der zu einem Element zugehört, einschließlich Attribute, Assoziationen und Tagged value. Um neue Eigenschaften für Elemente zu definieren kann man den Tagged value-Mechnismus anwenden. Ein Eigenschaft-Spezifikation hat die Form:

Schlüsselwort	Wert
Der Name der Eigenschaft	Ein willkürlicher String

3. Stereotypen :

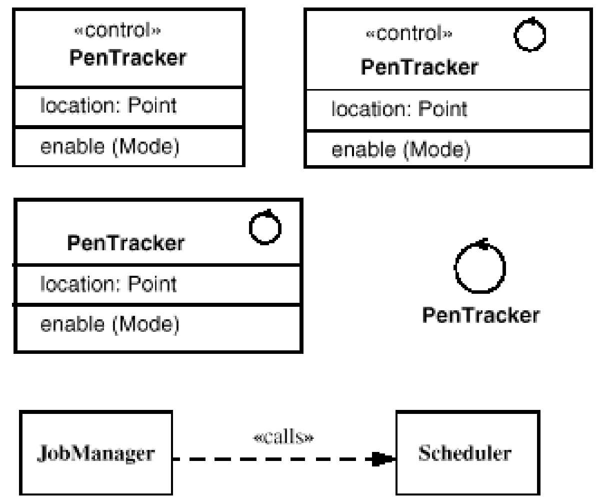


Abbildung A.117: Stereotypen

Sie repräsentieren eine Subklasse von einem existierenden Modellelement mit der selben Form (Attribute und Assoziationen) aber einer unterschiedlichen Absicht. Stereotypen repräsentieren eine Unterscheidung in der Anwendung. Sie werden durch guillemets „<< >>“ repräsentiert (sieheAbbildung A.117).

A.2.5.6 Statische Strukturdiagramme

- 1. Klassendiagramm
- 2. Objekt Diagramm
- 3. Classifier

- 4. Klasse
- 5. Attribut
- 6. Operation
- 7. Typ vs. Implementationsklasse
- 8. Interfaces
- 9. Parametesierte Klasse (Template)
- 10. Utility
- 11. Objekt
- 12. Assoziation
  - a. Or-Assoziation
  - b. Ordering
  - c. Navigabilität (navigability)
  - d. Multiplizität (multiplicity)
  - e. Rolename
  - f. Interface-Spezifizier
  - g. Änderbarkeit (changeability)
  - h. Sichtbarkeit (visibility)
- 13. Qualifier
- 14. Komposition
- 15. Links
- 16. Generalisierung
- 17. Abhängigkeit (dependency)
- 18. Abgeleitete Elemente

A.2.5.7 Die Unterschiede zum Objektmodell von OMT

1. Typ vs. Implementationsklasse

Klassen können durch Stereotypen in Typen und Implementationsklassen spezialisiert werden. Ein Typ charakterisiert eine änderbare Rolle, die ein Objekt annehmen und später abweisen kann. Ein Objekt kann mehrere Typen haben.

Eine Implementationsklasse definiert die physikalische Datenstruktur und Prozeduren eines Objektes in der zu implementierenden Sprache. Ein Objekt kann nur eine Implementationsklasse haben.

Die Implementation eines Typs durch eine Implementationsklasse wird durch eine gestrichelte Linie mit einem dreieckigen Pfeilkopf dargestellt (sieheAbbildung A.118).

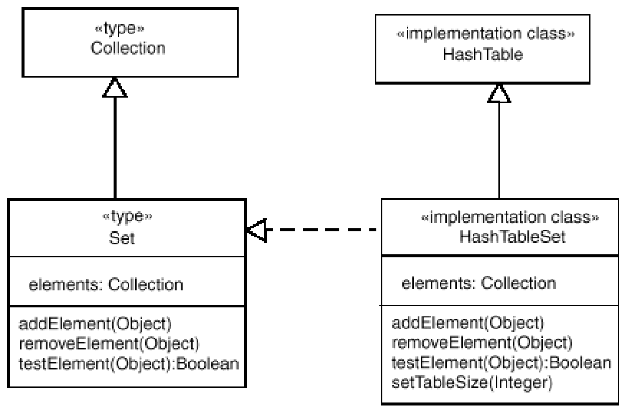


Abbildung A.118: Typ vs. Implementationsklasse

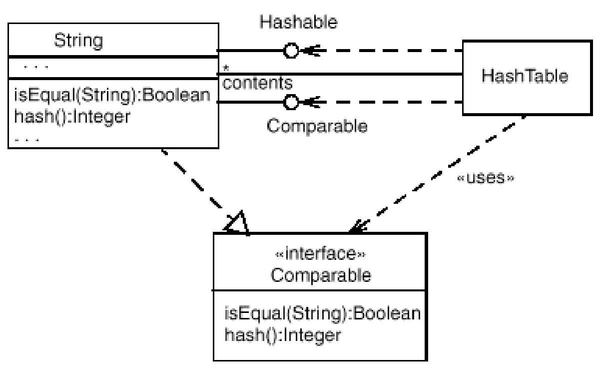


Abbildung A.119: Interface

2. Interface

Ein Spezifizier für extern sichtbare Operationen einer Klasse, Komponente, oder andere Entity ohne Spezifikation der internen Struktur. Interfaces haben nur Operationen. Sie können Generalisierungsbeziehungen haben (sieheAbbildung A.119).

3. Parametersierte Klasse (Template)

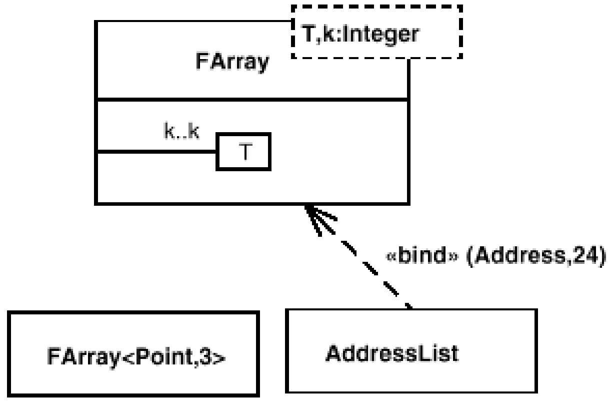


Abbildung A.120: Parametersierte Klassen

Ein Template ist der Deskriptor für eine Klasse mit einem oder mehreren ungebundenen formalen Parametern. Ein Template ist keine direkt brauchbare Klasse, da es ungebundene Parameter besitzt.

Ein gebundenes Element wird so gezeigt:

Template-Name $<$ Werteliste $>$

das gebundene Element ist von seinem Template ganz spezifiziert, deshalb kann sein Inhalt nicht erweitert werden (sieheAbbildung A.120).

4. Utility

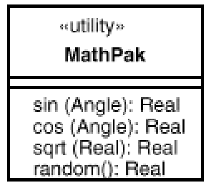


Abbildung A.121:

Utility ist eine Gruppierung von Variablen und Prozeduren in Form von einer Klassendeklaration und ist wie ein Stereotyp von einer Klasse modelliert.

5. Assoziation

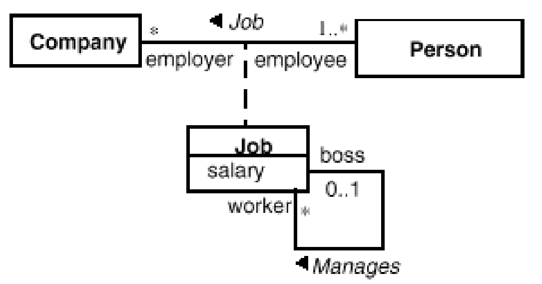


Abbildung A.122: Assoziation

a. OR-Assoziation {or}

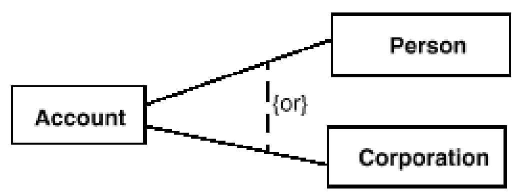


Abbildung A.123: OR Assoziation

Eine OR-Assoziation zeigt eine Situation, in der nur eine von mehreren Assoziationen in einer bestimmten Zeit für jedes einzelne Objekt instanziert werden kann (siehe Abbildung A.123).

b. Navigabilität

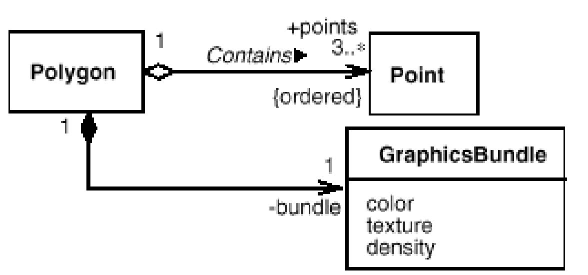


Abbildung A.124: Navigabilität

Navigabilität zeigt, daß die Navigation zu der angerichtete Klasse unterstützt wird (siehe Abbildung A.124).

c. Interface-Spezifizier (':' classifiername)

Beispiel: siehe oben

Sie spezifiziert das benötigte Verhalten, um die Assoziation zu ermöglichen. Falls der Interface-Spezifier ausgelassen wird, dann hat die Assoziation den vollen Zugriff zu der assozierten Klasse.

d. Änderbarkeit

Beispiel: siehe oben

Wenn die Links änderbar sind, d.h. Hinzufügen, Löschen, Bewegen möglich ist, dann braucht man keinen Indikator. {frozen} bedeutet, daß keine änderung möglich ist. {addOnly} bedeutet, daß zusätzliche Links hinzugefügt werden können, aber kein Link darf modifiziert oder gelöscht werden.

e. Sichtbarkeit (+, #, -)

Beispiel: siehe oben

Dies spezifiziert die Sichtbarkeit der Assoziation in der Richtung zu den gegebenen Rollenamen.

Komposition

Sie ist eine Form von Aggregation mit starkem Besitz und zufälliger Lebenszeit von der Teil zu dem Ganz.

Eine Assoziation in einer Komposition bedeutet, daß jede Tuple von Objekten, die durch einer Link verbunden sind, müssen alle zu der selben Container-Objekt gehören (siehe Abbildung A.125).

Abhängigkeit (dependency)

Abhängigkeit (siehe Abbildung A.126) zeigt eine semanitische Beziehung zwischen zwei oder mehrere Modellelemente. Die vordefinierte Abhängigkeiten sind:

trace, refine, uses, bind

Abgeleitete Elemente

Ein abgeleitetes Element wird von einem anderen berechnet. Dies wird nur gezeigt, damit alles klar und genau ist oder vielleicht ist es im Design berücksichtigt worden, obwohl es keine semantische Informationen hin zufügt (siehe Abbildung A.127).

