ENTWURF, REALISIERUNG UND TEST EINES OBJEKTORIENTIERTEN CAD-DATENMODELLS FÜR DIE TRAGWERKSPLANUNG

vom Fachbereich

Architektur, Raum- und Umweltplanung,
Bauingenieurwesen der Universität Kaiserslautern

zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte

DISSERTATION

von

Dipl.-Ing. Samer Hamwi

Kaiserslautern 2000

D386

Die Promotionskommission setzt sich wie folgt zusammen:

Vorsitzender:	Prof. Dr.-Ing. Wittek
1. Berichterstatter:	Prof. Dr.-ing. Wassermann
2. Berichterstatter:	Prof. Dr.-Ing. Streich

Datum der mündlichen Prüfung: 16. Februar. 2000

Inhaltsverzeichnis

Vorwort

Im Rahmen der Promotion haben mir verschiedene Menschen beigestanden. Ich möchte hier all denen danken, die mich während dieser Zeit tatkräftig unterstützt und auch ertragen haben. Ich kann mich leider nicht bei allen bedanken, nur einige Namen stellvertretend herausgreifen.

Mein besonderer Dank gilt meinem Doktorvater, Herrn Prof. Dr.-Ing. Wassermann, der zu dieser Dissertation sinnvolle Anregungen und Hinweise gegeben hat, sowie Herrn Dr. Ing. Heck, dem ganzen Lehrteam, unserer Sekretärin Frau Obry und unseren wissenschaftlichen Hilfskräften am Institut für Bauinformatik der Universität Kaiserslautern.

Mein Dank gilt Herrn Prof. Dr.-Ing. Wittek , der das Amt als Vorsitzender der Promotionskommission angenommen hat und Herrn Prof. Dr.-Ing. Streich, der die Dissertation begutachtet hat.

Ohne den Rückhalt meiner Familie - hier gilt mein Dank meinen Eltern, meinen Geschwistern und meiner Frau - wäre dieses Werk nicht in dieser Form entstanden.

Ich hoffe, diese Arbeit hat sich gelohnt, und ich habe damit einen Beitrag zur wissenschaftlichen Forschung geleistet.

1. Einleitung

1.1 Allgemeines

Die vorliegende Arbeit befaßt sich mit dem Entwurf, der Realisierung und dem Test eines objektorientierten Datenmodells für die Tragwerksplanung (Strukturanalyse) von Gebäuden. Ziel ist es, die Tragwerksplanung zu unterstützen und das Verständnis für das Tragverhalten komplizierter Tragstrukturen zu erleichtern.

Ein Modell kann im allgemeinen als eine Abstraktion eines Teils der Realität individueller Natur definiert werden. Ein Modell ist somit, je nach Betrachtungsweise, immer individuell. Für rechnergestützte Entwurfsysteme - CAD-Systeme (Computer Aided Design) - ein Informations- bzw. Datenmodell ist das zugrundeliegende Bestandteil. Das Ergebnis der vorliegenden Arbeit ist ein Modell von Daten der Tragwerksplanung, das durch das (auch im Rahmen dieser Dissertation) entwickelte CAD-System realisiert und ausgetestet werden soll.

Betrachtet man die Planungsphasen eines Gebäude, wie sie in der HOAI [18] festgelegt ist, so bildet die Tragwerksplanung einen erheblichen Arbeitsanteil. Dabei ist beispielsweise die Grundleistung des Tragwerksplaners an der Entwurfsplanung zu 12%, an der Genehmigungsplanung zu 30% und an der Ausführungsplanung zu 42% der Honorare anteilig zu Gesamtkosten festgelegt.

Wesentliche Aspekte der Abwicklung von Bauprojekten, wie kurze verfügbare Projektzeit, hohe Qualitätsanforderungen und Kostenersparnis erschweren neben der Komplexität der zu lösenden Aufgaben – insbesondere bei großen Bauprojekten - die Arbeit des Tragwerksplaners. Zum Forschungsgebiet gehört die optimale Unterstützung des Tragwerksplaners bei der Erledigung seiner Aufgaben durch den Einsatz rechnergestützter Entwurfssysteme (CAD-Systeme). Durch diese Systeme soll sowohl der Architekt beispielsweise bei Grundrißplanung, Kostenschätzung und Visualisierung als auch der Tragwerksplaner bei der 3D-Konstruktion von Bauteilen, 3D-Lastdarstellung, und Visualisierung durch Werkzeuge und vielfältige Darstellungsmöglichkeiten unterstützt werden [58].

Bereits heute werden Tragwerksplaner durch verschiedene Computersysteme wie Programme für Textverarbeitung, Tabellenkalkulation und Bemessung (z.B. unter Anwendung der Finite Elemente Methode, FEM), unterstützt. Der CAD-Einsatz ist ebenfalls in der Tragwerksplanung weit verbreiteter Stand der Technik (siehe Kapitel *2*) [76]. Das bedeutet, daß der CAD-Einsatz heute keine Vorteile bei der Auftragsvergabe durch Bauherren ergibt, der Verzicht jedoch Nachteile bringt.

CAD-Systeme der ersten Generation - Mitte der 80er Jahre - waren zeichnungsorientiert. Daß bedeutet, sie verfügten lediglich über grafische Zeichnungselemente (wie Punkte, Linien, Polygone, Kreise, Bögen, Texte, Schraffuren) und geometrische Elemente (wie z.B. Flächen, Prisma, Kugel, Kegel)

[60] [39], die lediglich zur grafischen Darstellung dienen. Sie hatten zwar Vorteile, wie einfache Änderung, Archivierung und Versand von Planungszeichnungen (wie z.B. das System AutoCAD der amerikanischen Firma Autodesk), waren den Erwartungen der Architekten und Bauingenieure jedoch zurückgeblieben.

Bauspezifische CAD-Systeme der neuen Generation – um 1990 - sind bauteilorientiert. Das bedeutet, Zeichnungsobjekte bei solchen Systemen sind eigenständige Informationseinheiten, die eine bauspezifische Bedeutung besitzen und durch ihre Eigenschaften jeweils ein Bauteil repräsentieren [56].

Die Entwicklung bauteilorientierter CAD-Systeme erfordert das Erstellen von Datenmodellen. Dabei entstehen unterschiedliche Datenmodelle, die verschiedene Phasen des Planungsprozesses von Gebäuden abbilden (wie beispielsweise: Gebäudemodelle für den Architekturentwurf [31], Tragwerksmodelle zur statischen Berechnung [76], Geländemodelle für topographische Messungen, oder Gebäudemodelle für die technische Gebäudeausrüstung). Die Entwicklung von CAD-Systemen beginnt damit eine neue Richtung anzunehmen, und zwar der Übergang von der zeichnungsorientierten zur produktorientierten Modellierung.

Ein Schwerpunkt der Forschung heute ist die Erstellung eines fachspezifischen Datenmodells und das Umsetzen auf CAD-Systeme für die Tragwerksplanung. Die Modellierung von CAD-Systemen in der Tragwerksplanung beinhaltet die Erstellung solches fachspezifischen Datenmodells mit den dazugehörigen Datenstrukturen, die eigenständige Informationseinheiten für Bauteile (wie z.B. Wand, Stütze, Träger, Fundament) bilden. Die Forschung hat die Vorteile des Einsatzes objektorientierter Konzepte in der Tragwerksplanung von der Analyse über das Design zur Programmierung gezeigt [67] [75]. Diese Vorteile erwecken den Wunsch, ein objektorientiertes Datenmodell für die Tragwerksplanung zu entwickeln, das es erlaubt, die Produktivität, die Wiederverwendbarkeit und die Erweiterbarkeit ohne Verlust der Übersichtlichkeit zu erhöhen. Die Anwendung der objektorientierten Entwurfsmethoden in der Softwareentwicklung ist ein Schwerpunkt der aktuellen Forschung im Bauwesen (DFG-Schwerpunkt SP-694 „Objektorientierte Modellierung in Planung und Konstruktion“) wie z.B. die Arbeit von P. Heck an der Universität Kaiserslautern [30].

In den Arbeiten von Niestroy [51] und Rüppel [65] wird das Tragwerk durch zwei objektorientierte Teilproduktmodelle beschrieben. Das Tragwerksmodell beschreibt die Gebäudedaten hinsichtlich der tragenden Bauteile und deren Topologie. Das statische Modell beschreibt die statisch relevanten Strukturdaten des Tragwerks. Ziel war es, aus den Entwurfsplänen des Architekten, die in Form einer CAD-Zeichnung vorliegen, Informationen für die Tragwerksanalyse in objektorientierter Form (Teilproduktmodelle) zu generieren. Niestroy und Rüppel [51] [65] mußten daher zwangsläufig auf die Datenmodelle der benutzten CAD-Systeme (Nemetschek, RIB, Hochtief Software) eingehen. Im wesentlichen wurden dabei die Probleme des Datenaustauschs auf Dateiebene behandelt.

Im Rahmen dieser Dissertation wird ein **objektorientiertes CAD-System mit objektorientiertem Datenmodell** für die Tragwerksplanung von Gebäuden entwickelt, das die CAD-Funktionalitäten unter Anwendung des objektorientierten Konzeptes realisiert (siehe Kapitel *3)*. Das bedeutet, daß **kein Basis-CAD-System** (sogenannt Low-cost-CAD-System) und **keine Standardsoftware als Basis-**

Modellierer für die Realisierung der CAD-Funktionalitäten, wie Kopieren, Verschieben, Identifizieren, Fangen, Zoomen usw. verwendet wird. Des weiteren wird **kein abstraktes Datenmodell für die rechnerinterne Darstellung** eingesetzt. Es wird für die rechnerinterne Datenhaltung von 3D-Objekten keine Grafikbibliothek für ein klassisches Darstellungsmodell, wie Drahtmodell (B-Rep), Volumen- oder Festkörpermodell (CSG) und Flächenmodell, eingesetzt. In dem in Kapitel *5 u. 6* vorgestellten CAD-System werden selbstentwickelte einfache geometrische 2D- und 3D-Objekte (Punkt, Linie, Polygon, Prisma etc.) mit eigenen Attributen als Grundelemente zur rechnerinternen Beschreibung der Geometrie eingesetzt.

Die vorliegende Arbeit enthält acht weitgehend aufeinander bauende Kapiteln. Der CAD-Einsatz als Stand der Technik und Schwerpunkt der Forschung in der Tragwerksplanung wird in **Kapitel *2*** diskutiert. Das Ziel dieser Arbeit und die Abgrenzung des Problembereichs und des Datenmodells werden in **Kapitel *3*** erläutert. **Kapitel *4*** führt die Grundlagen der objektorientierten Vorgehensweise und ihre eindeutige Vorteile bei der Softwareentwicklung von der Analyse über Design zur Implementierung ein. Das mit Hilfe der Objektorientierung entwickelte Datenmodell für CAD-System zur Unterstützung des Tragwerksplaners wird in **Kapitel *5*** vorgestellt. Die Realisierung des entwickelten CAD-Datenmodells erfolgt in **Kapitel *6*** und das Austesten in **Kapitel *7***. Eine Zusammenfassung der vorliegenden Arbeit mit einer abschließenden Betrachtung wird in **Kapitel *8*** vorgenommen.

Es erscheint angebracht, im folgenden Abschnitt einen Überblick über die Bauinformatik als junges Forschungsgebiet, das die Informations- und Kommunikationstechnik mit den fachlichen Anforderungen im Bauwesen verknüpft, zu geben.

1.2 Die Informationstechnik im Bauwesen

Die Bauinformatik, ein junges Forschungsgebiet im Bauingenieurwesen, ist in den letzten zehn Jahren an deutschen Universitäten entstanden. Sie beschäftigt sich mit der Umsetzung von Wissen, Erfahrung und Urteilsvermögen des Menschen auf Modelle und Prozesse des Bauwesens unter Nutzung moderner Informations- und Kommunikationstechnik.

Zur Erfüllung ihrer Aufgabe bei der Suche nach und bei der Entwicklung von Ingenieurlösungen, benutzt die Bauinformatik verschiedene Techniken, Methoden und Verfahren [35].

Die Techniken, die in engen Beziehungen zur technischen und angewandten Informatik stehen, umfassen folgendes: Geräte (wie der Arbeitsplatzrechner), Betriebsysteme, Techniken zur Datenhaltung (wie Architektur der Dateisysteme, Speichermedien), Datenübertragung Kommunikationstechnik (wie Netzwerke, Übertragungsprotokolle, Netzwerkbetriebsysteme, Netzwerkdateisysteme, usw.), Programmiertechniken (wie höhere und maschinennahe Programmiersprachen, Entwicklungstechniken von Programmen, usw.), und Entwicklungsumgebung (wie Editor, Compiler, Linker, Interpreter, Softwarebibliotheken usw.).

Für fachgebietsübergreifende Aufgaben aus dem Bauingenieurwesen und deren Realisierung greift die Bauinformatik Methoden aus der Informatik und der angewandten Mathematik auf. Diese Methoden umfassen die theoretischen Grundlagen, Datenstrukturen und Algorithmen. Es bestehen dabei enge fachliche Bezüge zu diesen Wissensgebieten. Im folgenden sind typische Methoden der Bauinformatik zusammengefaßt:

- Algebraische Methoden, wie lineare Algebra, Matrizen und Vektoren, Mengenoperation, Relationen zwischen Mengen.

- Numerische Methoden, wie numerische Interpolation, Integration und Differenziation, Finite Differenzen, Finite Elemente, Zeitverlaufsverfahren.

- Planungsmethoden, wie Systemplanung, Statistik, Optimierung, grafische Computersimulation.

- Darstellungsmethoden, wie Visualisierung, Grafik, technische Zeichnungen, technische Dokumentation, Textverarbeitung, Hypertext.

Typische Verfahren der Bauinformatik sind: Planungsverfahren, Entwurfsverfahren, Konstruktionsverfahren, Ausführungsverfahren und Nutzungsverfahren. Sie bilden die Grundlage für eine rechnergestützte Bearbeitung von Bauprojekten.

Die verschiedenen Phasen der Bauplanung erfordern das Erstellen von unterschiedlichen Datenmodellen, die diese Phasen mit der Unterstützung des Computers abbilden. Die dabei entstandenen Teilproduktmodelle müssen in ein gemeinsames Produktmodell für eine durchgängige Datenverarbeitung der Gebäudeplanung integriert werden. Die Erstellung und die Integration dieser Teilproduktmodelle ist eine wichtige Aufgabe der Bauinformatik. Unter Einsatz ihrer Methoden und Techniken befaßt sich die Bauinformatik mit dem systematischen Entwerfen von solchen Teilproduktmodellen, der schematischen Strukturierung der Modellobjekte und der Integration der Modellkomponenten für technische Daten und Normen.

Zusammenfassend sei gesagt, die Bauinformatik ist eine auf die fachspezifischen Belange des Bauingenieurwesens zugeschnittene Informatik. Zu ihrer Entwicklung tragen verschiedene Institutionen und Körperschaften bei:

- Bauunternehmen, dabei kommen in zunehmendem Maße computerorientierte Branchenlösungen zum Einsatz.

- Planungs- und Ingenieurbüros, wo heute die Aussage „ohne Computer geht es nicht“ gilt und die Computerlösungen für ausgewählte Tätigkeitsfelder im Bauwesen, wie CAD/CAE-Applikationen stark zum Einsatz kommen.

- Softwarefirmen, Computerhersteller und Beratungshäuser, die mit Bauwesen zu tun haben und auf nationaler bzw. internationaler Ebene Ingenieursoftware bzw. bauspezifische Software entwickeln.

- Hochschulen, die Forschung auf dem Gebiet betreiben und das Wissen verbreiten.

- Andere, wie Körperschaften des öffentlichen Rechts, die mit Ingenieuraufgaben betraut sind und intensiv Informationstechniken einsetzen, Behörden und Bauherren, die für administrative Aufgaben zunehmend integrierte DV-Systeme als Werkzeuge für Informationsmanagement einsetzen.

Der Stand der wissenschaftlichen Forschung bei der Suche nach der optimalen Technik für die Unterstützung der Arbeit in der Bauplanung und vor allem in der Tragwerksplanung wird im Kapitel *2* geschildert.

2. Stand der Technik und der Forschung

2.1 Entwicklung der CAD-Systeme in der Tragwerksplanung

Um den Einsatz von CAD-Systemen (Computer Aided Design) in der Tragwerksplanung besser diskutieren zu können, soll hier ein Überblick über die Entwicklung und den Einsatz computergestützter Systeme im Bauwesen und den Anfang des neuen Informationszeitalter in den 80er Jahren gegeben werden.

Die elektronische Datenverarbeitung (EDV) wurde schon um 1960 mit viel Optimismus für übliche Aufgaben im Bauwesen wie z.B. Rechenprogramme für Statik eingeführt. In der Forschung haben sich die Erwartungen ganz erfüllt, in der Praxis des Bauwesens hingegen waren die Voraussetzungen für den wirtschaftlichen Einsatz nicht erfüllt [53]. Sowohl in der Bundesrepublik Deutschland als auch im Ausland wurden Forschungen auf diesem Gebiet vorangetrieben.

Beispielsweise wurde in einem Vorschlag des Fachbereiches Bauingenieur- und Vermessungswesen der Universität Berlin und des Entwicklungszentrums EDV e.V. Stuttgart versucht, eine Standardisierung der Software durch die Entwicklung eines Informationssystems für das Bauwesen (ISB) zu erzielen [36]. Das ISB war ein speziell für das Bauwesen geeignetes und auf Großrechnern einzusetzendes Datenverarbeitungssystem, das aus einem Systemkern und zahlreichen Bausteine und Projektdaten bestand. Jeder der Bausteine (wie beispielsweise Geometrie, Lasten und Statik) enthielt Algorithmen und Daten, die für ein spezifisches Fachgebiet des Bauwesens benötigt werden.

Am Massachusetts Institute of Technology (Massachusetts/USA) wurde ein System ICES [62] mit einer Reihe von ingenieurmäßigen Bausteinen, die in den USA verbreitet sind, entwickelt. In Großbritannien gelang die Entwicklung eines ICES-ähnlichen Systems GENESYS [1], das von dem Ministry of Public Building and Works unterstützt war.

Das Entwicklungstempo in den 60er und 70er Jahren war jedoch hinter den Erwartungen zurückgeblieben. Die Erfahrungen dabei zeigten, daß eine große gemeinsame Datenbasis für das Bauwesen nicht zu erreichen war. Daraus sind viele Informationsinseln entstanden. Die Verbindung zwischen den Informationsinseln war nicht systematisiert.

Nach der revolutionären Entwicklung in der Computer Branche und vor allem die Erfindung des PC in den 80er Jahren hat sich die Nachfrage nach neuer Software gesteigert. Man wollte die neuen leistungsfähigen Rechner optimal nutzen und für neue Aufgaben einsetzen, dabei wurde auch an neue Einsatzgebiete im Bauwesen gedacht.

Der erste Einsatz von CAD-Systemen geht auf die ersten Entwicklungen interaktiver grafischer Datenverarbeitung Mitte der 80er Jahre zurück [23] [60]. Diese Systeme versprachen die Vereinfachung und Automatisierung von Verfahren in der Bauplanung [58]. Sie sollen dabei die Unterstützung des Computers und der elektronischen Datenverarbeitung (EDV) zur Erledigung bestimmter Aufgaben anbieten [56].

Die CAD-Systeme der ersten Generation waren noch nicht bauspezifisch. Es gab keine Unterschiede zwischen einem CAD-System für den Maschinenbau und einem CAD-System für das Bauwesen. Sie verfügten lediglich über grafische Zeichnungselemente (wie Punkte, Linien, Polygone, Kreise, Bögen, Texte, Schraffuren) und geometrische Elemente (wie z.B. Flächen, Prisma, Kugel, Kegel) [60] [39]. Ein Beispiel für ein solches Produkt stellt das System AutoCAD der amerikanischen Firma Autodesk dar. Damit konnten Architekten Baupläne und Bauplaner Bewehrungspläne erstellen. Eine Wand wird z.B. durch einen Polygonzug, eine Stütze durch ein Rechteck oder einen Kreis repräsentiert.

Diese CAD-Systeme hatten zwar Vorteile, wie einfache Änderung, Archivierung und Versand von Planungszeichnungen. Der Nachteil lag aber darin, daß sie auf die Vorstellung der Entwickler bauten, daß nur konventionelle Arbeit (wie z.B. auf dem Zeichnungsbrett) in einem elektronischen System nachgebildet werden soll. Sie waren zeichnungsorientiert. Zeichnungsobjekte besaßen keine bauspezifischen Eigenschaften und bildeten keine Informationseinheiten für Bauteile, wie z.B. Wand oder Stütze ab [39].

Erst Mitte der 80er Jahre kamen die ersten bauspezifischen CAD-Systeme auf den Markt und wurden zunehmend in allen Phasen der Gebäudeplanung eingesetzt [56]. Die bauspezifischen CAD-Systeme der neuen Generation sind bauteilorientiert. Zeichnungsobjekte haben dabei eine bauspezifische Bedeutung und repräsentieren durch ihre Eigenschaften jeweils ein Bauteil.

Die Anwendung der Bauteilorientierung auf CAD-Systeme erfordert das Erstellen von Datenmodellen, die Informationen bzw. Daten der abgebildeten Planungsphasen beinhalten. Die verschiedenen Phasen des Planungsprozesses von Gebäuden werden durch unterschiedliche Modelle abgebildet. Daraus entstanden verschiedene Produktmodelle wie beispielsweise: Gebäudemodelle für den Architekturentwurf [31], Tragwerksmodelle zur statischen Berechnung [76], Geländemodelle für topographische Messungen, Gebäudemodelle für die technische Gebäudeausrüstung. Die Entwicklung von CAD-Systemen fing damit an, einen Übergang von der zeichnungsorientierten zur produktorientierten Modellierung aufzuweisen.

Das folgende Beispiel zeigt die Vorteile der Produktmodellierung. Die Abbildung 2.1 stellt eine mehrschalige Außenwand dar, wie sie in einer Bauzeichnung vorkommt, die mit einem CAD-System erstellt werden kann.

Die Vorteile liegen darin, daß je nach Produktmodell des CAD-Systems eine eigene Informationseinheit, etwa eine mehrschalige Außenwand mit ihrer Öffnung und Typ (Architekturmodell) bzw. eine tragende Wand mit ihrer Öffnung (Tragwerksmodell), aus der Zeichnung (Punkte, Linien, Schraffuren, etc.) zu erkennen möglich ist (siehe Tabelle).

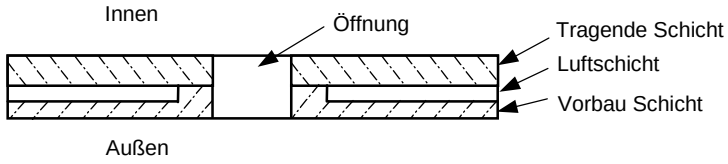


Abbildung 2.1: Eine mehrschalige Außenwand

In der folgenden Tabelle werden die Objekte, die in den unterschiedlichen Modellen erkennbar sind, aufgelistet:

Mehrschalige Wand		
Zeichnung	Produktmodelle	
	Architekturmodell	Tragwerksmodell
- Punkte - Linien - Linientypen - Schraffuren - Farben - etc.	- Mehrschalige Außenwand - Typ der Wand - Fenster	- Tragende Wand (die tragende Schicht). - Öffnung

Eine optimale Vorgehensweise bei der Erstellung von Produktmodellen (Modellierung) in der Bauplanung wird immer noch gesucht. Die wissenschaftliche Forschung zeigt die Vorteile des Einsatzes objektorientierter Konzepte von der Analyse über das Design zur Programmierung [67] [75]. Im folgenden Abschnitt wird der Stand der Forschung auf diesem Gebiet näher erörtert.

2.2 Stand der Forschung

Um einen Einblick in den Stand der aktuellen Forschung in der Bauinformatik zu bieten, werden im folgenden drei bezüglich der vorliegenden Dissertation relevante Forschungsgebiete vorgestellt:

- Integration und Automatisierung verschiedener CAD-Systeme und die dabei entstandenen Probleme des Datenaustauschs zwischen solchen Systemen.

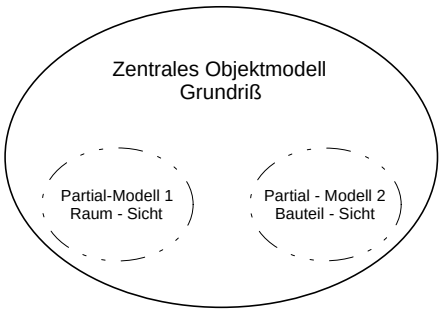
- Modellierung der Arbeitsphasen in der Bauplanung und die Anwendung der objektorientierten Vorgehensweise bei Produktmodellierung von CAD-Sytemen.

- Kommunikationstechniken für eine verteilte, gleichzeitige und kooperative Arbeit im Bauwesen.

Die Erfahrung in der letzten Jahre zeigt, daß sich die Qualität und Leistungsfähigkeit des Entwurfsprozesses im Bereich der Architektur und des Bauwesens nur durch zunehmende Automatisierung des Entwurfs und des Konstruktionsprozesses verbessern läßt. Nur durch die Integration der Bearbeitung von verschiedenen Informationen aus verschiedenen Planungsprozessen kann ein hoher Automatisierungsgrad erzielt werden.

Die Automatisierung durch den Einsatz von CAD-Systemen verspricht dabei eine schnellere Projektabwicklung mit höherer Qualität, was die Höhe der Investitions- und Einarbeitungskosten rechtfertigen kann. Möglichkeiten für eine durchgängige Unterstützung des gesamten Planungsprozesses von Gebäuden wurden in den letzten Jahren erforscht. In der Produktmodellierung von Rüppel [66] wurden Teilproduktmodelle für die verschiedenen Entwicklungszustände im Planungsprozess eingeführt.

Ein Forschungsschwerpunkt ist die optimale Vorgehensweise bei der Erstellung von Produktmodellen (Modellierung). Die wissenschaftliche Forschung hat dabei den Vorteil des Einsatzes objektorientierter Konzepte von der Analyse über das Design zur Programmierung in der Tragwerksplanung zeigt [67] [75]. Die Anwendung der objektorientierten Entwurfsmethoden in der Softwareentwicklung ist ein Schwerpunkt der aktuellen Forschung im Bauwesen. Forschungsprojekte der DFG (Deutsche Forschungsgemeinschaft) laufen heute mit verschiedenen Schwerpunkten im Bereich der objektorientierten Modellierung im Bauwesen. Am Institut für EDV-gestütztes Entwerfen, Berechnen und Konstruieren im Bauwesen der Universität Kaiserslautern wurde ein DFG-Forschungsprojekt im Rahmen des DFG-Schwerpunktprogrammes "Objektorientierte Modellierung in Planung und Konstruktion" (DFG-Schwerpunkt SP-694) mit dem folgenden Thema bearbeitet: *„Integration raum- und bauteilorientierter Daten in der Gebäudeplanung in einem zentralen Objektmodell“*, von P. Heck [30] [31].



Das zentrale Objektmodell Grundriß

Abbildung 2.3: Die Integration raum- und bauteilorientierter Daten in der Gebäudeplanung durch ein zentrales Objektmodell für Grundriß [31].

Ziel des o. g. Forschungsprojektes ist die Entwicklung eines zentralen Objektmodells für die Leistungsphase des Vorentwurfs und der Entwurfs-, Genehmigungs- und

Ausführungsplanung. Die Vorplanungsphase wird dabei als raumorientiertes Gedankenmodell angenommen, die Entwurfs-, Genehmigungs- und Ausführungsplanung als bauteilorientiertes Modell aufgefaßt. Dieses Objektmodell muß geeignete Partialmodelle für die raumorientierte und bauteilorientierte Bearbeitung eines Grundrisses enthalten.
Die wesentliche Forderung an das zentrale Objektmodell ist, daß raumorientierte und bauteilorientierte Daten gemeinsam enthalten sind und bearbeitet werden können.

Fragen wie:

- Welche Datenstrukturen und Funktionen repräsentieren die Objekte bezüglich ihrer Abbildung im zentralen Objektmodell?
- Welche Objektklassen sind für die integrale Bearbeitung von Räumen und Bauteilen notwendig?

- Welche Klassenhierarchien und Vererbungsmechanismen sind für die benötigten Objekte geeignet?

sind zu beantworten.

Das zentrale Objektmodell wird mit Hilfe objektorientierter Methoden OOAD (Object Oriented Analysis and Design) modelliert und in einem weiteren Schritt mit einer objektorientierten Programmiersprache OOP (Object Oriented Programming) implementiert.

Mit Hilfe der bisherigen Erfahrungen in den objektorientierten Technologien des Fachgebiets an der Universität Kaiserslautern entstanden Diplom- und Studienarbeiten auf dem Gebiet der Modellierung in der Tragwerksplanung von Gebäuden [14] [42] und der Kopplung mit anderen CAD-Datenmodellen im informationstechnischen Gesamtmodell der Bauplanung.

Ein Problem, daß sich durch die Anwendung verschiedener CAD-Systeme ergeben hat, ist der Datenaustausch. In der Bauplanung operiert heute der CAD-Anwender mit Objekten wie Wänden, Stützen und Trägern, nicht mehr mit Punkten und Strichen. Die bisher entwickelten Schnittstellen und Austauschformate für Datenaustausch erfüllen nicht mehr ihre Aufgaben. Es gibt heute Initiativen, die zur Standardisierung und Verbesserung des Datenaustauschs zwischen den verschiedenen CAD-Systemen beitragen sollten, wie STEP (Standard for Exchange of Product Model Data, bzw. ISO 10303) und IAI (Industry Alliance for Interoperability) [26]. Andere Forschungen beschäftigen sich mit Ansätzen, die zur Schaffung von durchgängigen rechnergestützten Planungs-, Fertigungs- und Nutzungsprozessen beitragen sollen. Ein erfolgversprechender Ansatz ist die Anwendung der Produktmodellierung im Bauwesen zur Erstellung digitaler Bauwerksmodelle, den Rüppel in seiner Forschung bearbeitet hat [66].

In den Arbeiten von Niestroy [51] und Rüppel [65] wurde das Tragwerk in zwei objektorientierten Teilproduktmodellen beschrieben. Das Tragwerksmodell beschreibt

die Gebäudedaten hinsichtlich der tragenden Bauteile sowie ihre Topologie. Das statische Modell beschreibt die statisch relevanten Strukturdaten des Tragwerks. Ziel war es, aus den Entwurfsplänen des Architekten, die in Form einer CAD-Zeichnung vorliegen, Informationen für die Tragwerksanalyse in objektorientierter Form (Teilproduktmodelle) zu generieren. Niestroy und Rüppel [51] [65] mußten daher zwangsläufig auf die Datenmodelle der benutzten CAD-Systeme (Nemetschek, RIB, Hochtief Software) eingehen. Sie haben im wesentlichen Datenaustauschprobleme der drei Systeme auf Dateiebene behandelt.

Betrachtet man die Tragwerksplanung von Gebäuden, so sind Einzelschritte als Teilprozesse der gesamten Planungsprozesses zu erkennen (Abbildung 2.2).

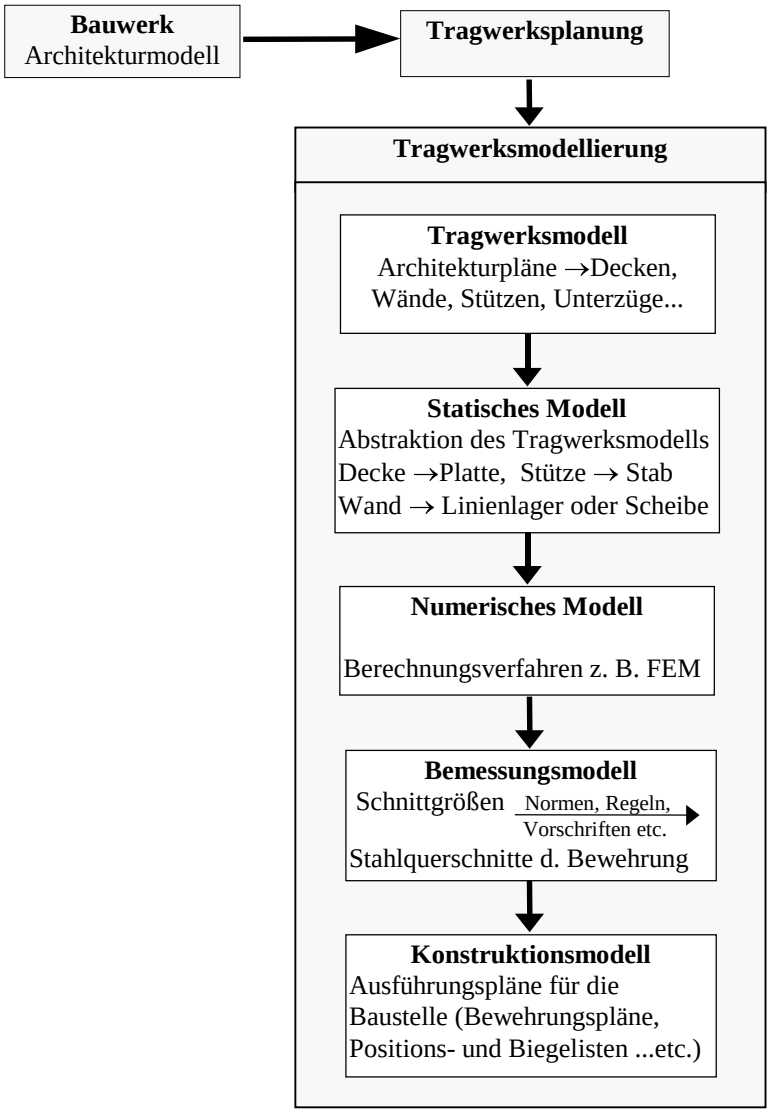


Abbildung 2.2: Modellierung der Teilprozesse in der Tragwerksplanung

In der Tragwerksplanung verlangt das Entwerfen einer Tragstruktur ein möglichst umfassendes Verständnis des Tragverhaltens eines Bauwerks, um eine Möglichkeit

der Lastabtragung aus vielen möglichen Lösungsvarianten auswählen und modellieren zu können. Die im Datenmodell des Tragwerks enthaltenen Informationen sollen daher detaillierte Kenntnisse über die Tragstruktur und das Tragverhalten ermöglichen. Dieses soll dabei helfen, die notwendigen Daten und Bemessungsparameter unabhängig von dem später ausgewählten Rechenverfahren zu generieren und visuell darzustellen.

Im Rahmen dieser Arbeit wird ein solches objektorientiertes Datenmodell entwickelt. Die Anwendung der objektorientierten Entwicklungs- und Modellierungstechnologie (OOM = Object Oriented Modeling) von der Analyse über das Design bis zur Programmierung soll als konzeptuelle Denkweise für computergestütztes Entwerfen in der Tragwerksplanung von Gebäuden realisiert werden.

Ein wesentliches Ziel dieser Dissertation ist die **Erstellung eines objektorientierten Datenmodells** für die Tragwerksplanung von Gebäuden. Im weiteren wird die **Entwicklung eines CAD-Systems** zur Realisierung und Austesten des erstellten Datenmodells vorgenommen. Das bedeutet, daß **kein Basis-CAD-System** (sogenannt Low-cost-CAD-System) wie bei den o. g. Arbeiten von Niestroy [51] und Rüppel [65], und **keine Standardsoftware als Basis-Modellierer** für die Realisierung der CAD-Funktionalitäten, wie Kopieren, Verschieben, Identifizieren, Fangen, Zoomen usw. verwendet wird (siehe Kapitel *5 u. 6*). Dazu werden nur selbstentwickelte einfache geometrische 2D- und 3D-Objekte (Punkt, Linie, Polygon, Prisma etc.) mit eigenen Attributen als Grundelemente zur rechnerinternen Beschreibung der Geometrie eingesetzt. Dabei ist ein eigenes objektorientiertes Datenmodell für einen objektorientierten CAD-Kern mit grafischem Grundsystem entstanden. Dieser stellt die Funktionalitäten eines CAD-Systems sowie die Modifizierung von grafischen Objekten zur Verfügung. Es wurde versucht, einen selbständigen CAD-Kern gemäß der objektorientierten Technologie zu entwickeln.

Zum Forschungsgebiet gehören auch die Modellierung von Arbeitsvorgänge, Konkurrierende Technik (Concurrent Engineering) und Kommunikation zwischen den verschiedenen Datenmodellen. Die Verfügbarkeit leistungsfähiger Kommunikations-infrastrukturen ist eine Grundvoraussetzung für den Einsatz von Systemen zum Concurrent Engineering. Diese Systeme ermöglichen den effizienten und insbesondere zielgerichteten Austausch von Produktdaten und Projektinformationen. Dazu werden Methoden und Modelle untersucht und weiterentwickelt. Relevante Aspekte wie die Organisationsstruktur, die Projektphasen, die Verfügbarkeit von Projektressourcen und die Verteilung von Anwenderrollen werden dabei berücksichtigt. Die von R. J. Sherer vorgestellten Arbeit [71] beschäftigt sich mit dem verteilten, gleichzeitigen Planen (Concurrency Unterstützung) und der Verwaltung von Projektdaten im Internet. Ziel dabei ist, die Abhängigkeit und Konsistenz der Daten mit paralleler Bearbeitung zu erreichen.

Ein Schwerpunkt der aktuellen Forschungsarbeiten ist die organisations-übergreifende Kommunikation für die bestehenden Techniken, wie etwa World Wide Web (WWW), die in einem einheitlichen Kommunikationsparadigma abgebildet und an komplexe Informationssysteme eines Unternehmens angebunden werden können. Die entwickelten Resultate werden in einem Anwendungsszenario evaluiert, welches die Phasen der Planung, der Bauausführung und der Gebäudeverwaltung (Facility Management) umfaßt [29].

Zur Lösung der Verwaltungs- und Kommunikationsaufgaben bei großen und vielfältigen Informationsmengen in einem Netzwerk wurden in den letzten 5 Jahren Ansätze in Form eines verteilten, objektorientierten Computersystems entwickelt und realisiert. So werden neue Begriffe in der Literatur wie „Kooperative Arbeit“ eingeführt. Solche Begriffe werden heute – nicht uneingeschränkt - auf die Problembereiche der Bauplanung übertragen. In der rechnergestützten Tragwerksplanung wird beispielsweise „Kooperative Tragwerksplanung“ in der Forschung untersucht, wie z.B. die Arbeit vom D. Bretschneider [9] zur Modellierung rechnergestützter, kooperativer Arbeit in der Tragwerksplanung. Dabei ist die verteilte Datenverarbeitung (Distributed Computing) im Netzwerk die grundlegende Technologie der kooperativen Arbeit.

Das im Rahmen dieser Arbeit entwickelte objektorientierte Datenmodell für die Tragwerksplanung bildet eine geeignete Basis für eine rechnergestützte, verteilte, kooperative Tragwerksplanung.

Der folgende Abschnitt gibt einen Überblick über die aktuellen Technologien, die für die Realisierung verteilter Datenverarbeitung eingesetzt werden.

2.3 Verteilte Datenverarbeitung

Früher wurde vorausgesetzt, daß die gesamte Datenverarbeitung auf einem Computer abläuft. Die Verteilung von Prozessen und Daten wurde nicht respektiert, [40], d. h. die Prozesse laufen auf dem Computer ab, auf dem auch die Daten gespeichert sind. In der Zeit der Netzwerke ist das nun selten der Fall, bei dem solche Grundarchitekturen respektiert werden sollen [8] [4], wie beispielsweise:

- Client/Server Architektur

- Verteilte Objekte (Distributed Objects) Architektur

Die meisten netzwerkorientierten Softwareanwendungen werden so konzipiert, daß sie mehr oder weniger die Netzwerkeigenschaften zusammen kombinieren und sich aus netzwerkorientierten Softwarekomponenten zusammensetzen, die von spezialisierten Softwarefirmen als fertige Bausteine angeboten werden.

Durch Unterstützung von starken Softwareherstellern sind in den letzten Jahren zwei Architekturen für verteilte Verarbeitung in der Netztechnologie entstanden:

- Der neue Standard CORBA (Common Object Request Broker Architecture) ist eine Architektur und Schnittstelle für Verteilte Objekte (Distributed Objects), der von OMG (Object Management Group) entwickelt wurde und seit 1990 existiert.

Das Ziel war, ein gemeinsames Konzept für die Interaktion zwischen verteilten Objekten zu entwickeln. Kommerzielle Versionen sind seit 1992 verfügbar.

- DCOM (Distributed Component Object Model) ist eine spezifische Verteilungslösung der Firma Microsoft, die 1996 zum ersten Mal als beta-Version erhältlich war. Dabei handelt es sich um ein Protokoll, das es den Softwarekomponenten ermöglicht, direkt über ein Netz in einer zuverlässigen, sicheren und rationellen Weise zu kommunizieren. Dem Architekturmodell von DCOM entspricht das Produkt OLE2 (Object Linking and Embedding), ebenfalls von Microsoft. Daher wurde DCOM vorher "*Network OLE*" genannt. DCOM ist passend für Anwendungen, die vollständig auf der Microsoft-Plattform laufen und im Rahmen eines Bürosystems Dokumente verarbeiten und z. B. keine Vererbung brauchen. Für diese Fälle kann DCOM eine gute Wahl sein.

CORBA-Produkte gibt es von über 20 verschiedenen Herstellern (AT&T, Digital, IBM, Microsoft, Novel, Oracle etc.). Sie unterstützen Microsoft und Nicht-Microsoft-Betriebssysteme. CORBA ist ein exzellenter Mechanismus, um Microsoft-Rechner mit UNIX-Servern zu verbinden. Das Architekturmodell von Microsoft COM (Component Object Model) kam später auf den Markt, so daß CORBA mehr Zeit hatte, um auszureifen.

Außerdem gibt es eine große Anzahl von Firmen, die CORBA ORBs (Object Request Broker) entwickeln. Diese Art von Wettkampf verbessert beide CORBA- und DCOM-Lösungen. DCOM kann ebenso parallel zu CORBA benutzt werden. Die Frage ist also nicht immer, ob man DCOM oder CORBA benutzen sollte.

CORBA erlaubt es einer Anwendung, eine Operation von einem Verteilten Objekt (Server) anzufordern, das diese Operation dann ausführt und die Ergebnisse an die Applikation zurückgibt in einer transparenten Weise, unabhängig von der Plattform, vom Betriebssystem oder von lokalen Überlegungen. Die Anwendung kommuniziert mit dem Verteilten Objekt, das im Moment die Operation ausführt.

Auf ein verteiltes Objekt kann man von überall auf dem Netzwerk zugreifen. Wo das verteilte Objekt abgelegt ist, ist für den Benutzer unerheblich. Ein verteiltes Objekt kann seinen Benutzer mit einer Vielzahl verwandter Möglichkeiten versorgen. Dies ist die grundsätzliche Client/Server-Funktionalität, bei der ein Client eine Anfrage an einen Server schickt, und der Server dem Client antwortet. Daten können von dem Client auf den Server übertragen werden und stehen im Zusammenhang mit einer bestimmten Operation eines bestimmten Objektes. Dann werden Daten in Form einer Antwort an den Client zurückgegeben.

Ein verteiltes Objekt ist nicht in jedem Fall ein CORBA-Objekt. Ein CORBA-Objekt ist ein Objekt, das verschiedene Regeln beachtet und auf das über ein bestimmtes Protokoll zugegriffen werden kann. Ein CORBA-Objekt ist nur in den meisten Fällen, aber nicht zwangsläufig verteilt. Es kann auch ein C++-Objekt sein. Ein CORBA-Objekt ist eine Spezialisierung eines verteilten Objektes. Damit ein verteiltes Objekt ein CORBA-Objekt sein kann, muß es bestimmte Eigenschaften haben. Das Objekt kann in einer Vielzahl von Programmiersprachen ausgeführt sein.

Verteiltes Rechnen kann es bestimmten Anwendungen erlauben, schneller zu laufen, wenn sie sich in einem einzelnen Prozeß befinden. In diesen Fällen spiegelt die Verteilung nicht die wirkliche weltweite Verteilung wider, es erlaubt jedoch, den Vorteil zusätzlicher Rechnerkapazitäten zu nutzen.

Es gibt viele Gründe, warum man Applikationen mit verteilten Objekten entwickeln sollte [40]:

- Verteilte Objekte können benutzt werden, um Informationen auf mehrere Applikationen oder Benutzer zu verteilen.
- Verteilte Objekte können Aktivitäten auf verschiedenen Rechnern synchronisieren.
- Verteilte Objekte können benutzt werden, um die Performance, verbunden mit einer bestimmten Aufgabe, zu verbessern.
- Verteilte Objekte können benutzt werden, um Applikationen, die auf einem PC laufen, mit Informationen zu verbinden, die von UNIX-Prozessen verwaltet werden.
- Verteilte Objekte können benutzt werden, um allen Personen an verschiedenen Orten zu ermöglichen, etwas zu einem bestimmten Geschäftsvorgang beizusteuern.
- Verteilte Objekte können es ermöglichen, einen Geschäftsvorgang zu modifizieren oder neu auszuführen, ohne die Applikationen zu verändern, die verteilte Objekte benutzen.

Verteiltes Rechnen kann dabei helfen, bestimmte Geschäftsprobleme zu lösen. Im heutigen Geschäftsleben ist die Verteilung von Geschäftsvorgängen üblich. Eine Firma z. B. sitzt an einem Ort, der Verkauf an einem anderen, und die Marketing-Abteilung an einem dritten.

Die Vorteile beim Einsetzen des Verteilten Rechnens als neue Architektur und Schnittstelle können dazu beitragen, neue Konzepte zu entwickeln und Lösungen zu den vielen Problemen des Informationsaustausches, der Kommunikation und Integration im Bauwesen zu finden. Der Einsatz solcher Systemarchitekturen im Bauwesen ist immer noch gering und meistens nur in der Forschung zu finden.

3. Zielsetzung und Modellkonzept

3.1 Allgemeine Zielsetzung

Ziel dieser Arbeit ist die Entwicklung eines objektorientierten CAD-Datenmodells für die Tragwerksplanung und die Integration in das informationstechnische Gesamtmodell zwischen Planung und Nutzung von Gebäuden.

Folgende Aufgaben werden dabei bearbeitet:

1. Erstellung eines objektorientierten Datenmodells für die Unterstützung der Tragwerksplanung von Gebäuden. Dieses bedeutet, daß das Datenmodell mit Hilfe der Objektorientierten Vorgehensweise von der Analyse über das Design zur Implementierung am Rechner entwickelt werden soll (siehe Kapitel *5*).

2. Realisierung und Test des entwickelten Datenmodells (siehe Kapitel *6*). Zu diesem Zweck soll ein CAD-System *MvCad* mit einer objektorientierten grafisch-interaktiven Benutzeroberfläche (GUI) und mit den erforderlichen CAD-Funktionalitäten entworfen, realisiert und ausgetestet werden.

Erkenntnisse aus früheren Arbeiten zeigen, daß eine große gesamte gemeinsame Datenbasis für das Bauwesen nicht zu erreichen ist (mehrere Versuche: ICES [62], ISB [36]). Daraus sind viele Informationsinseln entstanden. Die Verbindung zwischen den Informationsinseln ist nicht systematisiert. Das Erstellen eines Datenmodells jeweils für die einzelnen Planungsphasen in der Bauplanung zeigt sich als erforderlich.

Bauteile werden in dem in dieser Arbeit entwickelten Tragwerksmodell nicht als rein geometrische Elemente, sondern als Bauobjekte mit eigenständiger Information konzipiert, die über die Beschreibung der Geometrie hinausgeht. Die neue objektorientierte Entwicklungs- und Modellierungstechnologie (OOM) (siehe Kapitel *4*) soll dabei als konzeptuelle Denkweise beim computergestützten Entwerfen in der Tragwerksplanung realisiert werden.

Der Vorteil des in dieser Arbeit vorgestellten Tragwerksmodell liegt genau darin, daß nur die signifikante Informationen des Bauteils (wie Geometrie, Material und Tragverhalten) anhand der interaktiven grafischen Benutzerschnittstelle in der Ebene oder ggf. im Raum definiert werden. Mit Hilfe dieser Informationen wird dann das interne digitale Modell dieses Bauteils selbständig in dem gesamten Datenmodell des Tragwerks aufgebaut.

Dabei soll die Frage nach der Integration mit den einzelnen Datenmodellen in dem informationstechnischen Gesamtmodell der Bauplanung beantwortet werden. Dieses wird hier durch die Möglichkeit erfolgen, Daten in Form eines neutralen Informationsprotokolls zu lesen und schreiben bzw. mit anderen Datenmodellen

auszutauschen (Abbildung 3.1). Weiterhin soll die Integration prototypisch mit dem in [31] beschriebenen objektorientierten Datenmodell für die Gebäudeplanung untersucht und bewertet werden.

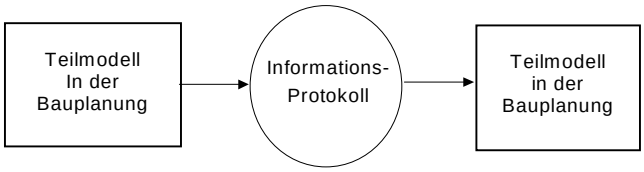


Abbildung 3.1: Integration des Tragwerksmodells in das Gesamtmodell

Anhand des neutralem Informationsprotokoll können die Bauteilobjekte von anderen CAD-Systemen importiert oder zu anderen Rechensystemen exportiert werden, was die Integration in das informationstechnische Gesamtmodell der Bauplanung ermöglichen kann. Standardisierte Austauschformate wie beispielsweise STEP (Standard for Exchange of Product Model Data, bzw. ISO 10303) oder IAI (Industry Alliance for Interoperability) können dann mit zusätzlichem Implementierungsaufwand zur Verfügung gestellt werden.

Ein wesentlicher Aspekt dieser Arbeit ist die Antwort auf die Frage der Anwendbarkeit der objektorientierten Methode bei der Softwareentwicklung im Bauwesen und die Integration des entwickelten Tragwerksmodells in das Gesamtmodell der Bauplanung. Im folgenden Abschnitt werden die Konzepte und Axiome zur Umsetzung dieser Entwicklungsmethode auf das im Rahmen dieser Dissertation entwickelte Datenmodell erläutert.

3.2 Konzepte und Axiome

- Das Tragwerksmodell besteht aus Objekten der Tragwerksplanung wie: Stütze, Platte, Unterzug, Linienlager, Aussparung, ..etc. (Abbildung 3.2).

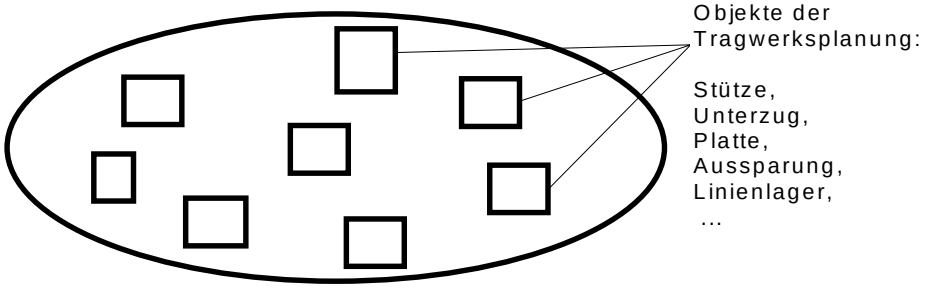


Abbildung 3.2: Problembereich des Tragwerksmodells und seine Objekte

- Jedes Objekt besitzt vollständige Informationen über Geometrie und baubezogene Eigenschaften, sowie die dazu nötigen Funktionen. Die objektspezifischen Daten (z.B. Material) und Eigenschaften (z.B. Methoden) sind in jedem Objekt verborgen.

- Jedes Objekt hat Voreinstellungswerte (default-Werte) zur Vervollständigung bei der Erzeugung, die jeder Zeit modifizierbar ist.

- Jedes Objekt besitzt die Fähigkeit zur Interaktion, d.h. Daten und Attribute durch CAD-Funktionalität zu ändern.

- Ein Objekt hat die Fähigkeit, mit den anderen Objekten zu kommunizieren. Dies geschieht durch Nachrichten senden und empfangen.

- Jedes Objekt hat eine Importfunktion zum Einlesen aus anderen CAD-Systemen. Ein Beispiel dafür ist das Importieren über die Postdateien (**.POS*) (des Systems *MicroFe der Firma mb-Programme Software im Bauwesen GmbH*).

- Jedes Objekt erzeugt ein Protokoll für die weitere Bearbeitung mit anderen Systemen (Abbildung 3.3).

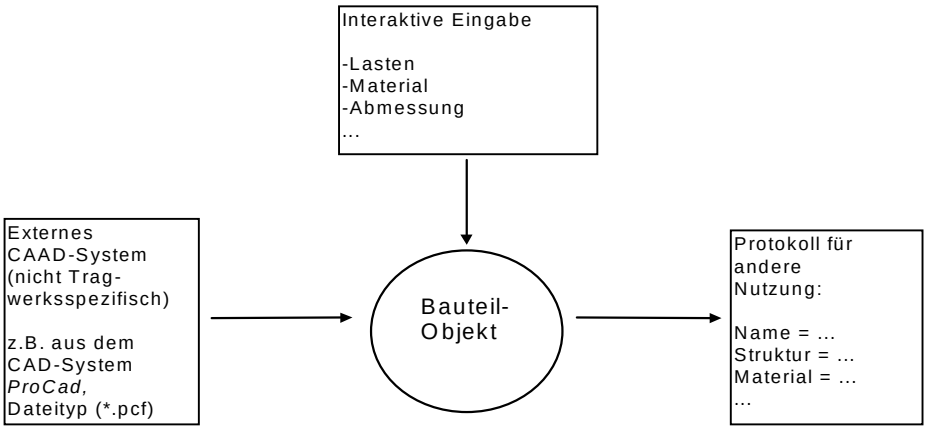


Abbildung 3.3: Informationsfluß der Bauobjekte

Das Entwerfen einer Tragstruktur verlangt vollkommenes Verständnis des Tragverhaltens eines Bauwerks, um eine Möglichkeit der Lastabtragung aus vielen möglichen Lösungsvarianten auszuwählen und zu modellieren.

Die im digitalen Datenmodell der Bauobjekte enthaltenen Informationen sollen daher detaillierte Kenntnisse über die Tragstruktur und sein Tragverhalten ermöglichen. Das soll später dabei helfen, die notwendigen Daten und Bemessungsparameter, unabhängig von dem später ausgewählten Rechenverfahren, zu generieren.

Im Rahmen dieses Forschungsprojekts kommen Tragwerke in Hochbauten aus Stahlbeton in Betracht. Das entwickelte Modell soll auch dazu dienen, weitere

Bauobjekte mit beliebigen Eigenschaften (Geometrie, Material und Tragform) modellieren zu können, damit das Datenmodell zu erweitern.

3.3 Abgrenzung

3.3.1 Abgrenzung des Problembereichs

Der im Rahmen dieser Arbeit betrachtete Problembereich umfaßt die Tragwerksplanung von Gebäuden des Massivbaus als Teilprozeß im gesamten Bauplanungsprozeß. Bauobjekte des Holzbaus oder Stahlbaus sind nicht Gegenstand der Betrachtung.

Oft wird das Tragwerksmodell mit einem Gebäudemodell integriert. Daher werden die Bauteile auch zweimal abgebildet, einmal im Gebäudemodell und das zweite Mal im Tragwerksmodell. Dabei ist das Zurückgewinnen der Informationen des Gebäudemodells nach der Übertragung in das Tragwerksmodell und dem Vornehmen gewisser Änderungen an den Tragwerksobjekten problematisch und kompliziert [51].

Eine wichtige Aufgabe ist es also, eine Schnittstelle für ein neutrales Format zu definieren, die eine softwareunabhängige Übergabe der Daten gewährleistet. Eine berechtigte Frage dabei wäre, wie groß der Unterschied zwischen dem Aufwand für das Modellieren einer Schnittstelle für den vollständigen Informationsaustausch zwischen Datenmodellen und dem Aufwand für das Modellieren eines zweiten Datenmodells ist?

Folgende Abgrenzungen des im Rahmen dieser Arbeit entwickelten Tragwerksmodells lassen sich bezüglich dem Problembereich formulieren:

- Das entwickelte Tragwerksmodell stellt ein Teilproduktmodell in der Tragwerksplanung dar und nicht ein integriertes Datenmodell für die gesamte Tragwerksplanung.

- Dieses Tragwerksmodell enthält Bauteile, **die die Tragstruktur eines Gebäudes des Massivbaus beschreiben und keine Bauobjekte des Holzbaus oder Stahlbaus**, wie: Stütze**,** Unterzug**,** Linienlager**,** Plattenbereich**,** Aussparung**,** Bettung**,** Belastung. Diese einzelnen Positionen sind prototypisch modelliert. Das heißt, es werden **nicht alle möglichen vorkommenden Typen jedes Bauteils in einem Gebäudemodell in dem Tragwerksmodell abgebildet**. Es wird beispielsweise nur ein Unterzug mit einem rechteckigen Querschnitt und nicht alle Typen des Querschnitts modelliert.

  Die Tragstruktur eines Gebäudes kann durch die oben genannten Bauteile, die in jedem Geschoß vorkommen, für die am meisten in der Praxis vorkommenden Fälle beschrieben werden. Andere Bauteile wie Treppe, Fundament oder Dach sind zwar im Datenmodell nicht abgebildet, können aber auch analog leicht addiert werden.

Auf der anderen Seite kann es beispielsweise für eine Decke im Gebäudemodell zahlreiche verschiedene Arten geben, wie rechteckige und dreieckige Decken, Decken mit unterschiedlichen Dicken oder Lagerungsarten wie Kragdecken, die ein- oder beidseitig gelagert sind. Diese werden im Tragwerksmodell reduziert, in dem sie mit einem polygonal berandeten Plattenbereich mit möglichen verschiedenen Dicken zusammengefaßt werden. Dabei unterscheiden sich nur die Randbedingungen auf den Seiten der Decke, die durch Unterzug oder Linienlager mit verschiedenen Freiheitsgraden zu bestimmen sind.

- Mit dem entwickelten Datenmodell kann der Tragwerksplaner ein digitales dreidimensionales Tragwerksmodell mit Tragwerksobjekten erstellen und durch ein neutrales lesbares Protokoll an andere Informationsmodelle weitergeben. Dabei können die Tragwerksobjekte durch CAD-Funktionalitäten und interaktives Konstruieren erzeugt, oder von einem anderen Datenmodell importiert werden.

- Der Informationsfluß aus diesem Tragwerksmodell an die nachfolgenden Teilmodelle in der Tragwerksplanung, wie z.B. an das statische Modell, erfolgt durch **ein neutrales lesbares Protokoll und keinen Datenaustausch über bekannte Austauschformate wie DXF, STEP, etc.** Dieses wird in Form einer Datei im Textformat (ASCII-Format) ausgegeben, und beinhaltet ausdrückliche Informationen über die Tragwerksobjekte und ihre Referenzobjekte. Die Implementierung einer standardisierten Schnittstelle zum Datenaustausch wie DXF oder STEP, dem sogenannten Pre- und Postprozessor, wäre nur noch eine Fleißarbeit.

  Ein Rückgewinnen der Informationen für ein Gebäudemodell ist durch das neutrale Protokoll nicht möglich, es kann nur mit Informationsverlust gebunden sein.

3.3.2 Abgrenzung des Datenmodells

Folgende Merkmale beschreiben die Eigenschaften des CAD-Datenmodells:

- **Kein abstraktes Datenmodell für die rechnerinterne Darstellung**, d.h. kein Drahtmodell (B-Rep) und kein Volumen- oder Festkörpermodell (CSG) oder Flächenmodell. Es werden einfache geometrische 3D- und 2D-Objekte (Punkt, Linie, Polygon, Prisma etc.) mit Geometrieattributen als Grundelemente zur rechnerinternen Beschreibung der Geometrie eingesetzt.

- **Kein Einsatz eines Low-cost-CAD-Systems oder einer Standard-Software** als Basis-Modellierer für die Realisierung der CAD-Funktionalitäten, wie Kopieren, Verschieben, Identifizieren, Fangen, Zoomen usw. sondern **es wird im Rahmen dieser Dissertation ein eigenes objektorientiertes CAD-System mit objektorientierten Datenstrukturen (Klassen) und Datenbasis entwickelt**, das diese CAD-Funktionalitäten u. a. im Rahmen des objektorientierten Konzeptes realisiert.

  Dabei besitzen die Objekte des Datenmodells in ihrer Datenstruktur neben den verborgenen Objektdaten wie Geometrie und bauspezifische Informationen (wie

Material), die CAD-Operationen wie Kopieren, Verschieben, Identifizieren, Fangen usw., die es erlauben, Objektdaten und -zustand zu modifizieren.

- Es wird auf die praktische Vorgehensweise bei der Organisationsstruktur der Informationen im Datenmodell und Daten eingegangen, und dafür werden klare Schnittstellen definiert. Dabei werden **Modellkonzepte für die Abstraktion von Schnittstellen** als Lösungsansätze für Probleme vorgestellt und anschließend prototypisch realisiert, die in der Praxis beim Konzipieren und Entwickeln von objektorientierten CAD-Systemen vorkommen. Beispiele dafür sind:

  ⇒ Trennung zwischen dem Modell und den Sichten (Views) auf dieses Modell durch ein Dokument -View-Architekturdesign,

  ⇒ Trennung zwischen der Darstellung der Objekte während der Konstruktion (Eingabesequenz) und der endgültigen Präsentation, und zwar die sogenannte Builder-Methode (Builder Design pattern),

  ⇒ Trennung zwischen Verwaltung und Organisation der grafischen und bauspezifischen Daten.

- Es wird **keine fertige Klassenbibliothek** für das Datenmodell eingesetzt, weder für grafische Elemente noch für die Mengenverwaltung. Ausnahme war die Realisierung der objektorientierten grafisch-interaktiven Benutzeroberfläche. Dafür kamen die Sichtklassen (View classes) der MFC (Microsoft Foundation Class Library) zum Einsatz.

  Als Beispiel dafür wird auf die auf dem Markt bekannte STL (Standard Template Library) für Mengenverwaltungsklassen (Listen, Tree, Set, Array usw.) verzichtet, weil Kompatibilitätsprobleme bei der ersten Implementierung der für die GUI eingesetzten MFC aufgetaucht sind, die möglicherweise bei der neuesten Version inzwischen gelöst wurden.

  Die Verwaltung der Daten erfolgt durch die im Rahmen dieser Arbeit entwickelten Mengenverwaltungsklassen (Containerklassen), die die Objekte verschiedener Klassen verwalten. Für die Datenhaltung wurden dynamische, doppelt verkettete Listen entworfen und in die Datenstruktur der Vewaltungsklassen eingefügt.

- **Eigenes Speicherkonzept**. Die Objekte des Tragwerksmodells werden mit ihrer Datenstruktur und ihren Verweisen (Zeiger) gespeichert (Kapitel *6*)**. Es wird keine relationale oder objektorientierte Datenbank** für die Speicherung der Daten eingesetzt.

