Anforderungen und Programmarchitektur objektorientierter 3D-CAD Programme im Bauwesen

Vom Fachbereich
Architektur/Raum- und Umweltplanung/Bauingenieurwesen
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte

Dissertation

von

Patrik Schunk

aus Kaiserslautern

Dekanin:	Prof. Dr. habil. Gabi Troeger-Weiß
1. Berichterstatter:	Prof. Dr.-Ing. Klaus Wassermann
2. Berichterstatter:	Prof. Dr.-Ing. Bernd Streich
Tag der mündlichen Prüfung:	26. Juni 2009

Kaiserslautern 2009

(D386)

Vorwort des Verfassers

Die vorliegende Arbeit entstand im Wesentlichen während meiner Zeit als wissenschaftlicher Mitarbeiter im Fachgebiet Bauinformatik des Fachbereichs Architektur / Raum und Umweltplanung / Bauingenieurwesen der Technischen Universität Kaiserslautern.

Mein besonderer Dank gilt dem Inhaber des Lehrstuhles Herrn Prof. Dr.-Ing. Klaus Wassermann für die wissenschaftliche Unterstützung und vielseitigen Anregungen in unseren Gesprächen.

Dem zweiten Berichterstatter, Herrn Prof. Dr.-Ing. Bernd Streich möchte ich für die Übernahme des Koreferates und Herrn Prof. Dr.-Ing. habil. Christos Vrettos für die Übernahme des Vorsitzes der Promotionskommission recht herzlich danken.

Mein weiterer Dank gilt den Mitarbeitern, Hilfswissenschaftlern und Auszubildenden des Lehrstuhles Bauinformatik, unter anderem Annette Reincke, Asbjörn Gärtner, Paul Schramenko, Edyta Wojtacki und Marco Hönsch für ihre Unterstützung bei meiner Arbeit.

Meinen Freunden und Kollegen, die mir den Rücken gestärkt und mich ermutigt hatten, diese Arbeit zu beenden, möchte ich danken. Bei Frau Corinna Bettinger und Herrn Daniel Wüst Danke ich für ihre Unterstützung bei der Berarbeitung meiner Dissertation.

Nicht zuletzt möchte ich meinen Eltern danken für ihre Unterstützung auf meinem Weg. Ohne ihre Förderung wäre mir die Erstellung dieser Arbeit nicht möglich gewesen.

Kaiserslautern, 2009 — Dipl.-Ing. Patrik Schunk

Inhaltsverzeichnis

"Das Zeichnen ist die Sprache des Ingenieurs"
Karl Culmann (1821-1881)

Zusammenfassung

Die vorliegende Arbeit befasst sich mit den Anforderungen und der Programmarchitektur moderner objektorientierter 3D-CAD-Programme im Bauwesen.

Die digitale Revolution des letzten Jahrzehnts hatte auch Auswirkungen auf die Bauplanung. Das Zeichenbrett wurde aus den Architektur- und Ingenieurbüros verbannt und die Arbeit wird in allen Bereichen durch den Computer bestimmt. In diesem Zeitraum setzten sich viele Innovationen im Bereich der CAD-Anwendungen durch. Heutige moderne CAD-Programme sind objektorientiert, verfügen über eine graphische Oberfläche und besitzen ein dreidimensionales Datenmodell. Die innovativen 3D-Datenmodelle der CAD-Programme ermöglichen nicht nur die Erstellung von Zeichnungen, sondern auch die Verwendung des Datenmodells für die Gebäudeausrüstung, Tragwerksplanung, Ausschreibung und Abrechnung des Bauwerkes. Die Programme besitzen eine mehrschichtige Architektur, welche die Möglichkeit der einfachen Änderung und Erweiterung der Anwendung bietet.

Ziel dieser Arbeit ist die Beschreibung der Programmarchitektur und der Anforderungen an die Funktionalität, die Oberfläche und das Datenmodell moderner, objektorientierter 3D-CAD-Systeme für die Verwendung im Bauwesen. Für die Demonstration einer modernen Softwarearchitektur wurde das CAD-Programm GoCAD entwickelt, in welchem verschiedene moderne Architekturmuster exemplarisch implementiert wurden.

Abstract

The resent paper concerned with requirements and software architecture of modern object-oriented 3D-CAD-Programs for building trade.

The digital revolution of the past ten years still influencing the building design today. The drafting board was banished out of architecture and engineering offices and the activity is designate in all areas by computers. Current modern CAD-Programs are object-oriented, they containing a graphic user interface and provide a three-dimensional data model. The innovative 3D data model of CAD-Progams allowed not only the creation of sketches, but also use of the data model for building services, planning of structural framework, tender and settlement of the building. The programs use a multilayer architecture, which offer the possibility of simple change and expansion of the application.

The destination of this dissertation is describing the software architecture and the requirements of the functionality, the user interface and data model of modern object-oriented 3D-CAD-Systems for use in building trade. For the demonstration of a modern software architecture the CAD-Program GoCAD is developed, in which exemplary several modern design patters are implemented.

1 Einleitung

CAD-Zeichnungen sind heute viel mehr als nur einfache Linien. Zu Beginn der Entwicklung von CAD-Programmen wurde die Arbeit mit Stift und Lineal am Zeichenbrett mit dem Computer simuliert. Dieses Prinzip wurde lange beibehalten, jedoch wurde mit Fortschreiten der Programmentwicklung immer mehr davon abgewichen. Heutige CAD-Programme im Bauwesen besitzen Intelligente Objekte, verschiedene Arten von Sichten und können mit anderen Programmen aus dem Bereich des Bauwesens interagieren.

Die Voraussetzungen für diesen Fortschritt wurden mit der Weiterentwicklung in der Soft- und Hardwaretechnik geschaffen, der in den neunziger Jahren einen Höhepunkt erreichte. Mitte der neunziger Jahre standen mit dem Durchbruch des PC als Massenprodukt kostengünstige Computer zur Verfügung. Mit der Einführung von Windows95 und WindowsNT wurden Betriebssysteme mit graphischen Oberflächen zum Standard und da es sich dabei um 32Bit Systeme handelte, konnte praktisch unbegrenzter Speicher verwaltet werden. Bei der Programmierung setzte sich die Objektorientierung durch, neue Techniken wie das Internet kamen auf. Zeitgleich mit den Neuerungen änderten sich auch die Ansprüche der Anwender an die Software.

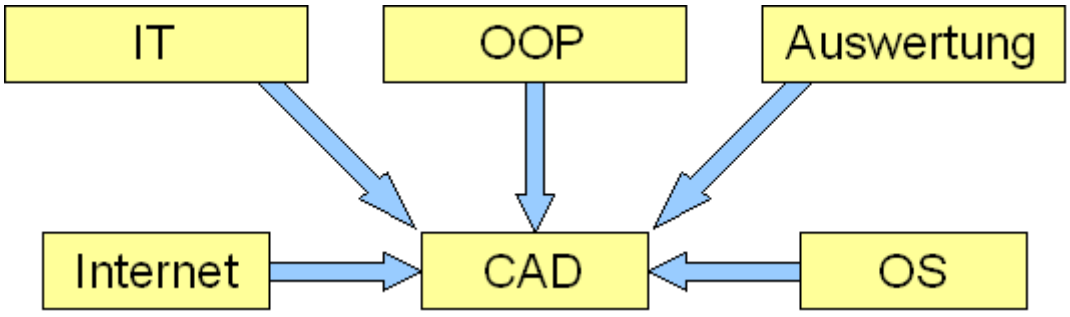


Abbildung 1.1: Einflüsse auf moderne CAD-Programme

Diese Entwicklungen hatten große Auswirkungen auf CAD-Anwendungen. Bis in die neunziger Jahre waren anspruchsvolle technische Programme wie CAD- und FEM-Programme meist teuren Workstations auf Unixbasis vorbehalten. Mit der Einführung von Windows95 und NT wurden die CAD-Programme, die meist noch ohne graphische Oberfläche liefen,

auf Windows umgestellt, und bekamen eine graphische Oberfläche. Teilweise erfolgte auch die Umstellung auf eine objektorientierte Programmstruktur. Die Umstellung auf Windows bedeutete einen weiteren Einschnitt, da die meisten Hersteller von Bausoftware ihre UNIX-Varianten aufgaben und sich auf Windows konzentrierten.

Im Rahmen dieser Arbeit wird die Weiterentwicklung der CAD-Programme im Bauwesen infolge des Fortschritts in der Hard- und Softwaretechnologie betrachtet. Es werden die Auswirkungen des Fortschrittes in der Technik auf CAD-Programme untersucht und dargestellt. Hierbei werden verschiedene Fragen gestellt.

- Welche Anforderungen werden an ein modernes CAD-Programm im Bauwesen gestellt?
- Welche Funktionalitäten besitzt ein modernes CAD-Programm?
- Wie ist ein CAD-Programm aufgebaut?
- Wie sieht das Datenmodell für ein CAD-Programm aus?
- Wie lassen sich die Objekte in einem CAD-Programm erzeugen?
- Wie lässt sich die Darstellung des Datenmodells in verschiedenen Sichten flexibel steuern?

Zur Beschreibung eines modernen CAD-Programmes wird sich auf das Programm Vicado bezogen. Dieses Programm wurde gewählt, weil es eines der wenigen kompletten Neuentwicklungen der letzten Jahre ist, und von Anfang an auf die aktuelle Softwaretechnologie ausgerichtet ist, während die meisten anderen Programme auf eine lange Entwicklung zurückschauen und entprechenden Ballast aus früheren Zeiten mitführen.

Zur Demonstration einer modernen Softwarearchitektur wurde das Programm GoCAD entwickelt, in welchem die in der Arbeit erläuterten Konzepte umgesetzt werden.

1.1 Aufbau der Arbeit

Die gesamte Arbeit ist in vier Teile unterteilt. Im ersten Teil, mit den Kapiteln 2 bis 5 werden verschiedene Grundlagen erläutert. Die UML wird in späteren Teilen der Arbeit zur Modellierung von Klassen und der Beziehung von Klassen untereinander verwendet. Desweiteren werden verschiedene Entwurfs- und Architekturmuster, die für die Arbeit verwendet werden, vorgestellt.

Der zweite Teil umfasst die Kapitel 7 bis 15. In diesem Teil wird der Aufbau und die

Funktionalität eines modernen CAD-Programmes im Bauwesen erläutert. Dies umfasst den Aufbau der Oberfläche, das Geometriemodell und den Datenaustausch.

In den Kapiteln 16 bis 20 im dritten Teil werden grundlegende Architekturprinzipien moderner CAD-Systeme erläutert.

Dies beginnt in Kapitel 16 mit dem Aufbau eines modernen CAD-Programmes.

Mit Hilfe des Builderkonzeptes Kapitel 18 wird eine Möglichkeit vorgestellt, die verschiedensten Objekte auf eine einheitliche Art und Weise, nur in Abhängigkeit von der Anzahl der für die Konstruktion benötigten Punkte zu erzeugen.

Das Painterkonzept in Kapitel 19 beschreibt eine view- und objektunabhängige Methode der Darstellung des Datenmodells in verschiedenen Ansichten. Die Darstellung wird von den Objekten gelöst und ermöglicht eine flexible Steuerung der Modelldarstellung.

Mit dem in Kapitel 20 erläuterten Referenzkonzept werden Möglichkeiten zur Verknüpfung einzelner Bauteile durch Referenzen untereinander vorgestellt.

Der abschließende vierte Teil umfasst die Kapitel 21 bis 24. In ihm wird exemplarisch die Umsetzung der im zweiten Teil erläuterten Entwurfs- und Architekturmuster mit dem CAD-Programm GoCAD beschrieben. GoCAD ist ein objektorientiertes CAD-Programm, das im Rahmen dieser Arbeit entwickelt wurde.

In Kapitel 21 wird der mehrschichtige Programmaufbau von GoCAD erläutert. Im darauf folgenden Kapitel wird auf das Datenmodell von GoCAD eingegangen. Es werden die einzelnen von der Basisklasse abgeleiteten Klassen erläutert und die Funktionalität der einzelnen Objekte vorgestellt. Des Weiteren wird die von GoCAD verwendete Datenverwaltung erläutert. Es wurde für das Programm ein einfaches Datenmodell entwickelt, das neben 2D-Elementen wie Linie und Kreis auch Bauobjekte wie Wand und Stütze besitzt. Die Speicherung der Geometrie der Objekte wird durch ein Brep-Modell realisiert.

Für einzelne Bauteile wurden mehrere Builder implementiert. Auf diese wird in Kapitel 23 eingegangen. Die einzelnen Builder unterscheiden sich in der Art der zu erzeugenden Bauteile und in der Anzahl der für die Konstruktion benötigten Punkte.

Das letzte Kapitel 24 befasst sich mit dem Painterkonzept und den damit verbundenen Möglichkeiten zur objekt- und viewabhängigen Darstellung der Bauteile.

1.2 Einsatz von CAD-Programmen im Bauablauf

CAD-Programme werden heute während des gesamten Planungs- und Bauablaufes im Bauwesen eingesetzt. Der Ablauf der Planung wird durch die HOAI (hoa06) (Honorarordnung für Architekten und Ingenieure) beschrieben. Diese unterteilt die Planung in 9 Phasen:

1. Grundlagenermittlung
2. Vorplanung
3. Entwurfsplanung
4. Genehmigungsplanung
5. Ausführungsplanung
6. Vorbereitung der Vergabe
7. Mitwirkung bei der Vergabe
8. Objektüberwachung
9. Objektbetreuung und Dokumentation

In fast jeder einzelnen dieser Phasen können heute CAD-Programme verwendet werden. Nach der Grundlagenermittlung können die ersten Entwürfe für das neue Bauwerk mit CAD-Programmen erstellt werden. Dies ermöglicht dem Planer, je nach verwendetem Programm, dem Bauherrn innerhalb kurzer Zeit die ersten Visualisierungen des Gebäudes zu präsentieren. Ausgehend von der Vorplanung können aus den erstellten Plänen durch Verfeinerung die weiteren Pläne der Entwurfsplanung, Genehmigungsplanung und Ausführungsplanung erstellt werden. Die einzelnen Planungen unterscheiden sich in ihrer Detaillierung, der Art der Darstellung und den Vermaßungen.

In der 6. Phase der „Vorbereitung der Vergabe“ werden die Mengen ermittelt und zusammengestellt als Grundlage für die Leistungsbeschreibung. Diese Mengenermittlung beruht auf den erstellten Plänen. Bei modernen CAD-Programmen gehen die Möglichkeiten weiter. Die Ausschreibungsprogramme können auf das Datenmodell des CAD-Programmes über Schnittstellen zugreifen und aus diesem die Mengen automatisch ermitteln und die Leistungsbeschreibung erstellen. Bei der Objektüberwachung, der 8. Leistungsphase, kann mit Hilfe des Datenmodells der Bauzeitplan erstellt, die einzelnen an der Ausführung des Gebäudes beteiligten Handwerker koordiniert und der Baufortschritt dokumentiert werden. Festgestellte Baumängel und Abweichungen von der Planung können direkt mit einem CAD-Programm erfasst werden. In der letzten und abschließenden 9. Phase der „Objektbetreuung und Dokumentation“ werden die erstellten Pläne systematisch zusammengestellt und dem Bauherrn übergeben.

Die bisher beschriebenen Phasen betreffen die Leistungen des Architekten. Schon in der ersten Phase werden andere an der Planung Beteiligte, wie Tragwerksplaner, Bauphysiker und TGA-Ingenieure bestimmt, die auch CAD-Anwendungen bei der Planung einsetzen. Es findet ein Informationsaustausch zwischen den einzelnen Planern statt. Das Datenmodell des Architekten kann dem Tragwerksplaner als Eingabe für FEM- und Statikprogramme oder zur Erstellung der Schal- und Bewehrungspläne dienen. Der Bauphysiker kann aus dem Datenmodell des CAD-Programmes die benötigten Informationen ermitteln um die erforderlichen Wärmeschutzberechnungen durchzuführen.

Teil I

Grundlagen

2 Unified Modeling Language - UML

Die Unified Modeling Language - vereinheitlichte Modellierungssprache wurde von der von der Object Management Group (OMG) entwickelt und standardisiert. UML ist eine Beschreibungssprache, um Strukturen und Abläufe in objektorientierten Softwaresystemen darzustellen.

Als Väter der UML gelten Grady Booch, Ivar Jacobson und James Rumbaugh. Zu Beginn der 90er-Jahre waren sie Protagonisten der objektorientierten Programmierung und hatten bereits ihre eigenen Systeme entwickelt. Als sie zusammen beim Unternehmen Rational Software beschäftigt waren, entstand der Ansatz, die verschiedenen Notationssysteme strukturiert zusammenzuführen.

1997 wurde die UML von der OMG als Standard akzeptiert und wird seit dem von ihr weiterentwickelt. Im Juni 2003 wurde die jüngste Version der UML, die Unified Modeling Language 2.0 oder UML2, von der OMG als Entwurf veröffentlicht, im März 2005 wurde sie verabschiedet.

Die UML 2.0 besteht aus drei Teilspezifikationen. Der erste Teil, die *Infrastructure Specification* ist die Basis der UML2. In ihm werden die am häufigsten verwendeten Elemente und Modellelemente beschrieben, die die restlichen Modellelemente spezialisieren.

Die Infrastructure Specification spezifiziert Konzepte wie die Klasse, die Assoziation oder die Multiplizität eines Attributs.

Der zweite Teil, die *Superstructure Specification* basiert auf der Infrastructure Specification und definiert die Modellelemente für bestimmte Einsatzzwecke. Zu den Konzepten, die in diesem Teil beschrieben werden, gehören der Anwendungsfall, die Aktivität oder der Zustandsautomat.

Die *Object Constraint Language* ist der dritte Teil der UML 2.0. In ihm wird die Object Constraint Language 2.0 (OCL2) definiert.

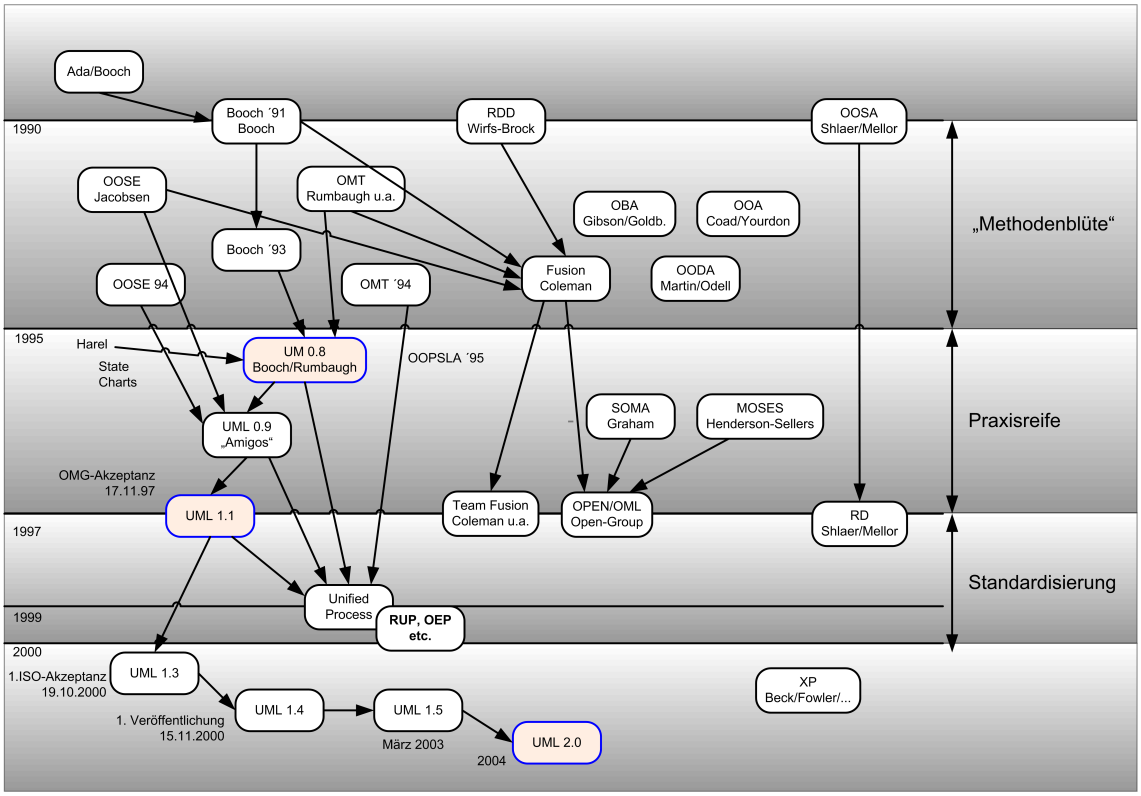


Abbildung 2.1: Historie der objektorientierten Methoden und Notationen

2.1 UML Diagrammtypen

Die Diagramme der UML2 lassen sich in zwei Gruppen unterteilen. Diese umfassen die Strukturdiagramme und Verhaltensdiagramme.

Bei den Strukturdiagrammen werden sechs verschiedene Diagrammarten unterschieden.

- Klassendiagramm
- Objektdiagramm
- Komponentendiagramm
- Kompositionsstrukturdiagramm
- Verteilungsdiagramm
- Paketdiagramm

Die Verhaltensdiagramme unterteilt die UML2 in sieben verschiedene Typen.

- Anwendungsfalldiagramm
- Zustandsdiagramm

- Aktivitätsdiagramm
- Sequenzdiagramm
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- Zeitverlaufsdiagramm

Dazu kommen noch die Kollaborationsdiagramme. Sie stellen einen abstrakten strukturellen Zusammenhang z.B. zur Dokumentation von Entwurfsmustern dar. Die im Folgenden benutzten Diagrammtypen werden näher erläutert.

2.1.1 Klassendiagramm

Mit Hilfe der Klassendiagramme werden Klassen mit ihren

- Methoden (Operationen/Funktionen)
- Attributen (Eigenschaften/Zustände)
- Beziehungen (Assoziation/Relation)

beschrieben. Die einfachste Darstellung für eine Klasse ist ein Rechteck. In dem Rechteck steht fett gedruckt der Namen der Klasse (Abb. 2.2).

Abbildung 2.2: Notation für eine Klasse

Neben dem Namen können optional noch die Methoden und Attribute der Klasse angegeben werden. Diese werden dann untereinander und von Klassennamen durch einen horizontalen Strich getrennt (Abb. 2.3)

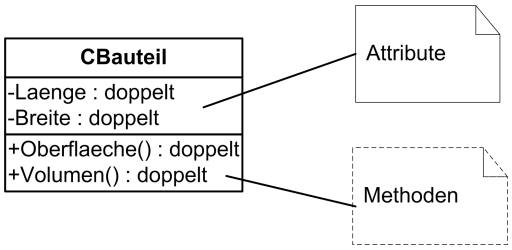


Abbildung 2.3: Notation für eine Klasse mit Methoden und Attributen

Die Attribute können neben den Namen auch noch Informationen über ihren Typ, die Sichtbarkeit, einen Initialwert und Zusicherungen enthalten. Bei dem Typ handelt es sich um eine Klasse oder einen primitiven Datentyp wie int und double.

Mit einer Zusicherung kann der Wertebereich für das Attribut eingeschränkt werden, indem ein Gültigkeitsbereich oder eine Aufzählung möglicher Werte in geschweiften Klammern hinter dem Attribut angegeben wird.

Hinter den Namen der Methoden stehen immer zwei runde Klammern. In ihnen werden die Parameter angegeben. Der Typ des Parameters steht durch einen Doppelpunkt getrennt vor dem Namen des Parameters. Mehrere Parameter werden durch Kommas getrennt.

Die Sichtbarkeit der Attribute und Methoden wird durch die folgenden Symbole beschrieben:

\+ **public** Jede Systemkomponente hat uneingeschränkten Zugriff.

- private Nur die Klasse selbst kann darauf zugreifen.

\# **protected** Die Klasse und die abgeleiteten Klassen können darauf zugreifen.

2.1.2 Parametrisierte Klassen

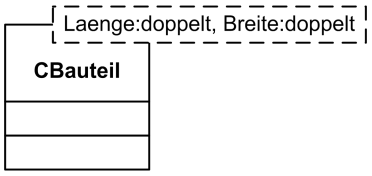


Abbildung 2.4: Notation einer parametrisierten Klasse

Die Notation der parametrisierten Klasse entspricht weitgehend der der Klasse. Zusätzlich besitzt die parametrisierte Klasse rechts oben ein Rechteck mit einer gestrichelten Linie. In diesem Rechteck werden der Typ und Name der Parameter angegeben. Name und Typ werden durch einen Doppelpunkt getrennt, mehrere Parameter durch Strichpunkte.

2.1.3 Generalisierung

Die Generalisierung wird durch einen gerichteten Pfeil mit leerer Spitze dargestellt. Die Spitze zeigt auf die Elternklasse. Abbildung 2.5 zeigt die Vererbungshierarchie der Klassen

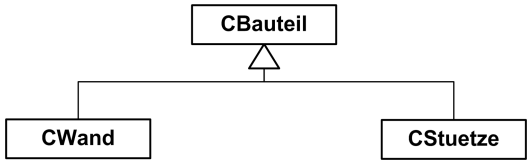


Abbildung 2.5: Notation für eine Vererbung

CBauteil, CWand und CStuetze. Die Klassen CWand und CStuetze sind beide von der Klasse CBauteil abgeleitet.

In Bild 2.6 ist ein Beispiel für eine Mehrfachvererbung dargestellt. Die Klasse CWandBuilder_2P ist von der Klasse CWandBuilderFamilie und der Klasse CZweiPBuilder abgeleitet. Sie besitzt die Methoden und Eigenschaften beider Basisklassen.

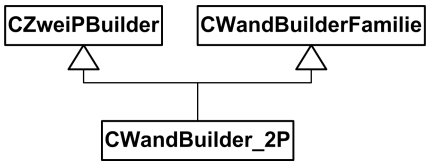


Abbildung 2.6: Notation für eine Mehrfachvererbung

2.1.4 Kommentar

Abbildung 2.7: Notation eines Kommentars

Kommentare werden durch ein Rechteck mit Eselsohr symbolisiert. Sie dienen zur Beschreibung des Diagramms.

2.2 Beziehungselemente

Die Abhängigkeiten und Beziehungen von Klassen untereinander werden mit Hilfe von Beziehungselementen dargestellt. Neben der Vererbung wird noch zwischen den nachfolgenden Beziehungselementen unterschieden.

Abbildung 2.8: Notation für eine Assoziation

2.2.1 Assoziation

Unter einer Assoziation versteht man die Beziehung (Relation) zwischen zwei Klassen. An der Assoziation können neben Klassen auch noch beliebige Typen beteiligt sein. Assoziationen sind mit einem eindeutigen Namen zu versehen. Die Anbindung einer Assoziation an eine Klasse wird als Assoziationsende bezeichnet. Die Darstellung einer Assoziation erfolgt durch einen durchgezogenen Strich. Zusätzlich kann an den Assoziationsenden eine Multiplizität angegeben werden. Über die Multiplizität wird angegeben, wie oft die Klasse mit der anderen Klasse assoziiert werden kann. Hierbei sind

- eine bestimmte Anzahl: 1
- ein Bereich: 2...5
- eine unbegrenzte Anzahl: *

möglich. Liegt das Minimum des Bereiches bei Null, so ist die Beziehung optional. Abbildung 2.9 zeigt die Assoziation zwischen einer Wand und einem Fenster. Während in der Wand unbegrenzt viele Fenster vorkommen können (nur begrenzt durch die Fläche der Wand), gehört ein Fenster immer nur zu einer Wand.

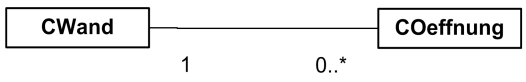


Abbildung 2.9: Notation für eine Assoziation mit Multiplizität

Ein Sonderfall der Assoziation ist die gerichtete Assoziation (Abb. 2.10). Bei ihr verläuft die Beziehung nur in einer Richtung. Die Richtung wird durch einen Pfeil am Assoziationsende angezeigt.

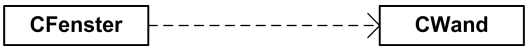


Abbildung 2.10: Notation für eine gerichtete Assoziation

2.2.2 Aggregation

Eine Aggregation ist eine Assoziation, bei der eine Klasse Bestandteil einer anderen Klasse, ein Teil des Ganzen ist. Die Aggregation wird durch einen Strich mit einer nicht ausgefüll-

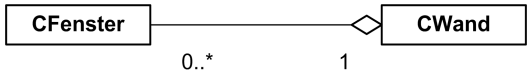


Abbildung 2.11: Notation für eine Aggregation

ten Raute am „Ganzen“ dargestellt. Bild 2.11 zeigt die Beziehung zwischen einem Fenster und einer Wand als Aggregation.

2.2.3 Komposition

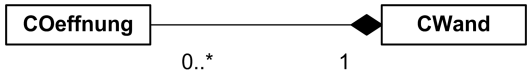


Abbildung 2.12: Notation für eine Komposition

Die Komposition ist eine Sonderform der Aggregation. Wie die Aggregation ist die Komposition auch ein Teil des Ganzen, jedoch ist das Teil existenzabhängig vom Ganzen. Abbildung 2.12 zeigt die Abhängigkeit einer Fensteröffnung von einer Wand. Während ein Fenster auch ohne eine Wand existieren kann, benötigt die Fensteröffnung eine Wand, in der sie sich befindet. Die Komposition wird ebenfalls durch einen Verbindungsstrich mit einer Raute dargestellt, diese ist im Gegensatz zur Aggregation jedoch gefüllt.

3 Entwurfmuster

In den 70er Jahren des vergangenen Jahrhunderts entwickelte sich die Idee der Entwurfsmuster in der Architektur als eine Sammlung von Mustern von Lösungen für verschiedene Problemstellungen. Mit Hilfe der Entwurfsmuster werden komplexe Architekturaspekte und -strukturen in Muster unterteilt und verknüpft.

Die Promotion von Erich Gamma im Jahr 1991 an der ETH Zürich gehört zu den ersten Veröffentlichungen über die Verwendung von Entwurfsmustern in der Softwareentwicklung. Er übertrug die Idee der Entwurfsmuster von der Architektur in die Softwareentwicklung. 1995 erschien mit "Design Patterns"(GHJV04) das Standardwerk zu dem Thema Entwurfsmuster von den Autoren Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Mehrere Entwurfsmuster, die für diese Arbeit von Bedeutung sind, werden im Folgenden näher betrachtet und erläutert. Die Entwurfsmuster werden in drei Gruppen eingeteilt:

- Erzeugungsmuster
  - Klassenmuster
    - Fabrikmethode (Factory Method, Virtual Constructor)
  - Objektmuster
    - Abstrakte Fabrik (Abstract Factory, Kit)
    - Erbauer (Builder)
    - Prototyp (Prototype)
    - Einzelstück (Singleton)
- Strukturmuster
  - Klassenmuster
    - Adapter (Adapter, Wrapper)
  - Objektmuster
    - Adapter (Adapter, Wrapper)

  - Brücke (Bridge, Handle/Body)
  - Kompositum (Composite)
  - Dekorierer (Decorator, Wrapper)
  - Fassade (Facade)
  - Fliegengewicht (Flyweight)
  - Stellvertreter (Proxy, Surrogate)

- Verhaltensmuster
  - Klassenmuster
    - Interpreter (Interpreter)
- Schablonenmethode (Template Method)
  - Objektmuster
    - Zuständigkeitskette (Chain of Responsibility)
    - Kommando (Befehl, Command, Action, Transaction)
    - Iterator (Iterator, Cursor)
    - Vermittler (Mediator)
    - Memento (Memento, Token)
    - Beobachter (Observer, Dependents, Publish-Subscribe, Listener)
    - Zustand (State, Objects for States)
    - Strategie (Strategy, Policy)
    - Besucher (Visitor)
    - Plugin (Plugin)

Bei der Verwendung von Entwurfsmustern werden meist verschiedene Muster kombiniert.

3.1 Entwurfsmuster Strategie

Erich Gamma stellt in seiner Promotion das Entwurfsmuster Strategie (engl. Policies oder Strategy) ab Seite 138 vor. Dieses Entwurfsmuster wird auch in dem Buch „Entwurfsmuster“ der Gang of Four ab Seite 373 behandelt. Von der Gang of Four wird der Zweck der Strategie wie folgt definiert:

Es gibt mehrere Möglichkeiten Klassen so zu gestalten, dass diese flexibel sind und einfach durch andere Klassen (Klienten) modifiziert werden können. Beim White-Box-Vorgehen werden Design und Implementationsentscheidungen dem Klienten als überschreibbare Methode zur Verfügung gestellt.

Definition 1: Strategie

"Definiere eine Familie von Algorithmen, kapsele jeden Einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es den Algorithmus unabhängig von ihm nutzenden Klienten zu variieren."

Eine zweite Möglichkeit sind Komponenten in der Form von Black-Box-Klassen. Bei der Verwendung von Black-Box-Klassen werden Designentscheidungen vom Objekt abgespalten und als eigene Klasse implementiert. Durch diese Strategieklassen kommt es zu einer Trennung von einem Mechanismus und der zur Verwirklichung verwendeten Strategie. Für jede Strategie wird eine eigene Klasse abgeleitet.