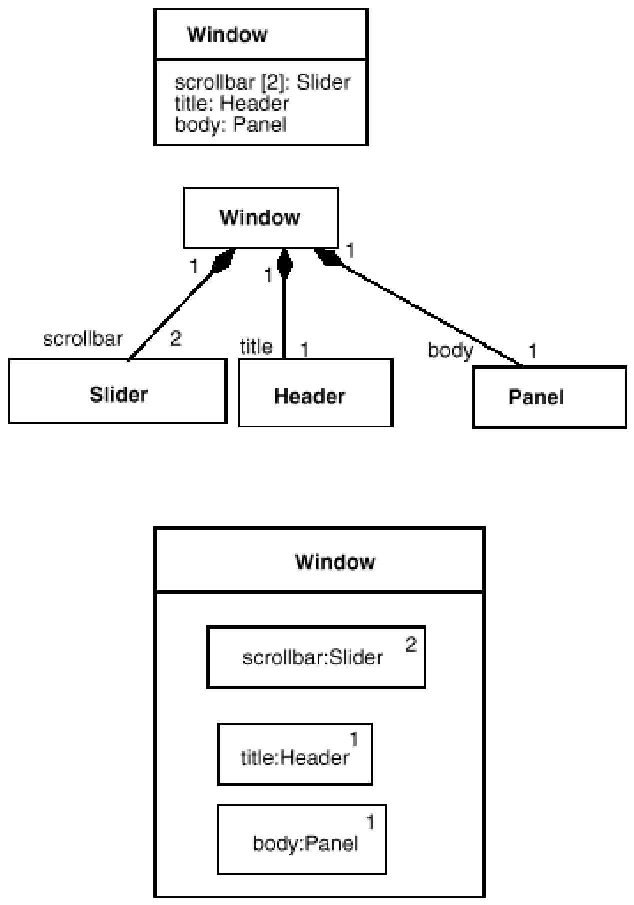


Abbildung A.125: Komposition

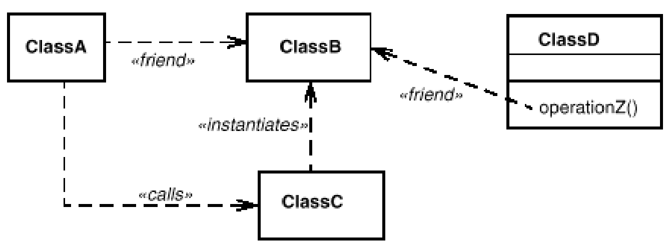


Abbildung A.126: Abhängigkeit

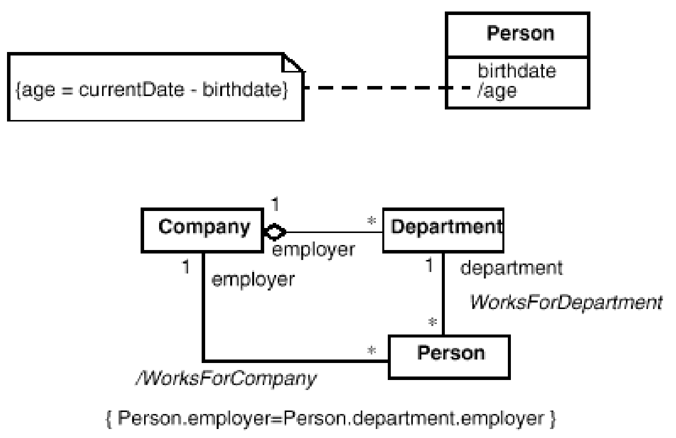


Abbildung A.127: Abgeleitete Elemente

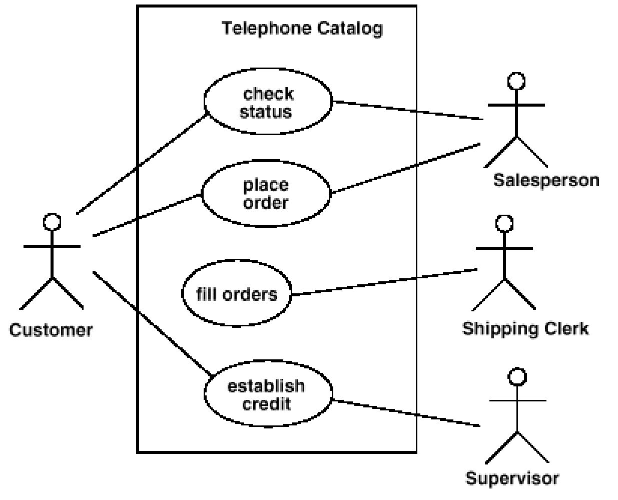


Abbildung A.128: Use Case Diagram

A.2.5.8 Use Case Diagram

Dieses Diagramm zeigt die Beziehung unter 'Actors' und 'Use Cases' innerhalb eines Systems (siehe Abbildung A.128).

Das Use Case-Modell repräsentiert die Funktionalität eines Systems.

Use Case

Use Case ist eine zusammenhängende Einheit von Funktionen, die von einem System oder einer Klasse zur Verfugung gestellt werden. Durch Sequenzen von Nachrichten, die austauschbar im System sind, können eine oder mehrere in außen stehenden Interactoren (sog. 'Actors') untereinander Aktionen durchführen.

Actor

Actor ist eine Rolle von einem Objekt oder mehreren Objekten außerhalb des Systems, daß mit dem System direkt als Teil von einer zusammenhängenden Arbeitseinheit (Use Case) interagiert. Er kann als eine Klasse mit dem Strereotyp <<actor>> gezeigt werden. Der Standard Stereotyp-Icon für einen Actor ist die 'Stickman' Figur.

Use Case-Beziehungen

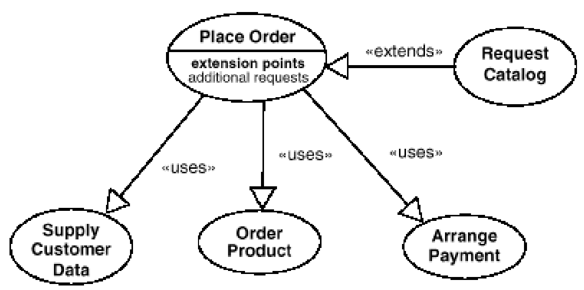


Abbildung A.129: Use Case-Beziehungen

Es gibt mehrere Standardbeziehungen unter Use Cases oder zwischen Actors und Use Cases (Abbildung A.129):

Communicates: Die Beteiligung eines Actors in einer Use Case. Das ist die einzige Beziehung zwischen den Actors und Use Cases.

Extends: Eine 'extends' Beziehung von Use Case A zu Use Case B zeigt, daß ein Instanz von Use Case B nimmt das Verhalten spezifiziert durch A auf.

Uses: Eine 'uses' Beziehung von Use Case A zu Use Case B zeigt, daß ein Instanz von Use Case A möchte auch das Verhalten spezifiziert durch B aufnehmen.

A.2.5.9 Interaktionsdiagramm

Interaktionsdiagramme kommen in zwei Formen vor :

- Das Sequenz Diagramm
- Das Kollaborationsdiagramm

Sequenz Diagramme zeigen die explizite Sequenzen von Nachrichten (Messages) und sind für Real-Time Sepzifikationen und komplexe Szenarien geeigneter.

Kollaborationsdiagramme zeigen die Beziehung zwischen Objekten und sind für Verständnis aller Effekten von einem gegebenen Objekt und für prozeduralen Entwurf geeigneter.

Sequenz Diagramm:

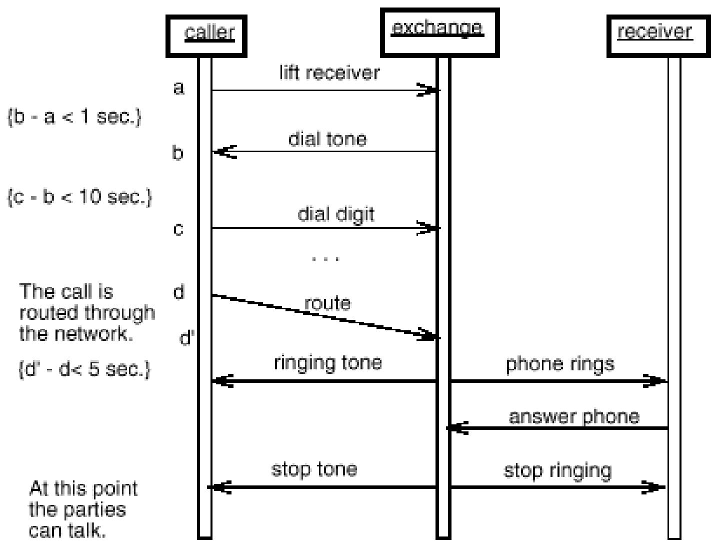


Abbildung A.130: Sequenz Diagramm

Ein Sequenz Diagramm zeigt eine Interaktion, die in Zeitsequenzen angeordnet ist. Insbesondere wird es durch Lebenslinie, die beteiligte Objekte in der Interaktion und die Nachrichten, die in einer angeordneten Zeitsequenzen ausgetauscht werden, gezeigt (siehe Abbildung A.130).

Ein Sequenz Diagramm hat zwei Dimensionen:

Eine vertikale Dimension zeigt die Zeit und eine horizontale Dimension präsentiert die verschiedenen Objekte. Die horizontale Ordnung von Objekten ist nicht signifikant.

Die Objekte eines Diagramms können zu SSwimlanes" gruppiert werden.

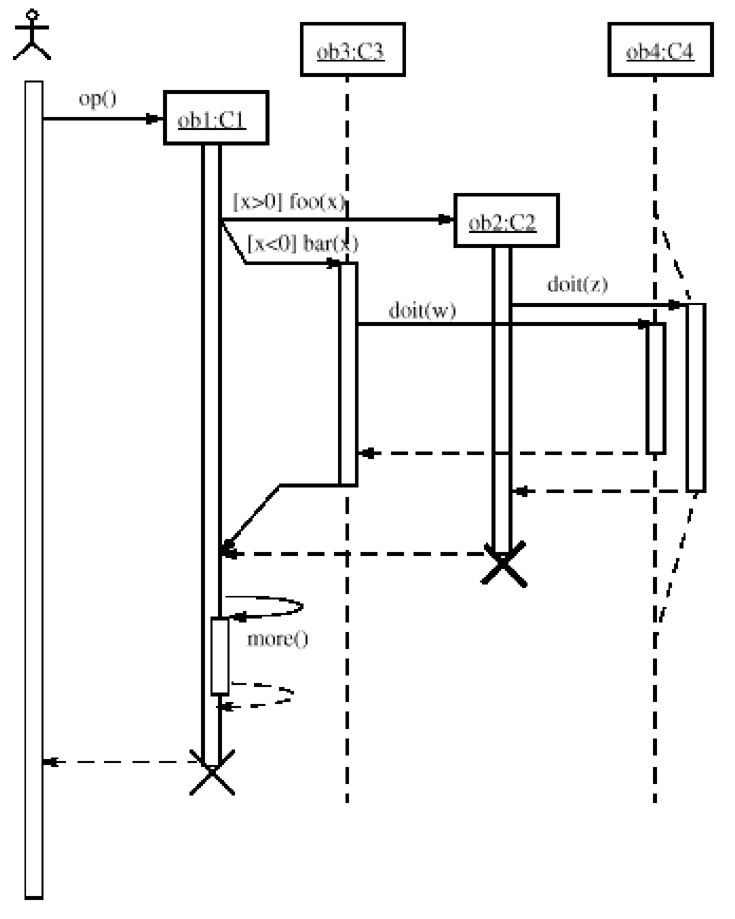


Abbildung A.131: Lebenslinien Diagramm

Das Objekt „Lebenslinie“

Eine Rolle beschreibt die Typ eines Objektes und möglicherweise wird es auch die Rolle übernehmen und seine Beziehungen zu anderen Rollen beschreiben (siehe Abbildung A.131).