  Das Konzept ist deshalb für das Speichern und Lesen von Objekten des entwickelten Datenmodells geeignet, weil das Modell keine komplizierten verschachtelten Beziehungen zwischen den Objekten und Klassen beinhaltet.

  Die Objekte im Datenmodell werden auf dem Massenspeicher in Dateien mit Textformat (ASCII-Format) gespeichert, so daß die Objektinformationen für jeden lesbar sind. Somit stellt das Speicherformat ein neutrales Protokoll dar, das auch dem Datenaustausch dienen kann.

Das für das Datenmodell entwickelte Speicherkonzept dient dazu:

⇒ Datenstruktur der Objekte in Textformat zu speichern,

⇒ Verweise jedes Objektes auf andere Objekte mitzuspeichern,

⇒ die Speicherungsdatei nur einmal öffnen zu müssen, um alle Objekte mit ihren Verweisen zu lesen.

Bei dem Speicherkonzept wird auf die Organisation des Lesens und Schreibens der Daten eingegangen. Die Datensicherung geschieht über ein eigenes Textformat (ASCII-Format).

3.4 Vorgehensweise

Der gesamte Entwicklungsprozeß erfolgt mit Hilfe der objektorientierten Technologie (Kapitel *4*). Das objektorientierte Konzept ist eine neue Denkweise für das Entwickeln und Modellieren (Object Oriented Modeling = OOM) von Softwareprodukten und keine Programmiertechnik. Sie ist bis zur letzten Phase unabhängig von einer Programmiersprache.

Der Weg dieser Entwicklung besteht aus folgenden Arbeitsschritten:

3.4.1 Analysieren des Problemraums (eine objektorientierte Analyse)

Die Tragstruktur, seine Komponente und deren Eigenschaften, sowie die Beziehung untereinander sollen hinsichtlich der Geometrie und des Tragverhaltens analysiert werden. Dabei werden Identifikation und Einschränkung des Problembereichs erzielt. Die häufig vorkommenden Bauteile im Hochbau aus Stahlbeton sollen das Thema der Analyse sein. Es wird hier von der objektorientierten Analyse (Object Oriented Analysis = OOA) [7] und Rumbaugh [63] als Analysemethode gesprochen, welche die Anforderung an das neue System aus dem Sprachgebrauch des jeweiligen Problembereichs (Tragwerksplanung) und aus der Perspektive von Objekten ableitet (Kapitel *4*)..

Die Erfahrung mit dem Einsatz der objektorientierten Analyse und von CASE-Tool (Computer Aided Software Engineering) wird herangezogen (Kapitel *4*)..

3.4.2 Entwerfen eines OO-Informationsmodells

Um ein objektorientiertes Informationsmodell zu erstellen, werden die Bauteile des Tragwerks (Komponente) klassifiziert und durch verschiedene bauteilbezogene Objekttypen mit Attributen und Methoden modelliert. Dabei sollen die Objekte des Konstruktionssystems in der vollständigen Geometrie und in den Funktionen entworfen werden (Kapitel *5*)..

Das daraus entstehende Objektmodell soll die Erweiterbarkeit und Wiederverwendbarkeit gewährleisten. Dadurch wird ermöglicht, weitere Softwarekomponenten einzubinden, und andere Teilproduktmodelle in das System zu integrieren.

Hier ist ein Iterationsverfahren mit der vorherigen Analyse zu erwarten, um andere Objekte und Beziehungen zu berücksichtigen. Dadurch soll eine bessere und vollständige Modellierung des Problemraums erreicht werden.

Die Ergebnisse der Analyse werden verwendet, um zu einem technischen Systementwurf zu gelangen, welcher anschließend in der Programmiersprache C++ implementiert werden kann.

Die Vorbereitungsphase zum Binden zwischen der inhaltlichen Beschreibung im Rahmen des Fachkonzepts und der Umsetzung in einer konkreten Implementierung wird in der Literatur als die Phase des objektorientierten Designs definiert (Object Oriented Design = OOD).

3.4.3 Realisierung durch Implementierung eines CAD-Systems

Ein wichtiges Werkzeug für die Präsentation der Tragstruktur ist die grafische Benutzeroberfläche (Graphical User Interface = GUI). Sie ermöglicht eine grafische Darstellung der Strukturelemente in unterschiedlichen Ansichten (z.B. 2D- und 3D-Ansichten). Der Benutzer kann außerdem mit dem System innovativ und interaktiv arbeiten (Kapitel *6*).

Die zu einem CAD-System gehörende Funktionalität, die in diesem System implementiert werden soll, erlaubt weiterhin, Erzeugung, Löschung und Änderungen an den Bauobjekten vorzunehmen.

Die Implementierung soll unter Microsoft Windows mit der Programmiersprache C++ als leistungsfähige objektorientierte Programmiersprache (Object Oriented Programming = OOP) [37], und in der Entwicklungsumgebung von Microsoft Visual C++, erfolgen. Dieses ist eine moderne, integrierte Arbeitsumgebung, die alle notwendigen Komponenten zur Entwicklung von Windows-Applikationen mit der Programiersprache C/C++ bietet.

Für die grafische Oberfläche wird die Document-View-Architektur von dem in der Entwicklungsumgebung integrierten Klassenbibliothek Microsoft Foundation Class Library (MFC) angewendet, die unterschiedliche Darstellungsklassen (View's) mit verschiedenen Eigenschaften erstellt. Damit kann die Datenbasis der Tragstruktur unterschiedlich, z.B. als 2D/3D-Ansichten oder als Texte, dargestellt werden.

4. Objektorientierte Technologien

In diesem Kapitel werden die Elemente und Begriffe der objektorientierten Technologien zunächst erläutert, und es werden die objektorientierten Modellierungsmethoden bei der Softwareentwicklung, sogenanntes Software-Engineering, von der Analyse über Design bis hin zur Implementierung charakterisiert. Dabei wird auf die objektorientierten Modellierungsansätze von Coad/Yourdon [11] und [12], Booch [7] und Rumbaugh [63] eingegangen. Wobei nicht alle Notationen beschrieben werden, sondern nur eine Übersicht über die Notation der OMT (Rumbaugh) am Ende dieses Kapitels.

Es wird auf die Bedeutung des Schlagwortes „**objektorientiert**“ eingegangen, das inzwischen zu jeder guten Software gehört. Neben der Einführung neuer Sprachmittel bedeutet das vor allem das Kennenlernen einer neuen Begriffswelt und ein neues Denken. Dabei scheint sich die Programmiersprache C++ als quasi etablierter Standard für einen Sprach-Hyprid durchzusetzen, der objektorientierte und nicht-objektorientierte Programmierung ermöglicht.

Im Rahmen dieser Arbeit werden die Notation und Darstellungsmöglichkeit nach der Objektmodellierungstechnik OMT (Object Modeling Technique) benutzt, die von James Rumbaugh für die objektorientierten Modellierungsmethoden bei der Softwareentwicklung von der Analyse über das Design bis hin zur Implementierung entwickelt wurde. Wobei nicht alle Feinheiten der Notation benutzt werden, sondern nur die, die zur Erläuterung der OO-Technologien und -Begriffe notwendig sind.

4.1 Einleitung

Die Entwicklung von Softwareprodukten hat sich in den letzten Jahren mit dem Übergang von klassischen Methoden zu OO-Methoden (objektorientierte Methoden) charakterisiert.

Ein historischer Überblick über die Entwicklung der Programmiersprachen ist in diesem Zusammenhang notwendig, um die Urväter und Vorgänger der heutigen OO-Programmiersprachen und OO-Methoden kennenzulernen, außerdem die Ziele aufzuspüren, welche bei der gesamten Entwicklungshistorie von Programmiersprachen und -konzepten verfolgt wurden.

In den fünfziger Jahren, mit der Erscheinung der ersten Rechner auf dem Markt, standen den Computerfreaks sagenumwobene 64 KByte Hauptspeicher zur Verfügung. Da diese Rechner der ersten Generation zudem recht träge arbeiteten, galt es als notwendig, speicherschonende und schnelle Programme zu entwickeln.

Es wurde aber schon recht bald erkannt, daß mit den zur Verfügung stehenden Programmiersprachen die Programme zunehmend unübersichtlicher und unleserlicher wurden. Große Änderungen an der Programmstruktur gab es bis vor ein paar Jahren nicht. Die bis zu diesem Zeitpunkt zur Verfügung stehende Entwicklungstechnik basierte ausschließlich auf der Lösung eines bestimmten Problems durch eine Funktion (Algorithmus), die das Gesamtsystem beschreibt. Diese Lösung wurde in immer kleiner werdende, feiner strukturierte Teilfunktionen zerlegt.

Dies führte zu Mehrfachverschachtelungen, so daß selbst die Schöpfer binnen einer Woche nicht mehr wußten, was ihre Programme bewirkten, geschweige denn, wie sie funktionierten. Programme mit Goto-Kaskaden, deren Verfolgung einer Dschungelexpedition gleichkommt, waren keine Seltenheit.

Da dies alles unzureichend war, ging man neue Wege. Das neu entstandene Konzept ist spätestens seit der Einführung moderner, prozeduraler Sprachen wie Pascal, Modula oder C jedem bekannt: **Die strukturierte Programmentwicklung**.
In der strukturierten Programmierung werden Unterprogramme und Module geschrieben, deren Kombination das Hauptprogramm ergibt. Der Unterschied zwischen einem Spaghettiprogramm und einem strukturiert programmierten war klar, die Vorteile waren deutlich.

Mit SIMULA 67 wurde die erste objektorientierte Programmiersprache zum Ende der 60er Jahre eingeführt, die Objekte, Klassen, Vererbung (Inheritance), und dynamische Erzeugung von Datentypen vorgesehen hat. Sie ist der Urvater objektorientierter Programmiersprachen, die am Anfang speziell für Simulationssysteme entworfen wurde. SIMULA 1 war eigentlich eine Simulationssprache, die spätere Version SIMULA 67 für allgemeine Zwecke wird jetzt einfach als SIMULA bezeichnet.

SMALLTALK war die nächste große und wichtige objektorientierte Programmiersprache, die auch Objekte, Klassen, Vererbung (Inheritance), effektive grafische Umgebung und starken Mechanismus zur dynamischen Erzeugung von Datentypen eingeführt hat. Danach folgte die nächste SMALTALK-basierte Generation der Programmiersprachen, sowie die, die SIMULA gefolgt sind, wie z.B. BETA.

In den Anfängen der 70er Jahre wurden die ersten integrierten objektorientierten Systeme bei der Firma XEROX im Rahmen des SMALLTALK-Systems konzipiert. Es wurde dabei auf dem Gebiet der Kommunikation zwischen Mensch und Maschine geforscht. Die daraus entstandenen Ideen wurden bei der Entwicklung der objektorientierten Programmiersprache SMALLTALK eingesetzt, die heute große Bedeutung besitzt. Diese Ideen wurden auch später in der Forschung auf dem Gebiet der künstlichen Intelligenz und Expertensystem-Shells aufgegriffen.

Objektorientierte Sprachen sind also schon relativ lange verfügbar, auch wenn sie zunächst nur ein Schattendasein führten. Ziel war es nun eine Sprache zu entwickeln, die so nah wie nur möglich an der Realität war, daß eine Umsetzung in die abstrakte Welt der Programmierung ohne größeren Denkaufwand zu realisieren war. Dies wurde erreicht durch die Entwicklung der **objektorientierten Programmierung.**

Der objektorientierte Ansatz verspricht für die Softwareentwicklung der neunziger Jahre ähnliche Bedeutung zu erlangen, wie die strukturierte Programmierung im vergangenen Jahrzehnt. Die Syntax der Formulierung ist leicht zu verstehen.

Bei der Systemanalyse waren objektorientierte Ansätze dagegen lange Zeit überhaupt nicht Gegenstand der Diskussion, und auch beim Design wurden bzw. werden bis heute vorwiegend die klassischen Ansätze verfolgt.

Erst Anfang der 90er Jahre fanden OO-Methoden (**O**bjekt**o**rientierte Methoden) eine stärkere Verbreitung durch die Veröffentlichungen von Coad/Yourdon [11] und [12], Booch [7], Rumbaugh [63] und anderen. Viele Autoren haben sich mit diesem Thema beschäftigt, so entstanden viele Modellierungsansätze für alle Phasen der Softwareentwicklung von der **OOA** (**O**bjekt**o**rientierte **A**nalyse) über **OOD** (**O**bjekt**o**rientiertes **D**esign) bis hin zur Implementierung **OOP** (**O**bjekt**o**rientierte **P**rogrammierung).

In den letzten Jahren gab es verschiedene Entwicklungen, die den Übergang von den klassischen Methoden zu OO-Methoden rechtfertigen bzw. erklären.

Der Ansatz der strukturierten Programmierung hat sich im Laufe der Zeit zum strukturierten Design und zur strukturierten Analyse weiterentwickelt. Es sollte deshalb nicht verwundern, daß der objektorientierte Ansatz als Konzept eine ähnliche Entwicklung durchläuft.

Der Entwurf eines Systems wird stark von der verwendeten Programmiersprache beeinflußt. Wenn strukturierte Programmiersprachen verwendet werden, muß der Systemdesigner auch strukturiert denken, was ihm in der Regel schwerer fällt, als objektorientiert zu denken.

Durch die Verfügbarkeit von OO-Sprachen kann der Systementwickler beim Design also auch mit ruhigem Gewissen OO-Methoden verwenden, ohne Probleme bei der Implementierung zu bekommen.

Heutige Softwareprodukte und Systeme unterscheiden sich von denen, die vor 10 bis 20 Jahren entwickelt wurden. Sie sind im allgemeinen größer und komplexer. Es wird später noch deutlich werden, daß sich der objektorientierte Ansatz gerade für große komplexe Systeme anbietet.

Die Benutzerschnittstelle ist eine wichtige Komponente jedes Systems und besitzt heute eine große Bedeutung. Der objektorientierte Ansatz ist auch hier mit seinem Nachrichtenkonzept dem strukturierten Ansatz vorzuziehen. Es wird oft von OO-Benutzerschnittstellen gesprochen, die es dem Systembenutzer ermöglichen, grafisch und interaktiv mit dem System zu arbeiten.

4.2 Grundlagen der objektorientierten Technologien

Die Begriffe in der objektorientierten Welt sind leider nicht einheitlich. Die Attribute in C++ heißen z.B. Members und in KEE Slots, wenn diese Attribute funktional sind und einen ausführbaren Quelltext beinhalten, heißen sie Member-Funktionen (Member Functions) in C++ und Methoden (Methods) in SMALLTALK und KEE.

Um Einblicke in die Begriffswelt der objektorientierten Technologien zu gewinnen, werden hier folgende wichtige Grundbegriffe zweckmäßig erläutert:

- Klassen und Objekte
- Beziehung zwischen Klassen und Objekten
- Polymorphismus (dynamisches Binden), virtuelle Methoden

4.2.1 Klassen und Objekte

Bei der Entwicklung im objektorientierten Modell sollen reale Objekte und ihre Beziehungen untereinander im Rechner nachgebaut werden.

Abstraktion ist ein gutes Mittel, Komplexität zu reduzieren. Dabei werden Dinge und Vorgänge zusammengefaßt oder mit einem Oberbegriff versehen. Auf diese Weise werden auch die komplexen Dinge und ihre Beziehungen auf das wesentliche reduziert und somit handhabbar.

Die höheren Programmiersprachen (High-Level-Language) haben eingebaute „build-in“ abstrakte Datentypen, alle objektorientierten Programmiersprachen ermöglichen benutzerdefinierte „user-defined“ abstrakte Datentypen.

Ein großer wesentlicher Fortschritt in der Geschichte der Programmiersprachen war die Möglichkeit, mehrere Daten zu einem Datenverbund zusammenzufassen. Somit wird ermöglicht, verschiedene zusammengehörende Eigenschaften in einem Datentyp zu bündeln. Die so entstandenen **Strukturen** erlauben es, daß eine Variable alle Daten enthält, die zu dem von der Variablen repräsentierten Sachverhalt gehören.

Die wichtigste Idee des objektorientierten Konzepts ist die, daß ein Programm letztendlich in Form von abstrakten Datentypen geschrieben werden soll, die einen Arbeitsrahmen für die Organisation dieses Programmes ermöglichen.

Wenn man objektorientiert denkt und programmiert, versucht man, die Objekte, die im Programmfeld eine Rolle spielen, zu ermitteln und zu implementieren. Dabei werden die Objekte, die ähnliche Eigenschaften oder Verhalten besitzen, klassifiziert und in Klassen gegliedert. Hier wird immer die Rede von Klassen und Objekte als Sprachmittel sein, um die objektorientierte Denkweise in allen Entwicklungsphasen des Softwareprodukts zu repräsentieren.

Nachfolgend sind ein paar wichtige Grundbegriffe und Elemente angegeben, die oft im Zusammenhang mit Klassen und Objekten vorkommen:

- **Ein Objekt**

Das Objekt steht im Mittelpunkt der Objektorientierung. Es gibt viele Definitionen für das Objekt. Ein gute abstrakte Definition für ein Objekt ist, daß ein Objekt ein **Etwas** oder ein **Ding** ist, das eine Rolle spielt, das verwendet wird, und Eigenschaften und Aufgaben besitzt.

Abbildung 4.1: Notation für Instanz eines Objektes

Der Objektbegriff ist in der Umgangssprache sehr allgemein belegt und daher für die präzise Definition innerhalb eines computerbasierten Modells ungeeignet. Es wird immer dann von Objekten die Rede sein, wenn der Unterschied zwischen realem Individuum und seiner künstlichen Entsprechung keine Rolle spielt.

Ein Softwareobjekt beinhaltet alle Daten und Strukturen, die es spezifizieren. Im einzelnen sind dies der Objektname (oder der Pointer) als Zugriffsschlüssel, die Attributliste des Objekts, die Attributwerte und Informationen über Verknüpfungen zu anderen Objekten. Die Modellierung von Objekten beinhaltet die Modellierung von Struktur und Verhalten.
Ein Objekt muß aber nicht unbedingt etwas konkretes, greifbares sein. Es kann beliebig abstrakt sein wie z.B. ein Vorgang oder eine Beziehung.
Es muß bei der Modellierung von Objekten zwischen struktureller Objektorientiertheit und verhaltensmäßiger Objektorientiertheit unterschieden werden. Ein voll-objektorientiertes System beinhaltet die strukturelle und die verhaltensmäßige Objektorientiertheit.

Es gibt verschiedene Notation zur grafischen Darstellung von Objekten, die je nach Autor unterschiedlich sind (Abbildung 4.1 und die Notation der OMT später in diesem Kapitel).

- **Eine Klasse**

Klassen sind die Stellvertreter für eine Menge von Individuen mit gleicher Struktur und gleichem Verhalten. In gleichartigen Objekten sind oftmals gemeinsame Teile zu erkennen. Die Extraktion dieser gemeinsamen Teile führt zur Klassenbildung (Abbildung 4.2).

Eine Klasse zeichnet sich dadurch aus, daß sie:

- die Methoden gleichartiger Objekte speichert
- die Namen der objektspezifischen Eigenschaften definiert
- klassenspezifische Eigenschaften speichert

Klasse ist ein technischer Begriff und eng verbunden mit dem Vorgang der Klassifikation, d. h. dem Zuordnen von Individuen gleichen Typs zu einer Obermenge. Klassen können selbst Superklassen untergeordnet sein bzw. Subklassen umfassen.

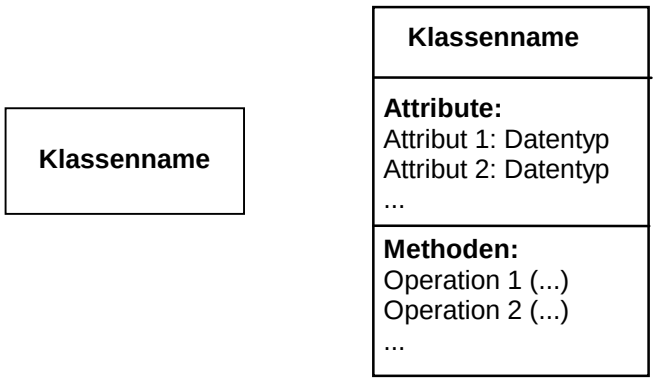


Abbildung 4.2: Notation für eine Klasse

Die Klasse ist der Mechanismus in der Programmiersprache C++ für die Implementierung benutzerdefinierter abstrakter Datentypen. Diese wird mit dem Schlüsselwort **class** deklariert.

- **Abstrakte Klassen**

Sie sind Klassen, von denen keine Instanzen gebildet werden können. In ihrer Klassenstruktur beinhalten sie Methoden, die so deklariert sind, daß sie keinen, bzw. einen leeren Rumpf haben und deshalb auch nicht definiert und nicht ausführbar sind.

Beim Design einer Klassenbibliothek werden abstrakte Klassen gebildet, um auf oberster Ebene einerseits Eigenschaften festzulegen und andererseits Methoden zu deklarieren, die anschließend von abgeleiteten Klassen realisiert werden. Damit wird ein Overhead vermieden, weil z.B. Eigenschaften nur einmal in der Oberklasse definiert werden und nicht mehrfach in den Unterklassen. Im allgemeinen werden einige Klassen einer Klassenhierarchie oft als abstrakte Klassen realisiert.

In C++ wird eine Klasse dadurch zur abstrakten Klasse gemacht, daß man mindestens eine Member-Funktion der Klasse als pure virtuelle Funktion **(pure virtual)** deklariert. In den abgeleiteten Klassen muß man diese Funktion dann definieren, damit man Instanzen (Objekte) von diesen Klassen bilden kann. Der Vorteil liegt darin, daß man die abgeleiteten Klassen zum Implementieren bestimmter

Methoden zwingt, die bei jeder Klasse unbedingt aber anders definiert werden müssen.

- **Basisklasse**

Objektorientierte Systeme werden Inside-out erstellt. Dabei werden bestehende Klassen mit ihrer internen Struktur eingebunden. Diese Klassen werden als Basisklassen bezeichnet, da sie eine Basisfunktionalität bereitstellen. Oft können Basisklassen am Markt in Form von Klassenbibliotheken gekauft werden.

- **Abgeleitete Klasse**

Eine Klasse, die Attribute, Methoden und Datentypen von einer oder mehreren Klassen ererbt, heißt eine abgeleitete Klasse. Dadurch bildet sich eine hierarchische Beziehung zwischen den Klassen.

- **Subklasse**

Eine Klasse, die direkt von einer anderen Klasse A abgeleitet ist, nennt man Subklasse der Klasse A.

- **Superklasse**

Eine Klasse, von der eine Klasse B direkt abgeleitet ist, nennt man Superklasse der Klasse B.

- **Elternklasse**

Unter Eltern werden diejenigen Superklassen verstanden, die einer Klasse in der Vererbungshierarchie direkt vorgeordnet sind.

- **Wurzelklasse**

Eine Wurzelklasse ist eine Klasse, die keine Superklassen hat.

- **Bibliotheksklasse**

Eine Bibliotheksklasse ist eine Klasse, die eigentlich in einem Projekt definiert ist und in anderer Objektbank nur wiederverwendet wird. Eine solche Klasse wird als Bibliotheksklasse markiert. Diese Klasse kann dann z.B. als Superklasse oder als Memberklasse (Memberdaten in der Klassenstruktur) benutzt werden.

- **Template**

Template-Klassen sind parametrisierte Schablonen für Klassen. Instanzen von Templates sind Klassen.

Man kann z.B. eine Klasse '*Liste*' definieren, unabhängig vom Typ der verwalteten Objekte. Diese Template-Klassen werden erst zur Compilierzeit in 'normale' Klassen übersetzt.

- **Container-Klasse**

Eine Container-Klasse ist eine Klasse, die andere Objekte enthält und verwalten kann. Sie dient als Objekt zur Verwaltung von Mengen anderer Objekte. Beispiele hierfür sind Listen, Bäume, Stapel etc.

Die **Mengenklassen** sind ein typisches Beispiel für Containerklassen, die andere Objekte zu einer Menge zusammenfassen und verwalten. Sie sind von ausgesuchtem Interesse, da es bei der Datenverarbeitung im wesentlichen um die Verarbeitung von großen Mengen von Daten geht, wie Bauteilverwaltung, Personenverwaltung, Auftragsverwaltung usw.

Um als Container-Klasse benutzt werden zu können, muß die Klasse typenlos oder als Template-Klasse mit genau einem Parameter, der den Typ der verwalteten Objekte angibt, definiert sein (siehe die Klasse *SH_Liste* für dynamische, doppelt verkettete Liste in Kapitel *6*).

- **Daten und Datentypen**

Die Daten eines Objekts sind die Attribute und Eigenschaften, die es beschreiben. Sie sind innerhalb des Objekts verborgen und es kann nur durch Schnittstellen von Außen auf sie zugegriffen werden, die dieses Objekt zur Verfügung stellt. Die Objektstruktur besteht aus einem oder mehreren Datenstrukturen, die der Klasse dieses Objektes zugeordnet sind. Diese Daten werden auch Member-Daten der Klasse genannt.

Eine Datenstruktur wird dann als abstrakter Datentyp bezeichnet, wenn Sie eine innere Merkmalsstruktur aufweist und wenn auf diese Merkmale Operatoren definiert werden können. Auf diese innere Struktur kann von außen nur über die Operatoren dieses Datentyps zugegriffen werden. Abstrakte Datentypen helfen, Software nach dem Geheimnisprinzip zu realisieren.

Programmiersprachen verfügen üblicherweise über fundamentale Datentypen (FDTs) wie z.B. **Int**, **char**, **float**, **double**, usw. bei der Programmiersprache C++. Die tatsächliche Größe in Byte dieser Datentypen ist nicht in der Sprache spezifiziert, die Größen sind grundsätzlich maschinenabhängig.

Von diesen Datentypen werden alle anderen Datentypen abgeleitet, da es oft notwendig ist, aus den fundamentalen Datentypen, mit Hilfe von verschiedenen Sprachmitteln, höhere Datentypen zusammenzusetzen.

Ein **typedef** ist ein Datentyp. In der Regel wird er dazu verwendet 'komplizierten' Datentypen wie z.B. einem Pointer auf eine Funktion, einen einfachen und sprechenden Namen zu vergeben, oder einen Punkt, eine Datenstruktur aus x- und y-Koordinaten zu definieren.

- **Methoden**

Eine Methode ist eine Funktion, die einer Klasse zugeordnet ist. Klassen stellen Funktionen als Schnittstellen zum Zugreifen auf ihre Daten zur Verfügung. Diese Funktionen werden auch Member-Funktionen der Klasse genannt. Methoden werden für das Message Passing verwendet.

Die **Virtuellen Methoden (virtual Function)** bezeichnen diejenigen Methoden einer Klasse, welche in Superklassen definiert sind und die auf die Subklassen vererbt werden. Sie implementieren die Polymorphie in C++.

- **Verkapselung**

Daten und Methoden sind in einer Klasse verkapselt. So kann man den Zugriff auf die Attribute, Methoden und Datentypen einer Klasse kontrollieren. Hierfür sind Zugriffsrechte sowohl für die Attribute als auch für die Methoden der Klasse vorgesehen.

Kapselung ist die Kombination einer Datenstruktur mit den Funktionen (Aktionen oder Methoden), die diese Daten manipulieren. Daten und Methoden werden also zu einem einzigen Datentyp zusammengefügt, in dem der Zugriff auf diese Daten und Methoden kontrolliert werden kann.

Bei der Definition einer Klasse muß schon bekannt sein, inwiefern andere und abgeleitete Klassen, die Elemente (Methoden und Attribute) dieser Klasse noch brauchen, und somit die Zugriffsrechte bestimmen. Durch den restriktiven Zugriff auf die privaten Daten einer Klasse kann man Fehlermeldungen, die auf eine falsche Behandlung der Daten schließen lassen, ausschließlich auf die Methoden zurückführen. Andere Funktionen kommen als Fehlerquelle nicht in Betracht.

In C++ sind solche **Zugriffsrechte** auf zwei Ebenen geregelt, deren Kombination den vererbten Zugriff definieren:

- Einmal bei der Regelung des Zugriffs auf die Member-Daten und Member-Funktionen. Dies erfolgt bei der Deklaration jedes Member-Elements schon bei der Deklaration der Klasse,

- und dann bei der Regelung des Zugriffs von einer Unterklasse auf eine Oberklasse.

Die Zugriffsrechte sind wie folgt definiert:

⇒ **private**:
Der Zugriff auf Elemente, die nach diesem Schlüsselwort stehen, können nur die Elementfunktionen zugreifen, die innerhalb derselben Klasse definiert sind.

⇒ **public**:
Der Zugriff auf Elemente, die nach diesem Schlüsselwort stehen, können alle Funktionen in demselben Gültigkeitsbereich, den die Klassendefinition hat, zugreifen.

⇒ **protected**:
Der Zugriff auf Elemente, die nach diesem Schlüsselwort stehen, können nur Elementfunktionen zugreifen, die innerhalb derselben Klasse definiert sind oder die in einer von der Klasse direkt abgeleiteten Klasse definiert sind.

Zugriff in Basisklasse	Zugriffsmodifizierer	Vererbter Zugriff
public	public	public
private	public	nicht zugreifbar
protected	public	protected
public	private	private
private	private	nicht zugreifbar
protected	private	private

In C++ kann das Zugriffsrecht aber nur bestimmten ausgewählten Klassen oder Funktionen (Friend Function, Friend class) vergeben werden. Die Restriktionen beim Zugriff auf private Elemente einer Klasse können dadurch umgangen werden, daß man beliebigen Funktionen oder Klassen durch die Deklaration friend den gleichen Status wie Klassenmethoden einräumt.

- **Überladen von Operatoren und Funktionen (Overloading)**

Das Überladen (Overloading) ist ein Mechanismus, der das Überladen von Operatoren und Funktionen erlaubt. Somit ist die Verwendung der Standardoperationen wie Multiplikation, Addition, Zuweisung usw. auch für benutzerdefinierte Datentypen und Klassen möglich.

In C++ müssen dynamisch gebundene Funktionen explizit durch das Schlüsselwort **virtual** gekennzeichnet sein.

- **Nachrichten Empfangen und Verschicken (Message Passing)**

Objekte haben Beziehungen zueinander, sie sind fähig ihre Eigenschaften durch Schnittstellen (Member-Funktionen) zu ändern. In objektorientierten Systemen kommunizieren Objekte über die Versendung von Nachrichten (Message). Dieses Empfangen und Verschicken von Nachrichten (Message Passing) wird in C++ als Methodenaufruf von Member-Funktionen realisiert (Abbildung 4.3).

An jedes Objekt einer Unterklasse in einer Klassenhierarchie können Nachrichten (Messages) mit demselben Namen gesandt werden. Dieses Verfahren ist folglich sehr gut verwendbar für generelle Operationen, die auf alle Objekte angewandt werden, die aber von den einzelnen Objekten unterschiedlich realisiert werden müssen.

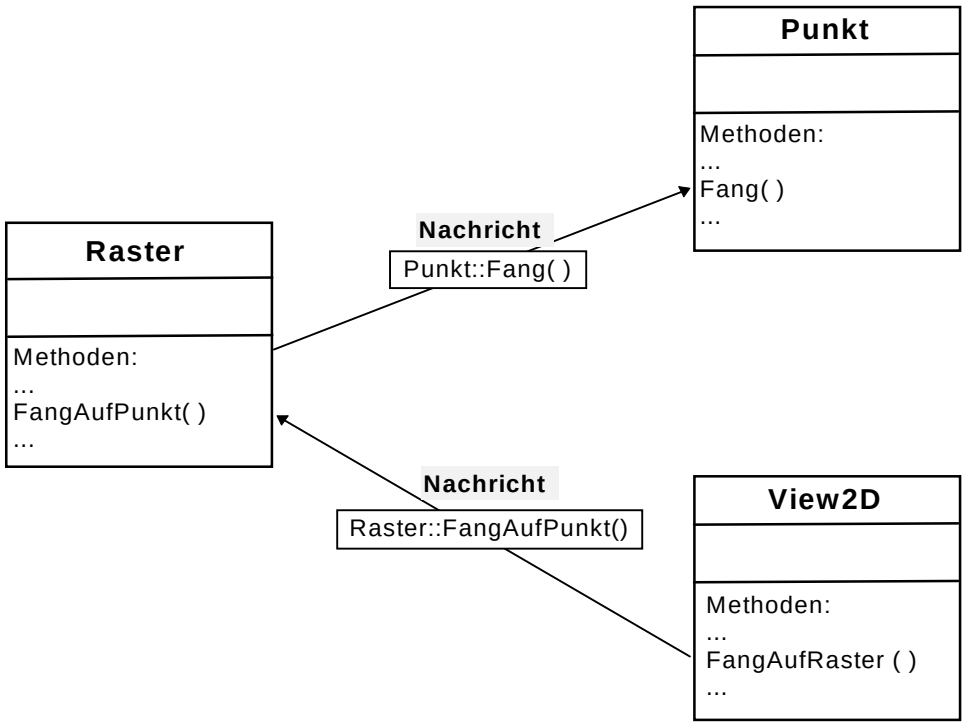


Abbildung 4.3: Austauschen von Nachrichten (Message Passing)

Die Vorteile liegen einmal in dem erreichten Abstraktionsniveau und zum anderen in der einfachen Wartung und Erweiterbarkeit von Systemen. Das Hinzufügen neuer Objekte ist problemlos möglich, da nur diese neuen Unterklassen mit den Methoden definiert werden müssen.

- **Ereignisse (Event)**

Ein Ereignis (Event) ist das Geschehen von Etwas an einem bestimmten Zeitpunkt, z.B. wenn ein Benutzer die linke Maustaste drückt. Es hat keine Dauer und geschieht in einer Richtung (one-way).

Ein Objekt kann durch Ereignisse auf ein anderes Einfluß nehmen. Dabei „sendet“ ein Objekt einem anderen Ereignisse. Dies geschieht durch eine Aktion.
Die Ereignisse dienen zur Übermittlung von Informationen zwischen den Objekten, sie sind normalerweise einzigartig, aber können in Klassen mit Attributen gruppiert werden.
Ein Interaktionsdiagramm wird erstellt , um die Ereignisflüsse zwischen Gruppen von Klassen zu verfolgen (event trace) und zu zeigen (Abbildung 4.4).

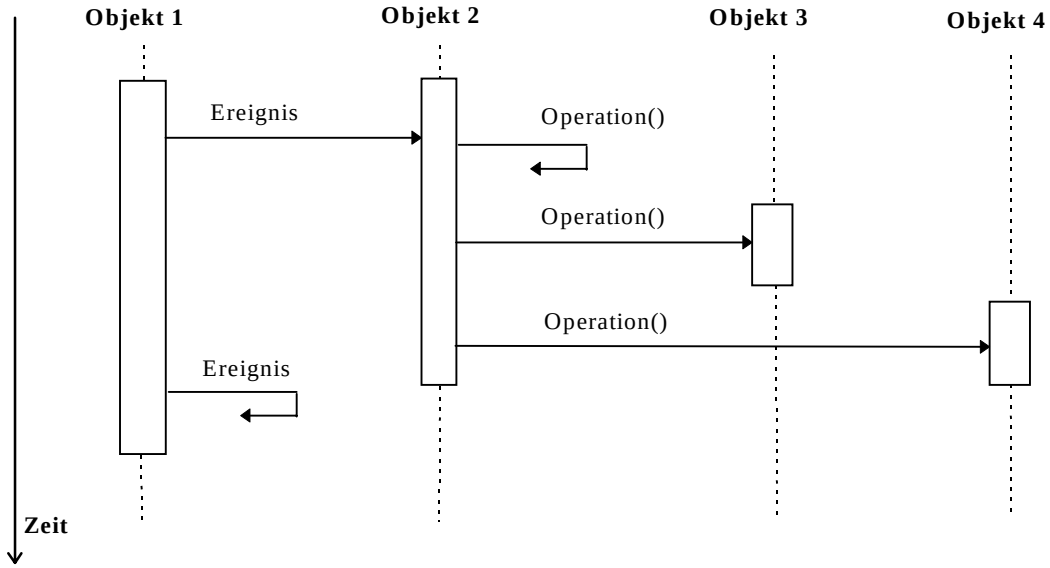


Abbildung 4.4: Interaktionsdiagramm

4.2.2 Beziehung zwischen Klassen und Objekten

Ein Verständnis der Beziehung zwischen Klassen und Objekten mit ihren Mechanismen ist ein wichtiger Schritt um Datenmodelle und Systeme in dem objektorientierten Stiel zu entwickeln und zu implementieren.
Der Einsatz objektorientierter Programmiersprachen wie z.B. C++ verlangt ein Verständnis des objektorientierten Konzepts. Dabei sind Klassen und Objekte der Mittelpunkt dieses Konzepts.

Im folgenden wird auf die wichtigsten Grundbegriffe eingegangen, die die Beziehungen zwischen Klassen und Objekten im Rahmen des objektorientierten Konzepts beschreiben.

- **Vererbung (Inheritance), Generalisierung/Spezialisierung**

Das Vererben von Eigenschaften bedeutet in objektorientierten Systemen, daß die Attribute, Methoden und Datentypen einer übergeordneten Klasse in den nachgeordneten Klassen identisch verfügbar sind. Somit müssen Eigenschaften, welche mehrere Klassen auszeichnen, nur einmal in einer Klassenhierarchie definiert werden (s. einfache Vererbung in Abbildung 4.5).

Die Vererbung stellt eine „is a“-Beziehung zwischen zwei Klassen dar, in der eine Klasse eine **Generalisierung** oder **Spezialisierung** einer anderen Klasse ist. Die generalisierte Klasse in dieser Beziehung wird Basisklasse, Elternklasse, und Superklasse genannt. Die spezialisierte Klasse wird abgeleitete Klasse, Subklasse Kindklasse genannt.

Durch Generalisierung oder Spezialisierung besteht die Möglichkeit, neue Klassen mit bestimmten Eigenschaften (Daten) und Methoden (Operationen) zu bilden.
Spezialisierung ist die Erzeugung neuer Klassen durch Vererbung von einer oder mehreren Superklassen [7]. Dabei sind zwei allgemeine Methoden der Spezialisierung anzuwenden [54]:

⇒ **Spezialisierung durch Addieren „by Addition“** von neuen charakteristischen Attributen, Operationen, oder Assoziationen zu einer existierenden Klasse, um eine neue Subklasse von dieser Klasse zu erzeugen.

⇒ **Spezialisierung durch Einschränkung „by Constraint“** von bestimmten charakteristischen Eigenschaften einer existierenden Klasse, der Superklasse.

Der Vererbungsmechanismus gestattet es dem Programmierer, gemeinsame Teile von Klassen herauszuziehen und in einer Oberklasse zusammenzufassen, man spricht dann von **Klassenbildung durch Generalisierung**. In einer Klassenhierarchie mit mehreren Ebenen führt die Extraktion von gemeinsamen Teilen auch zu einer Verbesserung der Lesbarkeit. Daher ist es auch sinnvoll, eine in (fast) allen Klassen vorhandene Methode in der gemeinsamen Oberklasse zu deklarieren und in abgeleiteten Klassen entsprechend zu definieren.

Bei der Vererbung muß zwischen der Vererbung von Struktur und der Vererbung von Werten unterschieden werden. Im Gegensatz zu Interpreter-Systemen wird in Compiler-Sprachen wie C++ ausschließlich die Vererbung von Struktur unterstützt, d. h. es werden die Attribute, Methoden und Datentypen einer Klasse ohne individuelle Werte vererbt

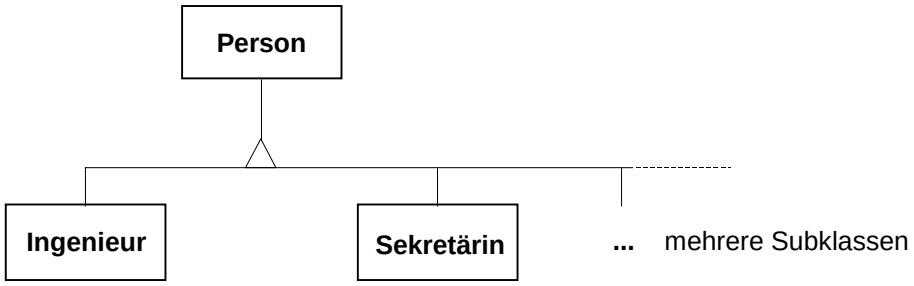


Abbildung 4.5: Einfache Vererbung

Die mehrfache oder multiple Vererbung (Abbildung 4.6) ist eine Eigenschaft einiger objektorientierter Sprachen, die es erlaubt, daß eine Klasse mehrere übergeordnete Klassen hat. In allen Fällen, in denen eine Subklasse mehrere Superklassen hat, werden alle Attribute, Methoden und Datentypen der übergeordneten Klassen auf die Subklasse vererbt.

Die Behandlung der Fälle, in denen gleichnamige Attribute, Methoden und Datentypen von mehreren Superklassen vererbt werden, ist in den meisten Sprachen unterschiedlich. In C++ ist die Subklasse die Vereinigungsmenge aller Attribute, Methoden und Datentypen der Superklassen.

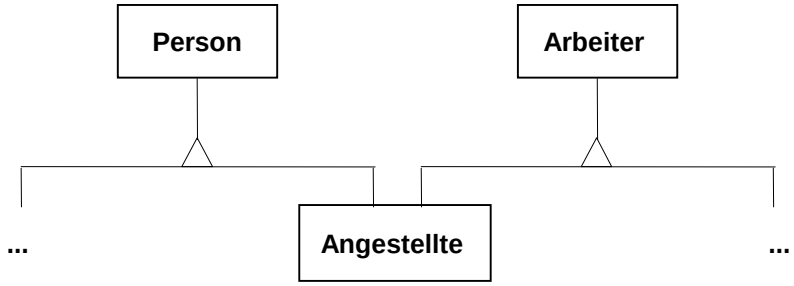


Abbildung 4.6: Mehrfache Vererbung

- **Verknüpfungen und Assoziationen**

Assoziationen stellen die Beziehungen der Objekte untereinander dar. Sie zeigen die Verbindungen und Abhängigkeiten zwischen den Objekten auf und sind bidirektional, müssen aber nicht bidirektional implementiert werden.

Eine Verknüpfung ist eine Instanz einer Assoziation, so daß eine Assoziation in einem Klassendiagramm mit mehreren Verknüpfungen in den Instanzdiagrammen korrespondieren kann. Der Grad der Assoziation hängt von dem behandelten Problem und dem entscheidenden Design ab, es ist aber zu empfehlen, binäre Assoziationen zu modellieren, da Assoziationen mit einem Grad größer 3 nur schwer zu zeichnen und noch schwerer zu verstehen sind (Abbildung 4.7).

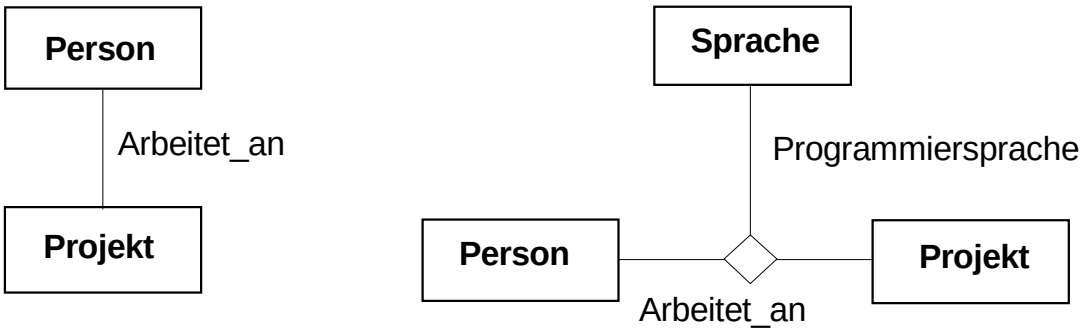


Abbildung 4.7: binäre (links) und ternäre (rechts) Assoziation

Eine Assoziation ist eine mit einem Namen versehene Verbindung zwischen zwei oder mehreren Objekten, der die Art der Beziehung ausdrückt. Als Konvention gilt, daß Assoziationen von links nach rechts gelesen werden. Da aber aus Platzgründen eine entsprechende Anordnung der Objekte nicht immer möglich ist, können Rollennamen für die Teilnehmer vergeben werden.

Eine Beziehung (Assoziation) kann als eins zu eins eindeutig (1:1), einseitig eindeutig (1:n) oder komplex (n:m) typisiert werden (Abbildung 4.8).

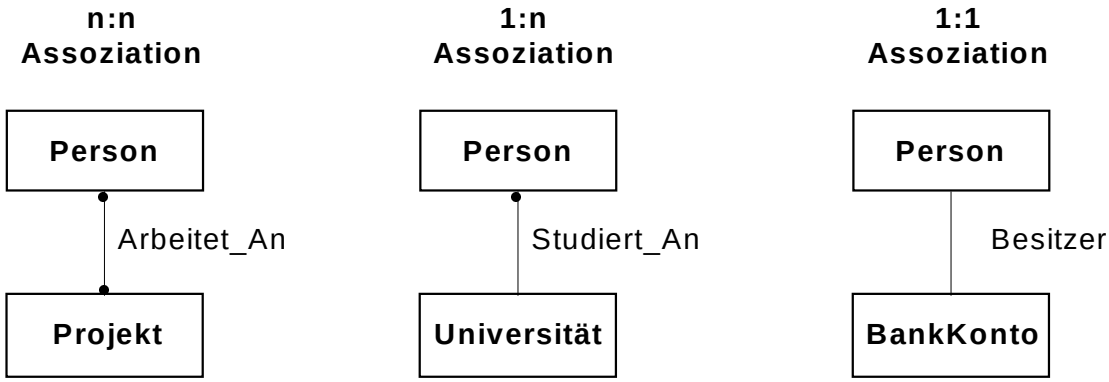


Abbildung 4.8: Kardinalität einer binären Assoziation

Einige Beziehungen können Eigenschaften besitzen, die wiederum als Attribute der Beziehung zu verstehen sind.

Einschränkungen an die vielfältigen Beziehungen (**Constraints on Links**) sind manchmal notwendig bei der Modellierung von Objekten. Um die Eigenschaften (Attribute) und Methoden (Operationen) der Beziehung zu definieren, werden Verbindungsobjekte (**Link Objects**) eingesetzt. Die Assoziation wird dabei als Klasse modelliert (Abbildung 4.9).

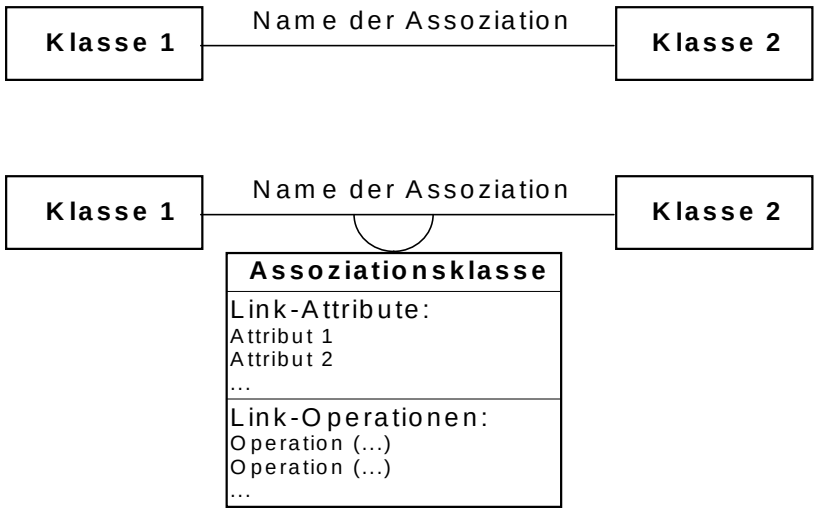


Abbildung 4.9: Assoziation als Klasse mit Verbindungsattributen (Link-Attribute)

Dadurch kann z.B. die Anzahl der Objekte bestimmt werden, die mit einem gegebenen Objekt verbunden werden sollen, oder ein lokales Koordinatensystem für

jedes Maschinenteil als Versatz von dem globalen Koordinatensystem der Maschine durch ein Verbindungsobjekt definieren, das mit jedem Teil verbunden ist.

- **Aggregation**

Die Aggregation ist im Prinzip eine spezielle Form der Assoziation, und zwar eine starke Form von Assoziation. Sie ist transitiv und asymmetrisch.

Abbildung 4.10: Aggregation

Eine Aggregation stellt eine „part-of"-Beziehung oder „has"-Beziehung zwischen zwei Objekten dar, in der ein Objekt aus mehreren Komponenten konstruiert wird. Dabei „hat" ein Objekt andere Objekte in seiner Datenstruktur, oder anders gesagt, diese Objekte sind teil von „part-of" diesem Objekt (Abbildung 4.10). Falls dies jedoch nicht eindeutig der Fall ist, sollte besser eine normale Assoziation verwandt werden.
Eine Aggregation kann fest, variabel oder rekursiv sein.

⇒ **fest:**
feste Struktur, feste Anzahl von Bestandteilen

⇒ **variabel:**
endliche Zahl an Ebenen, aber die Anzahl der Bestandteile kann variieren

⇒ **rekursiv:**
enthält, direkt oder indirekt, eine Instanz desselben Aggregats => potentiell unendliche Anzahl der Ebenen

4.2.3 Polymorphismus (dynamisches Binden), virtuelle Methoden

Polymorphismus ist die Möglichkeit, in einem System verschiedenen Objekten der gleichen Hierarchie unterschiedliche Methoden mit dem gleichen Namen zu geben, die sich in der Implementation unterscheiden.

An jede Unterklasse in einer Klassenhierarchie können Messages (Nachrichten) mit demselben Namen gesandt werden. Dieses Verfahren ist folglich sehr gut verwendbar für generelle Operationen, die auf alle Objekte angewandt werden, die aber von den einzelnen Objekten unterschiedlich realisiert werden müssen.

Die Vorteile liegen einmal in dem erreichten Abstraktionsniveau und zum anderen in der einfachen Wartung und Erweiterbarkeit von Systemen. Das Hinzufügen von

neuen Objekten ist problemlos möglich, da nur diese neuen Unterklassen mit den Methoden definiert werden müssen. In C++ zum Beispiel müssen dynamisch gebundene Funktionen explizit durch das Schlüsselwort **virtual** gekennzeichnet sein. In C++ können z.B. die Klassen ‘Linie’ und ‘Kreis’ beide abgeleitet von ‘Objekt’ sein. In allen drei Klassen wird die virtuelle Methode ‘Zeichnen’ definiert, die aber je nach Klasse unterschiedlich implementiert ist.

In einer Container-Klasse, die nicht weiß, ob es sich bei einem Objekt um einen Kreis oder eine Linie handelt, wird bei Aufruf der Methode ‘Zeichnen’ immer die korrekte Methode in der entsprechenden Klasse aufgerufen.

4.3 Objektorientierte Modellierung von Software

Die Produktion von Software ist der Hauptzweck der Softwarehäuser, daher ist sie vergleichbar mit der Produktion von Maschinen, die auch Planung, Erstellung und Wartung brauchen. Dies ist die Aufgabe des Ingenieurwesens (Software Engineering), wobei nichts näher liegt, als seine Methoden auf die Softwareproduktion zu übertragen.

Schon am Anfang der 70er Jahre hat Boehm [6] die Probleme bei der Erstellung von Software untersucht. Sein Fazit war: Weil die Software nicht nach den Regeln der Ingenieurskunst und mit richtigen Werkzeugen erstellt wird, scheitern viele Projekte, werden zu teuer oder verspäten sich enorm.

Die objektorientierten Technologien bei der Modellierung und Entwicklung von Softwareprodukten ist eine neue Denkweise, Probleme der realen Welt konzeptuell nach organisierten Modellen zu behandeln.

Objektorientierung bei der Softwareentwicklung heißt, daß die Software wie eine Sammlung von diskreten Objekten organisiert wird, die beide, Datenstruktur und Eigenschaften, bindet [6]. Dabei ist das Objekt der fundamentale Bausteine, der Datenstruktur (Attribute) und Eigenschaften (Operationen) in einem Element vereinigt. Objekte der realen Welt werden mit ihrer Datenstruktur und Eigenschaften auf Softwareobjekte übertragen. Dies steht im Gegensatz zur konventionellen Programmierung, wobei Datenstruktur und Eigenschaften nur frei und locker verbunden sind.

Eine verhaltensmäßige Objektorientiertheit kann dann angenommen werden, wenn in einem Objekt Eigenschaften und Funktionen miteinander verschachtelt werden. Das heißt, Daten und ihre zugehörigen Funktionen werden miteinander verwaltet.

Die objektorientierte Modellierung OOM umfaßt alle Phasen der Softwareentwicklung. Im einzelnen sind dies:

- die objektorientierte Analyse OOA
- das objektorientierte Design
- die objektorientierte Implementierung

4.3.1 Objektorientierte Analyse (OOA)

Die Analyse ist die erste Phase der Systementwicklung mit Hilfe der objektorientierten Technologien. Die **O**bjekt**o**erientierte **A**nalyse (**OOA**) wird definiert als Analysemethode, welche die Anforderungen an ein neues System aus dem Sprachgebrauch des jeweiligen Problembereichs und aus der Perspektive von Objekten ableitet. Andere Methoden sind z. B. die strukturierte Analyse (Structured Analysis) und die SADT. Diese beiden Methoden beruhen im wesentlichen auf der strukturierten Top-Down-Zerlegung aus der Sicht von Prozessen.

Die OOA ist Aufgabe der Analytiker. Dabei soll eine konzeptionelle Übersicht über das System erarbeitet werden. Abstimmung und Diskussionen mit dem Systembenutzer oder dem Auftraggeber ist in dieser Phase sehr wichtig, um eine Beschreibung des Problems zu finden.

Die Abstimmung und Koordination der einzelnen inhaltlichen Überlegungen in einem Softwareprojekt führen dazu, daß man die OOA als einen Prozeß betrachten muß, der nicht zu einem Optimum, sondern zu einer pragmatischen Lösung führt.

Die Produktion von Software bleibt ein intellektuelles Problem, da keine eindeutige Ableitungsmethode existiert, die ein gültiges Objektmodell erzeugt.

Es gibt viele Vorschläge und Ansätze, wie man am besten bei der OO-Analyse voran geht (wie bei Coad/Yourdon, Booch und Rumbaugh im „Methoden und Ansätze der Objektorientierten Modellierung“ später in diesem Kapitel).

4.3.2 Objektorientierte Design (OOD)

In der Phase des objektorientierten Designs (OOD) erfolgt die technische Spezifikation des Anwendungssystems. Die Ergebnisse der OOA werden genutzt, um zu einem technischen Systementwurf zu gelangen, welcher anschließend implementiert werden kann.

Das OOD ist ein Bindeglied zwischen der inhaltlichen Beschreibung eines Systems im Rahmen des Fachkonzepts und der Umsetzung in einer konkreten Implementierung. Es baut auf die Dokumentation der OOA auf und verwendet ihre Ergebnisse. Dabei werden diese Dokumente erweitert und verfeinert, um sie dann als Dokumente des OOD an die Programmierung weiterzugeben.

In anderen Worten, die OOA erfolgt in der Sprache der Fachabteilungen, und nur diejenige Sachverhalte werden erfaßt und modelliert, welche aus der Sicht der Fachexperten relevant und verständlich sind. Bei dem OOD werden solche Fragen beantwortet: Wie Daten gespeichert werden sollen? Wie sieht die grafische Benutzeroberfläche (Layout) aus? Welche zusätzlichen Klassen, Variablen, Methoden, etc. sind zur Umsetzung des Modells notwendig sind? Ob Optimierung notwendig ist, und welche? Etc.

Die Begriffsbildung ist analog zum strukturierten Design SD (Structured Design) der traditionellen Software-Entwicklung zu sehen. Jedoch erfolgt der Entwurf aus der Sicht von Objekten und nicht aus der Sicht von Prozeduren.

Das OOD wird von allen Autoren anders beschrieben. Es erfolgt bei den meisten keine Abgrenzung von inhaltlicher und technischer Systembeschreibung; die gleichen Begriffe wie Klassen Objekte, Vererbung, Message Passing etc., werden sowohl in der OOA als auch im OOD verwendet. Damit kann ein Softwareentwickler mit fundiertem Wissen über die traditionelle Software Engineering keinerlei Transparent erhalten, und die Objektorientierung im Zweifel für untauglich erklären.

Wie bei der Analyse gibt es auch verschiede Ansätze und Methoden für ein objektorientiertes Design (wie bei Coad/Yourdon, Booch und Rumbaugh im „Methoden und Ansätze der Objektorientierten Modellierung“ später in diesem Kapitel).

4.3.3 Objektorientierte Programmierung (OOP)

Die objektorientierte Programmierung (OOP) steht als Synonym für die Umsetzung der Entwürfe in Quelltexte. Oftmals wird die Unterstützung der Programmierung durch die diversen Werkzeuge mit dem Schlagwort Lower-CASE zusammengefaßt.

Die OOP ist mit der strukturierten Programmierung aus der Sicht von Objekten gleichzusetzen. Genauso wie dort Regeln definiert wurden, nach denen Programmierer zur Vermeidung von Spaghetti-Code vorgehen sollten, werden nun diejenigen Regeln vorgeschlagen, nach denen Programmierer im Sinne der Objektorientierung vorgehen sollen.

Die objektorientierte Programmierung ist in mehreren Sprachen zu realisieren. Diese wären unter anderen C++, Turbo Pascal und Smalltalk. Die in dieser Arbeit zugrundeliegende Sprache C++ ist eine hybride Sprache. Sie vereinigt die Konzepte der Vererbung und der abstrakten Datentypen mit der funktionalen Programmierung. Sie geht auf die Arbeiten von Stroustrup [74] u. a. zurück. Erstmals beschrieben wurde sie 1984. Die Sprachnorm für C++ ist mittlerweile in Version 2.0 verfügbar. C++ ist eine objektorientierte Programmiersprache.

Die OOP-Erweiterungen in C++ nützen das menschliche Denkverhalten des Klassifizierens und des Abstrahierens aus. C++ wurde aus diesem Grunde ursprünglich "C mit Klassen" genannt.

Die Vorteile liegen einmal in dem erreichten Abstraktionsniveau und zum anderen in der einfachen Wartung und Erweiterbarkeit von Systemen. Es erfolgt eine direkte Umsetzung des objektorientierten Designs in Quelltext in C++.

Das Hinzufügen neuer Objekte ist problemlos möglich, da nur diese neuen Unterklassen mit den Methoden definiert werden müssen.

4.4 Methoden und Ansätze der Objektorientierten Modellierung

Demnächst wird eine Übersicht über die wichtigsten objektorientierten Modellierungsmethoden für alle Phasen der Softwareentwicklung von der **OOA** über **OOD** bis hin zur Implementierung **OOP** gegeben, die seit Anfang der 90er Jahre durch die Veröffentlichungen von Coad/Yourdon [11] [12], Booch [7], Rumbaugh [63] sehr verbreitet sind.

4.4.1 Coad/Yourdon

Peter Coad und Edward Yourdon stellten in ihren Büchern [11] und [12] eine Methode für die objektorientierte Softwareentwicklung vor, die reiche Informationen über die OOA und OOD beinhaltet.

Die Methode von Coad/Yourdon ist weiterhin von großer Bedeutung und als Informationsquelle zu OOA und OOD zu beachten, wird aber in der Praxis selten angewendet.

Die Beschreibung der OOA und der OOD sind zwar in zwei Büchern inhaltlich getrennt, aber in der praktischen Anwendung der Methode sind sie zeitlich nicht als voneinander getrennte Phasen der objektorientierten Modellierung OOM zu sehen.

- **Analyse**

In der Analyse wird der Problembereich genau mit dem Ziel beobachtet, Aufgaben und Anforderungen an ein System aus der Perspektive von Objekten zu definieren. Es geht dabei um die Modellierung des Problembereichs, die Schritt für Schritt erfolgt.

Diese schrittweise Entwicklung verwendet ein Schichtmodell, das die Grundlagen dieser Methode bildet. Somit gelangt man von einer Abstraktion (Themen) zu Details wie Attributen und Diensten.

Dieses Schichtenmodell ist als Menge von Folien anzusehen, die nacheinander aufeinandergelegt werden. Es besteht aus folgenden Teilen:

1) Themen:
Ein Thema faßt eine Menge von zugehörigen Klassen zusammen. Dadurch wird die Komplexität des Problembereichs bei der Modellierung von großen Problemen zerlegt und für den Menschen anschaulicher

2) Klassen/Objekte:
Sie sind Dinge aus dem Problembereich. Es erfolgt eine Identifizierung der Klassen und Objekten des Problembereichs und eine Zusammenfassung von ähnlichen Klassen/Objekten. Es heißt dann, Klassen und Objekte zu finden

3) Struktur:
Bei der Identifizierung der Strukturen wird die Komplexität des Problembereichs ausgedrückt. Dabei werden die Beziehungen zwischen den Klassen (Spezialisierung, Generalisierung) und Objekten (Teil-von-Beziehung) zusammengefaßt. Es erfolgt eine Spezialisierung und Generalisierung von Klassen.

In den Büchern von Coad/Yourdon werden dazu folgenden Begriffe verwendet:

- Gen-Spec-Strukturen
- Whole-Part-Strukturen

4) Attribute:
Sie sind die Eigenschaften der Objekte, die zu identifizieren und spezifizieren sind. Durch den Wert seiner Attribute wird der aktuelle Zustand eines Objekts beschrieben. Daher besitzt jedes Objekt einen eigenen Wert für jedes Attribut.

Neben diesen Attributen werden Assoziationen zwischen Objekten als Instanz-Verbindungen modelliert. Dies erfordert die folgenden Schritte:

- Attribute identifizieren: Dabei sind folgende Fragen aus der Sicht eines Objekts zu stellen:
  - Wie werde ich beschrieben?
  - Was muß ich wissen?
  - In welchen möglichen Zuständen kann ich mich befinden, und welche Information benötige ich in jedem Zustand?
- Attribute positionieren
- Instanzen-Verbindungen identifizieren
- Die Spezialfälle betrachten
- Spezifikation und Erklärung der Attribute

5) Dienste:
Damit ist die Analyse des dynamischen Verhaltens der Objekte gemeint. Ein Dienst ist ein spezielles Verhalten, das ein Objekt aufweist. Die möglichen Werte der Attribute werden überlegt und betrachtet, um die Zustände eines Objekt zu identifizieren. Dafür werden zu jedem Objekt Zustandsdiagramme und Service-Charts vorgestellt.

Es wird bestimmt, welche unterschiedlichen Verhalten für dieses Objekts erforderlich sind. Bei der Spezifikation der Dienste wird das Verhalten des Objekts beim Übergang von einem Zustand in einen Folgezustand festgelegt

Man unterscheidet zwischen einfachen und komplexen Diensten.

⇒ Einfache Dienste: Sie decken den größten Teil des Objektverhaltens ab, wie z.B. die Konstruktion neuer Objekte, die Herstellung von Verbindungen zu anderen Objekten, der Zugriff auf Attribute oder Entfernung eines Objekts einer Klasse

⇒ Komplexe Dienste: Sie lassen sich in zwei Kategorien klassifizieren

- Berechnung von Ergebnissen
- Überwachungsfunktionen

Die Nachrichten-Verbindungen dienen zur Beschreibung der Abhängigkeit zwischen Objekten bei der Ausführung von Diensten, und zur Modellierung dieser Abhängigkeit.