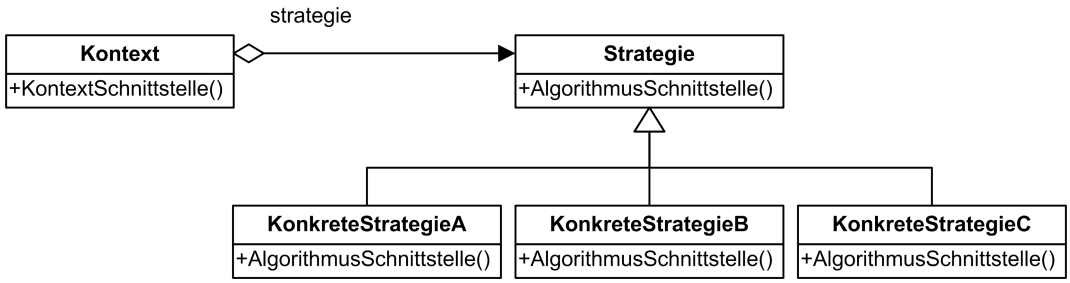


Abbildung 3.1: Struktur des Strategiemusters

Nach (GHJV04) Seite 374ff soll das Entwurfsmuster Strategie verwendet werden, wenn

- sich viele verwandte Klassen nur in ihrem Verhalten unterscheiden. Strategieobjekte bieten eine Möglichkeit, eine Klasse mit einer von mehreren möglichen Verhaltensweisen zu konfigurieren
- unterschiedliche Varianten eines Algorithmus benötigen werden. Es können zum Beispiel Algorithmen definiert werden, die unterschiedliche Vor- und Nachteile in der Geschwindigkeit und im Speicherplatzverbrauch besitzen. Strategien können verwendet werden, wenn diese Varianten als eine eigenständige Klassenhierarchie von Algorithmen implementiert werden.
- ein Algorithmus Daten verwendet, die Klienten nicht bekannt sein sollen. Verwenden Sie das Strategiemuster, um zu vermeiden, dass Sie komplexe algorithmenspezifische Datenstrukturen offen legen müssen.
- eine Klasse unterschiedliche Verhaltensweisen definiert und diese als mehrfache Bedingungsanweisungen in ihren Operationen erscheinen. Statt nun viele Bedingungs-

anweisungen zu verwenden, können Sie zusammenhängende Zweige von Bedingungsanweisungen in eine eigene Strategieklasse verlagern.

3.2 Entwurfsmuster Fliegengewicht

Das Entwurfsmuster Fliegengewicht wird in dem Buch „Entwurfsmuster“ (GHJV04) Seite 223ff behandelt, der Zweck dieses Entwurfsmusters ist wie folgt angegeben:

Definition 2: Fliegengewicht

Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können.

Fliegengewichte werden genutzt, um von Objekten Teile abzuspalten und in gemeinsam genutzt Objekte auszulagern. Der Vorteil der Fliegengewichte liegt in der Reduzierung des Speicheraufkommens, da der ausgelagerte Teil von mehreren Objekten gemeinsam genutzt wird und nicht für jedes Objekt alleine vorhanden sein muss. Bei den Fliegengewichten wird zwischen intrinsischem (innerem) und extrinsischem (äußerem) Zustand unterschieden. Der intrinsische Zustand ist vom Kontext des Fliegengewichtes unabhängig und wird in diesem gespeichert. Der extrinsische Zustand hängt vom Kontext ab und wechselt mit ihm. Ein mögliches Einsatzgebiet des Fliegengewichtes sind Formatierungsangaben.

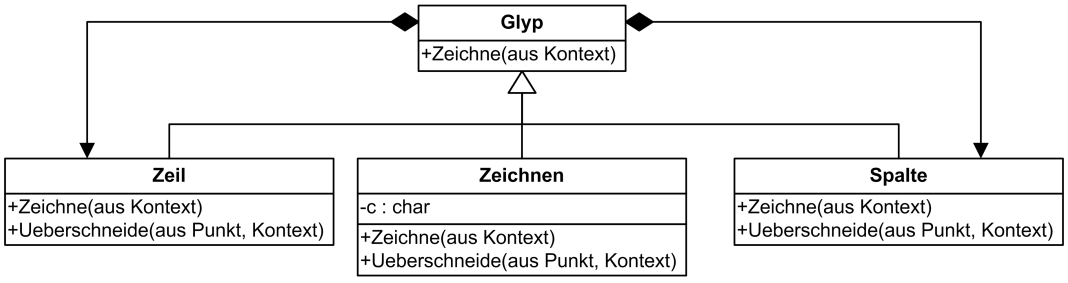


Abbildung 3.2: Klassenstruktur Fliegengewichtmuster

Abb.3.2 zeigt die Klassenstruktur eines Fliegengewichtsmusters als Formatierer in einem Textverarbeitungsprogramm. Glyph ist die abstrakte Basisklasse verschiedener graphischer Objekte.

Ein Fliegengewicht, das nur einen Buchstaben repräsentiert, kennt nur diesen. Die Position und der Zeichensatz sind dem Fliegengewicht nicht bekannt. Die weiteren Informationen

werden, wenn benötigt, dem Fliegengewicht übergeben. Der Zeilenglyph weiß wo seine Kindobjekte gezeichnet werden sollen, so dass diese horizontal angeordnet werden und versorgt die einzelnen Glyphen mit der Position beim Aufruf der Zeichenfunktion.

Der Klient übergibt der Fliegengewichtfabrik den Schlüssel und fordert das passende Fliegengewicht an. Die Fliegengewichtfabrik durchsucht die vorhandenen Fliegengewichte nach dem passenden Fliegengewicht. Wenn kein zugehöriges Fliegengewicht existiert, wird ein neues Fliegengewicht von der Fabrik erzeugt. Die Fliegengewichtfabrik gibt anschließend das zugehörende Fliegengewicht an den Klienten zurück, der dann auf diesen zugreifen kann.

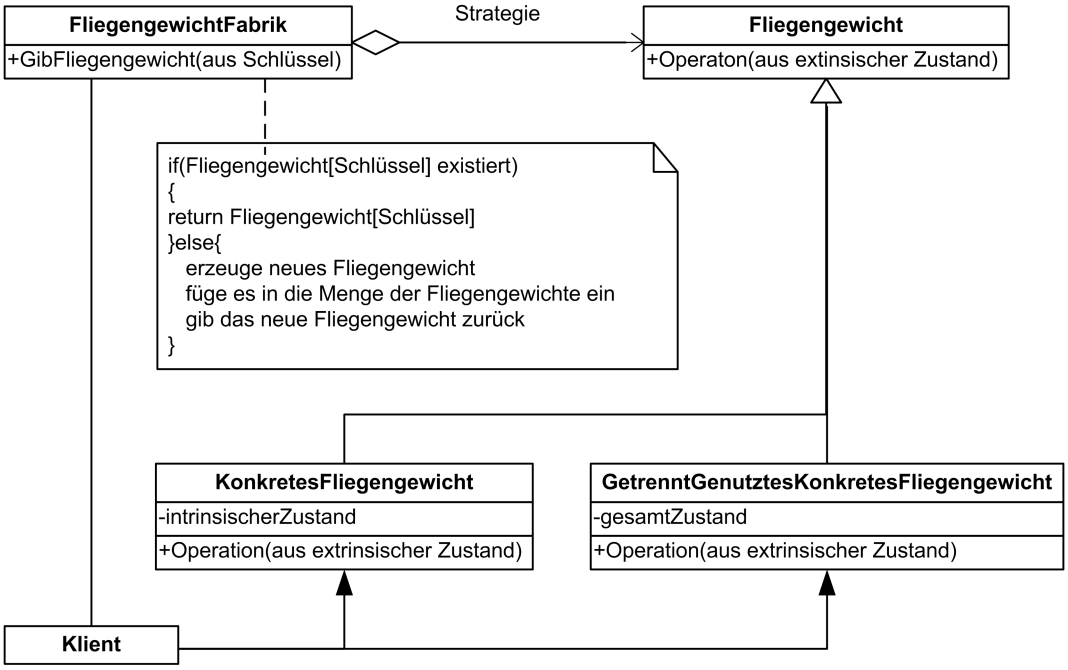


Abbildung 3.3: Struktur des Fliegengewichtmusters

In (GHJV04) wird empfohlen Fliegengewichte nur zu verwenden, wenn die folgenden Bedingungen eingehalten sind:

- Eine Anwendung verwendet eine große Menge von Objekten.
- Die Speicherkosten sind allein aufgrund der Anzahl von Objekten groß.
- Ein Großteil des Objektzustandes kann in den Kontext verlagert und somit extrinsisch gemacht werden.
- Viele Gruppen von Objekten können durch relativ wenige gemeinsam genutzte Objekte ersetzt werden, sobald einmal der extrinisische Zustand entfernt wurde.
- Die Anwendung hängt nicht von der Identität der Objekte ab. Da Fliegengewichtob-

jekte gemeinsam genutzt werden, liefern Identitätstests bei konzeptuell unterschiedlichen Objekten ein positives Ergebnis zurück.

Fliegengewichte können infolge des Aufsuchen des extrinsischen Zustandes zu Laufzeitkosten führen, dies kann aber durch den Speicherplatzgewinn aufgewogen werden, der umso größer ist, je mehr Objekte sich ein Fliegengewicht teilen. Der Speicherplatzgewinn ist nach (GHJV04) abhängig:

- von der Reduzierung der Gesamtmenge von Objekten, die sich aufgrund gemeinsamer Nutzung ergeben;
- von der Größe des intrinsischen Zustandes pro Objekt;
- davon, ob der extrinsische Zustand berechnet oder gespeichert wird.

Der Speichergewinn in einer Anwendung ist am größten, wenn Objekte einen großen Teil des intrinsischen und auch des extrinsischen Zustandes gemeinsam verwenden und der extrinsische Zustand berechnet werden kann. In diesem Fall wird Speicherplatz gegen Rechenleistung eingetauscht.

3.3 Entwurfsmuster Prototyp

Der Prototyp ist ein Entwurfsmuster zur Erstellung von Objekten und gehört zu den Erzeugungsmustern. Der Zweck des Entwurfsmusters Prototyp wird in dem Buch „Entwurfsmuster“ (GHJV04) wie folgt definiert:

Definition 3: Prototyp

"Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototypen"

Prototypen erzeugen neue Objekte durch Klonen oder kopieren eines Objektes, dieses Objekt wird Prototyp genannt.

Für die Erzeugung eines neuen Objektes bekommt ein Klient den Prototyp des Objektes, das geklont werden soll, als Parameter übergeben. Der Klient fordert den Prototypen auf, sich selbst zu kopieren und fügt anschließend das neue Objekt in das Dokument ein.

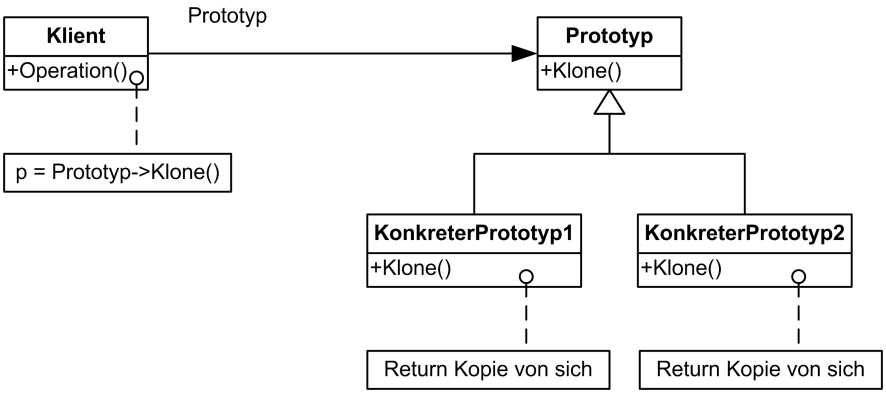


Abbildung 3.4: Struktur des Prototypmuster

In der Objekthierarchie der Prototypen wird in der Basisklasse der Prototypen eine Schnittstelle deklariert, um sich selbst zu klonen. Die abgeleiteten Klassen implementieren eine Operation, um sich selbst zu klonen.

Nach (GHJV04) sind Prototypen anwendbar, wenn ein System unabhängig davon sein soll, wie die Produkte erzeugt, zusammengesetzt und repräsentiert werden sollen, und

- wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden, beispielsweise durch dynamisches Laden, oder
- um zu vermeiden, eine Klassenhierarchie von Fabriken zu erstellen, die parallel zur Klassenhierarchie der Produkte verläuft, oder
- wenn Exemplare einer Klasse nur wenige unterschiedliche Zustandskombinationen haben können. Es ist möglicherweise bequemer, eine entsprechende Anzahl von Prototypen einzurichten und sie zu klonen, statt die Objekte einer Klasse jedes Mal von Hand mit dem richtigen Zustand zu erzeugen.

3.4 Entwurfsmuster Builder

Der Builder gehört wie der Prototyp zu den Erzeugungsmustern. Der Zweck des Builders wird in (GHJV04) auf Seite 119 wie folgt definiert:

Ein typisches Einsatzgebiet von Buildern sind Konvertierer von Texten in andere Formate. Eine Anwendung liest ein bestimmtes Textformat ein, das in ein anderes Format konvertiert werden soll. Wenn ein Formattoken durch den Einleser erkannt wird, stellt dieser eine Anfrage an den Textkonvertierer zur Konvertierung des Textes. Das Konvertiererobjekt ist für die Ausführung der Datenkonvertierung und die Repräsentation des

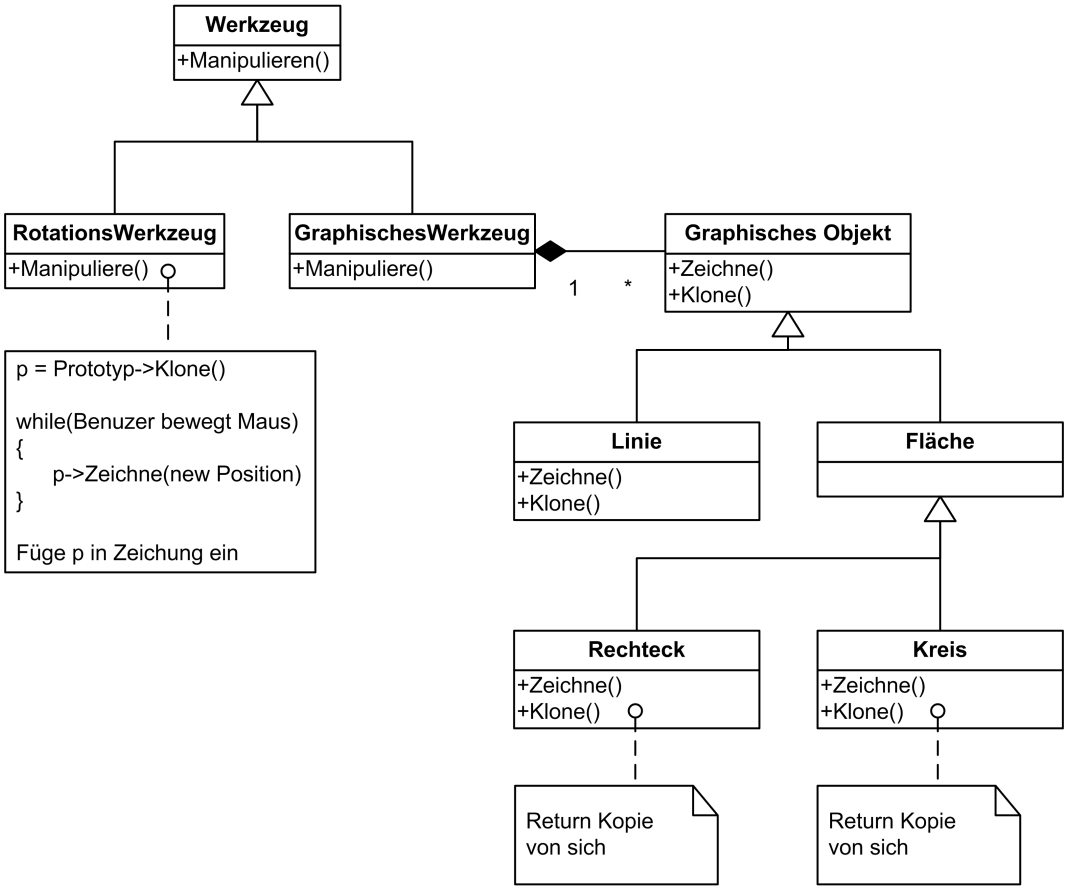


Abbildung 3.5: Entwurfsmuster Prototyp

Token in ein bestimmtes Format verantwortlich. Durch die Verwendung von Buildern ist die Anzahl der möglichen Formate unbegrenzt, dies erfordert nur, dass das Programm für jedes Format um einen Builder ergänzt wird. Die von dem Builder abgeleiteten Unterklassen sind Spezialisierungen des Builders für bestimmte Erzeugungsalgorithmen oder Objekte. Der Mechanismus für die Erzeugung und den Zusammenbau des Objektes hinter einer abstrakten Schnittstelle ist in der Erbauerklasse versteckt. Der Konverter ist von dem für das Einlesen des Dokumentes zuständigen Lesers getrennt, für Ergänzung eines neuen Konvertierers muss weder das Dokument noch der Einleser geändert werden.

Definition 4: Builder

Trenne die Konstruktion eines komplexen Objektes von seiner Repräsentation, sodass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.

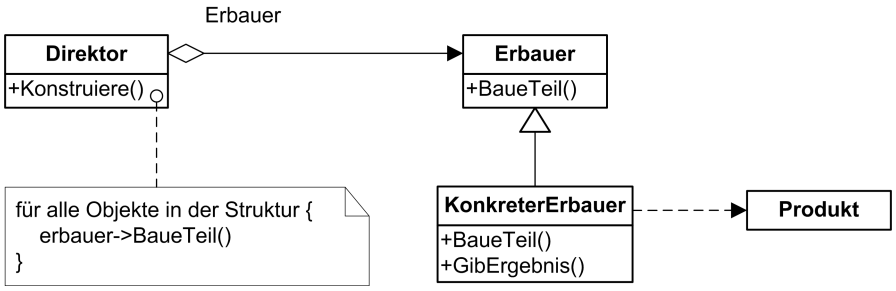


Abbildung 3.6: Struktur des Buildermuster

Die Konstruktion eines neuen Objektes geschieht durch den Direktor. Dieser verwendet dabei die Schnittstelle des Erbauers. Die Funktion `Konstruiere()` des Direktors ruft die Funktion `BaueTeile()` des „KonkreterErbauers“ auf. In der Basisklasse Erbauer ist die Funktion `BaueTeile()` als abstrakte Schnittstelle für die Erzeugung von Teilen des neuen Produktes spezifiziert. Die Basisklasse Erbauer deklariert die abstrakte Schnittstelle `BaueTeile()` für die Erzeugung des Produktes, diese wird in der abgeleiteten Klasse „KonkreterErbauer“ für ein bestimmtes Produkt und bestimmten Algorithmus definiert. Die Erbauerschnittstelle des Builders konstruiert und fügt die Teile zu einem neuen Objekt zusammen. Des Weiteren bietet der Erbauer eine Schnittstelle für die Rückgabe des neuen Produktes. Das komplexe Objekt, das gerade konstruiert wird, wird durch das Produkt repräsentiert. Der konkrete Erbauer erstellt eine interne Repräsentation des Produktes und legt den Prozess fest, durch den es zusammengesetzt wird. Hierzu gehören auch die Klassen, die die zu konstruierenden Teile definieren, einschließlich der Schnittstellen, welche die einzelnen Produktteile zum endgültigen Resultat zusammensetzen.

Der Klient erzeugt einen neuen Direktor und übergibt ihm das gewünschte Erbauerobjekt. Der Direktor informiert den Erbauer, wenn ein Teil des Produktes erzeugt werden soll. Der Erbauer bearbeitet die Anfragen des Direktors und fügt die einzelnen Teile zum Produkt hinzu. Hat der Erbauer alle benötigten Teile erhalten, informiert dieser den Klienten. Der Klient ruft die Funktion `GibErgebnis()` des Erbauers auf und erhält das erzeugte Produkt zurück.

Ein Builder soll in den folgenden Situationen verwendet werden:

- Der Algorithmus zum Erzeugen eines komplexen Objektes soll unabhängig von den Teilen sein, aus denen das Objekt besteht und wie sie zusammengesetzt werden.
- Der Konstruktionsprozess muss verschiedene Repräsentationen des zu konstruierenden Objektes erlauben.

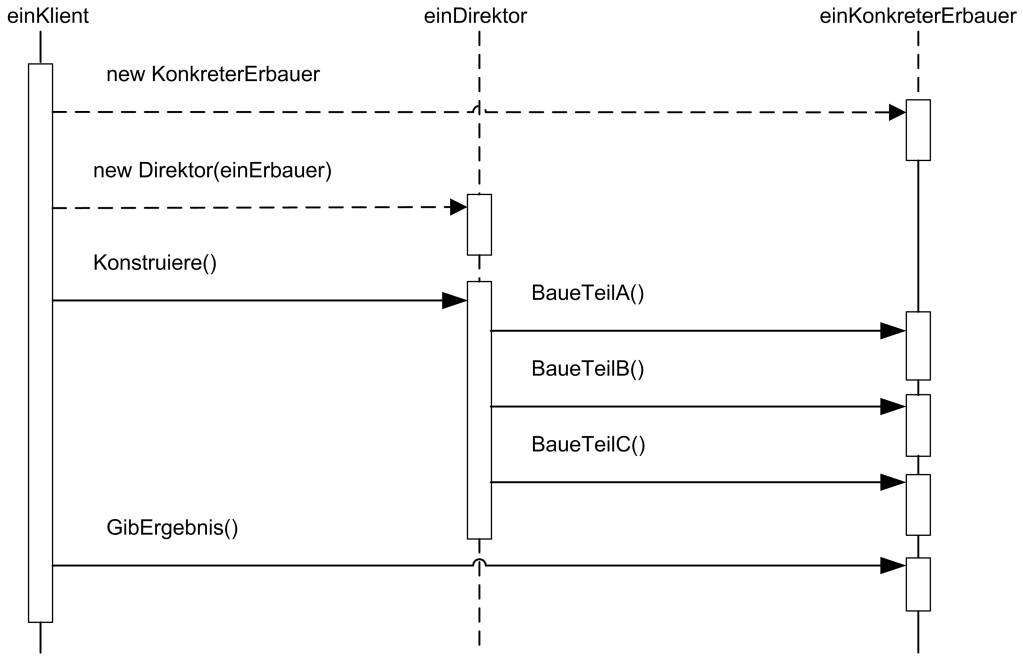


Abbildung 3.7: Entwurfsmuster Builder

3.5 Entwurfsmuster Beobachter

Das Entwurfsmuster Beobachter ist ein objektorientiertes Verhaltensmuster. Dieses wird in (GHJV04) auf Seite 287ff beschrieben. Die Aufgabe dieses Entwurfsmuster wird wie folgt angegeben:

Definition 5: Beobachter

Definiere eine 1 zu n Abhängigkeit zwischen Objekten, so dass die Änderung des Zustandes eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Wird ein System in verschiedene Klassen aufgeteilt, so muss häufig sichergestellt werden, dass die Konsistenz von miteinander in Beziehung stehenden Klassen aufrechterhalten wird. Dies soll meist nicht durch eine enge Kopplung der Klassen geschehen, da hierdurch die Wiederverwendbarkeit der Klassen eingeschränkt wird.

In Klassenbibliotheken für Benutzerschnittstellen werden oftmals die Benutzerschnittstellen von den Anwendungsdaten getrennt. Die Klassen für die Darstellung und die Anwendungsdaten können sowohl miteinander, als auch unabhängig voneinander verwendet werden.

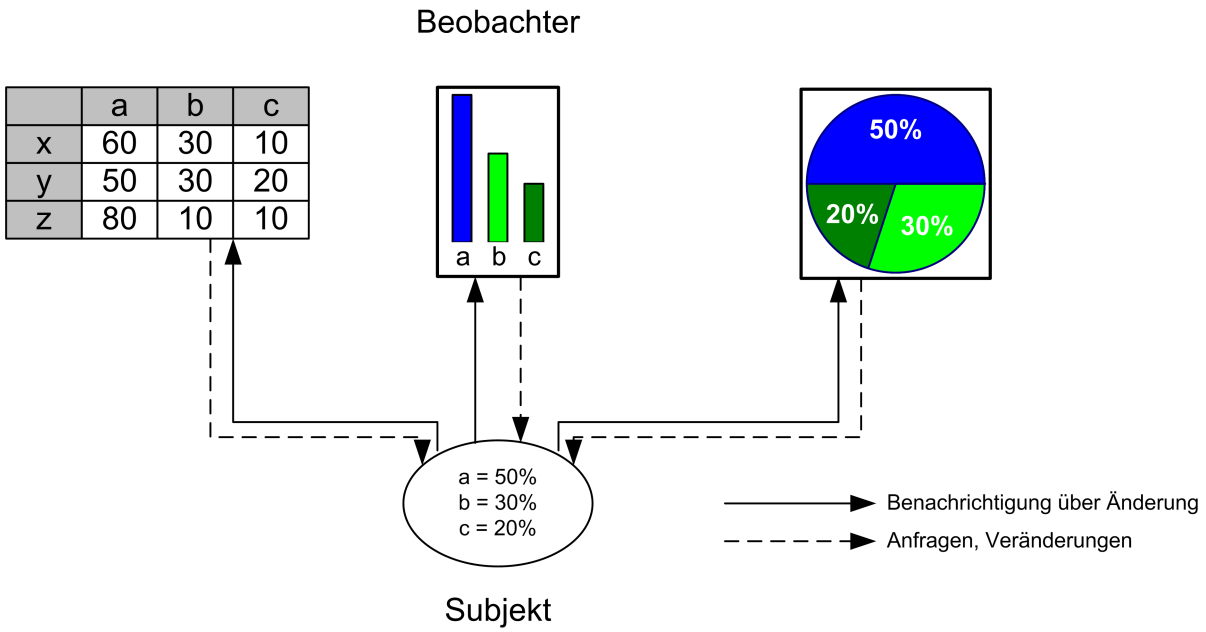


Abbildung 3.8: Schema des Entwurfsmusters Beobachter

Ein Typisches Beispiel hierfür sind Tabellenkalkulationsprogramme wie Excel. Die Daten von Excel können gleichzeitig in einer Tabelle oder in einem oder mehreren Diagrammen dargestellt werden. Die einzelnen Darstellungsformen der Daten kennen einander nicht, trotzdem verhalten sie sich so als ob sie sich kennen. Werden die Daten in der Tabelle geändert, erfolgt die automatische Anpassung der Diagramme.

Hieraus folgt, dass die Tabelle und die Diagramme vom Datenobjekt abhängig sind und über die Zustandsänderungen informiert werden. Die Anzahl der abhängigen Objekte kann unbegrenzt sein.

Das Beobachtermuster beschreibt die Implementierung dieser Objektbeziehungen. Die zentralen Bestandteile dieses Musters sind das Subjekt und die Beobachter. Das Subjekt kann eine unbegrenzte Anzahl von Beobachtern besitzen, die es bei einer Änderung benachrichtigt. Als Reaktion auf die Änderung werden die Beobachter mit dem Objekt synchronisiert.

Diese Art der Interaktion zwischen den Objekten wird auch als Publish-Subscribe bezeichnet. Das Subjekt ist der Publizierer der Nachrichten, die es aussendet ohne die Beobachter zu kennen. Eine unbegrenzte Anzahl von Beobachtern können den Empfang der Nach-

richten abonnieren.

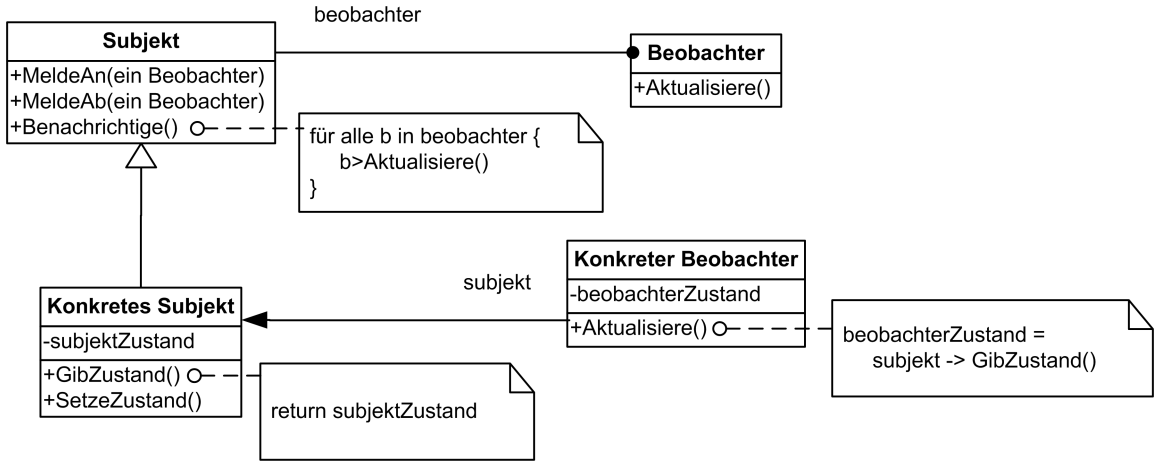


Abbildung 3.9: Struktur des Entwurfsmusters Beobachter

Die Struktur des Beobachtermusters ist in Abbildung 3.9 gezeigt. Die Bestandteile der Struktur sind:

Subjekt Das Subjekt kennt seine Beobachter. Die Anzahl der Beobachter ist unbegrenzt. Es bietet Schnittstellen zum An- und Abmelden der Beobachter.

Beobachter Der Beobachter definiert die Aktualisierungsschnittstelle für die Objekte, die über die Änderungen benachrichtigt werden.

Konkretes Subjekt Das Konkrete Subjekt speichert die für den Konkreten Beobachter relevanten Daten und benachrichtigt seine Beobachter bei Änderung des Zustandes.

Konkreter Beobachter Der Konkrete Beobachter speichert die Daten, die mit dem Subjekt in Einklang stehen sollen. Er implementiert die Aktualisierungsschnittstelle der Beobachterklasse um den Zustand mit dem Subjekt konsistent zu halten. Des Weiteren verwaltet er die Referenzen auf ein konkretes Subjekt.

Das Besuchermuster kann nach (GHJV04) 288ff zweckmäßig in den folgenden Fällen angewandt werden:

- wenn eine Abstraktion zwei Aspekte besitzt, von denen der eine von dem anderen abhängt. Die Kapselung dieser Aspekte in unterschiedlichen Objekten ermöglicht es Ihnen, sie zu variieren und sie unabhängig voneinander wieder zu verwenden.
- wenn die Änderung eines Objektes die Änderung anderer Objekte verlangt und Sie nicht wissen, wie viele Objekte geändert werden müssen.

- wenn ein Objekt in der Lage sein sollte, andere Objekte zu benachrichtigen, ohne Annahme darüber treffen zu dürfen, wer diese anderen Objekte sind. Mit anderen Worten: Sie wollen diese Objekte nicht eng miteinander koppeln.

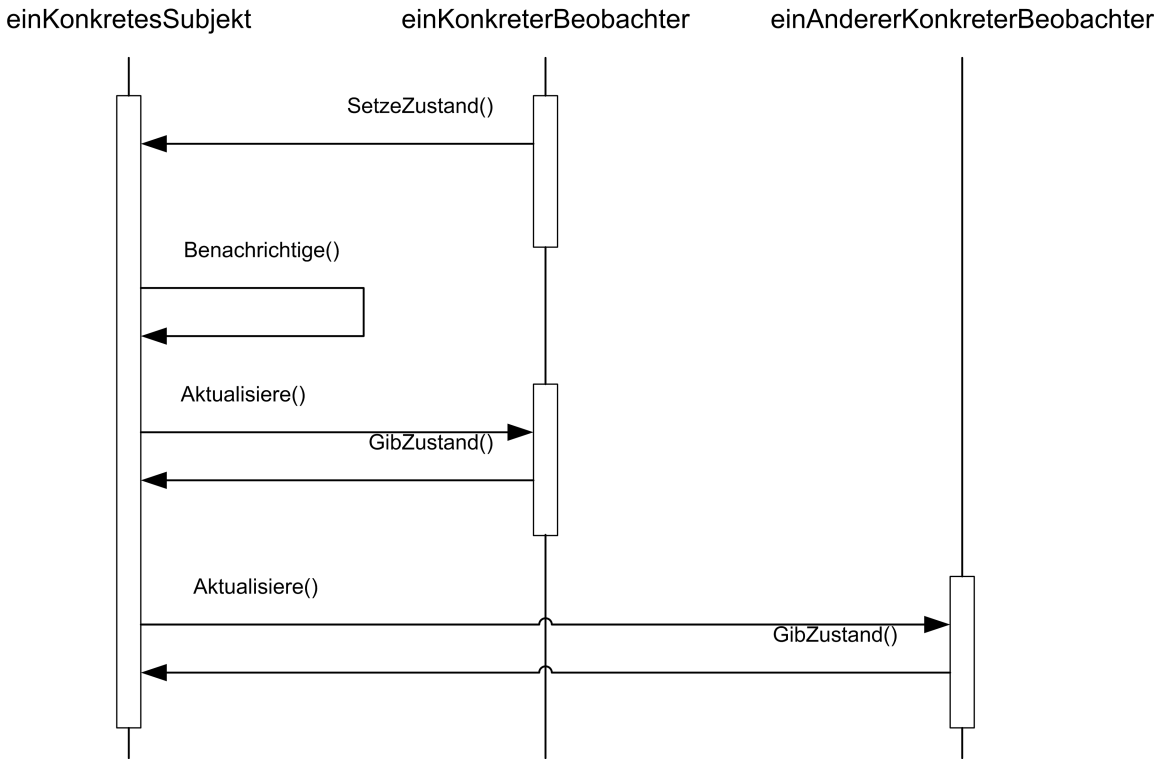


Abbildung 3.10: Interaktionsdiagramm des Beobachters

Die Interaktion des Beobachters mit dem Subjekt ist in Diagramm 3.10 dargestellt. Ein Konkreter Beobachter meldet sich beim KonkretenSubjekt an. Das KonkreteSubjekt benachrichtigt seine Beobachter, wenn sich sein Zustand ändert. Nach der Benachrichtigung des KonkretenBeobachters befragt der Beobachter das Subjekt nach Informationen um seinen Zustand mit dem des Subjektes abzugleichen.

4 Model-View-Controller

4.1 Grundlagen Model-View-Controller Konzept

Das Model-View-Controller-Konzept (MVC) wurde 1979 von Trygve Reenskaug zum ersten Mal beschrieben und in Smalltalk implementiert. Das Architekturprinzip wurde in der Zwischenzeit mehrfach variiert und hat sich mittlerweile als Standard etabliert. Der Grundgedanke des MVC-Konzeptes ist die Trennung der Präsentation von der fachspezifischen Semantik, die Anwendung wird in drei Teile aufgeteilt:

Model bezeichnet die Komponente, die das Fachwissen (Daten) enthält.

View bezeichnet die Komponente, mit der die Darstellung der Daten auf dem Bildschirm definiert wird.

Controller bezeichnet die Komponente, die für die Interaktion mit dem Benutzer zuständig ist. Er verarbeitet z. B. die Mausereignisse und Tastatureingaben.