Innerhalb eines Sequenz Diagramms ist die Existenz und die Dauer eines Objekts in einer Rolle dargestellt, aber die Beziehungen zwischen die Rollen ist nicht gezeigt.

Ein Objekt Rolle ist durch eine vertikal gestrichene Linie präsentiert: Lebenslinie". Die Lebenslinie zeigt die Existenz eines Objektes in einer bestimmten Zeit.

Ein Objektsymbol wird am Kopf der Lebenslinie gezeichnet. Falls ein Objekt im Laufe des Diagramms gebildet wird, dann wird die entsprechende Message mit seinem Kopffpeil zu dem Objektsymbol gerichtet. Falls nun das Objekt im Laufe des Diagramms zerstört wird, wird dann seine Zerstörung durch ein großes X markiert.

Falls ein Objekt während der Transaktion gestartet wird, wird es über das erste Pfeil gezeichnet, und genau so, wenn ein Objekt während der Transaktion beendet wird, hat es seine Lebenslinie über das letzte Pfeil.

Eine Lebenslinie kann in zwei oder mehrere nebenläufigen Lebenslinien geteilt werden, um eine Bedingung anzudeuten.

Aktivation (Activation)

Beispiel: siehe oben

Eine Aktivation ist eine Zeitperiode, in der ein Objekt eine Aktion entweder direkt oder durch eine untergeordneten Prozedur durchführt. Sie präsentiert sowohl die Dauer einer Aktion als auch die Kontrollrelation zwischen die Aktivation und ihren Anrufer (Stack frame). Eine Aktivation wird durch ein lange dünne Rechteck dargestellt. Die zu durchführende Aktionen können neben das Aktivation Symbol in Text geschrieben werden. Im Falle von nebenläufigen Objekten mit eigenem Kontrollfaden wird eine Aktivation die Dauer einer durchgeführten Operation von jedem Objekt zeigen.

Im Falle von prozeduralen Code wird eine Aktivation die Dauer einer aktiven Prozedur im Objekt oder einer untergeordneten aktiven Prozedur in anderen möglichen Objekten zeigen. Im Falle von rekursivem Aufruf eines Objektes mit einer existierenden Aktivation wird das zweite Aktivation Symbol rechts neben dem ersten Symbol gezeichnet werden.

Nachrichten (Messages)

Beispiel: siehe oben

Eine Nachricht ist eine Kommunikation zwischen Objekten, die Informationen vermitteln, mit der Erwartung, daß eine Aktion folgen wird. Das Empfangen einer Nachricht ist eine Art vom Ereignis (Event). Eine Nachricht wird durch einen horizontal durchgestrichenen Pfeil von der Lebenslinie eines Objektes zu der Lebenslinie eines anderen Objektes dargestellt. Der Pfeil ist mit dem Namen der Nachricht (Operation oder Signal) und seiner Wert markiert. Eine Nachricht kann auch mit einer Bedingung markiert sein.

Kollaboration Diagramm

Im Gegenteil zu dem Sequenz Diagramm zeigt eine Kollaboration Diagramm die Beziehung zwischen den Objekt Rollen. Ein Kollaboration Diagramm wird eben nicht die Zeit als eine separate Dimension darstellen, so daß die Nachrichtensequenzen und die nebenläufig Faden (Threads) aus einander gehalten werden müssen. Das geschieht durch die Numerierung der Sequenzen.

Kollaboration (Collaboration)

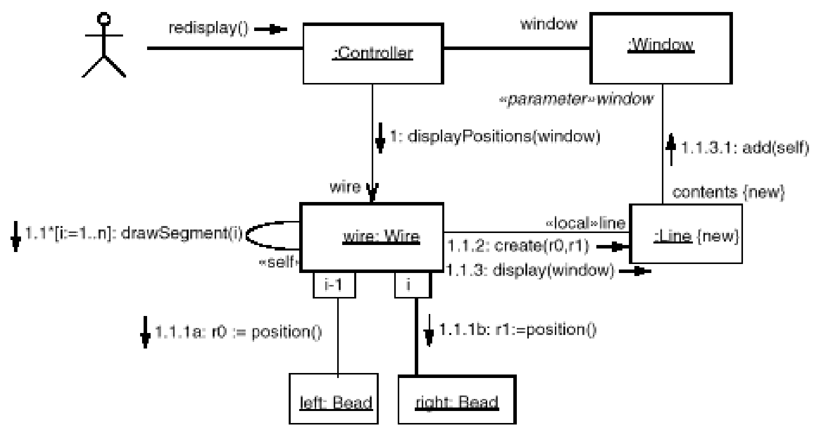


Abbildung A.132: Kollaboration

Eine Kollaboration ist eine Menge von Beteiligten und Beziehungen, die für eine gegebene Menge von Zielen bedeutungsvoll sind. Eine Kollaboration kann zu einer Operation oder Use Case angehängt werden, um das Kontext, in dem ihr Verhalten ereignet hat zu beschreiben (siehe Abbildung A.132).

Eine parametersierte Kollaboration stellt eine Entwurfskonstuktion dar, die in verschiedene Entwurfe wiederholt benutzt werden kann. Die Beschreibung von Verhalten bringt zwei Aspekte mit sich: Die strukturelle Beschreibung von seinem Beteiligten und die Verhaltensbeischreibung von seiner Ausführung. Beide Aspekte werden öfters zusammen in einem einzelnen Diagramm dargestellt, aber manchmal ist es sinnvoller die strukturelle Aspekte und die Verhaltensaspekte aus einander beschrieben werden. Der Struktur von Objekten nehmen Rollen in einem Verhalten an und ihre Beziehungen werden Kollaboration genannt. Das dynamische Verhalten von Nachrichtensequenzen, das zwischen Objekten ausgetauscht werden, wird eine Interaktion (Interaction) genannt. In einem Kollaboration Diagramm werden die Objekte, die während der Ausführung gebildet werden, mit {new} gekennzeichnet und die, die während die Ausführung zerstört werden, mit {destroyed} gekennzeichnet und die, die während der Ausführung gebildet und dann zerstört werde, mit {transient} gekennzeichnet. Da die Kollaboration Diagramme öfters für Prozedurentwurf benutzt werden, zeigen die Pfeile dir Richtung der Navigation der Links.

Die interne Nachrichten, die eine Methode implementieren, werden durch Nummer beginnend mit 1 numeriert.

Entwurfsstruktur

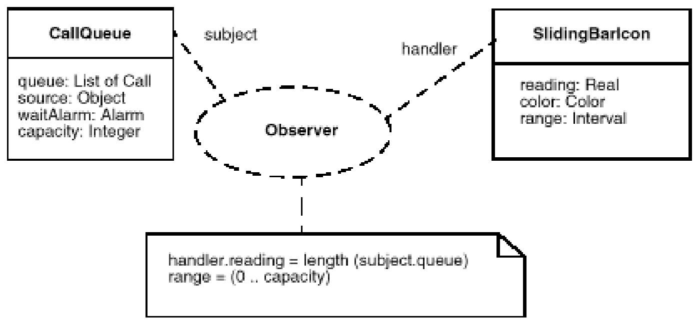


Abbildung A.133: Entwurfsstruktur

Eine Kollaboration kann zur Spezifikation der Implementierung von Entwurfsstrukturen (Abbildung A.133) benutzt werden. Für diesen Zweck ist es notwendig ihr Kontext und ihr Interaktionen spezifiziert werden. Es ist auch möglich eine Kollaboration als eine einzelne Entity vom Äußen" gesichtet wird. Dies kann z.B. zur Erkennung von Entwurfmustern in einem Systementwurf benutzt werden. Ein Muster ist eine parametersierte Kollaboration. In jeder Anwendung von Mustern werden die aktuelle Klassen durch die Parameter in Entwurfsdefinition ersetzt. UML beschreibt die strukturelle Aspekte und einige Verhaltensaspekte von Entwurfsmuster. Die Anwendung einer Kollaboration wird durch eine gestrichene Ellipse, die der Name von Kollaboration enthält, dargestellt. Eine gestrichene Linie wird vom Kollaborationssymbol zu jeder beteiligten Objekte oder Klassen gezeichnet. Jede Linie ist von Rolle der Beteiligten gekennzeichnet.

Interaktionen (Interactions)

Beispiel: siehe oben

Eine Interaktion ist eine Verhaltensspezifikation, die eine Sequenz von zwischen einer Menge von Objekten austauschenden Nachrichten umfasst, um eine Spezifikationszweck zu durchführen, z.B. Implementation von einer Operation. Um eine Interaktion zu spezifizieren, muß man zu erst eine Kollaboration spezifizieren, und das geschieht durch Zusammenstellen von der interagierenden Objekten und ihre Beziehungen. Hierdurch sind dann die Interaktionssequenzen spezifiziert. Interaktionen werden in Sequenz Diagramme oder in Kollaboration

Diagramme dargestellt. Beide Diagramme zeigen die Ausführung von Kollaborationen, jedoch zeigen die Sequenz Diagramme nur die beteiligte Objekte und nicht ihre Beziehungen zu anderen Objekten oder ihre Attribute. Deswegen zeigen sie nicht die ganze Kontextaspekte von einer Kollaboration.

*Sequenz Diagramme können explizit die Verhaltensaspekte von Kollaborationen zeigen, und zwar durch Hinzufügen von Nachrichten-Zeitsequenzen und explizite Darstellung von Akivationsmethoden.

*Kollaboration Diagramme zeigen die ganze Kontext einer Interaktion, inbegriffen die Objekte und ihre Beziehungen, die für eine bestimmte Interaktion relevant sind, und so sind sie für Entwurfszwecke geeigneter.