- **Design**

Die in der Analyse erstellte Spezifikation soll beim Design so erweitert werden, daß eine konkrete Implementierung möglich wird. Dabei sind neue Komponenten zu entwerfen.

Schichten- und Mehrkomponentenmodell

Durch Anwendung des in der Analyse eingeführten Schichtenmodells wird der Übergang von der Analyse zum Design möglich. Es sind im allgemeinen vier Systemkomponente zu entwerfen, wie Problembereich, Benutzerschnittstelle, Task-Management und Datenverwaltung.

1) Entwurf der Problembereichskomponente

Folgende Ergänzungen der bei der Analyse ermittelten Klassen sind nötig, um eine Implementierung zu ermöglichen:

- Einführung weiterer Wurzelklassen
- Anpassung der Vererbungshierarchien
- Effizienzbedingte Anpassungen zur Steigerung der Ausführungsgeschwindigkeit durch zusätzliche Attribute oder Methoden
- Unterstützung der Datenverwaltungskomponente, wie z.B. Dienste, damit sich ein Objekt selber speichern kann

2) Entwurf der Benutzerschnittstelle

Eine ergonomische Schnittstelle gehört zur Zeit zu jedem modernen Software-System, die in dem System integriert ist und dem Benutzer somit das innovative und interaktive Arbeiten ermöglicht. Beim Entwurf dieser Schnittstelle sollen die verschiedenen Benutzergruppen des Systems berücksichtigt und eine

Kommandohierarchie entwickelt werden, die sich an den vorhandenen Standards orientiert.

3) Entwurf der Task-Management-Komponente

Aus folgenden Gründen ist eine Task-Management-Komponente notwendig:

- Mehr-Benutzersysteme (Multi User Systems)
- Systeme mit Subsystemen
- Kommunikation mit anderen Systemen
- Mehrprozessor-Architekturen

Man geht folgend vor:

- Identifizierung der ereignisgesteuerten Tasks
- Identifizierung der zeitgesteuerten Tasks
- Zuordnung von Prioritäten, wobei die kritischen Tasks höhere Prioritäten erhalten
- Festlegung eines Koordinations-Tasks zur Steuerung der anderen, wenn mehr als zwei Tasks vorliegen

4) Entwurf der Datenverwaltungskomponente

Es werden folgende Ansätze zur Speicherung von persistenten Daten verfolgt:

- Flat-File-Management, wobei eine Datei zur Speicherung der Objektattribute angelegt wird
- relationale DBMS (Data Base Management System)
- Objektorientiertes DBMS

Die Stärke der Methode Coad/Yourdon liegt darin, übersichtliche Klassen, Attribute, Dienste, Strukturen und Themen in einem Diagramm, sowie eine detaillierte Spezifikation über Attribute und Dienste zu liefern.

Sowohl die Modellierung von Objektlebenszyklen und Zeitverhalten, als auch die ungenau definierten Schnittstellen zwischen den einzelnen Themen sind als schwache Aspekte bei dieser Methode anzusehen.

4.4.2 Booch

Eine andere, sehr bekannte OO-Methode für die Softwareentwickler hat Grady Booch, Chefentwickler bei der Firma Rational, in seinem Buch [7] vorgeschlagen. Die Methode umfaßt die Phasen der OOA und OOD in einem Buch und definiert eigene Notationen zur Dokumentation eines Systems, die Booch dazu entwickelt hat.

Booch beschreibt ein objektorientiertes System durch verschiedene Sichten (Views). Das logische Modell repräsentiert den Problembereich (Problem Domain) in Klassen- und Objektstrukturen. Die Systemarchitektur, oder das statische Modell wird nach und nach aufgebaut und in einem **Klassendiagramm** dargestellt, das die Beziehungen der Objekte untereinander meistens nur statisch zeigt.

Ein **Objektdiagramm** mit den Beziehungen der Objekte untereinander beschreibt die dynamischen Eigenschaften eines Systems und zeigt, wie Objekte durch Nachrichtenaustausch (Messages Exchange) aufeinander wirken.

Eine **Modul- und Prozeßarchitektur** beschreibt die konkrete Hardware mit Rücksicht auf die Softwarekomponente des Systems, wie die physikalische Zuordnung der Klassen und Objekte zu den Modulen, den Prozessoren und Geräten und deren Kommunikationsanschlüssen untereinander.

Die objektorientierte Analyse und Design bei der Entwicklung von Softwaresystemen ist nach *Booch* einer **iterativer und inkrementeller Prozeß**, bei dem Kreativität eine sehr wichtige Rolle spielt. Er betont aber, daß es dafür kein Kochbuchrezept gibt. Er unterstützt in seiner Methode zwei Prozesse zur Beschreibung der OO-Softwareentwicklung:

1) Mikro-Entwicklungsprozeß

Dieser Prozeß besteht aus folgenden Phasen:

- Identifizierung von Klassen und Objekten in einer bestimmten Abstraktionsebene, wobei es grundsätzlich um das Finden von Gemeinsamkeiten, um Dinge zu gruppieren geht, die eine allgemeine Struktur oder Verhalten aufweisen
- Identifizierung der Semantik dieser Klassen und Objekte
- Identifizierung der Beziehungen unter Klassen und Objekten
- Spezifizierung der Schnittstellen und die Implementierung von Klassen und Objekten

2) Makro-Entwicklungsprozeß

Dieser Prozeß bildet den Kontrollrahmen für den Mikroentwicklungsprozeß und umfaßt folgende Phasen:

- Konzeption: In dieser Phase erfolgt das Festlegen der Kernanforderungen und der Vorstellungen von Prototypen. Dabei gibt es keine strengen Regeln sondern kreative schnelle Arbeit, um *quick-and-dirty* Prototypen zu erstellen, die am Abschluß dieser Phase weggeworfen werden.

- Analyse: Ein Modell für das Systemverhalten soll in dieser Phase entwickelt werden. Es steht dabei die Frage „was passiert?“ und nicht „wie passiert es?“ im Vordergrund. Entwickler des Systems arbeiten hier mit den Benutzern zusammen, um das Vokabular des Problembereichs zu verstehen und schnelle Konzepte zu finden. Die zwei Produkte dieser Phase sind: Anforderungsdefinition und Risikenabschätzung.

  Neben der Analyse des Problembereichs zur Identifizierung von Klassen und Objekten gehört zu den Aktivitäten dieser Phase eine Szenarienplanung. Dabei werden Objekte mit ihren Verbindungen und den ausgetauschten Nachrichten in einem Objektdiagramm nach der Notation von *Booch* dargestellt.

- Design: das Hauptprodukt der Designphase ist, den Entwurf einer Systemarchitektur für die Implementierung zu schaffen, der die allgemeinen taktischen Richtlinien beschreibt, wie z.B. Sicherungsmanagement, generelle Kontrollmechanismen und Strukturen.

Folgende Aktivitäten und Arbeitsgänge sind in der Designphase enthalten:

1) Architekturplanung der verschiedenen Systemebenen. Es erfolgt eine logische Einteilung des Systems in Form von Klassen und Objekten sowie eine physikalische Einteilung in Form von Zusammenfassungen von Modulen und Zuordnungen von Funktionen und Prozessoren

2) Taktisches Design: Es wird eine Numerierung der Richtlinien durchgeführt und anschließend ein Szenario für jede dieser Richtlinien entwickelt

3) Release-Planung: Hier erfolgt die Identifizierung einer kontrollierten Serie von freigegebenen Versionen (Releases) . Dabei ist folgende Reihenfolge von Aktivitäten typisch:

   $\Rightarrow$ Ordnung der Szenarien nach den Kriterien: „fundamental“ und „nebensächlich“

   $\Rightarrow$ Zuordnung der einzelnen Funktionalitäten zu einer Serie von Releases

   $\Rightarrow$ Festlegen von Terminen für die Releases

   $\Rightarrow$ Aufteilung der einzelnen Aufgaben auf die Entwickler und Programmierer

- Evolution der Implementierung durch sukzessive Verfeinerung: Hier ist die Wiederholung des Mikro-Prozesses die einzige Aktivität in dieser Phase. Mit Rücksicht auf die dadurch entstehenden Kosten können folgende Änderungen vorgenommen werden:

⇒ Neue Klassen: Diese Änderung kommt oft vor und sie ist nicht sehr teuer

⇒ Änderung der Implementierung einer Klasse: Diese Aktion ist auch nicht besonders teuer dank der Kapselung bei der OO-Implementierung

⇒ Neuorganisation der Klassenstruktur: Diese Aktion kann sehr teuer kommen vor allem, wenn sie in späteren Entwicklungsphasen vorgenommen wird

⇒ Änderung der Schnittstellen einer oder mehreren Klassen: Eine solche Maßnahme kann unter Umständen sehr teuer werden

- Instandsetzung/Wartung: In dieser Phase erfolgt einfach die nächste Iteration des Makro-Prozesses

Die Methode von Grady Booch ist sehr bekannt in der Welt der Software Entwicklung. Sie stellt viele Konzepte und eine Reihe von Praxistips vor, wobei die meisten davon nichts mit ausdrücklich objektorientierter Softwareentwicklung zu tun haben, sondern eher relevant für die Softwareentwicklung im allgemeinen sind.

4.4.3 Objektmodellierungstechnik OMT von Rumbaugh

James Rumbaugh hat eine Modellierungsverfahren OMT (Object Modeling Technique) entwickelt [63], das den objektorientierten Ansatz bei der Software-entwicklung in allen Phasen von der Analyse über das Design bis hin zur Implementierung unterstützt.

Die OMT-Methodologie besteht aus drei Phasen: die Analyse, das Systemdesign und das Objektdesign. Während dieser drei Phasen werden drei Modelle benutzt, die die Grundlage der OMT bilden:

1) Objektmodell

Das Objektmodell hält die statische Struktur eines Systems fest. Es ist das wichtigste Modell unter den drei Modellen. Im Objektmodell werden die für die Anwendung wichtigen Konzepte aus der realen Welt durch Klassen und Objekte mit Attributen und Beziehungen zueinander beschrieben. Dafür werden Objektdiagramme erstellt, die eine graphische Repräsentation (Notation) darstellen. Sie dienen zur abstrakten Modellierung und sind nützlich für das spätere Design.

Im allgemeinen ist es nicht sinnvoll, Klassen und Instanzen in einem Diagramm zu verwenden. Es ist daher zwischen zwei Objektdiagrammen zu unterscheiden:

- Klassendiagramme: Sie beschreiben Klassen als Schablonen (Templates) oder Pattern für viele mögliche Instanzen. Es werden Attribute und Operationen für die Klassen angegeben, aber ohne Implementierung der Operationen.

- und Instanzdiagramme: Sie beschreiben das Verhalten bestimmter Mengen von Objektinstanzen zueinander. Dabei sind die Werte der Attribute, aber nicht mehr die Operationen mit angegeben.

Objektdiagramme beinhalten verschiedene Informationen wie:

- Verknüpfungen und Assoziationen der Objekte untereinander
- Rollennamen zur Beschreibung der Objektaufgabe in der Assoziation
- Multiplizität zur Bestimmung der Anzahl von Instanzen einer Klasse, die in Verbindung mit einer Instanz einer assoziierten Klasse stehen
- Verknüpfungsattribute zur Bestimmung der Eigenschaften einer Assoziation
- Spezielle Attribute (Qualifier) für qualifizierte Assoziationen, um präzisere Aussagen über eine Assoziation zu erhalten
- Ordnung für nähere Spezifizierung der Assoziationen
- Aggregationen als spezielle Form der Assoziation
- Generalisierung und Vererbung
- Abstrakte Operationen
- Propagieren von Operationen zur automatischen Anwendung auf ein Netz von Objekten
- Klassenattribute und -Operationen
- Candidate Keys, die als Menge von Attributen zur Beschreibung der Assoziationen dienen, um die Mehrdeutigkeit bei n-ären Assoziationen zu umgehen.
- Einfache Einschränkungen der Beziehungen
- Abgeleitete Objekte, Verknüpfungen und Attribute um gegebenenfalls die Verständlichkeit des Modells zu erhöhen.

2) Dynamisches Modell

Das dynamische Modell vermittelt die Aspekte eines Systems, die sich mit Veränderungen über die Zeit befassen. Am wichtigsten dabei ist die dynamische Modellierung von Ereignisseen und Zuständen, um den sequentiellen Ablauf von Operationen darzustellen und den Kontrollfluß zu modellieren.

Zur Erstellung des dynamischen Modells werden die Veränderungen an Objekten und ihre Beziehungen untereinander über die Zeit untersucht. Dazu gehören Zustandsdiagramme, Szenarien und Ereignisverfolgungen (Event Traces), Ereignisflußdiagramme und Ereignisgeneralisierungsdiagramme.

- **Szenario und Ereignisverfolgungen (Event-Traces)**
  Man benutzt ein Szenario, um eine Sequenz von Ereignissen darzustellen, die zur Informationsübertragung von einem Objekt zu einem anderen dienen. Ein *Event-Trace* wird als Interaktionsdiagramm zwischen Sender- und Empfängerobjekt zur Erweiterung eines Szenarios benutzt.

- **Ereignisflußdiagramm**
  Sie fassen die Ereignisse zwischen Gruppen von Klassen zusammen und beschreiben ihre Ereignisflüsse

- **Zustandsdiagramme**
  Sie bestehen aus den drei folgenden Elementen:

  1) den Zuständen, die durch Attributwerte und Assoziationen eines Objekts bestimmt werden
  2) den Ereignissen, die durch Aktionen geschehen werden
  3) und den Transitionen als Zustandsänderung durch ein Ereignis.

- **Ereignisgeneralisierungsdiagramme**
  Hier werden die Ereignisse in einer Hierarchie von Ereignisattributen organisiert dargestellt. Es können verschiedene Abstraktionsebenen von Ereignissen gebildet werden, wodurch die Ereignisstruktur anschaulich wird.

3) Funktionales Modell

Das funktionale Modell beschreibt die Berechnungsoperationen in einem System und zeigt dabei, wie die Ausgabewerte von den externen Eingabewerten abgeleitet sind. Es spezifiziert die Berechnungsergebnisse ohne Rücksicht darauf, wie oder wann gerechnet wurde.

Das funktionale Modell besteht aus mehreren Datenflußdiagrammen, die die Bedeutung der Operationen und Einschränkungen des Objektmodells und der Aktionen des dynamischen Modells gibt.

Ein **Datenflußdiagramme DFD** (Data Flow Diagrams) ist eine grafische Darstellung, die den Datenfluß von Quellobjekten durch Prozesse, die die Daten verändern, zu den Zielobjekten zeigt. Es besteht aus folgenden Elementen:

- Prozesse (Processes), die die Daten verändern,

- Datenflüsse (Data Flows), die Daten miteinander verbinden, wie z. B. die Ausgabe eines Objekts oder Prozesses mit der Eingabe eines anderen,

- Actor-Objekte (Actor Objects), die in einem Datenflußdiagramm als aktive Objekte zum Produzieren und Verbrauchen von Daten dienen,

- Datenspeicher (Data Stores), die in einem Datenflußdiagramme als passive Objekte ohne Operationen für einen späteren Zugriff darzustellen sind,

- Kontrollflüsse (Control Flows), die hilfreich sind, um die Kontrollflüsse im funktionalen Modell darzustellen und die Abhängigkeiten der Funktionen zu zeigen.

Es folgt eine Übersicht über die Phasen der OMT-Methodologie:

1) Analyse

Die erste Phase der OMT ist die Analyse. Der Problembereich soll richtig verstanden und spezifiziert werden, um eine konzeptionelle Übersicht über das entwickelte System erarbeiten zu können. Dies gelingt durch Dialoge mit dem Systembenutzer

und Fachleute sowie dem Einsatz von Kenntnissen und Erfahrungen, um ein richtiges Verständnis der realen Welt, seiner Umgebung und Aufforderungen zu schaffen. Man gelangt am Ende zu einer möglichst vollständigen Beschreibung des Problembereichs und seinen wesentlichen Aspekten, die durch die drei o. g. Modelle dargestellt sind, ohne eine konkrete Implementierung vorzugeben. Es hängt dann von dem Problem selbst ab, welches das wichtigste Modell ist.

Diese Phase beinhaltet folgende Schritte:

- Problembeschreibung
- Erstellen des Objektmodells: Man geht folgendermaßen vor:
  - Objekte und Klassen bestimmen
  - Datenlexikon vorbereiten
  - Assoziationen zwischen den Objekten bestimmen
  - Bestimmung der Attribute von Objekten und Verknüpfungen
  - Organisation und Vereinfachung der Objektklassen mittels Vererbung auf zwei Wegen: Bottom-Up und Top-Down
  - Sicherstellen, daß für voraussichtliche Anfragen (Queries) Zugriffspfade existieren
  - Iterieren des Objektmodells um herauszustellen, ob

    $\Rightarrow$ es überflüssige Klassen gibt oder welche noch fehlen
    $\Rightarrow$ es überflüssige Assoziationen gibt oder welche noch fehlen
    $\Rightarrow$ Assoziationen falsch angesetzt sind
    $\Rightarrow$ Attribute falsch angesetzt sind

  - Einteilen der Klassen in Module (Sheets).
- Erstellen des dynamischen Modells: Dabei sind folgende Schritte auszuführen:
  - Erstellung von Szenarien für das Systemverhalten, die Dialoge zwischen System und Benutzer darstellen.
  - Bestimmung von Ereignissen und Aktionen an oder von Benutzern oder externen Geräten.
  - Erstellen von Ereignisflußdiagrammen, die Ereignisse zwischen Gruppen von Klassen darstellen. Jedes Ereignis überträgt Informationen von einem Objekt zu einem anderen.

- Erstellen von Zustandsdiagrammen, wobei man als erstes die Ereignis-Traces verwenden kann, die als Sequenz von Ereignissen während der Ausführung eines Systems auftritt.

- Anpassung von Ereignissen zwischen Objekten und Überprüfung der gesamten Arbeit auf Vollständigkeit und Korrektheit .

- Erstellen des funktionalen Modells: Als letztes wird das funktionale Modell mit der Ausführung folgender Schritte erstellt:

  - Bestimmung der Ein- und Ausgabewerte als Parameter der Ereignisse zwischen dem System und der Außenwelt.

  - Erstellung von Datenflußdiagrammen in Schichten je nach Detaillierungsgrad.

  - Beschreibung von Funktionen in natürlicher Sprache oder Pseudocode, ohne konkrete Implementierung in der Analyse vorzunehmen.

  - Bestimmung der Einschränkungen zwischen Objekten als Vor- und Nachbedingungen, die nicht durch Eingabe-Ausgabe-Abhängigkeiten verknüpft sind.

  - Optimierungskriterien des Systems während des Designs, zum Beispiel Speicher- oder Zeitbedarf.

- Hinzufügen von Operationen: Die Definition von allen Operationen ist oft sehr schwer, da die Liste der nützlichen Operationen unendlich lang ist. Die nächsten Schritte sollen bei der Aufstellung der Operationsliste helfen:

  - Operationen aus dem Objektmodell, implizit im Modell enthalten, brauchen aber nicht explizit dargestellt zu werden, wie Schreiben und Lesen von Attributwerten.

  - Operationen aus Ereignissen, die zu einem Objekt gesandt werden.

  - Operationen aus den Zustandsaktionen und -aktivitäten

  - Operationen von Funktionen

  - „Shopping-List“-Operationen, die einfach durch das Verhalten der Klassen in der realen Welt mit einbegriffen werden.

  - Vereinfachung von Operationen durch Identifizierung von ähnlichen Operationen, Erweiterung der Definition einer Operation oder Benutzung von Vererbung.

- Iterieren der Analyse durch mehr Durchgänge um Inkonsistenzen und Ungenauigkeiten in einem oder mehreren Teilmodellen zu korrigieren. Allerdings, die Grenze zwischen Analyse und Design ist nicht so scharf

definiert, daher soll die iteration zur Verbesserung der Analyse nicht sehr weit getrieben werden.

In der Phase der Analyse entsteht ein Analysedokument, das eine Problembeschreibung, ein Objektmodell, ein dynamisches Modell und ein funktionales Modell beinhaltet.

2) Systemdesign

In dieser Phase soll eine Gesamtstruktur für das System definiert werden. Es wird für eine Systemarchitektur als Gesamtorganisation des Systems entschieden, die in Komponenten, oder sogenannte Subsysteme aufgeteilt ist.

Folgenden Entscheidungen sind vom Systemdesigner zu treffen:

- Aufteilung des Systems in Subsysteme: Dabei können folgende Strategien verwendet werden:

  – Schichten: Ein Schichtsystem stellt eine Menge von virtuellen Welten dar, die jeweils als Ausdruck der darunterliegenden Schichten und als Basis für die Implementierung der darüberliegenden dienen.

    Die Architektur dieser Schichten kann je nach Möglichkeit des Zurückgriffs nur auf die darunterliegende Schicht geschlossen, oder auf jede darunterliegende offen sein.

  – Partitionen: Dabei wird das System vertikal in schwach gebundene unabhängige Subsysteme aufgeteilt.

- Identifizierung der Aktivitäten, bei denen Objekte parallel zusammenwirken, und Feststellung dieser Objekte, um Prozesse und Tasks mit den gehörigen Objekten zu definieren.

- Verwaltung der Datenspeicherung: Eine Entscheidung über die Benutzung von Dateien oder Datenbankmanagementsystemen (DBMS).

- Verwaltung des Zugriffs auf globale Ressourcen. Diese sind die physikalischen Einheiten wie Prozessoren, Bandlaufwerke, Kommunikationssatelliten usw., Platz wie Laufwerke, Terminal einer Workstation und Maustaste usw., logische Namen wie Dateinamen, Klassennamen usw., und Zugriff aus gemeinsame Daten wie Datenbanken.

- Implementierung der Softwarekontrolle: Es gibt grundsätzlich zwei Arten von Kontrollflüssen in einem Softwaresystem:

  1) Externe Kontrolle: Das ist der Fluß von externen sichtbaren Ereignissen zwischen den Objekten im System. Es gibt drei Arten der Kontrolle für die externen Ereignisse:

- Prozedurgesteuerte Systeme: Die Kontrolle findet im Programmcode selbst statt. Der Systemzustand ergibt sich implizit aus dem Programmzähler, dem Stack und den lokalen Variablen.

- Ereignisgesteuerte Systeme: Die Kontrolle findet innerhalb eines Dispatchers (Verteilers) oder Monitors statt. Der Dispatcher ruft die Prozeduren auf, wenn ein entsprechendes Ereignis auftritt und bekommt die Kontrolle von ihnen sofort nach der Ausführung wieder zurück.

- Parallele Systeme: Die Kontrolle findet parallel in verschiedenen unabhängigen Objekten statt, die jeweils ein Task an sich sind. Solche Systeme sind von den meisten Programmiersprachen, so auch von C++, schlecht oder gar nicht unterstützt.

2) Interne Kontrolle: Der Kontrollfluß ist innerhalb eines Prozesses. Das Verhalten ist dabei vorhersehbar und als Austausch von Ereignisse zwischen Objekten anzusehen, die als Prozeduraufrufe zu verstehen sind.

- Bearbeitung der Randbedingungen: Wie Initialisierung des Systemzustands, Terminierung eines Tasks und Fehler.

- Setzen von *trade-off* Prioritäten für Designentscheidungen, wie z.B. Geschwindigkeit, Speicherbedarf, Portabilität, Kosten usw.

- Anwenden von oder Aufbauen auf bekannte Systemarchitekturen, die schon allgemein in existierenden Systemen eingesetzt werden.

In der Phase des Objektdesigns entsteht ein Systemdesigndokument, das eine Struktur der Basisarchitektur des Systems und eine *High-Level*-strategische Entscheidungen beinhaltet.

3) Objektdesign

Im Systemdesign wurde die Herangehensweise für eine Implementierung festgelegt, in der Phase des Objektdesigns werden die vollständigen Definitionen der Klassen und Assoziationen für eine konkrete Implementierung bestimmt. Dabei werden Details hinzugefügt oder neue Klassen addiert.

Die Schritte in dieser Phase sind:

- Kombinationen der drei Modelle: Die Aktionen und Aktivitäten aus dem dynamischen Modell sowie die Prozesse aus dem funktionalen Modell müssen in Operationen der Klasse im Objektmodell gewandelt werden.

- Entwurf von Algorithmen: Die Kosten der Implementierung von Operationen soll dabei beachtet werden. Entscheidungen können aufgrund der Rechenkomplexität, der Einfachheit der Implementierung, der Flexibilität oder einer feinen Abstimmung „fine-tuning“ des Objektmodells (was wäre wenn anderes strukturiert wäre?) getroffen werden.

Der Designer von Algorithmen muß folgende Schritte ausführen:

- Wahl der Algorithmen
- Wahl der Datenstrukturen
- Definieren von internen Klassen und Operationen
- Zuweisen der Verantwortlichkeiten

- Designoptimierung: Das Analysemodell wird als Rahmen für die Implementierung benutzt und umfaßt die logische Informationen über das System, während das Designmodell zusätzliche Details für den effizienten Zugriff auf diese Informationen addiert.

  Ein Gleichgewicht zwischen Effizient und Klarheit ist bei der Optimierung des Modells zu beachten.

  Bei der Designoptimierung soll der Designer folgendes tun:

  - Hinzufügen von redundanten Assoziationen, um die Kosten der Zugriffe zu minimieren

  - Neuordnung der Ausführungsreihenfolge für eine größere Effizienz

  - Speicherung von abgeleiteten Attributen, um die Neuberechnung komplexer Ausdrücke zu vermeiden

- Kontrollimplementierung: Es soll eine Auswahl der Strategien zur Implementierung der Kontrolle erfolgen. Der Designer soll eine Basisstrategie zur Realisierung der Zustandsdiagramme aus dem dynamischen Modell auswählen und verfeinern.

Es gibt drei Methoden, um ein dynamisches Modell zu implementieren:

- Benutzung der Kontrollstelle in einem Programm, um den Programmzustand zu fassen und zu definieren (Prozedurgesteuertes System)

- Direkte Implementierung von einem Mechanismus für den Maschinenzustand (Ereignisgesteuertes System)

- Benutzung von parallelen Tasks (Parallele Tasks)

- Anpassung der Klassenstruktur zur Erhöhung der Vererbung: Dies wird durch folgende Schritte erreicht:

  - Neuordnung von Klassen und Operationen, um den Grad der Vererbung zu erhöhen

  - Abstrahierung gemeinsamen Verhaltens von Klassengruppen

  - Einsetzen von Delegationen um Verhalten zu teilen, wenn Vererbung semantisch ungültig ist

- Entwurf der Implementierung von Assoziationen: Für eine Implementierung der Assoziationen sollte man erst feststellen, wie diese benutzt werden. Man unterscheidet dabei zwischen folgenden Arten der Assoziationen:

  – „One-Way“-Assoziationen

  – „Two-Way“-Assoziationen

  – Assoziationen mit Verknüpfungsattributen

- Objektrepräsentation durch genaue Festlegung der Objektattribute

- Physikalische Aufteilung der Implementierung: Die Klassen und Assoziationen werden in Packungen (z.B. in Module) gepackt, um separate Weiterbearbeitung durch verschiedene Personen zu ermöglichen. Die Aufteilung in Packungen ermöglicht folgendes:

  – Verbergung von Informationen (Information Hiding)
  – Kohärenz von Entitäten
  – Aufbauen von physikalischen Modulen.

In der Phase des Objektdesigns ist es sehr wichtig, alle Entscheidungen mit einem Designdokument zu dokumentieren, das als Erweiterung des Analysedokument anzusehen ist, und beinhaltet ein detailliertes Objektmodell, detailliertes dynamisches Modell und detailliertes funktionales Modell.

- **Notationsübersicht**

In Abbildung 4.11 und 4.12 ist beispielsweise die Notation für das Objektmodell nach der OMT von Rumbaugh zum Dokumentieren aller Phasen der Softwareentwicklung, die oft im Rahmen dieser Arbeit auftauchen werden.

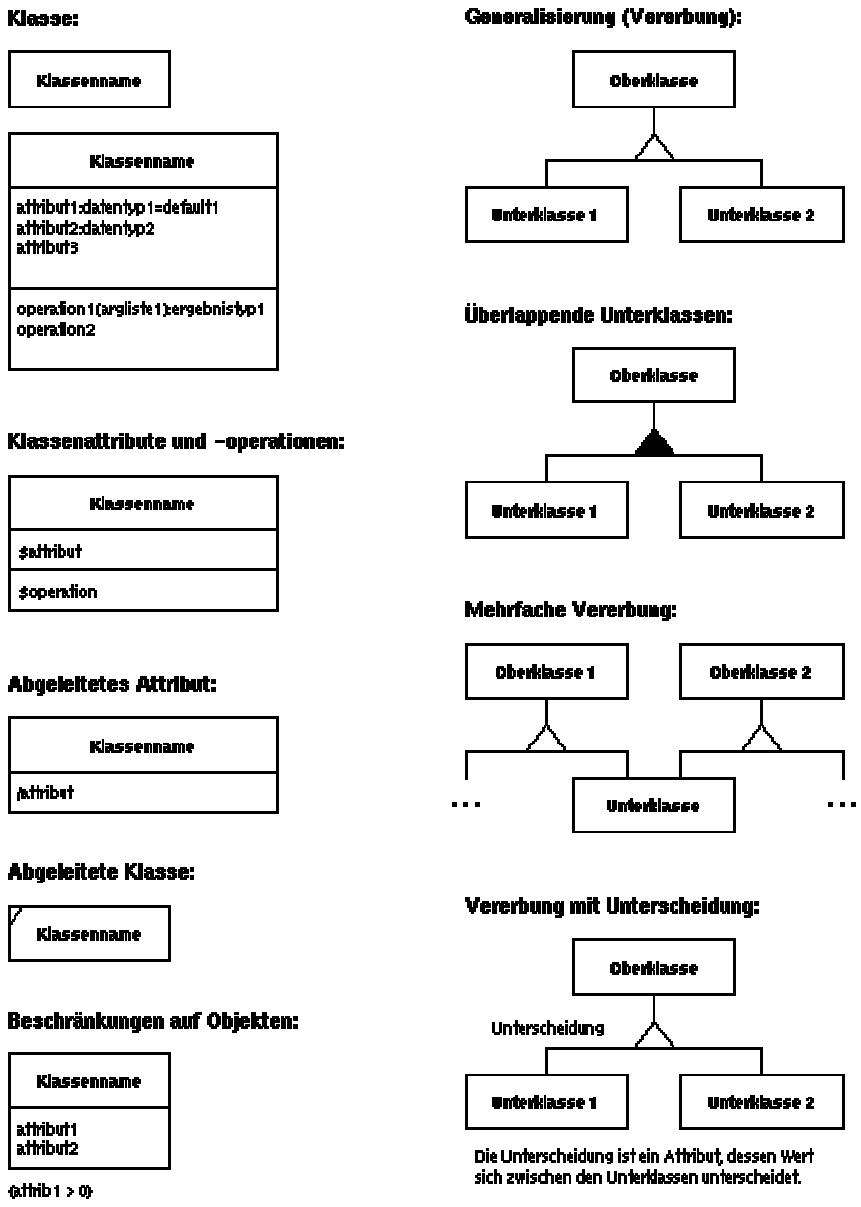


Abbildung 4.11: Notation für das Objektmodell nach der OMT von Rumbaugh [63]

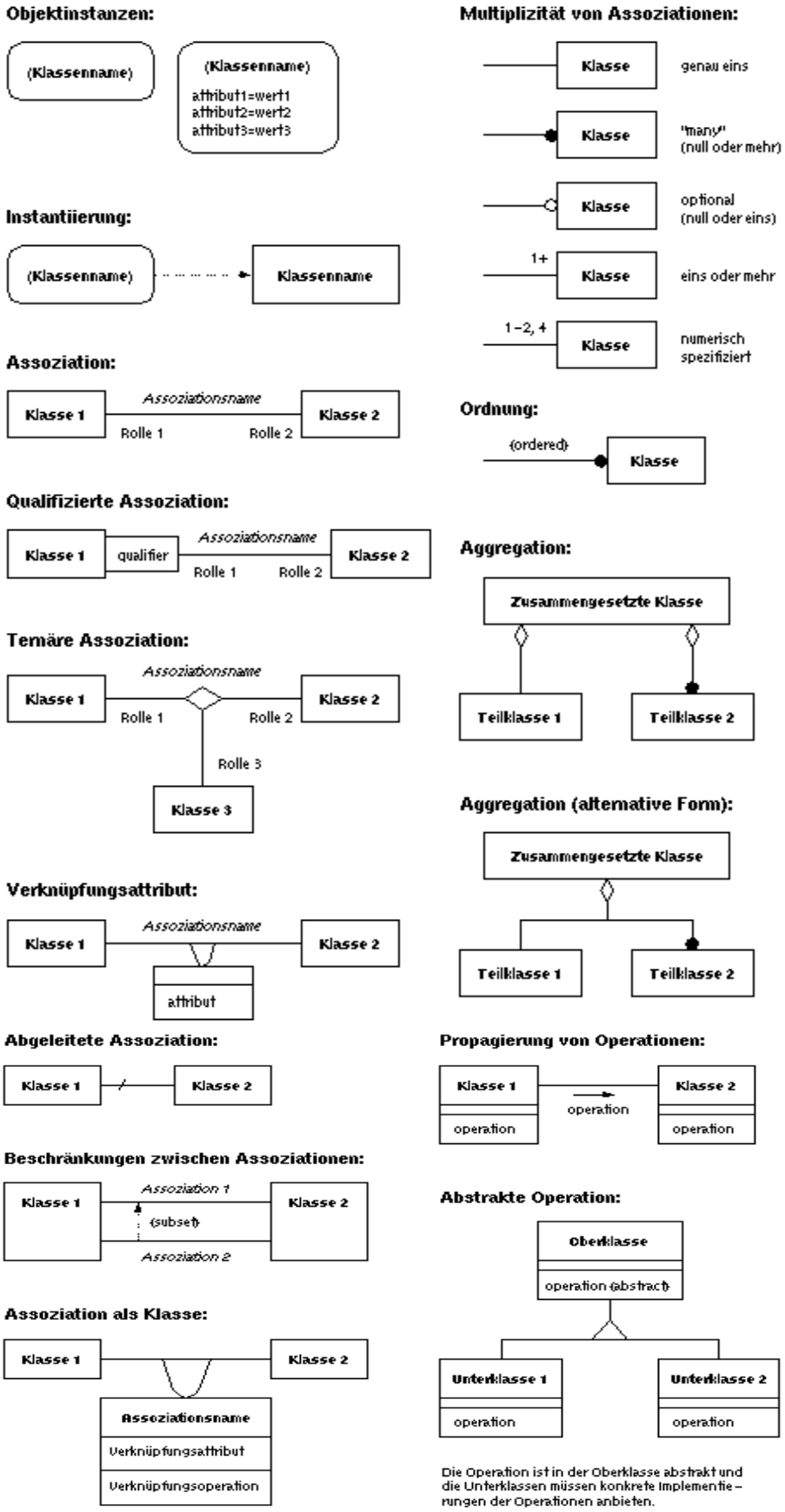


Abbildung 4.12: Notation für das Objektmodell nach der OMT von Rumbaugh [63]

4.5 Vorteile und Risiken von objektorientierter Entwicklung

Der Entwicklungsprozeß eines Softwareprodukts ist oft kompliziert und schwierig. Ein erfolgreiches Softwareprodukt läßt sich durch die Erfüllung der an sein Datenmodell gestellten Anforderungen charakterisieren.

Im allgemeinen gibt es kein fertiges Kochbuchrezept für die Entwicklung eines objektorientierten Datenmodells, wie Booch [7] sagt. Dies ist zurückzuführen auf die banale Weisheit, daß viele Wege nach Rom führen, aber ein gutes Datenmodell bezeichnet sich durch folgende Merkmale:

- **Das Datenmodell hat eine gut strukturierte Architektur**: Eine Architektur besteht aus einer Klassen- und einer Objektstruktur in verschiedenen (Abstraktions-) Ebenen und Teilen. Die Architektur ist unerheblich für den Endbenutzer, jedoch wichtig für ein verständliches, erweiterbares und neugestaltbares System. Es gibt zwar keine eindeutig „richtige“ Architektur für ein System, aber man kann durchaus gewisse Unterscheidungen zwischen guten und schlechten Architekturen treffen. So kann man einen objektorientierten Ansatz ohne weiteres als einen guten Anfang bezeichnen. Weitere Gemeinsamkeiten guter Software-Architekturen sind:

⇒ Sie setzen sich aus wohldefinierten Abstraktionsebenen zusammen, in der jede Ebene mittels einer wohldefinierten Schnittstelle eine klare Abstraktion repräsentiert.

⇒ Es gibt in jeder Ebene eine klare Trennung zwischen Schnittstelle und Implementierung, dadurch ist die Implementierung änderbar, ohne dabei die Annahmen der Kunden zu verletzen.

⇒ Einfache Architektur: Alltägliches Verhalten führt zu alltäglichen Abstraktionen und alltäglichen Mechanismen.

- **Das Datenmodell hat einen inkrementellen und iterativen Lebenszyklus:** Ein erfolgreicher Entwicklungsprozeß eines Datenmodells ist iterativ und inkrementell. Iterativ heißt, die objektorientierte Architektur wird sukzessive verfeinert. Die Erfahrung und die Ergebnisse jeder Version führen zur nächsten Iteration von Analyse und Design. Inkrementell heißt, jede Iteration der Analyse-Design-Evolution führt zu einer Verfeinerung und Verbesserung der strategischen und taktischen Entscheidungen. Man erhält dabei eine Lösung, die den Benutzeranforderungen immer näher kommt, aber trotzdem einfach, zuverlässig und anpassungsfähig ist. Diese Vorgehensweise steht im Gegensatz zum traditionellen Wasserfallmodell. Es handelt sich nicht um einen Top-Down- oder um einen Bottom-Up-Prozeß, sondern vielmehr um ein sogenanntes. ,“round-trip design“, d.h. um eine Verfeinerung der verschiedenen (noch immer konsistenten) logischen und physikalischen Sichten des Modells.

Folgende Vorteile objektorientierter Entwicklung sind zu nennen [7]:

- Fähigkeiten von objektorientierten Programmiersprachen können direkt eingesetzt und ausgenutzt werden, die einmal in dem erreichten Abstraktionsniveau und zum anderen in der einfachen Wartung und Erweiterbarkeit von Systemen liegen.

- Die Wiederverwertung von Softwarekomponenten wird unterstützt.

- Flexibilität und Leichtigkeit beim Vornehmen von Änderungen oder Erweiterungen existierender Systemen. Das Hinzufügen neuer Objekte ist problemlos möglich, da nur neue Unterklassen mit den Methoden definiert werden müssen.

- Die Entwicklungsrisiken werden weniger bei bewußter OO-Entwicklung.

- Andere Vorteile. Insbesondere reduziert der objektorientierte Ansatz die Entwicklungszeit und die Größe des Source-Codes. Dies wurde durch mehrere Fallstudien belegt.

Echte Nachteile gibt es bei objektorientierter Entwicklung nicht. Es können allerdings leicht welche entstehen, wenn man sich der Risiken dieses Vorgehens nicht bewußt ist.

Einige Risiken sind so vorzustellen:

- Performance. Das Senden von Nachrichten an die Objekte kostet Zeit. Daher haben objektorientierte Systeme üblicherweise gewisse Performance-Nachteile gegenüber herkömmlichen Programmen. Grund dafür ist das dynamische Binden von Methoden, das ca. 1.75 bis 2.5 mal mehr Zeit als ein einfacher Unterprogrammaufruf kostet. Andererseits sind in existierenden Systemen meist nur ca. 20% der Methodenaufrufe dynamisch. Bei streng getypten Sprachen und guten Compilern kann ein Großteil der Methodenaufrufe statisch gelöst werden.

  Der Aufbau des Systems in verschiedenen Abstraktionsebenen bringt ein weiteres Risiko mit sich. Ein Methodenaufruf einer höheren Ebene kann einen Methodenaufruf einer tieferen Ebene zur Folge haben, diese Methode kann eine Methode einer noch tieferen Ebene aufrufen und so weiter. Es entsteht eine Lawine von Methodenaufrufen, ohne daß wirklich irgendetwas passiert. Das kostet Zeit.

  Es gibt auch noch weitere Performance-Risiken, die aber nicht ernsthaft ins Gewicht fallen. Moderne Compiler schaffen in vielen Fällen schon selbst Abhilfe. Notfalls muß man das physikalische Design des Systems leicht ändern, ohne jedoch das logische Design anzugreifen.

- Anfangsschwierigkeiten. Aller Anfang ist schwer. Eine Umstellung auf objektorientierte Methoden verursacht am Anfang möglicherweise etwas höhere Kosten. Objektorientierte Sprachen kann man nicht in drei Tagen oder mit einem Buch wirklich lernen. Dazu ist unbedingt ein gutes Maß an Erfahrung nötig.

4.6 Hilfsmittel für den objektorientierten Entwurf (CASE-Tools)

Der Begriff CASE oder CASE-Werkzeug (Computer Aided Software Engineering) bezeichnet ganz allgemein die Menge der Computerprogramme, deren Aufgabe in der Unterstützung von Entwicklern beim Erstellen von Software liegt [40].

Man unterscheidet oftmals zwischen **Upper-CASE** und **Lower-CASE**. Upper-CASE bezeichnet diejenigen CASE-Werkzeuge, welche die Unterstützung der Phasen-Analyse und Entwurf zum Ziel haben. Diese Werkzeuge bilden Software Engineering -Methoden wie z. B. SA, OOA, etc. ab. Unter Lower-CASE versteht man die Werkzeuge der Programmierumgebung wie Compiler, Debugger, etc.

Die Zielgruppe von CASE-Werkzeug leitet sich aus dem Anspruch ab, den ganzen projektbezogenen Software-Lebenszyklus eines objektorientierten Programms zu begleiten. Somit ist die Gruppe der Adressaten durchaus heterogen. Im wesentlichen können die Adressaten von CASE-Werkzeugen in folgende Gruppen unterschieden werden.

- Projektleiter
- Analytiker
- Designer
- Programmierer

Teamarbeit ist in größeren Projekten eine unbedingte Voraussetzung zum Erfolg. Diese setzt seitens des Projektleiters eine intigrierte und konsistente Verwaltung der Arbeit der Teammitglieder im CASE-Werkzeug voraus. Hierbei kann CASE-Werkzeug eine Unterstützung anbieten, indem die Arbeitsergebnisse einzelnen Gruppen oder Personen eindeutig zuzuordnen sind.

Dies ist um so wichtiger, wenn Bausteine vergangener Projekte oder Drittanbieter wiederverwendet werden sollen.

Die Systemanalytiker beschäftigen sich mit der Analyse betrieblicher und technischer Abläufe mit dem Ziel, zu einem Modell zu gelangen, in welchem sich bestimmte Teile oder das System als Ganzes zu einer Automatisierung eignen. Hierzu muß der Analytiker die statischen und dynamischen Strukturen der realen Welt erkennen und sie in ein abstraktes Modell überführen. CASE-Werkzeug unterstützt den Systemanalytiker bei der Modellierung im Sinne der Objektorientierung.

Der Systemdesigner übernimmt das abstrakte Modell aus der Analyse und gibt ihm jene Eigenschaften, welche für die Abbildung des Modells auf einem Computer notwendig sind. Er spezifiziert die Daten- und Prozeßstruktur im Detail und bereitet so die eigentliche Programmierung, d.h. die Erstellung des Quelltextes vor.

Der Programmierer erhält die inhaltliche Spezifikation aus der Analyse ebenso wie das Strukturmodell aus dem Design und wandelt sie in einem lauffähigen Quelltext

(in diesem Fall C++ Quelltext) um. Er benötigt mächtige Werkzeuge zur Quelltext-Generierung, zum Editieren, zum Kompilieren und zum Testen (Debuggen). CASE-Werkzeug unterstützt diese Aktivitäten durch eigene Werkzeuge und durch die Integration anderer Werkzeuge wie Compiler, Debugger, Prototyper, etc. unter seiner einheitlichen Oberfläche.

Der Einsatz von CASE-Werkzeug kann darüber hinaus auch aus anderem Interesse erfolgen. Beispielhaft für viele andere sollen hier zwei weitere Zielgruppen vorgestellt werden:

Eine weitere wichtige Zielgruppe sind die Software Engineering Abteilungen und Methodenberater der Unternehmen. Diese wachen über die korrekte Einhaltung ihrer Spezifikationen zum generellen Einsatz von Methoden und Werkzeugen. Für sie, und selbstverständlich auch für alle anderen oben genannten Adressaten, ist die vollständige, richtige und zweckmäßige Dokumentation der Arbeitsergebnisse von hoher Bedeutung.

Ein sehr wichtiger Bereich ist die Ausbildung in den Unternehmen, an den Schulen und an den Universitäten/Hochschulen. CASE-Werkzeug kann Trainern helfen, neue Konzepte zu vermitteln. Den Teilnehmern von Seminaren, Schulungen, etc. kann die Objektorientierung am Beispiel und mit einem effizienten Werkzeug wesentlich intensiver vermittelt werden, als dies bei rein theoretischen Veranstaltungen möglich ist. So wird das *Know how* der Mitarbeiter für die Zukunft gesichert.

5. Datenmodell

In diesem Kapitel wird ein Datenmodell für die Tragwerksplanung vorgestellt. Dabei werden die realen Objekte der Tragwerksplanung von Gebäuden mit ihren Eigenschaften und Funktionalitäten sowie ihre Beziehung untereinander auf Softwareobjekte übertragen.

Die Bauobjekte werden identifiziert und in Klassen klassifiziert. Die dabei entstandenen Klassen stehen in einer Klassenhierarchie, in der Unterklassen von Basisklassen direkt oder nicht direkt abgeleitet werden. Die Klassen kann man auch in Klassenkategorien gliedern.

Die wichtigen Basisklassen der verschiedenen Klassenkategorien werden mit ihren Funktionalitäten und virtuellen Methoden beschrieben. Es wird auch auf die verschiedene Klassenkategorien im Datenmodell eingegangen. Dabei werden die Datenstruktur und die Methoden der Objekte erläutert.

Für die Datenhaltung und die Organisation der Objekte des Datenmodells werden im Rahmen dieser Arbeit besondere Mengenverwaltungsklassen entwickelt, die bei der Realisierung des Modellkonzepts im Kapitel *6* detailliert beschrieben sind. Dazu gehören dynamische Listen und Managerklassen, die die Objekte verschiedener Klassen vewalten.

Außerdem sind Klassen entwickelt, die die Basisfunktionalität übernehmen. Sie ermöglichen die Konstruktion und Positionierung von Objekten und entsprechen konventionellen Zeichnungshilfen wie z.B. Rasterfolien.

5.1 Anforderungen an das Datenmodell und dessen Abgrenzung

Die Umsetzung des im Kapitel *3* dargestellten Modellkonzeptes sowie die o. g. Anforderungen an das Datenmodell werden mit Hilfe der objektorientierten Technologie Kapitel *2* erreicht. Dabei wird sich für eine Vorgehenweise entschieden, die ähnlich der ist, die von Booch [7] beschrieben und die Makroentwicklungsprozeß genannt wird. Dieser Entwicklungsprozeß entspricht in etwa dem traditionellen Wasserfallmodell [40] und dient als steuernder Rahmen für den auch von *Booch* beschriebenen Mikroentwicklungsprozeß (Kapitel *4*). Dieser hat mehr Ähnlichkeit mit dem Spiralmodell von Boehm [6] und bildet den Rahmen für die iterative und inkrementelle Entwicklungsmethode. Eine typisch bekannte Reihenfolge von Arbeitsvorgängen ist die folgende:

- Identifizierung und Klassifizierung der Bauteiltypen, die die Komponenten der Tragstruktur eines Gebäudes repräsentieren, um deren Eigenschaften strukturgleich abbilden zu können.

- Identifizierung der Semantik der Bauteile und die Festsetzung von Attributen, Funktionalitäten und Abstraktionsebenen sowie die Entscheidung über die Vererbungshierarchie unter den verschiedenen Klassen (Mehrfach- oder Einfachvererbung). Verbindungen zwischen zwei Klassen können als Generalisierung/Spezialisierung oder Aggregationen identifiziert werden und ggf. werden neue Klassen erzeugt. Dabei sind Objektdiagramme bzw. Interaktionsdiagramme zur Beschreibung der Semantik der Szenarien (Klassendiagramme) sehr hilfreich.

- Bestimmung der Beziehungen unter den Objekten im Datenmodel, um sie durch Assoziationen/Aggregationen mit Kardinalität zu versehen, um anschließend eine Architektur für die Implementierung entsprechend der ausgewählten Programmiersprache C++ zu schaffen. Klassen werden hier zu Klassenkategorien (z. B. Tragwerksklassen oder Belastungsklassen) zusammengefaßt. Die getroffenen Entscheidungen werden geprüft. Die Verbindungen zwischen Klassen werden auf Notwendigkeit, Zweckmäßigkeit und Vollständigkeit genauer überprüft.

- Implementierung von Klassen und Objekten mit Hilfe der Programmiersprache C++.

- Implementierung einer objektorientierte grafisch-interaktive Benutzeroberfläche.

Im Rahmen dieser Arbeit werden die Bauteile, die in der Tragwerksplanung von Gebäuden vorkommen, zweckmäßig identifiziert und modelliert. In anderen Worten werden die realen Bauobjekte durch kompakte Softwareobjekte repräsentiert. Jedes dieser Objekte hat Informationen, die seine Struktur beschreibt und hat dazu Eigenschaften, die sein Verhalten bestimmen.

Im Bauwesen sind die physikalischen Eigenschaften, wie z.B. das Material, sowie die Geometrie von großer Bedeutung bei der Beschreibung von Bauteilen. Diese Informationen sind für den Tragwerksplaner sehr wichtig, um eine Strukturanalyse durchzuführen und zur Bestimmung der geeigneten Rechenmethode und deren Parameter.

Die objektpezifischen Daten (z.B. Material) und Eigenschaften (z.B. Methoden) sind in jedem Objekt verborgen. Diese Möglichkeit der Datenkapselung ist ein wichtiges Konzept der objektorientierten Modellierung und ist in der Programmiersprache C++ durch die Deklaration privat realisierbar.

Ein offensichtlicher Vorteil ist die dreidimensionale Modellierung der Bauobjekte, dies ermöglicht ein besseres Verständnis für die Tragstruktur durch die Möglichkeit, das Bauwerk aus verschiedenen Ansichten zu betrachten. Die für den Ingenieur wichtigen Daten der einzelnen Bauteile können dann in jeder Ansicht 2D/3D direkt abgefragt und gegebenenfalls modifiziert werden.

Das Tragwerksmodell besteht aus Objekten der Tragwerksplanung, die die verschiedenen Bauteile präsentieren, und die bei der Tragwerksplanung von Gebäuden vorkommen, wie: **Plattenbereich**, **Aussparung**, **Stütze**, **Unterzug**, **Linienlager**, **Bettung**, **Belastung**. Diese einzelnen Positionen beschreiben die

Tragstruktur eines Gebäudes, sie enthalten die Informationen über Geometrie, Material und ggf. Bemessungseigenschaften oder Belastungsgrößen [76].

Im Bauwesen ist die Geometrie der Bauteile einfacher als beim Maschinenbau. Die Beschreibung der Geometrie ist daher durch einfache Geometrie-Primitiven (Punkt , Linie, Polygon, ... etc.) mit den entsprechenden geometrischen Algorithmen (Schnittpunkt zweier Linien, Punkt innerhalb eines Polygons, Schnittpolygon einer Ebene mit einem 3D-Figur, .. etc.) und Grundoperationen möglich. Diese sind zwar in Software-Bibliotheken vorhanden und bei Software Herstellern erhältlich, werden aber im Rahmen dieser Arbeit aus folgenden Gründen selbst entwickelt:

- Der Polymorphismus als Konzept bei der objektorientierten Modellierung der geometrischen Objekte verlangt, daß sie alle aus einer Basisklasse abgeleitet werden, die zweckmäßig für das Datenmodell entwickelt ist.

- Man braucht nur bestimmte geometrische Primitiven, die die einfache Geometrie im Bauwesen beschreiben, aus denen die Bauteilobjekte zusammengesetzt werden können und nicht unbedingt abstrakte geometrische Modelle, wie B-Rep (Boundery-Representation) oder CSG (Constructive Solid Geometry).

- Übersichtliche und einfachere Anpassung der selbst entwickelten geometrischen Objekte zu dem Datenmodell (z.B. Mengenverwaltung, Schreib- und Lesefunktionen) und um bei der Integration zu einem CAD-Kern CAD-Funktionalitäten zu ermöglichen.

Die Daten des Tragwerksmodells können aus einem anderen Informationsmodell oder einem anderem CAD-System über eine dazu gedachte neutrale Austauschschnittstelle, z. B. über die Postdateien (*.pos) des Systems *MicroFE*, importiert werden. Dabei werden entsprechende Objekte des Tragwerksmodells generiert und dargestellt, die beliebige Bauwerkspositionen beschreiben. Die Tragstruktur wird in einer digitalen Datenbasis gemäß dem entwickelten Datenmodell aufgebaut.

Die Objekte besitzen die CAD-Funktionalitäten, wie Kopieren, Verschieben, Identifizieren, Fangen, usw., die es erlauben, verborgene Objektdaten wie Geometrie und bauspezifische Informationen (wie Material) zu modifizieren.

Eine interaktiv-grafische Benutzeroberfläche soll den Zugriff auf die Information jedes Objektes, und Änderungen ermöglichen.

An dieser Stelle sei aber noch einmal betont, daß die Kreativität bei der Entwicklung eine sehr wichtige Rolle spielt und es kein Kochbuchrezept geben kann, Booch [7].

Merkmale des entwickelten Datenmodells:

- Es ist ein objektorientiertes Datenmodell, das aus Klassen und Objekten besteht, die verschiedene Bauteile präsentieren, die in der Tragwerksplanung von Gebäuden und Hochhäusern vorkommen. Die Implementierung erfolgt mit der objektorientierten Programmiersprache C++.

- Kein abstraktes Datenmodell, d.h. kein Drahtmodell (B-Rep) und kein Volumen- oder Festkörpermodell (CSG). Es werden einfache geometrische 3D- und 2D-Objekte (Punkt, Linie, Polygon, Prisma etc.) als Grundelemente zur Beschreibung der Geometrie eingesetzt.

- Die Bauteile sind im Raum beschrieben, d.h. die 3D-Informationen über die Geometrie sind in jedem Bauteilobjekt enthalten. Dazu dienen die geometrischen Grundelemente als Bausteine bei der Bildung der Objektstruktur der Bauteilobjekte.

- Besondere Techniken und Mechanismen werden angesetzt, die beim Design einer Klassenbibliothek mit eng zusammengehörigen und aufeinander abgestimmten Klassen angewendet werden. Die Algorithmen zur Mengenverarbeitung oder zum Abspeichern und Laden werden so weit wie möglich zentral gelöst und dadurch die Implementierung einzelner Klassen so einfach wie möglich gemacht.

- Das Datenmodell nutzt den Polymorphismus als Konzept. In einer abstrakten Basisklasse wird zunächst festgelegt, welche Attribute und Eigenschaften alle aus dieser Basisklasse abgeleiteten Objekte generell besitzen (Abbildung 5.1). Dazu gehört z.B. eine eindeutige Identifizierungsnummer ID, sowie Funktionen zum Anlegen von Kopien, Abspeichern, Laden und zum Vergleichen von Objekten.

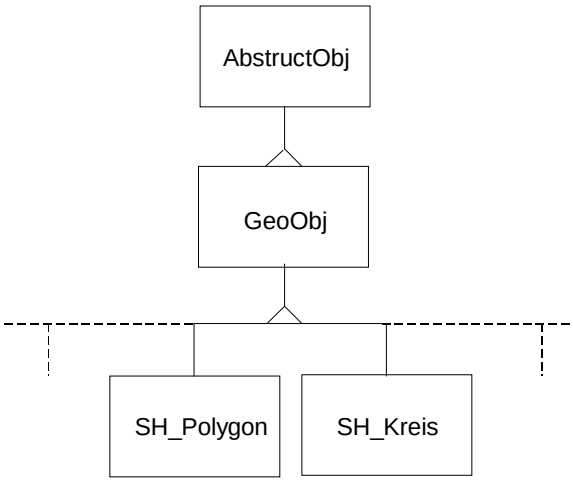


Abbildung 5.1: Einfache Vererbung der geometrischen Objekte

- Die Einfachvererbung ist als Vererbungsmechanismus bei der Bildung der geometrischen Objekte (Darstellungsobjekte) angewendet. Die Klasse *GeoObj* ist die Basisklasse für alle geometrischen Objekte . Sie legt fest, welche grafischen Attribute und Eigenschaften alle geometrischen Objekte besitzen.

- Da die 3D-Informationen über die Geometrie der Bauteile durch die geometrischen Primitiven beschrieben werden können, werden daher die Bauteile durch Aggregation („hat“) oder durch direkte Ableitung aus einem der grafischen Objekte gebildet. Die Bauteile werden durch Mehrfachvererbung aus zwei

verschiedenen Modellierungsmechanismen gebildet. Entscheidung darüber liegt bei dem Design einzelner Klassen und hängt mit der Funktionalität der entworfenen Klasse und ihren Datenelementen zusammen, das sie zu ihrer Präsentation braucht. Welcher Mechanismus eingesetzt wird, kann nur bei der Bildung jeder Klasse beantwortet, und bei der Implementierung getestet werden:

1. Durch direkte Ableitung aus einer grafischen Klasse und der bauspezifischen Klasse *Bauteil*. Damit besitzt das Bauteil, durch den Vererbungsmechanismus, alle Daten und Eigenschaften beider Basisklassen, sowohl grafische Eigenschaften und Funktionalitäten (z.B. Farbe, Strichdicke, Zeichenmethode, ... etc.) als auch bauspezifische Komponenten und Attributen (z.B. Material).

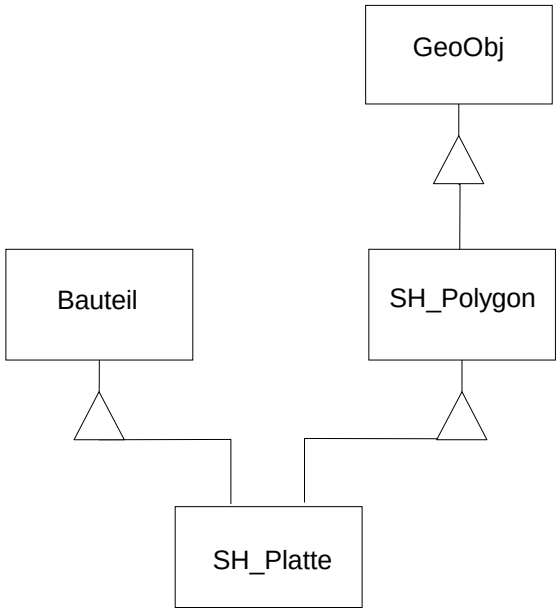


Abbildung 5.2: Mehrfache Vererbung der Tragwerksklassen

Der Vorteil dabei ist, daß die Member-Funktionen der Superklasse nicht noch mal in der Klasse implementiert werden sollen, sondern sie werden geerbt von der Superklasse. Somit sind diese Funktionen zu jedem Objekt dieser Klasse zur Verfügung gestellt und aufrufbar. Außerdem ist die gebildete Klasse wie eine Vereinigungsmenge aller Attribute, Methoden und Datentypen der Superklassen.

Bei der Bildung der Klasse *SH_Platte* ist z.B. die Klasse *SH_Polygon* als eine der beiden Superklassen, von denen *SH_Platte* abgeleitet ist (Abbildung 5.2) . Somit erbt *SH_Platte* alle Funktionen und Algorithmen von *SH_Polygon*, die bei dem Konstruieren der Platte durch einen Polygonzug oder zum mathematischen Operationen auf den polygonalen Umriß, der die Geometrie der Platte beschreibt.

2. Durch Kombination von Aggregation („hat“) und Vererbung aus der bauspezifischen Klasse *Bauteil* (Abbildung 5.3). Dabei hat das Bauteil ein grafisches Objekt als eine Grundkomponente (Member-Element) seiner Datenstruktur, die dann durch diese Grundkomponente und andere geerbte

Komponenten beschrieben wird, sowie zusätzliche Attribute, die die Besonderheiten des Bauteils bestimmen.

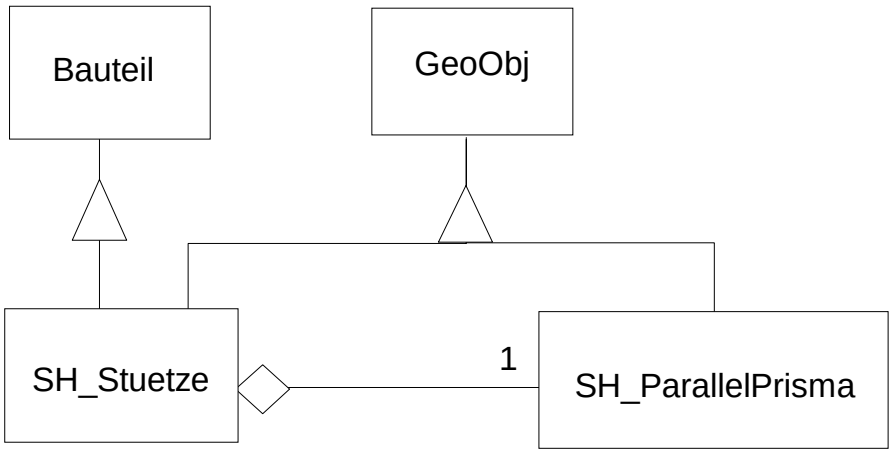


Abbildung 5.3: Aggregation der Tragwerksklassen

Der Nachteil bei diesem Mechanismus ist, daß die Member-Funktionen der Klasse des Member-Elements noch mal in der Klasse implementiert werden müssen, was mit Implementierungsaufwand verbunden ist. Auf der anderen Seite aber, können diese Funktionen nicht nur an das Member-Element weitergereicht sondern, je nach Aufgabestellung jeder Funktion, auch erweitert und manipuliert werden.

Bei der Bildung der Klasse *SH_Stuetze* wird z.B. ein Objekt der Klasse *SH_ParallelPrisma* als Member-Element in der Klasse aufgenommen. Die Klasse *SH_Stuetze* erbt von der Superklasse *GeoObj* die gleiche Funktionen wie die Klasse *SH_ParallelPrisma*. Somit sind diese Funktionen zu jedem Objekt der Klasse *SH_Stuetze* als auch der Klasse *SH_ParallelPrisma* zur Verfügung gestellt. In diesem Fall werden manche Funktionen an Member-Element *SH_ParallelPrisma* weitergereicht, andere aber werden neu implementiert oder manipuliert und dann an das Member-Element *SH_ParallelPrisma* weitergereicht.