Die drei Teile des MVC-Konzeptes sind nicht alleine funktionsfähig, sondern müssen sich gegenseitig unterstützen. Die Zusammenarbeit von Model, View und Controller ist so zu gestalten, dass das Model unabhängig von den beiden anderen Komponenten bleibt.

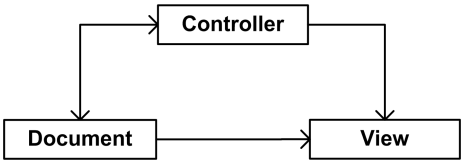


Abbildung 4.1: Schema des Document View Controller Konzeptes

Hierdurch ist es auf der einen Seite möglich, das Model unabhängig von der View und dem Controller zu entwerfen und zu implementieren. Das Modell kann auch nachträglich verändert werden, ohne dass dies Auswirkungen auf die Views hat. Auf der anderen Seite kann es mehrere unterschiedliche Views geben, ohne dass daraus Modifikationen des Models folgen müssen.

4.2 Document-View-Konzept

Die Firma Microsoft führte 1992 mit dem Microsoft c/c++ 7.0 die MFC (Microsoft Foundation Classes) eine objektorientierte Schnittstelle zur Windows-API ein (siehe (Sch94), (Kin98)). Anfang 1993 folgte die Version 2.0 der MFC mit der Version 1.0 des Visual Studio. Die Version 2.0 der MFC enthält neben Erweiterungen auch ein Application Framework auf der Grundlage des Document-View-Konzeptes. Bei dem Document-View-Konzept handelt es sich um die Umsetzung des MVC-Konzeptes durch die Firma Microsoft. Die MFC unterscheidet drei Varianten des Document-View-Konzeptes.

Single Document Interface (SDI) Das SDI ist die einfachste Umsetzung des Document-View-Konzeptes. Jedes Document, das geöffnet wird, benötigt eine eigene Instanz des Programmes. Beispiele für das SDI sind der Internetexplorer oder Wordpad.

Multiple Document Interface (MDI) Das Multiple Document Interface entspricht weitgehend dem SDI. Im Gegensatz zum Single Document Interface können beim MDI mehrere Documente in selben Programm geöffnet werden, ein mehrfaches Starten des Programmes ist nicht notwendig. Jedes Document wird hierbei in einem eigenen Unterfenster angezeigt, das sich frei im Hauptfenster platzieren lässt. Das MDI ist die unter MS-Windows meistverwendete Form des Document-View-Konzeptes.

Multiple Top-Level Windows Interface Beim Multiple Top-Level Windows Interface wird für jedes Dokument ein eigenes Hauptfenster geöffnet, auch wenn nur eine einzige Instanz der Anwendung läuft. Man kann das MTI als eine Mischung zwischen SDI und MDI ansehen. Nach außen tritt es wie ein Single-Document-Interface Programm auf, intern wird jedoch nur eine Instanz des Programmes gestartet. Verwendet wird dieses Modell u. a. in Office 2003.

4.3 Aufbau des Document-View-Konzeptes

4.3.1 Document

Der zentrale Bestandteil des Document-View-Konzeptes ist das Document. Der Begriff Document ist hierbei abstrakt anzusehen, bei ihm handelt es sich um die Daten, mit denen das Programm arbeitet. Bei Textprogrammen ist es ein echtes Dokument, bei Bildverarbeitungsprogammen sind es die Bilder. Die Document-Klassen und die View-Klassen

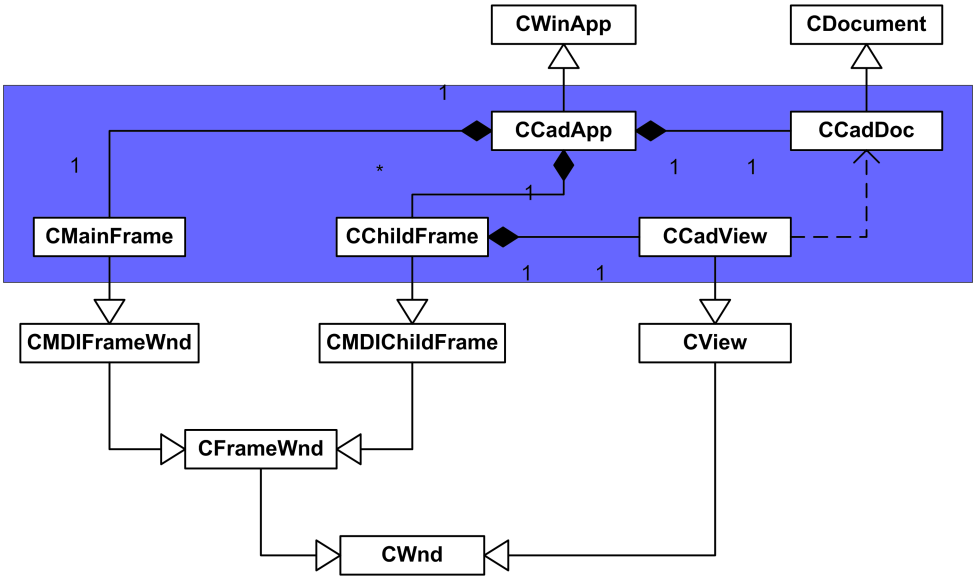


Abbildung 4.2: Schema des Document View Konzeptes der MFC

werden über die Klasse *CDocTemplate* bei einem *CDocManager*-Objekt registriert, das für die Verwaltung der *CDocTemplate* zuständig ist. Beim Öffnen eines neuen Documentes erzeugt das CDocManager-Objekt eine neue View. Das *CDocTemplate* wird nicht direkt verwendet, sondern eine der abgeleiteten Klassen *CSingleDocTemplate* oder *CMultiDocTemplate* unterschieden. *CSingleDocTemplate* wird bei SDI-Programmen, *CMultiDocTemplate* bei MDI-Programmen verwendet.

4.3.2 View

Der zweite Teil des Document-View-Konzeptes ist die View. Die View ist ein rahmenloses Fenster, tritt jedoch nach außen nicht als Fenster in Erscheinung. Sie ist ein Child-Window eines *CChildFrame*-Objektes, dessen Client-Area es vollständig einnimmt. Titelzeile, Statuszeile, Rahmen und ggf. Menü der „View“ gehören zum übergeordneten Framewindow. Die Aufgabe der View ist die Darstellung des Documentes oder Teile davon. Eine weitere Aufgabe der View ist die Steuerung der Interaktion mit dem Programmbenutzer, da sie die Benutzeraktionen mit der Tastatur und der Maus entgegennimmt.

4.3.3 Mainframe

Das Programm selbst wird durch den Mainframe repräsentiert. Er dient auch zur Unterbringung der Menüs, Toolbars etc. Bei Single Document Interface Programmen ist der

Mainframe und der Frame für Document identisch. Das Hauptfenster und der Frame für die View sind von der gleichen Basis abgeleitet.

4.3.4 Kommunikation View-Document

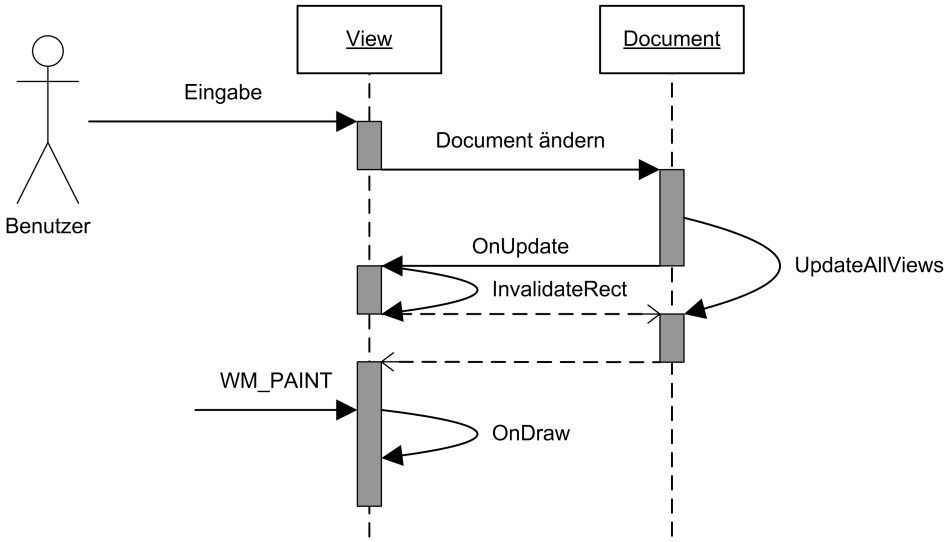


Abbildung 4.3: Schema des Document View Konzeptes der MFC

Die Benutzereingaben werden von der View entgegengenommen, daraufhin informiert die View das Document über die Änderung. Dies kann auf zwei verschiednen Arten geschehen:

1. Die View ruft die Membervariable des Documentes direkt auf, dies widerspricht jedoch der Trennung von Document und View. Ein weiteres Problem ist, dass das Document bei direktem Zugriff auf die Variable nicht mitbekommt, dass es sich geändert hat und kann somit die zugehörigen Views nicht über die Änderung informieren.
2. Die View ruft eine Member-Funktion des Documentes auf.

Nach der Entgegennahme der Änderung wird die Funktion `CDocument::UpdateAllViews` aufgerufen. Die Funktion `UpdateAllViews` geht über die Viewliste und ruft für alle zum Document gehörenden Views die Funktion `CView::OnUpdate` auf. Durch den Aufruf von `OnUpdate` wird das Viewfenster in der Standardimplementierung durch die Funktion `CWnd::InvalidateRect` als ungültig markiert. Anschließend sendet das Betriebssystem eine WM_PAINT-Nachricht, durch die die Funktion `CView::OnDraw` aufgerufen wird, die abschließend die View neu zeichnet.

5 Schichtmodelle in der Softwarearchitektur

Die Schichtmodelle sind Architekturmuster bei der Softwareentwicklung. Die Architekturmuster beschreiben, im Gegensatz zu den Entwurfsmustern in Kapitel 3, die ein Teilproblem beschreiben, den Grundaufbau einer Anwendung. Beim Schichtmodell werden die Komponenten einer Anwendung zu logischen Schichten zusammengefasst. Schichtarchitekturen werden auch oft in Verbindung mit Datenbanken und Netzwerken (Internet) verwendet, wobei die Daten auf einem zentralen Server liegen und die Ein- und Ausgabe auf einem lokalen Rechner erfolgt. Die Schichtmodelle funktionieren bei kleinen Programmen genauso wie bei großen Business-Anwendungen. Sie können auf einem Rechner laufen oder auf mehrere Computer verteilt sein. Der Vorteil der Schichtarchitektur liegt in der Flexibilität der Programmgestaltung. Die Programme werden wartungsfreundlicher, da die einzelnen Schichten ergänzt oder ausgetauscht werden können, ohne dass dies Auswirkungen auf die Anwendung hat.

Die Kommunikation zwischen den einzelnen Schichten geschieht über fest definierte Schnittstellen. Der Nachteil des Schichtmodells ist ein Geschwindigkeitsverlust, der sich als Folge des Durchreichens der Daten durch die einzelnen Schichten ergibt.

Schichtmodelle haben prinzipiell fünf verschiedene Ebenen, wobei aufeinander folgende Ebenen zu einer zusammengefasst oder eine Ebene weiter unterteilt werden kann. Die Bezeichnung unterscheidet sich je nach Anwendungsbereich des Modells:

Präsentationsschicht (externe Schnittstelle oder Client) Die Präsentationsschicht ist für die Darstellung der Daten und der Benutzereingaben zuständig.

Steuerung (Workspace oder Webschicht) Ist für die Steuerung der Anwendung zuständig.

Anwendungsschicht (Logikschicht, Businessschicht oder Enterprise Tier) Die Anwendungsschicht beinhaltet die Programmintelligenz, d.h. es werden in ihr die Daten verarbeitet.

Anwendungsobjekte (Resource Tier oder Datenverwaltung) Diese Schicht beinhaltet die möglichen Datentypen

Datenschicht (Server, Datenhaltung, Back End oder Database Tier) Die Datenschicht übernimmt das Speichern und Laden der Daten.

5.1 Einschichtige Architektur

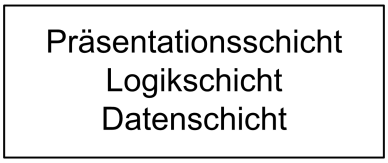


Abbildung 5.1: Einschichtige Architektur

Die einschichtige Architektur ist die älteste Architekturform. Sie besitzt nur eine einzige Schicht, die alle Komponenten enthält. Dies hat zur Folge, dass bei der Programmentwicklung große monolithische Blöcke entstehen, die schlecht zu warten sind.

5.2 Zweischichtige Architektur

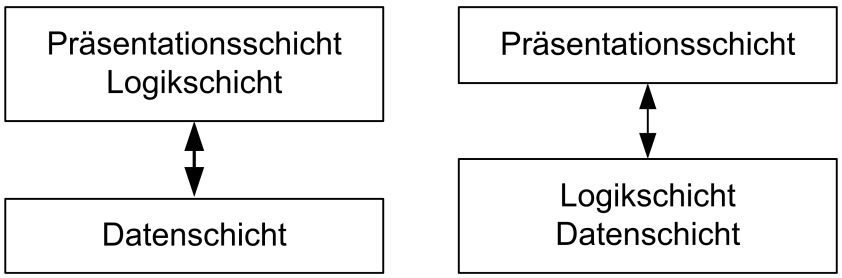


Abbildung 5.2: Zweischichtige Architektur

Die zweischichtige Architektur (engl. 2-Tier) teilt die Anwendung in zwei Schichten ein. Die erste Schicht ist die Ein- und Ausgabe (Präsentation), die zweite Schicht enthält die Daten. Bei dieser Architektur ist die Logikschicht entweder mit der Präsentationsschicht oder der Datenschicht eng verbunden.

- Präsentationsschicht und Logikschicht
- Datenschicht

oder

- Präsentationsschicht
- Datenschicht und Logikschicht

5.3 Dreischichtige Architektur

Die dreischichtige Architektur (engl. 3-Tier) ist das am weitesten verbreitete Schichtmodell. Bei der dreischichtigen Architektur wird zwischen

- Präsentationsschicht
- Logikschicht
- Datenschicht

unterschieden. Die Aufteilung entspricht weitgehend der Model-View-Controller Aufteilung (4), jedoch steht beim MVC-Konzept die Organisation der Mensch-Computer Interaktion im Vordergrund und nicht der allgemeine Programmaufbau.

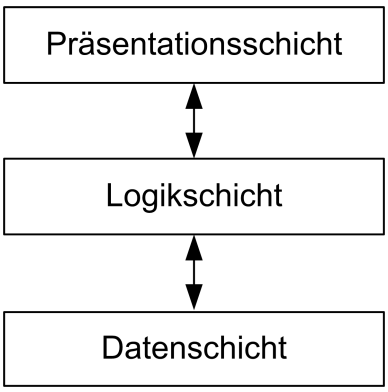


Abbildung 5.3: Dreischichtige Architektur

5.4 N-schichtige Architektur

Die n-schichtige Architektur ist die neueste Architekturform. Von einer n-schichtigen Architektur spricht man bei mindestens fünf Schichten. Eine mögliche n-schichtige Architektur ist die 5-schichtige Architektur; sie enthält alle Schichtarten:

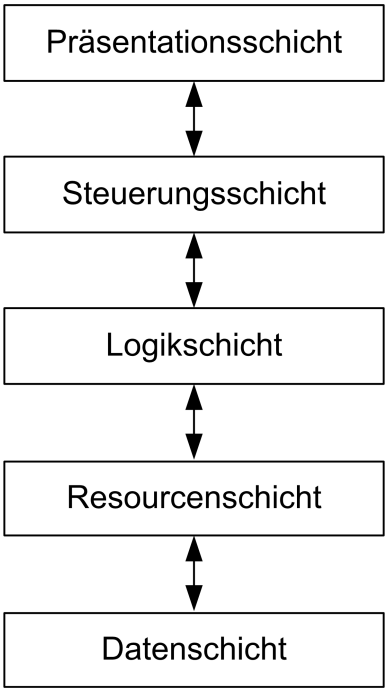


Abbildung 5.4: Fünfschichtige Architektur

- Präsentationsschicht
- Steuerung
- Anwendungsschicht
- Anwendungsobjekte
- Datenschicht

Bei der programmtechnischen Umsetzung des Architekturmusters können einzelne Schichten in weitere Unterschichten aufgeteilt werden.

6 Dreidimensionale Geometriemodelle

Für die Beschreibung der Geometrie im dreidimensionalen Raum werden 3D-Modelle verwendet. Aus diesen Modellen können die 2D-Modelle wie Ansichten und Projektionen hergeleitet werden.

Nach den Grundelementen, auf denen die Geometriemodelle aufbauen, lassen sich die Modelle in drei verschiedene Gruppen einteilen. In der Literatur werden zwischen Kanten-, Flächen- und Volumenmodellen unterschieden, die nachfolgend weiter erläutert werden.

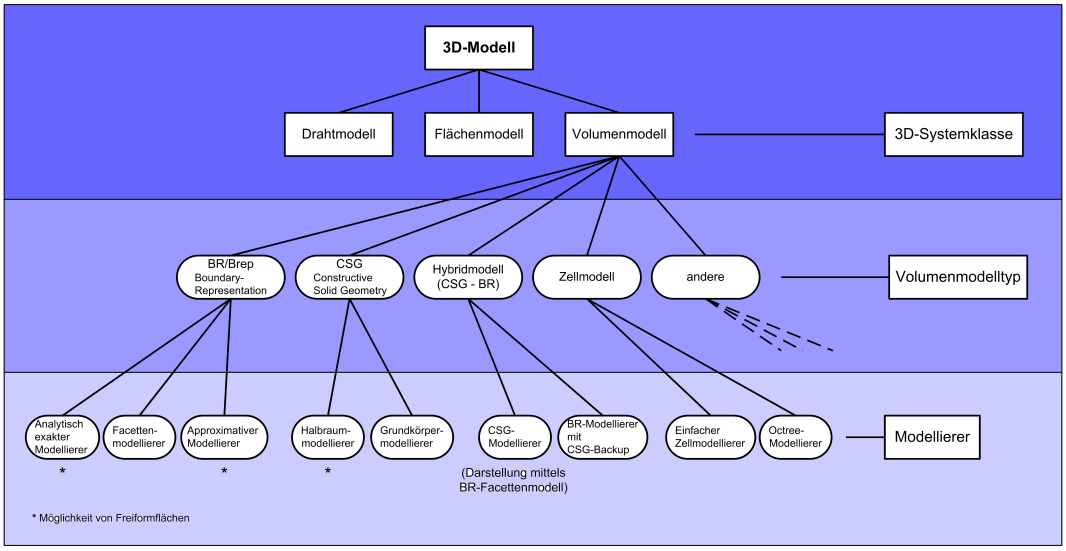


Abbildung 6.1: Einteilung der 3D-Modelle nach (Grä89)

6.1 Kantenmodell

Das Kanten- oder Drahtgittermodell ist das einfachste der 3D-Geometriemodelltypen. Das Kantenmodell speichert die Eckpunkte und Kanten des Modells, die für die Darstellung des Körpers verwendet werden. Die Kanten werden auf eine zweidimensionale Fläche projiziert, es entsteht ein dreidimensionaler Eindruck. Beim Darstellen eines Drahtgitter-

modells sind alle Kanten zu sehen, auch die, die auf der Rückseite des Körpers liegen und normal durch diesen verdeckt würden. Der Vorteil des Kantenmodells liegt in der geringen Datenmenge, die für die Darstellung benötigt wird und an der Geschwindigkeit bei der Darstellung. Nachteil des Drahtgittermodells sind die fehlenden Flächeninformationen. So lässt sich keine verdeckte Kantenberechnung oder Verschneidungen durchführen, auch ist die Darstellung nicht unbedingt eindeutig. Des Weiteren ist auch nicht möglich, physikalische Eigenschaften wie Oberfläche, Volumen und Trägheitsmomente zu berechnen.

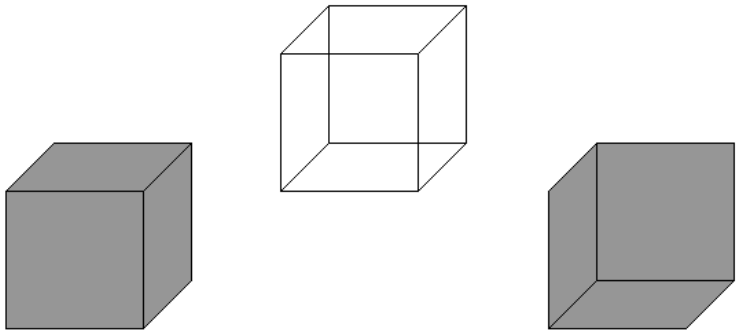


Abbildung 6.2: Würfel als Drahtgitter- und Flächenmodell

6.2 Flächenmodell

Die Weiterentwicklung des Kantenmodells ist das Flächenmodell. Das Flächenmodell beschreibt die Oberfläche des Körpers. Es wird häufig verwendet, wenn die Geometrie der Oberfläche nicht mehr alleine durch analytische Geometrie darstellbar ist. Mit dem Flächenmodell lassen sich verdeckte Kanten und Verschneidungen berechnen, auch sind die physikalischen Eigenschaften eingeschränkt ermittelbar. Der Nachteil der Flächenmodelle ist, dass die räumliche Integrität nicht garantiert ist. Es ist nicht klar, was die Innenseite und was die Außenseite des Körpers ist, der Flächenverband muss nicht geschlossen sein und mögliche Durchdringungen können nicht ausgeschlossen werden.

6.3 Volumenmodell

Die dritte Gruppe der 3D-Geometriemodelle sind die Volumenmodelle, die dreidimensionale Modelle über ihr Volumen repräsentieren. Bei den Volumenmodellen ist die Konsistenz der Daten sichergestellt. Sich selbst durchdringende Körper, die bei Draht- und

Flächenmodellen möglich sind, sind nicht mehr zulässig. Bauteile werden eindeutig und vollständig, es ist nicht möglich Körper mit fehlenden Kanten oder Flächen zu erzeugen.

Die Eigenschaften des Volumenmodells lassen sich berechnen, Sichtbarkeitsberechnungen und Verschneidungen von Objekten sind möglich. Volumenmodelle erfüllen alle Anforderungen, die an ein Geometriemodell für 3D-CAD-Programme gestellt werden.

Wie Abbildung 6.1 zeigt, lassen sich die Volumenmodelle wiederum unterteilen, u. a. in CSG-, Brep- und Hybridmodelle.

6.3.1 Brep-Modell

Das erste Volumenmodell, das beschrieben wird, ist das in Europa entwickelte Brep-Modell. Das Brep-Modell beschreibt die Geometrie eines Objektes über ihre Oberfläche.

Jeder Körper besitzt mindestens eine Oberfläche - Shell, die sich meist aus den einzelnen Flächen zusammensetzt. Es können mehrere Oberflächen vorhanden sein, wenn diese voneinander getrennt sind, wie die Innenfläche eines geschlossenen Hohlkörpers.

Die Flächen - Face werden durch ihre Kanten beschrieben. Jede Fläche gehört zu einer Oberfläche und besitzt mindestens drei Kanten. Die Kanten - Edge werden durch ihre Endpunkte bestimmt. Jede Kante gehört zu zwei Flächen und besitzt zwei Punkte. Die Eckpunkte werden durch die Koordinaten des Punktes im Raum x, y, und z beschrieben.

Das geometrische Modell des Brep-Systems ist durch die logischen Elemente wie Oberfläche, Fläche, Kante und Eckpunkte und den Relationen untereinander aufgebaut. Die hierarchische Zugehörigkeit der einzelnen Elemente, die Topologie und die Nachbarschaftsbeziehungen zwischen den Elementen wird durch eine Vorwärtsverzeigerung, oftmals auch durch eine Rückwärtsverzeigerung beschrieben.

Bei der Vorwärtsverzeigerung kennt jedes Element die Elemente aus denen es besteht, während sie bei der Rückwärtsverzeigerung die Elemente kennen, die sie beschreiben. Am Ende stehen im analytischen Teil die benötigten geometrischen Daten in Form von Koordinaten, Vektoren und Koeffizientengleichungen für Berechnungs- und Verknüpfungsoperationen.

Im Gegensatz zum CSG-Modell ist die Datenstruktur des Brep-Modells unabhängig von der Generierung.

6.3.2 Boundary-Modelltypen

Die Boundarymodelle können nach den Geometrie verarbeitenden Gesichtspunkten in drei verschiedene Modelle eingeteilt werden:

6.3.2.1 Das analytisch exakte Modell

Das analytisch exakte Modell ist das Genaueste der drei Brep-Modelltypen. Das Modell arbeitet nicht nur mit Flächen erster Ordnung, sondern auch zweiter und oftmals vierter Ordnung. Es verwendet somit nicht nur Flächen sondern auch Zylinder, Kegel, Kugel und Torus. Die Genauigkeit hängt nur von den verwendeten Algorithmen und der rechnerinternen Darstellung der Zahlen ab. Je mehr Flächen vorhanden sind, desto aufwendiger wird das System. Die Anzahl der benötigten Algorithmen steigt quadratisch, da jeder Körper mit jedem Körper verschnitten werden muss. Das Ergebnis der Verschneidungen sind verschiedene gerade Kanten, Kreise, Ellipsen, Kegelschnitte, Raumkurven 4. Ordnung und Splinekurven. Die verschiedenen Kantentypen benötigen wieder zahlreiche unterschiedliche Algorithmen. Die Algorithmenkomplexität hat Auswirkungen auf das Laufzeitverhalten des Systems. Der Vorteil des Modells liegt in der Weiterverarbeitbarkeit der Daten, für die Programmierung von Werkzeugen (NC-Programmierung) oder die Applikationssoftware von Spritzgussteilen. Die Menge der anfallenden Daten ist beim analytisch exakten Modell relativ gering, da auch gekrümmte Flächen mathematisch exakt beschrieben und nicht durch Teilflächen approximiert werden.

6.3.2.2 Das Facettenmodell

Das Facettenmodell oder Polyedermodell ist eine Vereinfachung des analytisch exakten Modells. Alle Flächen, auch die Gekrümmten werden durch ebene Flächen approximiert. Die Anzahl der Flächen hängt von der Größe der Teilflächen und der gewünschten Genauigkeit ab. Es existieren nur ebene Flächen und gerade Kanten wodurch die Komplexität der Algorithmen bei der Flächenverschneidung auf den trivialen Fall der Verschneidung zweier Flächen reduziert wird. Durch die Vereinfachung entstehen schnell Modell und Visualisierungsalgorithmen mit schnellen Antwortzeitverhalten.

Die Berechnungszeit steigt gewaltig bei Objekten mit gekrümmten Flächen gegenüber dem analytisch exakten Modell, da je nach Genauigkeit des Modells für die Approximation eine große Anzahl von Flächen benötigt wird.

Der Vorteil des Facettenmodells liegt in seiner Einfachheit und seiner Geschwindigkeit,

im Besonderen bei Objekten nicht gekrümmter Flächen. Bei gekrümmten Flächen steigt jedoch die Anzahl der Flächen und damit der Berechnungsaufwand überproportional, was die Geschwindigkeit der Anwendung reduziert und je nach Modell auch die Anwendung in die Knie zwingen kann. Ein weiterer Nachteil der Vereinfachung ist der Verlust der Genauigkeit, die ohne Zusatzinformationen nicht mehr aus dem Modell gewonnen werden kann. Eine Verwendung eines Facettenmodells für die Programmierung von Werkzeugmaschinen ist somit nicht möglich.

6.3.2.3 Das approximative Modell

Das approximative Modell hat große Ähnlichkeiten mit dem Facettenmodell, unterscheidet sich jedoch in den Grundlagen. Ebenso wie das Facettenmodell besitzt das approximative Modell nur einen Flächen- und Kantentyp, jedoch werden Parameterflächen und -kurven, meist auf der Basis von B-Splins - NURBS verwendet. Das approximative Modell beschreibt die unterschiedlichen Flächen und Kanten in einem einheitlichen Verfahren. Alle Algorithmen können allgemeingültig formuliert werden, da Fallunterscheidungen wie beim analytisch exakten Modell nicht benötigt werden. Dies bedingt kompakten Programmcode, ähnlich wie beim Facettenmodell, jedoch unterscheidet sich das Laufzeitverhalten. Während das Facettenmodell mit einfachen geometrischen Gleichungen arbeitet, wird beim approximativen Modell mit komplexen Interpolationsalgorithmen gearbeitet, die sehr rechenintensiv sind und ein ungünstiges Antwortverhalten zur Folge haben. Eine Verwendung des Modells für NC-Programme ist nur durch die Speicherung von Zusatzinformationen im System möglich, da die Genauigkeit des approximativen Modells nicht ausreichend ist.

6.3.3 CSG - Constructive Solid Geometry

Das CSG-Modell modelliert Körper indem es mit Hilfe von boolschen Operationen Objekte kombiniert. Als Grundobjekte (Primitive) dienen Geometrien, die sich auf einfache Art mathematisch beschreiben lassen wie Rechteck, Kugel, Zylinder, Kegel, Pyramide und Kreisring. Die zur Verfügung stehenden Operationen sind:

- Vereinigung (Union) $\cup$

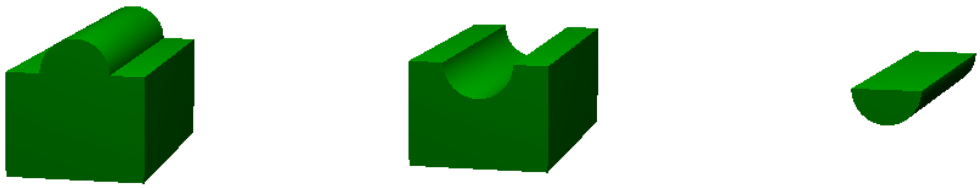
  Das neue Objekt wird durch die Vereinigung von zwei Objekten gebildet.
- Differenz (Difference) $\subset$

Erzeugung eines neuen Objektes indem von einem bestehenden Objekt ein anderes Objekt abgezogen wird.

- Schnitt (Intersection) ∩

  Beim Schnitt wird das neue Objekt als Schnittmenge zwischen zwei bestehenden Objekten gebildet.

Abbildung 6.3 zeigt exemplarisch die drei boolschen Operationen mit einem Würfel und einem Zylinder.



(a) Vereinigen (b) Differenz (c) Schnitt

Abbildung 6.3: Boolsche Operationen

Da die einzelnen Erzeugungsschritte der Geometrie meist nicht kommutativ sind, muss die Reihenfolge der Erzeugung gespeichert werden. Die Erzeugungsgeschichte des Körpers wird hierarchisch geordnet in einem CSG-Baum gespeichert. Die Primitiven bilden hierbei die Blätter des Baumes, die CSG-Operationen sind die einzelnen Knoten. Die Wurzel des Baumes bildet das Endergebnis der CSG-Konstruktion.

6.3.4 Mischformen - Hybride Modelle

Hybridmodelle sind Weiterentwicklungen des CSG-Modells auf die im Folgenden nur kurz eingegangen wird. Hybridmodelle beinhalten Datenstrukturen aus dem CSG- und Brep-Modell, wobei eine Datenstruktur dominant ist. Es wird unterschieden zwischen CSG-Modellierer mit Brep zur Darstellung und Brep-Modellierer mit CSG Backup.

6.3.4.1 CSG-Modellierer mit Brep zur Darstellung

Der größte Nachteil des CSG-Modells ist die geringe Geschwindigkeit bei der Visualisierung. Für die Visualisierung muss der gesamte CSG-Baum durchlaufen werden und abschließend für die Darstellung ein Brep-Modell aufgebaut werden. Dieses Durchlaufen

des Baumes und Aufbauen des Modells ist sehr zeitaufwendig. Eine Möglichkeit der Beschleunigung bietet die Ergänzung des Datenmodells mit einem Brep-Modell. In jedem Knoten wird neben den CSG-Operationen auch ein Brep-Modell, das das Aussehen des Modells beinhaltet, gespeichert. Bei der Darstellung muss nicht mehr der Baum durchlaufen werden, es kann direkt auf das vorhandene Brep-Modell zugegriffen werden. Dieser Geschwindigkeitsvorteil wurde jedoch zulasten des Speichers erkauft, da zwei Datenmodelle parallel gehalten werden, und nicht nur der Endzustand in einem Brep-Modell gespeichert ist, sondern auch für jeden Zwischenzustand ein Brep-Modell existiert. Das Brep-Modell kann nicht direkt angesprochen werden und Änderungen am Brep-Modell haben auch keine Auswirkungen auf das CSG-Modell, da keine Verbindung zwischen dem Brep-Modell und dem CSG-Modell besteht.

6.3.4.2 Brep-Modellierer mit CSG Backup

Die zweite Variante eines dualen Systems ist ein Brep-Modell als Primärstruktur und einem CSG-Baum als Sekundärstruktur. Das Boundary-Modell ist die Arbeitsdatenstruktur auf der Operationen beruhen. Das Modell enthält nur eine Brep-Struktur, meist als Facettenmodell. Dadurch müssen nur einfache geometrische Elemente wie Gerade, Kante und Ebene berücksichtigt werden.

Parallel zum Boundarymodell wird eine zweite Datenstruktur gehalten, in der die Eingabe protokolliert wird, jedoch besteht zwischen den beiden Modellen keine Verbindung. Durch das „Backup-Modell“ kann zu jedem Zeitpunkt die Auflösung des Facettenmodells beliebig verändert werden.

Das Ortzeitverhalten des Modells wird zulasten der Genauigkeit der Darstellung erhöht. Wenn die Modelleingabe beendet ist, kann das Modell aufgrund der Backupstruktur mit einer feineren Facettierung erneut erstellt werden.

6.3.5 Graphische Darstellung der Geometriemodelle

Die graphische Darstellung des Brep-Modells erfolgt unmittelbar aus den Modelldaten unter Auswertung der Geometrie und Topologie. Bei der graphischen Ausgabe wird Flächenweise vorgegangen. Bei gekrümmten Oberflächen werden zusätzlich projektionsabhängige Sichtkanten berechnet.

Verdeckte Kantenberechnungen (Hidden-Line) sind mit den vorhandenen Daten möglich, jedoch aufwändig, da Flächenverschneidungen in der Projektionsebene zeitaufwändig er-

rechnet werden müssen.