Kollaborationsrolle

Beispiel: siehe oben

Eine Rolle beschreibt in einer Kollaboration die Typ eines Objektes, die möglicherweise die Rolle annimmt und ihre Beziehung zu anderen Rollen beschreibt. Eine Kollaborationsrolle wird wie die Notation von Objekten dargestellt.

Multiobjekt

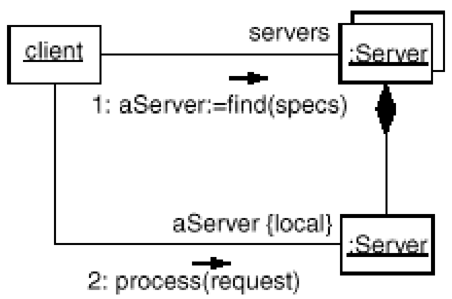


Abbildung A.134: Multiobjekt

Ein Multiobjekt präsentiert eine Menge von Objekten auf der 'many'-Ende einer Assoziation. Dies wird dazu benutzt, Operationen zu zeigen, die die ganze Menge anstatt ein einzelnes Objekt adressieren. Das unterlegte statische Modell bleibt bei dieser Gruppierung unverändert. Um eine Operation auf jedem Objekt in einer Menge von assoziierten Objekten durchzuführen, braucht man zwei Nachrichten (Messages): Eine Nchricht für die Iteration zum Multiobjekt mit ausgehenden Links zu jeder einzelnen Objekten, und dann eine Nachricht, die an jeder einzelnen Objekten durch eine (temporelle) Link gesendet wird. Dies kann in einem Diagramm sinnvoller durch die Kombination der Nachrichten in einer einzigen Nachricht, die eine Iteration und eine Anwendung zu jedem individuellen Objekt enthält, darstellen. Die Ziel-Rollname bekommt dann eine many-Indikator (*) um viele angeschlossene Links zu zeigen (siehe Abbildung A.134).

Aktivobjekt

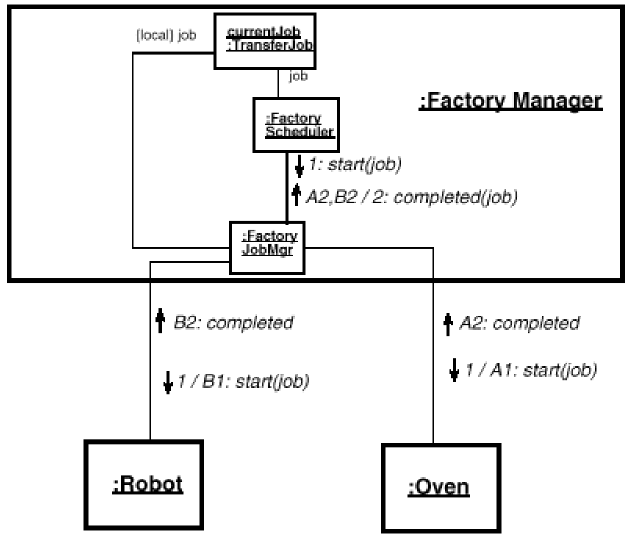


Abbildung A.135: Aktivobjekt

Ein Aktivobjekt hat einen eigenen Kontrollpfad und möglicherweise initialisiert die Kontrollaktivity. Ein passives Objekt behält die Daten, aber initialisiert keine Kontrollaktivity. Jedoch kann ein passives Objekt die erhaltene Nachrichten über die Bearbeitung einer Abfrage im Prozeß senden. Eine Rolle für ein Aktivobjekt wird wie ein Objektsymbol mit dicken Kanten dargestellt. Ein Eigenschaftsschlüsselwort {active} kann zur Darstellung von einem Aktivobjekt benutzt werden (siehe Abbildung A.135).

Nachrichtenflüsse (Message flows)

2: display (x, y)	simple message
1.3.1: p:= find(specs)	nested call with return value
[x < 0] 4: invert (x, color)	conditional message
A3,B4/ C3.1*: update ()	synchronization with other threads, iteration

Abbildung A.136: Nachrichtenfluß

Ein Nachrichtenfluß (Abbildung A.136) ist die Sendung einer Nachricht von einem Objekt zum anderen. Die Implementation einer Nachricht kann verschiedene Formen haben, z.B. ein Prozeduraufruf, Senden von einem Signal zwischen aktiven Pfaden (Threads), explizites Errichten von Events, usw.

Ein Nachrichtenfluß wird durch ein gekennzeichnetes Pfeil gezeigt. Es gibt verschiedene Kontrollflußtypen:

(Ein Prozeduraufruf oder andere verschachtelte Kontrollflusse. Die ganze verschachtelte Sequenz muß fertig sein, bevor die äußere Ebene der Sequenz fortgesetzt wird. Die möglich Anwendung ist in gewöhnliche Prozeduraufrufe oder in nebenläufige Aktivobjekten.

(Flache Kontrollfluß. Jeder Pfeil zeigt die Entwicklung der Sequenz zum nächsten Schritt. Normalerweise sind alle Nachrichten asynchron.

Asynchrone Kontrollfluß. Zum expliziten Anschaulichen von asynchrone Nachrichten zwischen zwei Objekte in einer Prozedur.

Nachrichtenmakierer (Message label)

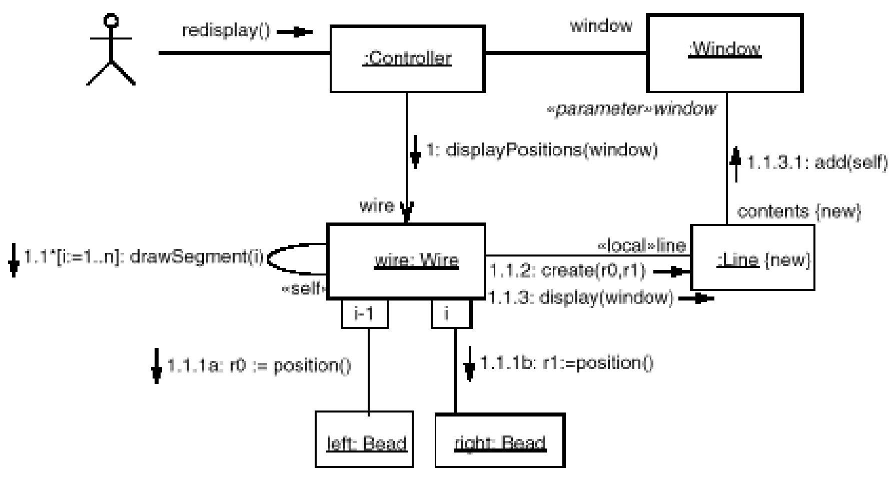


Abbildung A.137: Nachrichtenmakierer

Ein Markierer (Abbildung A.137) hat folgende Syntax:

predecessor guard-condition sequence-expression return value := message-name argument-list

*Predecessor (Vorgänger) : Ein Vorgänger ist eine durch Kommas getrennte Liste der Sequenz-nummern gefolgt von einem Schrägstrich ('/'). Das bedeutet, daß die Nachrichtenfluß nicht ermöglich wird bis alle Nachrichtenflusse, deren Sequenznummer aufgelistet sind, stattgefunden haben. Deshalb stellt Guard-condition eine synchronisation der Pfaden (Threads) dar.

*Sequenz-expression (Sequenzausdruck): Eine Sequenzausdruck ist eine durch Punkte getrennte Liste von Sequenztermen gefolgt von ein Kolon (':'). Jeder Term stellt eine Ebene der prozeduralen Verschachtlung in der gesamten Interaktion. Jeder Sequenzterm hat folgende Syntax :

[integer — name][recurrence]

Sequentielle Ordnung bedingte oder iterative Ausführung

*Signature (Signatur): Ein Signatur ist ein String, daß der Name, die Argumente und die Rückgabewerte einer Operation, Nachricht oder eines Signals anzeigt.

A.2.5.10 Statechart Diagramm

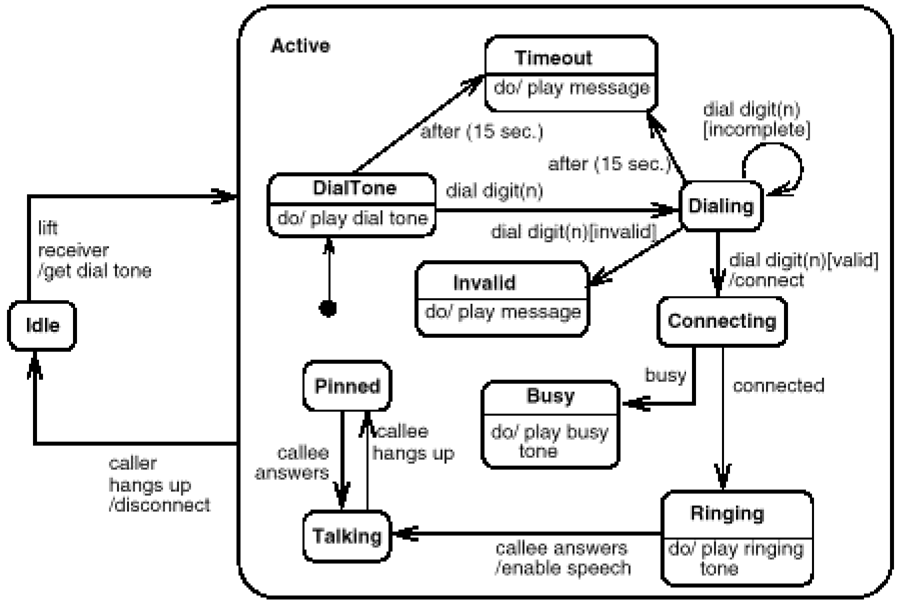


Abbildung A.138: Statechart Diagramm

Ein Statechart Diagramm (Abbildung A.138) stellt eine Zustandsmaschine dar. Ein Zustandsmaschine (Statemachine) ist ein Graph von Zuständen und Transitionen, daß die Antwort eines Objektes auf Reaktion von Empfang äußerer Stimulation beschreibt.

Zustände

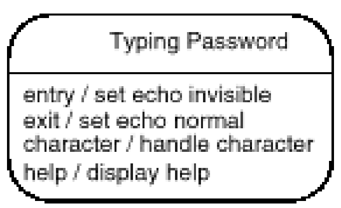


Abbildung A.139: Zustände

Ein Zustand ist eine Bedingung in Lebenszeit eines Objektes oder ist eine Interaktion, in der Bedingungen erfüllt, Aktionen ausführt oder auf Ereignisse wartet. Ein Zustand wird durch ein gerundetes Rechteck gezeigt und hat eine oder mehrere Aufteilungen. Das Namenabteil: enthält den optionalen Name von State. Das interne Transitionsabteil: enthält die interne Aktionen oder Activities, die in Antwort von angekommenen Ereignissen (Events) ausführt (siehe Abbildung A.139).