Der Implementierungsaufwand, den man dabei ersparen kann oder doch treiben muß, hängt natürlich von den Aufgaben und der Funktionalität des benötigten Element-Member (Datenelement), das in der Klasse aufgenommen werden soll, sowie von der Komplexität dieses Datenelements selbst und die Funktionen, die an ihn weiter geleitet werden sollen. Die grafischen Informationen (Farben, Strichstärken etc.) sind in den geometrischen Objekten enthalten. Durch den Vererbungsmechanismus werden diese Informationen in den vom *GeoObj* direkt oder nicht direkt abgeleiteten Klassen geerbt. Dazu gehören auch die Funktionen zum Setzen und Verändern dieser Informationen.

5.2 Klassenhierarchie und Basisklassen

Alle Klassen des objektorientierten Datenmodells sind durch die Vererbung hierarchisch geordnet. Dies bedeutet, daß die Attribute, Methoden und Datentypen einer übergeordneten Klasse in den nachfolgegeordneten Klassen identisch verfügbar sind. Damit wird der Nutzen des Vererbungsmechanismus klar, indem die Eigenschaften, welche mehrere Klassen auszeichnen, nur einmal in einer Klassenhierarchie definiert werden müssen.

Die mehrfache oder multiple Vererbung ist eine Eigenschaft einiger objektorientierter Programmiersprachen, die es erlauben, daß eine Klasse mehrere übergeordnete Klassen hat. Die Programmiersprache C++ besitzt diese Eigenschaft der Vererbung und wird ausschließlich durch die Vererbung von Strukturen der Klassen und nicht von Werten unterstützt. Dabei ist die Subklasse die Vereinigungsmenge aller Attribute, Methoden und Datentypen der Superklassen.

Durch Polymorphie können die Funktionen in den abgeleiteten Klassen durchaus verschieden implementiert werden. So können Mengenklassen entworfen werden, die beliebige Objekte der abgeleiteten Klassen verwalten können, da sichergestellt ist, daß alle dazu notwendigen Eigenschaften der Objekte vorhanden sind. Muß die Menge z.B. abgespeichert werden, muß einfach nur für jedes Element die Funktion zum Abspeichern aufgerufen werden.

Die Klassen im Tragwerksmodell sind in folgende Klassenkategorien gegliedert:

- Darstellungsobjekte (Grafische Objekte)
- Tragwerksobjekte
- Belastungsobjekte
- Objekte für Bewehrungsfelder.

Objekte und Klassen im Datenmodell haben Beziehungen zueinander, die die Eigenschaften der Beziehungen in der realen Welt soweit wie möglich modellieren sollen. Wie man am besten ein Objekt aus der realen Welt durch Software-Objekt modelliert, ist oft schwierig zu beantworten. Dazu soll nicht nur die Struktur analysiert und verstanden werden, sondern auch die Beziehung mit den anderen Objekten im Datenmodell. Es wird schnell schwieriger, alle Beziehungen in Betracht zu nehmen, je komplizierter die Strukturdaten des Objekts und die Beziehung verchachtelt werden.

Ein gutes Prinzip bei dem Klassen-Design ist der Versuch, soweit wie möglich kleine Klassen mit bestimmten Funktionalität zu entwerfen. Objekte dieser Klassen können als Bausteine bei der Bildung komplizierter Klassen eingesetzt werden. Dabei können Objekte dieser Klassen (Bausteine) in der Datenstruktur der neu gebildeten Klassen als Member-Elemente „**hat**“, oder nur als Zeiger auf sie aufgenommen werden.

So wird z.B. bei dem Entwerfen der Darstellungsklassen von einer Klasse *Punkt* ausgegangen, die eine Position im 3D-Raum durch drei Koordinaten x, y, und z beschreibt. Dazu gehört es auch, einige Methoden und Allgorithmen zu definieren , die verschiedene mathematischen Operationen (wie z.B. Transformation) auf den Punktvektor {x, y, z} oder Veränderungen an der grafischen Darstellung eines Punkts ermöglichen.

Um nun eine Linie im 3D-Raum zu beschreiben, wird eine Klasse *Linie* entworfen, die zwei Member-Elemente vom Typ *Punkt* in ihrer Datenstruktur aufnimmt. Das heißt, ein Objekt der Klasse *Linie* besitzt „**hat**“ zwei Objekte der Klasse *Punkt*. Die mathematischen Operationen auf die Linie werden auf seine Punkte durchgefürt (wie z.B. Transformation). Die meisten Operationen auf die Linie können auf seine Member-Elemente, nämlich die zwei Punkt-Objekte, weitergereicht werden. Wenn nun das Objekt *Liniegelösch* wird, werden die zwei Punktobjekte automatisch mitgelöscht.

Der Fall ist anders bei der Beziehung zwischen einem Objekt der Klasse *Bauteil*, oder einer von ihr abgeleiteten Klasse, und dem Objekt der Klasse *Material*. Dabei hat das Objekt der Klasse *Bauteil* nur einen Zeiger (Englisch: Pointer) auf das Objekt *Material*, es ist keine Besitzbeziehung „**hat**“. Wenn nun ein Objekt der Klasse *Bauteil* gelöscht wird, bleibt das Objekt der Klasse *Material* erhalten. Dadurch kann man mehreren Bauteilen ein einziges Material zuordnen, somit ist jede Änderung an diesem Material automatisch bekannt bei allen Bauteilen, die Zeiger auf dieses Material haben.

Ein Objektdiagramm wird erstellt, um Beziehungen zwischen den Objekten im Datenmodell zu verdeutlichen. Es weden zwei Arten von Objektdiagrammen nach der Objektmodellierungstechnik OMT (Object Modelling Technique) von Rambaugh (siehe Kapitel *4*) unterschieden: Klassendiagramme und Instanzdiagramme.

Im Objektdiagramm des Tragwerksmodells werden konventionelle Bezeichnungen und Symbole nach der Objektmodellierungstechnik OMT von Rumbaugh benutzt (Abbildung 5.4). Dabei werden Beziehungen als Assoziationen oder Agrigationen mit ihrer Kardinalität beschrieben.

Das vorgestellte objektorientierte Datenmodell wird nach dem sogenannten Mechanismus Inside-out erstellt. Dabei entstehen Klassen, die mit ihrer internen Struktur eingebunden werden. Diese Klassen werden als Basisklassen bezeichnet, da sie eine Basisfunktionalität bereitstellen.

In den Basisklassen werden die Methoden und Eigenschaften definiert, die alle Objekte der abgeleiteten Klassen generell besitzen. In der Sprache der objektorientierten Modellierung sagt man: Die abgeleiteten Klassen „erben“ die Attribute, Methoden und Datentypen von der Basisklasse.

Im Rahmen dieser Arbeit wurden einige Basisklassen entwickelt, die die CAD-Funktionalitäten und andere wichtige Attribute und Methoden zur Verfügung stellen, welche die Struktur und das Verhalten der Bauteile spezifizieren.

Nachfolgend sollen wichtige Basisklassen erläutert werden.

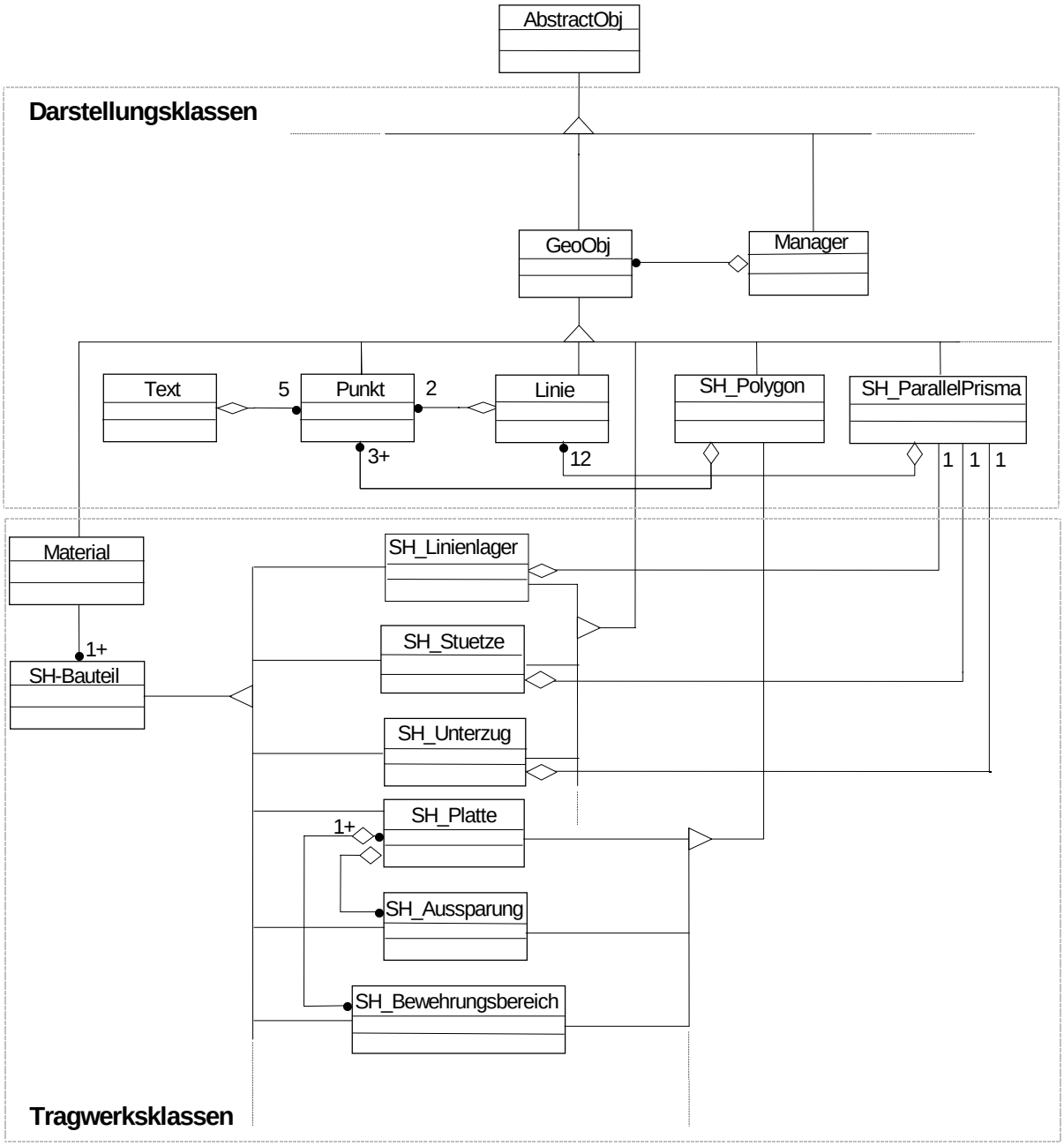


Abbildung 5.4: Objektdiagramm des Tragwerksmodells, nach Rambaugh (OMT)

5.2.1 Klasse *AbstractObj*

Diese Klasse ist eine Basisklasse für alle geometrischen Objekte. Hier wird die eindeutige Objekt-ID und der Klassenname des jeweiligen Objektes deklariert. Von dieser Klasse werden keine konkreten Objekte (Instanzen) erzeugt, sondern sie dient nur zur Zusammenfassung der oben genannten Eigenschaften für alle von ihr abgeleiteten Klassen. Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class AbstractObj
{
public:
        ...

protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
className	Klassenname	char	protected
objID	jeweilige Objekt-ID	const unsigned long	protected
maxObjID	aktuelle hoechste objekt-ID	static unsigned long	protected

5.2.2 Klasse GeoObj

Sie ist die Basisklasse für alle geometrischen Objekte. Sie legt fest, welche Eigenschaften alle geometrischen Objekte besitzen. Dazu gehören gemeinsame Attribute und Operationen, die für alle geometrischen Objekte aufgerufen werden können. Die Methoden und Operationen, die alle von dieser Klasse abgeleiteten Klassen erben, weden als virtuelle Funktionen durch das vorangestellte Syntaxwort virtual deklariert.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class GeoObj: public AbstractObj
{
public:
        ...
protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
owner	Besitzer des Objekts	Pointer auf *GeoObj*	protected
group	Gruppe des Objekts	Pointer auf *GeoObj*	protected
m_bVisibility	Sichtbarkeit des Objekts	BOOL	protected
m_bSelection	Markierung des Objekts	BOOL	protected
m_bActivity	Objekt ist aktive oder nicht	BOOL	protected
m_nPenColor	Stiftfarbe	int	protected
m_nPenStyle	Stifttyp	int	protected
orgLs	Eine dynamische Liste, in der das Objekt im Manager für Darstellungsobjekte eingetragen ist	pointer auf Liste mit Objekte vom Typ *GeoObj*	public

Folgende Methoden stehen generell für alle geometrischen Objekte zur Verfügung:

- **Methoden zur graphischen Darstellungsverwaltung**

Die Objekte, die graphisch dargestellt werden sollen, werden zur Laufzeit über Managerobjekte in einer verketteten Liste verwaltet. Folgende Methoden dienen dazu, ein Objekt dieser Klasse in einem Managerobjekt der Klasse *Manager* zu verwalten.

```
        ...
// Objekt zeichnen
        virtual void ObjDraw();

// Objekt verschieben
        virtual void ObjMove(Punkt moveP);

// Objekt in einen Objektmanager zur graphischen Darstellung einfügen
        virtual void ObjAdd(Manager*);

// Objekt aus einem Objektmanager entfernen
        virtual void ObjRemove(Manager*);

// Objekt wird bei der graphischen Darstellung angezeigt
        virtual void ObjShow();

// Objekt wird bei der graphischen Darstellung nicht angezeigt
        virtual void ObjHide();

// Objekt wird neu gezeichnet
        virtual void ObjUpdate();
        ...
```

- **Methoden zur Identifizierung der Owner-Objekte**

Geometrische Objekte können einen Besitzer (Owner) haben, eine Linie kann z.B. ein Parallelprisma als Besitzer haben. Ein Objekt kann auch zu einer Gruppe von geometrischen Objekten gehören, die gemeinsame Eigenschaften haben, und auf die gleichen Operationen durchgeführt werden. Im allgemeinen ist die Gruppe eines Objektes eine Folie (Layer) oder eine Ebene (Geschoß), in der es dargestellt ist.

```
GeoObj* GetOwner();
GeoObj* GetGroup();
void SetOwner(GeoObj* pOwnerObj);
void SetGroup(GeoObj* pGroupObj);
...
```

- **Methoden zur Veränderung der Darstellung**

// Farbe des Objektes abfragen
```
int GetPenColor() const;
```

// Linienstil des Objektes abfragen
```
int GetPenStyle() const;
```

// Farbe des Objektes ändern
```
void SetPenColor(int nPenColor) ;
```

// Linienstil des Objektes ändern
```
void SetPenStyle(int nPenStyle) ;
```

// Abfrage, ob Schalter auf Darstellen oder Verbergen steht
```
int Is_Visible() const;
```

// Abfrage, ob Schalter auf Aktiv oder Inaktiv steht
```
int Is_Activ() const;
```

// Schalter (Darstellen/Verbergen) ändern
```
virtual void SetVisibility(BOOL visiblmode);
```

// Schalter (Objekt ist Aktiv/Inaktiv) ändern
```
virtual void SetActivity(BOOL aktivmode);
...
```

- **Methoden zur Selektierung**

// Abfrage ob Schalter auf Selektiert oder Nicht selektiert steht
```
int Is_Selected() const;
```

// Schalter (Selektiert/Nicht selektiert) ändern
```
virtual void Select(BOOL selectmode);
```

- **Methoden zum Identifizieren und Fangen**

// Fragt ab, ob Objekt innerhalb einer Fangbox liegt
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);

// Fragt ab, ob Objekt in der Nähe eines Fangpunktes liegt
virtual BOOL Is_NearPoint(double x1, double y1, double radius);

// Fangpunkt wird auf einen Begrenzungspunkt eines Objektes gefangen
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);

// Fangpunkt wird auf eine Linie eines Objektes gefangen
virtual BOOL FangLinie(Punkt& fangPkt, double radius);

- **Methoden zur Ein-/ Ausgabe in einen Stream**

Diese Methoden dienen zum Lesen und Schreiben in eine formatierte Datei. Ausgabe erfolgt in Textformat (ASCII-Format).

// Lesen von einem Stream
virtual UINT ReadFrom(istream& strm);

// Schreiben in einen Stream
virtual void WriteOn(ostream& strm);
...
friend ostream& operator<< (ostream& os, const GeoObj& p);
friend istream& operator>> (istream& is, GeoObj& p);
...

- **Methoden für Ex- und Import**

Nach dem Modellkonzept soll jedes Objekt eine Import- und Exportfunktion als Schnittstelle zum Datenaustausch mit anderen CAD-Systeme besitzen. Die prototypisch folgenden implementierten Funktionen dienen beispielsweise zum Datenaustausch mit dem System *MicroFe* der Firma mb-Software im Bauwesen GmbH.

virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...

- **Methoden zum Einhängen eines Objektes in die Displayliste einer Folie**

Ein Objekt wird unterschiedlich dargestellt. Für jede Ansicht bestimmt das Objekt selbst , wie es dargestellt werden soll. Jede Folie hat eine dynamische Liste, in der ihre geometrischen Objekte verwaltet werden.

// Objekt in Displayliste einfügen
virtual void DrawInsert(SH_ViewObjManager, SH_ViewMap*);*

// Objekt aus Displayliste entfernen
virtual void DrawRemove(SH_ViewObjManager) ;*
...

- **Methode zum Duplizieren eines Objektes**

Ein neues Objekt wird mit den Eigenschaften des duplizierten Objektes erzeugt.

virtual inline GeoObj Clone ();*

- **Methode zum Öffnen eines Objektdialoges**

Ein Dialogobjekt wird erzeugt und Dialogbox dargestellt. In diesem Dialog kann man die Eigenschaften des Objektes ansehen und interaktiv ändern..

virtual void ExecDialog();
...

5.2.3 Klasse Material

In dieser Klasse werden die Materialeigenschaften eines Objektes des Tragwerksmodells beschrieben. Sie dient nicht als Basisklasse, sondern ist ein Objekt in der Datenstruktur (Member-Variable) der weiter unten beschriebenen Basisklasse *Bauteil*.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class Material: : public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_szMaterialName	Materialname. Wie z.B. Beton, Holz, ... etc.	char	protected
m_szMaterialBez	symbolischer Name. Wie z.B. B25, B35, NH II, etc.	char	protected
m_dE_Modul	Elastizitätsmodul	double	protected
m_dG_Modul	Schubmodul	double	protected
m_dDichte	Dichte	double	protected
m_Nue	Querkontraktion	double	protected

- **Methoden**

// Abfragen des Materialnamens
char GetMaterialName() const;*

// Abfragen der Kurzbezeichnung des Materials
char GetMaterialBez() const;*

// Abfragen des Schubmoduls
double GetSchubModul() const};

// Abfragen des Elastizitätsmoduls
double GetElastModul() const;

// Abfragen der Dichte
double GetDichte() const;

// Abfragen der Querdehnzahl
double GetNue() const ;

// Setzen der Materialeigenschaften
void SetMaterial(char mName, char* mBez, double roh,*
double E_Mod, double G_Mod, double nue);
...
...

5.2.4 Klasse Bauteil

Sie ist eine Basisklasse für alle Tragwerksobjekte. Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class Bauteil
{
public:
...
        ...
protected:
...
...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_szBauteilName	Bauteilname	char	protected
m_pMaterial	Materialwerte des Bauteils	Pointer auf *Material*	protected
m_bBem	Flag zeigt, ob das Bauteil mit oder ohne Bemessung	BOOL	protected
m_pBemessung	Bemessungsdaten des Bauteils	Pointer auf *SH_Bemessung*	public

Die Bemessungsdaten und Bemessungsverfahren können in der Klasse *SH_Bemessung* aufbewahrt werden, die folgende Datenstruktur hat:

```
class SH_Bemessung
{
public:
        CString m_strTyp;           // Bemessungsverfahren
        char* m_strDaten;           // Bemessungsparameter

        SH_Bemessung( )             // Konstruktor
        {
                m_strTyp = "";      // Initialisierung
                m_strDaten = " ";   // Initialisierung
        };

        ~SH_Bemessung( ) { };       // Destruktor
};
```

- **Methoden**

// Abfragen des Bauteilnamens
char GetBauteilName() const;*

// Bauteilname setzen
void SetBauteilName(char szBtName);*

// Abfragen der Materialdaten
Material GetMaterial() const;*

// Abfragen der Kurzbezeichnung des Materials
char GetMatBez()const ;*

// Gesamte Materialdaten neu setzen
void SetMaterial(Material newMatrial);*
...

5.3 Darstellungsobjekte

Die graphische Darstellung der Tragwerksobjekte in den Folien geschieht durch Darstellungsobjekte, die in der Datenmembern der Tragwerksobjekte enthalten sind. Sie sind in einer Hierarchie geordnet (Abbildung 5.5). Die Darstellungsobjekte werden je nach Darstellungsart des Tragwerksobjektes erzeugt. Eine Stütze wird z.B. durch ein Prisma *SH_ParallelPrisma* in der 3D-Sicht (View) präsentiert.

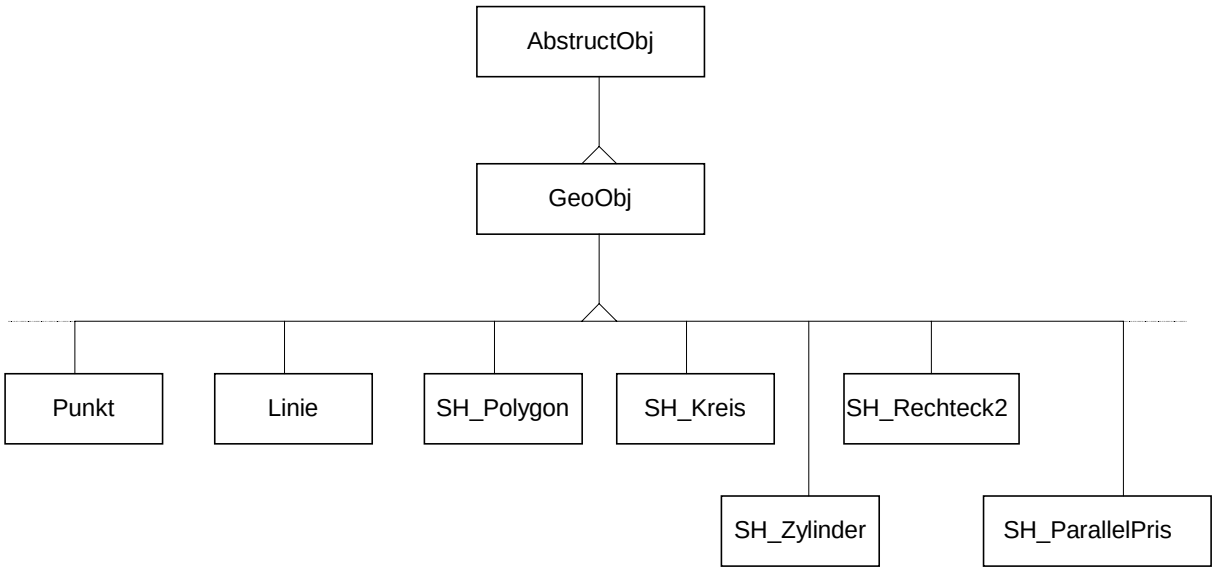


Abbildung 5.5: Klassenhierarchie der Darstellungsobjekte

Die Darstellungsobjekte dienen auch zum Erhalten der grafischen und geometrischen Informationen des Tragwerksobjekts. Wenn ein Tragwerksobjekt ein Objekt dieser Klassen besitzt, werden die grafischen Daten des Tragwerksobjekts in diesem Objekt gehalten, die je nach Darstellungsart an die dazugehörigen Darstellungsobjekte weitergegeben wird

Nachfolgend sollen die Darstellungsklassen und ihre Methoden kurz erläutert werden.

5.3.1 Klasse Punkt

Die Klasse Punkt ist das Elementarste aller Objekte. Jedes weiteres geometrisches Objekt besitzt einen oder mehrere Punkte. Der Punkt präsentiert durch die Koordinaten x, y, und z eine Position im 3D-Raum (Abbildung 5.6).

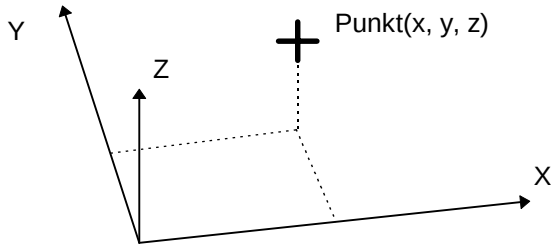


Abbildung 5.6: Ein Punkt beschreibt eine Position im 3D-Raum

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class Punkt : public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
nMarkerType	Darstellungsart des Punktes(Punktsymbol) Wie z.B. Kreuz, Kreis, Rechteck etc.	MARKER_TYPE	protected
dMarkerSizeX dMarkerSizeY symbolSizeX symbolSizeY	Größe des Punktsymboles	static double static double double double	protected
x	x-Koordinate	double	public
y	y-Koordinate	double	public
z	z-Koordinate	double	public

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Metohden zum Verändern der Koordinaten**

// Abfragen der Koordinaten

```
void SetXYZ(double xnew, double ynew, double znew);      Setzen der Koordinaten
void GetXYZ(double& px, double& py, double& pz);
...
```

Da die Koordinaten öffentliche Member der Klasse sind, kann man sie auch direkt ansprechen.

- **Methoden zum Verändern des Punktsymbols**

Ein Punkt kann mit verschiedenen Symbolen (Marker), wie z.B. ein kleines Kreuz, Rechteck, Kreis, etc. dargestellt werden. Dafür ist ein neuer Datentyp *MARKER_TYPE* definiert, wodurch der Punkt nach Wunsch unterschiedlich auf dem Bildschirm dargestellt werden kann.

```
// Markertyp setzen
    void SetMarkerTyp(MARKER_TYPE markTyp);

// Abfragen des Markertyps
    MARKER_TYPE GetMarkerTyp() const;
```

- **Geometrische Methoden**

Den Punkt kann man mit folgenden Methoden als Vektor ansprechen und auf ihm Operationen ausüben. Dafür stehen einige Funktionen zur Verfügung wie z.B.:

```
// Abstand zwischen zwei Punkte
    double PunktAbstand( const Punkt& p1, const Punkt& p2 )

// Betrag ermittlen
    double Abs() const;

// Quadratur des Vektors
    double AbsSqr() const;

// Normierung des Vektors
    int Normiere ();

// AbstandVonEbene: gibt den vorzeichenbehafteten Abstand von der Ebene zurück.
    double AbstandVonEbene( const Punkt& rEbenenPunkt, const Punkt&
                            rEbenenNormale ) const;
```

Es sind auch globale Funktionen defienirt, die geometrische Operationen erledigen aber nicht der Klasse gehören. Sie erledigen allgemeine Aufgaben und sind überall im Programm aufrufbar und bekannt, wie z.B.:

```
// für einen bestimmten Punkt P den Lotfusspunkt PktLotFuss auf der Linie
// berechnen. Globale Funktion.
    // Uebergabe: P relative Lage, Pa: Anfangpunkt, Pe: Endpunkt
    // Rückgabe wert :  = 0    Punkt liegt innerhalb der Gerade
    //                  = 1    Punkt liegt ausserhalb der Gerade
    int PunktLotFuss(Punkt& PktLotFuss, Punkt& P, double PaX, double PaY,
                     double PeX, double PeY)
```

- **Operatoren**

Die Notation der Operatoren ist die gleiche wie in C++.

```
...
Punkt& operator= (const Punkt&);
Punkt& operator- ();
Punkt& operator+= (const Punkt&);
Punkt& operator-= (const Punkt& p);
Punkt& operator*=(const double d);
...
...
```

Einige Funktionen und Operatoren sind als friend deklariert, damit können diese Funktionen auf die Daten der Klasse zugreifen, ohne daß sie Memberfunktionen der Klasse sein müßen.

```
// zwei Punkte zusammenaddieren
    friend Punkt operator+ (const Punkt& p1, const Punkt& p2);

// von einem Punkt subtrahieren
    friend Punkt& operator- (const Punkt&);

// zwei Punkte subtrahieren
    friend Punkt operator- ( const Punkt& p1, const Punkt& p2 );

// Skalarmultiplikation von links
    friend Punkt operator* (const double d, const Punkt& p);

// Skalarmultiplikation von rechts
    friend Punkt operator* (const Punkt& p, const double d);

// Skalarprodukt
    friend double operator* (const Punkt& p1, const Punkt& p2);

// Skalardivision
    friend Punkt& operator/ (const Punkt& p, const double d);
    ...
    ...
```

5.3.2 Klasse Linie

Eine Linie wird durch zwei Punkte definiert (Abbildung 5.7). Andere Attribute, zum Beispiel für die grafische Darstellung werden von der Basisklasse *GeoObj* geerbt.

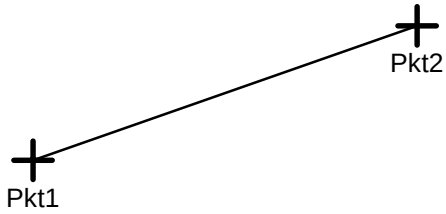


Abbildung 5.7: Eine Linie wird durch ihren Anfangs- und Endpunkt definiert

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class Linie : public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Pkt1	Anfangspunkt der Linie	Punkt	public
Pkt2	Endpunkt der Linie	Punkt	public

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);

virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
```

```
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Punkte**

```
// Anfangs- und Endpunkte der Linie setzen/verändern
    void SetP1(Punkt& newP);
    void SetP2(Punkt& newP);
    void SetPunkte(const Punkt& newP1, const Punkt& newP2);

// Linienlänge verändern, Punkt Pkt2 wird geändert
    void SetLaenge(double newL);

// Punkte abfragen
    Punkt& GetP1()  {return Pkt1;};
    Punkt& GetP2()      {return Pkt2;};
    void GetPunkte(Punkt& p1, Punkt& p2) const;
    void GetPunkte(Punkt* pP1, Punkt* pP2);
    ...

// Länge berechnen und abfragen
    double GetLaenge() const;

// Winkel zur x-Achse berechnen und abfragen
    double GetXAngle(int flag) const;
    ...
```

- **Geometrische Methoden**

```
// Schnitt der Linie mit einer Ebene
    int SchnittMitEbene( const Punkt& rEbenenPunkt, const Punkt&  rEbenenNormale,
    Punkt& rSchnittPunkt, double eps=EPS ) const;

// Schnitt mit einer Linie
    int SchnittMitLinie( const Linie& rLinie,Punkt& rSchnittPunkt,
                         double eps=EPS ) const;
    ...
    ...
```

5.3.3 Klasse SH_Polygon

Ein Polygon wird durch ein Feld von Punkten beschrieben (Abbildung 5.8). Die grafischen Attribute werden von der Basisklasse geerbt. Ob das Polygon geöffnet oder geschloßen ist, wird durch die Variable *m_bState* bestimmt (siehe folgende Tabelle).

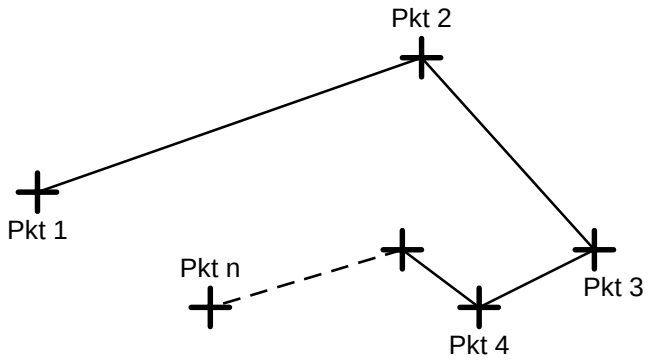


Abbildung 5.8: Ein Polygonzug von Punkten

Die Deklaration der Klasse *SH_Polygon* in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Polygon : public GeoObj
{
public:
        ...
protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_arrayPoints	Dynamische Liste der Punkte	SH_PUNKTARRAY	public
M_bState	Gibt an, ob ein Polygon geschlossen ist oder nicht TRUE = geschlossen, FALSE = geöffnet	BOOL	protected
m_bFillState	Ist ein Polygon geschlossen, kann es mit einer Farbe gefüllt werden: TRUE = gefüllt, FALSE = nicht gefüllt	BOOL	protected
m_nFillCoulor	Füllfarbe	int	protected

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Verwalten der Punktliste**

// Anzahl der Punkte abfragen

```
int NumPunkte() const;
```

// Ersten Punkt ermitteln

```
BOOL GetFirstPunkt( Punkt& rPunkt ) const;
```

// Letzten Punkt ermitteln

```
BOOL GetLastPunkt( Punkt& rPunkt ) const;
```

// Polygonpunkte ermitteln

```
long GetPoints(double (*xy)[2]);
```

// Punkt am Ende einfügen (Besitzer des Punkts ist jetzt das Polygon)

```
int AppendPunkt( Punkt& rPunkt);
int AppendPunkt( Punkt* rPunkt );
int AppendPunkt( double x, double y, double z );
```

// Punkt am Ende entfernen

```
int DeleteLastPunkt();
```

// Alle Punkte aus SH_Polygon löschen

void DeleteContents();
...

- **Methoden zum Verändern der Farb- und Geometrieeigenschaften**

// Polygon-Zustand(m_bState) setzen: geschlossen oder geöffnet
BOOL GetState() const;
void SetState(BOOL state);

// Füllzustand
BOOL GetFillState() const;
void SetFillState(BOOL fillState);

// Füllfarbe
void SetFillCoulor(int fillCoulor);
int GetFillCoulor() const;
...

- **Geometrische Methoden**

Das Polygon wird als Ebene betrachtet

// Normalenpunkt zurückgegeben.
Punkt GetNormale();

5.3.4 Klasse SH_Kreis

Ein Kreis wird durch zwei Punkte beschrieben. Die Linie, die durch die zwei Punkte gebildet wird, geht durch den Mittelpunkt des Kreises (Abbildung 5.9)

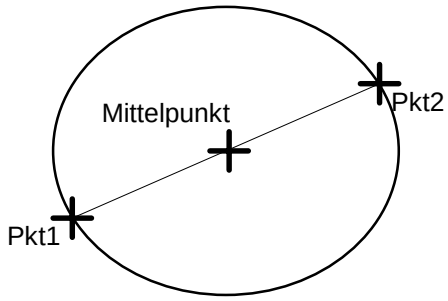


Abbildung 5.9: Ein Kreis

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Kreis : public GeoObj
{
public:
...

protected:
...

};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Pkt1	siehe Abbildung	Punkt	public
Pkt2	siehe Abbildung	Punkt	public

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Punkte**

// Punkte setzen/verändern

```
void SetP1(Punkt& newP);
void SetP2(Punkt& newP);
```

void SetPunkte(const Punkt& newP1, const Punkt& newP2);

// Durchmesser verändern, Pkt2 wird geändert
void SetDurchmesser(double newL);

// Punkte abfragen
Punkt& GetP1() {return Pkt1;};
Punkt& GetP2() {return Pkt2;};

// Beide Punkte abfragen
void GetPunkte(Punkt& p1, Punkt& p2) const;
void GetPunkte(Punkt pP1, Punkt* pP2);*

// Durchmesser abfragen
double GetDurchmesser() const;
...

5.3.5 Klasse SH_Zylinder

Ein Objekt dieser Klasse wird z.B. für die dreidimensionale Darstellung einer Rundstütze benötigt. Der Zylinder wird wie ein Kreis jedoch mit der Höhe beschrieben (Abbildung 5.10).

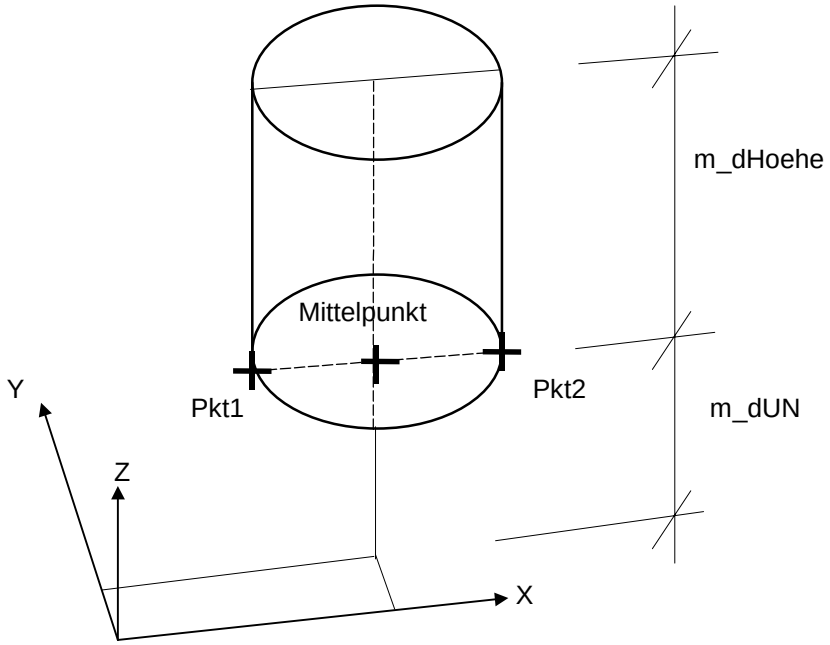


Abbildung 5.10: Zylinder

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Zylinder : public GeoObj
 {
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Pkt1	siehe Abbildung	Punkt	public
Pkt2	siehe Abbildung	Punkt	public
m_dHoehe	Die Höhe	double	public
m_dUN	Das untere Niveau	double	public

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
...
```

- **Methoden zum Setzen und Verändern der Parameter**

// Punkte setzen/verändern

void SetP1(Punkt& newP);
void SetP2(Punkt& newP);
void SetPunkte(const Punkt& newP1, const Punkt& newP2);

// Pkt1, Pkt2 oder beide abfragen

Punkt& GetP1() {return Pkt1;};
Punkt& GetP2() {return Pkt2;};
void GetPunkte(Punkt& p1, Punkt& p2) const;
void GetPunkte(Punkt pP1, Punkt* pP2);*

// Durchmesser berechnen und abfragen

double GetDurchmesser() const;

// Punkt Pkt2 über das Eingeben des Durchmessers verändern

void SetDurchmesser(double newL);
...

5.3.6 Klasse SH_Rechteck2D

Das Rechteck wird durch vier Linien beschrieben (Abbildung 5.11). Dadurch kann jede einzelne Seite des Rechtecks identifiziert und ihre unterschiedlichen grafischen Attribute wie Farbe, Linientyp oder Liniendicke zugewiesen werden.

Dieses Objekt wird z.B. für die zweidimensionale Darstellung einer Linienlagers benötigt.

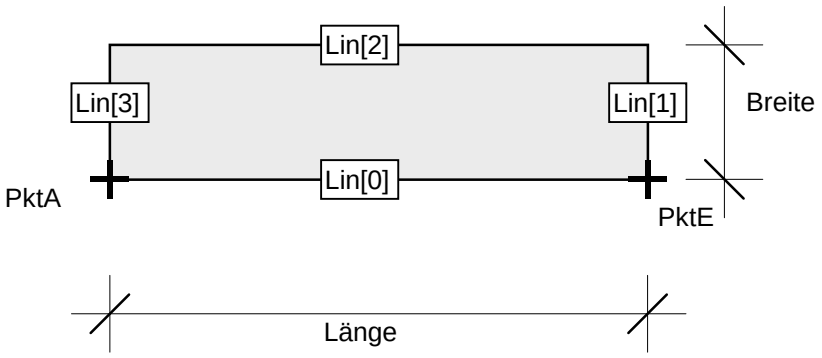


Abbildung 5.11: Rechteck mit 4 Linien

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Rechteck2D :   public GeoObj
{
public:
        ...
protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
PktA	unterer Linkspunkt	Punkt	protected
PktE	unterer Rechtspunkt	Punkt	protected
Lin[4]	Umrißlinien	Feld von 4 *Linie*	protected
Breite	siehe Abbildung	double	protected

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Setzen und Verändern der Parameter**

// Parameter setzen. Die Linien werden dabei auch gesetzt

```
void SetWd(const Punkt& anfP, const Punkt& endP, const double B);
void SetWd(const Punkt& anfP,const Punkt& anfPS, const Punkt& endP,const
                Punkt& endPS, const double B);
```

void SetLaenge(double newL);
void SetBreite(double newB);

// Anfangs- und Endpunkt holen
Punkt& GetPointA();
Punkt& GetPointE() {return PktE;};
void GetPunkte(Punkt& PA, Punkt& PE) const;

// Abfrage nach der Länge
double GetLaenge() const ;

// Abfrage nach der Breite
double GetBreite() const ;

// Abfrage nach dem Winkel zur x-Achse. Winkel wird berechnet und zurückgegeben
double GetXAngle(int flag) const;
...

5.3.7 Klasse SH_ParallelPrisma

Das Prisma wird durch zwölf Linien beschrieben (Abbildung 5.12). Dieses Objekt wird z.B. für die dreidimensionale Darstellung einer rechteckigen Stütze oder Linienlager benötigt.

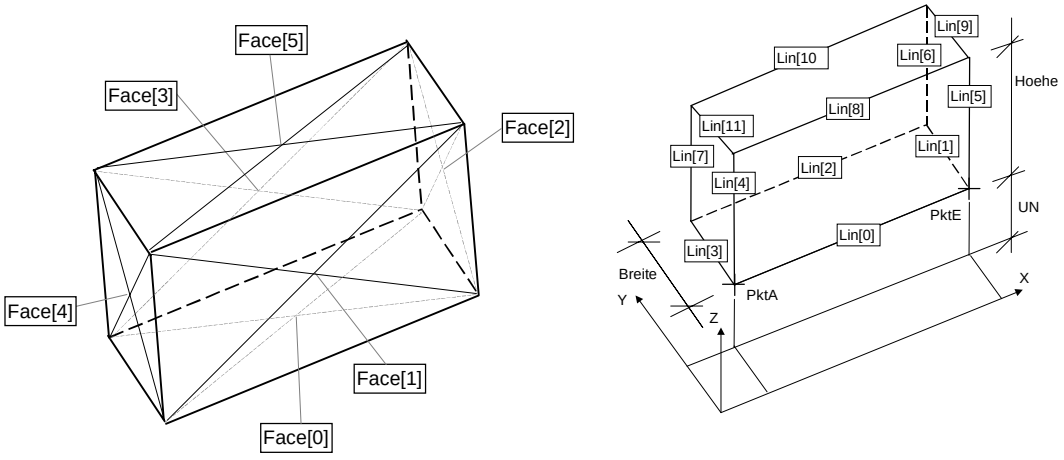


Abbildung 5.12: Ein Prisma wird durch 12 Linien modelliert

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_ParallelPrisma :  public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
PktA	Anfangspunkt (unten)	Punkt	protected
PktE	Endpunkt (unten)	Punkt	protected
Lin[12]	Die 12 Kanten	Feld von 12 *Linie*	protected
Hoehe	Höhe	double	protected
Breite	Breite	double	protected
UN	Das untere Niveau	double	protected

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier innerhalb der Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Parameter**

Mit diesen Methoden werden alle Parameter gesetzt. Die Linien werden dabei auch gesetzt.

```
void SetWd(const Punkt& anfP, const Punkt& endP, const double B,const double H,
        const double un);
void SetWd(const Punkt& anfP, const Punkt& anfPS, const Punkt& endP,
        const Punkt& endPS, const double B, const double H, const double un);
void SetLaenge(double newL);
void SetBreite(double newB);
void SetHoehe(double newH);
void SetUnterNiveau(double newUN);
...
double GetLaenge() const ;
double GetBreite() const ;
double GetHoehe() const;
double GetUnterNiveau() const ;
Punkt& GetPointA();
Punkt& GetPointE()  {return PktE;};
void GetPunkte(Punkt& PA, Punkt& PE) const;
void GetPoints(Punkt* points[24]);  Holt alle Start und Endpunkte der 12 Linien
double GetXAngle(int flag) const;     Winkel zur x-Achse berechnen und abfragen
...
// Holt die Flächen des Prismas
void GetFaces(SH_Polygon Faces[6]);

// Holt eine Fläche an der Stelle des angegebenen  Indexes
void GetFace(SH_Polygon Face ,int index);

// Holt Lin[index]
Linie& operator[]( int nIndex ) const;
...
```

- **Geometrische Methoden**

Die zugehörigen geometrischen Algorithmen und Operationen werden als Memberfunktionen deklariert, zum Beispiel wird die Schnittebene des Prismas mit einer Ebene als Polygonzug von Punkten dargestellt. Dafür ist folgende Methode zuständig:

```
int SchnittMitEbene( const Punkt& rEbenenPunkt, const Punkt& rEbenenNormale,
SH_Polygon& rSchnittPolygon, double eps= EPS ) const;
// Rückgabewert:
//      0: kein Schnitt                         3: Schnitt ist Fläche des Prismas
//      1: Schnitt ist Punkt des Prismas
//      2: Schnitt ist Kante des Prismas        4: echter Schnitt
```

5.3.8 Klasse SH_Text

Ein Text ist ein Zeichenkette mit Attributen und Ausgabeposition sowie Textausrichtung innerhalb eines Rechtecks.

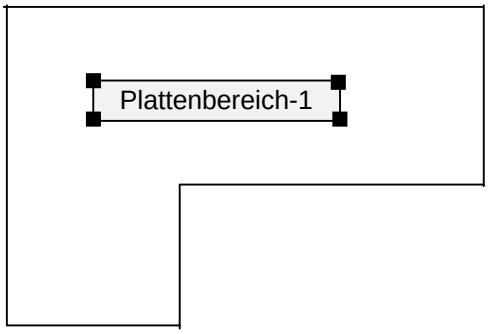


Abbildung 5.13: Ein Text wird in einem Rechteck definiert

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_Punkt	Anfangspunkt des Texts	*Punkt*	public
PktLu, PktRu, PktRo, PktLo	Eckpunkte des Rechtecks um den Text	*Punkt*	public
m_strText	Ein Textobjekt von MFC	*CString*	public
m_dAngle	Winkel des Texts mit der globalen x-Achse	Double	public
m_dSizeX	Zeichengröße in x-Richtung	Double	public
m_dSizeY	Zeichengröße in y-Richtung	Double	public
m_nTextAlign	Textausrichtung innerhalb des Rechtecks: 0 = linksbündig, unten 1 = linksbündig, Basislinie 2 = linksbündig, oben 3 = zentriert, unten 4 = zentriert, Basislinie 5 = zentriert, oben 6 = rechtsbündig, unten 7 = rechtsbündig, Basislinie 8 = rechtsbündig, oben	Int	public

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Text : public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als *virtual* deklariert sind, werden hier konkret für diese Klasse definiert, wie z.B. die folgenden Funktionen:

```
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Verändern/Abfragen des Texts und seinen Attributen**

// Punkte des Rechtecks um den Text setzen/abfragen

```
        void SetFangPunkte(Punkt& lu, Punkt& ru, Punkt& ro, Punkt& lo);
        void GetFangPunkte(Punkt& lu, Punkt& ru, Punkt& ro, Punkt& lo);
```

// Text verändern/abfragen

```
void SetText(char* pszText);
void SetText(CString str);
CString GetText() const;
char* GetText();
```

// Textausrichtung innerhalb des Rechtecks um den Text setzen/abfragen
void SetTextAlign(int nTextAlign);
int GetTextAlign() const;

// Textwinkel mit dem globalen x-Achse setzen/abfragen
void SetAngle(double dAngle);
double GetXAngle(int flag) const; // flag = 0 Winkel in Radian, sonst in Grad

// Zeichengröße des Texts setzen/abfragen
void SetCharHigh(double dHigh);
void SetCharWidth(double dWidth);
double GetCharHigh()const};
double GetCharWidth()cons;

// Textposition setzen/abfragen
void SetLocation(Punkt& pkt);
void GetLocation(Punkt& pkt);
void GetLocation(Punkt pPkt);*

// Textfarbe abfragen
COLORREF GetTextColor() const;

// Textlänge abfragen
int GetLaenge() const;
...

5.4 Tragwerksobjekte

Die Tragwerksobjekte sind die Softwareabbildung der realen Objekte, die in einem Geschoß im Gebäude vorkommen. Sie sind Bauteile, die Tragstruktur des Geschosses bilden, und in einer Hierarchie geordnet (Abbildung 5.14).

Die 3D-Informationen über die Geometrie der Bauteile werden durch die geometrischen Primitiven beschrieben. Daher werden die Bauteile durch Aggregation („hat“) oder durch direkte Ableitung aus einem der grafischen Objekte gebildet. Die grafischen Objekte (Darstellungsobjeket) dienen zur Darstellung der Tragwerksobjekte in den verschiedenen Sichten (Views).

Die Bauteile werden durch Mehrfachvererbung aus zwei verschiedenen Modellierungsmechanismen gebildet:

1. Direkte Ableitung aus einer grafischen Klasse und der bauspezifischen Klasse *Bauteil*. Damit besitzt das Bauteil, durch den Vererbungsmechanismus, alle Daten und Eigenschaften beider Basisklassen, sowohl grafische Eigenschaften und Funktionalitäten (z.B. Farbe, Strichdicke, Zeichenmethode, ... etc.) als auch bauspezifische Komponenten und Attributen (z.B. Material).

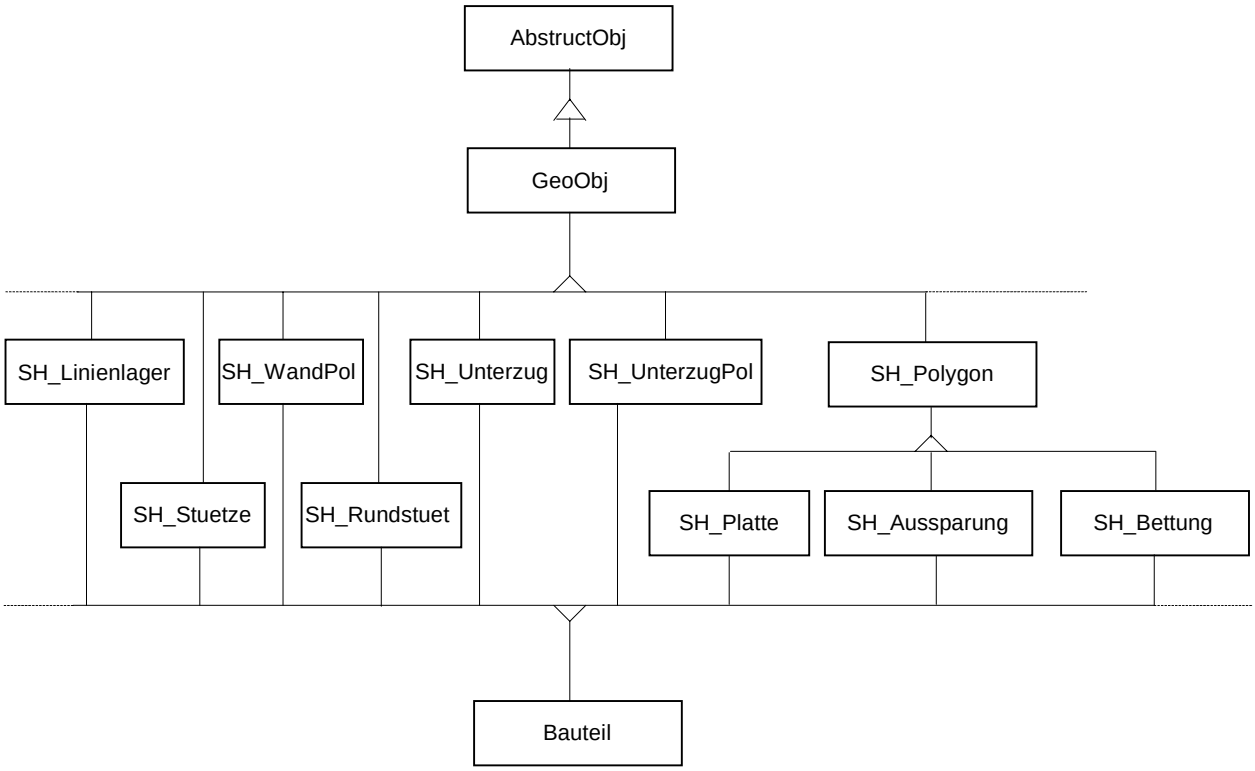


Abbildung 5.14: Klassenhierachie der Tragwerksklassen

2. Durch Kombination von Aggregation („hat“) und Vererbung aus der bauspezifischen Klasse *Bauteil*.

Die grafischen Informationen (Farben, Strichstärken etc.) werden durch den Vererbungsmechnismus vom *GeoObj* direkt geerbt. Dazu gehören auch die Funktionen zum Setzen und Verändern dieser Informationen.

Nachfolgend sollen die Tragwerksklassen und ihre Methoden erläutert werden.

5.4.1 Plattenbereich

Eine Klasse für den Plattenbereich *SH_Platte* ist von *SH_Polygon* und *Bauteil* abgeleitet und besitzt somit alle Eigenschaften und Methoden beider Klassen. Ein Plattenbereich kann somit durch die Eingabe eines Polygonzuges gesetzt werden und alle Funktionen eines Polygons direkt aufgerufen werden (Abbildung 5.15).

Zusätzlich wird einem Plattenbereich noch andere Eigenschaften, zum Beispiel einen Drillfaktor, konstante oder veränderliche Plattendicke zugeordnet, die als Sonderattribute der Klasse *SH_Platte* addiert werden. Das Niveau eines Plattenbereiches wird durch das Niveau des Stockwerks (*SH_Ebene*) bestimmt, zu dem dieser Plattenbereich zugeordnet ist.

Für Plattenbereiche mit veränderlicher Dicke, wird die untere schräge Seite der Platte durch drei Punkte definiert, die dann eine Ebene im 3D-Raum (Polygon) beschreiben. Die Plattendicke wird dann an jedem Polygonpunkt des Plattenbereiches als der Abstand zwischen der Plattenoberkante und dieser Ebene berechnet.

Eine Folie beschreibt im allgemeinen ein Geschoß oder eine Ebene (*SH_Ebene)*. Ein Plattenbereich kennzeichnet die Decke eines Geschosses.

Jeder Klasse *SH_Platte* ist eine dynamische Liste von Aussparungen *SH_Aussparung* zugeordnet (siehe die Klasse *SH_Aussparung* später in diesem Kapitel). Beim Setzen einer Aussparung muß erst ein Plattenbereich identifiziert werden, dem diese Aussparung zugeordnet wird. Dazu gehören Kontrollfunktionen zum Tasten, ob Aussparungen sich in einem Plattenbereich beim Setzen überlappen.

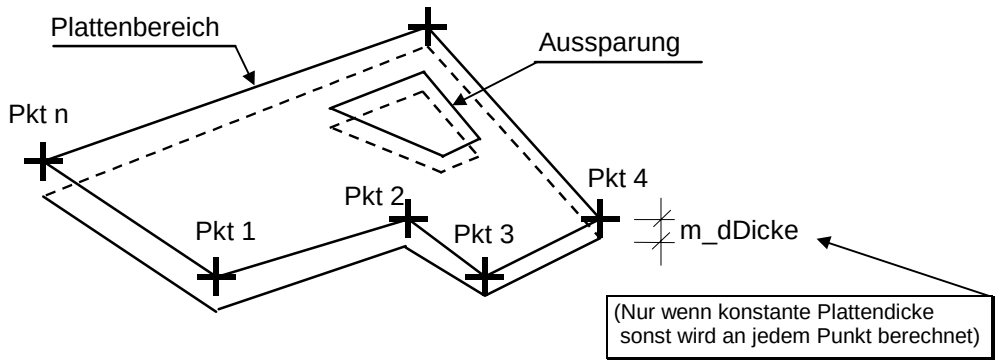


Abbildung 5.15: Plattenbereich mit Polygonzug zur Beschreibung der Geometrie

Wenn ein Plattenobjekt zerstört wird, sollen alle zugehörigen Aussparungsobjekte mit gelöscht werden. Beim Klassendesign heißt das, es besteht eine Beziehung (Assoziation) 1 zu n zwischen der Klasse *SH_Platte* und der Klasse *SH_Aussparung*.

Der Klasse *SH_Platte* ist auch eine dynamische Liste mit Zeigern auf Objekte vom Typ *SH_Bewehrungsbereich* zugeordnet (siehe Bewehrungsbereiche für die Beschreibung der Klasse *SH_ Bewehrungsbereich* später in diesem Kapitel).

Da ein Bewehrungsbereich nicht nur einer Platte zugeordnet sein könnte, soll darauf geachtet werden, daß beim Zerstören eines Objekts vom Typ *SH_Platte* nicht die dazu zugeordneten Objekte vom Typ *SH_Bewehrungsbereich* mit zerstört werden. Beim Design kann die Beziehung n zu n zwischen beiden Klassen modelliert werden.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Platte: public Bauteil, public SH_Polygon
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_dDicke	siehe Abbildung	double	protected
m_dDrillFaktor	Drillfaktor der Platte	double	protected
m_TopPolygon	siehe Abbildung	*SH_Polygon*	protected
m_arrayAusp	Dynamische Liste mit Aussparungen	Liste vom Typ *SH_Aussparung*	protected
m_arrayBew	Dynamische Liste mit Bewehrungsfelder	Liste vom Typ *SH_Bewehrungsbereich*	protected

Die Eigenschaften eines Objekts (Instanz) dieser Klasse werden durch folgenden Methoden (Funktionen) bestimmt:

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, und in der Klasse *SH_Polygon* implementiert, werden hier innerhalb der Klasse zum Teil nochmal definiert und zum Teil erweitert, um Operationen, z.B. auf die Aussparungen und andere Attribute, weiter zu führen. Solche Funktionen sind z.B.:

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
```

```
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);

...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
....
```

- **Methoden zum Verwalten der Punktliste**

Diese werden von der Klasse *SH_Polygon* geerbt, sie werden nicht in der Klasse noch mal implementiert und sind direkt aufrufbar (siehe die Klasse *SH_Polygon* vorher in diesem Kapitel).

- **Methoden als Schnittstelle zu den eigenen Daten der Klasse**

```
// Aktualisieren der Punkte des Polygons, falls konstante Dicke
    void UpdateTopPolygon();

// Aktualisieren des schiefen Polygons, falls die Dicke veränderlich ist.
    void UpdateSchiefPolygon(Punkt ebP1,Punkt ebP2,Punkt ebP3);

// Zugreifen auf die Attribute der Platte
    double GetDicke() const ; // falls konstante Dicke
    double GetDrillFaktor() const ;
    ...
    void SetDicke(double newD);
    void SetDrillFaktor(double dDrillFaktor);

// Polygon-Zustand setzen: geschlossen oder geöffnet
    BOOL GetState() const;
    void SetState(BOOL state);

// Füllzustand (Polygon ist gefüllt mit oder ohne Farbe)
    BOOL GetFillState() const;
    void SetFillState( BOOL fillState);
```

// Füllfarbe
void SetFillCoulor(int fillCoulor);
int GetFillCoulor() const;
...

- **Methoden zur Verwaltung der Aussparungenliste**

// neue Aussparung erzeugen und in die Liste eintragen
int AppendAusp(SH_Aussparung& rAusp);

// Aussparung in die Liste eintragen
int AppendAusp(SH_Aussparung pAusp);*

// Erste Aussparung ermitteln
BOOL GetFirstAusp(SH_Aussparung& rAusp) const;
SH_Aussparung GetFirstAusp();*

// Letzte Aussparung in der Liste ermitteln
BOOL GetLastAusp(SH_Aussparung& rAusp) const;
SH_Aussparung GetLastAusp();*

// Aussparung am Ende der Liste entfernen
int DeleteLastAusp();

// Alle Aussparungen in der Liste löschen
void DeleteAllAusp();
...

- **Methoden zum Überprüfen, ob Aussparungen sich überlappen.**

// Überprüfen, ob ein Punkt innerhalb einer Aussparung der Platte liegt
BOOL IsPunktInAusp(Punkt rPunkt);

// Überprüfen, ob eine Linie in einer Aussparung liegt
BOOL IsAuspInAusp(Linie& rSchnitt);
...

- **Methoden zur Verwaltung der zugeordneten Bewehrungsbereiche**

// Bewehrungsfeld in die Liste eintragen
*int AppendBew(*SH_Bewehrungsbereich * *p*Bew *);*

// neues Bewehrungsfeld erzeugen und in die Liste eintragen
*int AppendBew(*SH_Bewehrungsbereich *& rBew);*

// Erstes Bewehrungsfeld ermitteln
*BOOL GetFirst (*SH_Bewehrungsbereich *& rBew) const;*

SH_Bewehrungsbereich * *GetFirstBew();*

// Letztes Bewehrungsfeld in der Liste ermitteln
*BOOL GetLastBew(*SH_Bewehrungsbereich *& rBew) const;*
SH_Bewehrungsbereich * *GetLastBew();*

// Bewehrungsfeld am Ende der Liste entfernen
int RemoveLastBew();

// Alle Bewehrungsfelder in der Liste löschen
void DeleteAllBew();

// Überprüfen, ob ein Punkt innerhalb eines Bewehrungsfelds liegt
BOOL IsPunktInBew(Punkt rPunkt);
...

5.4.2 Aussparung

Die Klasse *SH_Aussparung* ist von *SH_Polygon* und *Bauteil* abgeleitet, sie besitzt somit alle Methoden und Eigenschaften beider Klassen. Zusätzlich wird eine Aussparung noch durch ihre Dicke bestimmt. Eine Aussparung kann durch die grafische Eingabe eines Polygonzuges gesetzt werden (Abbildung 5.16).

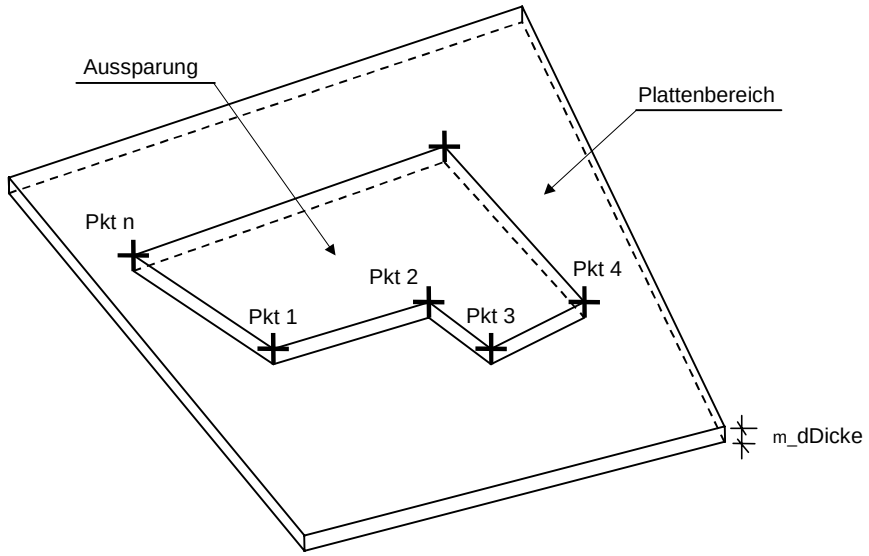


Abbildung 5.16: Aussparung

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Aussparung : public Bauteil, public SH_Polygon
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_dDicke	Dicke der Aussparung	double	protected
m_TopPolygon	oberes Polygon	*SH_Polygon*	protected

Die Eigenschaften eines Objekts (Instanz) dieser Klasse werden durch folgende Methoden (Funktionen) bestimmt:

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
...
```

- **Methoden zum Verwalten der Punktliste**

Diese werden von der Klasse *SH_Polygon* geerbt, sie brauchen nicht in der Klasse noch mal implementiert zu werden und sind direkt aufrufbar (siehe die Klasse *SH_Polygon* vorher in diesem Kapitel).