Die graphische Darstellung des CSG-Modells ist zeitaufwändiger, da diese nicht direkt erfolgen kann. In der Regel wird aus dem CSG-Modell zuerst ein Brep-Modell erzeugt, welches anschließend für die Darstellung verwendet wird.

Teil II

CAD-Programme

7 Funktionalität von CAD Programmen

Die Funktionalität von modernen CAD-Programmen im Bauwesen wird am Beispiel des Programms Vicado von mb-Software erläutert. Das Programm beruht auf der ORBIT-Technologie - Objekte und Relationen im Bauwesen mit Integrativer Technologie. Während die meisten CAD-Programme auf eine lange Entwicklung zurückschauen und noch Altlasten der Entwicklung mitschleifen, ist Vicado eine von Beginn an objektorientierte Neuentwicklung vom Ende der neunziger Jahre, die auf die modernen Technologien ausgerichtet ist.

Die Funktionalität des Programms wird mit Hilfe der Toolbars gezeigt, in denen die Funktionen der Anwendung angeordnet sind. Ein Grundprinzip der meisten Funktionen ist die Zweistufigkeit. Im ersten Schritt wird ausgewählt, was getan werden soll, während im zweiten Schritt festgelegt wird, wie es ausgeführt wird.

7.1 Dateifunktionen

Dateifunktionen (Abb. 7.1) ist die Standardtoolbar, wie sie in den meisten Programmen vorkommt. Sie enthält hauptsächlich projektbezogene und allgemeine Befehle, die sich nicht auf Objekte beziehen.

Abbildung 7.1: Toolbar Dateifunktionen

- Speichern, Drucken, Viewerausgabe
- Löschen, Ausschneiden, Kopieren, Einfügen
- Rückgängigmachen, Wiederherstellen
- Zoombox, Vergrößern, Verkleinern, auf markierte Objekte zoomen, auf alles zoomen

- Auffrischen
- Gruppieren, Gruppierung aufheben
- Hilfe, Neue Folie, Folie auswählen

7.2 Wasleiste

Die wichtigste Funktion in einem CAD-Programm ist die Erzeugung neuer Objekte. Die zur Erzeugung zur Verfügung stehenden Objekte sind in der Wasleiste angeordnet.

Abbildung 7.2: Wasleiste

Die Wasleiste (Abb. 7.2) ist ein Tab-Control-Element. Die zusammengehörenden Objekte sind je in einer eigenen Tab-Page angeordnet. Bei Vicado geschieht die Anordnung in alphabetischer Reihenfolge. Eine andere Möglichkeit ist auch eine Anordnung nach der Wichtigkeit der Objekte. Hierbei sind die Objektgruppen absteigend in der Häufigkeit ihrer Verwendung angeordnet. Die einzelnen Objektgruppen sind:

- Ausbau
- Bauteile
- Bemaßung
- Bewehrung
- Einbauteile
- FEM-Platten
- 2D Graphik
- Mengenermittlung
- Positionsplan
- Internet

Die vorhandenen Objektgruppen sind abhängig vom Anwendungsbereich des Programms, so sind z. B. FEM-Platten und Bewehrung ingenieurspezifische Gruppen, die im Architekturbereich nicht vorkommen.

7.3 Wieleiste - Beispiel Wand

Die Wie-Leiste (Abb. 7.3) ist die Ergänzung zur Was-Leiste. Die Wie-Leiste dient zur Auswahl weiterer Angaben zur Objekterzeugung. Es kann die Art der Objekterzeugung ausgewählt werden, da viele Objekte auf verschiedenste Art und Weise erzeugt werden können. Des Weiteren können Parameter für die Erzeugung des Bauteils angegeben werden, die über die graphische Konstruktion nicht gewonnen werden können. Die Parameter sind von Objekt zu Objekt unterschiedlich und können sich auch bei den unterschiedlichen Buildern für einen Objekttyp unterscheiden.

Abbildung 7.3: Wieleiste des Bauteils Wand

Mit dem Button am rechten Ende der Toolbar kann ein Eigenschaftsdialog zur Eingabe zusätzlicher Wandeigenschaften geöffnet werden. Die eingegebenen Werte können als Vorlage abgespeichert und später wieder verwendet werden.

Bei der Erzeugung einer Wand können die folgenden Parameter in der Wie-Leiste angegeben werden:

- Art der Wand (Einschichtig, Zweischichtig)
- Erzeugungsart (Polygon, Rechteck, Rund)
- Ausrichtung (mittig, links, rechts)
- Dicke, Höhe
- Niveau, Versatz
- Eigenschaften

7.4 Numerische Eingabe

Die numerische Eingabe ergänzt die graphische Konstruktion. Mit ihr (Abb. 7.4) können bei der Konstruktion die Koordinaten von Punkten in Kartesischen- oder Polarkoordinaten eingegeben werden. Die Einheiten für die Eingabe sind frei wählbar aus einer Liste gängiger Einheiten.

- x, y

Abbildung 7.4: Toolbar für nummerische Eingaben

- Abstand, Winkel
- Einheit, Eigenschaften

7.5 Objekte Manipulieren

Über Manipulieren (Abb. 7.5) kann die Lage von Objekten im Raum verändert werden. Durch die Kombination mit der Klonfunktion der Objekte können auch neue Objekte erzeugt werden, z. B. über Verschieben und Kopieren.

Abbildung 7.5: Toolbar Objekte manipulieren

- Verschieben, Drehen Spiegeln, Punktspiegeln
- Verschieben, Drehen Spiegeln, Punktspiegeln und Kopieren

7.6 Geometrie Manipulieren

Mit den Funktionen von Geometrie Manipulieren (Abb. 7.6) kann das Geometriemodell der Objekte bearbeitet werden.

Abbildung 7.6: Toolbar Geometrie manipulieren

Hierfür stehen verschiedene Funktionen wie

- Trimmen
- Punkt in Strecke einfügen, Polygonpunkt löschen
- Strecke auftrennen, Parallele Teilkante herausziehen

- Punkt (Kante) verschieben, Kante löschen, Kante abrunden
- Objekt teilen, vereinigen

zur Verfügung.

7.7 Ausrichten der Objekte

Mit Ausrichten von Objekten (Abb. 7.7) können ausgewählte Bauteile ausgerichtet werden.

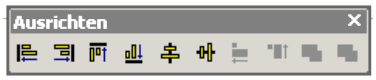


Abbildung 7.7: Toolbar Ausrichten

Die Position der Bauteile wird bestimmt durch das erste ausgewählte Bauteil, die weiteren Bauteile richten sich nach diesem. Vicado bietet mehrere Möglichkeiten zum Ausrichten der Objekte:

- an der linken/rechten Kante des Objektes
- an der oberen/unteren Kante des Objektes
- Objekte zentrieren
  - horizontal zentrieren
  - vertikal zentrieren
- Horizontal / Vertikal an Referenzpunkt ausrichten
- Objekt in den Vordergrund/Hintergrund

7.8 Baukörper

Die Toolbar Baukörper (Abb. 7.8) bietet weitere Möglichkeiten zur Manipulation der Bauteile.

Die Funktionen betreffen das Verschneiden der Objekte.

- Zu Eck verschneiden

Abbildung 7.8: Toolbar Baukörper

- Als T verschneiden
- Ausrunden
- Alles neu verschneiden

7.9 Baukörper Verschneiden

Baukörper verschneiden (Abb. 7.9) bietet die Möglichkeit die Verschneidung der Objekte manuell zu steuern.

Abbildung 7.9: Toolbar Verschneider

Moderne Programme regeln die Verschneidung der Objekte automatisch über vorgegebene Regeln. Die automatische Verschneidung kann aber nicht alle Möglichkeiten der Verschneidung nach dem Willen des Anwenders korrekt regeln. Für diese Fälle sind verschiedene Methoden zur manuellen Verschneidung vorgesehen:

- Diagonal verbinden
- Über Eck verbinden, erstes Bauteil rechtwinklig
- Über Eck verbinden, erstes Bauteil durchgehend T-förmig stoßen
- Bauteile nur voneinander abziehen
- Bauteile nicht verschneiden
- Bauteile ausschneiden, Bauteile werden ausgeschnitten
- Bei unsym. Eckverbindungen durchgehend, nichtdurchgehend
- Eckverschneiden, T verschneiden, ausrunden
- neu verschneiden

7.10 Konstruktionspunkt und Konstruktionslinie

Diese Toolbar (Abb. 7.10) steuert die Lage und die Ausrichtung des lokalen Ursprungs in der Arbeitsebene. Über Konstruktionslinie können des Weiteren Zwangsgeraden für die Konstruktion vorgegeben werden.

Abbildung 7.10: Toolbar Konstruktionspunkt und Linie

- Ursprung zurück in Ausgangslage
- Konstruktionsrichtung auf 0° zurückdrehen
- Konstruktionsrichtung um 90° drehen
- Ursprung setzen
- Konstruktionsrichtung übernehmen, konstruieren
- Zwangsgerade, Zwangsgerade Parallel -> zu Punktkonstruktion

7.11 Punktkonstruktion

Die Punktkonstruktionen (Abb. 7.11) werden verwendet, um über die graphische Oberfläche Punkte für die Erzeugung der Bauteile zu konstruieren.

Abbildung 7.11: Toolbar Punktkonstruktion

Typische Punktkonstruktionen sind:

- Mittelpunkt einer bestehenden Strecke
- Mittelpunkt konstruieren
- Lotfußpunkt, Senkrechte auf einer Strecke
- Schnittpunkt konstruieren

7.12 Sichten

Über die Toolbar Sichten (Abb. 7.12) wird die Erzeugung von neuen Sichten (siehe 17.2) und die Anordnung der Sichten auf dem Bildschirm gesteuert.

Abbildung 7.12: Toolbar Sichten

Die verschiedenen Sichten die erzeugt werden können sind:

- 2D-Sicht, 3D-Sicht, Plansicht, neue Graphische Sicht
- Schnitt, Detail
- Sicht Duplizieren

Die Sichten können in den folgenden Positionen automatisch angeordnet werden

- überlappend, übereinander
- nebeneinander, übereinander und nebeneinander
- Sichten links rechts

7.13 Planview

Abbildung 7.13: Toolbar Planview

Für die Erzeugung von Plänen werden die vorhandenen Sichten auf einer neuen Sicht der Planview angeordnet. Die Steuerung dieser Anordnung geschieht über die Toolbar Planview (Abb. 7.13). Für die Erzeugung einer Planview stehen die folgenden Funktionen zur Verfügung:

- Aktiven Planbereich löschen
- Eigenschaften
- Schriftfeld anzeigen

- Schriftfeldsymbol
- Neue 2D-Sicht in Plan
- Neue 3D-Sicht in Plan
- Bestehende Sicht in Plan übernehmen

7.14 Fangen - Raster

Über die Toolbar Fangen und Raster (Abb. 7.14) können die Einstellungen für die Raster- und Fangfunktionen des Programms direkt bearbeitet werden.

Abbildung 7.14: Toolbar Raster und Fangmodi

Die Nachfolgenden Einstellungen sind direkt auswählbar.

- Punkt (Ecke) fangen
- Linie fangen
- Hilfslinie fangen
- Schnittpunkt fangen
- Raster aktivieren/deaktivieren
- Raster sichtbar/unsichtbar
- geschnittene Objekte fangen

7.15 FE-Ergebnisse

Die Toolbar FE-Ergebnisse (Abb. 7.15) ist eine ingenieurspezifische Toolbar. Über sie wird die Kopplung des CAD-Programms mit dem FE-Programm MicroFe gesteuert. Nach der Berechnung der erforderlichen Bewehrung können die Ergebnisse eingelesen und zur Weiterverarbeitung dargestellt werden. Vicado bietet die folgenden Möglichkeiten der Ergebnisdarstellung:

- Auswahl der Lage
- Zahlenwerte

- x/y Isolinien
- x/y/xy Flächen
- Daten laden/entladen
- Eigenschaften

Abbildung 7.15: Toolbar Fe-Ergebnisse

7.16 Visualisierung

Die Visualisierung (Abb. 7.16) hat in den vergangenen Jahren immer weiter an Bedeutung gewonnen. Besonders wichtig ist sie im Architekturbereich für die Präsentation des Entwurfes.

Abbildung 7.16: Toolbar Visualisierung

Die Funktionen die Vicado für die Visualisierung bietet sind:

- Durchwandern des Modells
- Arbeitsebene herauf/herunter wechseln
- Betrachten
- Raytracing
- Speichern des aktuellen Bildes
- Transparent
- Tag/Nacht
- Schatten ein/aus

7.17 Videorecorder

Der Videorecorder (Abb. 7.17) gehört zu der Visualisierung des Gebäudemodells. Mit Hilfe des Videorecorders kann der Gang durch das Gebäudemodell als Film aufgenommen und

zur Präsentation abgespielt werden. Der „Videorecorder“ bietet die gleiche Funktionalität wie ein normaler Videorecorder:

- Wiedergabe der Aufzeichnung
- Vor, Zurück spulen
- Stop, Pause
- Standbilder
- Aufnahme
- Auswerfen

Abbildung 7.17: Toolbar Videorecorder

7.18 Koordinatensysteme, Ursprung und Raster

7.18.1 Koordinatensystem

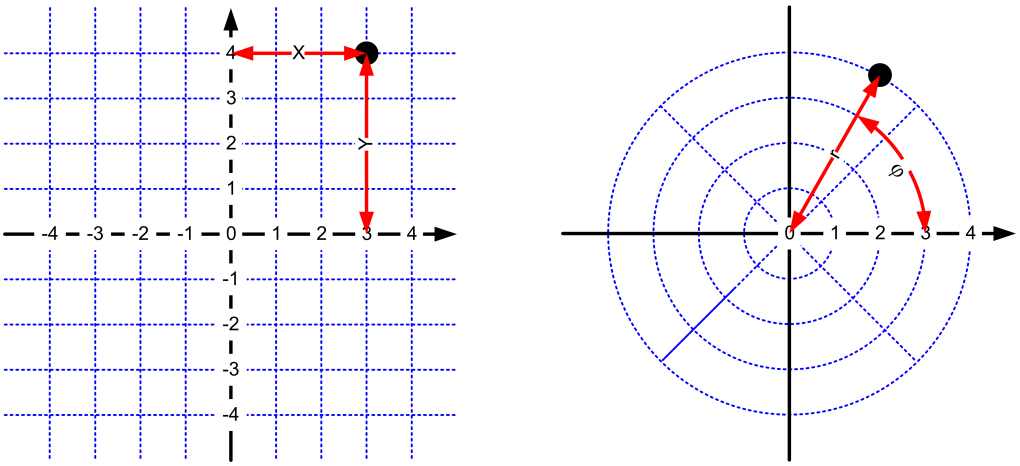
CAD-Programme bieten mehrere Koordinatensysteme zur Verwendung an. Während im Programm in der Regel nur ein globales Koordinatensystem vorhanden ist, können mehrere lokale Koordinatensysteme vorkommen. Das globale Koordinatensystem dient zur Positionierung der Bauteile im Raum, die lokalen Koordinatensysteme zur Konstruktion der Bauteile. Die lokalen Koordinatensysteme können vom Programmanwender frei gesetzt werden, oftmals beziehen sie sich auf den letzten Punkt der vorhergegangenen Konstruktion, da das neue Bauteil meist in Relation zur letzten Konstruktion steht.

7.18.1.1 Arten von Koordinatensystemen

In der Ebene werden bei CAD Programmen in der Regel zwei Arten von Koordinatensystemen verwendet, das kartesische und das polare Koordinatensystem.

Kartesische Koordinaten Abb. 7.18(a) Das kartesische Koordinatensystem ist benannt nach dem lateinischen Namen seines Erfinders René Descartes. Die horizontale Achse wird als x-Achse, die vertikale Achse als y-Achse bezeichnet. Die Position wird über ein Koordinatenpaar, den Werten auf der x- und y-Achse angegeben.

Polarkoordianten Abb. 7.18(b) Im Polarkoordinatensystem wird die Position über den Abstand zum Koordinatenursprung r und einen Winkel φ definiert. Der Winkel wird entweder linksdrehend, gegen den Uhrzeigersinn mathematisch positiv oder rechtsdrehend mathematisch negativ angegeben.



(a) Kartesisches Koordinatensystem (b) Polarkoordinatensystem

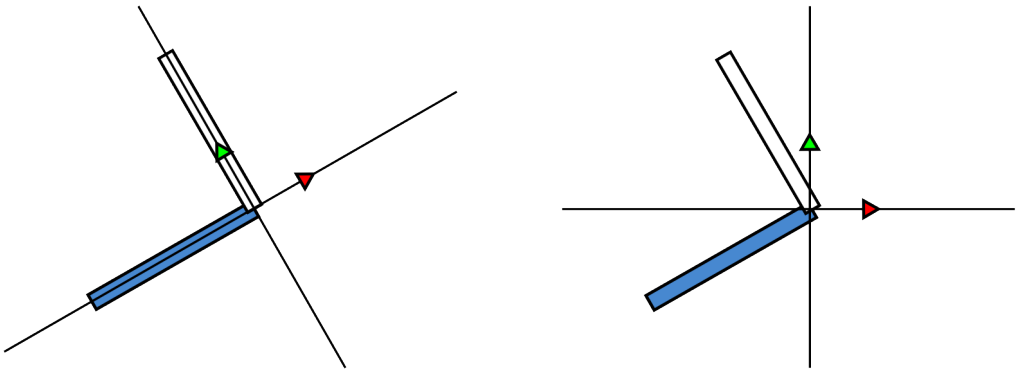
Abbildung 7.18: Verschiedene Koordinatensystemarten

7.18.2 Der Koordiantenursprung

Der Koordinatenursprung eines CAD-Programms ist meist kein fester sondern ein lokaler Ursprung. Der lokale Ursprung wandert bei der Konstruktion mit und positioniert sich auf dem letzten Punkt des vorangegangenen Konstruktionsschrittes oder der entsprechenden Konstruktion.

Der Vorteil des mitwandernden Ursprungs liegt darin, dass sich Koordinateneingaben relativ auf den letzten Punkt beziehen und dass die Ausrichtung des Koordinatensystems sich an der bisherigen Konstruktion orientiert. Die im Allgemeinen bevorzugte rechtwinklige Konstruktion kann direkt ausgewählt werden.

Bild 7.19 zeigt die Konstruktion von Wänden mit und ohne mitdrehendes Koordinatensystem. Teilbild 7.19(a) zeigt eine Eingabe mit dem mitdrehenden Koordinatensystem, die neue Wand steht in einem 90° Winkel auf der vorhergegangenen Wand, der Winkel kann über das 90° Raster gefangen oder ohne Probleme numerisch eingegeben werden. Bei dem nicht mitdrehenden Koordinatensystem kann der Winkel nicht direkt gefangen werden, eine Eingabe des Winkels ist nicht möglich, da hierfür der Winkel zwischen der vorhandenen Wand und dem Koordinatensystem bekannt sein muss. Ein Drehen des Ko-



(a) Mitwanderndes Koordinatensystem

(b) Nicht mitwanderndes Koordinatensystem

Abbildung 7.19: Koordinatensysteme

ordinatensystems auf die Richtung der Wand ist ein zusätzlicher Arbeitsschritt, der das Arbeiten verzögert.

7.18.3 Raster

Das Raster soll den Anwender bei der Eingabe von Objekten unterstützen. Es erspart die numerische Eingabe von Punkten, da die gewünschte Koordinate oder Strecke gefangen werden kann. Hierfür muss das Raster mit Maßen übereinstimmen, die oft benötigt werden.

Als Rastermaß hat sich im Bauwesen ein 12,5cm Raster für Längen herausgebildet. Dieses Raster geht auf das Jahr 1872 zurück, als in Deutschland per Gesetz das so genannte alte Reichsformat für Ziegel eingeführt wurde, mit den Abmessungen 25cm x 12cm x 6,5cm bzw. auf das sich daraus entwickelte Normalformat. Die Länge von 25cm oder ein Vielfaches davon ist noch heute für die Abmessungen von Mauersteinen maßgebend. Im Betonbau ist dieses Raster von geringerer Bedeutung, da hier die Abhängigkeit von der Größe eines Baustoffes nicht gegeben ist. Das Winkelraster beruht auf einem 15° Raster, mit 15, 30, 45, 60, 75 und 90 Grad. Am häufigsten werden hierbei die Winkel 90, 45, 30 bzw. 60 Grad verwendet.

7.19 Verschneiden der Objekte

Die einzelnen Bauteile in einem Gebäude begrenzen sich gegenseitig. An den Schnittstellen zwischen den einzelnen Bauteilen müssen diese miteinander verschnitten werden, damit ein konsistentes Gebäudemodell erzeugt werden kann. Während die Verschneidung früher durch den Konstrukteur manuell vorgenommen wurde, bieten moderne Programme heute automatische Verschneidroutinen an.

7.19.1 Automatisierung der Verschneidung

Die Automatisierung der Bauteilverschneidung ist eine weitere Hilfestellung für den Anwender. Sie erleichtert und beschleunigt die Eingabe des Modells, indem durch das Programm versucht wird, die gewünschte Verschneidung der Bauteile automatisch zu erzeugen.

Das Verschneiden der Bauteile ist einmal abhängig von der Art der Bauteile die verschnitten werden, aber auch bei Bauteilen der gleichen Art gibt es Unterschiede. Für einen intelligenten Verschneidungsalgorithmus müssen all diese Punkte berücksichtigt werden, damit er bei möglichst vielen Fällen korrekte Anwendung finden kann.

- Berücksichtigung der Bauteilart

  Einzelne Bauteilarten begrenzen in der Regel andere. Wände ebenso wie Stützen beginnen auf Fundamenten oder Decken, und enden unterhalb von Decken und Dächern. Hieraus ergibt sich für die Verschneidung ein Vorrang der begrenzenden Bauteile.

- Unterschiede zwischen Bauteilen gleicher Art

  Die Bauteile, die am häufigsten miteinander verschnitten werden, sind Wände. Bei den Wänden wird für die Verschneidung unterschieden zwischen Innenwänden und Außenwänden, wobei die Außenwände aufgrund ihrer begrenzenden Funktion eine höhere Verschneidungspriorität besitzen. Ein weiteres Kriterium zur Bestimmung der Verschneidungspriorität ist die Wanddicke, da die dickere Wand in der Regel die durchgehende Wand ist. Des Weiteren lässt sich auch aus dem Material, bedingt durch den Herstellungsprozess, eine Verschneidungsreihenfolge herleiten. Da die Stahlbetonbauteile in einem Gebäude meist die tragenden Teile sind und das Mauerwerk nur füllende Aufgabe besitzt, erhalten Stahlbetonwände gegenüber dem Mauerwerk eine höhere Verschneidungspriorität.

7.19.2 Verschneidungsarten von Mauern

Für die Verschneidung zweier Wände gibt es mehrere Möglichkeiten, sechs von ihnen werden näher erläutert:

Abbildung 7.20: Toolbar Verschneider

1. Diagonal

   Beim diagonalen Verschneiden bildet die Winkelhalbierende des Winkels, den die beiden Wände bilden die Grenze.

2. Über Eck, 1. Bauteil rechtwinklig

   Die Erste zum Verschneiden markierte Wand behält ihre rechtwinklige Form, sie bildet die Grenze zum Verschneiden der weiteren Wand.

3. Über Eck, 1. Bauteil durchgehend

   Diese Verschneidungsart ähnelt optisch der diagonalen Verschneidung, jedoch geht die erste Wand durch. Die Form des verschnittenen Bauteils wird durch die Schnittpunkte der Außenkanten der beiden verschnittenen Wände bestimmt.

4. 1. Bauteil T gestoßen

   Beim T-Stoß geht die zweite Wand durch, die erste Wand wird an ihr gestoßen.

5. 1. Bauteil vom Zweiten abziehen

   Hierbei wird keine neue Form des Bauteils berechnet. Es wird die Schnittmenge der Geometrie zwischen den beiden Bauteilen vom zweiten Bauteil abgezogen. Das erste Bauteil behält seine ursprüngliche Geometrie.

6. Nicht verschneiden

   Ist die Deaktivierung des automatischen Verschneidens.

Neben der Bestimmung der Ausbildung eines Knotenpunktes besteht auch die Möglichkeit das globale Verschneidungsverhalten einer Wand zu bestimmen.

1. Bauteil schneidet aus

   Die ausgewählte Wand ist durchgängig, alle anderen Wände werden durch diese Wand durchtrennt.

2. Bauteil schneidet nicht aus

   Die ausgewählte Wand ist untergeordnet, alle schneidenden Wände gehen durch die Wand hindurch.

3. Bei unsymmetrischen Bauteilen durchgehend

   Bei einem unsymmetrischen Wandanschluss behält die Wand ihre Form, die anderen Wände werden an ihr gestoßen.

4. Bei unsymmetrischen Bauteilen nicht durchgehend

   Die Wand wird bei einem unsymmetrischen Anschluss untergeordnet, die angeschlossenen Wände behalten ihre Form, die gewählte Wand wird abgeschnitten.

8 Graphische Oberfläche

8.1 Entwicklung der Benutzeroberfläche

Im Jahr 1983 präsentierte Microsoft zum ersten Mal Windows, zwei Jahre später kam es auf den Markt. Der Erfolg von Windows stellte sich jedoch erst mit den Dreier-Versionen in den Neunziger Jahren ein. Anfangs war Windows ein graphischer Aufsatz für DOS. Später entwickelte Microsoft, ursprünglich mit IBM zusammen (als OS/2 Nachfolger), mit Windows NT ein stabiles graphisches Betriebssystem. Der endgültige Durchbruch wurde mit Windows95 erreicht. Windows95 ist ebenso wie Windows NT ein 32Bit Betriebssystem, das auf relativ kostengünstiger Hardware läuft.

CAD-Programme haben zu diesem Zeitpunkt meist schon eine mehrjährige Entwicklungsphase hinter sich. Die Softwarehersteller lieferten zu diesem Zeitpunkt ihre Programme meist für mehrere verschiedene Plattformen (UNIX, Dos) aus. Dies änderte sich erst mit dem Durchbruch von Windows, in der Mitte der neunziger Jahre in deren Folge die Hersteller ihre Nicht-Windowsversionen aufgaben.

Die weitere Entwicklung der graphischen Oberfläche des Betriebssystems ist nicht nur an das Betriebssystem gekoppelt, sondern auch an einzelne Produkte der Firma Microsoft. Mit dem Internet Explorer 6 wurde beispielsweise ein neuer Stiel für die Toolbars eingeführt. Einen Sprung beim Konzept der Benutzeroberfläche machte Microsoft mit der Einführung von Office 2007. Mit dem neuen Officepaket wurde auf die klassischen verschachtelten Menüs verzichtet. Bild 8.1 zeigt als Beispiel die Oberfläche von Word 2007.

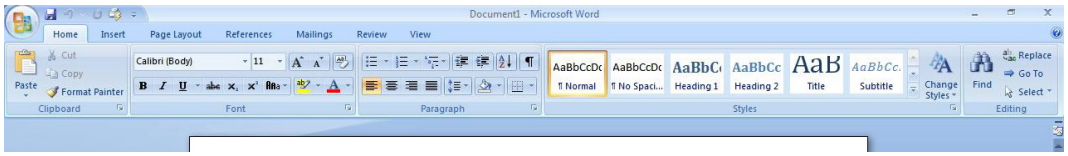


Abbildung 8.1: Menüzeile von Word 2007

Das Menü wurde bei Word2007 durch ein Tabsheet ersetzt, das die einzelnen Menüpunkte

enthält, indem diese in Toolbars integriert wurden.

Das Programm Word ist auf die Eingabe von Text voreingestellt, da dies der vordringliche Zweck des Programms ist. Die Eingabe anderer Objekte in den Text, wie Tabellen oder Graphik, werden einmal über das Tabsheet „Einfügen“ aktiviert. Es ist davon auszugehen dass die anderen Softwarehersteller diesem Trend folgen werden und mit der Zeit die Menüstrukturen, die noch aus DOS-Zeiten stammen immer mehr aus ihren Programmen entfernen. Andere Firmen hatten ähnliche Konzepte schon vor Jahren eingeführt. Beispiele hierfür sind Borland mit der Oberfläche der Builder-Familie oder Corel Corporation mit CorelDraw.

Im Jahre 1995 brachte die Firma mb-software das CAD-Programm Arcon mit einem für die damalige Zeit neuen Bedienungskonzept auf den Markt. Während die meisten CAD-Programme über Jahre gewachsen sind, wurde bei Arcon ein CAD-Programm von Grund auf neu entwickelt. Das Programm war von Anfang an auf Windows ausgerichtet, dies zu einem Zeitpunkt, zu dem noch nicht alle CAD-Anwendungen von DOS auf Windows umgestellt waren.

Arcon besitzt eine graphisch orientierte Oberfläche, Ballast aus DOS-Zeiten sind bei diesem Programm nicht vorhanden. Das Anwendungskonzept war damit den meisten anderen CAD-Programmen voraus.

Arcon besitzt noch ein Menü, jedoch lassen sich über die Toolbars nicht nur die Objektart sondern auch der konkrete Objekttyp und die Erzeugungsart auswählen. Links am Rand des Fensters befindet sich die Was-Leiste, mit dieser kann ausgewählt werden, was erzeugt werden soll. Direkt neben der Was-Leiste ist eine Konstruktionsleiste eingeblendet, mit der die Konstruktionsmethode ausgewählt werden kann. Die Darstellungseigenschaften der Pläne und Visualisierung kann über die obere Toolbar gesteuert werden.

Im Jahr 2001 brachte die mb-Software AG mit Orbit (später in Vicado umbenannt) ein weiteres CAD-Programm auf den Markt. Mit den Erfahrungen, die Sie mit Arcon gewonnen hatten, wurde ein objektorientiertes Programm, welches von Grund auf auf moderne Konzepten ausgerichtet war, neu entwickelt.

Orbit reduziert das Programmmenü auf das Mindeste, sämtliche Arbeitsfunktionen wurden in die Toolbars ausgelagert. Das Menü beinhaltet nur noch die notwendigsten Funktionalitäten, die standardmäßig in jedem Windowsprogramm vertreten sind und sich nur schlecht in die Toolbars auslagern lassen.

Das Hauptsteuerungselement des Programms ist die Toolbar mit der Wasleiste. Über diese wird gesteuert, welches Objekt erzeugt werden soll. Die einzelnen Objekttypen sind

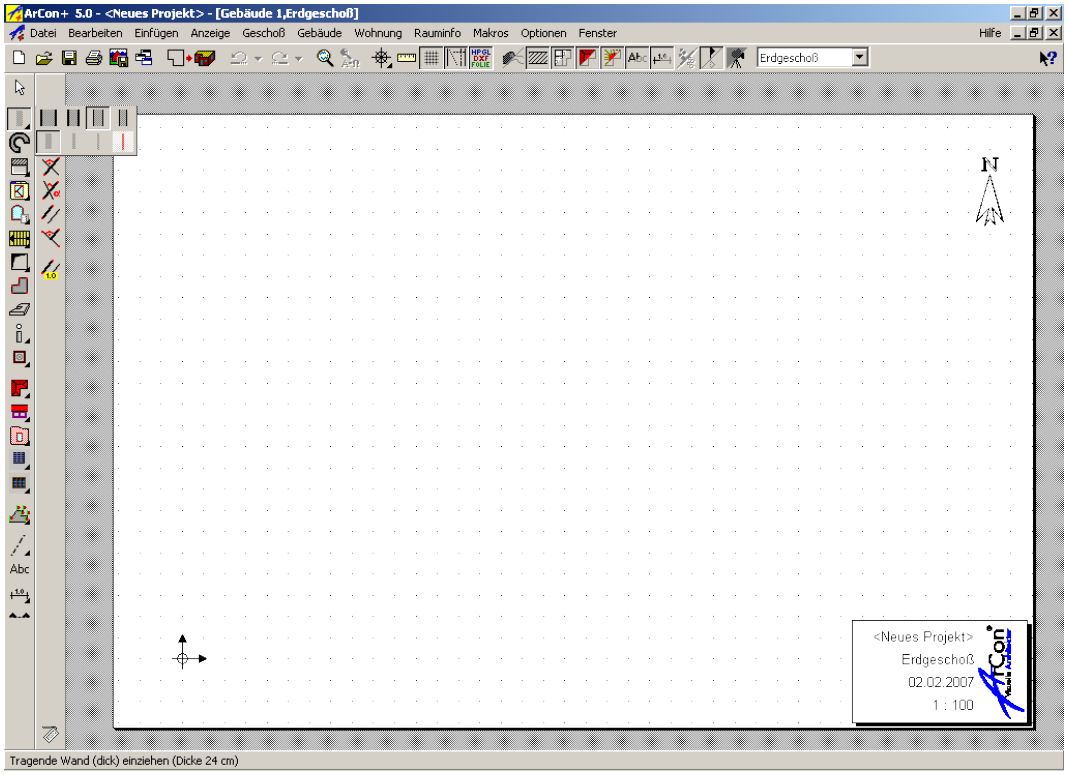


Abbildung 8.2: Oberfläche von Arcon 5.0

in Gruppen eingeteilt, denen jeweils eine eigene Registrierkarte zugeordnet wurde. Die wesentlichen Eigenschaften des Objektes können über die Wie-Leiste gesteuert werden, dies umfasst die Konstruktionsart aber auch die wichtigsten Eigenschaften die nicht über die Konstruktion erzeugt werden. Bei Wänden gehören z. B. Wandaufbau, Baustoff, Höhe und Dicke der Wand dazu. Die weiteren Einstellungen können über den Eigenschaftsdialog eingestellt werden, dieser kann über einen Knopf in der Wasleiste aufgerufen werden. Diese Einstellungen können als Formatvorlagen abgespeichert und später für andere Objekte wieder verwendet werden.

8.2 Moderne CAD-Oberfläche

Die Abbildung 8.4 zeigt die Benutzeroberfläche des CAD-Programms Vicado der mb AEC Software GmbH an dem die Hauptbestandteile einer CAD-Oberfläche erläutert werden.