Event-name Argument-list '[' guard-condition ']' '/' action-expression

spezielle reservierte Schlüsselwörter:

'entry' '/' action expression

'exit' '/' action expression

Das folgende Schlüsselwort stellt der Aufruf einer verschachtelten Zustansmaschine:

'do' '/' machine-name (argument list)

Ereignisse (Events)

Beispiel: siehe oben

Ereignisse haben verschiedene Formen:

*Eine bestimmte Bedingung wird wahr. Das ist ein ChangeEvent. Dies wird durch das Schlüsselwort when gefolgt von einem boolischen Ausdruck in Klammer notiert. Wenn die Wert der Ausdruck sich von False auf True ändert, dann passiert das Ereignis. Beispiel: when (blabce ¡ 0)

*Empfang eines expliziten Signal von einem Objekt zu dem anderen. Das ist ein SignalEvent.

*Empfang eines Aufrufes für eine Operation durch ein Objekt. Das ist ein CallEvent.

*Der übergang von einem bestimmten Periode nach einem Ereignis oder das Stattfinden von einer gegebenen Zeit oder einem Datum. Das ist ein TimeEvent. Dies wird durch das Schlüsselwort after gefolgt von einer bestimmten Zeit in Klammer notiert. Beispiel: after (10 seconds).

Einfache Transitionen

right-mouse-down (location) [location in window] / object := pick-object (location)
^ object.highlight ()

Abbildung A.140: Transitionen (einfach)

Eine einfache Transition (Abbildung A.140) ist eine Beziehung zwischen zwei Zustände zur Andeutung, daß ein Objekt im ersten Zustand im zweiten Zustand eintreten und bestimmte spezifizierte Aktionen durchführen möchte, wenn ein Ereignis stattfindet.

Transition String:

Ein boolischer Ausdruck		Ein prozeduraleAusdruck	event-signature '[' guard-condition ']' '/' action-expression'̂' send-claus
beschreibt ein Event mit seinen Argumenten	Ein Spezialfall von einer Aktion mit dem	format: destination-expression'.'	destination-massage-name '(' argument'.' ... ')

Ein boolischer Ausdruck Ein prozeduraleAusdruck

event-signature '[' guard-condition ']' '/' action-expression '̂' send-clause

beschreibt ein Event Ein Spezialfall von einer Aktion mit dem

mit seinen Argumenten format: destination-expression'.' destination-

massage-name '(' argument'.' ... ')

Komplexe Transitionen

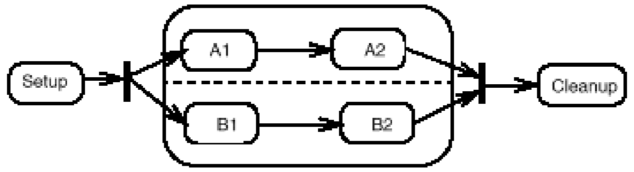


Abbildung A.141: Transitionen (komplex)

Ein komplexe Transition kann mehrere Quellzustände und Zielzustände haben. Sie präsentiert eine Synchronisation und/oder eine Spaltung von Kontrolle in nebenläufige Faden (Threads) ohne nebenläufige Unterzustände. Ein komplexe Transition ist dann möglich wenn alle Quellzustände besetzt sind. Danach werden alle Zielzustände durch die komplexe Transition besetzt. Sie wird durch eine dicke und senkrechte Stab dargestellt (Synchronization bar) (siehe Abbildung A.141).

Transition zu verschachtelten Zuständen

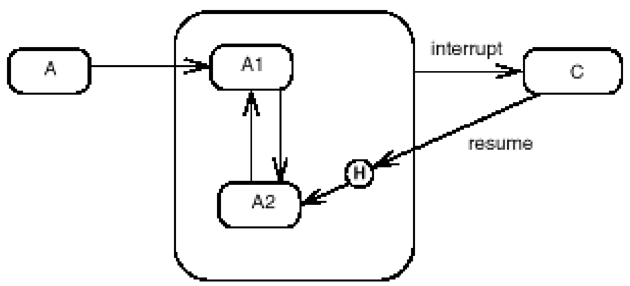


Abbildung A.142: History-Indicator

Eine Transition von einem komplexen Zustand kann zu allen Zustände innerhalb der Zustandsregion getragen werden. Sie wird von verschachtelten Zustände geerbt. Eine Zustandsregion kann einer History-state-indicator beinhalten, daß durch eine kleine Kreis mit dem Buchstabe 'H' dargestellt wird. Ein Historie-Indicator (siehe Abbildung A.142) kann beliebige Anzahl von ankommende Transitionen von außerhalb der Zustände haben. Aber sie kann höchstens eine nichtgekennzeichnet ausgehende Transition haben. Dies bezeichnet die Default previous state", falls die Region nie betretten wurde. Der Historie-Indicator kann auch für tiefe Historie (H*) sein.

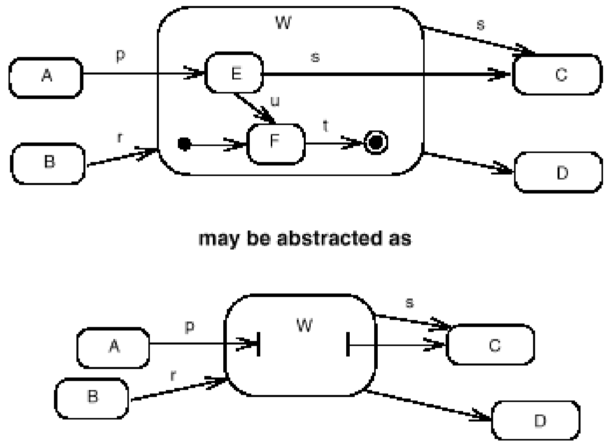


Abbildung A.143: Stubbed Transition

Stubbed Transition (Abbildung A.143): Wenn die verschachtelte Zustände von einem nicht gekennzeichneten Endzustand kommen oder zu einem nicht gekennzeichneten Anfangszustand gehen, können sie dann unterdrückt werden.

Eine interne Transition bleibt in einem einzelnen Zustand. Sie zeigt ein Ereignis, das keine änderung im Zustand macht. Sie wird durch die Schlüsselwörter 'entry' und 'exit' gezeigt.

A.2.5.11 Activity Diagramm

Ein Activity Modell ist eine Variation von einer Zustandsmaschine, in der die Zustände Activities sind. Activities präsentieren die Ausführung von Operationen. Transitionen werden durch die Vervollständigung von Operationen ausgelöst. Der Zweck dieses Diagramms ist die Konzentration auf Flüsse, die durch interne Prozesse abgeleitet werden. Wenn alle oder die meiste Ereignisse die Vervollständigung von intern erzeugen Aktionen präsentieren, soll dann das Activity Diagramm benutzt werden. (Das ist die prozedurale Kontrollaufruf). Wenn die asynchrone Ereignisse stattfinden, soll dann State Diagramm benutzt werden (siehe Abbildung A.144).

Action State

Ein Action State (Abbildung A.145) ist eine Abkürzung für einen Zustand mit einer interne Aktion und mindestens eine ausgehende Transition.

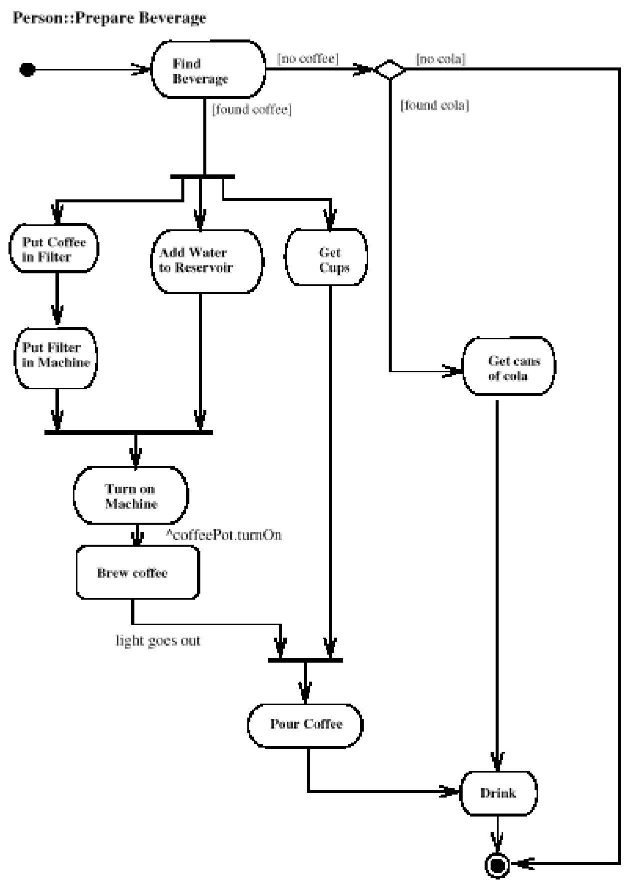


Abbildung A.144: Activity Diagramm

Abbildung A.145: Action State

Swimlanes

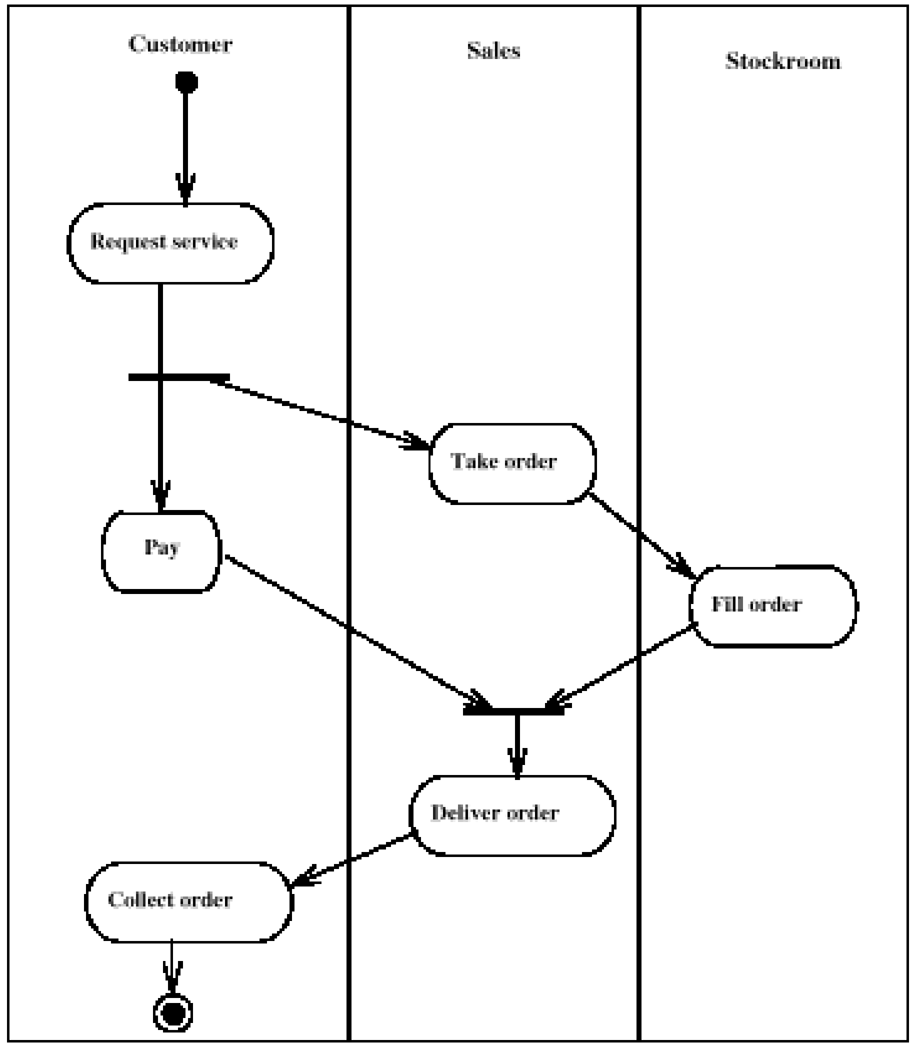


Abbildung A.146: Swimlanes

Ein Activity Diagramm kann in Swimlanes (Abbildung A.146) geteilt werden. Sie werden von benachbarten Swimlanes durch vertikale Linien von beiden Seiten getrennt. Jede Swimlane präsentiert die Verantwortlichkeit für den Teil von gesamten Activity, und kann vielleicht von eine oder mehrere Objekten implementiert sein. Jede Aktion wird an einer Swimlane zugewiesen.