- **Methoden als Schnittstelle zu den eigenen Daten der Klasse**

// Aktualisieren der Punkte des oberen Polygon

```
double GetDicke() const ;
void SetDicke(double newD);
void UpdateTopPolygon();
...
```

Im Datenmodell sind zwei Stützen aus Stahlbeton mit verschiedenen Querschnitten modelliert, und zwar Rechteck- und Rundstütze. Stützen mit anderen Querschnittsformen lassen sich analog modellieren und implementieren.

5.4.3 Stütze

5.4.3.1 Rechteckstütze

Die Klasse *SH_Stuetze* ist von *GeoObj* und *Bauteil* abgeleitet, sie besitzt somit alle Methoden und Eigenschaften beider Klassen. Für die grafische Information und Geometrie besitzt sie ein Objekt der Klasse *SH_ParallelPrisma* (Abbildung 5.17).

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Stuetze: public Bauteil, public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Ausrichtung zur Koordinatenachsen wird durch zwei Punkte, Anfangs- und Endpunkt bestimmt. Die Breite, die Tiefe, die Höhe und das untere Niveau einer rechteckigen Stütze werden durch entsprechende Attribute der Klasse *SH_Stuetze* erfaßt (siehe Tabelle der Strukturdaten unten).

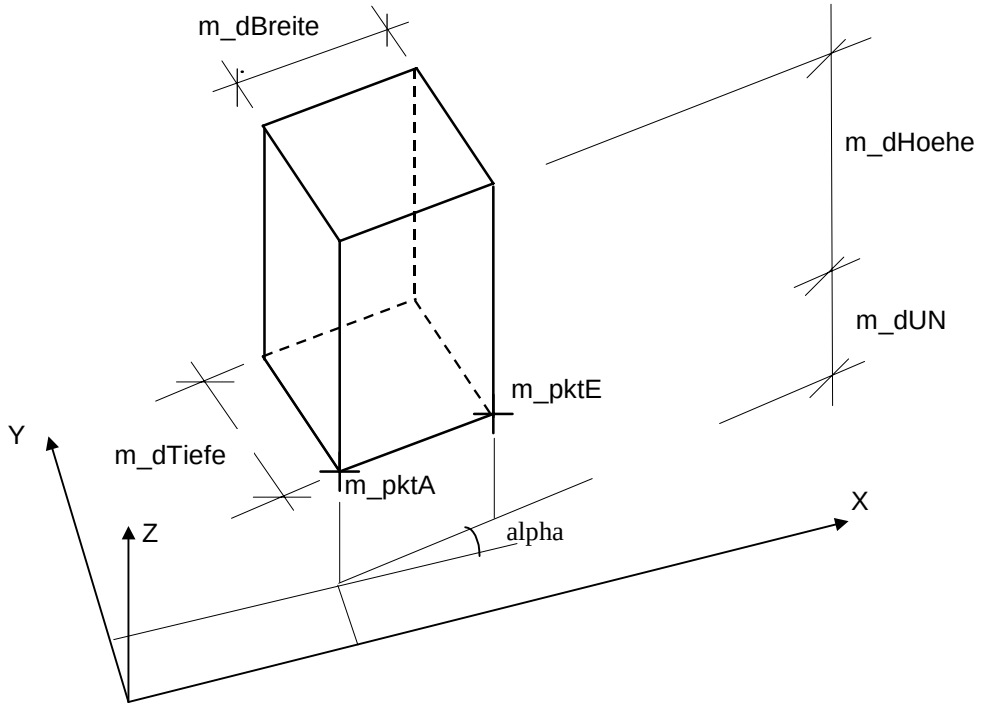


Abbildung 5.17: Rechteckstütze

Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Datenmember	Erklärung	Bemerkungen	Zugriffsrechte
m_pktA	Position der Stütze.	*Punkt*	protected
m_pktE	Gibt die Richtung der Stütze an. (Winkel zur X-Achse).	*Punkt*	protected
Wd3d	Graphischer Repräsentant (Darstellungsobjekt) der Stütze	*SH_ParallelPrisma*	protected
m_dHoehe	Höhe der Stütze	double	protected
m_dUN	Unteres Niveau	double	protected
m_dBreite	Breite der Stütze	double	protected
m_dTiefe	Tiefe der Stütze	double	protected

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
```

```
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Parameter**

```
// Mit diesen Methoden werden alle Parameter gesetzt.
    void SetParameter(const Punkt& anfP, const Punkt& endP,const double T,
                      const  double B, const double H, const double un);

    void SetParameter(const Punkt& anfP,const double alpha,const double T,
                      const double B, const double H, const double un0);
    ...
// Zugriff auf  die einzelnen Attribute
    void SetBreite(double newB);
    void SetHoehe(double newH);
    void SetUnterNiveau(double newUN);
    ...
    double GetBreite() const ;
    double GetHoehe() const;
    double GetUnterNiveau() const ;
    Punkt& GetPointA();
    Punkt& GetPointE()  {return PktE;};
    void GetPunkte(Punkt& PA, Punkt& PE) const;
    ...
// Zugriff auf die Punkte des Darstellungsobjekts (12 Linien mit 24 Punkte)
    void GetPoints(Punkt* points[24]);
    void GetTiefe(const Punkt& anfP,  Punkt& endP, const double un);
                Endpunkt setzen auf
                                                Anfangspunkt + Tiefe
    double GetTiefe() const ;
    void SetTiefe(double newT);
    ...
// Winkel zur x-Achse berechnen und abfragen
    double GetXAngle(int flag) const;
    ...
    ...
```

5.4.3.2 Rundstütze

Dies ist eine Stütze mit einem kreisförmigen Querschnitt (Abbildung 5.18). Die Klasse ist von *GeoObj* und *Bauteil* abgeleitet. Sie besitzt somit alle Methoden und Eigenschaften beider Klassen.

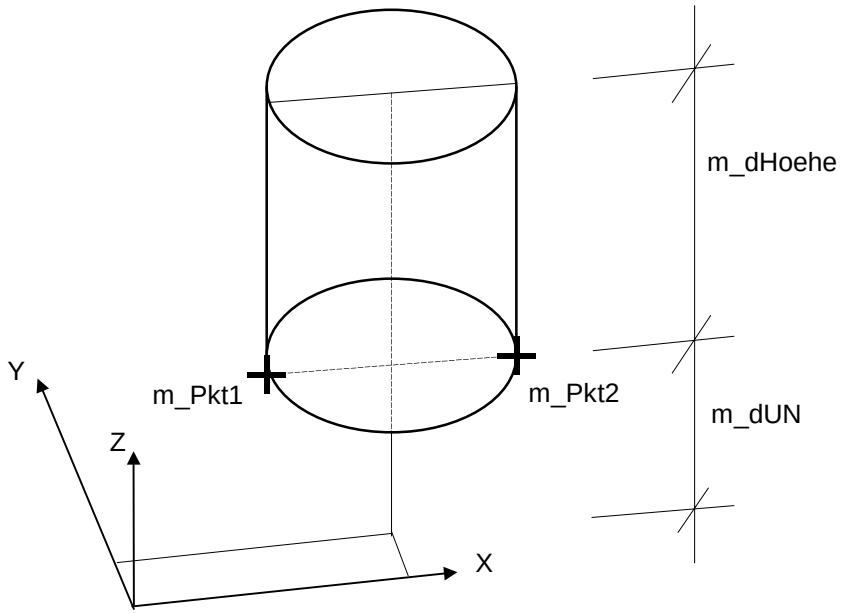


Abbildung 5.18: Rundstütze

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Rundstuetze : public Bauteil, public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Auch diese Klasse besitzt ein geometrisches Objekt als Darstellungsobjekt. Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Punkt m_Pkt1	Position der Stütze.	*Punkt*	public
Punkt m_Pkt2	Gibt die Richtung und Durchmesser der Stütze an.	*Punkt*	public
m_dHoehe	Höhe der Stütze	double	public
double m_dUN	Unteres Niveau	double	public

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Punkte**

// Pkt1 setzen/verändern

void SetP1(Punkt& newP);

// Pkt2 setzen/verändern

void SetP2(Punkt& newP);

// Beide Punkte setzen/ändern

void SetPunkte(const Punkt& newP1, const Punkt& newP2);

// Punkt Pkt2 über das Eingeben des Durchmessers verändern

void SetDurchmesser(double newL);

// Zugreifen auf die Punkte Pkt1 und Pkt2

Punkt& GetP1() {return Pkt1;};
Punkt& GetP2() {return Pkt2;};

void GetPunkte(Punkt& p1, Punkt& p2) const;
void GetPunkte(Punkt pP1, Punkt* pP2);*

// Durchmesser abfragen
double GetDurchmesser() const;
...

5.4.4 Unterzug

5.4.4.1 Einzelner Unterzug

Dieser ist ein einzelner Unterzug (Abbildung 5.19), der mit Anfangs- und Endpunkt zu bestimmen ist.

Die Klasse *SH_Unterzug* ist von der Klasse *GeoObj* und *Bauteil* abgeleitet, sie besitzt somit alle Methoden und Eigenschaften beider Klassen.

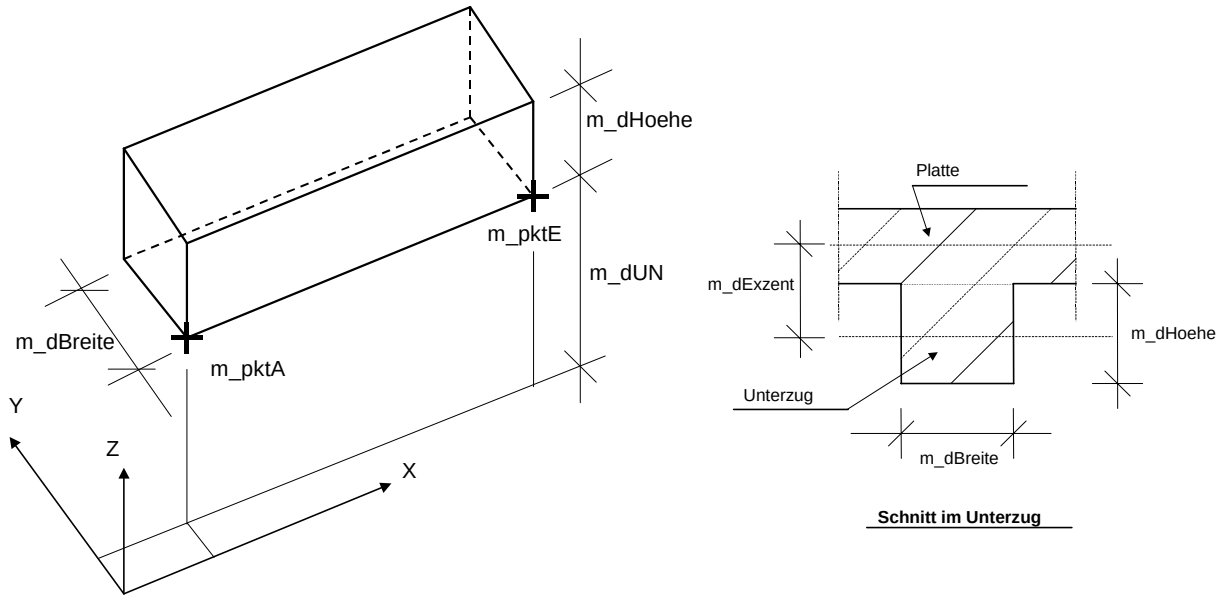


Abbildung 5.19: Unterzug

Ein Objekt von dieser Klasse besitzt ein geometrisches *SH_ParallelPrisma*-Objekt als Darstellungsobjekt.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Unterzug: public Bauteil, public GeoObj
{
public:
...
...
protected:
...
...
}
```

Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_pktA	Position des Unterzugs.	*Punkt*	protected
m_pktE	Gibt die Richtung der Stütze an.	*Punkt*	protected
Wd3d	Graphischer Repräsentant des Unterzugs .	*SH_ParallelPrisma*	protected
m_dHoehe	Höhe des Querschnitts.	double	protected
m_dBreite	Breite des Querschnitts.	double	protected
m_dUN	Unteres Niveau.	double	protected
m_dTorsion	Torsionsfaktor des Unterzuges.	double	protected
m_dExzent	Exzentrizität des Querschnitts.	double	protected

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);

virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
```

```
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Parameter**

// Mit diesen Methoden werden alle Parameter gesetzt.

```
void SetParameter(const Punkt& anfP, const Punkt& endP, const double B,
        const double H, const double un);
void SetParameter(const Punkt& anfP,const Punkt& anfPS, const Punkt& endP,
        const Punkt& endPS ,const double B,const double H, const double un);
void SetLaenge(double newL);
void SetBreite(double newB);
void SetHoehe(double newH);
void SetUnterNiveau(double newUN);
...
double GetLaenge() const ;
double GetBreite() const ;
double GetHoehe() const;
double GetUnterNiveau() const ;
Punkt& GetPointA();
Punkt& GetPointE()  {return PktE;};
void GetPunkte(Punkt& PA, Punkt& PE) const;
...
```

// Alle Start und Endpunkte der 12 Linien des Darstellungsobjektes holen

```
void GetPoints(Punkt* points[24]);
...
void SetTorsion(double torsFakt);
void SetExzent(double exzent);
double GetTorsion() const ;
double GetExzent() const ;
...
...
```

5.4.4.2 Unterzug als Polygonzug

Dieses ist ein Unterzug-Polygon, das durch verschiedene Punkte als Polygonzug zu bestimmen ist (Abbildung 5.20).

Die Klasse *SH_UnterzugPoly* ist von der Klasse *GeoObj* und *Bauteil* abgeleitet, sie besitzt somit alle Methoden und Eigenschaften beider Klassen. Sie besitzt eine dynamische doppelt verkettete Liste, die die Objekte von Unterzügen verwaltet.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_UnterzugPoly : public Bauteil, public GeoObj
{
public:
...
...
protected:
...
...
};
```

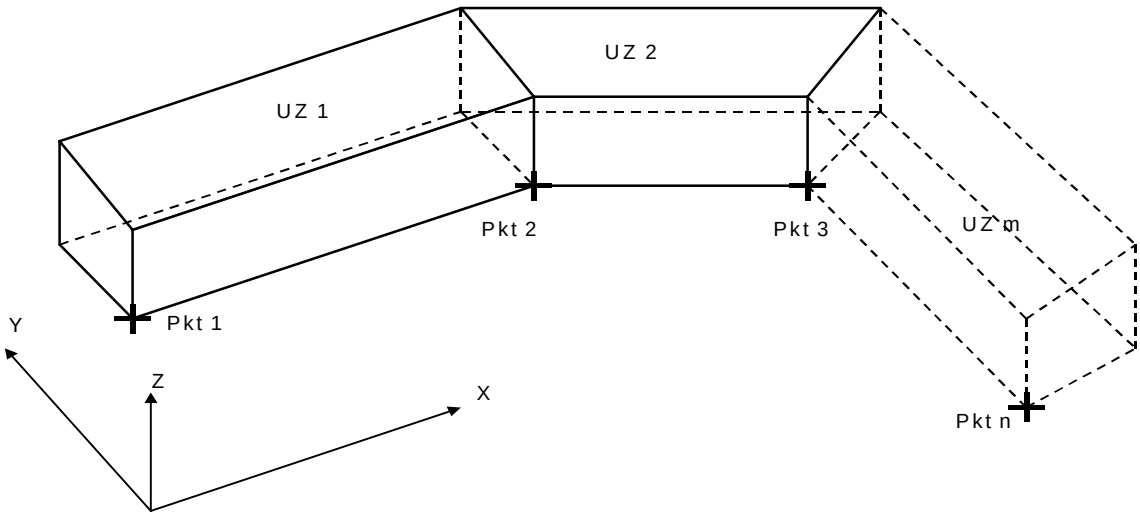


Abbildung 5.20: Polygon von Unterzügen

Ein Objekt von dieser Klasse besitzt eine dynamische Liste von Unterzügen und eine Liste von geometrischen Punkten. Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_arrayPoints	Dynamische Liste von Punkten.	SH_PUNKTARRAY	public
m_arrayWand	Dynamische Liste von Unterzügen.	SH_UNTERZUG_ ARRAY	public
m_bState	Gibt an, ob ein Polygon geschlossen ist oder nicht. TRUE = geschlossen, FALSE = geöffnet	BOOL	protected

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
...
```

- **Methoden zum Verwalten der Punktliste**

Eine dynamische Liste von Punkten wird zuerst definiert:

typedef SH_List< Punkt> SH_PUNKTARRAY;

Die neue Liste besitzt somit alle Methoden der template-Liste *SH_List*, die als Schablone für alle dynamischen Listen eingesetzt werden kann.

// Alle Punkte löschen
void DeleteContents();

// Punkt am Ende entfernen
int DeleteLastPunkt();

// Alle Punkte und Unterzüge löschen
void DeleteContents();

// Anzahl der Punkte abfragen
int NumPunkte() const;

// Ersten Punkt ermitteln
BOOL GetFirstPunkt(Punkt& rPunkt) const;

// Letzten Punkt ermitteln
BOOL GetLastPunkt(Punkt& rPunkt) const;

// Alle Punkte ermitteln (z.B. zum Verändern)
*long GetPoints(double (*xy)[2]);*

// Punkt am Ende einfügen (Besitzer des Punkts ist jetzt das Polygon)
int AppendPunkt(Punkt& rPunkt);
int AppendPunkt(Punkt rPunkt);*
int AppendPunkt(double x, double y, double z);
...

- **Methoden zum Verwalten der Liste der Unterzüge**

Eine dynamische Liste von Unterzügen wird zuerst definiert:

typedef SH_List< SH_Unterzug> SH_UNTERZUG_ARRAY;

Die neue Liste besitzt somit alle Methoden der template-Liste *SH_List*, die als Schablone für alle dynamischen Listen eingesetzt werden kann.

// Ersten Unterzug ermitteln. Referenz auf den ersten Unterzug
BOOL GetFirstUnterzug(SH_Unterzug& rUz) const;

```
// Zeiger auf den ersten Unterzug wird zurückgegeben
        SH_Unterzug* GetFirstUnterzug();

// Letzten Unterzug ermitteln. Referenz rUz wird auf die Werte des letzten
// Unterzuges gesetzt
        BOOL GetLastUnterzug( SH_Unterzug& rUz ) const;

// Zeiger auf den letzten Unterzug wird zurückgegeben
        SH_Unterzug* GetLastUnterzug();

// Unterzug am Ende einfügen. Es wird ein Zeiger mit den Eigenschaften von rUz
// erzeugt und in die Liste eingehängt.
        int AppendUnterzug( SH_Unterzug& rUz);

// Zeiger wird direkt in die Liste eingehängt.
        int AppendUnterzug( SH_Unterzug* pUz );

// Unterzug am Ende entfernen
        int DeleteLastUnterzug();
        ...
```

- **Methoden zum Verändern der geometrischen Eigenschaften**

Zum Beispiel wird durch die Methode *SetState( )* bestimmt, ob der Polygonzug geschloßen oder geöffnet ist.

```
// Polygon-Zustand (m_bState) setzen:
//      geschlossen = TRUE
//      geöffnet    = FALSE
//
        BOOL GetState() const;
        void SetState(BOOL state);
        ...
        ...
```

5.4.5 Linienlager

Für den Tragwerksplaner ist eine tragende Wand, die eine Decke stützt, eine Linienlagerung mit einer gewissen Steifigkeit, die durch die Abmessungen und das Material der Wand bestimmt werden kann.

Ein Linienlager wird im Datenmodell durch eine Wand beschrieben (Abbildung 5.21). Die Lagerungsart und die Steifigkeit sind in einem Objekt der Klasse SH_Freiheitsgrade gehalten. Diese können dann interaktiv in einer Dialogbox für jedes Linienlagerobjekt gesetzt und editiert werden. Die Lagersteifigkeit wird durch die

Steifigkeit der Wand ermittelt, die durch seine Länge, Breite, Höhe und Elastizitätsmodul zu bestimmen ist.

5.4.5.1 Linienlager als Einzelobjekt

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Linienlager: public Bauteil, public GeoObj
{
public:
        ...
protected:
        ...
};
```

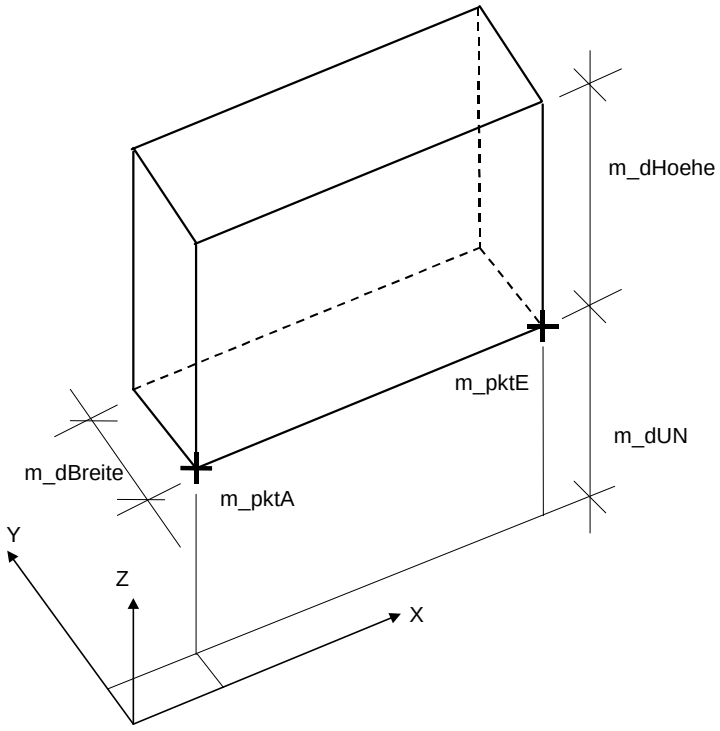


Abbildung 5.21: Linienlager als Wand

Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
PktA	siehe Abbildung	*Punkt*	protected
PktE	siehe Abbildung	*Punkt*	protected
Wd3d	Graphischer Repräsentant der Linienlager	*SH_ParallelPrisma*	protected
Hoehe	Höhe	double	protected
UN	Unteres Niveau (Z- Abstand zum Nullpunkt)	double	protected
Breite	Breite	double	protected
m_FGRAD	Randbedingungen,wird direkt als Methode angesprochen	*SH_Freiheitsgrade*	public

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
```

Methoden zum Verändern der Parameter

// Mit diesen Methoden werden alle Parameter gesetzt.

```
void SetParameter(const Punkt& anfP, const Punkt& endP, const double B,
        const double H, const double un);
```

void SetLaenge(double newL);
void SetBreite(double newB);
void SetHoehe(double newH);
void SetUnterNiveau(double newUN);
...
double GetLaenge() const ;
double GetBreite() const ;
double GetHoehe() const;
double GetUnterNiveau() const ;
Punkt& GetPointA();
Punkt& GetPointE() {return PktE;};
void GetPunkte(Punkt& PA, Punkt& PE) const;
void GetPoints(Punkt points[24]);*
...

- **Freiheitsgrade, die Klasse *SH_Freiheitsgrade***

Die Klasse wird benutzt, um die Freiheitsgrade der Linienlagerung zu setzen. Jeder Freiheitsgrad, Translation in Z-Richtung, Rotation in X-Richtung oder Y-Richtung wird aktiviert mit einer bestimmten Steifigkeit oder nicht aktiviert (Abbildung 5.22).

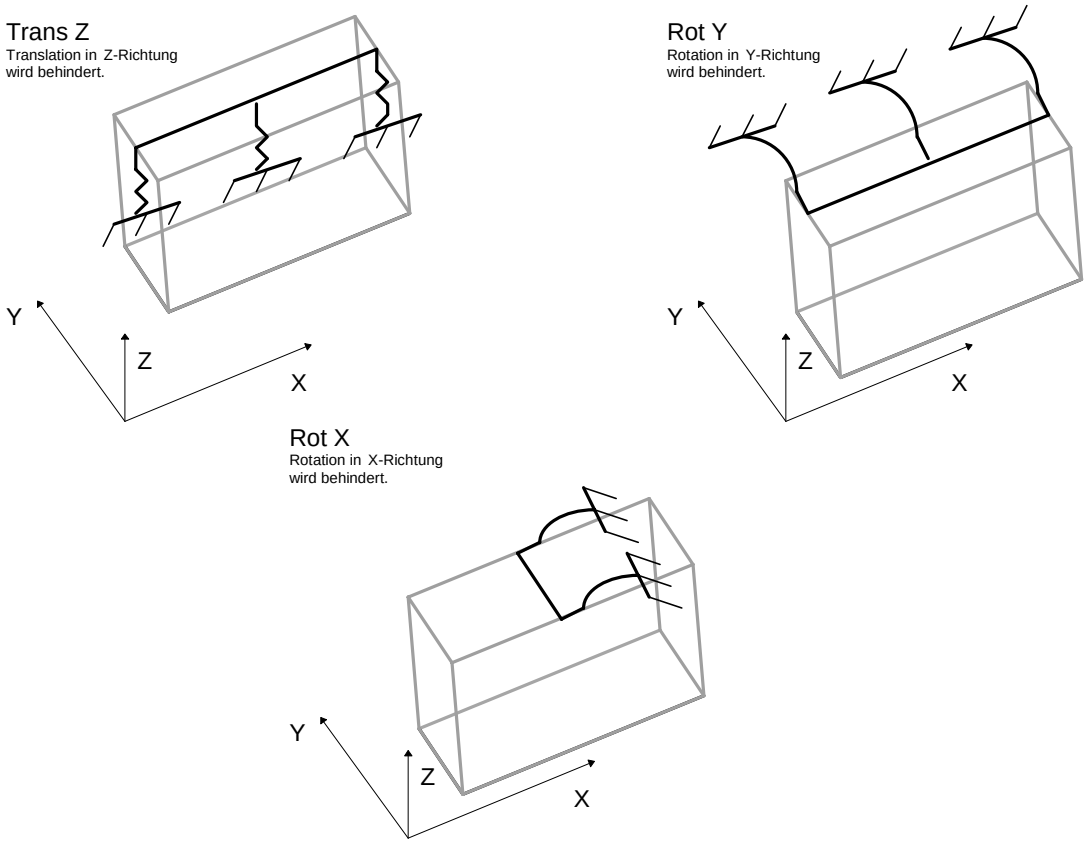


Abbildung 5.22: Freiheitsgrade des Linienlagers

Die Deklaration der Klasse in der Programiersprache C++ erfolgt wie folgt:

```
class SH_Freiheitsgrade
{
public:
        struct FrGrad
        {
           BOOL bIsActive;
           double Steif;
        };

        FrGrad TransZ;
        FrGrad RotY;
        FrGrad RotX;

        // Konstruktor und Destruktor
        SH_Freiheitsgrade();
        ~SH_Freiheitsgrade(){};
};
```

Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Steif	Wert der Steifigkeit siehe Abbildung	double	public
TransZ	Translation in Z-Richtung	struct FrGrad	public
RotY	Rotation in Y-Richtung	struct FrGrad	public
RotX	Rotation in X-Richtung	struct FrGrad	public

Man kann durch eine Mehrfachauswahl die Freiheitsgrade und die zugehörigen Steifigkeiten in einer interaktiven Dialogbox angeben, für die eine Linienlagerung als Randbedingung dienen soll.

- **Methoden**

Da die verschiedenen Variablen der Freiheitsgrade als public Member-Daten innerhalb der Klasse deklariert sind, kann auf die eigenen Daten direkt zugegriffen werden.

// Ist die Randbedingung gesetzt (TRUE/FALSE)

Freiheitsgrad . bIsActive = TRUE → *Freiheitsgrad ist gesetzt*

Freiheitsgrad . bIsActive = FALSE → *Freiheitsgrad ist nicht gesetzt*

5.4.5.2 Linienlager als Polygonzug

Dies ist ein Polygon von Linienlagern, das durch verschiedene Punkte als Polygonzug zu bestimmen ist (Abbildung 5.23).

Die Klasse *SH_WandPoly* ist von der Klasse *GeoObj* und *Bauteil* abgeleitet, sie besitzt somit alle Methoden und Eigenschaften beider Klassen. Sie besitzt eine dynamische doppelt verkettete Liste, die die Objekte von Linienlagern verwaltet.

Es erfolgt keine Verschneidung der Linienlager, da dies für den Tragwerksplaner nicht von Interesse ist.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_WandPoly : public Bauteil, public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

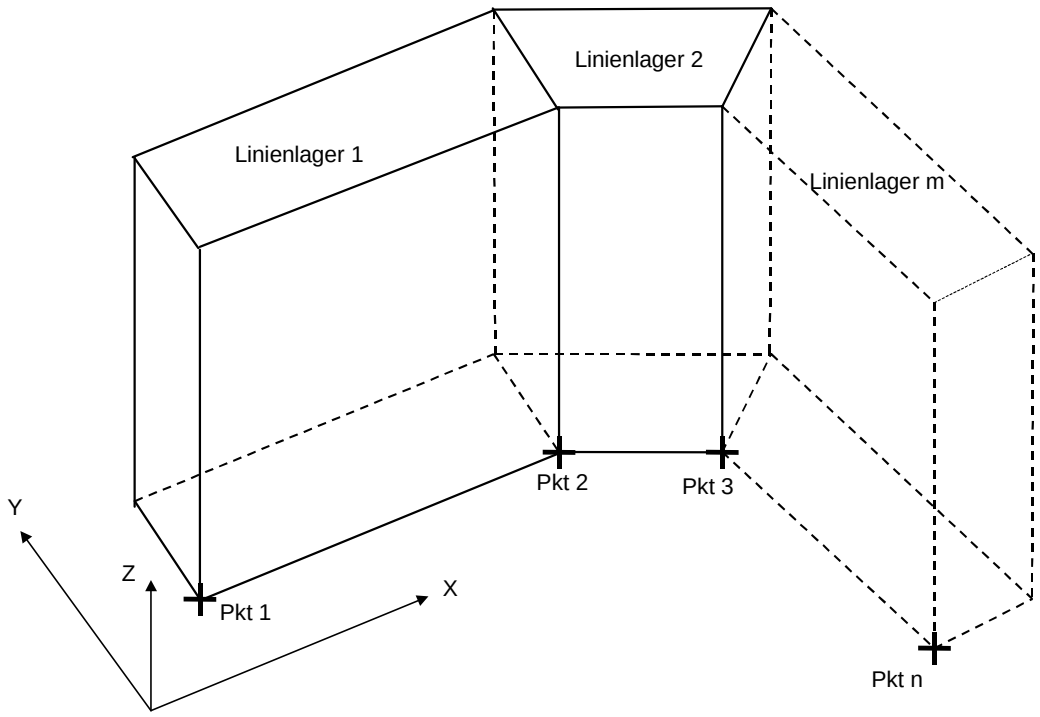


Abbildung 5.23: Linienlager als Polygonzug

Ein Objekt von dieser Klasse besitzt eine dynamische Liste von Linienlagern und eine Liste von geometrischen Punkten. Die Strukturdaten bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_arrayPoints	Dynamische Liste von Punkten.	SH_PUNKTARRAY	public
m_arrayWand	Dynamische Liste von Linienlagern	SH_WAND_ARRAY	public
m_bState	Gibt an ob ein Polygon geschlossen ist oder nicht: TRUE = geschlossen FALSE = geöffnet	BOOL	protected

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
....
```

- **Methoden zum Verwalten der Punktliste**

Eine dynamische Liste von Punkten wird gebraucht. Sie wird wie gefolgt definiert:

```
typedef SH_List< Punkt> SH_PUNKTARRAY;
```

Die neue Liste besitzt somit alle Methoden der template-Liste *SH_List*, die als Schablone für alle dynamischen Listen eingesetzt werden kann.

// Alle Punkte löschen
void DeleteContents();

// Punkt am Ende entfernen
int DeleteLastPunkt();

// Alle Punkte und Unterzüge löschen
void DeleteContents();

// Anzahl der Punkte abfragen
int NumPunkte() const;

// Ersten Punkt ermitteln
BOOL GetFirstPunkt(Punkt& rPunkt) const;

// Letzten Punkt ermitteln
BOOL GetLastPunkt(Punkt& rPunkt) const;

// Alle Punkte ermitteln (z.B. zum Verändern)
*long GetPoints(double (*xy)[2]);*

// Punkt am Ende einfügen (Besitzer des Punkts ist jetzt das Polygon)
int AppendPunkt(Punkt& rPunkt);
int AppendPunkt(Punkt rPunkt);*
int AppendPunkt(double x, double y, double z);
...

- **Methoden zum Verwalten der Liste der Linienlager**

Eine dynamische Liste von Unterzügen wird zuerst definiert:

typedef SH_List< SH_Wand> SH_WAND_ARRAY;

Die neue Liste besitzt somit alle Methoden der template-Liste *SH_List*, die als Schablone für alle dynamischen Listen eingesetzt werden kann.

// Erstes Linienlager ermitteln. Referenz rUz wird auf die Werte des ersten
// Linienlagers gesetzt
BOOL GetFirstUnterzug(SH_Unterzug& rUz) const;

// Zeiger auf den ersten Unterzug wird zurückgegeben
SH_Unterzug GetFirstUnterzug();*

// Letzten Unterzug ermitteln. Referenz rUz wird auf die Werte des letzten
// Unterzuges gesetzt
BOOL GetLastUnterzug(SH_Unterzug& rUz) const;

// Zeiger auf den letzten Unterzug wird zurückgegeben
SH_Unterzug GetLastUnterzug();*

// Unterzug am Ende einfügen. Es wird ein Zeiger mit den Eigenschaften von rUz
// erzeugt und in die Liste eingehängt.
int AppendUnterzug(SH_Unterzug& rUz);

// Zeiger wird direkt in die Liste eingehängt.
int AppendUnterzug(SH_Unterzug pUz);*

// Unterzug am Ende entfernen
int DeleteLastUnterzug();
...

- **Methoden zum Verändern der geometrischen Eigenschaften**

// Polygon-Zustand (m_bState) setzen:
// geschlossen = TRUE
// geöffnet = FALSE
//
BOOL GetState() const;
void SetState(BOOL state);
....

5.4.6 Bettung

Mit einer Bettung wird die Flächenauflagerung von elastisch gelagerten Platten beschrieben, in denen der Spannungszustand auch von der Steifigkeit des Bodens abhängig ist. Mit der Querkontraktionszahl Mue und dem Steifemodul des Bodens wird die Federung der Bettung durch eine Berechnung nach dem Steifezifferverfahren bestimmt. Die Umgrenzung der Bettung wird durch einen Polygonzug beschrieben (Abbildung 5.24).

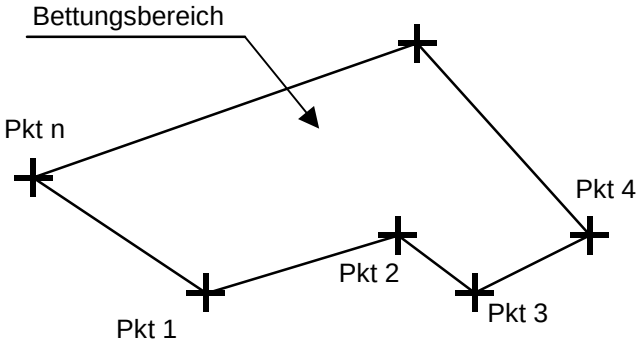


Abbildung 5.24: Die Geometrie eines Bettungsbereichs wird durch ein Polygon beschrieben

Die Klasse *SH_Bettung* ist von *SH_Polygon* und *Bauteil* abgeleitet und besitzt somit alle Eigenschaften und Methoden beider Klassen.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Bettung : public Bauteil, public SH_Polygon
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
TransZ	Bettungsdaten für Bettungszifferverfahren.	Struktur *Betdat*	public
m_dMue	Querkontraktionszahl Mue des Bodens (Wenn Steifezifferverfahren ausgewählt wurde)	double	public

Die Bettungsdaten werden beim Bettungszifferverfahren in einer Struktur vom Typ *Betdat* gehalten:

```
struct Betdat
        {
        BOOL bIsActive;     // TRUE → Bettungszifferverfahren ist ausgewählt.
                            // FALSE → Steifezifferverfahren ist ausgewählt

        double Steif;       // Steifigkeit der Bettung
        double Schwell;     // Schwellwert gibt an ab welchem Betrag bei MicroFe
                            // Pressungen ausgegeben werden.
         };
```

- **Methoden der Basisklasse**

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
```

```
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
```

- **Methoden zum Verwalten der Punktliste**

Diese werden von der Klasse *SH_Polygon* geerbt, sie werden nicht in der Klasse noch mal implementiert und sind direkt aufrufbar.

// Alle Punkte aus *SH_Polygon* löschen
void DeleteContents();

// Anzahl der Punkte abfragen
int NumPunkte() const;

// Ersten Punkt ermitteln
BOOL GetFirstPunkt(Punkt& rPunkt) const;

// Letzten Punkt ermitteln
BOOL GetLastPunkt(Punkt& rPunkt) const;

// Polygonpunkte ermitteln (z.B. zum Verändern)
*long GetPoints(double (*xy)[2]);*

// Punkt am Ende einfügen (Besitzer des Punkts ist jetzt das Polygon)
int AppendPunkt(Punkt& rPunkt);

// Punkt am Ende entfernen
int DeleteLastPunkt();

Es sind auch Methoden zum Setzen der Default-Werte wie *DefaultDat()* vorhanden.

5.5 Belastungsobjekte

5.5.1 Allgemeines

Die Belastungsdaten werden in Lastfällen organisiert. Ein Lastfall besitzt eine dynamische Liste von Lastobjekten, die ihm zugeordnet wurden. Die Lastfälle wiederum werden nicht in die Verwaltungsliste der Tragwerksobjekte eingetragen, sondern werden über eine gesonderte dynamische Liste verwaltet. Die Lastobjekte und ihre Darstellungsobjekte werden in den zugehörigen *SH_ViewObjManagern* und Folien (z.B. *SH_Ebene*) eingefügt. So kann man die Darstellung der Belastungsdaten über die Lastfälle steuern.

5.5.2 Lastfälle

Ein Lastfall besteht aus mehreren Lasttypen (Lastfigur). Ein Objekt der Klasse SH_Lastfall besitzt eine dynamische Liste von Zeigern auf Lasten, die zu diesem Lastfall zugeordnet sind. Außerdem kann jedem Lastfall eine Nummer und einen Name vergeben.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Lastfall
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_arrayLast	Dynamische Liste der Lasten des Lastfalls	*BASELIST*	public
M_szLastName	Lastfallbezeichnung	char*	protected
m_nLastfallNr	Lastfall-Nummer	int	protected

- **Methoden zum Setzen/Verändern der Attribute**

// Lastfallbezeichnung abfragen
char GetLastName() const {return m_szLastName;};*

// Lastfallbezeichnung setzen
void SetLastName(char szBtName);*

// Lastfallnummer abfragen
int GetLastfallNr() ;

// Lastfallnummer setzen
void SetLastfallNr(int Nr);
...

- **Methoden zur Verwaltung der Lastenliste**

// Lastobjekt in Liste einhängen
int AppendLast(GeoObj& rLast);
int AppendLast(GeoObj pLast);*

// Lastobjekt am Ende der Liste entfernen
int DeleteLast();

// Alle Lastobjekte löschen
void DeleteContents();

// Lastfall darstellen
void ShowLastfall();

// Lastfall nicht darstellen
void HideLastfall();
...

5.5.3 Basisklasse für die Belastungsobjekte

Alle Lastklassen werden durch Mehrfachvererbung gebildet. Sie werden von der Basisklasse *SH_Last* und direkt von *GeoObj* oder von einer anderen geometrischen Klasse wie *SH_Polygon* abgeleitet (Abbildung 5.25). Es handelt sich dabei um drei Belastungsarten:

- Punktlast: Einzellast oder Drehmoment
- Linienlast
- Flächenlast

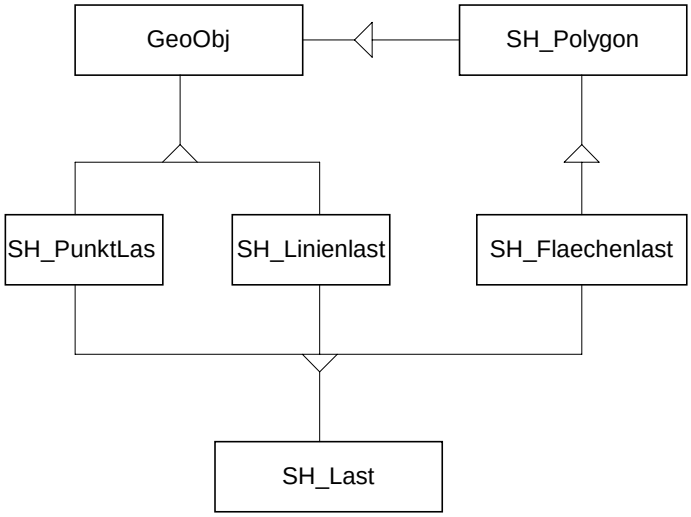


Abbildung 5.25: Klassifikation der Belastungsobjekte

Die Basisklasse für alle Belastungsobjekte *SH_Last* legt fest, welche Attribute und Eigenschaften alle Belastungsobjekte besitzen, wie z.B. Lastintensität, Lastfallnummer und ob es sich um eine ständig angreifende oder wechselhafte Last handelt.

Die Deklaration der Klasse Basisklasse *SH_Last* in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Last
{
public:
        ...
protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_bStaendig	Gibt an ob es eine ständige oder veränderliche Last ist. TRUE = ständige Last FALSE = veränderliche Last	BOOL	public
m_dIntensity	Intensität der Last	double	protected
m_szLastName	Lastfallbezeichnung	char*	protected
m_nLastfallNr	Lastfall - Nummer	int	protected

- **Methoden**

// Lastfallbezeichnung abfragen
char GetLastName() const {return m_szLastName;};*

//Lastfallbezeichnung setzen
void SetLastName(char szLstName);*

// Lastfallnummer abfragen
int GetLastfallNr();

// Lastfallnummer setzen
void SetLastfallNr(int Nr);

// Abfragen der Intensität
double GetIntensity() const;

// Setzen der Intensität
void SetIntensity(double p);
...

5.5.4 Punktlast

Eine Punktbelastung wird durch ihre Position und Intensität angegeben. Im Datenmodell werden nur die Einzellasten und Drehmomente modelliert, die parallel zu den und um die globalen Koordinatenachsen x, y und z angreifen (Abbildung 5.26). Die Lastgröße wird nach ihrem Intensitätsbetrag ausgewertet, die Wirkungsrichtung der Last wird durch das Vorzeichen festgelegt. Dabei kann es sich um eine Einzellast, die in jede Richtung x, y oder z angreift, oder um ein Drehmoment, das sich um die x-, y- oder z-Achse dreht, handeln.

Die Richtung der Last ist in der positiven Richtung der Koordinatenachse, wenn die Last positive Intensität hat und umgekehrt, die Richtung der Last ist in der negativen Richtung der Koordinatenachse, wenn die Last negative Intensität hat.

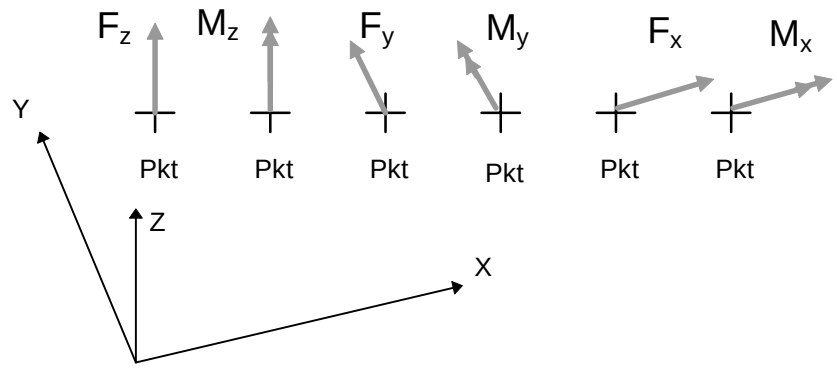


Abbildung 5.26: Lasttypen

Die Klasse *SH_PunktLast* ist von *SH_Last* und *GeoObj* abgeleitet und besitzt somit alle Eigenschaften und Methoden beider Klassen. Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Punktlast : public SH_Last, public GeoObj
{
public:
        ...
        ...
protected:
        ...
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Punkt Pkt	Position der Last		public
m_Lasttyp	Art der Belastung, siehe Abbildung 5.26	*LastTyp*	public

- **Deklaration der Lasttypen**

Lasttypen beinhalten Eizellasten und Momente in Richtung der globalen Koordinatenachsen x, y und z. Ein neuer Datentyp für Aufzählung der Lasttypen kann in C++ wie folgendt definiert werden:

```
enum LastTyp  {     Fz,     // Einzellast in Z-Richtung
                    Fy,     // Einzellast in Y-Richtung
                    Fx,     // Einzellast in X-Richtung
                    Mx,     // Einzelmoment in X-Richtung
                    My,     // Einzelmoment in Y-Richtung
                    Mz,     // Einzelmoment in Z-Richtung
              };
```

- **Methoden der Basisklasse** *GeoObj*

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
```

```
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
```

- **Methoden**

// Lasttyp setzen

void SetLastTyp(LastTyp typ);

// Alle Parameter setzen

void SetLast(Punkt newPkt,LastTyp typ,double intens);

// Position der Last setzen oder abfragen

void SetPunkt(Punkt& newP);
Punkt GetPunkt() const;

5.5.5 Linienlast

Eine Linienlast wird durch zwei Punkte beschrieben (Abbildung 5.27). Die Intensität der Last kann an jedem Punkt angegeben werden, so daß eine trapezförmige Linienlast gebildet werden kann.

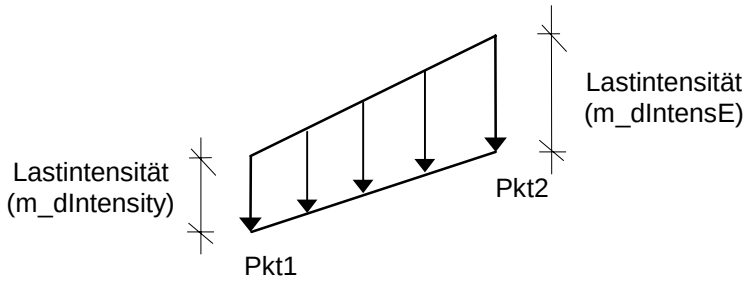


Abbildung 5.27: Linienlast, zwei Punkte und die Lastintensität an jedem Punkt

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Linienlast : public SH_Last, public GeoObj
{
```

public:
...
protected:
...
};

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
Pkt1	Anfangspunkt	*Punkt*	public
Pkt2	Endpunkt	*Punkt*	public
m_dIntensE	Intensität am Endpunkt. Intensität des Anfangspunkt kommt von der Basisklasse	double	public
m_Lasttyp	siehe *SH_Punktlast*	*LastTyp*	public

- **Methoden der Basisklasse** *GeoObj*

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
```

```
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
```

- **Methoden zum Verändern der Punkte**

```
// Punkte setzen/verändern
    void SetP1(Punkt& newP);
    void SetP2(Punkt& newP);
    void SetPunkte(const Punkt& newP1, const Punkt& newP2);

// Punkt Pkt2 über das Eingeben der Länge verändern
    void SetLaenge(double newL);

// Punkte abfragen
    Punkt& GetP1() {return Pkt1;};
    Punkt& GetP2() {return Pkt2;};
    void GetPunkte(Punkt& p1, Punkt& p2) const;
    void GetPunkte(Punkt* pP1, Punkt* pP2);


// Länge berechnen und abfragen
    double GetLaenge() const;

// Winkel zur x-Achse berechnen und abfragen
    double GetXAngle(int flag) const;
    ...
```

- **Methoden zum Verändern der Lastattribute**

Intensität des Anfangspunktes wird durch die Methode der Basisklasse beschrieben.

```
// Intensität des Endpunktes setzen/abfragen
    void SetIntensE(double intens);
```

```
        double GetIntensE();

// Lasttyp setzen
        void SetLastTyp(LastTyp typ);
        ...
```

5.5.6 Flächenlast

Eine Flächenlast wird durch ein Polygon beschrieben (Abbildung 5.28). Die Intensität der Last kann an jedem Punkt angegeben werden.

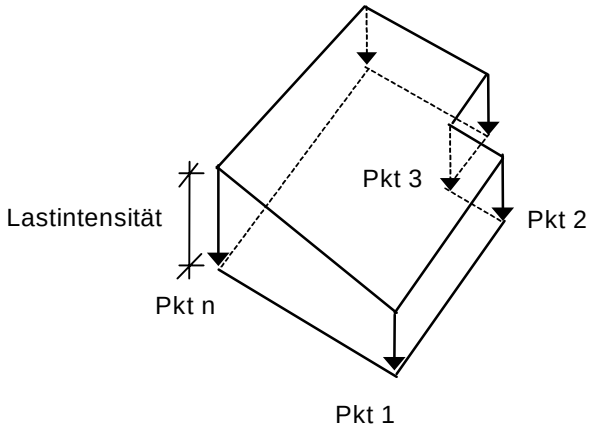


Abbildung 5.28: Flächenlast, Polygonzug von Punkte und Lastintensität an jedem Punkt

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_Flaechenlast : public SH_Last, public SH_Polygon
{
public:
        ...
protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_TopPolygon	Oberes Polygon	*SH_Polygon*	protected
m_arrayIntens	Liste von Lastintensitäten, die zu den Polygonpunkten zugeordnet sind	*SH_INTENSARRAY*	public

- **Methoden der Basisklasse** *GeoObj*

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
...
virtual void DrawInsert(SH_ViewObjManager* pObm,SH_ViewMap*
                        pViewMap );
virtual void DrawRemove( SH_ViewObjManager* );
...
virtual void ExecDialog();
...
```

- **Methoden zum Verwalten der Intensitätsliste**

// Erste Intensität ermitteln

BOOL GetFirstIntens(double& intens) const;

// Letzte Intensität ermitteln

BOOL GetLastIntens(double& intens) ;

// Intensität am Ende der Liste hinzufügen

int AppendIntens(double& intens);

// Intensität am Ende entfernen

int DeleteLastIntens();

// Alle Intensitäten löschen

void DeleteAllIntens();

...

5.6 Bewehrungsobjekte (Bewehrunksbereiche)

Ein Bewehrungsbereich (Bewehrungsfeld) wird mit einem Polygonzug von Punkten bestimmt. Es kann sich über mehrere Plattenbereiche strecken, und kann oben oder unten bezüglich der Lage im Querschnitt des Plattenbereiches stehen (Abbildung 5.29).

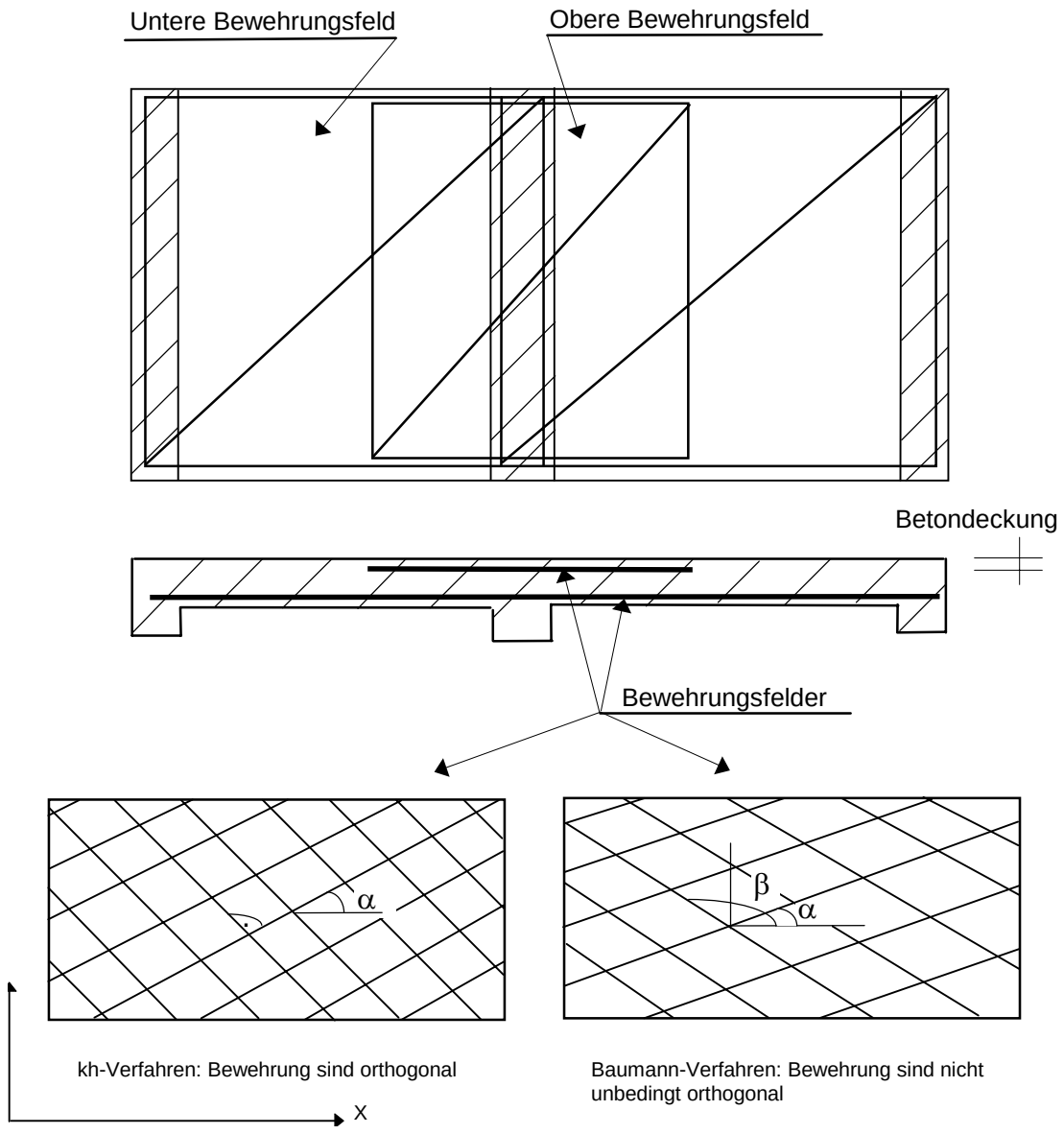


Abbildung 5.29: Bewehrungsfelder in einem Plattenbereich

Eine Klasse *SH_ Bewehrungsbereich* hat mehrere Attribute, wie die Betondeckung, die der Abstand von der Betonoberfläche zur Schwerachse der Bewehrung darstellt. Außerdem ist der Winkel der Bewehrungsrichtung bezüglich der positiven x-Achse

für die Bewehrung in x- und y-Richtung je nach Bemessungsverfahren so zu definieren:

kh-Verfahren: Die Bewehrung in y-Richtung ist orthogonal zu der Bewehrung in x-Richtung im Bewehrungsbereich, damit ist der Winkel für die Bewehrung in y-Richtung durch den Winkel für die Bewehrung in x-Richtung zu bestimmen.

Baumann-Verfahren: Die Bewehrung in y-Richtung muß nicht orthogonal zu der Bewehrung in x-Richtung im Bewehrungsbereich, damit sind die zwei Winkel, der für die Bewehrung in x-Richtung, und der für die Bewehrung in y-Richtung, unterschiedlich.

Die Klasse Bewehrungsbereich *SH_ Bewehrungsbereich* ist von der Klasse *SH_Polygon* und *Bauteil* abgeleitet und besitzt somit alle Eigenschaften und Methoden beider Klassen. Ein Bewehrungsbereich kann somit durch die Eingabe eines Polygonzuges gesetzt werden und alle Funktionen einer Polygonklasse können direkt aufgerufen werden.

Die Materialeigenschaften *Material* und Bemessungsverfahren *SH_Bemessung* des Bewehrungsbereiches werden von der Basisklasse *Bauteil* geerbt.

Jedem Plattenbereich ist eine dynamische Liste von Bewehrungsbereichen zugeordnet. Beim Setzen eines Bewehrungsbereichs muß mindestens eine Platte identifiziert werden, zu der dieser Bewehrungsbereich zugeordnet wird. Dazu gehören Kontrollfunktionen zum Testen, ob sich der Bewehrungsbereich beim Setzen in mindestens einem Plattenbereich befindet.

Die Deklaration der Klasse in der Programmiersprache C++ erfolgt wie folgt:

```
class SH_ Bewehrungsbereich: public Bauteil, public SH_Polygon
{
public:
        ...
protected:
        ...
};
```

Die Strukturdaten der Klasse bestehen aus folgenden Komponenten und Attributen:

Member-Daten	Erklärung	Datentyp	Zugriffsrechte
m_bObenUnten	Lage des Bewehrungsbereichs	BOOL	protected
m_dBetonDeckung	Betondeckung	double	protected
m_RefObjList	Alle Objekte, zu denen der Bewehrungsbereich zugeordnet ist	*SH_Liste*	
m_dWinkelX	Winkel mit der positiven x-Achse der Bewehrung in x-Richtung	double	protected
m_dWickelY	Winkel mit der positiven x-Achse der Bewehrung in y-Richtung	double	protected

Die Eigenschaften eines Objekts (Instanz) dieser Klasse werden durch folgende Methoden (Funktionen) bestimmt:

- **Methoden der Basisklasse**

Die Methoden, die in der Basisklasse *GeoObj* als virtual deklariert sind, werden hier innerhalb der Klasse definiert, wie z.B. die folgenden Funktionen:

```
...
virtual void ObjDraw();
virtual void ObjAdd(Manager* mng);
virtual void ObjRemove(Manager* mng);
virtual void ObjMove(Punkt moveP);
...
virtual BOOL Is_InBox(double x1, double y1, double x2, double y2);
virtual BOOL Is_NearPoint(double x1, double y1, double radius);
virtual BOOL FangPunkt(Punkt& fangPkt, double radius);
virtual BOOL FangLinie(Punkt& fangPkt, double radius);
...
virtual void Select(BOOL selectmode);
...
virtual UINT ReadFrom(ostream& strm);
virtual void WriteOn(ostream& strm);
virtual void ReadPos(istream& strm) ;
virtual void WritePos(ostream& strm,int Typ);
...
virtual void ExecDialog();
...
```

- **Methoden zum Verwalten der Punktliste**

Diese werden von der Klasse *SH_Polygon* geerbt, sie werden nicht nochmal in der Klasse implementiert und sind direkt aufrufbar.

- **Methoden als Schnittstelle zu den eigenen Daten (protected member)**

// Zugreifen auf die Attribute

```
double GetBetondeckung() const;
double GetWinkelX() const ;
double GetWinkelY() const ;
...
void SetBetondeckung(double d);
void SetWinkelX();
void SetWinkelY();
```

// Lage setzen/abfragen: Oben oder unten