Die Hauptbestandteile des GUI sind:

- Menü
- Die Statusleiste. Sie dient zur Anzeige von Informationen und dem Programmsta-

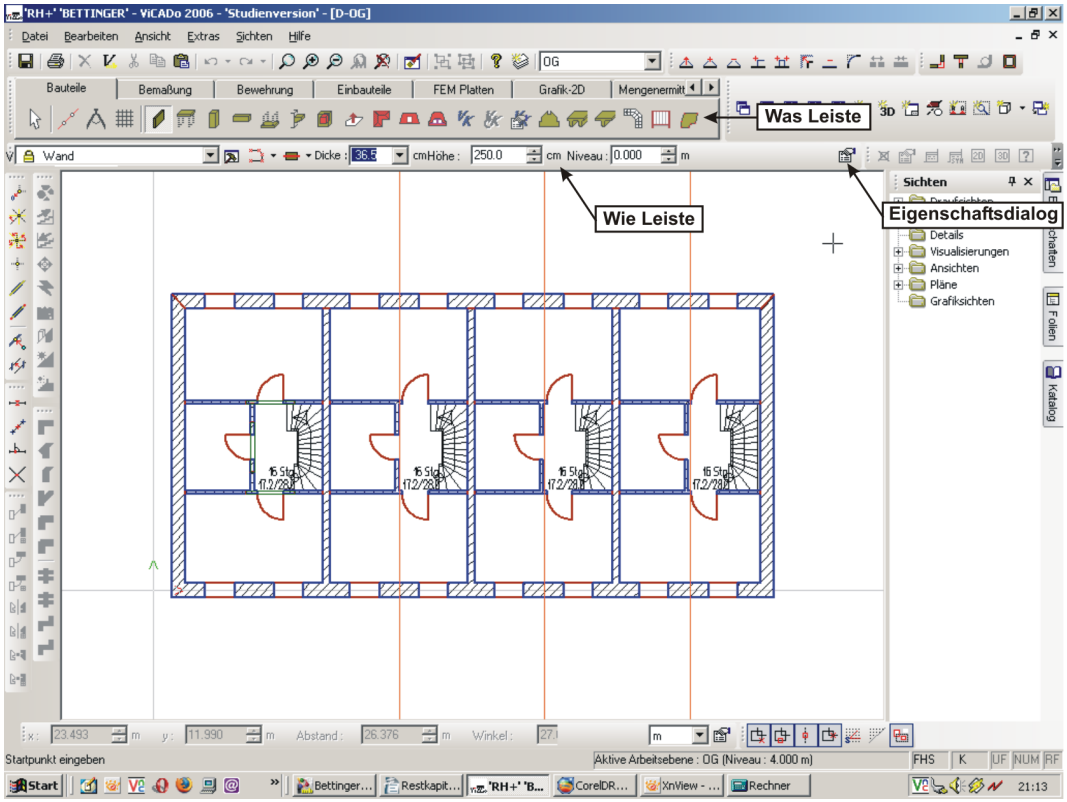


Abbildung 8.3: Oberfläche von Vicado

tus. Sie ist unterteilt in mehrere Bereiche, Teile in denen sich die Informationen dynamisch ändern und Bereiche die bestimmte Zustände anzeigen.

- Toolbars, auch Werkzeugkästen genannt, enthalten Knöpfe über die verschiedene Befehle ausgewählt werden können.

  - Leiste mit Knöpfen. Da es sich bei Toolbars um von CWnd abgeleitete Objekte handelt, können diese nicht nur die typischen Buttons, sondern jedes beliebige von CWnd abgeleitete Objekt wie z. B. Auswahlbox aufnehmen.
  - Da die Bilder der Toolbarbuttons nicht immer aussagekräftig sind, werden in der Statusleiste Erläuterungen zu den Knöpfen eingeblendet. Später wurden die so genannten Tooltips eingeführt, kleine Textfenster mit Erklärungen, die sich einblenden, wenn die Maus sich über dem Knopf befindet.

- Angedockte Fenster. Durch die angedockten Fenster wird die Arbeitsfläche nicht verdeckt, jedoch verkleinern sie diese.
- Views, der eigentliche Arbeitsbereich des Programms

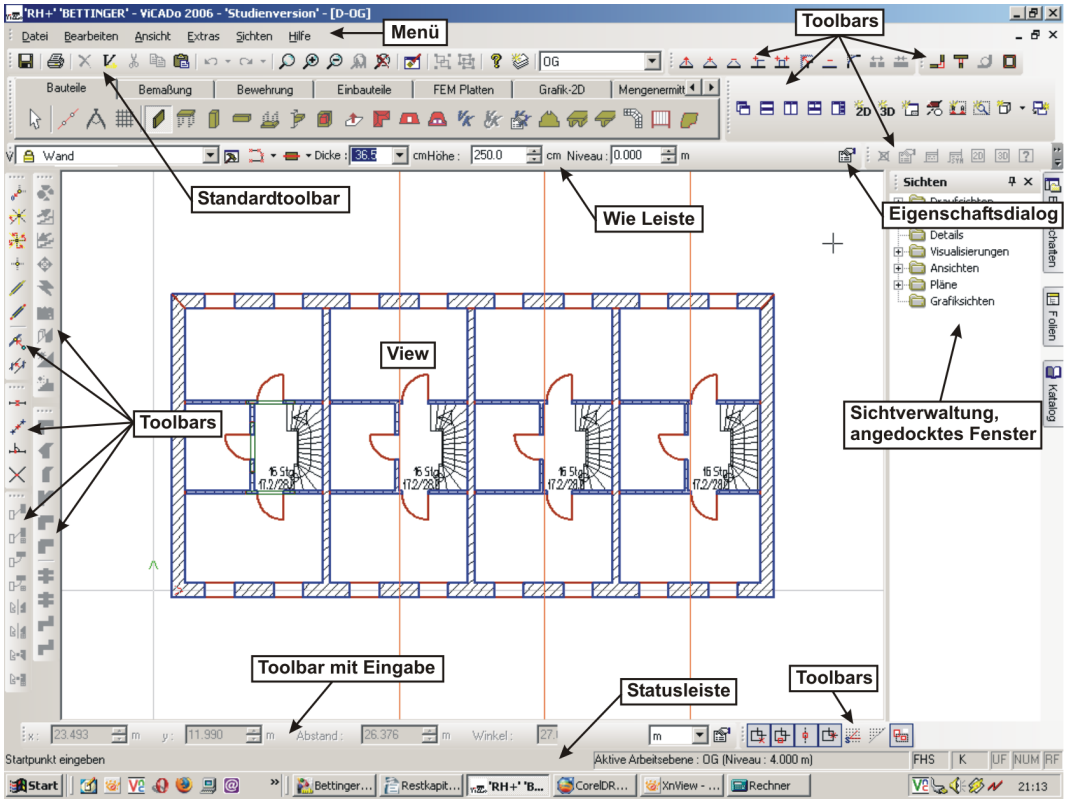


Abbildung 8.4: Oberfläche von Vicado

8.2.1 Menü des Programmes Vicado

Das Menü von Vicado ist begrenzt auf das Mindeste. Es enthält Standardeinträge, die in jedem Programm vorhanden sind und Einträge die sich nicht ohne weiteres auf Toolbars verteilen lassen.

Die Einteilung des Menüs ist windowskonform, sie orientiert sich an der Einteilung und an den Bezeichnungen der typischen Windowsprogramme.

Standard Funktionen Alle Funktionen, die mit dem Laden, Speichern, Drucken, neue Projekte anlegen und Programm beenden zusammenhängen

Bearbeiten Allgemeine Funktionen Rückgängig, Wiederherstellen, kopieren etc.

Ansicht Steuert Sichtbarkeit der Teile der Programmoberfläche und die Sichteigenschaften der aktiven Sicht.

Extra Modell- und Programmdaten und -einstellungen

Sichten Erzeugen und Anordnen der Sichten, Normal meist Fenster genannt, da die meisten Programme nur eine Sicht besitzen

Datei	Bearbeiten	Ansicht	Extra	Sichten	Hilfe
Schließen	Rückgängig	Symbolleisten	Einstellung	Neue 2D Sicht	Hilfsthemen
Speichern	Wiederherstellen	Statuszeile	Anpassen	Neue 3D Sicht	Info über Vicado
Speichern unter	Eigenschaften	Eigenschaftsleiste	Modell-Versionen	Neue Planzusammenstellung	
Alles Speichern	Ausschneiden	Sichtenverwaltung	Maßrastereinstellungen	Neue Graphiksicht	
Folie neu	Kopieren	Folienverwaltung	Stammdatenverwaltung	Sicht dublizieren	
Modelleigenschaft	Einfügen	Katalog	Variableneditor	Überlappend	
Drucken	Löschen	Nummerische Eingabe	Stifte	Nebeneinander	
Seitenansicht	Gruppieren	Raster	Linientype	Übereinander	
Importieren/Exportieren	Gruppierung aufheben	Arbeitsebene	Schraffuren	Über- und Nebeneinader	
"history"Dateien	Neues Objekt einfügen	Auffrischen		Standardanordnung	
Beenden		Sichtbarkeit			
		Sichteigenschaften			
		Zoom			
		Modellstruktur			
		Ganzer Bildschirm			

Abbildung 8.5: Menü des CAD Programmes Vicado

Hilfe Hilfe zum Programm und Informationen über das Programm

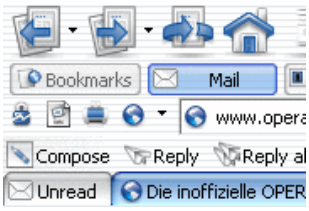
Alle weiteren Funktionen für die Konstruktion und die Bearbeitung des Modells können nur direkt über die graphische Oberfläche angesprochen werden.

8.3 Look and feel

Unter „look und feel“ versteht man das Aussehen und die Handhabung von Programmen. Die Programme sollen ein gewohntes Erscheinungsbild liefern, das der Anwender kennt und indem er sich „wohlfühlt“. Somit soll ein intuitiveres Arbeiten mit dem Programm ermöglicht werden. Das „look and feel“ wird von Programmen geprägt, die eine große Verbreitung besitzen. In diesem Fall sind es nicht nur unbedingt CAD-Programme, sondern Programme, die jeder kennt und verwendet, wie z. B. der Internet Explorer oder das MS-Office Paket, hierbei besonders die Textverarbeitung Word und das Tabellenkalkulationsprogramm Excel. Bei einem Marktanteil dieser Produkte von bis zu 90% kann man davon ausgehen, dass jeder Anwender sie kennt und mit ihrer Funktionsweise vertraut ist. Hierdurch kommt die Erwartungshaltung des Anwenders, dass die anderen Programme genauso aussehen und sich auch entsprechend verhalten sollen.

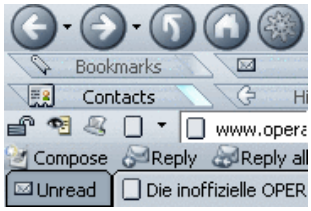
Ein interessantes Konzept für ein gewohntes „look and feel“ bietet der Softwarehersteller Opera für die Oberfläche seines Browser, für den er verschiedene Skins anbietet. Diese Skins imitieren optisch die Oberfläche der Konkurrenzprodukte IE und Firefox/Mozilla,

der Umsteiger bekommt damit eine optisch gewohnte Oberfläche geboten, was den Umstieg erleichtert.



(a) Coca Skin

(b) IE6 XP Skin



(c) Netscape Skin

Abbildung 8.6: Verschiedene Opera Skins

Eine Erweiterung dieses Konzeptes ist die Anpassung der Menüs und der Toolbars auf die gleiche Art und Weise, wenn eine freie Konfiguration dieser Teile schon bei der Programmierung der Anwendung berücksichtigt wurde. Die Anpassung kann dann durch Änderung einer Konfigurationsdatei auf einfache Art und Weise geschehen. Der Anwender findet dann die Toolbuttons und Menüpunkte an den gewohnten Punkten. Eine weitere Erweiterung auf vereinheitlichte Dialoge ist nicht mehr mit einem vertretbaren Aufwand machbar, da die Programmierung der Oberfläche einen Großteil des Programmieraufwandes eines Programms ausmacht.

Das Look and feel des Programms muss eine Balance finden zwischen den Anforderungen der verschiedenen Anwendern Bild 8.7.

Abbildung 8.7: Dreieck Oberfläche

- Der Anwender will sich bei einer neuen Version eines Programms nicht umstellen
- Der Umsteiger erwartet, dass das Programm so funktioniert wie er es vom Konkurrenzprodukt gewohnt ist.

- Der Neueinsteiger will eine leicht verständliche Bedienung, er versucht Parallelen zu bekannten anderen Programmen zu ziehen.

Die einzelnen Ansprüche widersprechen sich teilweise, ein möglicher Weg ist die Anpassung der Programme an einen Marktführer wie z. B. Mircosoft um ein einheitliches Bedienungskonzept zu erhalten. Die Gefahr dabei ist jedoch, dass die Individualität der Programme und eigene Konzepte verloren gehen, wenn diese dem vorgegebenen Konzept widersprechen, selbst wenn diese besser sind, da sie mit dem gewünschten „look and feel“ nicht übereinstimmen. Durch die Anpassung des „look and feel“ werden die Programme austauschbar. Dies entspricht Wünschen des Anwenders, jedoch nicht unbedingt denen des Herstellers.

8.4 Programmsteuerung

Im Allgemeinen geschieht die Programmsteuerung durch Auswahl eines Steuerelementes mit der Maus und klicken einer Maustaste. Daneben existieren weitere interessante Konzepte. Die AutoCAD-Familie bietet verschiedene Funktionalitäten, die mit der Maus gesteuert werden.

- Zoom

  AutoCAD besitzt nur einen Button für die Zoomfunktionen vergrößern und verkleinern. Welche Variante des Zoombefehls ausgeführt werden soll, wird durch die Richtung der Mausbewegung bestimmt.

  - Zoom in: Maus nach rechts unten.
  - Zoom out: Maus nach links oben.

- Auswahl

  Ähnliche wie der Zoom funktioniert auch die Auswahlbox. Durch die Richtung der Mausbewegung wird bestimmt ob nur die Objekte, die in der Box liegen gefangen werden sollen oder auch die Objekte die von der Box gekreuzt werden.

  - In Box: Maus nach rechts unten. Es werden nur Objekte ausgewählt, die komplett im Auswahlbereich liegen.
  - Kreuzen: Maus nach links oben. Es werden Objekte ausgewählt die komplett oder teilweise in der Auswahlbox liegen.

Die Mausfunktionen ersparen dem Anwender Mausbewegungen und Mausklicks, somit beschleunigen sie das Arbeiten.

8.4.1 Mausgesten

Mausgesten sind ein Konzept zur Steuerung von Programmen, sie wurden im Jahr 2000 von der Norwegischen Firma Opera für ihren Webbrowser eingeführt[1]. Mausgesten sind definierte Bewegungen der Maus, denen ein bestimmter Befehl zugeordnet ist. Mit Hilfe der Mausgesten kann in Opera u. a. zwischen einzelnen geöffneten Seiten navigiert werden, Fenster geschlossen, kopiert oder minimiert werden und eine Seite zurück oder vor im Fensterverlauf gewechselt werden.

Im Bereich der CAD-Programme haben sie bis jetzt noch keinen Einzug gefunden, jedoch wären sie auch hier z. B. für die folgenden Funktionen interessant: Rückgängig und Wiederherstellen, navigieren durch die einzelnen Views und Views duplizieren.

[1] Wikipedia, http://de.wikipedia.org/wiki/Mausgesten, Version 17:14, 17. Jan. 2007

9 Dynamische Konstruktionshilfe

Die dynamischen Konstruktionshilfen sind eine intelligente Unterstützung der Anwender während der Konstruktion.

In einer Erörterung des „Mooreschen Gesetzes“ [1] bezieht sich Raymond Kurzweil nicht auf die „Transistoren pro Chip“ sondern auf die Rechenleistung eines 1000 Dollar Computers. Er betrachtet dabei die Leistung seit der Einführung mechanischer Rechenmaschinen über Röhren bis zu den heutigen PCs. In den Anfangsjahren 1910-1950 verdoppelte sich die Rechenleistung alle drei Jahre, 1950-1966 alle zwei Jahre und seitdem ungefähr jährlich. Der 1000 Dollar (Euro) Rechner ist eine interessante Größe, da es sich bei ihm um einen durchschnittlichen PC handelt, der auch in vielen Ingenieurbüros verwendet wird.

Betrachtet man die erforderliche Rechenleistung für CAD im gleichen Zeitraum, so stellt man fest, dass der Leistungsbedarf stufenweise steigt und größtenteils von der Art und der Funktionalität des Geometriemodells abhängig ist.

Bei den Standardanwendungen von CAD-Programmen ist bei den Computern eine große Leistungsreserve vorhanden, die nicht genutzt wird. Die größte Rechenleistung wird bei 3D-CAD Programmen bei der Berechnung der Verschneidungen und der Visualisierung benötigt, während der Konstruktion sind bei Rechnern noch Reserven vorhanden.

Die Konstruktion erfolgt im klassischen Sinn in zwei Schritten. Zuerst wählt der Programmnutzer aus wie die Konstruktion erfolgen soll, im zweiten Schritt erfolgt die eigentliche Konstruktion.

Die dynamischen Konstruktionshilfen arbeiten während der Konstruktion in den „Rechnerpausen“ um dem Zeichner die Arbeit zu erleichtern. In diesen Zeiträumen berechnet das Programm alle möglichen Schnittpunkte, Lote, „Fangen auf Linien“ und zeigt sie dem Anwender an, sofern sich die Maus in der Nähe der berechneten Punkte befindet.

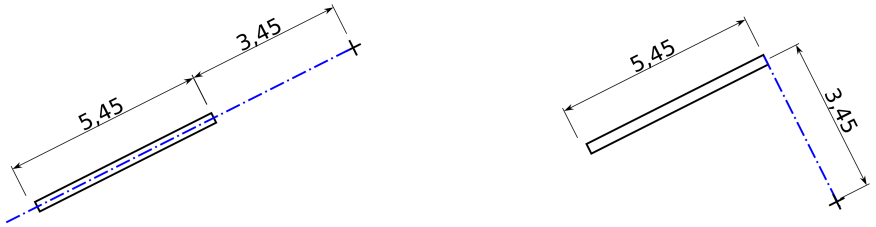
[1] Das Mooresche Gesetz sagt aus, dass sich die Komplexität integrierter Schaltkreise mit minimalen Komponentenkosten etwa alle zwei Jahre verdoppelt.

Der Anwender erspart sich die Auswahl der gewollten Konstruktion. Die Konstruktion erfolgt schneller, da es z. B. ausreicht in die Nähe eines Schnittpunktes zweier Geraden mit der Maus zu kommen um diesen Punkt auszuwählen und nicht mehr die beiden Geraden auszuwählen. Die Ersparnis liegt in diesem Fall bei zwei Mausklicks und einem Teil der Mausbewegung.

Die dynamischen Konstruktionshilfen können die bisherigen Konstruktionsmethoden nicht komplett ersetzen, besonders wenn

1. sich auf Linien bezogen wird, die außerhalb des aktuellen Bildausschnitts liegen, z. B. bei großen Plänen
2. viele Linien/Schnittpunkte im Zielbereich existieren, sodass der gewünschte Punkt schlecht auswählbar ist.

Eines der wenigen Programme, das die dynamischen Konstruktionshilfen einsetzt, ist das Programm Revit der Firma Autodesk. Die dynamischen Konstruktionshilfen entsprechen den typischen Punktkonstruktionen von CAD-Programmen. Hierzu gehören:



(a) Verlängerung Achse (b) Verlängerung kurze Seite

Abbildung 9.1: Konstruktionshilfe Achse

- Verlängerung der Achse

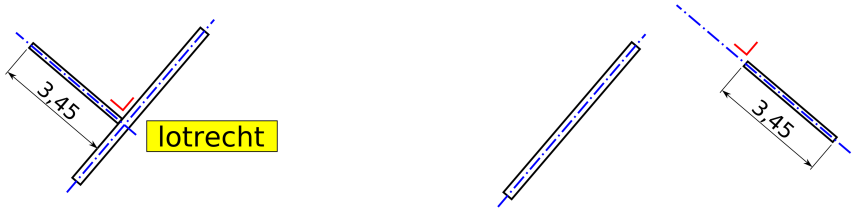
  Befindet sich die Maus über der Achse eines Bauteils oder deren Verlängerung (Abb. 9.1(a)), so wird diese Achse und der Abstand zum Bauteil angezeigt.

- Verlängerung von Seiten

  Befindet sich die Maus über der Verlängerung einer Seite eines Bauteils (Abb. 9.1(b)), so wird diese Verlängerung und der Abstand zum Bauteil angezeigt.

- Lot auf Achse

  Befindet sich die Maus während der Eingabe einer Strecke auf einer Achse und steht diese Strecke senkrecht auf ihr (Abb. 9.2(a)), wird dies durch ein Symbol signalisiert.

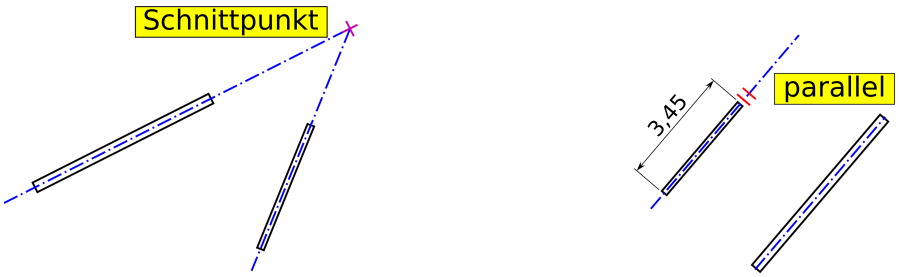


(a) Lot auf Bauteil (b) Lotrecht

Abbildung 9.2: Konstruktionshilfe Lot

- Lot auf Achsverlängerung

  Befindet sich bei der Eingabe einer Strecke der Mauszeiger auf einer Verlängerung einer Achse und steht diese Strecke senkrecht auf ihr Abb. 9.2(b), wird dies durch ein Symbol signalisiert.



(a) Schnittpunkt (b) Parallel

Abbildung 9.3: Konstruktionshilfe Schnittpunkt und Parallele

- Schnittpunkt

  Befindet sich die Maus im Schnittpunkt zweier Achsen bzw. Achsverlängerungen (Abb. 9.3(a)), so wird dieses durch ein Symbol signalisiert.

- Parallel

  Ist eine Konstruktion parallel zu einer Bauteilachse (Abb. 9.3(b)), wird dieses durch ein Symbol angezeigt.

  Die in den Abbildungen 9.2(a) bis 9.3(b) dargestellten Konstruktionshilfen lassen sich um beliebige weitere Punkte ergänzen.

10 Bauspezifisches Datenmodell

Der zentrale Bestandteil des CAD-Programms ist das Datenmodell. Das Datenmodell bestimmt durch die dazugehörigen Objekte und Funktionalitäten den Funktionsumfang und den Anwendungsbereich eines CAD-Programms. Die verschiedenen Anwendungsbereiche eines CAD-Programms

- Bauwesen
- Maschinenbau
- Elektrotechnik
- etc...

bedingen unterschiedliche Datenmodelle und Funktionalitäten. Von den verschiedenen möglichen Anwendungsbereichen eines CAD-Programms wird im Weiteren nur die Verwendung im Bauwesen betrachtet. Auch im Bauwesen gibt es verschiedene Anwendungsbereiche für CAD-Programme, welche unterschiedliche Anforderungen an das Datenmodell stellen:

- Architektur
- Ingenieurbau
- Innenarchitektur
- etc.

Der Unterschied liegt in den verwendeten Bauteilen, den zu erzeugenden Plänen (Entwurfspläne, Ausführungspläne, Schal- und Bewehrungspläne) und der Auswertung des Datenmodells (Wohnflächenermittlung, Kostenermittlung, etc.).

Zu dem Thema Datenmodell in CAD-Systemen im Bauwesen entstanden mehrere Dissertationen, u. a. an der TU Kaiserslautern. Dr. Peter Heck beschäftigte sich in seiner Arbeit (Hec98) mit einem objektorientierten Datenmodell für die raum- und bauteilorientierte

Bearbeitung von Gebäuden in der Vorplanung. Er entwickelte ein objektorientiertes Datenmodell mit verschiedenen Sichten (Bauteilsicht, Raumsicht) für die Vorplanungsphase und der Funktionalität zur Flächen-, Rauminhalt- und Kostenermittlung.

In seiner Arbeit entwickelte Dr. Samer Hamwi (Ham00) ein objektorientiertes Datenmodell für die Tragwerksplanung. Aus diesem Datenmodell können die Informationen für die Berechnung des statischen Systems mit einem FEM-Programm gewonnen werden.

10.1 Zentrales Objektmodell

Moderne CAD-Programme besitzen ein Zentrales Datenmodell (ZDM). Während frühere Datenmodelle anwendungsspezifisch waren, sind die aktuellen Datenmodelle allgemeine Modelle, welche die gesamte Planung eines Bauwerkes mit Entwurf, Ausführungsplanung, Bauwerksberechnung und Bauphysik abdecken. Das ZDM kann sich aus mehreren Teilmodellen zusammensetzen, welche die unterschiedlichen Aspekte des Gebäudemodells beschreiben. Mögliche Teilsichten des Datenmodells sind u. a.:

- Bauteilsicht
- Raumsicht,
- FEM-Sicht

Die typische Sicht auf das Datenmodell ist die Bauteilsicht, da sich die Grundrisse aus den einzelnen Bauteilen (Wände) zusammensetzen.

10.1.1 Einteilung der Objekte

Die Elemente die in einem CAD-Programm im Bauwesen verwendet werden, lassen sich nach dem folgenden Schema gliedern:

- Bauteile
  - Einfache Bauteile
    - Wand, Stütze
    - Unterzug, Balken
    - Öffnung, Fenster, Tür
    - Decke, Fundament

    - Komplexe Bauteile
        - Dach, Gaube, Dachfenster
        - Treppe, Geländer
        - Gelände
- Bemaßung
    - Einzelmaß
    - Maßkette
- Bewehrung
- Graphische Elemente
    - Punkt, Gerade, Bogen, Kreis, Polygon, Rechteck, Ellipse, Ellipsenbogen
    - Bild
    - Text

Diese Gliederung beschränkt sich auf CAD-Programme für die Entwurfs- und Ausführungsplanung. Bei anderen Anwendungsbereichen z. B. technische Gebäudeausrüstung kommen weitere Bauteilgruppen hinzu. Mögliche ergänzende Gruppen sind:

- Rohre incl. der Armaturen
- Elektroleitungen
- Heiz- und Lüftungsrohre

10.1.2 Klassenhierarchie des Datenmodells

Die Klassenhierarchie des Datenmodells für ein CAD-Programm im Bauwesen ist auszugsweise in Abbildung 10.1 dargestellt. Alle Bauteile sind von einer Basisklasse abgeleitet. Die Basisklasse enthält die grundlegenden Funktionen und Eigenschaften der Bauteile.

In der ersten Ableitungsebene sind die verschiedenen Objektgruppen, von denen die einzelnen Objekte abgeleitet werden. Die Gruppen sind:

- Hilfslinien
- 2D Graphik
- Maßlinien
- Bewehrung

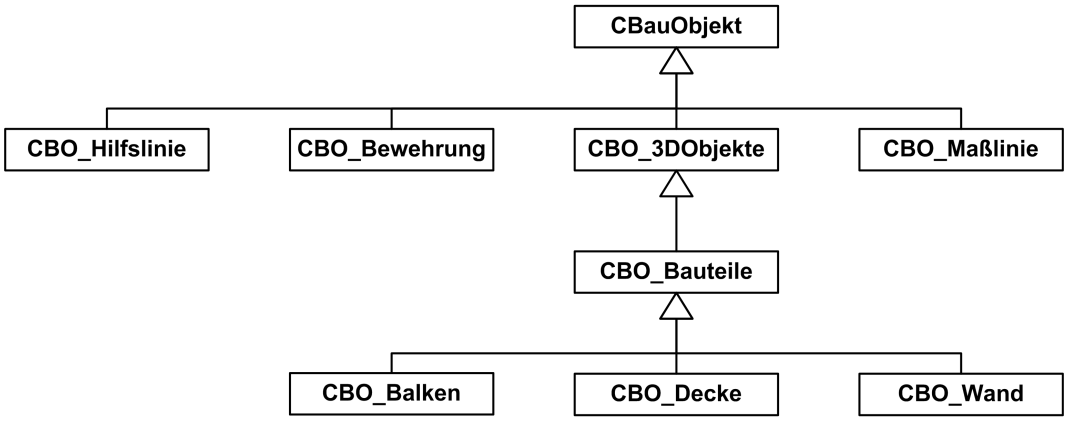


Abbildung 10.1: Ableitungshierarchie des Datenmodells

- 3D Objekte

Die Gruppe 3D-Objekte ist die Oberklasse für alle Objekte, welche das 3D-Geometriemodell für die Beschreibung der Geometrie verwenden. In erster Linie sind dies die Bauteile aus denen sich das Gebäude zusammensetzt, hinzu kommen allgemeine 3D-Objekte. Die unterste Ebene bilden die einzelnen Objekte, die in dem Programm verwendet werden.

10.1.3 Eigenschaften der Bauteile

Die Objekte eines CAD-Programms repräsentieren reale Bauteile. Diese Bauteile besitzen verschiedene Eigenschaften, die auch im Programm repräsentiert werden. Die Eigenschaften eines Bauteils sind die Daten und Funktionen, welche ihre Eigenschaften beschreiben. Alle Bauteile besitzen einen gleichen Grundstock an Eigenschaften. Durch das Hinzufügen weiterer Eigenschaften können daraus neue Bauteile gebildet werden. Die Informationen, welche die Bauteile in einem CAD-Programm besitzen sind abhängig vom Verwendungszweck des Programms. Die Eigenschaften der Bauteile sind:

- Geometrie
- Topologie
- Material
- Abmessungen Länge, Breite, Höhe
- Mittellinie
- Mittelpunkt des Objektes - Setzpunkt, Fangpunkt

Die wichtigste Eigenschaft eines Bauteils ist seine Geometrie, welche die Form des Bauteils beschreibt, hinzu kommt die Topologie, die die Lage im Raum beschreibt. Die Geometrie

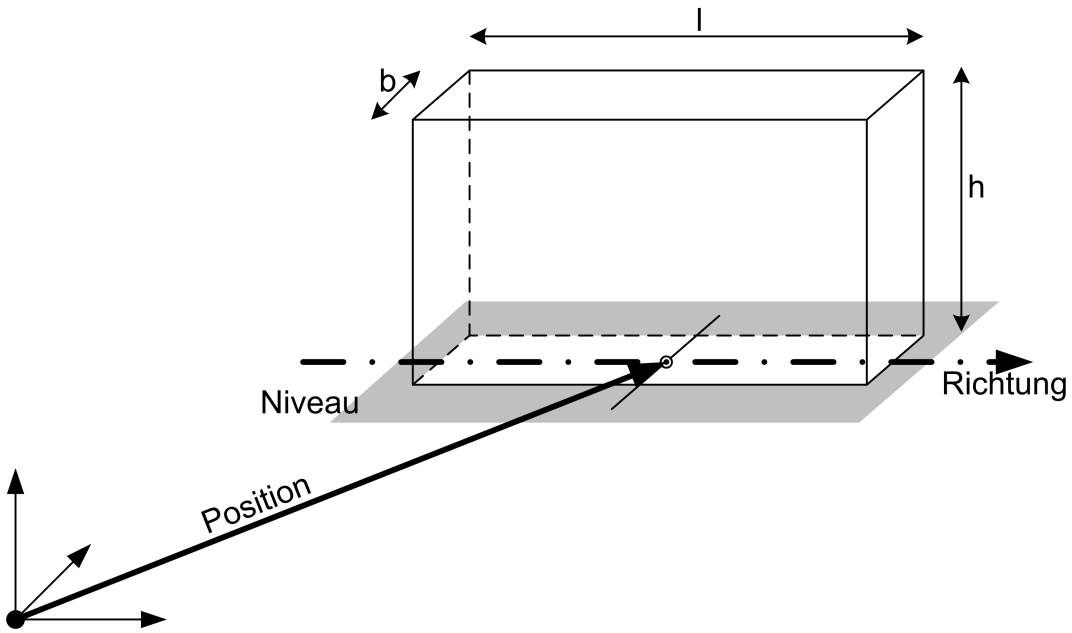


Abbildung 10.2: Darstellung von Bauteileigenschaften

der Bauteile wird in modernen Programmen durch ein 3D-Geometriemodell beschrieben (siehe Kapitel 11). Die Speicherung der Geometrie durch Parameter ist für die Ausgangsgeometrie möglich, jedoch kaum noch nach Änderung der Geometrie durch den Anwender. Die Haltung Bauteilparameter Länge, Breite, Höhe etc. ermöglichen eine einfache Änderung der Bauteilegeometrie oder die Wiederherstellung der Ausgangsgeometrie nach Manipulation der Bauteile. Eine weitere Eigenschaft der Bauteile ist das Material. Über dies werden verschiedene physikalische Eigenschaften der Objekte beschrieben. Diese werden für die Darstellung (Oberfläche) oder auch für weitere Anwendungen wie Wärmeschutznachweis, oder Tragwerksplanung (Wärmeleitfähigkeit, E-Modul), je nach Ausrichtung des Programms, verwendet. Die physikalischen Eigenschaften, Materialien etc. können direkt beim Bauteil gespeichert oder auch zentral durch eine Datenbank zur Verfügung gestellt werden. Bei einer zentralen Speicherung wird im Bauteil nur auf den Eintrag in der Datenbank verwiesen. Die Daten der Datenbank können von allen Bauteilen verwendet werden, neue Daten müssen nur einmal in die Datenbank eingetragen werden.

10.1.4 Funktionalität der Bauteile

Die Funktionalität der Bauteile lässt sich in Grundfunktionalität, Gruppen- und objektspezifische Funktionalität unterteilen. Die Grundfunktionalität ist bei allen Objekten gleich, sie wird in der Basisklasse deklariert, meist aber erst in den abgeleiteten Klassen implementiert, da sie an die einzelnen Bauteile angepasst wird. Zu den Grundfunktionen

der Objekte zählen die Funktionen zum

- Laden und Speichern der Objekte
- Transformieren der Objekte
- Erzeugen von Objekten durch Transformieren und Kopieren

Die gruppenspezifische Funktionalität beinhaltet die Funktionen, welche eine Gruppe zusammengehörender Bauteile besitzt:

- Manipulation der Geometrie

Die objektspezifische Funktionalität sind Funktionen die nur bei einem bestimmten Objekttyp vorkommt. Hierzu zählen:

- Funktionen zum Ändern der bauteilspezifischen Eigenschaften

Die implementierte Funktionalität der Bauteile wird bestimmt vom vorgesehenen Anwendungsbereich des Programms.