Action-Objekt Fluß Beziehungen

Objekt verantwortlich für eine Aktion: Dieses kann man durch Zeichnen von Lebenslinien und Markierung der Stellen von Aktionen auf Lebenslinien darstellen. Jede Lebenslinie stellt ein eindeutiges Objekt dar. Für diese Form

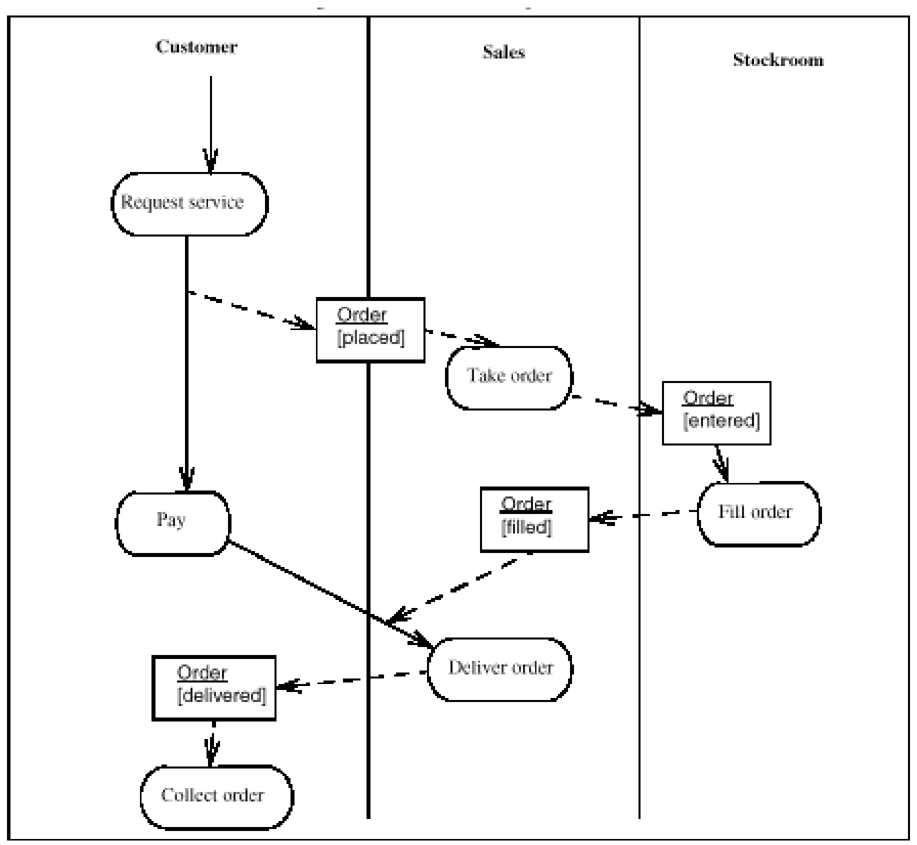


Abbildung A.147: Action-Objekt Fluß Beziehungen ist ein Sequenz Diagramm geeigneter und zu empfehlen. Falls aber keine Lebenslinie gezeigt wird, sind dann einige Objekten in Swimlane für die Aktion verantwortlich. Das Objekt wird aber nicht gezeigt.

Objektfluß: Objekte, die durch einer Aktion etwas hin zufügen oder von den etwas herausgehen, können durch ein Objektsymbol dargestellt werden. Ein gestrichene Pfeil wird die Beziehung zeigen (siehe Abbildung A.147).

Objekt in State

Beispiel: siehe oben

Ein Objekt kann mehrfach in einem Diagramm dargestellt werden. Bei jeder Erscheinung zeigt es eine unterschiedliche Punkt in seinem Leben. Um dies unterscheiden zu können, wird der Zustand vom Objekt in jeder Punkt in einer eckigen Klammer geschrieben und dem Namen von Objekt angehängt. Z.B. PurchaseOrder[approved]. Diese Notation kann auch in Collaboration Diagramm benutzt werden.

Kontroll-Icons

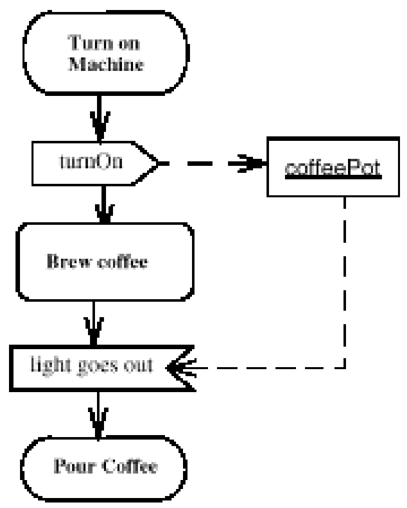


Abbildung A.148: Kontroll-Icons

Die Kontroll-Icons sind für Konstruktion von Activity Diagramme nicht notwendig, aber viele Benutzer ziehen dies auch vor (siehe Abbildung A.148).

Stereotypen:

- Signal receipt
- Signal sending

A.2.5.12 Implementation Diagramm

Implementation Diagramme zeigen Implementations-aspekte, inbegriffen Source-Code Strukturen und Run-time Implementationsstruktur. Sie kommen in zwei Formen vor : 1- Component Diagramm zeigen der Struktur von Code selbst und 2- Deployment Diagramm zeigen der Struktur von Run-time System.

Component Diagramm

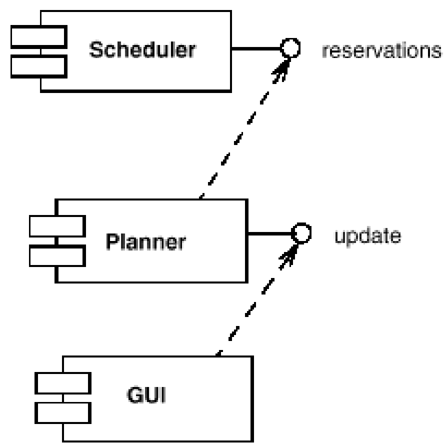


Abbildung A.149: Component Diagramm

Das Component Diagramm zeigt die Abhängigkeit zwischen Software Komponente, inbegriffen Quellcode Komponenten, binary Code Komponenten und die ausführbare Komponenten. Ein Component Diagramm hat nur eine Typform und keine Instanzform. Ein Component Diagramm ist ein Graph von Komponenten verbunden durch Abhängigkeitsbezihungen. Ein Diagramm, das Komponenten und Typen enthält, kann für die Anschaulichung von Compiler Abhängigkeiten benutzt werden. Dies wird durch gestrichene (Abhängigkeits-) Pfeile von einem Client Komponente zu einer lieferenden Komponente zeigen (siehe Abbildung A.149).

Deployment Diagramm

Ein Deployment Diagramm ist ein Graph von Knoten verbunden durch Kommunikationsassoziationen. Knoten können Komponenteninstanzen enthalten. Dieses zeigt dann, daß der Komponente lebt bzw. auf die Knoten ausgeführt wird. Komponenten können Objekte haben, d.h. das Objekt ein Teil von Komponente ist. Komponente werden durch gestrichene Abhängigkeitspfeil mit einander verbunden. Die Stereotyp <<support>> zeigt welche Komponente auf welche Knoten läuft. Die Stereotyp der Abhängigkeitsbeziehung <<becomes>> stellt die Wanderung von Komponenten von Konten zu Knoten bzw. die Wanderung von Objekten von Komponenten zu Komponenten (siehe Abbildung A.150).

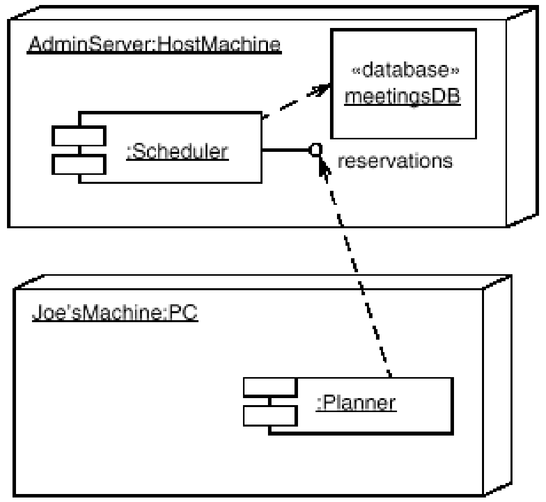


Abbildung A.150: Deployment Diagramm

Knoten

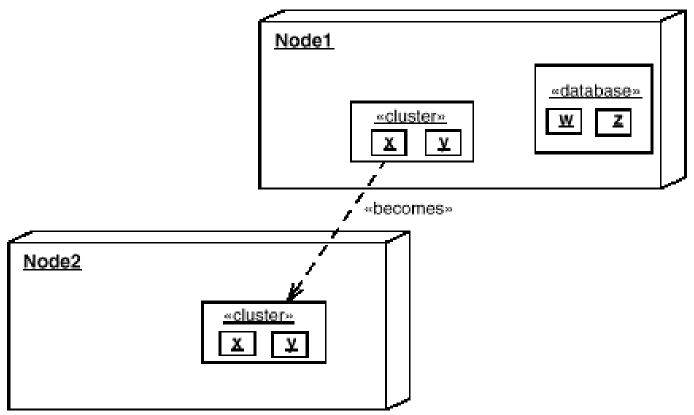


Abbildung A.151: Knoten

Knoten (Abbildung A.151) ist ein run-time physikalisches Objekt, daß eine Entwicklungsprozeß präsentiert. Generell hat ein Knoten mindestens ein Speicher und mehrfache Prozeßfähigkeiten.