```
BOOL IsAbove() const;
```

void SetPosition(BOOL pos);

- **Methoden zur Verwaltung der zugeordneten Referenzobjekte**

4// Platte in die Liste eintragen
*int AppendPlatte(SH_Platte * pPlatte);*

// Neue Platte erzeugen und in die Liste eintragen
int AppendPlatte(SH_Platte& rPlatte);

// Erste Platte ermitteln
BOOL GetFirstPlatte(SH_Platte & rPlatte) const;
*SH_Platte * GetFirstPlatte();*

// Letzte Platte in der Liste ermitteln
BOOL GetLastPlatte(SH_Platte & rPlatte) const;
*SH_Platte * GetLastPlatte();*

// Platte am Ende der Liste entfernen
int DeleteLastPlatte();

// Alle Platten in der Liste löschen
void DeleteAllPlatten();

// Überprüfen, ob ein Punkt innerhalb einer Platte des Bewehrungsbereichs liegt
BOOL IsPunktInPlatte(Punkt rPunkt);
...

6. Realisierung des Modellkonzepts, das CAD-System MvCad

Der nächste Schritt ist die Implementierung und Erzeugung eines lauffähigen Programms, um das Datenmodell, seine Klassen und Methoden, zu testen.

Im Rahmen dieser Arbeit wird ein CAD-System **MvCad** zur Realisierung des im Kapitel *5* Datenmodells entwickelt.

6.1 Anforderung an das System

Aufgabe des entwickelten CAD-Systems ist die Zusammenstellung der verschiedenen Daten wie Strukturdaten des Tragwerksmodells bzw. grafische Daten für die Darstellung, ihre Organisation sowie die Ermöglichung verschiedener Sichten auf die Strukturdaten der Bauteile. Weiterhin das Modifizieren oder Erweitern des Informationsgehaltes durch die CAD-Funktionalitäten und die intuitive interaktive Benutzeroberfläche.

Das entwickelte CAD-System setzt sich aus verschiedenen Abstraktionsebenen zusammen, die durch wohldefinierte Schnittstellen repräsentiert sind. Es soll folgende wichtige Anforderungen erfüllen:

- Haltung und Verwaltung der Objekte im Tragwerksmodell
- Haltung und Verwaltung der grafischen Informationen
- Eine objektorientierte grafisch-interaktive Benutzeroberfläche (GUI)
- Verschiedene Sichten auf die Daten (Multiple Views)
- Ein Objekt kann auf unterschiedliche Arten konstruiert und dargestellt werden. Das heißt, Trennung zwischen dem Konstruktionsprozeß der Objekte mit ihrer Präsentation währenddessen, und der endgültigen Darstellung dieser Objekte.
- Eine Schnittstelle zum Datenaustausch durch Konvertierung von Datenstrukturen von und nach Textformat (ASCII-Format).

Gemäß dem Modellkonzept besteht ein Bedarf, die Objekte des Tragwerksmodells von ihrer Darstellungsart zu trennen und in unterschiedlichen Sichten zu präsentieren. Damit hat man die Möglichkeit, ein Bauobjekt in verschiedenen Sichten (Views) darstellen zu können. Dabei hat jede View ihre grafischen Daten wie Linienfarbe, Rasterdarstellung, Zoomfaktor und -grenze usw., die separat verwaltet werden soll.

Weiterhin ist eine grafisch-interaktive Benutzerschnittstelle in den modernen CAD-Systemen ein Muß geworden, um eine grafische Eingabe der Daten zu ermöglichen und mit dem System interaktiv und innovativ zu arbeiten.

6.2 Systemkonzept

Für die Unterstützung der Implementierung des objektorientierten Systems MvCad ist die Entwicklungsumgebung von Microsoft VC++ eingesetzt, die als integrierte moderne Entwicklungsumgebung für Softwareprodukte unter dem Betriebsystem Windows bezeichnet ist.

Für die unterschiedlichen Darstellungen der Objekte des Tragwerksmodells kommt der Einsatz von der Document-View-Architecture von MFC (Microsoft Foundation Classes Library) zu Nutze, um verschiedene Sichten (Views) in separaten Fenstern mit Zugriff auf den gleichen Datengehalt des Dokuments zu ermöglichen.

Eine grafisch-interaktive Benutzerschnittstelle wird mit Hilfe der MFC realisiert. Dabei stellen die View-Klassen mit ihren Antwortfunktionen auf die vom Benutzer erlösten Ereignisse eine objektorientierte Schnittstelle mit zahlreichen grafischen Elemente zur interaktiven Arbeit mit dem System dar.

Das Designkonzept des Systems **MvCad** ist in der Abbildung 6.1 und 6.1a gezeigt, es besitzt folgende wichtige Merkmale:

- Die Datenhaltung des Tragwerksmodells wird durch dynamische, doppelt verkettete Listen gelöst. Dabei werden die Objekte in die Liste dynamisch ein- und ausgefügt. Damit wird die Speichergröße dieser Listen in der Laufzeit des Programms je nach Bedarf vergrößert und verkleinert, und muß nicht vorher definiert sein. Eine dynamische Liste wird immer für die Mengenverwaltung gebraucht.

- Die Verwaltung der Objekte wird durch sogenannte Container-Klassen (Behälter-Klassen) geregelt, deren Objekte andere Objekte verwalten. Dazu gehören insbesondere Mengenklassen. Die Mengenklassen sind also Klassen, die Mengen von Objekten verwalten, wie z.B. alle Objekte, die zu einer Ebene oder einem Geschoß gehören. Diese Klassen besitzen dynamische Listen und haben Methoden für die Listenverwaltung, sowie die Schnittstellen für den Zugriff auf die Objekte in der Liste (siehe die Tamplate-Klasse *SH_List* in diesem Kapitel).

- Eine objektorientierte grafisch-interaktive Benutzeroberfläche (GUI) wird durch Benutzung der Microsoft Klassenbibliothek MFC (Microsoft Foundation Class Library) realisiert [41]. Dabei stellen die Windows-Klassen und deren abgeleitete View-Klassen die Schnittstellen zur Ein- und Ausgabe auf und von den verschiedenen Geräten wie Tastatur, Maus, Bildschirm, Drucker, Plotter etc. Diese Klassen ermöglichen den Zugriff auf die Geräte unabhängig von Hersteller oder Marken, da das Betriebsystem durch eine Vielzahl von Treiberprogrammen (Englisch: Driver) diese Aufgabe übernimmt und die dazu nötigen Funktionen zur Verfügung stellt, so daß eine Windows-Applikation (ein Programm, das unter dem

Betriebsystem Windows läuft) nicht direkt auf die Geräte zu zugreifen braucht. Jedes Ereignis von den Geräten, sei es eine Mausbewegung oder Tastendruck auf der Tastatur, kann abgefangen und darauf eine entsprechende Reaktion in Form einer Funktion geschrieben werden. Diese Funktionen gehören zu den View-Klassen, so daß jedes View-Objekt alle Nachrichten über die Ereignisse von den Geräten durch das Betriebsystem übermittelt bekommt, und braucht nur die Methode zu schreiben, die bei jenem Ereignis ausgeführt werden soll.

- Die verschiedenen Sichten auf die Daten (Multiple Views) werden durch verschiedene View-Klassen realisiert. Da verschiedenen View-Typen, 2D, 3D und Text, für die unterschiedliche Darstellungen der Objekte im Tragwerksmodell konzipiert sind, werden drei unterschiedliche View-Klassen von den entsprechenden View-Klassen der MFC abgeleitet und im CAD-System implementiert. Dabei kommt die Dokument-Sicht-Architektur (View-Document Architecture) der MFC zum Einsatz, die in diesem Kapitel später näher erläutert wird.

- Die Haltung der grafischen Informationen der verschiedenen Sichten (Views) wird durch Sonderklassen *ViewMap* geregelt. Ein Objekt dieser Klassen hält die wichtigen Sichtdaten für sein View-Objekt, wie z.B. die Daten für die Transformation von Benutzerkoordinaten zu Bildschirmkoordinaten oder andersherum, Rasterinformationen, Zoom- und Sichtgrenzen, Farben, Fangradius, ... etc. Zu jeder View-Klasse gehört eine ViewMap-Klasse, die zuständig für ihre grafische Information ist. Diese ViewMap-Klasse wird von einer Basisklasse für alle ViewMap-Klassen *SH_ViewMap* abgeleitet, und ihre Datenstruktur entsprechend der Besonderheit des zugehörigen View-Typs bestimmt. Dabei erbt sie die Daten und die Methoden der Basisklasse für die Darstellung grafischer Objekte

  Die Verwaltung der ViewMap-Objekte und die damit verknüpften View-Objekte erfolgt in Mengenverwaltungsklassen, die auch ihrerseits von einer Basisklasse *SH_ViewObjManager* für alle Verwaltungsklassen der View-Objekte abgeleitet wird. Ein Objekt dieser Klassen verwaltet die View-Objekte, die zu einer Gruppe von Tragwerksobjekten (z.B. Geschoß, Folie) gehören und den gleichen Sichttyp (View type), wie z.B. 2D- , 3D- oder Text-View haben.

- Eine Schnittstelle für die interaktive Erzeugung von Objekten durch den Benutzer wird durch den Einsatz des sogenannten Builder-Konzepts [22] modelliert. Dies ist eine Konstruktionsmethode von Objekten, die es ermöglicht, ein Objekt auf unterschiedlichen Eingabesequenzen mit verschiedenen Darstellungsarten während der Konstruktionsphase zusammenzubauen. Somit wird die Konstruktion von komplexen Objekten von der Repräsentation dieser Objekte getrennt.

- Eine Schnittstelle zum Datenaustausch wird durch Konvertierung von Datenstrukturen (z.B. ASCII-Format) erzeugt. Da sich der Programmieraufwand mit der Implementierung von Schnittstellen für den Informationsaustausch mit der Anzahl der Austauschformate (DXF, STEP, ..etc.) potentiell erhöht, und für das CAD-System MvCad nicht ein Ziel ist, wird nur eine Schnittstelle für den Datenaustausch der Tragwerkspositionen aus dem System *MicroFe* (Firma mb-Programme, Software im Bauwesen GmbH) implementiert.

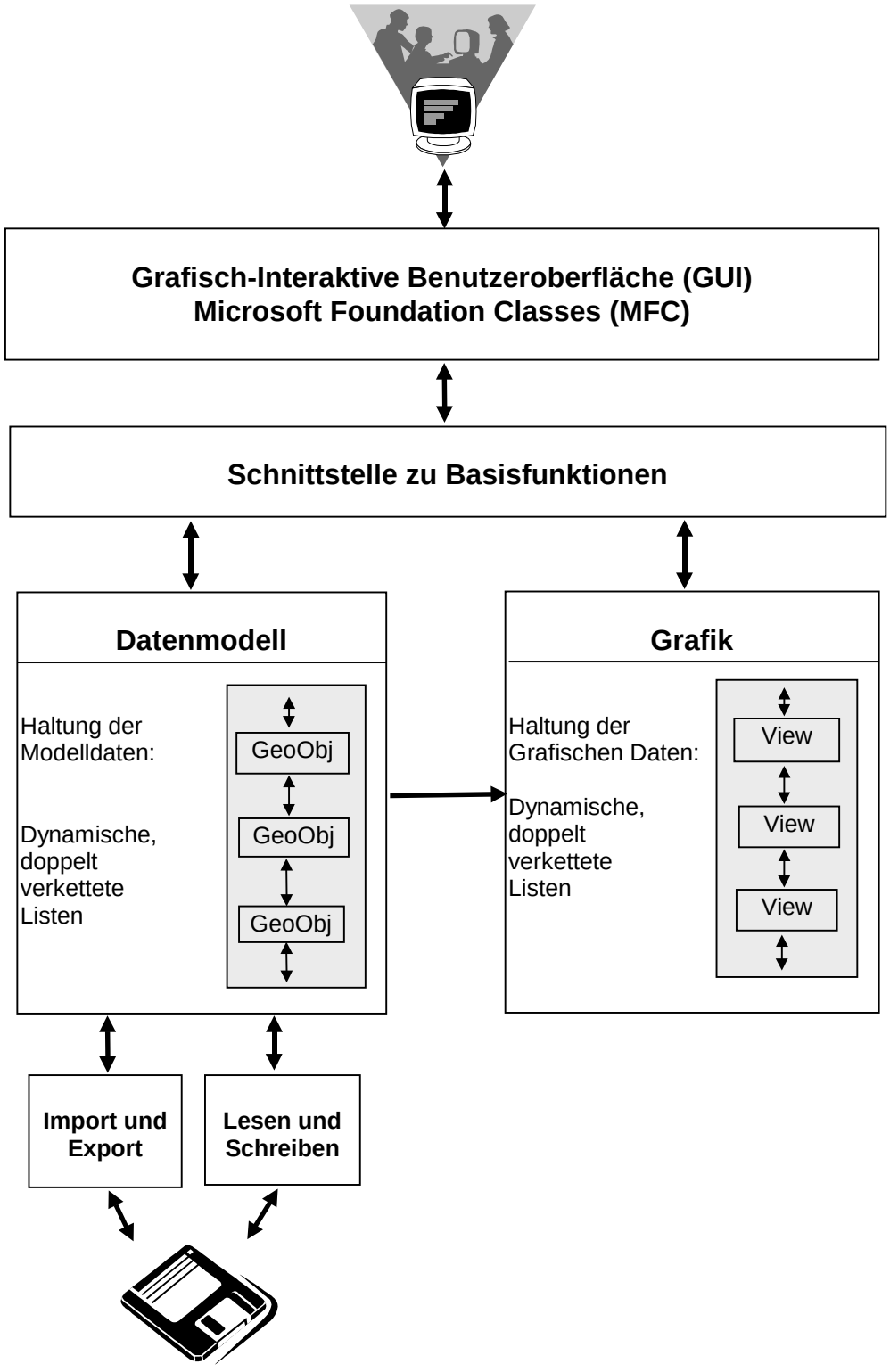


Abbildung 6.1: Designkonzept des Systems MvCad

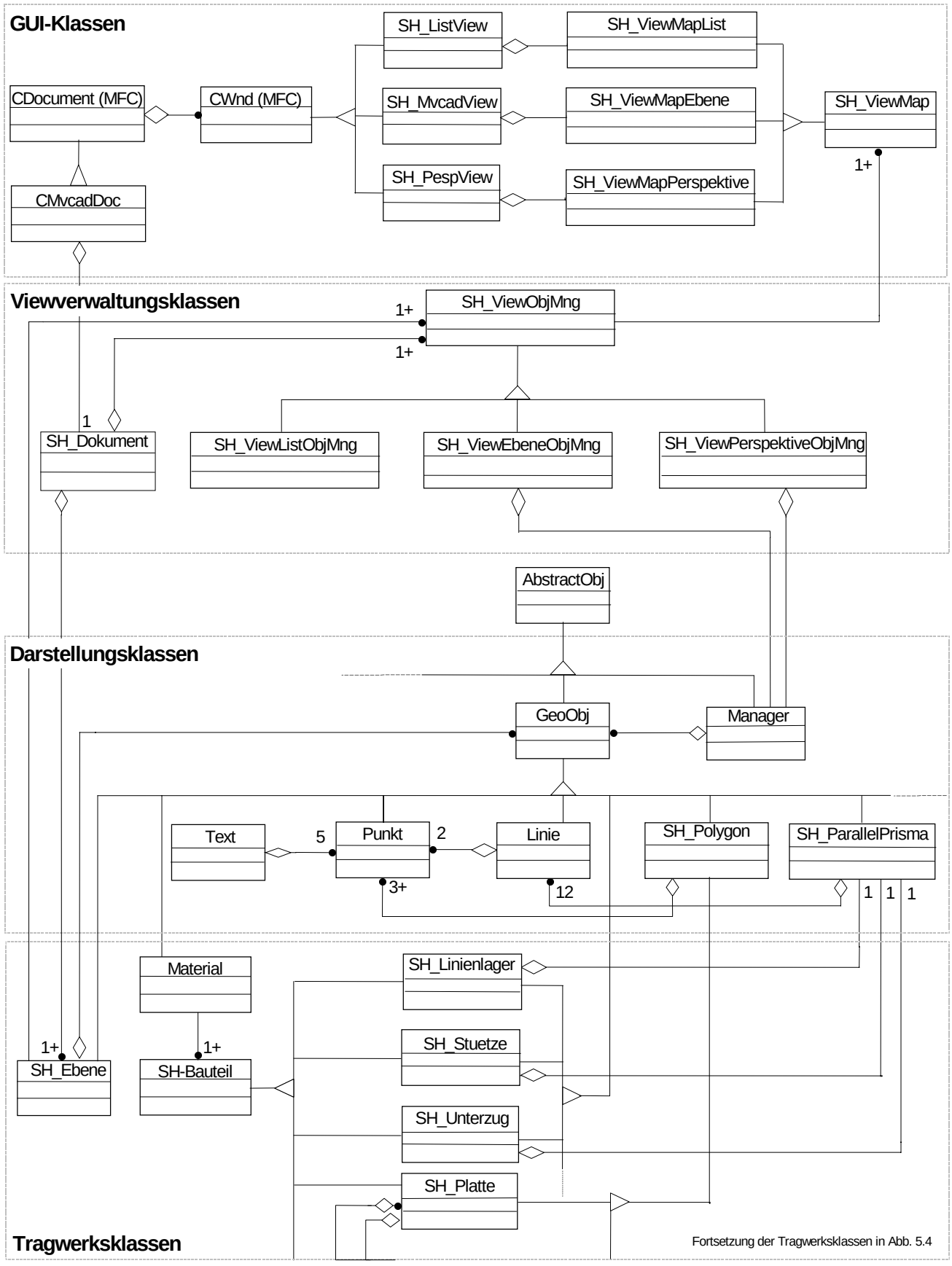


Abbildung 6.1a: Objektmodell des CAD-Systems MvCad

6.3 Datenhaltung und Datenorganisation

Während des Programmlaufs werden Objekte in Mengen erzeugt, die irgendwo im Speicher stehen. Es besteht immer der Bedarf, auf diese Objekte jeder Zeit zurück zu greifen und verschiedene Operationen durchzuführen. Zweckmäßig werden gleichartige Objekte separat gespeichert und zusammen verwaltet. Dafür sind sogenannte Objektcontainer zuständig. Bauteilobjekte, Darstellungsobjekte und grafische Objekte werden in unterschiedlichen Objektcontainern gehalten, deren Größe in der Laufzeit des Programms dynamisch veränderbar ist.

6.3.1 Dynamische Liste

Die grundlegende Datenhaltung des Programmes wird mit einer doppelt verketteten dynamischen Liste realisiert.

Diese Liste verwaltet Zeiger auf Objekte und die Größe der Liste wird erst zur Laufzeit festgelegt, indem Zeiger in die Liste aufgenommen werden. Dem Konzept des Cad - Systems entspricht, daß man eine beliebige Anzahl von Objekten zur Laufzeit erzeugt. Daraus ergibt sich eine dynamische Speicherverwaltung, in der der Speicher zur Laufzeit angefordert und freigegeben wird. Den Zugriff auf ein Objekt erhält man durch seinen Zeiger, sprich seine Speicheradresse. Darüber hinaus werden verschiedene Typen erzeugt.

Die *dynamische Liste*, die zur Verwaltung der Objekte herangezogen wird, muß also die Fähigkeit besitzen, Zeiger verschiedenen Typs zu verwalten.

Die wichtigsten Vorteile einer dynamischen Liste sind im einzelnen [70]:

⇒ Der erste Vorteil einer dynamischen Liste gegenüber eines Arrays ist, daß eine dynamische Liste während der Laufzeit ihre Größe ändern kann. Und vor allem braucht man vorher nicht die maximale Größe der Liste zu wissen.

⇒ Der zweite Vorteil ist, daß sie mehr Flexibilität in der Organisation der Daten zeigt. Diese Flexibilität zeigt sich durch den schnellen Zugriff auf einen Datenmember der Liste

Die dynamische Liste wird zweckmäßig in der Programmiersprache C++ durch die sogenannte Template-Klasse realisiert. Dafür wird sie als „Template“ (Schablone) deklariert und implementiert.

Die Template-Klassen sind parametrisierte Schablonen für Klassen. Die Instanzen von Templates sind Klassen. Es ist auch möglich , ein oder mehrere Typen in Klassen zu parametrisieren. Dies sind vor allem die sogenannten ContainerKlassen, die dazu dienen, andere Objekte zu verwalten.

Das heißt, die Tamplate-Klasse wird nicht für einen bestimmten Typ definiert, sondern für einen noch festzulegenden Typ. Die im Rahmen dieser Arbeit entwickelte Template-Liste *SH_List* bietet also die Möglichkeit, diese zu implementieren, obwohl der Typ der verwalteten Objekte noch nicht feststeht.

Da diese Struktur nach außen gekapselt ist, bestehen Methoden zum Zugriff auf die Objekte in der Liste. Die Template-Klasse *SH_List* hat Methoden um Elemente

- in die Liste aufzunehmen
- aus der Liste zu entfernen

Fernen bestehen noch Methoden

- zum Zugriff auf die Elemente in der Liste
- zur Modifizierung der Liste
- zum Suchen nach bestimmten Objekten in der Liste

Die Deklaration der Template-Klasse *SH_List* in C++ erfolgt folgendermaßen :

```
template <class T>
class SH_List
{
protected:
        struct Node
        {
        Node* pNext;
                Node* pPrev;
                T* object;
        };
        // Member-Daten
        Node* pNodeFirst;
        Node* pNodeLast;
        Node* pNodeAct;
        long nNodeCount;                // Anzahl der Elemente in der Liste
public:
        SH_List();                      //Konstruktor
        SH_List(const SH_List<T>&);     // Copy-Konstruktor
        ~SH_List();                     //Destruktor

        //Abfrage, ob die Liste leer ist
        BOOL IsEmpty() const;

        // Abfrage nach Anzahl der Elemente in der Liste
        long GetCount() const;

        // Zuweisungsoperator
        const SH_List<T>& operator= (const SH_List<T>&);
```

```
        // Liste enleeren, alle Elemente in der Liste werden gelöscht
         void Clear();

         // Addieren vor dem ersten und nach dem letztem Element in der Liste
         // wenn Object in der Liste schon vorhanden ist, wird nicht nochmals addiert
         void AddFirst(T* abstObj);
         void AddLast(T* abstObj);
         void Add(T* abstObj);

        //Ausfuegen des ersten und letztem Element
        T* RemoveFirst();
        T* RemoveLast();
        BOOL Remove(T* abstObj);
        long RemoveAll();

        // Iteration
        T* GetFirst() const;
        T* GetLast() const;
        T* GetAct()const;
         T* GetAt(SH_POSITION position) const;
         T* GetNext(SH_POSITION position) const;
         T* GetPrev(SH_POSITION position) const;
         SH_POSITION GetFirstPos() const;
         SH_POSITION GetNextPos(SH_POSITION position) const;
         SH_POSITION GetLastPos() const;
                SH_POSITION Find(T* abstObj);    //  Suche über die ganze Liste

         // Modifizieren
         void SetFirst(T* abstObj);
         void SetAct(T* abstObj);
         void SetLast(T* abstObj);
};
```

Der grundlegende Zugriff auf die Elemente der Liste hat man über die Struktur *Node.* Hat man einen Zeiger auf ein Objekt *object*, kann man über die Zeiger *pNext* und *pPrev* Zugriff auf das nächste oder vorherige Element erlangen.

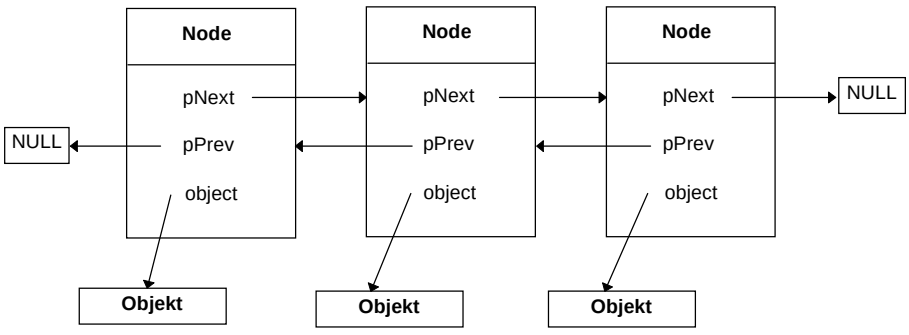


Abbildung 6.2: Eine doppelt verkettete Liste

Um den Anfang und das Ende der Liste beim Suchen zu erkennen, wird immer am Anfang und am Ende der Liste ein NULL-Pointer eingefügt

Eine neue Liste für einen bestimmten Datentyp (Objekttyp) der deklarierten Template-Liste *SH_List* wird beim Gebrauch folgendermaßen definiert :

```
typedef   SH_List<GeoObj>  BASELIST;
```

Dabei muß angegeben werden, welcher Typ als Parameter verwendet werden soll, somit wird die Schablone für den entsprechenden Typ realisiert. In diesem konkreten Fall wird eine Liste von ***GeoObj***-Objekten durch folgende Deklaration

```
BASELIST      m_BaseList;
```

erzeugt. Die Liste BASELIST verwaltet Objekte der Klasse *GeoObj* und Objekte der von der Klasse *GeoObj* abgeleiteten Klassen.

Beispiel einer Schleife über die komplette Liste, um alle Objekte vom Typ *rType* zu deaktivieren:

```
for ( SH_POSITION pos = m_BaseList.GetFirstPos( ); pos != NULL;
                                        pos = m_BaseList.GetNextPos( pos ) )
{
GeoObj* pObj = m_BaseList.GetAt( pos );   // Zeiger auf das Objekt holen
if ( typeid( rType ) == typeid( *pObj ) )      // seinen Typ abfragen
        {                                      // ist es vom Typ rType
                pObj->SetActivity(FALSE);  // Objekt wird deaktiviert
        }
}
```

6.3.2 Verwaltung der Darstellungsobjekte

Darstellungsobjekte sind elementare Objekte, die zur grafischen Repräsentation komplexer Objekte herangezogen werden (Abbildung 6.3). So wird z. B. eine Platte grafisch durch zwei Polygone (ein Objekt der Klasse *SH_Polygon*) und ein Linienlager durch ein Prisma (ein Objekt der Klasse *SH_ParalelPrisma*) repräsentiert.

Die Darstellungsobjekte werden in einem Objekt der Klasse *Manager* verwaltet. Dieses Manager-Objekt ist verantwortlich für das Zeichnen und Identifizieren der Objekte, sowie andere Funktionalitäten, wie das Fangen auf Kanten oder Ecken eines Objekt. Dafür sind die entsprechenden Schnittstellen in jedem Objekt (in seiner Klasse) definiert, der Manager leitet die Nachrichten (Messages) an die Objekte durch diese Schnittstellen weiter.

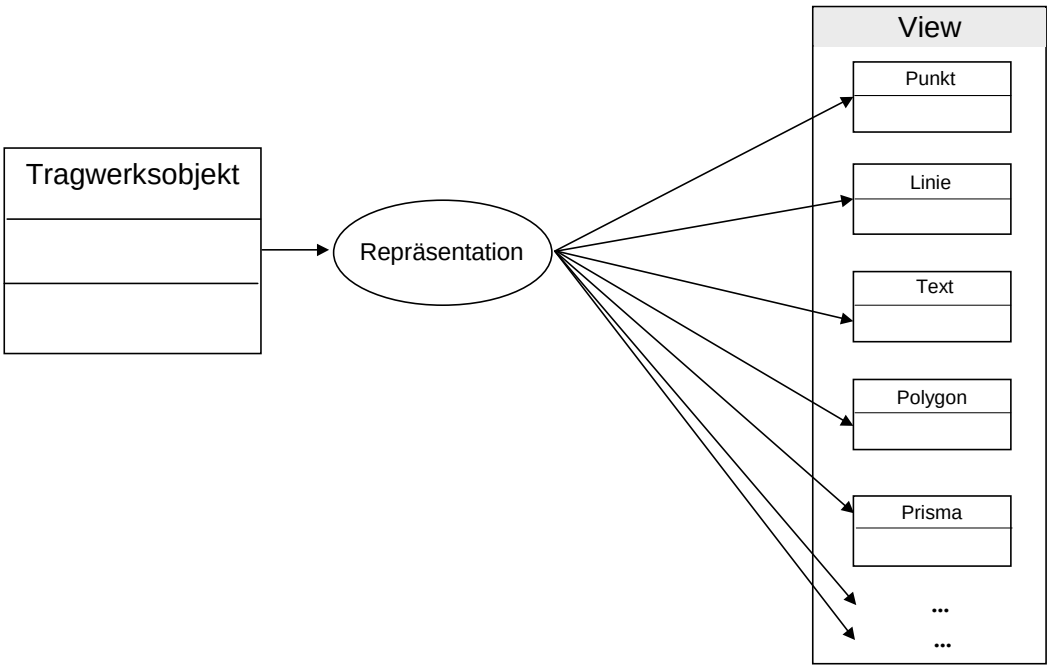


Abbildung 6.3: Mögliche Repräsentation eines Tragwerkobjekts durch Darstellungsobjekte

In einem Manager-Objekt werden die Darstellungsobjekte über dynamische Listen *OrgList* mit der Definition

typedef SH_List< GeoObj> OrgList

verwaltet. Jeder Objekttyp der Darstellungsobjekten wird in einer gesonderten Liste verwaltet. Das heißt, dieser Manager besitzt Listen für die notwendigen grafischen Objekte wie Texte, Punkte, Linie, Polygone etc. (Abbildung 6.4). Er ist zuständig für die Organisation der Darstellungsobjekte in den grafischen Views, indem jeder ViewObjektManager, der alle Views gleicher Art verwaltet, ein Objekt von der Klasse *Manager* besitzt. Damit können alle Views, die von einem ViewObjektManager verwaltet werden, auf die gleichen Darstellungsobjekte präsentieren.

Zur Verwaltungsaufgabe besitzt der Manager verschiedene Methoden wie z. B. Funktionen zum Aus- und Einfügen von Objekten in seine Verwaltungslisten, zum Identifizieren und Selektieren eines Objekts

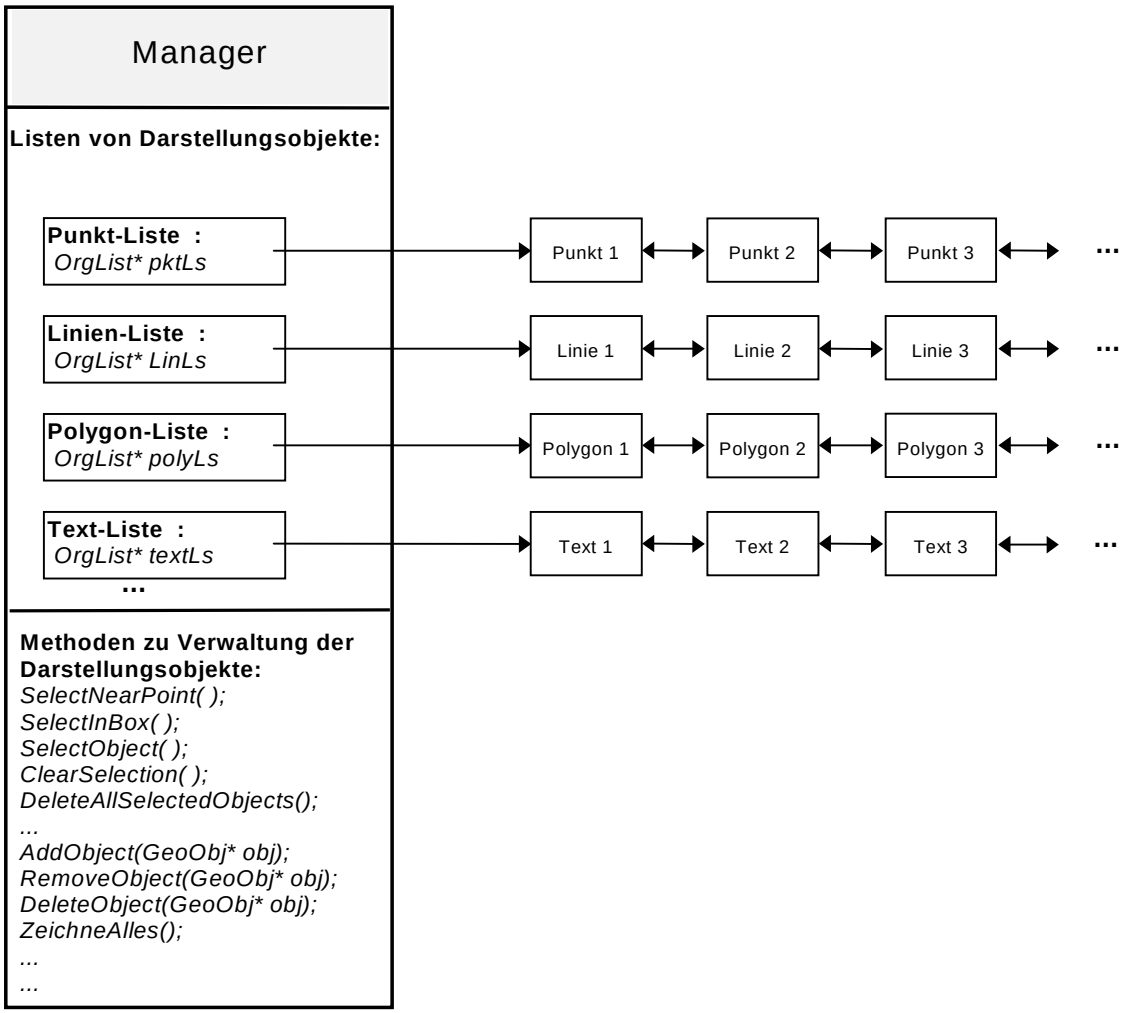


Abbildung 6.4: Managerklasse für Verwaltung der Darstellungsobjekte

Der Manager dient gewissermaßen als Schnittstelle zwischen der Repräsentation eines Objektes und ihm selbst. Da für den Benutzer nur die grafische Repräsentation, sprich das Darstellungsobjekt, sichtbar ist, kann er auch nur auf diese interaktiven Aktionen ausführen. Und über das Darstellungsobjekt kann dann das Bauteilobjekt selbst identifiziert werden .

Folgendes ist die Deklaration der Manager-Klasse in der Programmiersparache C++:

```
class Manager
{
 public:
        ...
        ...
// Liste für die selektierte Objekte
OrgList* m_listSelectedObjects;
```

```
// Hier werden die Listen für die einzelnen Darstellungsobjekte deklariert
        // Liste mit Punkten
        long pkt_count;
        OrgList *pktLs;

        // Liste mit Linien
        long lin_count;
        OrgList *linLs;

        // Liste mit Polygone
        long poly_count;
        OrgList *polyLs;
        ...
        ...
// Alle Darstellungsobjekte  auf ein Ausgabegerät (Bildschirm, Drucker etc.) zeichnen
 void ZeichneAlles(int nr = 0);
        ...

// Methoden zum Selektieren eines Objektes
        long SelectNearPoint(double x1, double y1, double radius);
        long SelectInBox(double lox, double loy, double rux, double ruy);
        ...
        ...
// Methoden zum Löschen eines Objektes
        BOOL DeleteObject(GeoObj* obj);
        BOOL RemoveObject(GeoObj* obj);
        ...
        ...
// Methoden zum Fangen auf ein Objekt
        BOOL FangaufPunkt(Punkt& fangPunkt,double radius);
        BOOL FangaufLinie(Punkt& fangPunkt,double radius);
        ...
        ...
        ...
};
```

6.3.3 Verwaltung der verschiedenen Sichtobjekte (Views)

Im Konzept dieser Dissertation kann ein Objekt des Tragwerksmodells unterschiedlich dargestellt werden. Das heißt, das Modell enthält die Daten und die Views stellen diese Daten in verschiedenen Arten dar. Dabei kommuniziert das Modell mit den Views wenn seine Daten geändert werden, und die Views kommunizieren mit dem Modell, um auf diese Daten zuzugreifen.

Im System MvCad wurden drei verschiedene Sichtobjekte implementiert, und zwar eine 2D-Sicht *CMvcadView*, eine 3D-Sicht *CPerspView* und eine Text-Sicht *CSHListView* (Abbildung 6.5).

In den verschiedenen Sichten repräsentiert sich ein Objekt auf verschiedene Art. Das gleiche Objekt ist in der 2D-Sicht im Grundriß und in der 3D-Sicht in einer Perspektive zu sehen. Ferner besteht noch die Möglichkeit, die Objektdaten in der Text-Sicht als Text auszugeben.

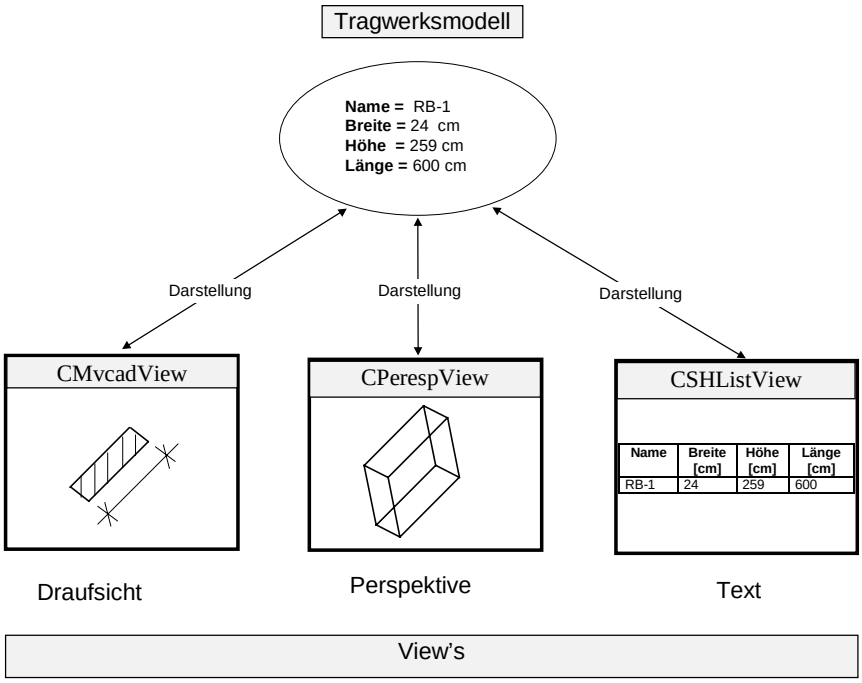


Abbildung 6.5: Die Entkopplung der Daten im Tragwerksmodell von ihren Darstellungsmöglichkeiten

Da für jede Sichtart mehrere View-Objekte erzeugt werden können, besteht der Bedarf, diese durch eine Container-Klasse zu verwalten.

Die Klasse *SH_ViewObjManager* ist die Basisklasse für die Verwaltungsklassen aller Sichttypen. Die von ihr abgeleiteten Klassen sind zuständig für die Verwaltung der für die Darstellung erforderlichen grafischen Objekte. In ihnen werden die Darstellungsobjekte für die jeweilige Sicht (View) verwaltet.

Für jeden Viewtyp ist eine neue *ViewObjManager*-Klasse abzuleiten. Die Klasse beschreibt, wie aus Anwendungsobjekten entsprechende Darstellungsobjekte erzeugt und in die Displaylisten eingehängt werden.

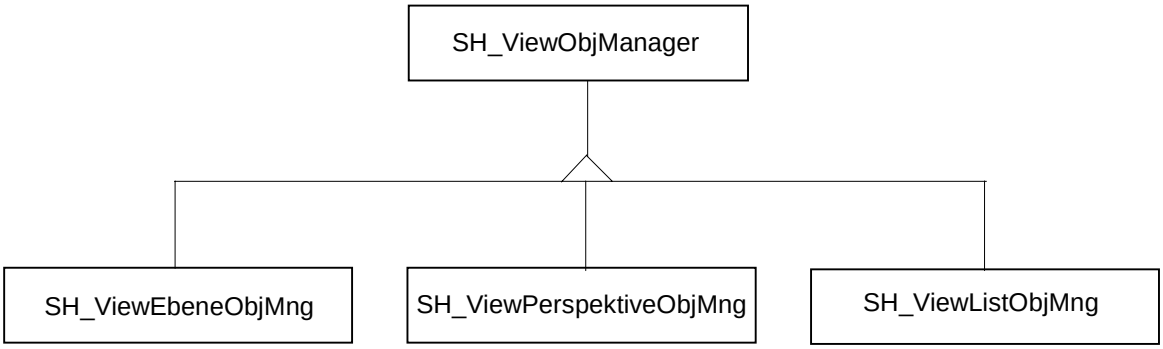


Abbildung 6.6: Klassenhierarchie der verschiedenen View-Manager

In vorliegendem Programm sind drei Sichttypen implementiert, dies sind eine 2D-, eine 3D- und eine Textsicht. Jedem Sichttyp (View type) entspricht genau eine *SH_ViewObjManager*-Klasse. Die Klassenhierarchie der verschiedenen View-Manager ist in der Abbildung 6.6 gezeigt.

Ein Objekt der Klasse *SH_ViewPerspektiveObjMng* verwaltet z.B. alle Daten der zugehörigen 3D-View-Objekte und besitzt eine dynamische Liste von Assoziation-Klassen *SH_ViewMap*.

Eine Klasse vom Typ *SH_ViewMap* verwaltet die für eine bestimmte Sicht (View) relevanten grafischen Daten. Sie ist das Bindeglied zwischen der von MFC generierten Sichtklasse CView (View-Klasse von MFC) und den eigenen Viewklassen, die zur Darstellung der Grafik erzeugt wurden. Eine von *SH_ViewMap* abgeleitete Klasse muß in jede CView eingehängt werden. Die Klasse *SH_ViewMap* besitzt einen Zeiger auf den für die Verwaltung der Darstellungsobjekte verantwortlichen *SH_ViewObjManager* und die selbst erzeugte Viewklasse, die die View-spezifischen Daten wie die Umrechnung von Bildschirm- in Weltkoordinaten, Raster- und Fangeinstellungen verwaltet.

Jedem View-Objekt ist eine Gruppe von Tragwerksobjekten, die in einer Gruppierungsklasse *SH_Ebene* (z.B. ein Geschoß oder eine Folie) verwaltet werden, zugeordnet. In ihr werden die dem View-Objekt zugehörigen Objektdaten verwaltet.

Die Gruppierungsklasse *SH_Ebene* besitzt die Tragwerksobjekte und nicht die Darstellungsobjekte, die zur graphischen Darstellung herangezogen werden.

Diese Folien werden in der eigenen Dokument-Klasse *SH_Document* über eine dynamische Liste verwaltet. Diese Klasse ist zuständig für die Datenhaltung des Tragwerksmodells.

Die MFC-Dokument-Klasse CMvcadDoc besitzt ein Objekt der Klasse *SH_Document* und leitet alle Operationen auf die Tragwerksobjekte an ihn weiter.

Ein Objekt von MFC-View hängt ein Objekt der Klasse *SH_Viewmap* im Create-Event in das zuständige *SH_ViewObjManager* ein. Die Verwaltung der für den Benutzer wichtigen Repräsentation obliegt also den *SH_ViewObjManager*.

Ein Objekt der Klasse *SH_ViewObjManager* besitzt ein Manager-Objekt der Klasse *Manager*, das die Darstellungsobjekte verwaltet. Für die Zuordnung der Darstellungsobjekte zu den gehörigen Tragwerksobjekten ist eine Liste von Verknüpfungsobjekten (Assoziationsobjekte) zuständig, die eine Verbindung zwischen einem Tragwerksobjekt und seinen Darstellunksobjekten in dem jeweiligen *SH_ViewObjManager* herstellt. Ein Verknüpfungsobjekt stellt eine Assoziation 1 zu n (one-to-many) dar. Jedes Assoziationsobjekt hat einen Zeiger auf ein Tragwerksobjekt und eine Liste von Zeigern auf seine gehörenden Darstellungsobjekte. Somit sind die Darstellungsobjekte einem einzigen Tragwerksobjekt zugeordnet. In der Abbildung 6.7 ist eine Assoziationsklasse gezeigt, die die Beziehung eines Tragwerksobjekts zu seinen Darstellungsobjekten in einer Ebene hält.

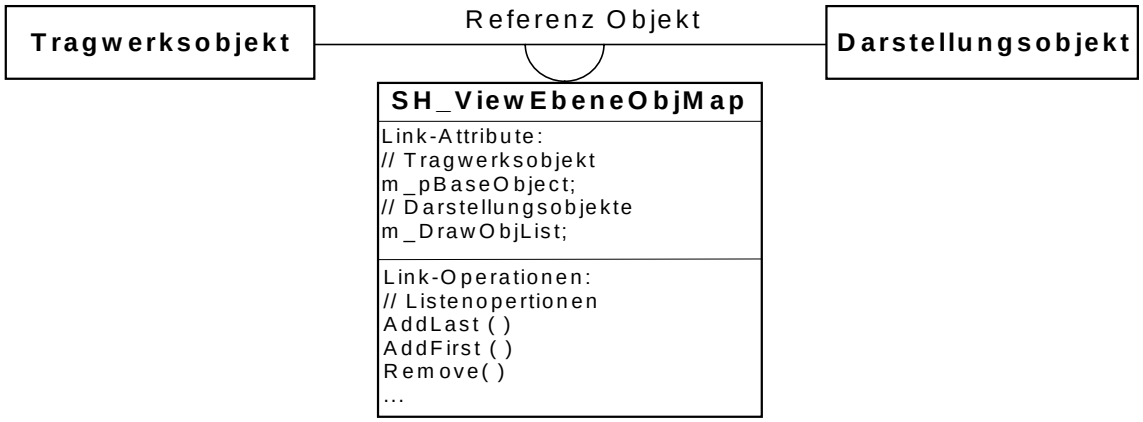


Abbildung 6.7: Assoziationsklasse zwischen einem Tragwerksobjekt und seinen Darstellungsobjekten in einer Ebene

Folgendes ist ein Beispiel für die Deklaration der Klasse *SH_ViewPerspektiveObjMng* in der Programmiersparache C++:

```
class SH_ViewPerspektiveObjMng : public SH_ViewObjManager
{
        ...
        ...
// Memberdaten:
        // Ein Manager für die Verwaltung der Darstellungsobjekte
        Manager*            m_pOBM;

        // Liste der Objekt-Maps, Zuordnung enthaltener Tragwerksobjekte zu seinen
        // Darstellungsobjekten
```

```
        SH_VIEWPERESPEKTIVEOBJMAPLIST   m_ObjMapList;
        ...
        ...
// Memberfunktinen:

        // Mit der Methode DrawInsert() werden die Darstellungsobjekte in den
        // Manager eingefügt und eine Objekt-Map zur Verbindung mit dem
        // zugehörigen Tragwerksobjekt erzeugt.
        ...
        // Polygon in Displayliste einfügen (für alle Views)
                virtual void DrawInsert( GeoObj*, SH_Polygon*, SH_ViewMap* = 0
);

        // Linie in Displayliste einfügen (für alle Views)
                virtual void DrawInsert( GeoObj*, Linie*, SH_ViewMap* = 0 );
        ...
        ...
        // Mit der Methode DrawRemove() wird die Objekt-Map eines Tragwerks-
        // objektes gelöscht und seine Darstellungsobjekte werden aus dem Manager
        // ausgetragen. Objekt wird aus Displayliste entfernt (für alle Views)
                virtual void DrawRemove( GeoObj* );
        ...


        // Ist ein Tragwerksobjekt selektiert wird über folgende Methoden die
        Selektierung an // das Darstellungsobjekt weitergegeben.
        ...
        // Objekt selektieren
                virtual void SelectObject( GeoObj* );

        // Objekt deselektieren
                virtual void DeselectObject( GeoObj* );

        // Alle Objekte deselektieren
                virtual void DeSelectAllSelected();
        ...
        ...
// Methoden dienen zum Verwalten der Objekt-Map-Liste.
        ...
        // zu Darstellungsobjekt gehöriges ObjektMap suchen
                SH_ViewPerspektiveObjMap* FindObjectMap( GeoObj* );

        // Darstellungsobjekt in ObjektMap-Liste hinzufügen
                void InsertObjectMap( GeoObj*, GeoObj* );

        //  zu Darstellungsobjekt gehöriges Tragwerksobjekt  suchen
        virtual GeoObj* FindBaseObject( GeoObj* pDrawObj );
        ...
        ...
};
```

6.4 Grafisch-Interaktive Benutzeroberfläche (GUI)

Der Einsatz der Fenstertechnik bei der Realisierung von Benutzeroberflächen hat sich in den letzten Jahren in der Softwarebranche etabliert und die moderne Software in allen Bereichen bezeichnet. So besitzen moderne Programme im Bauwesen eine grafische Oberfläche als einheitliches Erscheinungsbild, die anhand von Fensterkomponenten dem Benutzer ermöglichen, mit System intuitiv und interaktiv zu arbeiten. Dabei wird der Ablauf und die Verzweigung des Programms vom Benutzer bestimmt, und nicht sequentiell von dem Programm vorher definiert.

Die ausgewählte Plattform für das entwickelte CAD-System *MvCad* ist das Betriebsystem Microsoft Windows 95. Für die Darstellung der Oberfläche wird die Fenstertechnik benutzt, die alle Windows-Programme (Windowsanwendung) einprägt und vom Betriebssystem stark unterstützt wird [19] [61].

Folgendes ist eine kurze Beschreibung der Komponenten und der Eigenschaften eines Windowsprogramms.

6.4.1 Elemente einer Windowsanwendung (Windows Application):

Windowsprogramme, auch Windowsanwendung genannt (Windows Application), zeigen an der Oberfläche einen typischen Aufbau, bedingt durch die von allen Programmen genutzten Darstellungselemente, die Windows zur Verfügung stellt [19].

Zwar muß der Programmierer einer Windowsanwendung recht viel Aufwand betreiben, um ein wirklich funktionales Programm zu erhalten, allerdings wird er durch die Mechanismen von Windows und die zur Verfügung stehenden Funktionen und grafischen Elemente sehr stark unterstützt.

- **Fenster**

Für die Darstellung eines Programms sowie der Dialogelemente ist das Fenster das zentrale Element.

Es gibt etliche verschiedene Fenstertypen unter Windows, von denen man benötigte Fenster ableiten kann.

Hauptfenster sind spezielle Ausführungen der Fensterklassen, die überall auf dem Bildschirm erscheinen dürfen. Die Abbildung 6.8 zeigt das Hauptfenster des System MvCad.

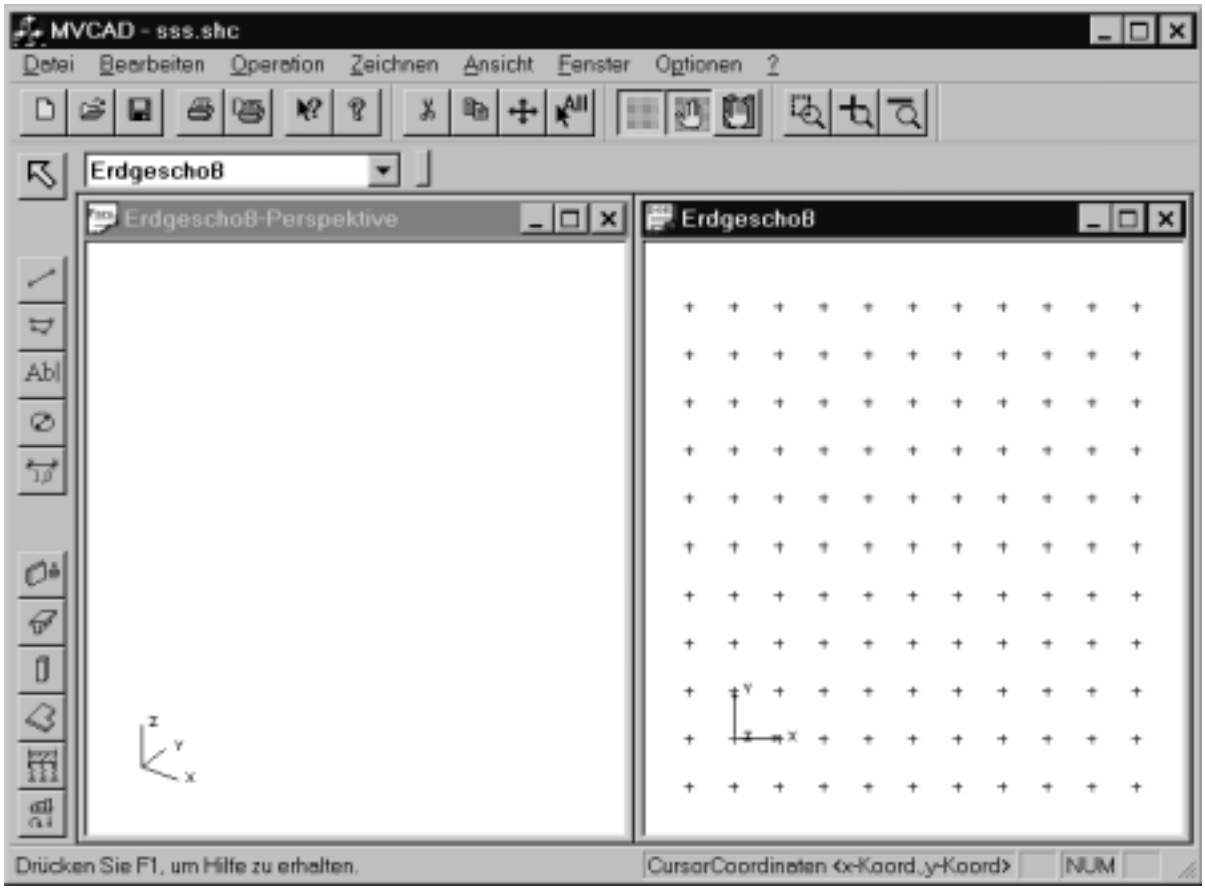


Abbildung 6.8: Programmfenster von MvCad

Ein anderes Beispiel für Fenstertypen sind die Dialog- und Hilfefenster eines Programms.

Kindfenster (*WS_CHILD*) sind nur innerhalb der Fensterfläche ihres Elternfensters anzuordnen, wenn sie darüber herausragen sollten, werden sie abgeschnitten.

„Controls“ sind vordefinierte Fensterklassen, die vor allem zur Gestaltung der Dialogfenster genutzt werden.

Name	**Beschreibung**
BUTTON	Schaltfläche
COMBOBOX	Kombinationslistenfeld
EDIT	Eingabefeld
LISTBOX	Listenfeld
MDICLIENT	MDI-Fenster (Multy Document Interface)
SCROLLBAR	Rollbalken
STATIC	Statisches Element

- **Menüs**

Durch das Menüsystem wird der Benutzer zu den verschiedenen Möglichkeiten des Programms geleitet. Dadurch entfällt das früher bei PC-Programmen nötige Auswendiglernen der Befehle des Programms.

Um den Aufbau eines Menüs und vor allem seine Abarbeitung, braucht der Programmierer sich kaum zu kümmern. Er muß lediglich definieren, welche Menüpunkte dargestellt werden sollen, welche Befehlsnummern mit einem Menüpunkt in Zusammenhang stehen und auch die Funktionen schreiben, die für die Ausführung der Menübefehle zuständig sind. Um den Rest, also die Darstellung, die Reaktion auf die Maus und die Tastatur und die richtige Verteilung der Befehle kümmert sich Windows.

- **Dialoge**

Frühere Programme haben sich mit dem Benutzer meist nur über sehr kurze Bildschirmausgaben „unterhalten“, sowie oftmals keine Hilfestellung bei ihrem Aufruf gegeben.

Die Verbesserung, die Windows hier bietet, sind Dialoge. Dialoge können aus beliebig vordefinierten Fensterklassen (*Controls*) gestaltet werden, die der Kommunikation mit dem Anwender dienen. Mit diesen Dialogen kann auch eine Meldung ausgegeben werden, die zur Kenntnis genommen wird (*OK*) oder zum Abbrechen der kritischen Handlung (*Weiter, Abbrechen*) führt. Die Ausgabe erfolgt in kleinen Fenstern mit festem Rahmen (*MessageBox*).

Prinzipiell gibt es drei Sorten von Dialogen:

1. Nicht modale Dialoge
   die es zulassen, daß sie, ohne sie zu beachten mit der Arbeit fortfahren. Die Anwendung wird also nicht blockiert, bis das Dialogfenster wieder geschlossen wird.

2. *Modale Dialoge*
   blockieren dagegen die Anwendung solange, bis sie wieder geschlossen sind. Ein Beispiel dafür ist der Dateiauswahldialog.

3. *System-modale Dialoge*
   halten das gesamte System solange an, bis der Anwender das Dialogfenster bedient hat.

- **Weitere grafische Elemente**

Windows stellt Ihnen neben diesen Hauptelementen für die Benutzeroberfläche noch einige weitere Bestandteile zur Verfügung, die das Programm funktionaler machen können:

⇒ Symbolbilder:

Die Symbolbilder (*Icons*) können benutzt werden, um in einer Symbolleiste Befehle zu repräsentieren. In diesem Fall ist das Programm so aufgebaut, daß es nach einem Anklicken des Symbols eine bestimmte Aktion startet.

⇒ Mauscursor:
Der Mauscursor hat die Standardform des Pfeils, kann aber bei Bedarf angepaßt werden. Er verleiht dem Programm eine höhere Aussagekraft.

⇒ Bitmap:
Das wichtigste grafische Gestaltungselement ist das Bitmap. Meist handelt es sich dabei um zu dem Programm dazugeladene Bilder im Paintbrush-Format (.BMP, .PCX) oder in einer speziellen Form kodierte Grafikdateien.

- **Geräteunabhängige Schnittstelle für grafische Ausgabe GDI (Graphics Device Interface)**

  Windows unterscheidet nicht zwischen Text und Grafik, wie man z. B. vom Betriebsystem DOS gewohnt war, sondern es wird die Ausgabe von beiden gemeinsam behandelt. Das Betriebsystem stellt eine Vielzahl von Funktionen zur grafischen Ausgabe zur Verfügung, wie z.B. zur Ausgabe von Texten, Punkten, Linien oder Bitmaps. Diese Funktionen werden unter dem Begriff **GDI** zusammengefaßt.

  Ein großer Vorteil des GDI ist, daß die Ausgabe von Grafik geräteunabhängig erfolgt. Das heißt, es ist ziemlich gleichgültig, ob die Ausgabe auf dem Bildschirm oder einem Laserdrucker erfolgt. Windows besorgt die nötige Umsetzung.

- **Der Gerätekontext (Device Context DC)**

  Ein Gerätekontext (Device Context **DC**) ist eine Verbindung zwischen einem Windows-Programm und einem Gerätetreiber (Device Driver). Jedesmal, wenn ein Programm etwas auf dem Bildschirm oder Drucker ausgeben will, braucht es einen Gerätekontext.

  Ein Gerätekontext ist eine Datenstruktur, in der viele Informationen, wie zum Beispiel die Vorder- und Hintergrundfarbe, Schriftart, Farbmuster usw., gespeichert sind. Damit der Aufwand für den Programmierer bei dem Aufruf der Grafikfunktionen nicht beträchtlich erhöht wird, sind Parameterlisten in Form des Gerätekontextes vordefiniert.

- **Interne Abläufe:**

Im Gegensatz zu klassischen Programmen ist der Programmaufbau so, daß nicht das Programm, sondern der Anwender die Reihenfolge seiner Arbeitsschritte bestimmt.

Windowsprogramme sind ereignisorientiert. Alle Informationen werden durch Meldungen, die durch ein Maus- oder Tastaturereignis oder durch andere Fenster ausgelöst werden, ausgetauscht.

Eine Mausbewegung oder ein Klick verursacht eine Meldung, die für das Fenster gedacht ist, welches gerade das Zugriffsrecht auf die Maus besitzt, und eine Tastatureingabe wird durch das Fenster entgegengenommen, welches den Fokus (Aufmerksamkeit) von Windows besitzt. Die Meldungen werden zu einem Paket zusammengeschnürt, das Auskunft darüber gibt, welches Fenster von der Meldung betroffen ist und welche Meldung eingegangen ist.

- **Grundlogik eines Windowsprogramms**

Jedes Windowsprogramm funktioniert prinzipiell nach dem gleichen Schema:

1. Anmelden und Definieren der Fensterklasse des Hauptfensters vor der ersten Benutzung, sowie die Fensterklassen anderer im Verlauf des Programms benötigter Fensterobjekte.

2. Initialisierung des Programmfensters.

3. Bearbeitung der *Application Message Queue*, das heißt der Entnahme und Vorbereitung der darin befindlichen Meldungen.

4. In der Bearbeitungsfunktion des Hauptfensters stehen dann als große Fallunterscheidung die Funktionsaufrufe und Anweisungen, mit denen das Programm auf Ereignisse reagiert.

5. Ausgehend von diesem Hauptfenster wird in die weiteren Funktionen des Programms verzweigt und die Ausgaben angezeigt.

6. Wenn eine Reaktion auf ein Benutzerereignis erfolgt ist, kehrt das Programm wieder zur Meldungsschleife zurück, und das nächste vorhandene Ereignis kann nach dem gleichen Schema abgearbeitet werden.

Zur Realisierung der Benutzerschnittstelle in Form einer objektorientierten grafischen Benutzeroberfläche GUI (Graphical User Interface) wird die Microsoft Klassenbibliothek MFC (Microsoft Foundation Class Library) für Windows angewendet. Sie stellt die Funktionen für Windows-Programmierung in Klassen und Strukturen der Programmiersprache C++ dar und verkapselt die API-Funktionen (Application Programming Interface), die von der Software-Firma Microsoft als Schnittstelle für Programmierung von Windows-Applikationen unter dem Betriebsystem Windows zur Verfügung gestellt wird.

6.4.2 Microsoft Foundation Class Library (MFC)

- **Einführung**

MFC (Microsoft Foundation Class Library) ist die C++ -Programmierungsschnittstelle für das Betriebsystem Microsoft Windows. Sie bringt viele Vorteile [41], nicht zuletzt folgende

- Man kann (manchmal muß) die API-Funktionen mit MFC weiter benutzen
- Eine Sammlung von Klassen, die für allgemeine Zwecke entworfen sind, z.B. für: Array, List, Maps, Strings, Time, Date, File access, Scrolling Windows, Splitter Windows etc.
- Unterstützung für die Ausgabe auf Drucker und die Ansicht der Druckausgabe
- Unterstützung für OLE-Automatisierung und *OLE-Controls*
- Die Möglichkeit der Benutzung der *Microsoft Visual Basic controls*
- Unterstützung von Kontext-abhängiger Hilfe (*context ensetive help*)
- Unterstützung für ODBC (Open Database Connectivity) für den Zugriff auf Datenbanken verschiedener Formate.
- Viele Besonderheiten

- **Das *Application Framework***

- Der Arbeitsrahmen der Applikation (*Application Framework)* ist eine Gruppe von objektorientierten Softwarekomponenten, die die Funktionalität enthält, um eine Windows-Applikation zu erzeugen.
- Diese Softwarekomponenten stellen eine Gruppe von Klassen der MFC-library dar, die so entworfen sind, daß sie in jedes Windows-Programm eingebunden werden können, dabei bilden sie die Struktur des Programms selbst.

- Ein Beispiel für ein *Application Framework* einer minimalen Applikation kann mit dem *AppWizard* (Hilfsprogramm zur Erstellung von Windows-Applikationen) erzeugt werden, es hat nur zwei abgeleitete Klassen aus den MFC-Klassen *CWinApp* und *CFrameWindows.*

  - Ein *Application Framework* der *MFC library* enthät typischerweise abgeleitete Klassen der folgenden Klassen:

    CWinApp, AFrameWnd, CDocument und *CView*

- **MFC Message Mapping**

  - Jede Windows-Nachricht (*Message*) muß mit einer Member-Funktion (*Member function)* verbunden werden.

  - Diese C++ Member-Funktionen sind wegen Speicherplatzersparnis (4 Bytes für jede virtuelle Funktion) für das begleitete *Dispatch Table* (vtabel) nicht virtuell deklariert, deshalb werden Macros für die Verbindung mit den entsprechenden Messages benutzt (message mapping).

    z.B.: Windows Message → Member Function

    WM_LBUTTONDOWN → OnLButtonDown ( )

- **Document and Views**

  - Die Document-View Architektur trennt die Daten (im Dokument) von der Benutzeransicht (User's view).

  - Die Basis-Dokument-Klasse behandelt die Menüelemente (*Menu Items*) wie Datei Öffnen (*File Open*) und Datei Speichern (*File Save*). Die abgeleitete Klasse von der Dokument-Basis-Class kümmert sich um das tatsächliche Lesen und Schreiben.

  - Der Arbeitsrahmen der Applikation (*Application Framework)* übernimmt die Arbeit für die Darstellung der Dialog-Boxen, z.B. für Datei Öffnen (*File Open*), Datei Speichern (*File Save*) und Speichern unter (*File Save As*) .....

  - Die Basis-View-Klasse stellt ein Rahmenfenster (*Frame Window*) dar

  - Die abgeleitete View-Klasse arbeitet mit der Dokument-Klasse zusammen, um die Applikation auf dem Bildschirm und auf dem Drucker darzustellen. Sie behandelt auch mit der Basis-View-Klasse alle *Windows Messages*.

- MFC behandelt alle Interaktionen zwischen Dokumenten, Views, Frame Windows und Applikationsobjekten, meistens durch virtuelle Funktionen.

- Man kann die Member-Funktionen der Dokument-Klasse überschreiben, z.B. um bestimmte Dateien in der *File-Open-Dialog-Box* voranzustellen.

- Das Dokument hat eine Liste mit ihren Sicht-Objekten (Views). Damit ist der Zugriff auf alle vorhandenen View-Objekte über das Dokument verwaltet, wie z.B. die Methoden *RemoveView( )* und *AddView( )* und zum Neuzeichnen aller Views *UpdateAllViews( )*.

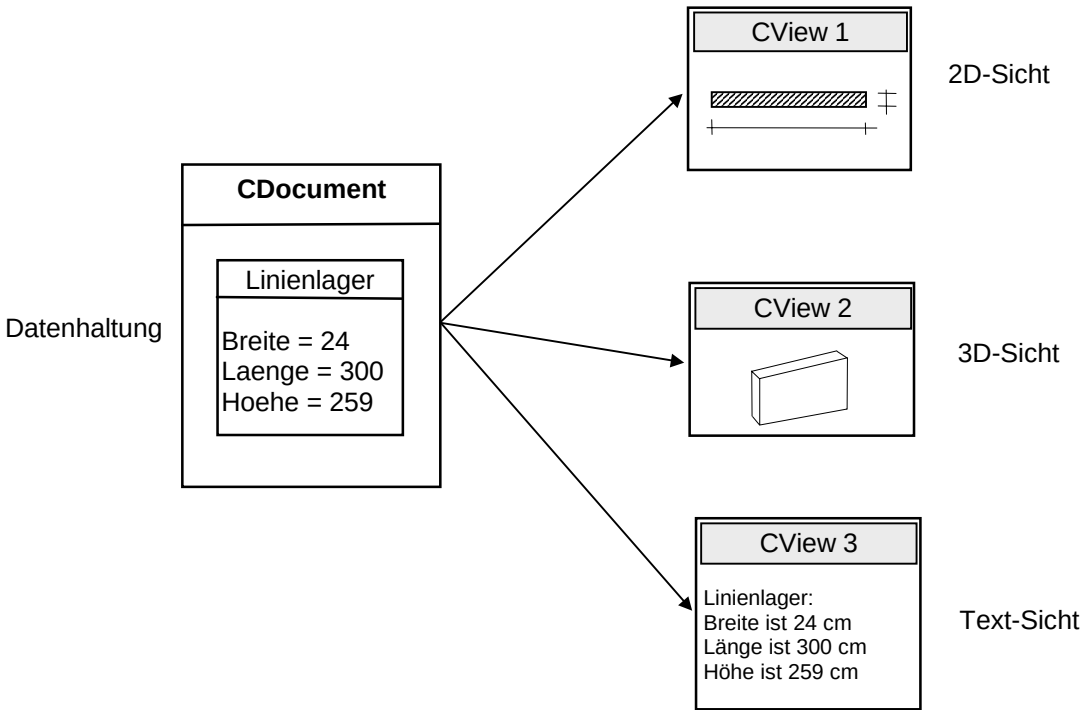


Abbildung 6.9: Document-View Architektur (MFC)

6.5 Konzept für ein grafisches Interaktionsmodell

6.5.1 Vorstellung des Konzepts

Bei komplizierter Objektstruktur besteht der Bedarf, das Objekt auf unterschiedlichen Eingabesequenzen mit verschiedenen Darstellungsarten während der Konstruktionsphase zusammenzubauen. Die Eingabesequenzen in einem CAD-System sind oft sehr verschachtelt und voneinander abhängig, so daß die Erzeugung von Objekten zu einem komplizierten Prozeß wird, die recht überdacht werden soll und ein Modellkonzept für die interaktive Erzeugung von Objekten durch den Benutzer modelliert wird.

Eine Methode der Erzeugung von Objekten wird in der Literatur auch als *Builder* bezeichnet [22]. Builder trennen die Konstruktion von komplexen Objekten von der Repräsentation dieser Objekte, so daß derselbe Konstruktionsprozeß verschiedene Repräsentationen erzeugen kann.

Das Builder-Konzept wird als Konstruktionsmethode von Objekten benutzt, es findet viele Anwendungsbereiche und wird eingesetzt wenn

- der Algorithmus für das Erzeugen eines komplexen Objektes unabhängig von den Komponenten ist, aus denen das Objekt zusammengebaut ist, und wie diese erstellt sind.

- der Konstruktionsprozeß eines Objekts unterschiedliche Darstellungen eines Objektes erlauben soll.

Das Builder-Modell hat folgende Komponenten:

- **Builder** (Basisklasse)
  - stellt eine abstrakte Schnittstelle zur Erzeugung eines Produkt-Objektes dar

- **Konkrete Builder** (jeder der abgeleiteten Builder-Klassen)
  - erzeugt ein Produkt-Objekt unter Zuhilfenahme der Builder-Schnittstelle
  - definiert und merkt sich die Darstellung des Produkts, das es erzeugt
  - bietet eine Schnittstelle zum Anfordern des Produktes

- **Director** (Verwaltungsklasse der Builder)
  - erzeugt ein Objekt unter Zuhilfenahme der Builder-Schnittstelle

- **Produkt**
  - stellt ein komplexes Objekt während der Erzeugung dar. Konkrete Builder erzeugen die interne Darstellung eines Produktes und definieren den Prozeß, mit dem ein Objekt zusammen gebaut wird.

  - beinhaltet Klassen, die die festgelegten Teile definieren, einschließlich der Schnittstellen für den Zusammenbau der Teile in das Endresultat.

Merkmale des Builder-Konzeptes [22]

1. ***Man kann die interne Präsentation eines Produktes variieren.*** Das Builder-Objekt stellt eine abstrakte Schnittstelle bereit, mit der der Direktor das Produkt erzeugen kann. Diese Schnittstelle erlaubt dem Builder die interne Struktur und Repräsentation eines Produktes zu verbergen. Weiterhin wird ebenfalls verborgen wie ein Produkt zusammengesetzt wird. Da das Produkt durch eine abstrakte Schnittstelle erzeugt wird, kann man, wenn man die interne Repräsentation eines Produktes ändern will, einfach einen neuen Builder definieren.

2. ***Der Code für die Erzeugung und die Repräsentation werden getrennt.*** Die Builder-Methode stärkt die Modularität, indem sie die Erzeugung eines komplexen Objektes von dessen Repräsentation abkapselt. Der Client muß nichts über die Klassen, die die interne Struktur eines Produktes beschreiben, wissen, da diese Klassen nicht im Builder - Interface erscheinen. Jeder konkrete Builder enthält den Code zur Erzeugung des jeweiligen Produktes. Der Code muß nur einmal geschrieben werden. Verschiedene Direktoren können dann mit dem Builder verschiedene Varianten eines Produktes erzeugen.

3. ***Man hat eine genauere Kontrolle über den Erzeugungsprozeß.*** Die Builder-Methode erzeugt ein Produkt Schritt für Schritt unter der Kontrolle des Direktors. Erst wenn das Produkt fertiggestellt ist, bekommt es der Direktor vom Builder übertragen.

6.5.2 Implementierung (Objekt-Maker)

Meistens besteht eine Basisklasse für alle Builder-Klassen, die die Operatoren als Schnittstelle definiert, welche vom Direktor aufgerufen werden können. Diese Operatoren machen in ihrer Default-Einstellung nichts. Eine konkrete Builder-Klasse überschreibt Operatoren oder Komponenten dieser Klasse, welche für ein Objekt dieser Klasse interessant sind. Hier sind noch einige Implementierungskriterien [22]:

- Eine Schnittstelle zum Aufstellen und Erzeugen der Objekte. Builders konstruieren ihre Produkte Schritt für Schritt (step-by-step). Deshalb muß die Builder-Klasse als Schnittstelle so allgemein sein, daß Produkte (Objekte in der Konstruierungsphase) für alle möglichen konkreten Builder erzeugt werden können.

- Keine abstrakte Basisklasse

- Keine rein virtuellen Funktionen (pure virtual function) sondern Default-Funktionen in der Basisklasse der Builder. Damit müssen abgeleitete Builder-Klassen nur solche Funktionen überschreiben, die für sie interessant sind.

Das Konzept der Builder wird durch ein Klassenmodell realisiert, das für das CAD-system und sein Datenmodell geeignet und jederzeit nach Bedarf beim Einbauen neuer Bauobjekte oder Eingabesequenzen erweiterbar ist. Das Klassenmodell, das sogenannte Builder-Modell, hat folgende Komponenten:

⇒ **Builder**
Die Klasse *SH_ObjectMaker* stellt eine Basisklasse für alle Erzeugungsklassen oder sogenannte *ObjectMaker* dar. Die konkrete Arbeitsweise der Schnittstelle zur Erzeugung eines Objektes wird in der zugehörigen Erzeugungsklasse *SH_StuetzeMaker*) definiert.

⇒ **Konkrete Builder** (abgeleitete Builder-Klassen)
Sie sind von der Basisklasse *SH_ObjectMaker* abgeleitete „ObjectMaker“. Mit ihnen werden die Objekte gesetzt. Jeder Klasse, der zu erzeugenden Objekte, entspricht eine *ObjectMaker*-Klasse. So wird ein Objekt der Klasse *SH_Stuetze* durch ein Objekt der Klasse *SH_StuetzeMaker* erzeugt.

In diesen *ObjectMaker*-Klassen werden die virtuellen Funktionen und die Algorithmen, die für das Konstruieren und Setzen eines Objektes notwendig sind, realisiert. Außerdem werden in dieser Klasse die interessanten Schnittstellen überschrieben, welche für Maus-Aktionen wie z.B. *OnMouseMove( ), OnLButtonDown( )*, *OnRButtonDown( )*, *OnRButtonDown( )* etc., oder Tastaturtasten wie z.B. *OnChar( )* bei der Erzeugung eines Objektes wichtig sind.

⇒ **Direktor** (*SH_Workstack*)
Dieses Objekt wird durch die Klasse *SH_WorkStack* beschrieben. Er leitet die Benutzerreaktionen während der Erzeugung eines Objektes an das dazugehörige *ObjectMaker*-Objekt weiter. Dabei dienen *die virtuellen Funktionen* der Basisklasse *SH_ObjectMaker* als Schnittstelle.

⇒ **Produkt** (Linienlager, Stütze, ... etc.)
Ein *Produkt* ist das zu erzeugende Objekt während der Erzeugung. Im konkreten Fall dieser Dissertation ist dies ein Linienlager, eine Stütze, eine Platte, eine Aussparung, ein Unterzug etc. Dabei ist für jede Konstruktionsmethode des Objekts ein ObjektMaker-Klasse gedacht, die aus der Basisklasse *SH_ObjektMaker* abgeleitet ist. Für die Eingabe des Linienlagers durch seinen Anfangs- und Endpunkt ist z.B. ein ObjektMaker zuständig, der sich von dem ObjektMaker bei der Eigabe eines Polygonzuges von Linienlager unterscheidet. Erst wenn das Objekt fertig konstruiert wird, wird in die Datenbasis das System eingetragen.

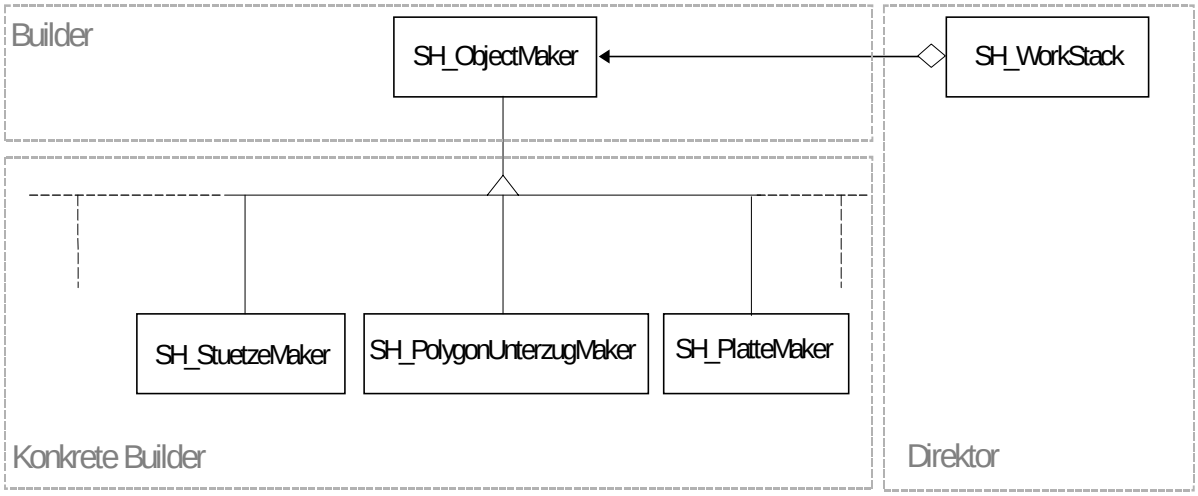


Abbildung 6.10: Konzept des Builders bei der Konstruktion und Erzeugung von Objekten

Zusammenwirken der Komponenten im Builder-Modell

- Der Benutzer erzeugt das Direktor-Objekt und konfiguriert es mit dem ausgewählten Builder-Objekt.
- Der Direktor benachrichtigt den Builder wann ein Teil des Produktes erzeugt werden soll.
- Der Builder leitet die Nachrichten des Direktors weiter und fügt Teile dem Produkt zu.
- Der User empfängt über das Dokument *SH_Document* das Produkt vom Builder.
- Das Dokument *SH_Document* fügt das fertig konstruierte Objekt in die Objektliste ein.
- Das Dokument leitet die Nachricht an das fertig konstruierte Objekt weiter und teilt ihm mit, in welchen Ansichtenmanager *SH_ViewObjManager* seine Darstellungsobjekte einfügen soll.
- Das fertig konstruierte Objekt sendet eine Nachricht an den zuständigen Ansichtenmanager *SH_ViewObjManager* (2D-View, 3D-View, Text-View) und übergibt ihm seine Darstellungsobjekte.
- Der Darstellungsobjektmanager *Manager* fügt die Darstellungsobjekte in die entsprechende Darstellungsobjektliste ein.

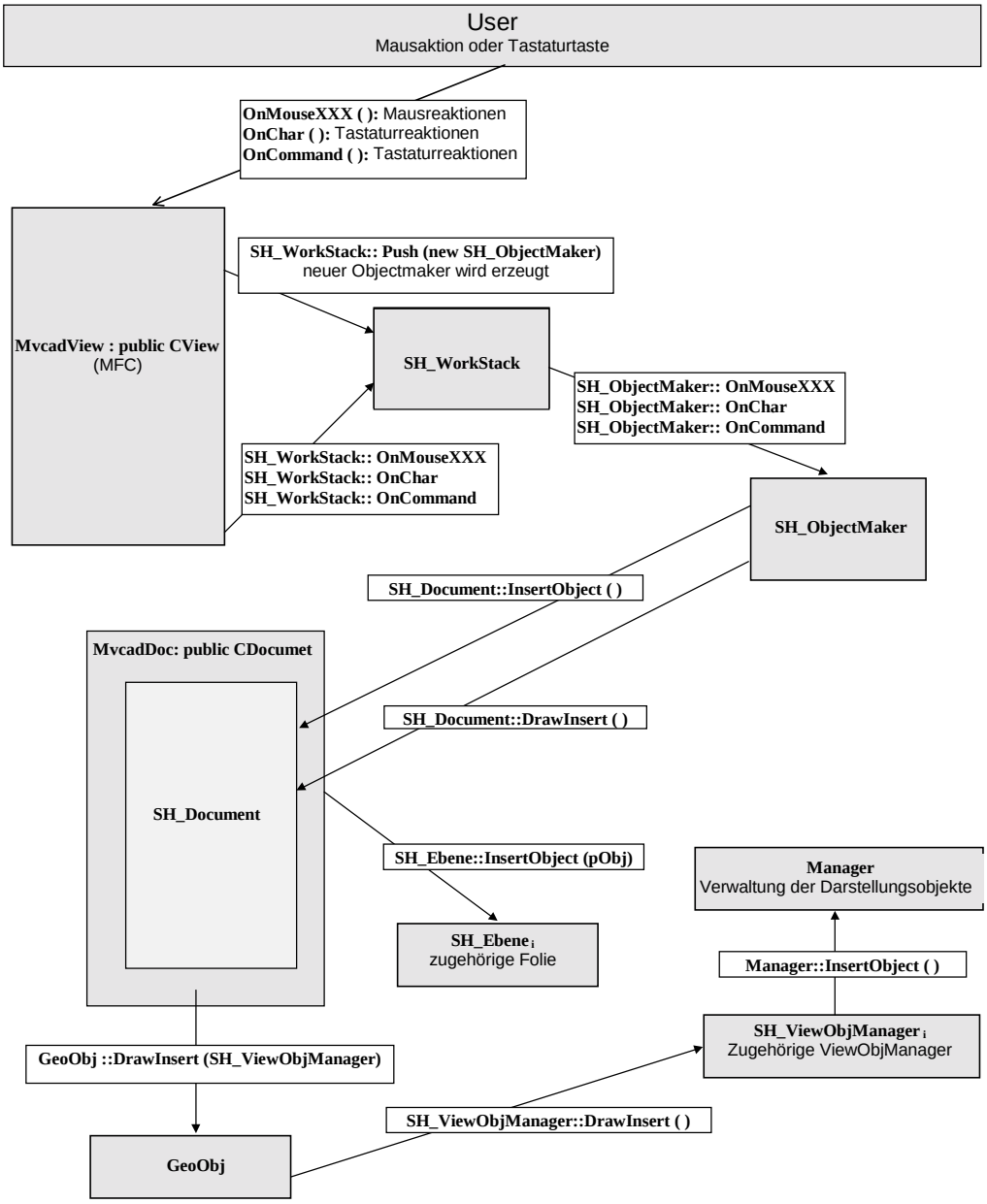


Abbildung 6.11: Absenden von Nachrichten (Message passing) zum Erzeugen eines Objektes

Das Interaktionsdiagramm (Abbildung 6.11) zeigt das Versenden und Empfangen von Nachrichten (Message Passing) zwischen den Objekten bei der Eingabesequenz, um ein Linienlager zu erzeugen.

6.5.3 Beispiel Code

An folgenden Auszügen des Programmes soll die Builder-Methode näher erläutert werden.

Der „Builder“ definiert für Mausreaktionen des Benutzers folgende Schnittstelle:

```
class SH_ObjectMaker
{
        ...
        ...
        public:
                //  Reaktion auf Maus-Bewegung
                virtual void OnMouseMove ( UINT nFlags, CPoint pkt,
                                SH_ViewMap* pViewMap, CView* pView );

                //  Reaktion auf Drücken der linken Maustaste
                virtual SH_MAKER_RESULT OnLButtonDown ( UINT nFlags,
                                CPoint pkt, SH_ViewMap* pViewMap, CView* pView );

                //  Reaktion auf Drücken der rechten Maustaste
                virtual SH_MAKER_RESULT OnRButtonDown ( UINT nFlags,
                                CPoint pkt, SH_ViewMap* pViewMap, CView* pView );
        ...
        ...
};
```

Die Schnittstelle fängt u. a. folgende Mausreaktionen ab :

Aktion	**Funktion**
Mausbewegung	*OnMouseMove ( )*
Drücken der linken Maustaste	*OnLButtonDown ( )*
Drücken der rechten Maustaste	*OnRButtonDown ( )*
. . . .	. . . .

Diese Funktionen werden nicht als *pure virtuelle Funktionen* definiert, damit abgeleitete Klassen nur solche Funktionen überschreiben müssen, die für sie interessant sind.

So wird die Funktion *OnMouseMove( )* in abgeleiteten Klassen nicht überschrieben, sondern dient nur zur Darstellung der Mausposition in der Statuszeile.

Eine wichtige Funktion der Schnittstelle ist *OnLButtonDown( ).* Durch sie werden die Punkte, die ein Objekt zum Teil beschreiben, gesetzt. So wird z.B. die Länge einer Linie intern durch zwei Punkte beschrieben. Also ist *OnLButtonDown( )* ein sehr wichtiges Werkzeug zum Setzen eines Objektes.

Im Gegensatz zur Linie, dessen Länge durch zwei Punkte beschrieben wird, wird die Geometrie eines Polygons durch mehrere Punkte beschrieben. Eine Linie ist also erzeugt, wenn zwei Punkte gesetzt sind. Im Gegensatz wird die Erzeugung eines Polygons erst dann abgeschlossen, wenn dies explizit durch eine Benutzer-Aktion gewünscht wird.
Aufgrund dieser Kriterien ist die Funktionalität von *OnLButtonDown( )* in den konkreten Buildern für eine Linie und ein Polygon anders.

6.5.3.1 Erzeugung einer Linie

Ein Obkjekt der Klasse *SH_LinieMaker* ist zuständig für das Aufbauen, Erzeugen einer Linie.

Beim konkreten Builder des Polygons wird durch die Funktion *OnLButtonDown( )* im wesentlichen nur ein Punkt in die Punktliste des Polygons aufgenommen.

```
SH_MAKER_RESULT SH_LinieMaker :: OnLButtonDown (
        UINT nFlags  ,                  //  für gleichzeitigen Tastendruck
        CPoint pkt,                     //  Maus-Position in Pixel-Koordinaten
        SH_ViewMap* pViewMap,           //  Viewmap-Objekt
        CView* view   )                 //  betroffenes View-Objekt
{
        // Funktion der Basisklasse aufrufen
        SH_ObjectMaker :: OnLButtonDown( nFlags, pkt, pViewMap, view );
        ...
        if ( !m_nStatus++ )    // erster Punkt
        {
                // Ansatzpunkt für Gummilinie und Cursortyp setzen
                m_pktLast = m_pktAkt;
                m_CursorTyp = GUMMI_LINIE_CURSOR;  // Kursor-Typ einstellen

                // Cursor neu zeichnen
                UpdateAllViews ( VIEW_DRAW_CURSOR );
        }
        else     // falls zweiter Punkt
        {
                m_pLinie->SetP1 (m_pktLast);        // erster Punkt der Linie setzen
                m_pLinie->SetP2 (m_pktAkt);         // zweiter Punkt der Linie setzen
                return  LinieBeenden ( pViewMap )  // Eingabe beenden;
        }
        return  GO_ON;                              // Kontrolle weiter geben
}
```

Über die Variable *m_nStatus* wird unterschieden, ob es sich um den Anfangs- oder Endpunkt der Linie handelt. Handelt es sich um den Endpunkt, wird die Geometrie der Linie mit den Funktionen *SetP1( )* und *SetP2( )* der Klasse *Linie* gesetzt, und schließlich über die Funktion *LinieBeenden( )* in die Datenstruktur des Programmes eingefügt, die Variable m_nStatus wird wieder auf Null gesetzt, um weitere Sequenzen für die Linieneingabe anzufangen. Wird die Funktion *OnLButtonDown( )* erneut aufgerufen, erzeugt diese den Anfangspunkt einer neuen Linie usw.

Rückgabewerte der ObjectMaker-Funktionen ist eine Variable, die zur Aufzählung dient und in der Programmiersprache C++ von Typ *enum* wie folgt deklariert werden kann:

```
enum SH_MAKER_RESULT
{
        DONE,               // Maker-Objekt ist fertig:
        GO_ON,              // Maker-Objekt wird fortgesetzt
        BACK_POINT          // Maker-Objekt ist fertig und gibt Punkt zurück
};
```

6.5.3.2 Erzeugung eines Polygonzugs von Punkte

Beim konkreten Builder des Polygons wird durch die Funktion *OnLButtonDown( )* im wesentlichen nur ein Punkt in die Punktliste des Polygons aufgenommen.

```
SH_MAKER_RESULT SH_PolygonMaker :: OnLButtonDown (
        UINT nFlags,                    //  Für gleichzeitigen Tastendruck
        CPoint pkt,                     //  Maus-Position in Pixel-Koordinaten
        SH_ViewMap* pViewMap,           //  Viewmap-Objekt
        CView* view  )                  //  betroffenes View-Objekt
{
        // Eingabe eines Punkts
        ...
        ...
        // Punkt an das Polygon hängen
        m_pPolygon->AppendPunkt ( m_pktAkt );
        ...
        ...
        // weitere Punkte
        return  GO_ON;
}
```

Wie man aus diesem Beispiel sehen kann, haben die Funktionen der Schnittstelle bei den konkreten Buildern eine unterschiedliche Funktionalität.

Die Builder-Methode kapselt den Prozeß, wie ein Objekt erzeugt wird. Im konkreten Fall ist für den Anwender der Schnittstelle von außen nur ersichtlich, daß mit der Funktion *OnLButtonDown( )* ein Punkt gesetzt wird. Wie dieser beim Erzeugen eines Objektes intern benutzt wird, hängt dann vom konkreten Builder ab.

Ein Interaktionsdiagramm zeigt das Zusammenspiel und Nachrichtensendung im Laufe der Zeit zwischen den in einem Vorgang (Szenario) beteiligten Objekten. Die Abbildung 6.12 zeigt das Interaktionsdiagramm einer Eingabesequenz beim Drücken der linken Maustaste (2. Punkt) während der Erzeugung eines Linienlagere-Objekts durch die Eingabe des Anfangs- und Endpunkts.

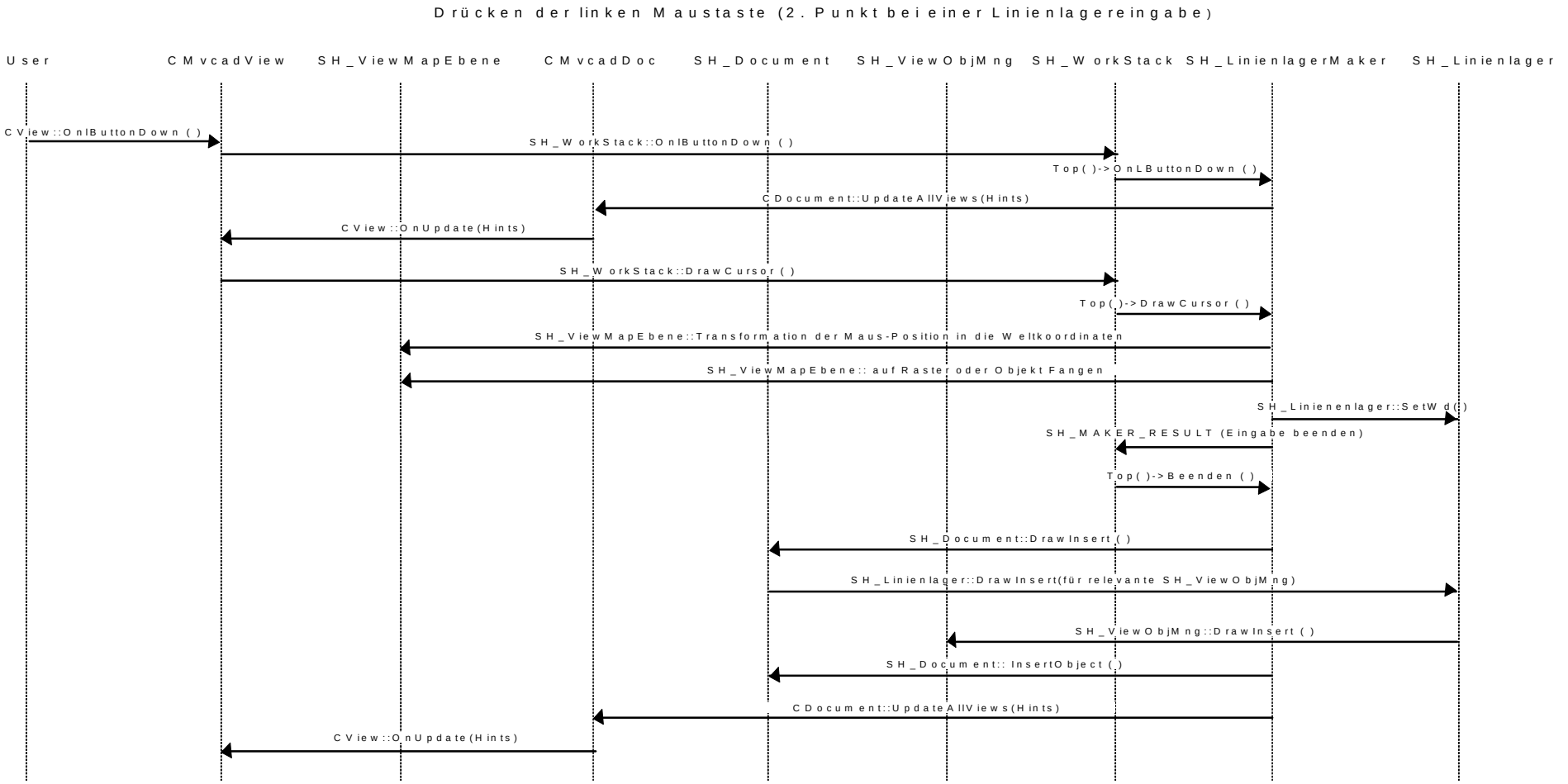


Abbildung 6.12: Interaktionsdiagramm für eine Eingabesequenz (Drücken der linken Maustaste, 2. Punkt bei der Linienlagereingabe)

6.6 Datenaustausch

Informationsaustausch erfolgt auf unterschiedlichen Ebenen:

- Nutzung standardisierter Datenschnittstellen
- Nutzung der rechnerinternen Datenstrukturen
- Realisierung auf der internen Ebene
- Realisierung auf der externen Ebene

Es besteht Bedarf an Informationsaustausch, grafischen und bauspezifischen Daten zwischen allen baubeteiligten Gruppen in allen Phasen der Planung von Bauwerken.

1. Die erste Gruppe sind die Planungsingenieure und Architekten mit verschiedenen perspektivischen Ansichten (Design-Phase).
2. Die zweite Gruppe sind die Bauingenieure als Tragwerksplaner mit Schal- und Bewehrungs-planung.
3. Fachplanung, wie z.B. Klima, Heizung, Lüftung, um die benötigte Aussparung einplanen zu können.
4. Die dritte Gruppe ist der Auftraggeber der Planer, der aus seiner Seite gewisse Änderungen, z.B. aufgrund der Kosten, vornehmen würde.
5. Die vierte Gruppe sind die ausführenden Firmen.

Konvertierer, auch Pre- und Postprozessoren genannt, sind traditionellerweise zuständig für das Lesen und Schreiben von fremdem Austauschformat (nicht das eigene Systemformat) und dienen zur Konvertierung von Datenstrukturen. Für jedes Austauschformat (DXF, STEP, etc.) soll ein Pre- und Postprozessor programmiert werden, dadurch erhöht sich der Programmieraufwand potentiell mit der Anzahl der Schnittstellen zu anderem Austauschformat.

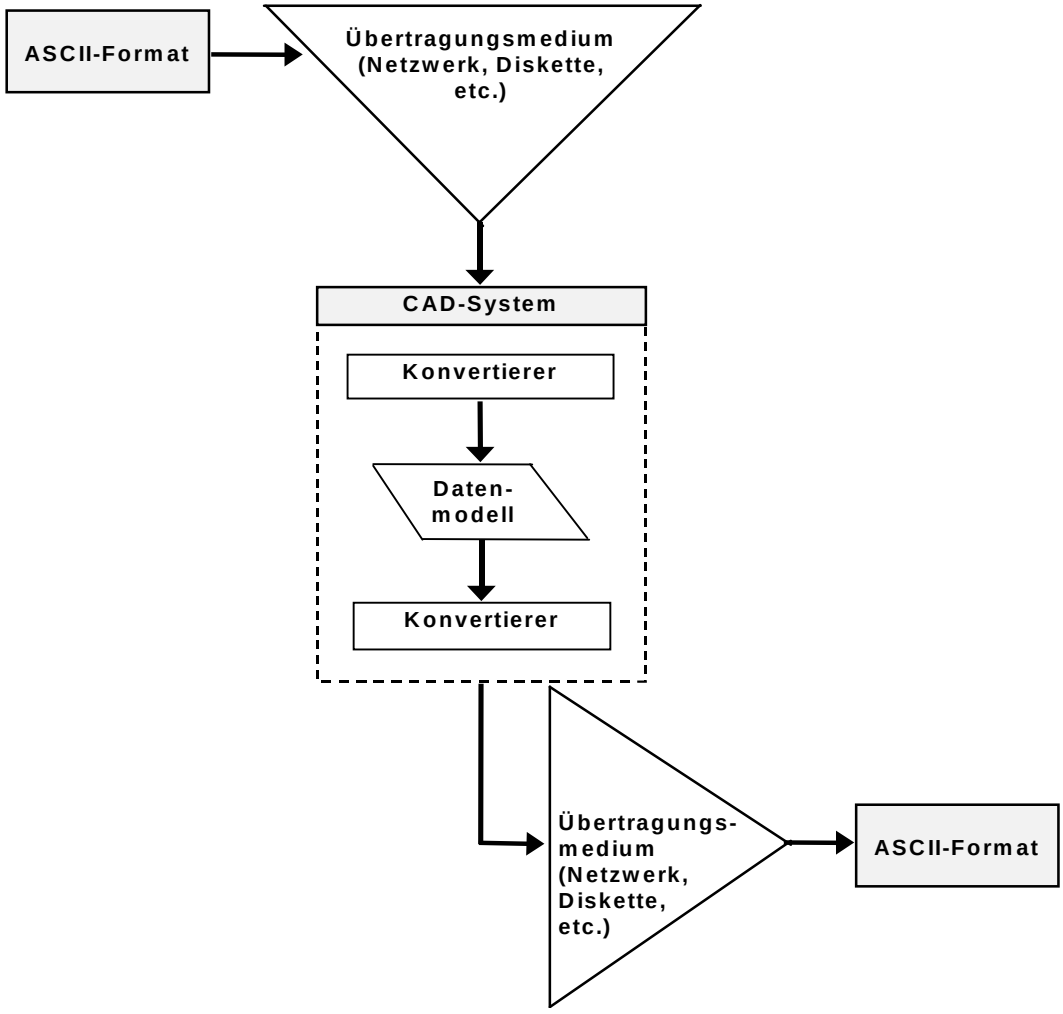


Abbildung 6.13: Konvertierung von Datenstrukturen von und nach ASCII_Format

Im Rahmen dieser Arbeit ist nur eine Austauschschnittstelle implementiert. Es besteht eine Austauschmöglichkeit über die sogenannten Positionsdateien (*.pos) mit dem Systems MicroFe. Für das Import- und Export steht ein besonderes Format zur Verfügung.

Da MicroFe ein Programm zur Tragwerksberechnung auf Basis der Finiten-Elemente-Methode ist, können auch nur solche Objekte Importiert bzw. exportiert werden, die vom System MicroFe unterstützt werden, die Bauteilpositionen genannt sind [76]. So werden nur Objekte folgender Klassen beim Importieren und Exportieren berücksichtigt :

⇒ Tragwerksobjekte:

- *SH_Aussparung*
- *SH_Bettung*
- *SH_Linienlager*
- *SH_LinLagPoly*
- *SH_Platte*
- *SH_Rundstuetze*
- *SH_Stuetze*
- *SH_Unterzug*
- *SH_UnterzugPoly*

⇒ Belastungsobjekte:

- *SH_Punktlast*
- *SH_Linienlast*
- *SH_Flaechenlast*

Ferner bezieht sich der Import und Export von Daten aus dem System MicroFe nur auf die Plattenberechnung als informative Auskunft über die Bemessungsmethoden, die im Bauteilobjekt gespeichert werden können. Im folgenden soll die Vorgehensweise beim Export und Import kurz erläutert werden.

6.6.1 Export von Daten

Wie beim Import ist auch beim Export vor allem eine virtuelle Funktion

```
GeoObj::virtual void WritePos(ostream& strm)
```

der Basis Klasse *GeoObj* zuständig für das Speichern im Formate (*.pos). Jedes Objekt das gespeichert werden soll, gibt seine Dateninformationen in einen Datei-Stream über die Funktion *WritePos(ostream& strm),* die in jeder Tragwerksklasse und der für System MicroFe relevante Elemente konkret implementiert.

Die Organisation des Schreibens dieses Formates geschieht hier auch wieder durch ein Objekt der Klasse *SH_Archiviere*. Dieses Objekt wird in der Dokumentklasse *SH_Document* erzeugt.

Ferner muß hier angemerkt werden, daß jeweils nur die Daten einer Ebene exportiert werden können. Dies begründet sich darauf, das beim System MicroFe auch jeweils nur z.B. Platten in einer Ebene berechnet werden können.

Beim Export von MicroFe wird durch das Dokument die jeweils aktuelle Ebene durch Aufruf der Funktion *SH_Ebene::SchreibePos(SH_Archiviere& arch)* mit dem entsprechenden Archivierungsobjekt als Übergabeparameter angesprochen.

Die Ebene gibt nun seine Tragwerksobjekte über die Funktion *WritePos(GeoObj* pObject)* der Klasse *SH_Archiviere* in den Ausgabe-Stream. Das Objekt der Klasse *SH_Archiviere* wiederum ruft die virtuelle Funktion *WritePos(ostream& strm* der Basisklasse *GeoObj* auf. welche dann konkret die Daten eines Tragwerksobjektes in den Ausgabe-Stream gibt. den Ablauf der Aktionen ist in der Abbildung 6.14 mit Hilfe eines Interaktionsdiagramms zu verfolgen.

Die virtuelle Funktion *WritePos(ostream& strm)* wird nur bei solchen Klassen überschrieben, die auch sinngemäß durch das System MicroFe unterstützt werden.

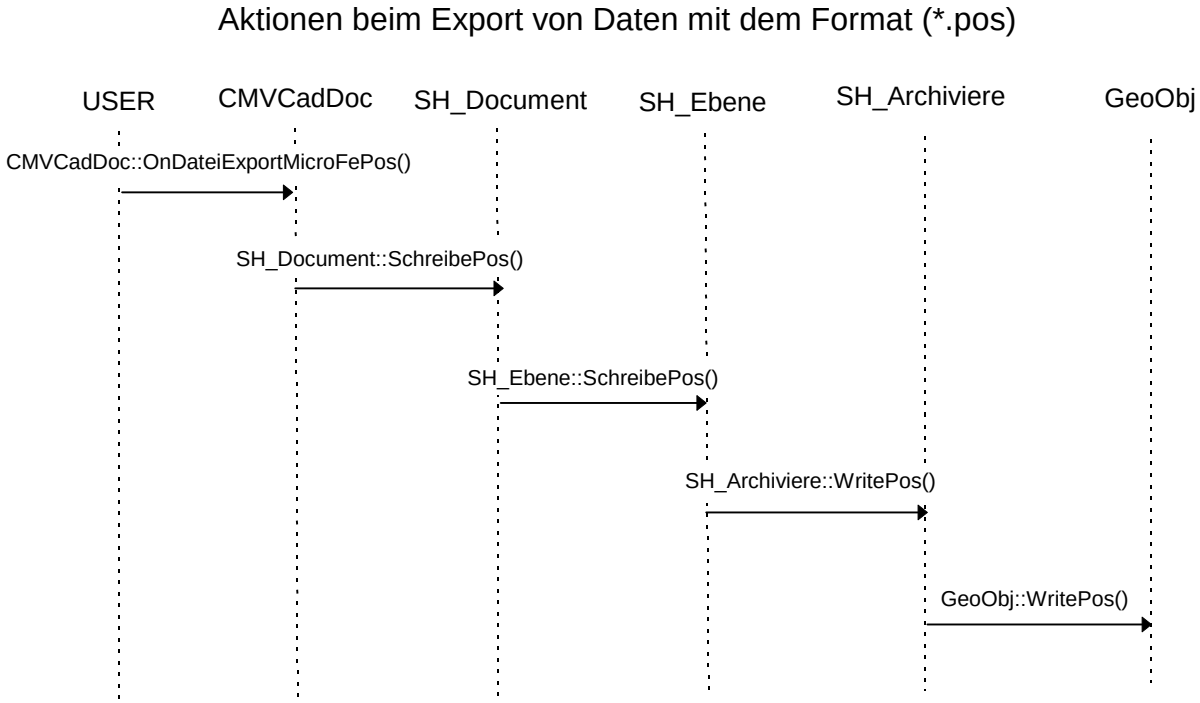


Abbildung 6.14: Interaktionsdiagramm für den Export von Daten im Format (*.pos)

6.6.2 Import von Daten

Im Gegensatz zum Export werden beim Import über das Pos-Format des Systems MicroFe keine Funktionen der jeweiligen Tragwerksobjekte angesprochen, sondern das Einlesen geschieht über die Importfunktion *ImportPos(CString& strDateiname)* der Dokumentklasse *SH_Document*.

Dies begründet sich darauf, daß die MicroFe - Daten variieren können und zusätzlich noch Daten bestehen, die von den eigenen Tragwerksobjekten nicht unterstützt werden (z.B. Bemessungsdaten).

Es besteht also die Notwendigkeit, die Daten sequentiell einzulesen. Dies geschieht über ein Objekt der Klasse *SH_DateiIO.*

Die Methoden dieser Klasse besitzen die Möglichkeit in einer Datei Schlagwörter zu suchen, und Daten hinter diesen Schlagwörtern einzulesen, wie z.B. die folgenden Methoden:

```
...
CString GetString_IO ( LPCTSTR lpszSection, LPCTSTR lpszEntry, char brak='\n' );
int GetInt_IO ( LPCTSTR lpszSection, LPCTSTR lpszEntry );
double GetDouble_IO ( LPCTSTR lpszSection, LPCTSTR lpszEntry );
...
```

Ein Objekt der Klasse *SH_DateiIO* ist mit einem Datei-Stream verknüpft. In diesem Stream werden durch oben genannte Methoden Daten in der Sektion *lpszSection* nach dem Eintrag *lpszEntry* eingelesen. Dies basiert auf dem Aufbau einer Ini-Datei.

Hier ist ein Auszug aus einer Initialisierungsdatei für Material (Material.ini):

```
...
[Material]
Anz= 3
Baustoffe = Beton, Stahl, Holz;

[Beton]
Anz= 2
Symb = B25, B35;

B25:
EMod= 30000.0
GMod= 16000.0
Rho= 2.5
Nue= 0.2
...
```

Um hier z. B. den Wert des E-Moduls für eine vorhandene Betonklasse B25 einzulesen, muß man diesen Wert über die Sektion B25 den Eintrag Emod ansprechen. Die Materialdaten werden dann in einer Dialogbox angezeigt, um einem Objekt zu zuordnen. Es besteht auch die Möglichkeit, über eine besondere Dialogbox auf die Materialdatenbank zu zugreifen, um ein Material einzufügen, zu löschen oder die Materialdaten zu editieren. Auf diese Weise werden auch aus einer Positiondatei des Systems MicroFe die Daten sequentiell eingelesen.

Am Anfang einer Positionsdatei stehen die Positionsnamen, die dann als Sektion zum Einlesen der Daten verwendet werden (siehe die Positionsdatei am Ende dieses Abschnitts).

```
...
[Positionen]
  Pos-Liste = PL-1 RB-1 ST-1 ST-2
...
```

Der Typ eines Objektes steht dann nach dem Eintrag „Typ = “ (z.B. Typ = Plattenbereich) in der jeweiligen Position - Sektion.

Durch diese Typangabe werden dann die jeweils einzulesenden Objekte identifiziert und die benötigten Daten für das jeweilige Objekt werden dann aus der Positions - Sektion eingelesen. Danach wird ein Objekt mit den eingelesenen Daten erzeugt und in die Datenstruktur des Dokumentes eingehängt.

Ein Plattenbereich mit veränderlicher Dicke wird z. B. mit dem System MicroFe durch die Eingabe von drei Punkten definiert (Abbildung 6.15). Die einzelnen Positionen (ein Plattenbereich mit veränderlicher Dicke, eine Rundstütze, eine recheckige Stütze und ein Linienlager) werden vom System MicroFe in eine positiondatei exportiert, die vom CAD-System MvCad gelesen (importiert) werden kann. Diese einzelnen Positionen werden dann im CAD-System MvCad wieder hergestellt.

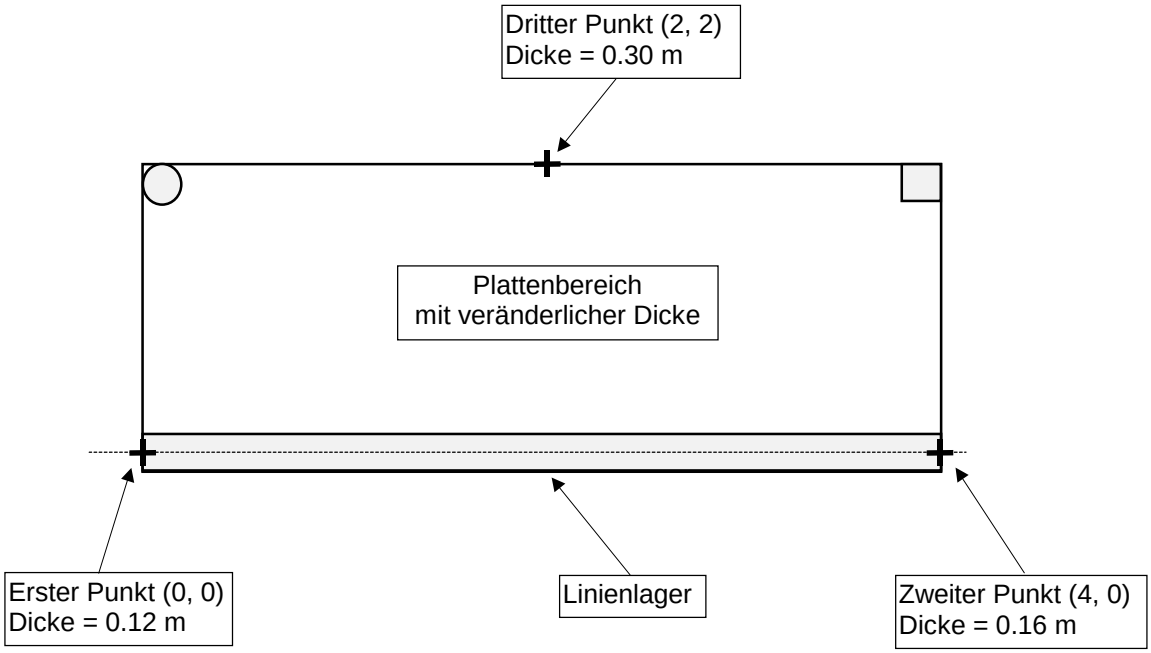


Abbildung 6.15: Platte mit veränderlicher Dicke, (Eingabe mit System MicroFE)

Im folgenden sind die Positionen dargestellt wie sie in der Positionsdatei (*.pos) angegeben sind:

```
[Positionen]
 Pos-Liste = PL-1 RB-1 ST-1 ST-2
 PL-1 = {
   Typ = Plattenbereich
   Geometrie = {
     Typ = Polygon
```

```
  X = 0 4 4 0 0
  Y = 0 0 2 2 0
 }
 Material = {
  Typ = isotrope Platte
  Dicke = {
   Typ = veraenderlich
   X = 0 4 2
   Y = 0 0 2
   d = 0.12 0.2 0.3
  }
  E-Mod = 3e+007
  rho = 2.5
  nue = 0.2
  Drillfakt = 0
 }
 Belastung = {
  g = -5
  p = -3.5
  Lasttyp = PL-1
 }
 Bemessung = {
  Typ = KH-Verfahren
  Bn = 25
  BSt = 4
  BStb = 4
  Staffelung = 0
  Fertigteil = 0
  h'xo = 2
  h'yo = 3
  h'xu = 2
  h'yu = 3
  asgxo = 0
  asgyo = 0
  asgxu = 0
  asgyu = 0
  w = 0
 }
 Lastfelder = {
  Liste = FELD-1
  FELD-1 = {
   Typ = Polygon
   X = 0 4 4 0 0
   Y = 0 0 2 2 0
   Lastfall = {
    Typ = automatisch
    Nr = 2
   }
  }
 }
}
```

```
ST-1 = {
  Typ = Stuetze
  Material = {
    Typ = Rundstuetze
    Durchmesser = 0.3
  }
  Geometrie = {
    Typ = Punkt
    X = 0.15
    Y = 1.85
    Alpha = 0
    Verfeinerung = 1
  }
 Art = Druckfeder
  Auflagerung = {
    Fg = TransT, RotR, RotS
    Steif = 706858, 11928.2, 11928.2
    l = 3
    E-Mod = 3e+007
  }
  Bemessung = {
    h'm = 4
    d0 = 20
    Bn = 25
    BSt = 4
    BStb = 4
    asm = 8
    Korrekturfaktor = 0
    tau_erhoehung = 0
    Staffelung = 0
  }
}
ST-2 = {
  Typ = Stuetze
  Material = {
    Typ = Rechteckstuetze
    b = 0.3
    d = 0.3
  }
  Geometrie = {
    Typ = Punkt
    X = 3.85
    Y = 1.85
    Alpha = 0
    Verfeinerung = 1
  }
  Art = Druckfeder
  Auflagerung = {
    Fg = TransT, RotR, RotS
    Steif = 900000, 20250, 20250
    l = 3
```

```
    E-Mod = 3e+007
   }
   Bemessung = {
    h'm = 4
    d0 = 20
    Bn = 25
    BSt = 4
    BStb = 4
    asm = 8
    Korrekturfaktor = 0
    tau_erhoehung = 0
    Staffelung = 0
   }
  }
  RB-1 = {
   Typ = Linienlager
   Art = Druckfeder
   Fg = TransT
   Geometrie = {
    Typ = Linienzug
    X = 0 4
    Y = 0 0
   }
   Material = {
    Typ = Linienlager
    d = 0.3
    h = 3
    E-Mod = 3e+007
    Steifigkeiten = 3e+006
   }
  }
[Belastung]
 Last-Liste =
[Macros]
 Macro-Liste =
[Positions-Bezeichnung]
 Bezugsmasstab = 100.000000
 Textbreite[cm] = 0.250000
 Texthoehe[cm] = 0.350000
 Zeichen_im_Kreis = 1
 Pen = 3
 Polygon-Offset[m] = 0.100000
 Fuellmuster = 0.600000
```

6.7 Datensicherung und Speichermodell, neutrales Protokoll

Ein bekanntes Problem bei der Datensicherung von Objekte ist das Erhalten der Beziehungen zwischen diesen Objekten beim Speichern und sie beim Lesen wieder zu erkennen. Dabei soll jedes Objekt wissen, welche Komponente oder Objekte zu ihm gehören und auf welche es nur Verweis (Zeiger) hat.

Da die Verwaltung und Speicherung der Objektdaten ohne Anwendung der Datenbanktechniken weder durch das Einsetzen rationaler noch objektorientierter Datenbank geschieht, wird im Rahmen dieser Arbeit ein für das Datenmodel geeignetes Speicherkonzept überlegt und implementiert, das die Daten und deren Strukturierung sichert, und als lesbares neutrales Protokoll zum Weitergeben der Daten dient. Daher geschieht die Datensicherung über ein eigenes Textformat (ASCII-Format).

Die Ziele und die Aufgaben des für das System *MvCad* entwickelten Speicherkonzeptes sind im Einzelnen:

- Datenstruktur der Objekte in Textformat zu speichern
- Verweise jedes Objektes auf andere Objekte mitzuspeichern
- Die Speicherungsdatei nur einmal öffnen zu müssen, um alle Objekte mit ihren Verweisen zu lesen. Damit wird die Zeit und die mechanische Beanspruchung bei dem Zugriff auf das Speichermedium (Platte, Diskette, etc.) gering gehalten.

Hier soll nun auf die Organisation des Lesens und Schreibens der Daten eingegangen werden. Das eigene Format (**.shc*) *MVCad*-Format dient standardmäßig zum Sichern der Objektdaten des Datenmodells im Kapitel *6*. Die Objekte werden mit ihrer Datenstruktur und ihren Verweisen (Zeiger) gespeichert.

Das Einlesen und Schreiben dieses Formates geschieht über die virtuellen Funktionen, die in der Basisklasse *GeoObj* für diesen Zweck deklariert und in jeder davon abgeleiteten Klasse konkret gemäß ihrer Datenstruktur implementiert sind:

```
class GeoObj
{
        ...
        GeoObj::virtual UINT ReadFrom( istream& strm );
        GeoObj::virtual void WriteOn( ostream& strm );
        ...
};
```

Jedes Objekt, das gespeichert werden soll, gibt seine Dateninformationen in einen Datei-Stream über die Funktion *WriteOn(ostream& strm).* Der Datei-Stream stellt eine Verbindung zu einer Datei auf einem physikalischen Speichermedium dar. Die Daten eines Objektes werden aus einem Datei-Stream über die Funktion *ReadFrom(istream& strm)* eingelesen.

Die Organisation des Schreibens und Lesens geschieht über ein Objekt der Klasse *SH_Archiviere* . Dieses Objekt wird im Dokument *SH_Document* erzeugt, um die Daten des Tragwerksmodells, welche im Dokument in den Verwaltungslisten gespeichert sind, in Dateien zu speichern oder aus Dateien zu lesen. Dabei muß durch einen Übergabeparameter bestimmt werden, ob das erzeugte Objekt zum Lesen oder zum Speicher gedacht ist.

Die Motivation zur Erstellung der Klasse *SH_Archiviere* begründete sich darauf, daß die Möglichkeit bestehen sollte, ein Objekt mit der Information seiner Zeiger auf andere Objekte zu speichern, und diese Verknüpfungen beim Lesen wieder-herzustellen.

Dieses Konzept nachfolgend erläutert werden.

6.7.1 Speichern von Daten

Ein Objekt hat im Konzept dieses CAD-Systems eine eindeutige Nummer ID, die genau dieses Objekt identifiziert. Diese ID wird im Datei-Format gespeichert und dient beim Einlesen zur Identifizierung eines Objektes . So werden beim Speichern eines Objektes seine eigene ID und die ID's der Objekte gespeichert, auf die es einen Zeiger besitzt. Somit besteht die Möglichkeit die Zeiger-Verknüpfungen beim Lesen wieder herzustellen.

Auszug aus dem Dateiformat :

```
...
[Platte]
  Name= Pl-1
  ID = 41
  TypID = 7
   [Zeiger-Info =]
     Anz = 1
     Liste = 5
...
```

Dieser Auszug des Datei-Formates zeigt die Zeigerinformation eines Objektes der Klasse *SH_Platte*. Dabei hat die Platte einen Zeiger auf das Objekt mit der ID 5.

Neben dem Schlagwort *ID* steht die eigene *TypID* des Objektes. Das Schlagwort *TypID* dient zur internen Erkennung des Objekt-Typs, in diesem Falle ein Objekt der Klasse *SH_Platte*. Hierfür sind ID und *TypID* als Ganzzahl definiert.

Nach dem Eintrag *Zeiger-Info* stehen die Verknüpfungen zu anderen Objekten, dabei werden die ID's dieser Objekte aufgelistet. In diesem Fall hat die Platte einen Zeiger auf ein anderes Objekt, das die ID 5 besitzt und später in der Datei beschrieben ist.

6.7.2 Lesen von Daten

Beim Lesen kommt die eigentliche Aufgabe der Klasse *SH_Archiviere* zum Vorschein. Sie muß nun die Verknüpfung der Objekte wiederherstellen.

Das Kernstück ist dabei eine dynamische Liste *SH_List* von Objekten der Klasse *SH_LinkInfoMap*, die als neue Containerklasse in C++ wie gefolgt zu definieren sind:

typedef SH_List< SH_LinkInfoMap > SH_LINK_LIST;

Ein Objekt der Klasse *SH_LinkInfoMap* hängt also an jedem Knoten dieser Containerklasse (Abbildung 6.16), und beinhaltet Informationen über jedes gelesenes Objekt wie:

- einen Zeiger des zu verwaltenden Objektes
- ID dieses Objektes als ganze Zahl
- Objekttyp des Objektes (Platte, Unterzug, Linienlager etc.) als ganze Zahl
- Liste von ID's der Objekte, mit dem das zu verwaltende Objekt verknüpft ist

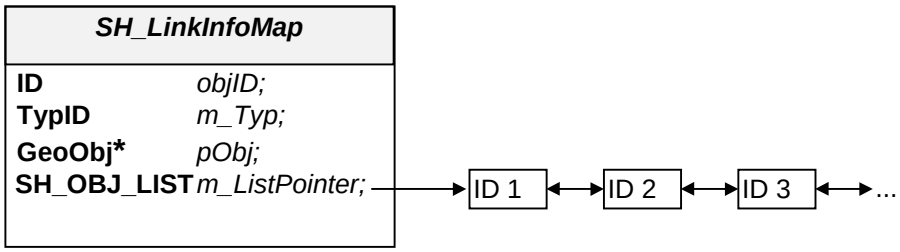


Abbildung 6.16: Ein Verknüpfungsobjekt zum Speichern der Beziehungen zwischen einem Objekt und seinen Referenzenobjekten

Die Implementierung der Klasse *SH_LinkInfoMap* ist C++ wie folgendes:

```
class SH_LinkInfoMap
{
private:
        // ID von des gelesenen Objekts (pObj)
        long objID;
        // Typ des Objektes (pObj)
        ObjTyp m_Typ;
        // Objekts des Knotens
        GeoObj* pObj;
```

```
public:
        // Liste mit den Pointern die pObj besitzt
        SH_OBJ_LIST m_ListPointer;

        // Konstruktoren
        SH_LinkInfoMap();
        SH_LinkInfoMap(GeoObj* pnObj);

        // Destruktor
        ~SH_LinkInfoMap();

        // Service-Funktionen zum Zugreiffen  auf die Daten
        void SetID(long ID){objID=ID;};
        long GetID(){return objID;};
        void SetTyp(ObjTyp nTyp){m_Typ=nTyp;};
        ObjTyp GetTyp(){return m_Typ;};
        ObjTyp FindTyp(GeoObj* pnObj);
        ...
};
```

Die oben schon erwähnte Liste *SH_LINK_LIST* mit Objekten der Klasse *SH_LinkInfoMap* wird beim Lesen aufgefüllt. Für jedes Objekt in der Datei wird ein Verknüpfungsobjekt der Klasse *SH_LinkInfoMap* erzeugt und in die Liste eingefügt.

Nachdem alle Objekte eingelesen und in diese Liste eingehängt sind, werden die Verknüpfungen hergestellt. Dies geschieht mit der Funktion *SH_Archiviere::LinkObjects()* unter Zuhilfenahme der virtuellen Funktion *GeoObj::UpdatePointer(GeoObj* pPointer)*.

Nun soll kurz die Arbeitsweise der Methode *LinkObjects( )* erläutert werden, die zur Klasse *SH_Archiviere* gehört (Abbildung 6.17).

Es wird über alle Knoten der *SH_LINK_LIST* gegangen, und für das Objekt *pObj* am jeweiligen Knoten über die Liste der Verknüpfungen *m_ListPointer* iteriert, die zur Herstellung der Beziehungen zwischen den Objekten zuständig ist.

Mit der Funktion *UpdatePointer(GeoObj* pPointer)* werden die Verknüpfungen auf Objektebene wieder hergestellt. Hier ist ein Auszug aus der Implementierung der Funktion *LinkObjects()*, die zur Herstellung der Beziehungen zwischen den Objekten zuständig ist:

```
void SH_Archiviere::LinkObjects()
{
        ...
        SH_POSITION Pos;
        ...
        for( Pos = m_LinkList.GetFirstPos();
                        Pos != NULL;
                        Pos = m_LinkList.GetNextPos(Pos) )
                {
```

```
                    SH_LinkInfoMap* pInfo =m_LinkList.GetAt(Pos);
                    GeoObj* pObj=pInfo->pObj;
                    ...
                    //Pointer-Update der Objekte
                    for( pos = pInfo->m_ListPointer.GetFirstPos();
                              pos != NULL;
                              pos = pInfo->m_ListPointer.GetNextPos(pos) )
                         {
                              long* pID= pInfo->m_ListPointer.GetAt(pos);
                              GeoObj* refPointer=GetObjWithID(*pID);
                              if( refPointer )
                              pObj->UpdatePointer(refPointer);
                         }
               }
     }
```

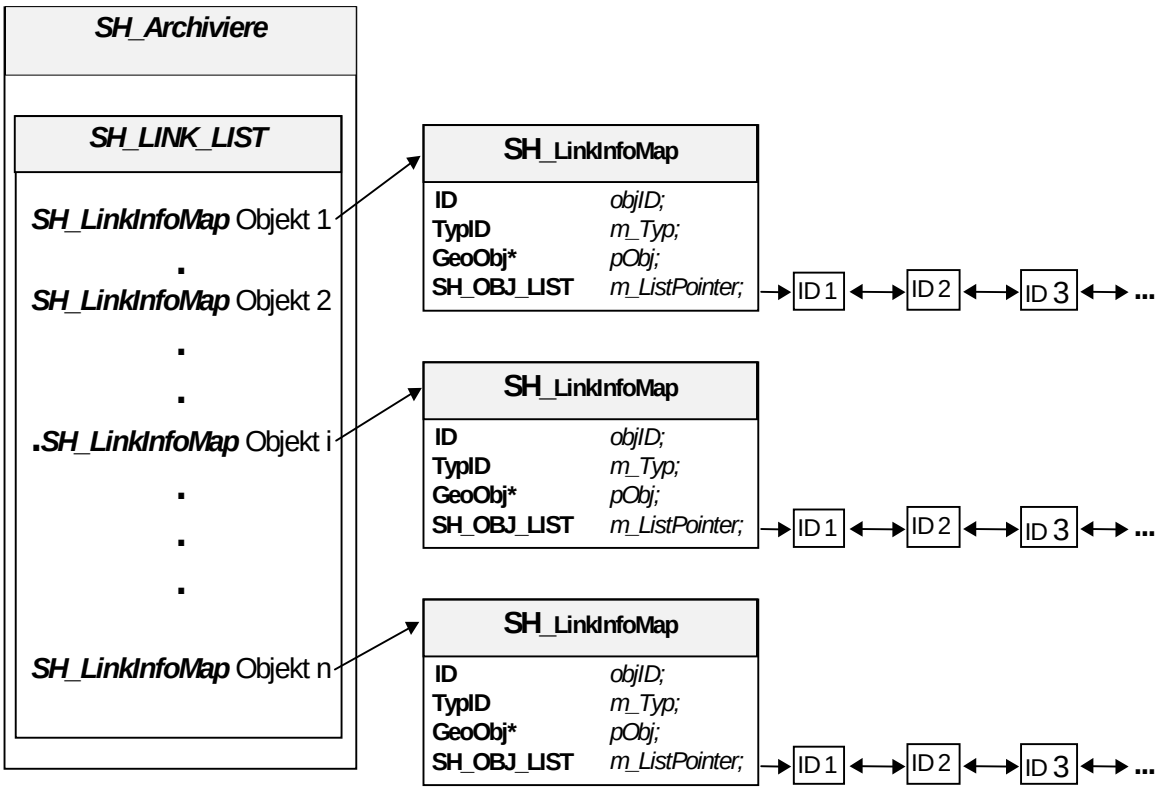


Abbildung 6.17: Eine Liste mit Informationsobjekten über ein Objekt und seine Referenzenobjekte

Nach der Erzeugung der Klasse *SH_Archiviere* werden die Input- und Output-Streams mit einer Datei geknüpft und zum Lesen oder Schreiben geöffnet. Es stehen verschiedene Methoden zum Öffnen und Schließen von Dateien, Schreiben und Lesen von Objekten und zur Identifizierung des Objekttyps bereit.

Im Folgenden ist ein Auszug aus der Deklaration der Klasse *SH_Archiviere*:

```
class SH_Archiviere
{
// Member-Daten:
public:
        ifstream m_streamIn;                //Input - Stream
        ofstream m_streamOut;               //Output - Stream
        SH_LINK_LIST m_LinkList; // Zeiger-Liste für die Linkinformationen

private:
        const char* m_szDateiName;          // Dateiname des zu öffnenden stream
        int m_nFlag;                        // Lesen (READ) oder Schreiben (WRITE)
        BOOL m_nStatus;                     // ist Stream offen oder geschlossen

// Member-Funktionen:
public:
        // Destuktor/Konstruktor
                SH_Archiviere(const char* DateiName, int flag);
                ~SH_Archiviere();
        // Öffnen/Schließen von Datenstream
                BOOL OpenStream();
                BOOL CloseStream();

        // Lesen/Schreiben von Objekten
        void WriteObject(GeoObj* pObject);  //Object im MVCad-Format schreiben
        GeoObj* ReadObject();               // Object aus MVCad-Format lesen
        void WritePos(GeoObj* pObject);     //Object im Pos-Format schreiben
        GeoObj* ReadPos();                  // Object aus Pos--Format lesen

//Service-Funktionen
        ObjTyp FindObjTyp(GeoObj* pnObj);
        GeoObj* GetObjWithTyp(ObjTyp Typ);
        GeoObj* GetObjWithID(long ID);
        void LinkObjects();

        // Knoten in die Linkinformationsliste einhängen
        void AddLinkInfo(GeoObj* pObj,long ID,ObjTyp typ);