11 Geometriemodelle in CAD Programmen

11.1 Die Bedeutung der Geometriemodelle

Das Geometriemodell ist einer der wichtigsten Bestandteile der CAD-Programme. Nach den 2D-Modellen folgten die 2 1/2D und 2 3/4D-Modelle, gegen Mitte der 90er Jahre folgten die 3D-Modelle für die breite Masse der CAD-Programme, gleichzeitig mit der Umstellung der Programme auf Objektorientierung. Das Geometriemodell hat mehrere Aufgaben in einer CAD-Anwendung; zum einen ist es die Repräsentation der Geometrie des Objektes, die physikalischen Eigenschaften (Volumen, Oberfläche, Trägheitsmoment, etc.) des Objektes, zum anderen wird das Modell benötigt, um Beziehungen zwischen einzelnen Objekten zu definieren.

11.2 Auswahlkriterien für ein Geometriemodell

Bei der Auswahl des Geometriemodells sind verschiedene Auswahlkriterien zu berücksichtigen, die sich gegenseitig beeinflussen. Bei der Auswahl des passenden Geometriemodells muss zwischen den einzelnen Kriterien ein vertretbarer Kompromiss getroffen werden. Dabei muss auch die Hardware, auf der das Programm laufen soll, berücksichtigt werden,

- Geschwindigkeit

  Die Geschwindigkeit ist das wichtigste Kriterium für die Auswahl, da davon der Benutzerkomfort bei der Anwendung abhängt. Das Programm muss direkt auf die Eingabe der Anwender reagieren und diese umsetzten.

- Speicher

  Der Speicherbedarf des Geometriemodells sollte so groß wie nötig und so gering wie möglich sein. Großer Speicherbedarf hat meist geringere Geschwindigkeit zur Folge.

- Genauigkeit

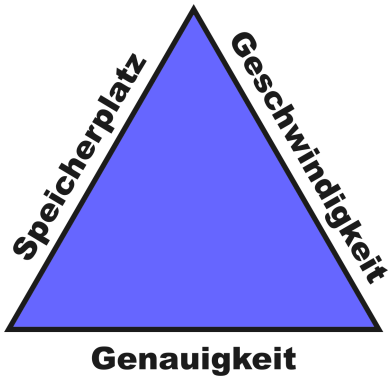


Abbildung 11.1: Ansprüche an die Geometriemodelle

Das Einsatzgebiet des Programms bestimmt die benötigte Genauigkeit des Modells. Während im Maschinenbau die Genauigkeit bei Bruchteilen von Millimetern liegt, ist sie im Bauwesen im Millimeterbereich. Große Genauigkeit hat meist negative Auswirkungen auf die Geschwindigkeit und den Speicherbedarf des Programms. Die Genauigkeit muss so groß sein, dass sie die Ansprüche des Modells erfüllen kann.

11.2.1 Vor- und Nachteile der einzelnen Geometriemodelle

Im Folgenden werden die Vor- und Nachteile der beiden meist verbreiteten Geometriemodelle zusammengefasst:

- CSG Modell

  Die CSG-Modelle sind sehr genau bei einem geringen Speicherbedarf. Der Nachteil der CSG-Modelle ist die geringere Geschwindigkeit, da für jede Berechnung der Verschneidungen und der Darstellung das Modell jeweils neu berechnet werden muss. Mischformen für eine Erhöhung der Geschwindigkeit haben einen höheren Speicherbedarf zur Folge. Einzelne Teile des Modells wie Kanten und Seiten können nicht referenziert werden.

- Brep-Modell

  Der Vorteil der Brep-Modelle (Facettenmodelle) ist ihre hohe Geschwindigkeit, da die Außenflächen immer vorhanden sind und nicht extra berechnet werden müssen, dies geht aber zulasten eines höheren Speicherbedarfs. Bei gekrümmten Flächen steigt der Speicherbedarf stark. Die vielen Teilflächen haben bei Verschneidung einen höheren Aufwand und damit Geschwindigkeitseinbußen zur Folge. Die weiteren, genaueren Formen des Brep-Modells sind aufwendiger in der Implementierung und

langsamer in der Ausführung, da die Algorithmen aufwendiger sind. Die einzelnen Flächen des Brep-Modells können referenziert werden.

11.3 Geometriemodell für das Bauwesen

Das Brep-Modell stellt einen guten Kompromiss für die Erfüllung der Anforderungen an ein Geometriemodell im Bauwesen dar. Die im Bauwesen gestellten Anforderungen an die Genauigkeit werden durch das Brep-Modell erfüllt. Die geringe Genauigkeit, der größere Rechenaufwand bei gekrümmten Flächen infolge des Facettenmodells haben nur wenig Auswirkungen auf die Anwendung, da die meisten Objekte ebene Flächen besitzen. Nachteilig wirkt sich das Modell bei der Visualisierung aus, da hier runde Objekte Kanten besitzen. Dies lässt sich jedoch bei der Visualisierung durch Smoothing-Algorithmen, die die Kanten „ausrunden“ großteils optisch kaschieren.

Die Flächen des Brep-Modells bieten die Möglichkeit, direkt angesprochen zu werden. Die Modellflächen stehen für Flächen, die auch beim realen Bauteil existieren. Die Flächen beinhalten, wie auch das Volumen, wichtige Informationen für die Bauausführung. Sie können für die Ermittlung der Massen von Maler, Verputzer, Fliesenleger und Estricharbeiten verwendet werden. Große Bedeutung haben die Flächen auch bei den Rohbauarbeiten von Stahlbetonbauwerken, da die Flächen des Objektes gleich der Schalfläche sind.

Die 3D-CAD-Programme verwenden die Geometriemodelle zur Berechnung beliebiger Schnitte und Ansichten. In Kombination mit dem MVC-Konzept (siehe Kapitel 4) ermöglicht dies, im Gegensatz zu den 2D-Modellen, dass sich alle Zeichnungen nach einer Änderung an dem Modell auf dem aktuellen Stand befinden.

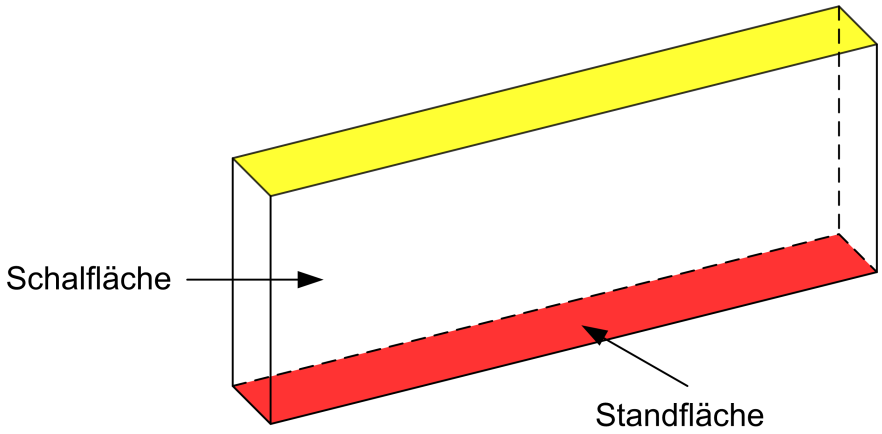


Abbildung 11.2: Bedeutung von Flächen am Beispiel einer Wand

11.3.1 Bedeutung der Geometriemodelle

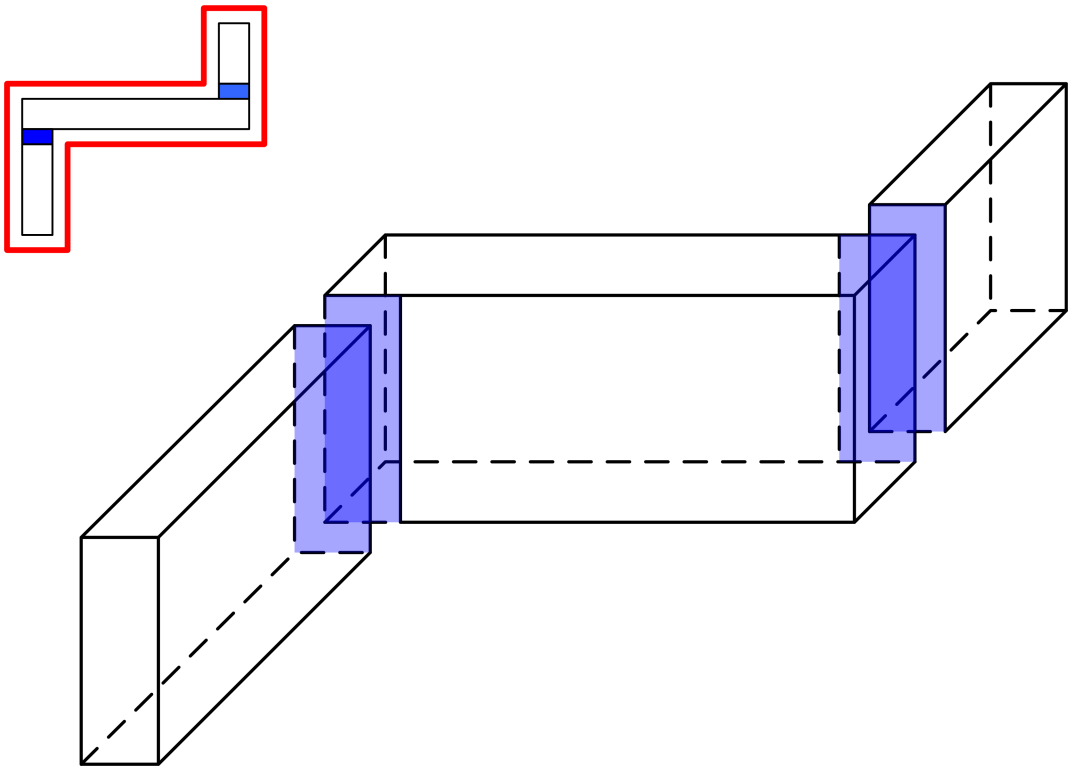


Abbildung 11.3: Schalflächen eines Wandzuges

Die Abbildung 11.2 zeigt die sechs Flächen einer Betonwand. Die einzelnen Flächen habe eine unterschiedliche Bedeutung für den Bauablauf. Die rote Fläche ist die Grundfläche, auf der die Wand steht. Über diese Fläche hat das Bauteil Kontakt zu anderen Objekten. Die Fläche benötigt aber keine Schalung oder weitere Bearbeitung wie Verputzen. Weiß sind die seitlichen Flächen der Wand. Diese Flächen benötigen eine Schalung und werden meist später weiterbearbeitet. Die Wand wird durch die gelbe Fläche nach oben abgeschlossen. Diese Fläche benötigt in der Regel auch keine Schalung, in wieweit die Fläche später eine Weiterbearbeitung erfährt ist vom Gesamtaufbau abhängig. Berührt die Wand über diese Fläche eine Decke, gibt es keine Weiterbearbeitung, ist es eine freie Fläche hängt es von der weiteren geplanten Verwendung ab.

Abbildung 11.3 zeigt einen Ausschnitt aus einem Wandzug. Bei diesem Wandzug kommt ein weiterer Effekt zum Vorschein. Während bei einer einzelnen Wand die Schalfläche der „Umfangsfläche“ entspricht, fallen bei einem Wandzug einzelne Teile weg. Die Stirnflächen der beiden senkrechten Wände und die dazugehörige Kontaktfläche (blau markiert) der horizontalen Wand benötigen keine Schalung. Es bleibt nur die rote Umfangsfläche des gesamten Wandzuges übrig. Diese lässt sich jedoch nur bestimmen, wenn die Verknüpfungen der einzelnen Wände bzw. der Flächen der Wände bekannt sind.

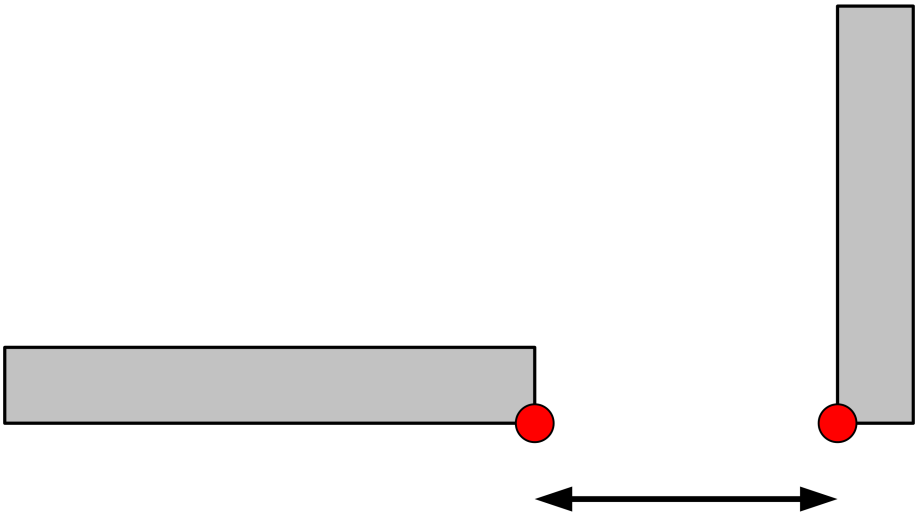


Abbildung 11.4: Referenzpunkte für Bemaßung

In Bild 11.4 ist der Abstand zwischen zwei Wänden vermaßt. Die Maßlinie hat als Referenzpunkt für Anfang und Ende zwei Eckpunkte der Wände. Dies ist nur möglich, wenn diese Punkte auch im Geometriemodell existent sind. Beim CSG-Modell sind diese Punkte nicht vorhanden, sie können somit nicht referenziert werden und nach einer Änderung am Modell sind die Referenzierungen zerstört, die Maßlinien können sich nicht anpassen. Im Brep-Modell sind diese Kanten genau wie in der Realität vorhanden und können referenziert werden. Die Maßlinie kann sich nach einer Änderung des Abstandes zwischen den beiden Wänden an die neue Geometrie anpassen.

11.3.2 Vergleich der Geometriemodelle von Vicado und Bocad

Vicado ist das CAD-Programm der Firma mb AEC Software GmbH für Architektur und Ingenieurbau, das Programm Bocad der Bocad GmbH kann im gleichen Bereich eingesetzt werden, stammt jedoch aus dem Stahlbau.

Der Vergleich wird anhand eines Doppel-T-Trägers durchgeführt, da gerade bei diesem der Unterschied deutlich wird. Vicado nähert die Ausrundungen des Trägers je durch sechs Flächen an 11.5(a), während Bocad die Rundungen unterschlägt. Die Folge ist, dass das Brep-Modell von Bocad zwölf Flächen benötigt um die Mantelfläche des Trägers zu beschreiben während Vicado 36 Flächen, die dreifache Anzahl benötigt.

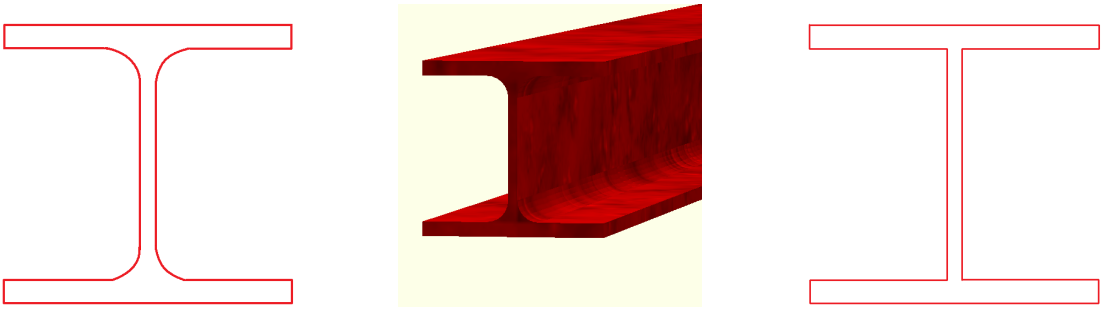
Die einzelnen Flächen sind Rechtecke, so dass Vicado den dreifachen Speicherbedarf gegenüber von Bocad in diesem Fall hat. Der erhöhte Speicherbedarf macht sich erst bei Modellen bemerkbar, die das gesamte System an seine Grenzen bringen.

Der Vorteil der größeren Anzahl von Flächen wirkt sich positiv auf die Visualisierung aus. Wie in Bild 11.5(b) zu sehen ist, reichen die sechs Sekanten aus um die Ausrundungen der Profile mit ausreichender Genauigkeit darzustellen.

Bei der Verschneidung von Objekten macht sich die größere Anzahl der Flächen negativ bemerkbar. Betrachtet werden zwei Träger die miteinander verschnitten werden, die Stirnflächen der Träger werden bei der Betrachtung nicht berücksichtigt:

$$\begin{aligned} 12 \text{ Flächen} \cdot 12 \text{ Flächen} &= 144 \text{ Verschneidungen bei Bocad} \\ 36 \text{ Flächen} \cdot 36 \text{ Flächen} &= 1296 \text{ Verschneidungen bei Vicado} \\ 1296 \text{ Flächen} : 144 \text{ Flächen} &= 9 \end{aligned}$$

Bei diesem einfachen Beispiel muss bei Vicado die neunfache Anzahl von Flächen gegenüber Bocad verschnitten werden. Dies bedeutet auch einen neunfachen Zeitaufwand. Dies kann bei größeren Modellen zu Problemen bei der Bearbeitung führen, da die Berechnung der Verschneidung das Programm ausbremst.



(a) Vicado (b) Visualisierung Vicado (c) BocadSchnitt

Abbildung 11.5: Strahlträger in Vicado und Bocad

12 Referenzen in CAD Programme

Das Wort Referenz (von lat. referre 'auf etwas zurückführen, sich auf etwas beziehen, berichten'; vgl. franz. référence 'Bericht, Auskunft' bzw. engl. reference 'Verweis, Bezug, Hinweis') bezeichnet: in der Informatik einen Verweis auf eine Variable, verwandt mit Zeigern, siehe Referenz (Programmierung). Als Relation (lat. relatio: „das Zurücktragen“) wird im Allgemeinen eine bestimmte Beziehung zwischen Gegenständen oder insbesondere zwischen Objekten bezeichnet.

Eine der Entwicklungen bei den CAD-Programmen der letzten Jahre sind so genannte intelligente Programme. Intelligente Programme reagieren automatisch auf Änderungen am Datenmodell. Grundlage für die Intelligenz in CAD-Programmen ist die Möglichkeit Beziehungen zwischen den einzelnen Objekten zu implementieren.

Die Implementierung der Beziehungen zwischen den Objekten geschieht über Referenzen, die auf andere Objekte oder deren Geometrie verweisen. Beispiele hierfür sind die Beziehung:

- einer Öffnung zu einer Wand
- von Wänden untereinander
- oder von Wänden und Stürzen zu Decken

Weitere Möglichkeiten der Anwendung von Referenzen sind Verweise auf die Geometrie der Objekte:

- Position von Objekten
- Bemaßung

- Rezepte [1]

Die Beziehungen zwischen den Bauteilen lassen sich nach verschiedenen Gesichtspunkten einteilen, zum einen nach der Art der Beziehung:

Natürliche - logische Beziehung Natürliche Beziehungen sind Beziehungen zwischen einzelnen Objekten, die auf einem logischen Zusammenhang beruhen. Fenster und Türen (Wandöffnungen) sind Bestandteile von Wänden und können ohne diese nicht existieren, Wände und Stützen werden durch Decken begrenzt. Durch natürliche Beziehungen können Bauteile untergeordnete Bestandteile von anderen Objekten werden.

Künstliche Beziehung Künstliche Beziehung sind Beziehungen zwischen einzelnen Objekten, die durch den Anwender festgelegt werden. Bei der künstlichen Beziehung hat der Anwender die Möglichkeit, beliebige Beziehungen zwischen beliebigen Objekten zu definieren.

zum anderen nach der Art der Abhängigkeit:

Funktionelle Abhängigkeiten Bei funktionellen Abhängigkeiten benötigt ein Bauteil ein anderes Bauteil, damit es seine Funktion erfüllen kann z. B. Öffnung - Wand.

Geometrische Abhängigkeiten Durch die geometrische Abhängigkeit wird die Geometrie eines Objektes durch ein anderes Objekten beschrieben oder begrenzt z. B. Innenwand - Außenwand.

Durch die Referenzen können Bauteile entweder gleichberechtigt sein oder ein Bauteil einem anderen Bauteil untergeordnet werden.

Infolge der Beziehungen zwischen den Objekten reagieren diese auf Änderungen des Gebäudemodells nach festgelegten Regeln. Werden Objekte manipuliert, d. h. verschoben, gedreht, kopiert etc. wirkt sich dies auch auf die untergeordneten Objekte aus, sie werden entsprechend mitverschoben, -gedreht oder -kopiert. Eine Änderung an der Geometrie eines Objektes bewirkt eine Anpassung der Geometrie der abhängigen Objekte.

[1] Mit Hilfe der Rezepte werden Regeln für die Konstruktion von Bauteilen abgespeichert. Die Verwendung und der Aufbau der Rezepte wird in einem eigenen Kapitel behandelt.

12.1 Beschreibung der Anwendung von Referenzen

Im Folgenden werden verschiedene Möglichkeiten der Verwendung von Referenzen für die Beziehungen zwischen einzelnen Bauteilen beschrieben und erläutert. Die Erläuterung geschieht an Beispielen der in dem Programm Vicado implementierten Referenzen, u. a.:

- Vermaßung
- Wände
- Öffnungen (Fenster)

Abb. 12.1(a) zeigt eine Wand mit Öffnungen und Maßkette. Nach dem Verschieben eines Fensters Abb. 12.1(b) passt sich die Maßkette automatisch an und aktualisiert die Maße. Die Referenzen der Maßlinien beziehen sich auf einzelne Kanten des Geometriemodells der Wände.

(a) Ausgangssituation (b) Endsituation

Abbildung 12.1: Referenzen: Maßkette

Zwei miteinander verschnittene Wände werden in Abb. 12.2(a) gezeigt. Bei einem Verschieben einer der Wände bleibt die Verknüpfung zwischen den Wänden erhalten. Die Verknüpfung zwischen den Wänden bewirkt, dass die Verbindung zwischen den Wänden aufrechterhalten wird. Die zweite Wand wird so lange verlängert, bis die beiden Wände wieder verschnitten werden können Abb. 12.2(b).

In Abb. 12.3(a) wird ein Fenster in einer Wand gezeigt. Nach einer Änderung der Abmessungen des Fensters wird ein erneutes Verschneiden des Fensters mit der Wand ausgelöst Abb. 12.3(b), die Öffnung in der Wand wird auf die neue Fenstergröße angepasst. Durch die Referenz wird eine Verknüpfung zwischen dem Fenster und der Wand erstellt, die das Update auslöst.

(a) Ausgangssituation (b) Endsituation

Abbildung 12.2: Referenzen: Wand verschieben

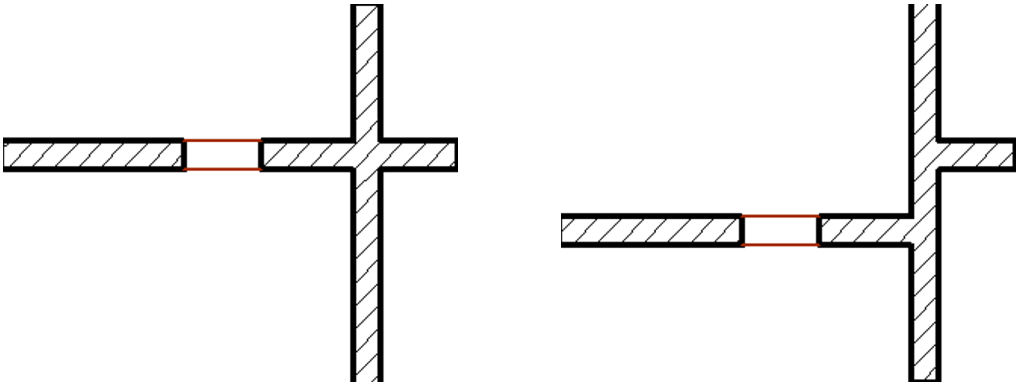
(a) Ausgangssituation (b) Endsituation

Abbildung 12.3: Referenzen: Öffnung ändern

Wird die Wand verschoben Abb. 12.4(a), so bewirkt die Verknüpfung zwischen der Wand und dem dazugehörigen Fenster, dass dieses mitverschoben wird Abb. 12.4(b).

Bei einem Löschen des Fensters aus der Wand Abb. 12.5(a) sorgt die Verknüpfung dafür, dass das Geometriemodell der Wand angepasst, die Öffnung für das Fenster entfernt und die Wand wieder geschlossen wird Abb. 12.5(b).

Ein Ändern der Breite der Wand (Abb. 12.6(a)) sorgt für eine Anpassung des der Wand untergeordneten Fensters. Die Fensteröffnung in der Wand wird neu erzeugt, das Fenster wird entsprechend den in den Eigenschaften gespeicherten Angaben neu in der Wand positioniert (Abb. 12.6(b)).



(a) Ausgangssituation (b) Endsituation

Abbildung 12.4: Referenzen: Wand mit Fenster verschieben

(a) Ausgangssituation (b) Endsituation

Abbildung 12.5: Referenzen: Wand löschen

(a) Ausgangssituation (b) Endsituation

Abbildung 12.6: Referenzen: Wandbreite ändern

13 Rezepte - assoziative Geometrie

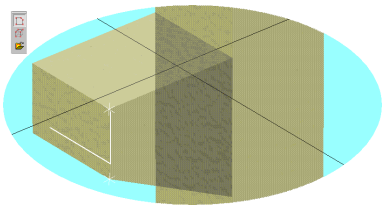
Rezepte sind eine konsequente Anwendung und Weiterentwicklung der Referenzen (siehe Kapitel 12) zwischen einzelnen Bauobjekten. Sie werden jedoch im Zusammenhang mit der Erzeugung der Bauobjekte angeführt, da sie zur Objekterzeugung dienen. Rezepte werden unter anderem in dem auf or.bit-basierendem CAD-Programm Vicado der mb AEC Software GmbH verwendet. Sie entstanden aus dem in STRAKON zugrunde liegenden Konzeptes assoziativer Geometrie und der Umsetzung der Bewehrung2000 aus STRAKON in or.bit.

Der Grundgedanke der assoziativen Geometrie ist, dass Körper nicht wie es üblich ist über ihre Geometrie, sondern über ihre Anbindung an andere Objekte beschrieben werden. Unter Geometrie versteht man die Lage der Komponenten des Körpers (Punkte, Kanten etc.) im dreidimensionalen Raum, während die Anbindung die relative Lage der Komponenten in Bezug auf andere Geometrien ist.

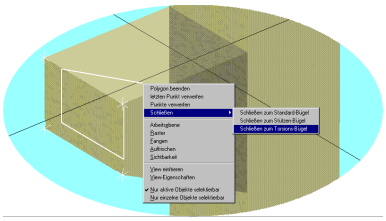
13.1 Erläuterung des Prinzips der Rezepte

Das Prinzip der Rezepte wird anhand des Beispiels Bewehrung erläutert. Die Bewehrung hat eine Sonderstellung unter den Bauobjekten, da sie als einziges Objekt grundsätzlich von der Geometrie eines anderen Objektes abhängig ist. Sie wird in der Regel durch die Abstände zur begrenzenden Oberfläche und den Abständen der einzelnen Bewehrungseisen untereinander definiert.

Die Bilder 13.1(a)-13.1(b) zeigen die Eingabe der Bewehrung für eine kurze Konsole. Die Eingabe erfolgt in zwei Schritten. Im ersten Schritt (Abb. 13.1(a)-13.1(b)) wird die Geometrie des Bügels als geschlossenes Polygon eingegeben. Im zweiten Schritt (Abb. 13.2(a)-13.2(b)) wird der Verteilungsbereich der Bügel als Strecke Anfang-Endpunkt eingegeben. Die Geometrie der folgenden Bügel ergibt sich aus dem Abstand der Schwerelinie zur Begrenzungsfläche die im ersten Schritt beim Prototyp festgelegt wurde.

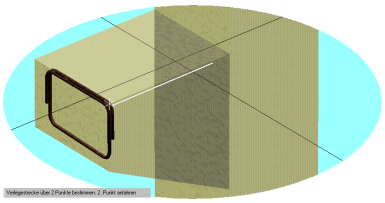


(a) Schritt 1

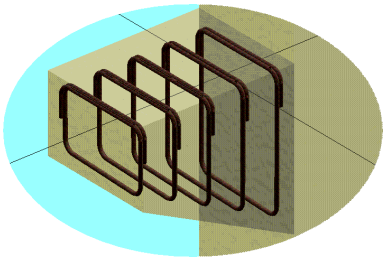


(b) Schritt 2

Abbildung 13.1: Beispiel Rezept



(a) Schritt 3



(b) Schritt 4

Abbildung 13.2: Beispiel Rezept

Als weitere Angabe für die Konstruktion wird die Anzahl der Bügel benötigt. Dies kann über die absolute Anzahl der Bügel oder über die Verteilung, den Abstand zwischen den Bügel geschehen. Die Angabe der absoluten Anzahl hat den Nachteil, dass sich bei Änderung der Geometrie auch der Bewehrungsgrad cm^2/m ändert, während bei der Definition über den Abstand der Bewehrungsgrad konstant bleibt.

13.2 „Aufbau“ der Rezepte

Bei den Rezepten wird als Daten nicht die Geometrie der Körper, sondern vielmehr ihre Anbindung an die anderen Körper gespeichert, aus denen die angebundenen Körper ihre Geometrie selbst berechnen können.

Infolge der Assoziativität reagieren die angebundenen Körper auf Geometrieänderung der übergeordneten Körper. Wird die Lage oder die Form des Anbindungskörpers durch den Benutzer geändert, passt der angebundene Körper seine Geometrie automatisch entsprechend den abgespeicherten Regeln an und berechnet die Geometrie neu. Die Speicherung der Erstellungsparameter ermöglicht auch die Korrigierbarkeit der erstellten Objekte. Der angebundene Körper passt sich nach den abgespeicherten Rezepten an, wenn einer der in die Konstruktion eingehenden Parametern, wie die Randabstände, geändert wird.

Ein einmal eingegebenes Rezept kann auch als Prototyp dienen. Dies ermöglicht es auch durch Kopieren das Rezept auf ein anderes Objekt zu übertragen, ggf. müssen einzelne Angaben angepasst werden. Ein Rezept kann somit auf andere, auch stark abweichende Geometrie ohne Problem angewendet werden.

Bei der Eingabe der Rezepte wird es dem Anwender ermöglicht, die Konstruktion über das Auswählen vorhandener Geometrien zu steuern, anstatt selbst die Geometrie des Körpers zu konstruieren. Aus den ausgewählten Geometrien werden die Beziehungen ermittelt, die als Rezept dienen, mit dessen Hilfe der Körper seine eigene Geometrie selbstständig konstruieren kann.

13.3 Bezugsobjekte für die Rezepte

Für die Speicherung der Rezepte werden Bezugsobjekte benötigt. Als Bezugsobjekte können theoretisch alle vorhandenen Komponenten der Geometrie, auf die bezogen wird, verwendet werden:

- Punkte
- Kanten
- Flächen

Vergleicht man die Möglichkeiten der Anbindung näher, so ergeben sich Vor- und Nachteile der einzelnen Komponenten, die zum Ausschluss einzelner Komponenten führen. Die beiden untersten Komponenten in der Hierarchie sind die Punkte und Kanten. Bei der Manipulation der Geometrie eines Objektes werden meist Punkte und Kanten in das Modell eingefügt bzw. gelöscht, sie sind somit die Objekte, die am wenigsten konstant sind und somit für die Anbindung am wenigsten geeignet sind. Die nächsten Objekte sind die Flächen des Körpers. Diese sind besser geeignet, da sie bei der Änderung der Körpergeometrie meist erhalten bleiben.

13.4 Grundzüge der Implementierung des Bezuges

Die Anbindung an einen Körper wird in einem Rezept für die Konstruktion der Geometrie verwendet. Die Geometrie besteht im Wesentlichen aus 3D-Polygonen, somit ist die wichtigste Aufgabe des Rezeptes die Konstruktion dieser Polygone aus der Geometrie der Anbindungskörper. Die Rezepte bestehen aus zwei Komponenten:

- Punktrezepte

  Punktrezepte sind Regeln zur Konstruktion von Punkten aus den Flächen des Anbindungskörpers. Diese Punkte dienen als Eckpunkte für Polygone.

- Polygonrezepte

  Polygonrezepte sind Regeln, die aus den Punkten der Punktrezepte Polygone erzeugen.