Knoten können als Typ oder Instanz dargestellt werden. Sie können durch Assoziationen mit anderen Knoten verbunden werden. Die Assoziation kann dann eine Stereotyp zur Darstellung der Art der Kommunikationspfad haben. (z.B. die Art von Kanal oder Netzwerk)

Komponente

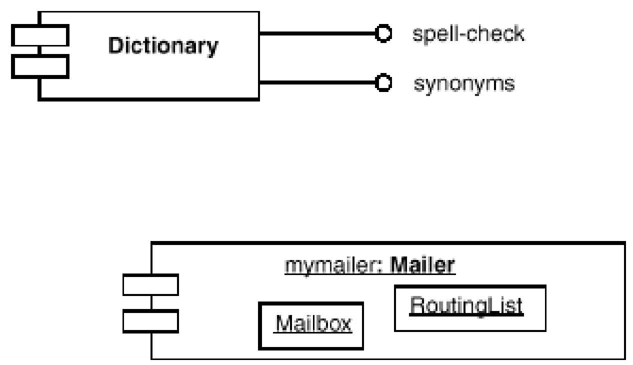


Abbildung A.152: Komponente

Eine Komponente (Abbildung A.152) präsentiert ein verteilbares Stück von Systemsimplementation, daß sowohl Software Code (Quellcode, Binäre Code, ausführbare Code) als auch Geschäftsdokumente für ein menschliche System enthält.

Die Komponenten können zur Darstellung von Compiler und run-time Abhängigkeit oder Informationsabhängigkeit für eine menschliche Organisation benutzt werden.

Die Lage von Komponenten und Objekten innerhalb von Objekten

Die Lage von einem Instanzen (inbegriffen Objekte, Komponenteninstanzen, Knoteninstanzen) innerhalb anderen Instanzen kann durch physikalische Verschachtelung in Form von Aggregation oder zusammengesetzte Assoziationspfaden dargestellt werden. Alternativ kann ein Instanz eine Eigenschaft location" besitzen, dessen Wert der Name von enthaltenen Instanz ist.

A.3 Autorenliste

Dokumente	Autoren
Vorwort	Martin G., Martin Mann, Holger Neumann
Einleitung	Martin G., Martin Mann, Holger Neumann
Projektplan	Andreas Rugo, Christiasn Cordes, Marc Plaßmeier, Holger Neumann
Anforderungsdefinition	Holger Neumann, Martin Mann, Verena Schmidt
OMT Vortrag / Objektmodell	Ersin Canli, Siyamak Ghotbi
OMT Vortrag / Erweitertes Objektmodell	Andreas Rugo, Frank Patz
OMT Vortrag / Dynamisches Modell	Karlo Wagner, Marcus Koch, Derya Dogan
OMT Vortrag / Funktionales Modell	Chun-Hon Yap, Arne Lindow
Fallstudie / Autovermietung	Sandro Wendt, Oliver Spittel
Fallstudie / Konferenzorganisation	Karlo Wagner, Siyamak Ghotbi
Fallstudie / Scotland Yard	Marc Plaßmeier, Christian Cordes
Fallstudie / Netztopologie	Ersin Canli
Fallstudie / Garagentor	Chun-Hon Yap, Arne Lindow
Infrastruktur	Marc Plaßmeier
Systementwurf I	Andreas Rugo, Oliver Spittel, Rainer Spittel, Holger Neumann, Frank Patz, Hamid Habet, Martin Mann, Verena Schmidt
Sprachentwurf I / Basisnotation	Ersin Canli
Semantik I	Chun-Hon Yap
Systementwurf II	Andreas Rugo, Oliver Spittel, Rainer Spittel, Holger Neumann, Martin Mann
Sprachentwurf II / State-Charts	Ersin Canli, Karlo Wagner
Sprachentwurf II / Activity-Charts	Siyamak Ghotbi, Derya Dogan
Sprachentwurf II / OCL (Constraints)	Chun-Hon Yap, Marcus Koch
Semantik II	Andreas Rugo
Implementierung	Ersin Canli
Schlußwort	Martin G., Martin Mann, Holger Neumann
Handbuch	Ersin Canli, Holger Neumann, Marc Plaßmeier
Featureliste / OMT	Ersin Canli, Siyamak Ghotbi, Andreas Rugo, Frank Patz, Karlo Wagner, Derya Dogan, Chun-Hon Yap, Arne Lindow, Marcus Koch
Featureliste / OOD	Frank Patz, Marcus Koch
Featureliste / BOOCH	Ersin Canli, Karlo Wagner, Derya Dogan, Marcus Koch
Featureliste / STATEMATE	Siyamak Ghotbi, Arne Lindow, Chun-Hon Yap, Marcus Koch
Sprachvergleich / UML	Siyamak Ghotbi
Autorenliste	Ersin Canli
Literaturverzeichnis	Marc Plaßmeier
Stichwortverzeichnis	Marc Plaßmeier

Index

Übersetzung, 147

About, 370
abstrakte Klasse, 250, 251, 380
  abstrakte Oberklasse, 250
  abstrakte Operation, 251
  abstrakte Unterklasse, 250
Activity, 267
  ControlActivity, 267, 292
  PseudoActivity, 267, 292
  SimpleActivity, 267, 291
Activity Chart, 267, 274, 286
Aggregation, 15, 31, 37, 38, 57, 157, 166, 249, 359, 380
Aggregationsbeziehung, 369
Aggregationsstrukturen, 38
Akteur, 72
Aktivität, 267
Aktivitätsdiagramm, 267
Anforderungsdefinition, 145
Ankerpunkt, 164, 169, 269, 366
Ansicht, 152, 170
Ansichtskomponente, 170
Antisymmetrie, 249
Anwendungsgebietskomponente, 151, 152
Arbeitsblatt, 34
Architektur, 147
Assoziation, 15, 27, 156, 166, 195, 246–250, 358, 377
  abgeleitete, 51
  als Klasse, 49, 385
  binäre, 247, 248, 359, 366
  geordnete, 378
  n-äre, 167, 247, 255, 359, 367
  qualifizierte, 30, 249, 358, 378
  ternäre, 247, 249
Assoziationsattribut, 29, 247, 378
Attribut, 26, 156, 188, 245, 249, 250, 253, 358, 376
  abgeleitetes, 51
Attributname, 245
Attributwert, 245
Auswahl, *siehe* Selection
AWT, 151

Basisnotation, 145, 244
Bearbeitungsmodus, 174
Begrenzung, 42
Benutzungsfunktionalität, 147
Benutzungsschnittstellenkomponente, 151, 169
Beziehung, *siehe* Assoziation
Beziehungstyp, 156
Blatt, 152, 162
Box-and-Line-Drawing, 143

Code Generator, 148
Concurrent Versions System, 113
Connector, 369
CVS, *siehe* Concurrent Versions System, 113

Datenbanksystem, 254
Datenfluß, 267, 269
Datenflußdiagramm, 70
  Akteur, 289
  Datenfluß, 289
  Datenspeicher, 289
  Kontrollfluß, 289
  Prozeß, 289
Datensicht, 152, 244
Datenspeicher, 73, 269
Datenverwaltungskomponente, 151
DBImporter, 150
Delegation, 46, 382
Delete, 363
deSIGN, 356
Diagramm, 151, 152, 162, 360
Diagramm Element, 273
Diagrammelement, 162, 269, 274, 364

Dialog, 20
Abfragedialog, 20
Informationsdialog, 20
Optionsdialog, 20
spezieller, 20
Diamantsymbol, 367
DMC, *siehe* Datenverwaltungskomponente
dynamische Sicht, 150

EBNF-Form, 282
Edit Mode, 364
Einschränkung, 160, 255, 256
Eintität, 256
emacs, 117
Entity, 255
Entwurfserweiterungen, 214
Entwurfsmuster, 47, 146
Entwurfsprobleme
Modellstruktur, 144
Enwurfsmuster
abstrakte Fabrik, 146
EPS-Grafik, 110
Ereignis, 52, 279
Ereignisfolge, 54
Ereignisklasse, 53
nebenläufiges, 53
Szenario, 53
Erweiterung, 41
Event, *siehe* Ereignis
Event Modell, 217
Event-Handler, 170

Fehlermeldungen, 373
First Sheet, 363

Generalisierung, 32, 38, 41, 42, 57, 154, 167, 249, 252, 359
Oberklasse, 250
Subklasse, 250
gmake, 357
grafischer Editor, 148, 152, 244
grafisches Element, 151, 152
GUI, 151
GUI-Bibliothek, 146, 151

HIC, *siehe* Benutzungsschnittstellenkomponente
Hilfe, *siehe* Online Help
HTML, 101, 103
HyperText, 107

IDE, 148, 172, 357
Infrastruktur, 101
Insert
Aggregation, 369
Association, 366
Class, 365
Generalisation, 369
n-ary Association, 367
Instanz, 245, 247, 250, 252, 254, 376
Intervallmenge, 247

JAVA, 116, 141, 144, 147, 148, 357, 362
Anweisungen, 120
Applet, 118, 134
Applikation, 118, 134
Array, 123
AWT, 170
Bezeichner, 120
Client/Server-Prinzip, 137
Compilieren, 134
Datentypen, 129, 130
DO-WHILE, 127
Dokumentationskommentare, 121
Exception, 135
Exceptionhandler, 135
FOR, 127
Grunddatentypen, 121
IF, 126
Interfaces, 131
JDK, 117, 132
Klasse, 123
Klassenbibliotheken, 132, *siehe* Klassenbibliotheken
Kommentare, 120
Konstanten, 124
Methode, 129
Multithreading, 134
Netzwerk, 136
objektorientiert, 116
objektorientierte Programmierung, 128
Packages, 133
Programmentwicklung, 134
Socket, 137
Sprachelemente, 120
Strings, 124

SWITCH, 126
Thread, 134, 135
URL, 136
Vererbung, 130
Virtual Machine Code, 116
WHILE, 127
Java, 6
Operatoren, 125
JDK, 356, 357

Kandidatenschlüssel, 47, 158, 254, 255, 385
Kardinalität, 28, *siehe* Multiplizität, *siehe* Multiplizität, *siehe* Multiplizität, 378
Klasse, 15, 25, 45, 150, 154, 164, 244, 245, 247, 249, 250, 254, 269, 357, 365, 376
abstrakte, 41
konkrete, 40
Vereinigungsklasse, 44
Klassendiagramm, 148, 164, 216, 217
Klassenname, 245
Kommando, 147, 152, 178
Kommandoschnittstelle, 172
Komponente, 150
Kontrollfluß, 73, 269
Kontrollpunkt, 166, 370