        //dem aktuellen Infoknoten wird eine ID zugefuegt
        void AddLink(long ObjID);
        ...
};
```

7. Test und Integration des Datenmodells

7.1 Umfang der Realisierung des Datenmodells und des Designkonzepts

Die Realisierung des im Rahmen dieser Arbeit entwickelten Datenmodells für ein CAD-Systems zielt darauf hin, dieses Datenmodell und sein Designkonzept auf Machbarkeit, Richtigkeit, Effizienz und Performanz mit Hilfe eines richtigen lauffähigen CAD-Systems zu untersuchen. Dazu gehören CAD-Funktionalitäten und Konstruktionsmittel, die die Arbeit mit diesem System erleichtern und in keinem CAD-System fehlen dürfen [47] [39].

Es wurden für das CAD-System MvCad folgende Komponenten realisiert:

- Alle Bauteilobjekte im Datenmodell und deren Darstellungsobjekte, sowie die Belastungs- und Bewehrungsobjekte. Wegen der vielfältigen Querschnittsarten einiger Bauteile wurden nur ein oder zwei, meist im Gebäudebau vorkommende Querschnitte implementiert, z.B. eine rechteckige und eine runde Stütze sowie ein rechteckiger Unterzug.

- Die Schnittstelle zwischen den Daten und ihren Ansichten (Views) als Basisarchitektur für die objektorientierte interaktive Benutzeroberfläche (GUI). Dies wurde mit Hilfe der „Document-View“ Architektur der Microsoft Klassenbibliothek MFC für drei unterschiedliche Views 2D-, 3D- und Text-View prototypisch implementiert.

- Die Schnittstelle für die interaktive Erzeugung von Objekten durch den Benutzer. Dies wurde durch den Einsatz des sogenannten Builder-Konzepts, indem für jede Konstruktionsart oder Eingabesequenz eines Objekts ein Objekt-Maker (Builder) implementiert werden soll. Somit wird die Konstruktion von komplexen Objekten von der Repräsentation dieser Objekte getrennt. Für das Erzeugen und das Zusammenbauen eines Linienlager-Objektes wurden z. B. zwei Konstruktions-methoden, d. h. zwei Objekt-Maker implementiert. Die erste Methode ist durch die Eingabe des Anfangs- und Endpunkts und die zweite ist durch die Eingabe eines Polygonzugs von Punkten möglich.

- Es wurden nicht alle möglichen Konstruktionshilfen und Werkzeuge implementiert, wie Raster, Hilfslinien und Zeichenlineale. Ein rechteckiges Raster wurde prototypisch als eine Klasse implementiert. Für das Konstruieren und Positionieren der Bauobjekte besitzt jede grafische 2D-View ein Objekt dieser Rasterklasse als Hilfsmittel.

- Die CAD-Funktionalitäten wie das Zoomen, Identifizieren, Markieren und Fangen auf Punkte oder Linien von Raster oder Objekten in den grafischen Views .

- Für eine bessere 3D-Dimensionierung der Bauteile wurde das Erzeugen und das Konstruieren von Objekten in der 2D- und 3D-View gleichzeitig realisiert. Die perspektivische Sicht (View) ermöglicht die 3D-Sicht auf ein Bauobjekt während und nach dem Konstruktionsprozeß. Die 3D-Visualisierung ist dabei nicht implementiert. Dies kann durch Hidden-Line Algorithmus realisiert werden, das in der Literatur zur Visualisierung von 3D-Objekten zu finden ist.

- Die Speicherung von Objekten in einem Textformat (ASCII-Format). Dabei hat jedes Objekt seine eigene Speichermethode und alle Objekte werden vom Dokument über ein Verwaltungsobjekt zentral gespeichert oder eingelesen. Die Serialisierungsmethoden der MFC für Speicherung der MFC- oder der von ihr abgeleiteten Objekte wurden nicht benutzt.

- Eine Schnittstelle in Textformat (ASCII-Format) zum Datenaustausch von Datenstrukturen typischer Baupositionen im Ingenieurbau von Gebäuden mit dem Finite-Elemente-System MicroFe der Firma mb-Software. Dabei hat jedes Objekt sein eigene Methode für Import und Export der Objektdaten.

7.2 Test des CAD-Systems

Ziel des Testes ist die Untersuchung des Datenmodells, die CAD-Funktionalität und die Performanz des CAD-Systems MvCad.

- **Datenmodell und seine Objekte**

Zum Testen des entwickelten Datenmodells wurden Daten importiert, die mit dem System *MicroFe* der Firma *mb-Programme, Software im Bauwesen GmbH*, für die Tragwerksplanung generiert wurden. Hierfür wurden die Daten einer Tragstruktur mit dem System *MicroFe* erzeugt und in Positionsdateien in Form von Bauwerkspositionen gespeichert. Die einzelnen Positionen beschreiben die Tragstruktur, die verschiedene Bauteile (Plattenbereich, Stütze, Linienlager, Bettung, Unterzug, Aussparung, Belastung etc.) darstellen [76]. Sie enthalten die Informationen über Geometrie, Material und ggf. Bemessungseigenschaften oder Belastungsgrößen.

Anschließend wurden die Strukturdaten in das neu erstellte CAD-System MvCad importiert werden, Abbildung 7.1. Dabei werden Objekte generiert und dargestellt, die beliebige Bauwerkspositionen beschreiben. Die Tragstruktur wird in einer digitalen Datenbasis gemäß dem entwickelten Datenmodell aufgebaut, und die Fähigkeit des implementierten Datenmodells getestet. Die Tragstruktur wird in verschiedenen 2D- und 3D-Sichten dargestellt. Es wird die Möglichkeit untersucht, die Strukturdaten einzelner Bauteile in den verschiedenen Sichten (Views) direkt abzufragen, zu editieren, zu speichern, zu lesen oder zu exportieren.

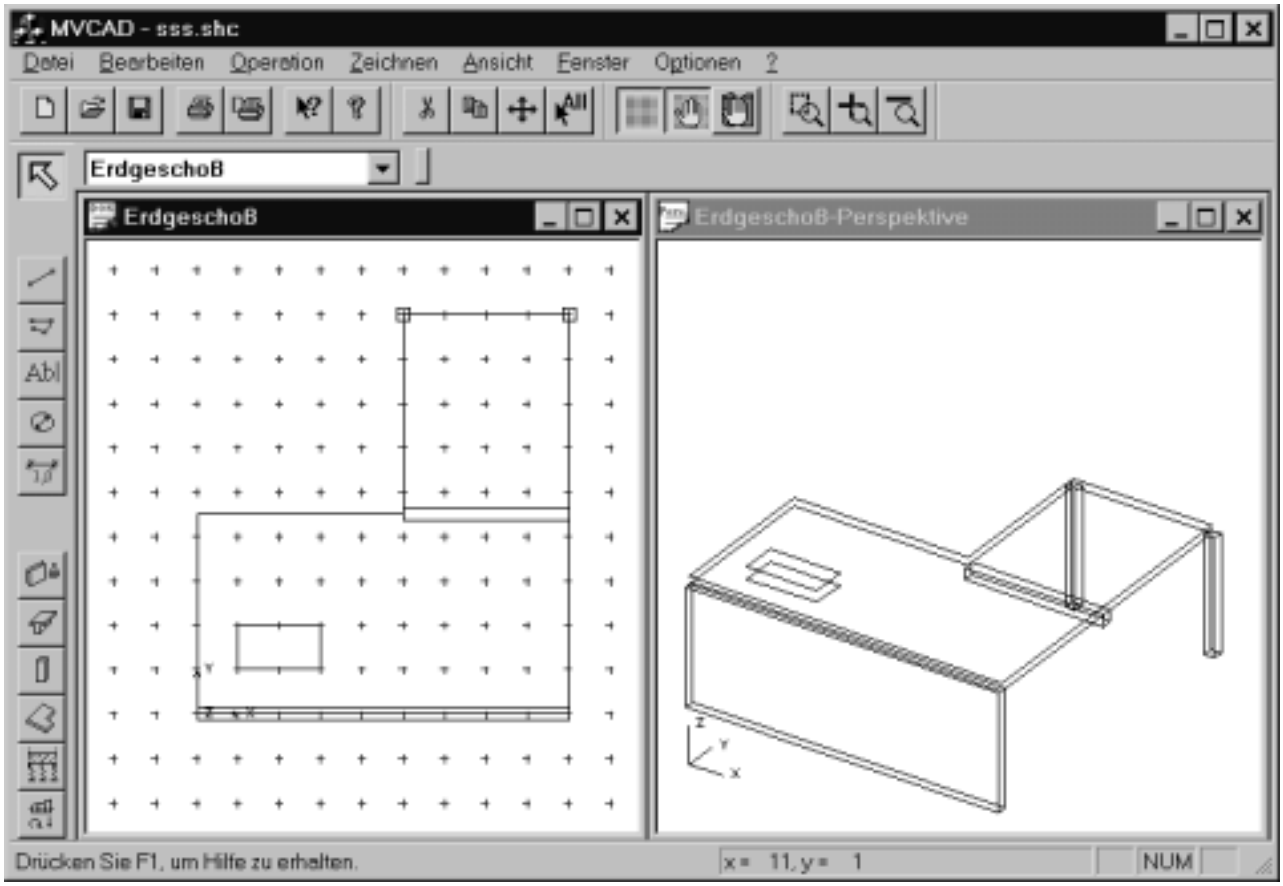


Abbildung 7.1: Hauptfenster des CAD-Systems MvCad mit der Tragstruktur einer Geschoßplatte dargestellt in der 2D- und 3D-View

- **CAD-Funktinalität**

⇒ **Identifizieren (Markieren) und Ändern (Editieren) von Objekten**
Es besteht die Möglichkeit, ein Objekt oder mehrere Objekte in allen Views durch Mausklick oder durch Aufziehen einer Box zu identifizieren (markieren) und anschließend auf sie verschiedene Operationen wie Kopieren, Verschieben und Löschen auszuführen. Außerdem steht eine interaktive Verarbeitung der Strukturdaten und der Darstellungseigenschaften (editieren) jedes Objekts mit Hilfe eines Dialogfensters (Dialogbox) zur Verfügung (Abbildung 7.4).

Ein Objekt kann in der 2D- oder 3D-View identifiziert werden, dies gibt die Möglichkeit, Objekte einer komplizierten Tragstruktur zu fangen und zu verarbeiten, die in der 2D-View nicht einfach zu fangen sind.

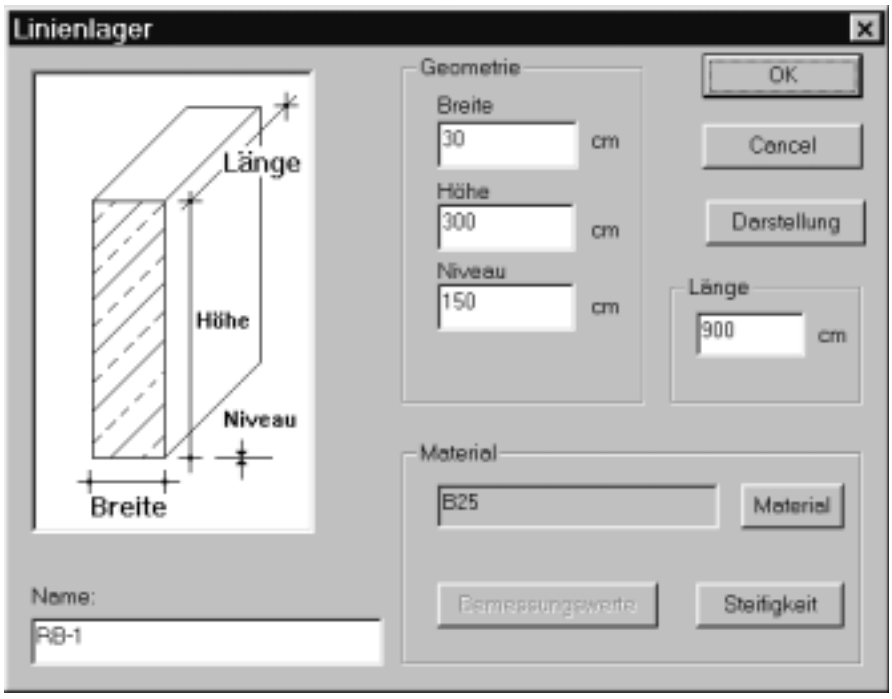


Abbildung 7.4: Ein Dialogbox zum Editieren der Daten eines Linienlager-Objekts Einsatz Einsatz

⇒ **Erzeugen, Konstruieren und setzen von Objekten**
Das Erzeugen neuer Objekte geschieht über eigene Klassen. Zu jedem Tragwerksobjekt gibt es ein Konstruktionsobjekt. Dieses Konstruktionsobjekt (Object-Maker) besitzt ein Objekt in seiner Struktur als Daten-Member vom Typ des zu setzenden Objekts. Nach Beendigung des Konstruktionsprozesses eines Objekts fügt der Objektmaker eine Kopie dieses Objekts in die Datenstruktur des Dokumentes und in den Manager der Sichtobjekte *SH_ViewObjManager* ein. Das Setzen eines neuen Objektes kann nur aus einer 2D-View geschehen. Die mit MFC erzeugte 2D-View entspricht der Klasse *MvcadView* (Abbildung 7.1).

Im folgenden ist die Erzeugung von Linienlager- und Belastungsobjekten erläutert:

1. Erzeugung eines Linienlager-Objekts:

   Durch Selektieren des entsprechenten Menü-Befehls oder beim Klicken eines Symbols der Toolbar (Abbildung 7.2) wird ein Konstruktionsobjekt (Object- Maker) gestartet. Dieses konstruiert ein Linienlager entweder mittels Anfangs- und Endpunkt oder mit Hilfe eines Polygonzugs.

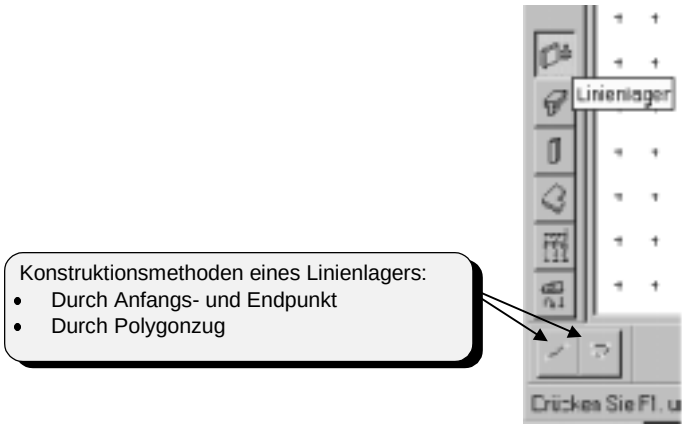


Abbildung 7.2: Erzeugung eines Bauteilobjekts - Linienlager - in der 2D-View mit verschiedenen Konstruktionsmethoden

2. Das Setzen der Belastungsobjekte geschieht pro Lastfall. Die ObjectMaker für die Belastungsobjekte werden alle aus dem Objectmaker für einen Lastfall aufgerufen. Das einzelne Setzen von Lasten ist nicht möglich, sondern eine Belastung ist immer einem Lastfall zugeordnet.

   Um die Eingabe der Belastung zu steuern, wird ein Objekt der Klasse *SH_LastfallMaker* erzeugt, das eine Dialogbox für die Eingabe verschiedener Belastungen auf dem Bildschirm einblendet. Es handelt sich um eine nicht-modale Dialogbox. Das heißt, diese Dialogbox bleibt solange aktiv, bis die Belastungseingabe beendet wird.

   Mit dieser Methode wird eine Last mit den Eigenschaften der temporären Last in die *BASELISTE* des Dokumentes eingetragen und die Darstellungsobjekte der Last werden in die zugehörigen *View-Objekt-Manager* eingetragen. Die temporäre Last wird auf NULL gesetzt und eine neue Last kann eingegeben werden.

Es können auch während des Konstruktionsprozesses eines Objekts mit Hilfe eines Konstruktionsobjekts „ObjektMacker“ andere Konstruktionsprozesse gestartet werden. Es kann zum Beispiel der Mittelpunkt eine Strecke während des Eingabeprozesses eines Linienlagers durch Starten eines entsprechenden ObjektMaker. Dies hilft dabei, unterschiedliche, komplizierte und mehrfach verschachtelte Eingabe- und Arbeitsprozesse besser zu gestalten.

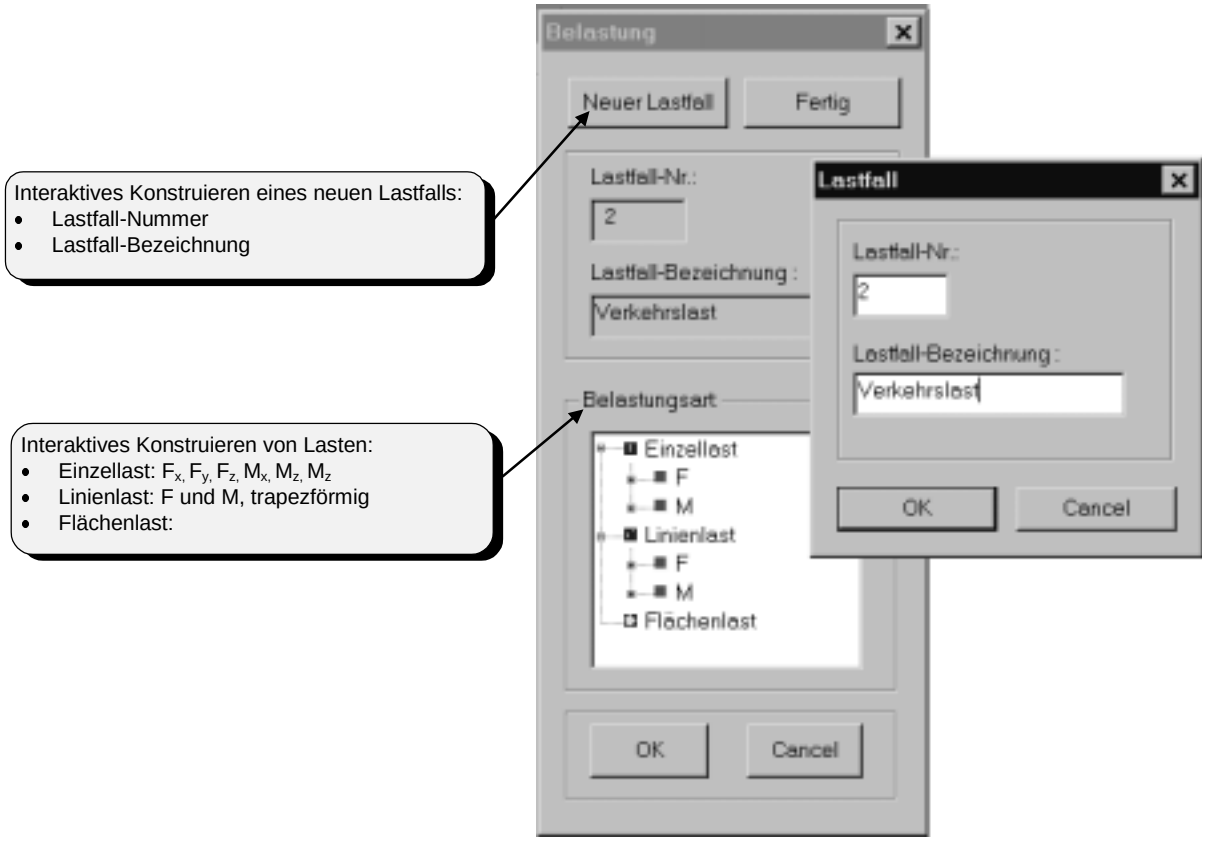


Abbildung 7.3: Interaktives Dialogbox zur Eingabe von Lastfällen und Belastung

- **Performanz**

⇒ **Schnelligkeit**
Wie schnell verschiedene Operation eines Programms durchgeführt werden, hängt wesentlich von der Hardware wie Prozessor, Grafikkarte und Arbeitsspeicher ab. Ebenso spielen die Systemstruktur, die Leistung der dynamischen Listen, die Komplikation der Beziehung zwischen den Objekten und Verzweigung der Funktionsaufrufe eine große Rolle.

Es ist bemerkbar, daß die Schnelligkeit des Prozessors die Suche in den Objektlisten weitgehend bestimmt, wodurch die Grundfunktionalitäten eines CAD-Systems wie z.B. Identifizieren, Kopieren oder Verschieben eines Objekts beeinflußt werden. Dabei ist die Leistung der Videokarte, d. h. der Videospeicher und der Grafikprozessor sind sehr wichtig beim Zeichnen auf dem Bildschirm, vor allem beim Zeichnen in mehreren Views, z. B. in 2D- und 3D-View gleichzeitig.

Es wurde die Schnelligkeit des CAD-Systems MvCad beim Kopieren und Identifizieren eines Objekts untersucht. Dieser Test wurde auf zwei Computer mit unterschiedlichen Prozessoren und Arbeitsspeichern ausgerüstet. Der erste Computer ist mit einem Pentium Prozessor 133 MHz, Arbeitsspeicher RAM (Random Access Memory) 32 MB und Videospeicher VRAM 2 MB, der zweite Computer mit einem Pentium Prozessor 233 MHz, Arbeitsspeicher RAM 128 MB und Videospeicher VRAM 4 MB ausgestattet.

Bei diesem Test wurden folgenden Bedingungen zugrundegelegt:

- Kopieren und Identifizieren wurden auf ein Linienlager-Objekt durchgeführt.
- Beim Kopieren eines Objekts wurde immer von einem einzigen vorhandenen Objekt ausgegangen.
- Nach viermaliger Durchführung des Tests wurde der Mittelwert berechnet.
- Der Test wurde nur einmal in der 2D-View und einmal in 2D- und 3D-View durchgeführt.

Folgende Tabelle zeigt die Ergebnisse der Schnelligkeit beim Kopieren eines Linienlager-Objektes auf Computer mit einem Prozessor Pentium 133 MHz, 32 MB RAM, 2 MB VRAM:

Anzahl der Objekte	**Schnelligkeit beim Kopieren eines Objektes (PC: Pentium 133 MHz, 23 MB RAM, 2 MB VRAM)**	
	in 2D-View [sec]	**in 2D und 3D-View [sec]**
50	0,220	0,870
100	0,610	3,185
500	9,976	61,883
1000	33,856	258,530
1500	72,660	414,580

Folgende Tabelle zeigt die Ergebnisse der Schnelligkeit beim Identifizieren eines Linienlager-Objektes auf Computer mit einem Prozessor Pentium 233 MHz, 128 MB RAM, 4 MB VRAM:

Anzahl der Objekte	**Schnelligkeit beim Kopieren eines Objektes (PC: Pentium 233 MHz, 128 MB RAM, 4 MB VRAM)**	
	in 2D-View [sec]	**in 2D und 3D-View [sec]**
50	0,082	0,440
100	0,210	1,480
500	5,040	36,828
1000	17,922	144,690
1500	37,580	323,017

Folgende Tabelle zeigt die Ergebnisse der Schnelligkeit beim Identifizieren eines Linienlager-Objektes auf Computer mit einem Prozessor Pentium 133 MHz, 32 MB RAM, 2 MB VRAM:

Anzahl der Objekte	Schnelligkeit beim Identifizieren eines Objektes (PC: Pentium 133 MHz, 23 MB RAM, 2 MB VRAM)	
	in 2D-View [sec]	in 2D und 3D-View [sec]
50	≈ 0,000	≈ 0,000
100	≈ 0,000	0,040
500	0,050	0,122
1000	0,098	0,462
1500	0,177	0,548

Folgende Tabelle zeigt die Ergebnisse der Schnelligkeit beim Identifizieren eines Linienlager-Objektes auf Computer mit einem Prozessor Pentium 233 MHz, 128 MB RAM, 4 MB VRAM:

Anzahl der Objekte	Schnelligkeit beim Identifizieren eines Objektes (PC: Pentium 233 MHz, 128 MB RAM, 4 MB VRAM)	
	in 2D-View [sec]	in 2D und 3D-View [sec]
50	≈ 0,000	≈ 0,000
100	≈ 0,000	≈ 0,000
500	0,028	0,037
1000	0,052	0,056
1500	0,058	0,060

⇒ **Speicherbedarf**
Der gesamte Speicherbedarf einer Windows-Applikation ist aus folgenden Gründen nicht einfach zu berechnen:

- Es werden temporäre Objekte erzeugt, die auch zur Laufzeit gelöscht werden.
- Objekte der Oberfläche gehören der Klassenbibliothek MFC und stehen in einer Klassenhierarchie, bei der die Speichergröße der Objekte nicht einfach zu ermitteln.
- Verschachteln der Zeiger in einigen Objekten, d. h. Objekte mit Zeigern auf Objekte, die auch in ihrer Struktur Zeiger besitzen.

Die Speichergröße der Bauteilobjekte kann mit geringem Aufwand ermittelt werden, da jedes Objekt eine Methode besitzt, die die Speichergröße der Objektstruktur ausrechnet und schon bei der Entwicklung des Datenmodells vorgesehen war.

Wenn der Arbeitsspeicher des Computer nicht ausreicht, werden temporäre Dateien vom Betriebsystem auf der Festplatte ausgelagert. Dies ist bemerkbar, wenn sehr viele Objekte erzeugt werden. Außerdem führt es zu einer großen Verzögerung des Systems. Die Grenze, an der das System mit der Auslagerung auf die Festplatte beginnt, hängt von der Speichergröße des Computers und von den anderen Applikationen, die auf dem Computer gleichzeitig ablaufen, ab.

Es stellt sich die Frage, ob die Auslagerung der Objekte dem Betriebsystem überlassen, oder ob die Speicherverwaltung vom CAD-System übernommen werden soll, was mit viel mehr Programmieraufwand verbunden ist. Dabei ist ein Kostenvergleich zwischen dem Programmieraufwand und dem immer billiger werdenden Arbeitsspeicher und leistungsfähigen Prozessoren nötig.

7.3 Bewertung

Durch die Entwicklungsarbeit und das Testen des CAD-Systems konnten folgende Erkenntnisse genommen werden:

- Die Technologie der Objektorientierung bei der Produktmodellierung scheint ein guter Ansatz zu sein, um besonderes flexible wartbare Softwarelösungen im Bauwesen zu finden. Dies hat sich in allen Phasen der Produktmodellierung von der Analyse über das Design bis hin zur Implementierung bewährt.

- Als neutrales Teilproduktmodell für die Tragwerksplanung kann das CAD-System unter Berücksichtigung der fachlichen Belange der Tragwerksplanung einen neutralen Datenbestand zur Verfügung stellen. Alle anderen CAD-Systeme für Berechnung, Bemessung und Konstruktion können diesen Datenbestand weiter benutzen. Dies ist für den Tragwerksplaner insbesondere beim iterativen Entwurf- und Konstruktionsprozeß komplexer Tragwerken von Vorteil.

- Die objektorientierte Benutzeroberfläche (GUI = Graphical User Interface) muß aufgabenorientiert und benutzerfreundlich gestaltet werden. Dies war durch die Fenstertechnik und die grafischen Elemente mit allen zugehörigen Funktionalitäten, die das System Windows zur Verfügung stellt, in einer angemessenen Zeit realisierbar. Mit den heute zur Verfügung stehenden Techniken und Werkzeugen kommt es nun mehr auf den Designer an, wie weit die GUI benutzerfreundlich, intuitiv und aufgabenorientiert ist.

- Die Einfachheit bei der Modellierung eines Problembereiches und beim Design eines Datenmodells spielt eine große Rolle bei der Erweiterung und Wartung sowie für die Performanz des entstehenden CAD-Systems. Dies wurde von Anfang an bei der Konzipierung des im Rahmen dieser Arbeit entwickelten Datenmodells berücksichtigt, und durch die Entwicklung einer eigenen grafischen Bibliothek mit einfacher Hierarchie und eigener Verwaltungsobjekten mit dynamischer Speichergröße realisiert.

- Die Trennung zwischen den unterschiedlichen Designkomponenten des CAD-Systems durch wohl definierte erkennbare Schnittstellen hat sich als gutes Konzept bewährt. Dadurch wurden Schnittstellen zum Beispiel zwischen den Daten (Dokument) und den Sichten (Views), zwischen dem Erzeugen eines Objektes und seinem Konstruktionsprozeß, entwickelt. Dies ermöglicht ein besser strukturiertes, flexibles und effizienteres Modelldesign.

- Ohne relationale oder objektorientierte Datenbanken geht es in der Zukunft nicht mehr. Ein neutrales Speicherkonzept für das Datenmodell wurde zwar ohne Einsatz von Datenbanken zur Speicherung der Bauobjekte mit ihren Referenzobjekten modelliert und realisiert. Dies war durch die einfache Modellierung des Problembereiches und die Klassenhierarchie gelungen. Dies ist bei den komplizierten Datenmodellen sehr schwierig, vor allem die Pflege des Speicherungsformats bei neuen Versionen des Systems und der Einsatz im Netzwerk machen es fast unmöglich.

- Die Performanz des CAD-Systems ist durch die Schnelligkeit und Speicherbedarf charakterisiert. Dies hängt wesentlich von der Hardware wie Prozessor, Grafikkarte und Arbeitsspeicher, als auch von der Systemstruktur wie Schnelligkeit der Listensuche, Beziehung zwischen den Objekten und Verzweigung der Funktionsaufrufe, ab. Die Schnelligkeit des Prozessors beeinflußt die Suche in den Objektlisten und somit auch die Grundfunktionalität eines CAD-Systems (z.B. Identifizieren, Kopieren oder Verschieben eines Objekts). Die Eigenschaften und die Leistung der Videokarte sind sehr wichtig beim Zeichnen auf dem Bildschirm vor allem beim erneuten Zeichnen (update) der Views und beim gleichzeitigen Arbeiten in mehreren Views (z.B. in 2D- und 3D-View). Der Weg der Suche für ein Objekt in den Listen der Datenhaltung ist neben der Leistung der Hardware maßgebend für die Schnelligkeit. Dies wird wesentlich vom Strukturdesign der dynamischen Listen für die Datenhaltung und vom Design der Beziehungen zwischen den Objekten im Datenmodell beeinflußt. Die Frage, die sich oft stellt, ist, ob man eine eigene dynamische Container-Bibliothek entwickeln und weiter pflegen soll, oder eine externe gekaufte einsetzt. Es wurden zwar eigene Container-Klassen im Rahmen dieser Arbeit entwickelt, vorteilhafter wäre jedoch der Einsatz einer externen, fertigen und geprüften Contontainer-Bibliothek von anderen Softwarehersteller, wie z.B. die STL (Standard Template Library) der Firma Hewlett Packard. Die Vorteile liegen darin, daß bei solchen standardisierten Contontainer-Bibliotheken die Leistung, die Richtigkeit, die Weiterentwicklung und Pflege von der Seite der Softwarehersteller gewährleistet sind.

8. Zusammenfassung und Abschlußbetrachtung

8.1 Zusammenfassung

In Rahmen dieser Arbeit wurde ein objektorientiertes Datenmodell für ein CAD-System zur Unterstützung der Tragwerksplanung entwickelt, realisiert und getestet.

In **Kapitel 1** wurde die Bauinformatik und ihre Aufgaben als ein relativ neues Forschungsgebiet im Bauingenieurwesen, das die Wissensgebiete der Informatik mit den spezifischen Anforderungen des Bauwesens verknüpft, vorgestellt. Es wurden die Techniken, Methoden, Modelle und Prozesse der Informationsverarbeitung, die die Bauinformatik zur Entwicklung moderner Ingenieurlösungen benötigt, dargestellt. Darüber hinaus wurde ein Überblick über die Historie der Bauinformatik und die Ursprünge der Informationstechnik im Bauwesen vermittelt.

Der Stand der Wissenschaft wurde in **Kapitel 2** geschildert. Es wurde festgestellt, daß das Computergestützte Design CAD (Computer Aided Design) ein gutes Mittel ist, um die integrierte Informationsbearbeitung zur Verfügung zu stellen und alle Software- und Hardware-Komponenten zu einem komplexen Werkzeug zur Unterstützung beim Planen, Konzipieren, Entwerfen und Ausarbeiten von Bauobjekten zusammenzufassen. Außerdem wurde auf Integrationsansätze von Arbeitsprozessen im Bauwesen eingegangen und welche Vorteile dadurch erzielt werden können, sowohl die bekannten Vorteile der Nutzung von CAD, wie beispielsweise maßstabsloses Zeichnen, erhöhte Planungsqualität und verbesserte Änderungsdienste, als auch Vorteile bei der Bearbeitungszeit. Weiterhin wurde die verteilte Verarbeitung wie verteiltes Rechnen (Distributed Computing) und verteilte Objekte (Distributed Objects) als eine neue Software-Architektur in der Zeit der Netzwerke aufgezeigt, wo Daten und Programme nicht auf dem gleichen Rechner vorhanden sein sollen. Es wurden zwei dafür neu entstandene Architekturen von zwei starken Softwareherstellern dargestellt: Der neue Standard CORBA (Common Object Request Broker Architecture), der von OMG (Object Management Group) entwickelt wurde, und die spezifische Verteilungslösung DCOM (Distributed Component Object Model) der Firma Microsoft.

Die Zielsetzung dieser Arbeit ist in **Kapitel 3** festgelegt. Es wurde die Aussage getroffen, daß eine große gemeinsame Datenbasis für das Bauwesen nicht zu erreichen ist (mehrere Versuche: ICES, IST). Daraus sind viele Informationsinseln entstanden. Die Verbindung zwischen den Informationsinseln ist nicht systematisiert. Ein Datenmodell ist jeweils für die einzelnen Prozesse im Bauwesen erforderlich. Die Entwicklung eines CAD-Systems mit einem objektorientierten Datenmodell für die Tragwerksplanung von Gebäuden, sowie das Realisieren und Testen wird als Ziel der vorliegenden Arbeit dargelegt. Der gesamte Entwicklungsprozeß erfolgt mit Hilfe der objektorientierten Technologie. Die Objektorientierte Modellierung OOM (Object Oriented Modeling) von Teilproduktmodellen wurde im Bereich der Tragwerksplanung bisher nur in der Forschung angewendet. Der Weg dieser Entwicklung geht von der objektorientierten Analyse OOA (Object Oriented Analysis),

über das objektorientierte Design OOD (Object Oriented Design) bis hin zur objektorientierten Programmierung OOP (Object Oriented Programming).

In **Kapitel 4** wurden die Elemente und Begriffe der objektorientierten Technologien zunächst erläutert und die objektorientierten Modellierungsmethoden bei der Softwareentwicklung, das sogenannte Software-Engineering, von der Analyse über Design bis hin zur Implementierung charakterisiert. Dabei wurde auf die objektorientierte Modellierungsansätze von Coad/Yourdon, Booch und Rumbaugh eingegangen. Es wurde die Bedeutung des Schlagwortes **„***objektorientiert*" geschildert, das inzwischen zu jeder guten Software gehört. Neben der Einführung neuer Sprachmittel bedeutet das vor allem das Kennenlernen einer neuen Begriffswelt und ein neues Denken. Dabei scheint sich die Programmiersprache C++ als quasi etablierter Standard für einen Sprach-Hybrid durchzusetzen, der objektorientierte und nicht-objektorientierte Programmierung ermöglicht.

Ein objektorientiertes Datenmodell für die Tragwerksplanung wurde in **Kapitel 5** vorgestellt. Dabei wurden die realen Objekte der Tragwerksplanung von Gebäuden mit ihren Eigenschaften und Funktionalitäten sowie ihre Beziehung untereinander auf Softwareobjekte übertragen. Die Bauobjekte wurden identifiziert und in Klassen spezifiziert. Die dabei entstandenen Klassen stehen in einer Klassenhierarchie, in der Unterklassen von Basisklassen direkt oder nicht direkt abgeleitet werden. Die Klassen wurden in Klassenkategorien gegliedert. Die Basisklassen der verschiedenen Klassenkategorien wurden mit ihren Funktionalitäten und virtuellen Methoden beschrieben. Es wurde auf die Klassen der verschiedenen Klassenkategorien im Datenmodell eingegangen sowie auf die Datenstruktur und die Methoden der Objekte.

In **Kapitel 6** wurde das Designkonzept für das Pilot-CAD-System *MvCad* zur Realisierung des im Kapitel *5* dargestellten Datenmodells erläutert. Aufgabe des entwickelten CAD-Systems war die Zusammenstellung und die Haltung der verschiedenen Daten, wie die Strukturdaten des Tragwerksmodells und die grafischen Daten für die Darstellung bis hin zur Organisation, Speicherung und dem Austausch dieser Daten mit anderen Systemen. Es wurden die entwicklte Klassen für die Mengenverwaltung und Datenhaltung im Modellkonzept erläutert, die die Objekte verschiedener Klassen in dynamischen Listen und Containerklassen verwalten. Es wurde das gesamte Designkonzept des CAD-Systems *MvCad* geschildert. Dabei wurden die Document-View Architektur mit den verschiedenen Sichten wie 2D-, 3D- und Textsicht, die mit Hilfe der Microsoft Klassenbibliothek MFC realisiert ist, die Abstraktionsebenen, die durch klar definierte Schnittstellen repräsentiert sind und die objektorientierte Benutzeroberfläche GUI (Graphical User Interface) mit den CAD-Funktionalitäten, beschrieben.

Der Realisierungsumfang des im Kapitel *6* beschriebenen Designkonzepts wurde in **Kapitel 7** geschildert und das CAD-System *MvCad* wurde getestet. Dabei wurden Erzeugung, Identifizierung und Änderung (Editieren), Import und Export der Objekte im Datenmodell sowie die CAD-Funktionalität des Systems und die Benutzeroberfläche getestet. Es wurden weiterhin die Schnelligkeit und Performanz untersucht. Anschließend wurden Teilproduktmodell, Designkonzept und die Vorteile der eingesetzten Technologien diskutiert und bewertet.

8.2 Abschlußbetrachtung und Ausblick

Die Bauinformatik als Vermittler der Informationstechnik im Bauwesen hat heute und in der Zukunft die Aufgabe, effiziente, leistungsfähige und moderne Technologien aus dem Bereich der Software und Hardware einzusetzen, um eine durchgängige integrierte Datenverarbeitung für den Gesamtbauprozeß vom Entwurf über die Ausführung bis hin zur Wartung von Bauwerken zu gewährleisten.

Dazu können folgende Überlegungen herangezogen werden:

- Die Gestaltung der Software-Lösungen für Computersysteme im Bauwesen besteht im Wesentlichen aus der fachlichen und technischen Modellierung des DV-Systems. Darüber hinaus ist eine durchgängige Methodik erforderlich, die durch geeignete Modellierungsmethoden und Werkzeuge unterstützt wird und die garantiert, daß die fachlich definierten Aufgaben technisch realisiert werden können.

- Die objektorientierte Vorgehensweise ist heute und auch in den nächsten Jahren eine sichere Strategie bei der Entwicklung von Software-Produkten. Die Software kann effizient modelliert und realisiert werden. Die objektorientierte Modellierung eröffnet durchaus viele Möglichkeiten, die Produktivität weiter zu steigern, die Qualität zu verbessern und auch auf neue Anforderungen flexibel reagieren zu können. Eine dieser Möglichkeiten entsteht durch die Integration organisatorischer Vorgehensweisen, die sich in fachliche und technische Bestandteile gliedern.

- Der Prozeß der Software-Entwicklung läuft nicht streng sequentiell ab. Rückgriffe auf vorgehende Entwicklungsphasen und entsprechende Modifikationen der zugehörigen Software-Dokumente sind oft in den verschiedenen Phasen des Entwicklungsprozesses nötig, z. B.:

  $\Rightarrow$ Rückkehr vom Entwurf zur Aufnahme der Problemanalyse, nachdem Mißverständnisse oder Lücken in den Anforderungsdefinitionen erkannt wurden.

  $\Rightarrow$ Rückkehr von der Implementierung zum Entwurf, nachdem entdeckt wurde, daß ein Software-Modul mit der für ihn gültigen Schnittstelle überhaupt nicht oder ineffizient implementiert werden kann.

  Durch neue Qualifizierungsanforderungen wird die Welt der Organisatoren, Analytiker, Designer, Implementierer und Anwender verändert. Hiervon ist auch das Projektmanagement betroffen, das in Zukunft nicht mehr mit einer sequentiellen Vorgehensweise entsprechend dem „Wasserfall-Modell“ rechnen kann. Vielmehr wird ein paralleles Vorgehen mit Meilensteinen und Iterationen entsprechend dem dynamischen „Prototyping-Modell“ benötigt.

- Produktivitätssteigerung in der Softwareentwicklung und eine signifikante Erhöhung der Wiederverwendbarkeit können durch den Einsatz objektorientierter CASE-Werkzeuge (Computer Aided Software Engineering) in Kombination mit Objekt-Datenbank-Repositories erfolgen.

- Die bisherigen Überlegungen im Bereich der Software-Entwicklung haben die Verteilung von Prozessen und Daten nicht respektiert. Es wird schweigend davon ausgegangen, daß die Datenverarbeitung auf einem Computer abläuft, auf dessen Platten auch die Daten gespeichert sind. Dies ist bei den modernen netzwerkorientierten Anwendungen immer seltener der Fall. In der Zeit der Netzwerktechnologie brachte die Entwicklung der Preise und der Leistung die Idee mit, in den objektorientierten Anwendungen auftretende Objekte als verteilt zu betrachten. Die darauf basierende Software-Architektur kann die Granularität der verteilten Daten wesentlich verkleinern.

- Den neuen Software-Anwendungen in verteilten Netzen genügen die zentralen Netzdienste, die von Netzwerk-Betriebsystemen wie Novell Netware oder Windows NT zur Verfügung gestellt sind, allerdings nicht mehr. Dies sorgte für die Geburt einer völlig neuen Softwarekategorie, nämlich die „Middleware“, die sich als Software-Komponente „zwischen“ dem Anwender und dem eigentlichen Server befindet, eben in der Mitte. Sie wickelt ihre Arbeit nicht im direkten Kontakt mit dem Endbenutzer (FrontEnd-Software) ab, aber stellt auch nicht selber die Server-Dienste zur Verfügung. Es entstand dafür eine Technologie der sogenannten ORB (Object Request Broker), die als Vermittler zwischen dem Client als Nachfrager eines bestimmten Dienstes und dem Diensterbringer, der sich irgendwo innerhalb des Netzes befindet. Hier prallen derzeit zwei unterschiedliche Konzepte aufeinander, die auf die Integration der ORB-Technologie setzen: DCOM (Distributed Common Object Model) der Softwarehersteller Microsoft und CORBA (Common Object Request Broker Architecture) der Softwaregruppe OMG (Object Management Group). Die Vorteile beim Einsetzen dieser neuen Software-Architektur mit ihren klar definierten Schnittstellen für verteilte Datenverarbeitung - verteiltes Rechnen (Distributed Computing) - können dazu beitragen, neue Aspekte zu entwickeln und Lösungen zu den vielen Problemen des Informationsaustausches, der Kommunikation und der Integration im Bauwesen zu finden.

- Für die Datenhaltung in objektorientierten Anwendungen gilt auch heute noch folgender Kompromiß: Die Speicherung der gemeinsamen Unternehmensdaten erfolgt auf Mainframes, auf dezentralen Daten-Servern und auf verteilten Objekt-Servern, überwiegend in relationalen Datenbanksystemen. Objektorientierte Datenbanken sind zwar seit einiger Zeit im Gespräch, scheinen sich aber nur langsam durchzusetzen. Die Entwicklungstendenz geht hier zusammen mit Internet-Anwendungen zu Systemen mit verteilten Objekten, bei denen die Objekte rational gespeichert werden. Dabei werden die Objekte und ihre Funktionalität vom Objekt-Server bereitgestellt und über einen ORB verteilt. Bei Client/Server-Lösungen werden, im Gegensatz hierzu, die Objekte und ihre Funktionalität auf den einzelnen Client separat zur Verfügung gestellt und auf dem Server lediglich Datensätze verwaltet.

- Die Planung der Qualitätsanforderungen ist produktabhängig. Damit die Qualität eines Software-Produkts und ihre Entwicklung verglichen werden können, sollen möglichst Qualitätsmerkmale angegeben und Qualitätsmaße festgelegt werden. Es sollen Auswertungsziele, Art der Erfassung der Meßwerte und das Interpretieren der Meßergebnisse definiert werden. Als Qualitätsmerkmale eines Software-Produkts können zum Beispiel Zuverlässigkeit, Übertragbarkeit, Effizient und Benutzerfreundlichkeit definiert werden, darunter sind die Vollständigkeit, Korrektheit, Konsistenz und Benutzbarkeit gegliedert. Weiterhin können die Wartbarkeit und Pflege zu den Qualitätsmerkmalen eins Software-Produkts gehören, darunter sind die Änderbarkeit, Verständlichkeit, Testbarkeit und Dokumentation des gesamten Software-Entwicklungsprozesses zu verstehen.

Literatur

[1] Alcock, D. G.; Shearing, B. H.: GENESYS, An Attempt to Rationalise the Use of Computers in Structural Engineering, The Structural Engineer, London, Vol. 48, Nr. 4, April 1970

[2] Ammeraal, Leendert: Computer Graphics for the IBM PC, John Wiley & Sons, 1987

[3] Bauwesen 15, Normen über Kosten von Hochbauten, Flächen, Rauminhalte. 2. Auflage. Berlin, Köln: Beuth Verlag GmbH, 1981

[4] Berson. A.: Client/Server Architecture. McGraw-Hill Verlag, 1992

[5] Berufsfeld Bauingenieure, Ingenieursoziologische Einführung und Handreichung zur Ingenieurbefragung. Universität Kassel, Umdruck zur Lehrveranstaltung 'Soziale und ökonomische Bedingungen der Ingenieurarbeit im Bauwesen' SÖI, WS 95/96 /Neuauflage WS96/97

[6] Boehm, B.: A Spiral Model of Software Development and Enhancement, Seite 22, Software Engineering Notes vol. 11(4) August 1986

[7] Booch, Grady: Objektorientierte Analyse und Design. Bonn, Paris, Reading, Mass. u.a.: Addison-Wesley, 1994

[8] Borghoff, U.M.; Schlichter, J.H.: Rechnergestützte Gruppenarbeit. Eine Einführung in Verteilte Anwendungen. Springer Verlag, 1995

[9] Bretschneider, D.: Modellierung rechnerunterstützter, kooperativer Arbeit in der Tragwerksplanung, Vortschritt-Berichte VDI, VDI-Verlag Düsseldorf, Reihe 4 Bauingenieurwesen Nr. 151, 1998

[10] Burger, Peter, Duncan Gillies: Interactive Computer Graphics, Addison-Wesley, 1989

[11] Coad, Yourdon: Object-oriented analysis, Prentice-Hall, 1991

[12] Coad, Yourdon: Object-oriented design, Prentice-Hall, 1991

[13] Diaz, J.: Objektorientierte Modellierung geotechnischer Ingenieursysteme. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 173, 1995

[14] Diederich, D.: Entwicklung und Implementierung von Algorithmen zur Berechnung von Wandverschneidungen für die Grundrißdarstellung im Programmsystem PreCad. Universität Kaiserslautern: Studienarbeit Bauinformatik, 1995

[15] Eigner, Martin: Einstieg in CAD, Hanser-Verlag, 1985

[16] Encarnação, J., Schlechtendahl, E.G.: Computer Aided Design, Springer-Verlag, 1983

[17] Encarnação, J.: Aktuelle Themen der Graphischen Datenverarbeitung, Springer-Verlag, 1986

[18] Erich, R.: HOAI 91, Honorarordnung für Architekten und Ingenieure, RudolfMüller Verlag, Köln 1995

[19] Erlenkötter, Reher: C für Windows, Rowohlt Verlag, 1993

[20] Fellner, W.D.: Computer Grafik, BI-Wissenschaftsverlag, 1988

[21] Ficke, H.-G.: - Gruppe Hannover - CAD-Systeme für den Architekturbereich

[22] Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John: Design Patterns, Addison-Wesley, 1995

[23] Gorny, P,. Viereck, A.: Interaktive grafische Datenverarbeitung. B.G. Teubner Verlag, 1984

[24] Grieger, Ingolf: Graphische Datenverarbeitung. 2.Auflage. Springer-Verlag, 1987

[25] Gupta, R., Horowitz, E. Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD. Englewood Cliffs, N.J.: Prentice Hall, 1991

[26] Haas, W.: Datenaustausch und Datenintegration, Sonderdruck aus Beratende Ingenieure, Springer-VDI-Verlag, November 1997

[27] Hammer, Hans: Handbuch für CAD im Bauwesen - Das Baupaket. 1.Auflage IRB-Verlag

[28] Harrington, Steven: Computergrafik - Einführung durch Programmierung, McGraw-Hill, 1988

[29] Hartmann, D.: Aktuelle Forschungsgebiete, Workshop: „Forschung in der Bauinformatik“, Vortrag, 23.-25. März 1999, Hotel Schloß Berg, Starnberger See

[30] Heck, P.: Object-oriented CAD-model for building design. Proc. 6th. ICCCBE. Berlin, Germany, 1995

[31] Heck, P.: Objektorientierte Modellierung am Beispiel der Integration raum- und bauteilorientierter Daten in einem zentralen Objektmodell. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 131, 1994

[32] Hein, Olaf: Graphentheorie für Anwender. Mannheim: Bibliographisches Institut, 1977

[33] Heuer, A.: Objektorientierte Datenbanken. Bonn, München, Paris, u.a.: Addison-Wesley, 1992

[34] Hinz, O.: Ein objektorientiertes Modell für Entwurf und Berechnung von Grundbaukonstruktionen. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 131, 1994

[35] Holz, K.-P.; Weitendorf, D.: Was ist Bauinformatik, Dokument im Internet (http://www.bauinf.tu-cottbus.de/Bauinformatik/Bauinformatik.html), TU Cottbus, Institut für Bauinformatik.

[36] Informationssystem für das Bauwesen (ISB), 4. Fassung (Juni 1970). Projektvorschlag der Fakultät für Bauingenieurwesen der Technischen Universität Berlin und des Entwicklungszentrums EDV Bau e. V., Stuttgart

[37] Jell, Thomas, von Reeken, J.: Objektorientiertes Programmieren mit C++. 2. bearb. und erw. Aufl.. München, Wien: Carl Hanser Verlag, 1993

[38] Kahlen, Hans: CAD-Einsatz in der Architektur. Stuttgart, Berlin, Köln: W. Kohlhammer, 1989

[39] Koller, Rudolf: CAD - Automatisches Zeichnen, Darstellen und Konstruieren, Springer-Verlag, 1989

[40] Kroha, P.: Softwaretechnologie, Prentice Hall Verlag, 1997

[41] Kruglinski, David J.: Inside Visual C++, Microsoft Press1996

[42] Leixner, M.:Implementierung von Tragwerksobjekten in einem objekt-orientierten CAD-System, Universität Kaiserslautern: Diplomarbeit Bauinformatik, April 1997

[43] Lennerts, K.: Objektorientierte Modellierung der Baustelle unter dem Gesichtspunkt des Materialflusses. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 99, 1993

[44] Lennerts, K.: Objektorientierter Entwurf von ESBE - Eine Vorgehensweise. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 131, 1994

[45] Meier, A.: Methoden der grafischen und geometrischen Datenverarbeitung. Stuttgart: B.G. Teubner, 1986

[46] Meier, Andreas: Erweiterung relationaler Datenbanksysteme für technische Anwendungen. Springer-Verlag, 1987

[47] Meißner, von Mitschke-Colande, Nitsche: CAD im Bauwesen, Springer Verlag, 1992

[48] Mitchell, William, J.: Computer-Aided Architectural Design, Mason/Charter Publishers, 1977

[49] Mitschang, Bernhard: Ein Molekül-Atom-Datenmodell für Non-Standard-Anwendungen, Springer-Verlag, 1988

[50] Newman, William, M.: Grundzüge der interaktiven Computergrafik, McGraw-Hill, 1986

[51] Niestroj, C.: Objektorientierte Analyse für den bauteilorientierten Datenaustasch von der Objekt- zur Tragwerksplannung, Dissertation, TH Darmstadt, 1993.

[52] Noltemeier, Hartmut: Graphentheorie mit Algorithmen und Anwendungen. Berlin: Walter de Gruyter, 1976

[53] Pahl, P. J.: Der Aufbau des Informationssystems für das Bauwesen, Vorträge Betontag 1971, Deutscher Beton-Verein E. V., Wiesbaden 1971

[54] Papurt: Inside the Object Model, Sigs Books, 1995

[55] Park, Chan S.: Interactive Microcomputer Graphics, Addison-Wesley, 1985

[56] Pawelski, Michael: CAD-Leitfaden für Architekten, Verlag C.F. Müller, 1987

[57] Peter Pahl, Entwicklungstendenzen der Informationstechnik im Bauwesen, Vortrag an der Fakultät für Bauingenieurwesen der Technischen Universität Berlin, Mai 1998

[58] Pfeiffer, Thomas: CAD für Bauingenieure. Braunschweig: Vieweg-Verlag, 1989

[59] POET 2.1 Programmers & Reference Guide. Hamburg: POET Software GmbH, 1993

[60] Pomaska, Günter: 3D-Grafik auf dem PC, Vogel-Verlag, 1986

[61] Richter: Windows Programmierung für Experten, Microsoft Press, 1995

[62] Roos, D.: ICES System Design. Massachusetts Institute of Technology Press, Cambridge, Mass., September 1967

[63] Rumbaugh, Blaha, Premerlani, Eddy, Sorensen: Object Oriented Modeling and Design, Prentice-Hall, 1991

[64] Rüppel, U.: Integration von Teilprozessen des Bauplanungsprozesses mit objektorientierten Schnittstellen basierend auf STEP-2DBS. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 4 Nr. 116, 1992

[65] Rüppel, U.: Objektorientierter Datenaustausch zwischen Entwurfs- und Tragwerksplaner. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 99, 1993

[66] Rüppel, U.: Produktmodellierung im Bauwesen. Forum Bauinformatik "Junge Wissenschaftler forschen". VDI-Verlag Reihe 20 Nr. 131, 1994

[67] Schäfer, Steffen: Objektorientierte Entwurfsmethoden: Verfahren zum objektorientierten Softwareentwurf im Überblick. Bonn; Paris, Reading, Mass u.a.: Addison-Wesley, 1994

[68] Schenkengel, K.U.: Definition, Modellierung und Implementierung der interaktiven Funktionen für die Grundrißbearbeitung. Universität Kaiserslautern: Studienarbeit Bauinformatik, 1996

[69] Schwarz, Heinz: Daten- und Informationsverarbeitung Berlin: Ernst & Sohn Verlag, 1988

[70] Sedgewick, R.: Algorithmen in C++. Bonn, München, Paris u.a: Addison-Wesley, 1992

[71] Sherer, R. J.: Verteiltes, gleichzeitiges Planen, Verwalten und Projektsteuerung im Internet, Workshop: „Forschung in der Bauinformatik“, Vortrag, 23.-25. März 1999, Hotel Schloß Berg, Starnberger See

[72] Spur, Günter, Krause, Frank-Lothar: CAD-Technik, Hanser-Verlag, 1984

[73] Stoyan, H., Wedekind, H.: Objektorientierte Software- und Hardwarearchitekturen. Stuttgart: B.G. Teubner, 1983

[74] Stroustrup, B.: Die C++ Programmiersprache. 2. überarbeitete Auflage. Bonn, München, Paris u.a: Addison-Wesley, 1992

[75] Vetter, M.: Objektmodellierung. Stuttgart: B.G. Teubner, 1995

[76] Wassermann, K.: Eine objektorientierte FEM-Modellierung in Form von Bauteilpositionen, 4. FEM/CAD-Tagung Darmstadt, Erfahrungsaustausch und technologietransfer im Bauwesen. VDI Verlag, Reihe 20, Nr. 214, U. Meißner, K. Wassermann.

[77] Weber, H.R.: CAD-Datenaustausch und Datenverwaltung, ZGDV: Beiträge zur grafischen Datenverarbeitung, Springer-Verlag, 1988

[78] Wirth, N.: Algorithmen und Datenstrukturen. 3. überarb. Auflage. Stuttgart: B.G. Teubner, 1983

Glosar

- **Automatisierung:** Die Automatisierung i.allg. ist der Einsatz von Maschinen zur Erledigung häufig vorkommender Schritte in einem Prozeß, um die Produktivität durch Sparen an Zeit und Arbeit zu erhöhen. Dabei wird der Automatisierungsgrad dadurch bestimmt, wieweit der Mensch in diesen Prozeß eingreift. Die Automatisierung in der Welt der Informationstechnik wird durch den Einsatz von Computersystemen zur Erledigung häufig vorkommender Arbeitsschritte in den Entwurfs- und Produktionsphasen gekennzeichnet.

- **Bauteilorientierung:** Die Bauteilorientierung ist eine Vorgehensweise, die die in einem Gebäude vorkommenden Gegenstände als Bauteile klassifiziert. Damit stellen die Bauteile die Grundeinheiten dieser Klassifizierung dar. Diese Grundeinheiten besitzen gemeinsame Eigenschaften.

- **CAD:** CAD ist die Abkürzung für: Computer Aided Design. Es bedeutet das rechnergestützte Entwerfen und Konstruieren. Damit ist der Einsatz des Computers in den Planungs- und Konstruktionsphasen, besonderes in den gebieten Maschinenbau, Elektrotechnik und Bauingenieurwesen, gemeint.

- **CAD-System:** CAD-System ist die Abkürzung für: Computer Aided Design System. Damit ist ein Softwareprogramm zur Unterstützung der Planungs- und Konstruktionsphasen am Computer, besonderes in den gebieten Maschinenbau, Elektrotechnik und Bauingenieurwesen, gemeint.

- **Daten:** Daten stellen eine Abstraktion der ausgewählten Informationen über einen Teil der realen Welt (der Wirklichkeit) dar. Sie sind verschlüsselte Informationen in Zeichen und/oder Zeichenkombinationen zum Zweck der Speicherung und/oder Weiterverarbeitung auf elektronischen Datenverarbeitungsanlage.

- **Datenbasis:** Eine Datenbasis ist eine Menge von Informationen, die in Datenstrukturen dargestellt sind und zum Zweck der Speicherung und/oder Weiterverarbeitung auf elektronischen Datenverarbeitungsanlage dienen.

- **Datenmodell:** Ein Datenmodell ist die Abbildung einer Menge von Informationen für eine rechnerinterne Darstellung.

- **Datenstruktur:** Eine Datenstruktur ist eine dem Rechner angepaßte Darstellungsform einer Menge von Informationen (wie z.B. Felder, Listen, Vektoren). Sie ist eine Festlegung der Darstellung von Daten für die Rechnerverarbeitung.

- **Gebäudemodell:** Ein Gebäudemodell ist die Abbildung der Gesamtheit aller Informationen über das Gebäude (wie z.B. Geometrie, Topologie, Bauteile) für eine rechnerinterne Darstellung.

- **Grafisches Zeichnungselement:** Ein grafisches Zeichnungselement ist ein zugrundeliegendes Darstellungsobjekt, das für die zweidimensionale Abbildung von einem Gegenstand auf einer Fläche eingesetzt wird (wie z.B. Punkt, Linie, Kreis). Alle Zeichnungselemente müssen die Objekthöhe 0 haben. Es dürfen keine 3D-Informationen enthalten sein.

- **Klassen:** Eine Klasse ist die Abstraktion von einer Menge gleichartiger Objekte, die gemeinsame Eigenschaften (Attribute) und gemeinsames Verhalten (Methoden) besitzen. Ein Objekt ist ein Exemplar (Instanz) seiner Klasse.

- **Modell:** Ein Modell ist eine Abstraktion eines Teils der Realität. Ein Modell in der Welt der Informationstechnik ist die Abbildung der Abläufe und der Informationen des zu erstellenden Gegenstands für eine rechnerinterne Darstellung.

- **Modellierung:** Die Modellierung in der Welt der Informationstechnik ist die isolierte Betrachtung der existierenden Abläufe und Informationen des zu erstellenden Gegenstands. Die Vorgehensweise bei der Modellbildung (Modellierung) für eine rechnerinterne Darstellung kann in verschiedene Phasen (Modellierungsphasen) gegliedert werden:

  1. Zerlegen des Problembereiches
  2. Identifizieren der vorkommenden Gegenstände im Problembereich
  3. Strukturieren der Gegenstände (Syntax, Beziehungen)
  4. Beschreiben der Gegenstände (durch Attribute)
  5. Beurteilen

- **Objektorientierung:** Die Objektorientierung ist eine methodische Vorgehensweise (Weltanschauung), die die in der realen Welt vorkommenden Gegenstände als Objekte ansieht. Die Objekte stellen dabei die Grundeinheiten dar, die gemeinsame Eigenschaften (Attribute) und gemeinsames Verhalten (Methoden) in einer Einheit vereinigen und der Außenwelt als abgeschlossene Gebilde gegenüberstehen.

- **Produktmodell:** Ein Produktmodell ist die Abbildung der Gesamtheit aller Informationen über einen Gegenstand. Ein Produktmodell ist ein explizites Modell des zu erstellenden Produkts, das alle relevanten produkt- und produktionsbezogenen Informationen für eine rechnerinterne Darstellung erfaßt und beschreibt.

- **Prozeß:** Ein Prozeß ist der Ablauf eines Vorgangs zur Erledigung einer Aufgabe bzw. einer Arbeit durch mehrere Schritte.

- **Struktur:** Eine Struktur im Kontext objektorientierter Analyse und Design ist die konkrete Darstellung des Status eines Objekts. Ein Objekt teilt seinen Status nicht mit einem anderen Objekt, obwohl alle Objekte derselben Klasse dieselbe Darstellung ihres Status besitzen.

- **Strukturdaten:** Die Strukturdaten sind die Menge von Informationen, die die Eigenschaften einer Struktur erfassen und beschreiben. Sie stellen eine Abstraktion der ausgewählten Informationen über diese Struktur dar.

16. 01.1960	Geboren in Homs - Syrien
1964 – 1973	Grund- und Aufbauschule, Damaskus - Syrien
1973 – 1977	Sekundarschule, Damaskus - Syrien
1977 – 1982	Studium Bauingenieurwesen, Universität Damaskus, Damaskus - Syrien.
1982 – 1988	Bauplaner und Aufsichtsingenieur, die Allgemeine Gesellschaft für Entwerfen, Studien und Technische Beratung, Damaskus - Syrien.
1988 – 1989	Selbständiger Bauingenieur, Damaskus - Syrien
1989 – 1992	Studium Bauingenieurwesen, Konstruktiver Ingenieurbau, Universität Karlsruhe - Deutschland.
1992 – 1997	Wissenschaftlicher Mitarbeiter im Fachgebiet EDV-gestütztes Entwerfen, Berechnen und Konstruieren im Bauingenieurwesen (Bauinformatik) der Universität Kaiserslautern - Deutschland.
1998 – 1999	Softwareingenieur, Forschungs- und Entwicklungsabteilung, Quark Inc., Denver, Colorado - USA.
Seit Oktober 1999	Fertigstellen der Promotionsarbeit am Fachgebiet EDV-gestütztes Entwerfen, Berechnen und Konstruieren im Bauingenieurwesen (Bauinformatik) der Universität Kaiserslautern - Deutschland.

Kaiserslautern, November 1999

Samer Hamwi