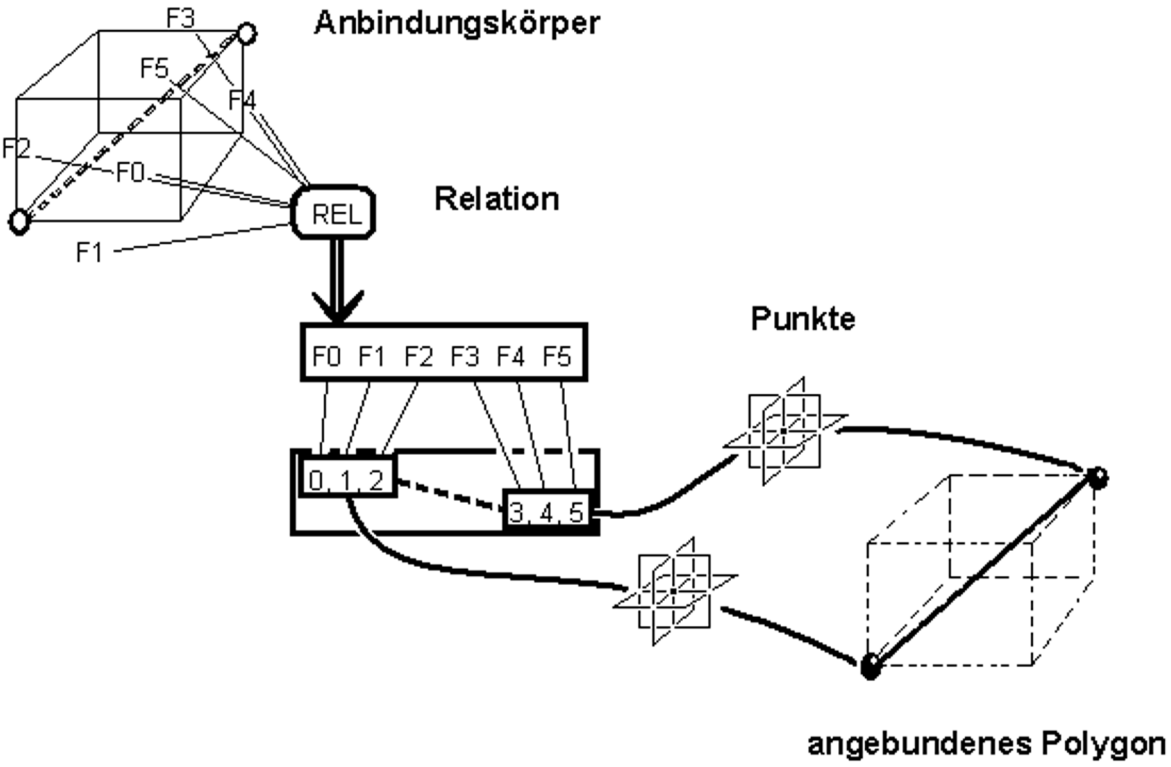


Abbildung 13.3: Darstellung der Anbindung durch ein Punktrezept

13.4.1 Beschreibung der Punktrezepte

Die Punktrezepte sind Konstruktionsvorschriften für Punkte aus verschiedenen Flächen der Anbindungskörper. Eine der einfachsten Konstruktionsvorschriften ist die Berechnung eines Punktes als Schnittpunkt dreier Ebenen. Hierfür können alle Flächen des Anbindungskörpers oder dazu parallele Ebenen verwendet werden. Das einfachste Punktrezept ist somit die Angabe dreier Bezugsflächen in einem Flächenvektor mit dem Abstand der

Ebenen zu diesen Flächen. Komplexere Punktobjekte können durch Manipulationen, Verschiebung und Drehung des Ergebnispunktes des einfachen Rezeptes unter Einbeziehung von z. B. zusätzlichen Ebenen erzeugt werden.

13.4.2 Beschreibung der Polygonrezepte

Die einfachste Variante eines Polygonrezeptes ist die Erzeugung eines Polygons aus den Ergebnispunkten mehrerer Punktrezepte. Durch die Angabe von gekrümmten Kanten oder Automatismen zur Erzeugung weiterer Punkte können komplexere Polygonrezepte erstellt werden.

Die Bezugsflächen werden nur benötigt, wenn die Geometrie nach einer Änderung neu berechnet werden muss. Hierfür muss die Geometrie nicht im angebundenen Körper enthalten sein. Eine Möglichkeit ist die Kapselung des Bezuges in einer Relation, die die Bezugsflächen auf Aufforderung in geordneter Reihenfolge zurückgibt. Als Abgabeformat ist ein Vektor von Flächen zu wählen. Die Relation, die den Bezug zu den Flächen eines oder mehrerer Körper kapselt, ist eine eigene Klasse mit einem Vektor für die Flächen.

14 Bemaßung in CAD-Programmen

14.1 Assoziative Bemaßung

Die Bemaßung dient zur graphischen Darstellung der Abmessungen des Modells. Die einzelnen Abmessungen wie Radien können entweder direkt aus dem Modell übernommen oder wie der Abstand zwischen zwei Punkten berechnet werden. Die Abmessungen sind unabhängig von dem Maßstab der Darstellung.

Als Weiterentwicklung der Bemaßung entstand die assoziative Bemaßung. Die assoziative Bemaßung ist nicht nur ein graphisches Objekt, sie steht als intelligentes Objekt in Beziehung zum Datenmodell.

Bei der Assoziativen Bemaßung steht die Bemaßung in Beziehung zum Datenmodell. Die Beziehung ermöglicht es der Bemaßung auf Änderungen im Datenmodell zu reagieren. Hierdurch wird sichergestellt dass die Bemaßungen in den einzelnen Views immer auf dem aktuellen Stand sind. Ein Anpassen der Bemaßungen durch den Anwender kann hierdurch entfallen. Durch eine freie Konfiguration der Elementanbindung wird es ermöglicht, festzulegen, welche Elementabmessungen vermaßbar sind.

14.1.1 Arten der Bemaßung

In CAD-Anwendungen werden typischerweise zwei Gruppen von Maßtypen verwendet. Dies sind:

1. Einzelmaße mit
    - Winkelbemaßung
    - Höhenknotenbemaßung
    - Radialbemaßung
    - Bogenlängenbemaßung

- Niveaubemaßung
- Raumbemaßung

2. und Intervallmaßketten

   - Intervallmaßketten bestehen aus mehreren, in Intervallen entlang einer Linie oder eines Bogens angeordneten Maßen. Die Maßketten können verschiedene Maße enthalten, z. B. Elementabmessungen, Abstände zwischen zwei Punkten.

14.2 Erzeugung der Bemaßung

Für die Erzeugung der Bemaßung gibt es zwei Möglichkeiten:

1. interaktiv durch den Anwender
2. automatisch durch andere Anwendungen über eine entsprechende Schnittstelle

Die Möglichkeiten der ersten Art werden näher erläutert. Bei der interaktiven Erstellung der Bemaßung werden vom Anwender Punkte ausgewählt, mit deren Hilfe die Bemaßung bestimmt wird. Die Auswahl der Punkte für die Bemaßung kann auf zwei verschiedene Arten erfolgen. Die erste Art ist die direkte Auswahl einzelner Punkte mit der Maus, die zweite Art ist die Berechnung der Maßpunkte über eine Schnittlinie durch das Modell. Bei der Berechnung der Maßpunkte konstruiert der Anwender über zwei Punkte eine Gerade. Anschließend werden die Schnittpunkte der Geraden mit den Kanten berechnet. Diese Punkte dienen als Ausgangspunkte für die Maßlinien. Die individuellen Punkte werden frei vom Anwender gewählt, sie sind unabhängig vom Gebäudemodell und bleiben auch bei Änderungen am Modell unverändert. Bei den Festpunkten handelt es sich um Punkte des Geometriemodells die referenziert werden. Da bei den Festpunkten auf das Geometriemodell referenziert wird, passen sich Maßketten bei Änderung des Modells an. Sie zeigen immer die korrekten Maße an, ohne dass sich der Anwender um die Anpassung der Maßketten kümmern muss. Eine weitere wichtige Eingabe ist die Orientierung der Maßkette im Raum. Hier gibt es drei Möglichkeiten:

1. Horizontal
2. Vertikal
3. Schräg

Die meisten Maßlinien sind parallel zu den Mauern, durch die Rechtwinkligkeit der Gebäude horizontal und vertikal auf dem Plan. Bei der Eingabe der Maßlinie gibt es verschiedene Möglichkeiten für die Richtung:

Eingabe der Richtung Richtung wird vorgegeben, horizontal und vertikal auf dem Plan

Erkennen der Richtung Die Richtung wird aus den eingegebenen Punkten konstruiert, diese Methode ist nur eindeutig ohne weitere Regelung, wenn die Punkte auf einer Geraden liegen.

Konstruktionsrichtung oder senkrecht zur Konstruktionsrichtung Als Richtung der Maßlinie wird die Konstruktionsrichtung oder die Senkrechte zur Konstruktionsrichtung verwendet, sie wird im vorhinein festgelegt.

14.2.1 Darstellung der Bemaßung

In den einzelnen Planungsabschnitten (Genehmigungsplanung, Ausführungsplanung) unterscheiden sich auch die Bemaßungen. In einem Fall werden die Rohbaumaße im anderen die Fertigmaße benötigt. Eine intelligente Steuerung der Bemaßung passt sich automatisch den Erfordernissen an. In Abhängigkeit der Einstellungen in der View werden die erforderlichen Maße dargestellt. Die Darstellung der Bemaßung erfolgt viewspezifisch, da nur dadurch gewährleistet wird, dass in den einzelnen Sichten unterschiedliche Bemaßungen dargestellt werden können.

14.2.1.1 Darstellung von Maßketten

Für die Darstellung von Maßketten gibt es verschiedene Möglichkeiten, einige von diesen sind auch länderspezifisch. Während es in Deutschland typisch ist, die einzelnen Maße über die Maßketten zu schreiben, werden z. B. in Schweden die Einzelmaße aufaddiert. Die Maßlinie wird nicht durchgezogen gezeichnet sondern nur unterhalb oder oberhalb des Maßes (Abb. 14.1).

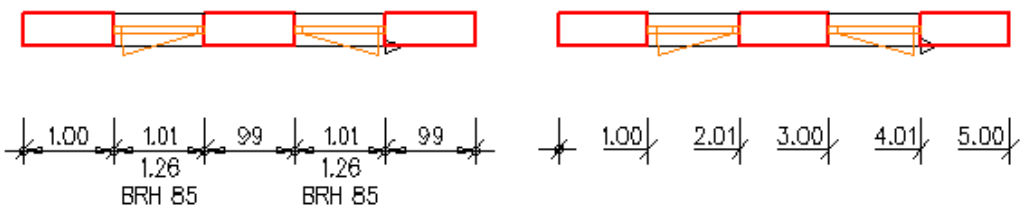


Abbildung 14.1: Darstellung verschiedener Bamaßungsarten

14.2.1.2 Steuerung der Bemaßung

Die Steuerung der Darstellung der Bemaßung erfolgt genau wie bei den Bauobjekten über die Painter (siehe Kapitel 19). Mit dem Painter lassen sich Darstellungsprototypen für die Bemaßung festlegen, welche Voreinstellungen für die Bemaßungen enthalten. Die Voreinstellung umfassen:

1. allgemeine graphischen Attribute wie
    - Linienart
    - Linientyp
    - Farbe
2. und spezielle Attribute wie
    - Einheit
    - Art der Begrenzung

Durch die Implementierung mehrerer Prototypen kann der Anwender zwischen den verschieden Möglichkeiten der Darstellung wählen. Durch die Veränderung eines Prototyps kann ein neuer Prototyp erzeugt und mit einem neuen Namen gespeichert werden. Entsprechend der Funktionalität der Painter kann ein Bemaßungsprototyp für die Darstellung eines einzelnen Maßes, aller Maße in einer View oder für die gesamte Bemaßung verwendet werden.

14.2.2 Manipulation der Bemaßung

Ebenso wie die Bauobjekte lässt sich die Bemaßung manipulieren. Zu der Funktionalität gehören:

1. Bemaßungspunkt
    - einfügen
    - löschen
    - verschieben
2. Beschriftung
    - verschieben
    - bearbeiten

3. Maßzahl
    - verschieben
    - bearbeiten
4. Maßkette
    - verschieben
    - bearbeiten
5. Automatisch platzierte Bemaßung
    - tauschen
    - sortieren

15 Datenaustausch im Bauwesen

15.1 Datenaustausch

Der Austausch von Daten hat im Bauwesen eine wichtige Bedeutung. Während des Planungsablaufes sind meist verschiedene Büros beteiligt, die verschiedene Teilaufgaben bearbeiten. Der Planungsablauf wird anhand der HOAI §15 (2) „Leistungsbild Objektplanung für Gebäude, Freianlagen und raumbildende Ausbauten“ näher betrachtet, die Bereiche Ingenieurbau, Städteplanung, Landschaftsbau bleiben hierbei unberücksichtigt. Die HOAI führt für die Objektplanung für Gebäude, Freianlagen und raumbildende Ausbauten die folgenden Phasen während des Planungsablaufes auf:

1. Grundlagenermittlung
2. Vorplanung
3. Entwurfsplanung
4. Genehmigungsplanung
5. Ausführungsplanung
6. Vorbereitung der Vergabe
7. Mitwirkung bei der Vergabe
8. Objektüberwachung

In der ersten Leistungsphase der Grundlagenermittlung werden die ersten Überlegungen angestellt, welche weiteren fachlich Beteiligte in den Planungsprozess einbezogen werden müssen. Hierzu zählen u. a.

- Tragwerksplaner
- Bauphysiker
- Technische Gebäudeausrüstung
- Anlagenbauer (bei Industriebauten)

Schon in der zweiten Leistungsphase, der Vorplanung, werden die ersten Leistungen der verschiedenen Planer integriert, es findet der erste Datenaustausch der Beteiligten statt.

Während der Entwurfsplanung wird ein Planungskonzept durchgearbeitet. Es werden die Beiträge anderer an der Planung fachlich Beteiligter bis zum vollständigen Entwurf eingearbeitet. In der folgenden Genehmigungsplanung wird der bisherigen Planungsstand dargestellt.

In der fünften Leistungsphase folgt die Ausführungsplanung, wobei die bisherigen Ergebnisse verfeinert werden. Hierbei findet ein intensiver Austausch zwischen den an der Planung Beteiligten statt, bis eine ausführungsreife Lösung entsteht. So muss sich z. B. der Tragwerksplaner mit dem Gebäudeausrüster austauschen, damit die Öffnungen für die Versorgungsleitungen frühzeitig in der Planung berücksichtigt werden können.

Bei der Vorbereitung der Vergabe, der sechsten Leistungsphase, werden die Leistungsverzeichnisse erstellt, hierzu werden die Planungen der einzelnen Beteiligten (Architekt, Tragwerksplaner, Gebäudeausrüster etc.) benötigt.

Bei der siebten Phase werden die Verdingungsunterlagen für die Leistungsbeschreibung zusammengestellt. Die Planungsunterlagen werden oftmals Bestandteil der Verdingungsunterlagen.

Die achte Phase, die Objektüberwachung, benötigt die Planungen zur Ausführung. Mögliche Abweichungen zur Planung müssen wieder in Plänen dokumentiert werden.

In der abschließenden neunten Phase, der Objektbetreuung und Dokumentation, werden die zeichnerischen Darstellungen und rechnerischen Ergebnisse des Objekts zusammengestellt.

15.2 Formate zum Datenaustausch

Während der Planung und der Ausführung eines Bauwerks sind verschiedene Büros beteiligt. Der aktuelle Planungsstand muss während der Planung und Ausführung zwischen den verschiedenen Büros ausgetauscht werden. Die meisten Programme kommen nur mit dem eigenen Format und Formaten der Programme aus derselben Firma zurecht. Mit der Zeit haben sich verschieden Formate für den Datenaustausch herauskristallisiert.

15.2.1 DXF und DWG

Als Quasistandard im CAD-Bereich hat sich das DXF- Drawing Exchange Format etabliert. Das Format wurde von der Firma Autodesk 1982 für AutoCAD eingeführt. Mit jeder neuen Version von AutoCAD kommt eine neue veränderte Variante des DXF-Formates auf den Markt. Das DXF-Format ist ein Vektorformat, das Geometrieelemente wie Punkt, Linie, Kreisbogen, Text, Blöcke, Bemaßungen und Solids unterstützt. Die Dokumentation des DXF-Formates ist von Autodesk offengelegt.

Das DWG-Format ist ein weiteres Format von der Firma Autodesk. Im Gegensatz zu DXF ist die Dokumentation des Formates nicht offengelegt. Das DWG Format wird vom ADT Autodesk Architectural Desktop verwendet. Diese Dateien können nur vom ADT ohne Informationsverlust geöffnet werden, bei allen anderen Programmen gehen alle Informationen, die über die Geometrie hinausgehen, verloren.

Das DXF-Format hat sich als Standard etabliert, eignet sich jedoch nicht für den Austausch bei modernen CAD-Programmen im Bauwesen, da keine weiteren Informationen als die Geometrie oder Zeichnung weitergegeben werden. DXF-Zeichnungen können von vielen Programmen, auch FEM-Programmen eingelesen werden, und dienen dann als Vorlage für die weitere Arbeit. Die Geometrie muss nachgezeichnet werden und die zusätzlichen Angaben wie Material müssen erneut eingegeben werden. Bei der Weitergabe von Zeichnungen im DXF-Format treten immer wieder Fehler auf, da kein Export-Import ohne Probleme verläuft.

Das DWG-Format ist als allgemeines Austauschformat auch nicht geeignet, da es kein offenes Format ist. Alle weiteren Informationen, die bei modernen Programmen im Modell vorhanden sind, können über dieses Format nicht weitergegeben werden.

15.2.2 IFC-Format

Das IFC-Format wurde von der IAI International Alliance for Interoperability ins Leben gerufen. Die Mitglieder sind Firmen, die mit dem Bauwesen in Verbindung stehen wie Planungsbüros, Softwarehäuser und Hochschulen. Das Ziel der IAI ist:

„Die Produktivitätssteigerung im Planungsprozess und die Vermeidung von Fehlern durch Informationsinkonsistenz“

Die IFC beschreibt alle Bauteile eines Bauwerkes als Objekte, die von allen Programmen,

die die IFC unterstützen auch wieder als Objekte interpretiert werden. Die IFC stellt ein intelligentes Datenmodell für den Anwender zur Verfügung.

Die IFC wird als Produktmodell angesehen und dient als Schema zur Festlegung des Datenaufbaus des zu beschreibenden Produktes und der Beschreibung der Beziehung untereinander. Die IFC versucht in jeder Bauphase die Daten der vorherigen Phasen mit einzubinden.

Die Konzepte der IFC beschreiben:

- die grundlegenden Methoden des Objektes (ID, Besitzer)
- die grundlegenden Informationen über Relationen zwischen den Objekten
- die Typenbeschreibung der auftretenden Gegenstände
- die Eigenschaften der zu beschreibenden Objekte
- die geometrische Repräsentation und Platzierung der Objekte

Das IFC-Format ist das Format, das am besten geeignet ist, die Daten zwischen modernen CAD-Programmen auszutauschen, da neben der Geometrie auch noch die Objekteigenschaften und die Beziehungen zwischen den Objekten weitergegeben werden können. Im Gegensatz zum DXF-Format ist die Verbreitung noch geringer. Nicht jedes Programm, das Dateien im IFC-Format liest kann diese auch schreiben.

Die Dateien eines Modells im IFC-Format können bedingt durch das Gebäude und den Planungsstand sehr umfangreich sein, was zu langen Importzeiten führen kann. Je nach Anwendung müssen auch Daten importiert werden, die für die konkrete Anwendung nicht benötigt werden.

15.2.3 PDF - Portable Document Format

Das PDF-Format wurde von der Firma Adobe Systems entwickelt und erstmals 1993 veröffentlicht. Beim Portable Document Format handelt es sich um ein plattformübergreifendes Dateiformat für Dokumente. PDF ist eine vektorbasierte Seitenbeschreibungssprache. Für praktisch jedes Betriebssystem sind heute kostenlose Programme für das Lesen von PDF-Dateien vorhanden.

Die Version 1.4 des PDF-Formates[1] wurde im September 2005 von der International Organiziation of Standardization (ISO) als Standard für die Langzeitarchivierung von Dokumenten anerkannt: ISO 19005-1.

[1] http://www.heise.de/newsticker/meldung/63957

Der Einsatzbereich des PDF-Formats liegt im elektronischen Austausch von Plänen. Eine weitere Bearbeitung dieser Pläne ist im Allgemeinen nicht vorgesehen. Aus den CAD-Zeichnungen können kleine PDF-Dateien erzeugt werden, die sich leicht über E-Mail austauschen lassen können. Damit können Handwerker versorgt oder Bauherren über die aktuelle Planung informiert werden. Im Gegensatz zu den CAD-Formaten sind die Dateien für jedermann lesbar, da die entsprechenden Programme kostenlos verfügbar sind. Die Anerkennung als ISO-Standard stellt auch die Lesbarkeit der Dateien in einigen Jahren sicher.

Teil III

Programmarchitektur

16 Allgemeiner Programmaufbau

In den letzten Jahren der Programmentwicklung hat sich auch bei CAD-Programmen ein mehrschichtiger Aufbau der Anwendungen herausgebildet. Als Beispiel hierfür wird zunächst AutoCAD, das weltweit meist verbreitete CAD-Programm betrachtet. AutoCAD bietet verschiedene Anwendungen an, die alle auf demselben Kern aufbauen.

- AutoCAD Mechanical - Maschinenbau
- AutoCAD Architecture - Bau
- AutoCAD MEP - Gebäudeausrüstung
- AutoCAD Map 3D - Karten
- AutoCAD Civil 3D - Tiefbau

Außer den Produkten die Autodesk selbst anbietet, gibt es verschiedene Firmen, die auf AutoCAD aufbauend eigene Produkte wie ProSteel3D vermarkten.

Als weiteres Beispiel wird mit Vicado ein bauspezifisches CAD-Programm betrachtet. Obwohl Vicado kein allgemeines, sondern ein auf das Bauwesen spezialisiertes CAD-Programm ist, werden auch hier verschiede Varianten angeboten. Die einzelnen Varianten des Programms sind für die verschiedenen Teilbereiche des Bauwesens ausgelegt:

- ViCADo.arc - Entwurfs- und Ausführungspläne
- ViCADo.ing - Positions-, Schal- und Bewehrungspläne
- ViCADo.pos - Positionspläne
- ViCADo.plan - Nachbearbeitung von mit Arcon erzeugten Plänen

16.1 Schichtaufbau von Vicado

Vicado besitzt wie die meisten CAD-Programme einen mehrschichtigen Aufbau, der eine flexible Erweiterung des Programms ermöglicht. Die Graphik 16.1 zeigt den vereinfachten

Schichtaufbau von Vicado. Die Basis des Programms bildet der CAD-Kern, dieser beinhaltet die allgemeine Funktionalität des Programms (siehe Kapitel 16.2). Die darauf folgende Schicht beinhaltet die Bauobjekte, welche den Anwendungsbereich des Programms bestimmen. Die oberste Schicht sind die verschiedenen Anwendungen des Programms.

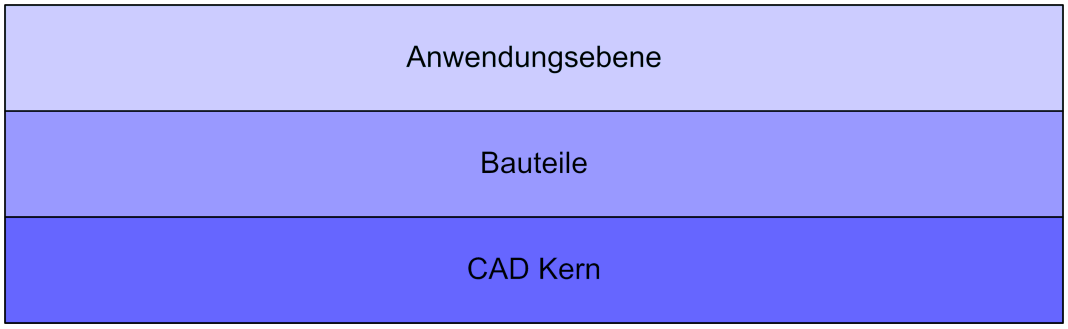


Abbildung 16.1: Schichtaufbau eines CAD-Programms

Für die verschiedenen Anwendungen werden unterschiedliche Bauobjekte benötigt, welche zusammen mit der Anwendung zum Kern ergänzt werden.

16.2 Der Programmkern

Der CAD-Kern ist die Basis eines CAD-Programms, auf ihm bauen die verschiedensten Anwendungen des Programms auf. Der Kern beinhaltet die allgemeine Funktionalität eines CAD-Systems, die unabhängig von den darauf aufbauenden Anwendungen ist.

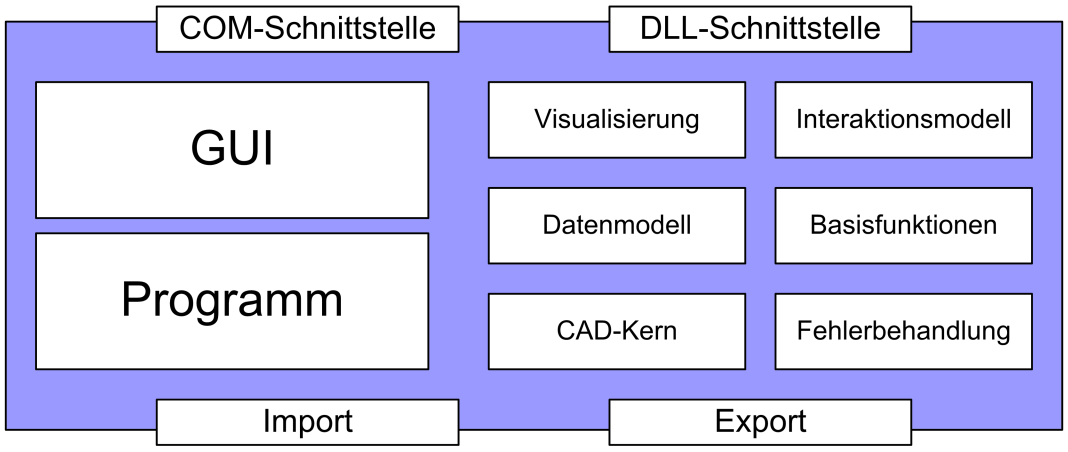


Abbildung 16.2: Der CAD-Kern

Abbildung 16.2 zeigt die Grundstruktur des CAD-Kerns. Der Kern besitzt zwei Schnittstellen nach außen. Die eine Schnittstelle ermöglicht die Erweiterung des Programms, über

DLLs können ergänzende Programmteile hinzugeladen werden. Mit der zweiten Schnittstelle wird die Kommunikation mit anderen Programmen ermöglicht. Über diese Schnittstelle können Daten für anderen Programme exportiert bzw. importiert werden.

Der CAD-Kern stellt die Teile des Programms, die unabhängig von der Art der Anwendung sind, zur Verfügung. Er beinhaltet das eigentliche Programmgerüst mit der Graphischen Oberfläche und der Schnittstelle zur Programmerweiterung. Ein zentraler Bestandteil des CAD-Kerns sind die Grundlagen des Datenmodells mit der Objektpersistenz und der Visualisierung des Datenmodells. Hinzu kommen die Projektverwaltung und die Verwaltung der einzelnen Positionen des Datenmodells. Des Weiteren kommen die Basisfunktion des CAD-Programms zum Bearbeiten des Datenmodells, das Interaktionsmodell und die Fehlerbehandlung zum Kern hinzu.

16.3 Bauteilebene

Bauteile				
Baukörper	Fertigteile	Bewehrung	Einbauteile	FEM
Wand Decke Fundament Stütze etc.	Hohlwand Vollwand Volldecke Elementdecke etc.	Stabstahl Matten Gitterträger Bügel etc.	Tür Fenster Transportanker Hülsen etc.	Linienlager Punktlager FE-Netz Flächenlast etc.

Abbildung 16.3: Bauteilebene von CAD-Programmen

Die Bauteilebene stellt die verschiedenen Datentypen des CAD-Programms zur Verfügung, sie beinhaltet die Objekte welche das Programm erzeugen kann. Diese Ebene ist anwendungsspezifisch. Zum einen sind die Bauteile abhängig von der Ausrichtung des Programms: Maschinenbau, Bauwesen, Elektrotechnik etc., zum anderen lassen sich in diesen Bereichen die verwendeten Bauteile auf verschiedene Anwendungen aufteilen. Betrachtet wird im Folgenden wieder eine CAD-Anwendung für das Bauwesen.

Die verschiedenen Bauteile lassen sich in Gruppen unterteilen. Die Bauteilgruppen sind zum Teil allgemeine Gruppen, teilweise anwendungsspezifische Gruppen. Die Allgemeine Gruppen sind die Baukörper wie Wände und Decken und Einbauteile wie Türen und Fenster. Hinzu kommen Gruppen für andere Anwendungen wie den Ingenieurbau; dies sind u. a. die Bewehrungsbauteile oder die FEM-Objekte für die Kopplung des CAD-Programms

zu einem FEM-Programm. Daneben existieren weitere Gruppen für spezielle Anwendungen wie die Fertigteile. Die Gruppen entsprechen der Anordnung der Bauteilbuilder in der Wasleiste (siehe Kapitel 7.2). Die Bauteile gehören teilweise zur Grundausstattung eines CAD-Programmes im Bauwesen oder werden mit den Anwendungen zur Bauteilebene hinzugeladen. Dies können einzelne ergänzende Bauteile oder auch ganze Bauteilgruppen sein.

16.4 Anwendungsebene

Die oberste der drei Ebenen ist die Anwendungsebene. Auf dem Kern und der Bauteilebene können die verschiedensten Anwendungen aufbauen. Die Anwendungen verfügen über unterschiedliche Funktionalität mit denen das Datenmodell bearbeitet und ausgewertet wird.

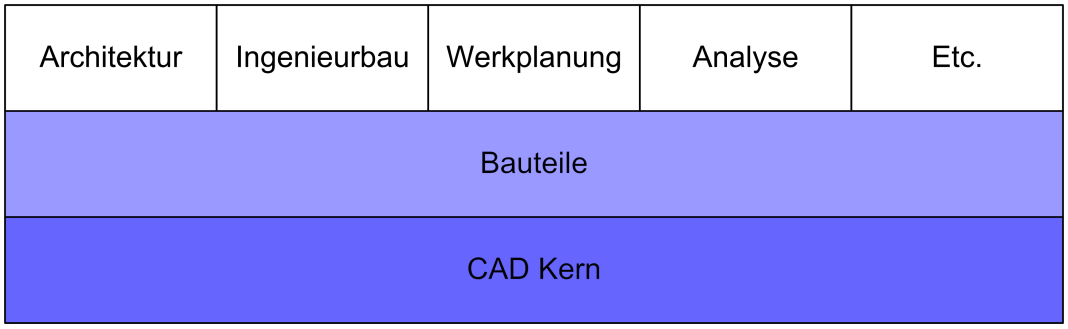


Abbildung 16.4: Anwendungsebene von CAD-Programmen

Die Anwendungen, die auf den beiden Schichten aufbauen, sind aus den ganzen Bereichen des Bauwesens. Sie können über die klassische CAD-Anwendung Architektur und Ingenieurbau hinausgehen und z. B. AVA-Anwendungen umfassen. Auf den CAD-Daten können auch Finite-Elemente oder Baustatik Anwendungen aufbauen, so dass eine doppelte Systemeingabe entfällt. Die Ergebnisse dieser Anwendung fließen in das Datenmodell des CAD-Programmes zurück, wodurch eine Konsistenz der Daten sichergestellt wird.

17 MVC-Koncept in CAD Programmen

In diesem Kapitel wird auf die Anwendung des Model-View-Controller Konzeptes bei CAD-Programmen eingegangen, speziell auf Modell und View und weniger auf den Controller, der eine steuernde Funktion besitzt. Der Aufbau und die Funktionsweise des Model-View-Controller Konzeptes wird in Kapitel 4 behandelt.

Die Idee des MVC-Konzepts stammt zwar schon aus den siebziger Jahren, konnte sich jedoch erst in den neunziger Jahren durchsetzen. Das Konzept beschreibt einen mehrschichtigen Programmaufbau und trennt dabei die Datenhaltung von den Ansichten und kapselt sie in eigenen Klassen.

Seit der Einführung von 3D-Geometriemodellen für CAD-Anwendungen steht nicht mehr der Plan, sondern das Gebäudemodell im Mittelpunkt. Der Anwender erzeugt ein virtuelles dreidimensionales Modell des zu planenden Gebäudes, welches in verschiedenen Views (Sichten), den Plänen, dargestellt wird. Diese Vorgehensweise entspricht genau, nicht nur nach den Bezeichnungen, dem Model-View-Controller Prinzip. Das MVC-Prinzip bietet Möglichkeiten zur flexiblen Plangestaltung. Bei einer Änderung des Modells werden die verschiedenen Ansichten angepasst. Es können die verschiedensten Sichten (Kapitel 17.2) durch das allgemeine Datenmodell bedient werden.

17.1 Das Modell

Das Modell ist die Datenhaltungsschicht des MVC-Konzepts. Sie beinhaltet die Liste zur Speicherung und Verwaltung der Bauteile ein Objekt der Klasse *CObjektliste*. Hinzu kommen die Methoden zum Bearbeiten des Datenmodells und Laden bzw. Speichern des Modells. Die Abbildung 17.1 zeigt das Klassendiagramm einer vereinfachten Variante des Modells.

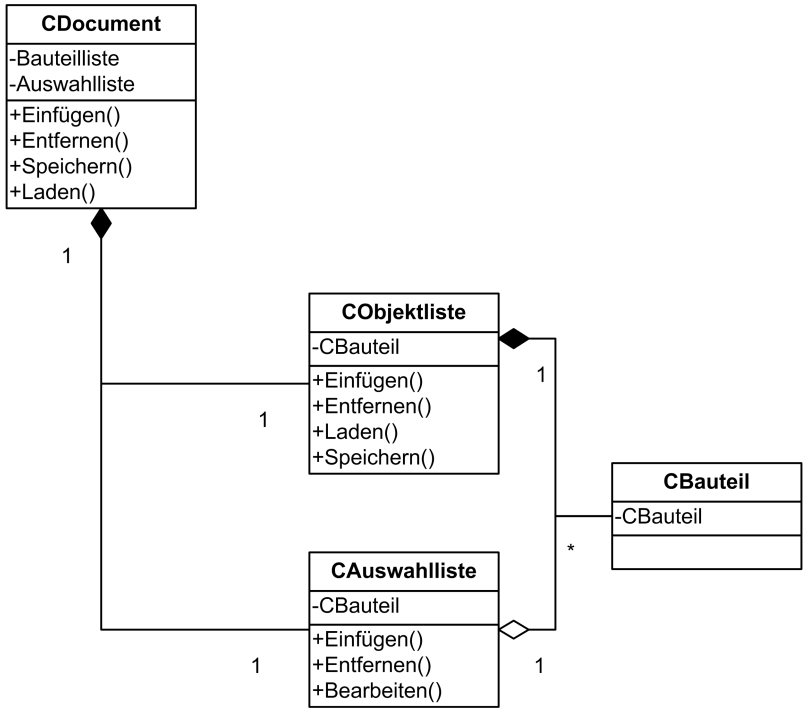


Abbildung 17.1: Vereinfachtes Klassendiagramm des Dokumentes

17.1.1 Folien in CAD-Programmen

17.1.1.1 Folien im klassischen Sinn

Die meisten Programme arbeiten mit Layern (engl. für Ebene oder Schicht) oder auch Folien genannt. Das Konzept der Folien stammt vom Arbeiten am Reisbrett, das eins zu eins auf den Computer übertragen wurde. Eine Zeichnung wurde auf mehrere Transparentpapiere verteilt, der komplette Plan erscheint nur, wenn man mehrere Folien übereinander legt. Eine mögliche Aufteilung ist:

1. Grundriss
2. Bemaßung
3. Einrichtung
4. Elektroinstallation
5. Entwässerung
6. etc.

Aus praktischen Gründen ist diese Methode am Reisbrett auf ungefähr 5 Lagen begrenzt, da bei mehr Folien der unterste Plan nicht mehr sichtbar ist. Am Computer ist die

Anzahl der Lagen praktisch unbegrenzt. Dieses Prinzip wurde von verschieden CAD-Programmherstellern übernommen, für die Organisation der Zeichnung wurden Folien eingeführt. Einzelne Bauteile wurden in verschiedene Folien gelegt. Sie wurden dadurch zu Gruppen zusammengefasst und können ein- und ausgeblendet werden. Durch die Folien kann auch die Ausgabe des Planes gesteuert werden. Durch das Ein- und Ausblenden von Folien wird bestimmt, was auf dem Plan zu sehen ist. Die verschiedenen Stiftstärken werden in vielen Programmen durch verschiedene Stiftfarben symbolisiert. Eine Farbe kann einen Stift programmweit bestimmen oder auch nur in einer Folie. Ein und dieselbe Farbe kann somit verschiedene Bedeutungen in einer Zeichnung haben.