Last Sheet, 363
LaTeX, 101
Layer, 150
Link, 246, 247, 377
Load Translation Table, 362
Load/Save
Serialisierungsmethode, 360, 361
Textformmethode, 360
logisches Element, 151, 152, 154, 160, 267

Mehrfachvererbung, 44, 253, *siehe* Vererbung, 382
Member, 156
Menü, 152, 172
Menüleiste, 359
Message, 380
Metadaten, 46, 253, 385
Klassenatribut, 382
Klassenattribut, 253, 254
Klassenoperation, 253, 254, 385
Metamodell, 151
Methode, 26, 245, 250, 251
Methodenauswahl, 10
Modelchecker, 150
Modell, 150, 152, 254
Modul, 34, 150, 160, 267
Modulverwaltung, 148
Multiplizität, 157, 247, 255, 358, 367

n-äre Assoziation, 167
New Diagram, 360
New Sheet, 363
Next Sheet, 363

Oberklasse, 252, 253
Objekt, 25, 245, 247–249
abgeleitetes, 51
Objekt-Serialisierung, 146
Objektdiagramm, 26
Objektmodell, 244, 255
objektorientierte Programmiersprache, 245, 250
OMT, 141, 244, 248, 254–256, 289
dynamisches Modell, 24
funktionales Modell, 24
Objektmodell, 23
Online Help, 370
Open Diagramm, 360
Operation, 26, 156, 195, 245, 250, 252, 253, 358, 377
Fortpflanzung, 39
Operationsspezifikation, 73

Parameter, 156
PDC, *siehe* Anwendungsgebietskomponente
persistente Datenspeicherung, 151
Persistenz, 146, 148, 152
polymorphe Operation, 245
Postscript, 107, 147, 148
Previous Sheet, 363
Print, 362
EPS, 362
PS, 362
Programmiersprache, 247, 254
Projektablauf, 141

Projektmanager, 148
Projektplan, 143
Propagation, 380
Property, 364
Protokoll, 110
Prozeß, 72

Qualifizierer, 158, 214
Qualität, 9
Quit, 360

RCS, *siehe* Revision Control System
rechtwinklige Verbinder, 220
Relation, 245, 246, 249, 250, 358, 366
relationales Datenbanksystem, 12, 147, 148, 255, 362
Repository, 113–115
Reverse Engineering, 150
Revision Control System, 113
Revisionskontrolle, 113
Rolle, 157, 248
Rollenname, 29, 248, 358, 378

Save, 361
Save As, 361
Selection, 364
Serialisierungsverfahren, 152, 183
SGML, 7, 101, 106, 107, 147
SGML Tools, 108
  emacs, 108
  mv2folder, 110
  nsgmls, 109
  stpmake, 109
SGML-DTD, 147
Sheet, 363
Show Translation Table, 363
SIGN, 146
SIGN Modeling Language, 139, 147, 152
SIGN Panel, 172
SIGN Window, 170
Spezialisierung, 252
Sprachentwurf, 269
SQL, 147, 148, 362
State, *siehe* Zustand
State Chart, 267, 269, 278
STATEMATE, 278
statische Sicht, 139
StP, 356
Struktursicht, 152
Subklasse, 252, 253
SUN, 357
Systementwurf, 139, 143, 145, 244
  Activity Chart, 143, 144
  Architektur, 143
  Datensicht, 143
  Diagramm, 144, 145
  dynamische Sicht, 143
  Entwurfsprobleme, 143, 144
  funktionale Sicht, 143
  Klasse, 145
  Klassendiagramm, 143–145
  Layout, 145
  logische Element, 145
  logisches Element, 144
  Metamodell, 144
  Modell, 144, 145
  Modellelemente, 145
  Präsentation, 146
  Prototyp, 143
  State Chart, 143, 144
  Struktursicht, 143
  Use Case, 143
  Zustandsdiagramm, *siehe* State Chart

Textbox, 167, 218
Toolbar, 172, 359
Transformation, 16
  JAVA, 362
  Java, 206
  SQL, 196, 362
Transition, 56
Transitivität, 249

UML, 141

Verbinderendpunkte, 214, 215, 217
Verbindung, 27
Vererbung, 32, *siehe* Generalisierung, 249, 250, 359, 380
Vererbungsbeziehung, 369
Vergleichsmethoden, 244
Vergrößerung, *siehe* Zoom
Verknüpfung, *siehe* Link

Wartbarkeit, 9
Wiederverwendbarkeit, 10

Zielsprache, 244
Zoom, 370
Zustand, 279
Zustandsdiagramm, 56, 280
  Aktion, 57
  Aktivität, 57
  Bedingung, 56, 280
  Generalisierung, 58
  Nebenläufigkeit, 60
  Operation, 57, 281
  strukturiertes, 57
  Zustand, 54

Literaturverzeichnis

[AH98] H. J. Kreowski A. Habel. Material zur Vorlesung Graphtransformation im SS98, 1998.

[AO94] R. Apt and E. R. Olderog. *Programmverifikation*. Springer-Verlag, 1994.

[Bal96] Heide Balzert. Objektorientierte Systemanalyse. *Spektrum Akademischer Verlag*, page 148, 1996.

[Boo91] Grady Booch. *Object Oriented Design with Application*. Benjamin/Cummings, Redwood City, 1991.

[Bra96] Neil Bradley. *The Concise SGML Companion*. Addison-Wesley Longman, Reading, MA, USA, November 1996.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. UML Summary, UML Semantics, UML Notation Guide, UML Object Constraint Language Specification, UML Extensions for Objectory Process for Software Engeneering, 1997.

[CY91a] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, London, 2 edition, 1991.

[CY91b] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, London, 1991.

[DN96a] P. Scholz D. Nazareth, F. Regensburger. A Compositional Way to Model Parallel Systems. Technical report, PDCS'96, 1996.

[DN96b] P. Scholz D. Nazareth, F. Regensburger. Mini-Statecharts: A Lean Version of Statecharts. Technical Report TUM-I9610, Universität München, 1996.

[DN96c] P. Scholz D. Nazareth, F. Regensburger. Mini-Statecharts: A Lean Version of Statecharts. Technical Report TUM-I9610, Universität München, 1996.

[EF97] M. Eckrich and M. Fuchs. Semiformale Entwurfsmethoden bei BMW. *it+ti – Informationstechnik und Technische Informatik*, 39(3):29–33, 1997.

[EFT94] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical Logic.* Undergraduate Texts in Mathematics. Springer-Verlag, Berlin, 2nd edition, 1994. (1st ed., 1984).

[Fla97] David Flanagan. *Java in a Nutshell: a desktop quick reference.* Java Ser. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1997.

[Fla98] David Flanagan. *Java in a Nutshell, Deutsche Ausgabe.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1998.

[Gen34] G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934.

[GH94] E. Gery and D. Harel. Executable Object Modeling with Statecharts. Technical Report CS94-20, Weizmann Institute of Science, Faculty of Mathematical Sciences, January 1, 1994.

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1996.

[Gog96a] M. Gogolla. Material zur Vorlesung Datenbanksysteme I/II im WS96/97 bzw. SS97, 1996.

[Gog96b] M. Gogolla. Projektantrag Software through Pictures (StP), 1996.

[Gol94] Charles F. Goldfarb. *The SGML guide.* Interleaf, Inc., Waltham, MA, USA, 1994.

[GP98] Martin Gogolla and Francesco Parisi Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques.* Technische Universität München, TUM-I9803, 1998.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts (Extended Abstract). In *Proceedings, Symposium on Logic in Computer Science*, pages 54–64, Ithaca, New York, 22–25June 1987. The Computer Society of the IEEE.

[HU94] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, N. Reading, MA, 1994.

[JCJO92] I. Jacobson, M. Christenson, P. Jonsson, and G. Oevergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, 1992.

[JP97] P. Scholz J. Philipps. Compositional Specification of Embedded Systems with Statecharts. Technical report, TAPSOFT'97, 1997.

[KR96] H. Kubicek and K. Reimers. Material zur Vorlesung Informationstechnikmanagement im WS96/97, 1996.

[KRCP96] M. Koch, A. Rugo, C. Cordes, and M. Plassmeier. Scotland Yard, 1996.

[Kre97] H. J. Kreowski. Material zur Vorlesung Theorie Regelbasierter Systeme im WS97/98, 1997.

[Küh96] Ralf Kühnel. *Die Java Fibel Programmierung interaktiver Homepages für das World Wide Web*. Addison-Wesley, Reading, MA, USA, 1996. Includes CD-ROM.

[LL96] Charles L. Perkins Laura Lemay. *Java in 21 Tagen*. SAMS, 1996.

[Ltd97] Formal Systems (Europe) Ltd. Failures-Divergence Refinement, FDR2 User Manual, 1997.

[MM96] J. Malcolm and R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, School of Science, 1996.

[Pel95] J. Peleska. The Languages of STATEMATE, 1995. Material zur Vorlesung Softwaretechnik im WS95/96.

[Per95] A. Peron. Statecharts, Transition Structures and Transformations. *Lecture Notes in Computer Science*, 1995.

[PPCH94] C. Peterson, J. Peleska, and W. P. de Roever C. Huizing. Formal Semantics for Ward & Mellors Transformations Schemas and its Application to Fault-Tolerant Systems. Technical Report 9420, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Kiel, Germany, 1994.

[PS91] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. *Lecture Notes in Computer Science*, pages 244–264, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objekt-oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

[RG92] K. S. Rubin and A. Goldberg. Object-Behavior Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall international series in computer science, 1997.

[Sca95] B. Scattergood. *Tools for CSP and Timed CSP*. PhD thesis, University of Oxford, 1995.

[Sch96] P. Scholz. A Lightweight Formalism for the Specification of Reactive Systems. Technical report, SOFTSEM'96, 1996.

[Sha94] Oliver Sharp. The Perl Programming Language. *Dr. Dobb's Journal of Software Tools*, October 1994.

[SSC+96] H. Schottmüller, A. Semitchastnyi, E. Canli, J. Pelaez-Avila, and T. Klein. Graphische Darstellung der Netztopologie, 1996.

[UB93] Fachbereich Informatik Universität Bremen. Studienordnung 1. Oktober 1993, 1993.

[US94] A. C. Uselton and S. A. Smolka. A Compositional Semantics for Statecharts using Labeled Transition Systems. Technical report, State University of New York, Department of Computer Science, Stony Brook, New York, USA, 1994.

[Wir86] Wirth. *Compilerbau*. Springer, Berlin, 1986.

[WSCP96] Larry Wall, Randal L. Schwartz, Tom Christiansen, and Stephen Potter. *Programming Perl*. Nutshell Handbook. O'Reilly & Associates, 2nd edition, 1996.