17.1.1.2 Bedeutung der Folien in modernen Programmen

Folien haben den Übergang vom klassischen CAD-Programm zum modernen Programm mitgemacht, dabei hat sich ihre Bedeutung verändert. Anfänglich waren die Folien die einzige Möglichkeit der Organisation der Zeichnung. Durch die Verwendung objektorientierter Programmiersprachen, von Bauobjekten statt graphischen Objekten sowie eines zentralen Datenmodells kommen bei modernen Programmen weitere Möglichkeiten hinzu.

Bei der Verwendung des zentralen Datenmodells werden verschiedene Sichten für die Darstellung des Modells verwendet. In der Sicht kann die Sichtbarkeit der Objekttypen direkt über den Typ gesteuert werden, ohne dass diese hierzu in Folien angeordnet werden müssen Abb. 17.2.

Durch die Objektorientierung bei der Darstellung entfällt auch die Zuweisung der folienabhängigen Stifte, da diese objekt- und sichtabhängig sind (siehe auch Painter Kapitel 19).

Die Hauptaufgabe von Folien in modernen CAD-Programmen ist die Organisation des Datenmodells. Das zentrale Datenmodell ist eine virtuelle Abbildung des gesamten Gebäudes, für die Arbeit am Modell werden jedoch nur Teile davon benötigt. Die natürliche Aufteilung eines Gebäudes ist die Aufteilung in einzelne Stockwerke Abb. 17.3.

Bei dieser Aufteilung werden die Bauteile eines Stockwerkes einer Folie zugewiesen. In einer Sicht, kann durch die Aktivierung der Folien bestimmt werden, welche Stockwerke sichtbar sind.

In der Abbildung 17.4 wird schematisch die Organisation des Dokumentes dargestellt. Die einzelnen Bauteile sind in Folien organisiert, welche zusammen das Projekt ergeben.

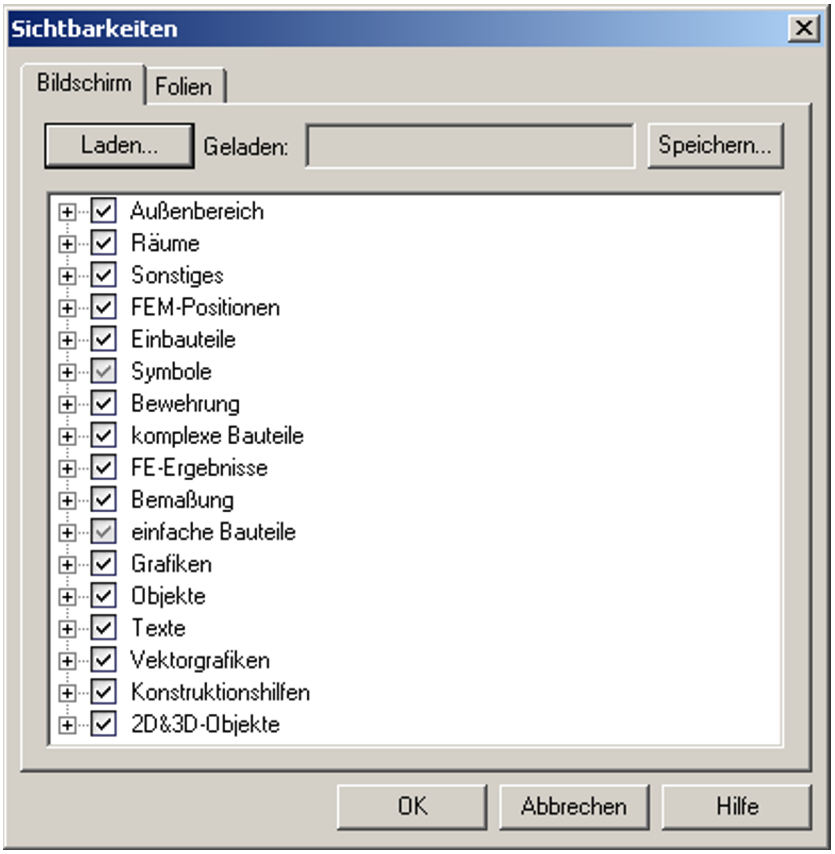


Abbildung 17.2: Steuerung der Sichtbarkeit von Objekttypen in Vicado

17.2 Verschiedene Sichtarten in CAD-Programmen

Die verschiedenen CAD-Programme verfügen heute über eine größere Anzahl von verschiedenen Sichttypen. Die in einem Programm vorhandenen Sichttypen sind abhängig vom Anwendungszweck des Programmes. Beispiele dieser Sichtarten sind in Abbildung 17.5 zu sehen. Viele der heute vorhandenen Viewtypen sind durch die Entwicklungsgeschichte der CAD-Programme bedingt. Die in einer Anwendung möglichen Viewtypen werden vom Datenmodell bestimmt, da die Darstellung in den Views aus diesem berechnet werden. Zu den heute gängigen Viewtypen zählen:

- 2D Sicht
- 3D Sicht
- Textsicht
- Plansicht

Jede dieser Viewarten hat ihren eigenen speziellen Zweck. Daneben existieren noch weitere spezielle Sichten auf das Datenmodell oder Teile davon wie Details und Dialoge.

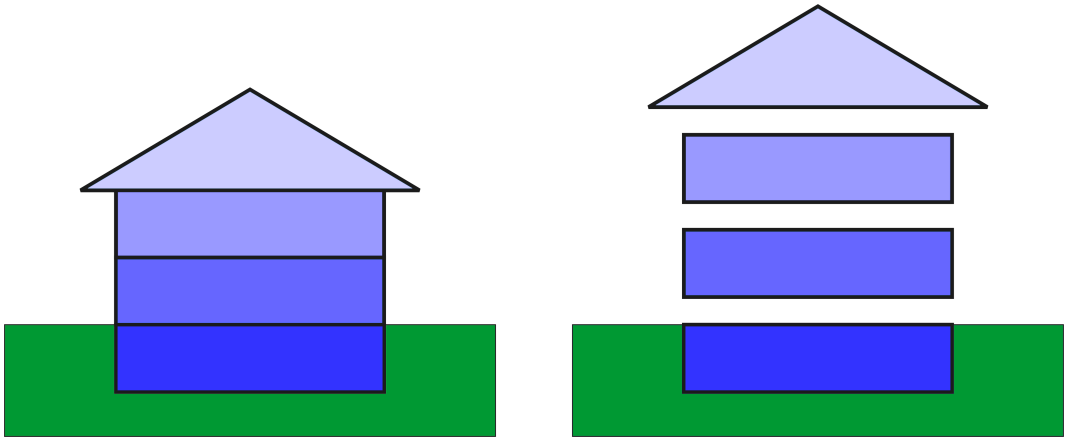


Abbildung 17.3: Zusammensetzung eines Gebäudes aus Stockwerken

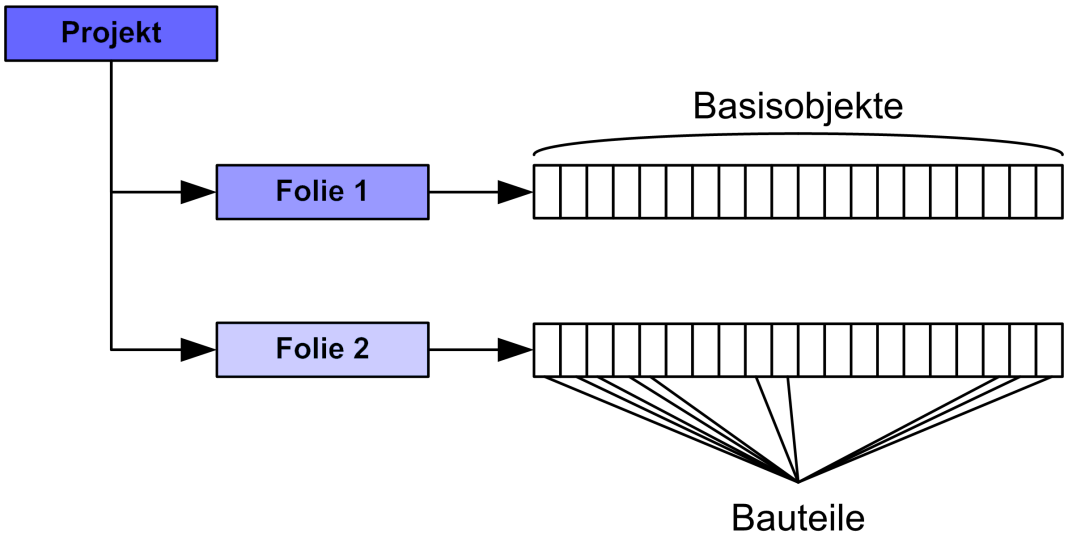


Abbildung 17.4: Organisation der Daten im Dokument

17.2.1 2D-View

Mittlerweile haben auch im Architekturbereich 3D-CAD-Programme den Markt erobert. Die Arbeit in diesen Programmen erfolgt jedoch nicht im dreidimensionalen Raum, sondern in der Regel in einer vom Anwender definierten Ebene. Dies ist auch bedingt durch das Arbeiten am Monitor, der eine Ebene bildet und somit kein direktes dreidimensionales Arbeiten möglich ist. Als Arbeitsebene bieten sich bei der Planung von Gebäuden die Stockwerke an. Dies entspricht auch der klassischen Planungsweise und der Denkweise des Konstrukteurs. Verschiedene Programme bieten die Möglichkeit der freie Konstruktion an. Umgesetzt wird dies, indem der Anwender beliebige Arbeitsebenen im Raum wählen kann. Die Eingabe erfolgt dann auf diese Ebene bezogen und ist dadurch wieder 2-dimensional. Die verschiedenen Anwendungsmöglichkeiten von 2D-Views sind:

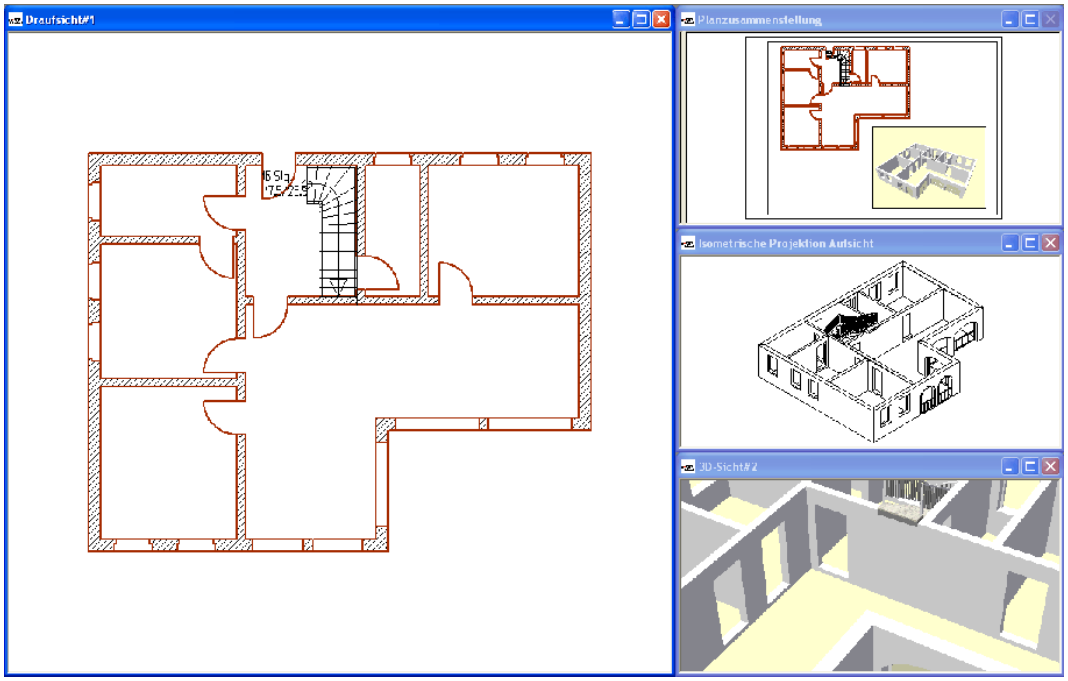


Abbildung 17.5: Viewklassen

- Grundrisse
- Ansichten
- Querschnitte
- Detail

Die verschiedenen 2d-Views Grundrisse, Ansichten, Schnitte und Details sind programmiertechnisch gesehen dasselbe. Grundrisse sind horizontale Schnitte durch das Gebäude, Querschnitte sind vertikale Schnitte durch das Gebäude und Ansichten sind Schnitte außerhalb des Gebäudes mit Blick auf das Bauwerk. Details sind vergrößerte Ausschnitte aus einer vorher genannten Sichtart, die mehrere Einzelheiten zeigen.

17.2.2 3D-View

Die modernste Sicht in einem CAD-Programm ist die 3D-View. Die 3D-View dient weniger der Konstruktion als der optischen Kontrolle des Ergebnisses oder der Präsentation der Planung. Die Art 3D-View ist abhängig vom Datenmodell, welches im CAD-Programm verwendet wird, da dieses bestimmt welche Informationen für die 3D-Ansichten vorhanden sind. Aus einem Kantenmodell können keine Flächen dargestellt werden, aus Flächenmodellen können keine Volumen ermittelt werden. Heutige 3D-Programme verwenden Volumenmodelle wie Brep oder CSG. Aus diesen komplexen Volumenmodellen lassen sich alle

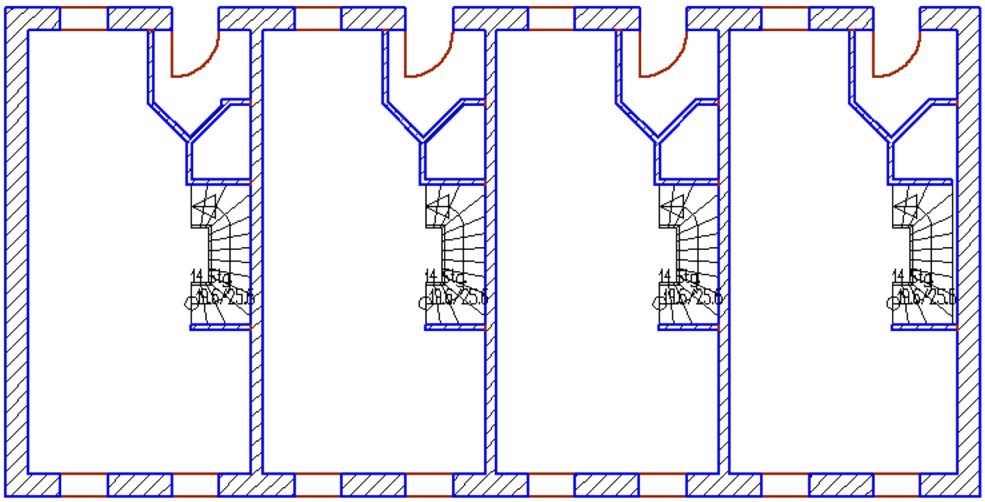


Abbildung 17.6: Ansicht einer 2D-View von Vicado

möglichen 3D-Sichten erzeugen:

- Drahtgitter - Kantensicht
- verdeckte Linien
- photorealistische Ansichten
- Animationen

Drahtgitter Ansichten sind mit die ältesten und mit am weitesten verbreiteten Ansichten. Dies liegt nicht an der Qualität der Darstellung, vielmehr an der in der Vergangenheit beschränkten Rechenkapazität der Computer. Die Bauteile werden nur durch die Projektion der Kanten dargestellt. Diese Darstellung ist von einem realen Bild weit entfernt, konnte aber auf den Computern in annehmbarer Zeit dargestellt werden.

Der nächste Schritt war die Berechnung der verdeckten Linien in der Ansicht. Das Ergebnis dieser Rechnung entsprach der eines Gebäudes, jedoch ohne Farbe und Texturen, diese Berechnungen sind noch sehr zeitaufwendig. Mitte der 90ger Jahre dauerte die Darstellung eines kleines Einfamilienhauses mit Berücksichtigung der verdeckten Kanten noch mehrere Minuten, nach jeder Änderung musste diese Rechnung erneut durchgeführt werden.

Eine Revolution setzte Mitte der 90ger Jahre ein. Durch die immer schneller werdende Hardware (erste 3D-Beschleuniger) werden photorealistische Abbildungen in annehmbarer

Abbildung 17.7: Ansicht einer 3D-View von Vicado

Zeit möglich. Es entsteht die modernste Form der 3D-View, die animierte View. Die auf dem Markt erscheinende Hardware ermöglicht Animationen in Echtzeit, diese können als Filme abgespeichert werden.

17.2.3 Textview

Die Textview ist kein Plan im klassischen Sinn, vielmehr eine Textsicht auf das Modell. Es gibt verschiedene Möglichkeiten von Textviews:

Einfache Liste Auflistung der vorhandenen Objekte ohne eine bestimmte Ordnung

Geordnete Liste Bei den geordneten Listen sind die Objekte nach einem bestimmten Muster angeordnet. Die Objekte können nach

- Objekttyp
- Folien
- Räume

geordnet sein.

Hierarchische Liste Die hierarchische Liste ist eine Sonderform der geordneten Liste. Bei ihr sind die Objekte in einem Baum angeordnet. Die Bäume können nach

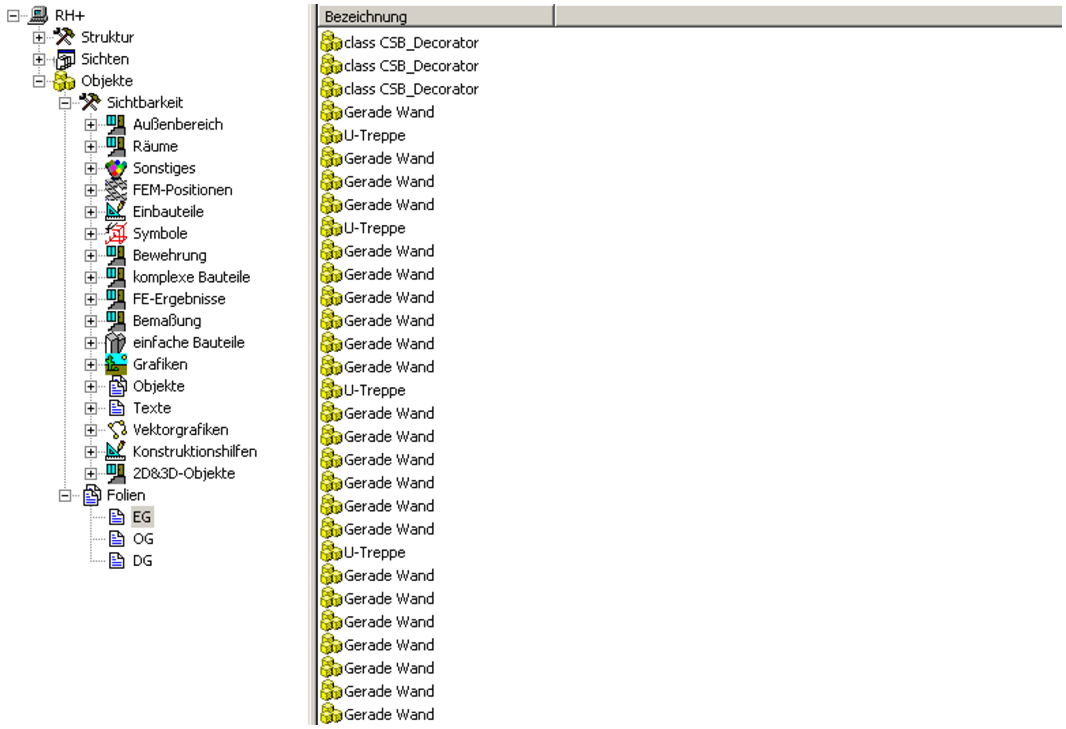


Abbildung 17.8: Ansicht einer Textview von Vicado

- dem Aufbau des Modells: Gebäude - Stockwerk - Raum ...
- der Hierarchie der Objekte: Objekttyp - Wand - Betonwände - Wandstärke ...

geordnet sein.

Gefilterte Liste Bei der gefilterten Liste kann der Anwender beeinflussen, welche Daten angezeigt werden, indem verschiedene Filter zur Auswahl der Objekte gesetzt werden.

Interaktive Liste Die interaktive Liste ist keine neue Listenart sondern eine spezielle Form der bisher vorgestellten Listen. Jede der vorgestellten Listen kann auch eine interaktive Liste sein. Bei der interaktiven Liste kann der Anwender die Objekte der Liste bearbeiten:

- Löschen von Objekten
- Verändern von Objekteigenschaften
- Kopieren von Objekten

Beim Arbeiten mit der interaktiven Textview kann es leicht zu Problemen kommen, da keine visuelle Kontrolle stattfindet. Die meisten Interaktionen sind ohne weiteres möglich, bei der Positionierung (Erzeugung) hat der Anwender jedoch keinen

Bezug zu den anderen Objekten, sodass er nicht erkennt, ob sich einzelne Objekte überschneiden.

17.2.4 Die Planview

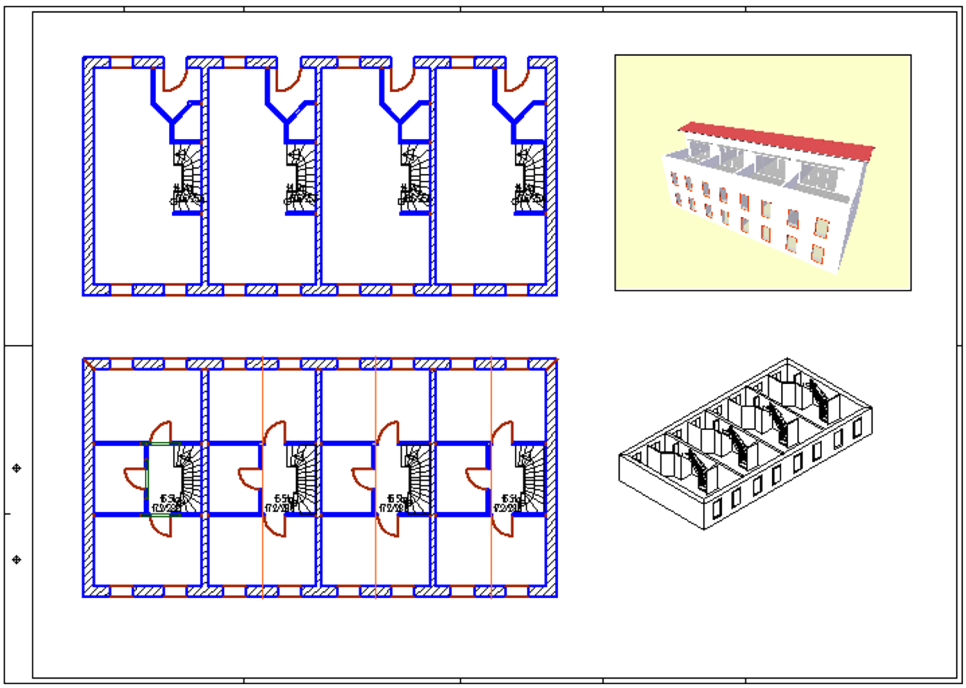


Abbildung 17.9: Ansicht einer Planview von Vicado

Die Planung des Gebäudes geschieht am digitalen Gebäudemodell. Das letztendliche Produkt einer jeden Planung ist ein zweidimensionaler Plan für die Baustelle. Die Pläne sind eine Zusammenstellung verschiedener Sichten auf das vorhandene Gebäudemodell. Die Pläne werden für verschiedene Zwecke benötigt:

- Entwurfsplan
- Genehmigungsplan
- Ausführungsplan
- Schal- und Bewehrungsplan

Die traditionelle Version zur Erstellung von Plänen ist die Erzeugung von graphischen Darstellungen aus dem vorhandenen Modell, diese werden auf dem Plan platziert. Bei dieser Methode werden die Verknüpfungen zwischen der Zeichnung und dem Modell zerstört. Bei einer nachträglichen Änderung des Gebäudemodells müssen die betroffenen Sichten

neu erzeugt und anschließend auf dem Plan wieder platziert werden. Einige Programme sind mittlerweile so weit, dass diese Aktualisierung auf Knopfdruck stattfindet. Jedoch ist es dann meist immer noch nicht möglich, diese Sichten nachträglich im Maßstab und Ausschnitt zu ändern.

Ein flexibler Ansatz ist es, die vorhandenen Views für den Plan zu verwenden. Die am meisten in den Plänen verwendete Darstellungen sind die Grundrisse der Stockwerke. Diese sind während der Konstruktion als eigenständige View erzeugt worden. Ein Plan lässt sich auf einfache Weise erzeugen, indem man die vorhandenen Sichten in einem neuen Plan positioniert. Bei dieser Planview handelt es sich somit um eine Multiview in der mehrere Views gleichzeitig angezeigt werden können.

Eine auf einem Plan positionierte Ansicht ist eine mit dem Dokument verknüpfte View. Sie wird bei einer Änderung des Modells automatisch aktualisiert. Die Eigenschaften der einzelnen Views lassen sich in der Plansicht nachträglich ändern, so dass der Ausschnitt und der Maßstab nachträglich angepasst werden kann.

Eine Ansicht wird in der Regel nur einmal in einem Plan benötigt. Von der Verwendung einer View in mehreren Plänen wie z. B. ein und demselben Grundriss in der Genehmigungsplanung, Ausführungsplanung für den Rohbau, Ausführungsplanung für die Installation ist abzuraten, da in den einzelnen Plänen je nach Art eine andere Darstellung gewählt und andere Details gezeigt werden.

- Genehmigungsplanung nur Außenmaße
- Rohbau, Ausführungsplanung mehr Maße, Schlitze, Details
- Ausführung Installation, Bemaßung der Einbauten statt des Bauwerkes

Verwendet man ein und die selbe View in mehreren Plänen, hätte dies schwer zu kontrollierende Auswirkungen, da der Anwender schnell die Übersicht verliert welche View in welchem Plan verwendet wird. Die Auswirkung wird an einer View beschrieben, die in der Ausführungs- und auch in der Genehmigungsplanung verwendet wird. Würde in der Genehmigungsplanung eine Maßkette gelöscht, wird diese auch in der Ausführungsplanung entfernt. Diese wird aber für die Ausführung benötigt und der Plan wird dadurch unvollständig.

Sollte es doch einmal nötig sein, dass ein und dieselbe View in verschiedenen Plänen benötigt wird, lässt sich dies einfach durch die Möglichkeit der Erstellung einer View als Kopie einer vorhanden View bewerkstelligen. Die neue View bekommt einen eindeutigen Namen, so dass auch sichtbar ist, welche View zu welchem Plan gehört.

17.3 Viewabhängige Daten

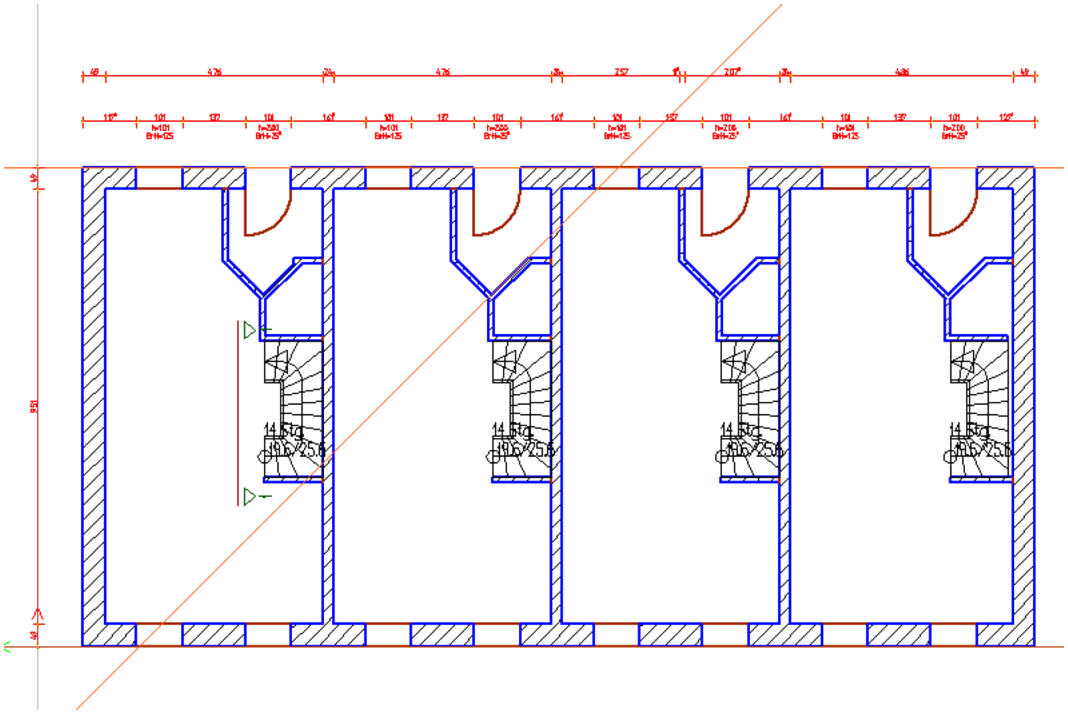


Abbildung 17.10: View mit viewspezifischen Elementen von Vicado

Die Darstellungen in den einzelnen Views werden in der Regel aus dem Datenmodell ermittelt. Daneben werden in den Views Elemente angezeigt, die nicht zum Datenmodell gehören. Die viewspezifischen Elemente dienen zur Ergänzung oder Abänderung der vorhandenen Zeichnung oder unterstützen bei der Konstruktion des Gebäudemodells: Die Objekte lassen sich in drei Gruppen einteilen:

- Ergänzung der Zeichnung: Bemaßung
- Änderung/Ergänzung der Zeichnung: graphische Elemente
- Objekte für die Konstruktion: Hilfslinien

Diese Elemente sind auch stets verbunden mit der View in der Sie erzeugt wurden. Durch die Verknüpfung mit der View wird eine viewbezogene Datenhaltung benötigt. Hierfür gibt es zwei Möglichkeiten.

Eine Möglichkeit ist die Speicherung der Daten bei der View. Mit dem Abspeichern der View müssen auch die dazugehörigen Daten gesichert werden. Bei einem Löschen der View werden auch die viewspezifischen Daten gelöscht und sind verloren.

Die andere Möglichkeit ist die zentrale Speicherung der Daten beim Gebäudemodell. Dies

bedeutet einen höheren Verwaltungsaufwand, da diese Objekte der View zugeordnet werden müssen. Beim Verlassen des Programms müssen diese Verknüpfungen mit abgespeichert werden. Gleichzeitig muss sichergestellt werden, dass nach dem Löschen einer Ansicht auch die zentral gespeicherten Daten entfernt werden, damit kein Datenmüll im Modell zurückbleibt.

17.3.1 Bemaßung

Eine Bemaßung wird für eine Sicht mit einem bestimmten Zweck erzeugt. Je nach Verwendungszweck des Planes ändert sich auch die Bemaßung. In der Genehmigungsplanung sind gegenüber der Ausführungsplanung nur wenige Maße vorhanden. In einer Detailansicht ist nur ein Ausschnitt mit den für das Detail benötigten Maßen zu sehen.

17.3.2 Graphische Elemente

Die graphischen 2D-Elemente dienen zur Ergänzung der aus dem Datenmodell erstellten Sichten. Zu den graphischen Elementen zählen:

- Linie, Bogen, Ellipsenbogen, Polygon
- Flächen wie Kreis, Rechteck, Ellipse, geschlossenes Polygon
- Bilder
- Texte

17.3.3 Hilfslinien

Hilfslinien werden für die Konstruktion des Bauwerksmodells benötigt. Die Konstruktion eines Objektes findet in der Regel nur in einer View statt. Die Hilfslinien werden somit nur in dieser View benötigt. In den anderen Views wirken sie in der Regel störend, da sie dort keine Bedeutung haben und sich mit den zu der View gehörenden Hilfslinien überlagern. Der Planer wird durch die Fülle der Hilfslinien abgelenkt und verwirrt. Bei den Hilfslinien lassen sich

Geraden Die gängigsten Hilfslinien sind Geraden. Sie dienen z. B. zur Erzeugung von Achsen.

Radiale Kreise

Polygone

unterscheiden. Zusätzlich kann man auch noch lokale Raster (radial und kartesisch) im weiteren Sinn zu den Hilfslinien zählen.

18 Das Builderkonzept

18.1 Idee der Builder

Das Builderkonzept ist ein Konzept zur Erzeugung von Bauteilen in einem CAD-Programm. Die Eingabe der Bauteile erfolgt in der Regel graphisch, dabei ist das Vorgehen unabhängig von den Bauteilen. Das Builderkonzept beruht auf dem Entwurfsmuster Builder (siehe Seite 20ff). Beim Builderkonzept wird die Erzeugung von dem Objekt getrennt, hierdurch können die Objekte auf verschiedene Art und Weise erzeugt werden. Die Erzeugungsalgorithmen sind unabhängig von den Objekten und sie können dadurch geändert oder ergänzt werden, ohne dass dies eine Auswirkung auf die Objekte hat.

18.2 Aufgaben des Builder

Die Funktionen des Builders lassen sich in drei verschiedene Teile aufteilen, in Aktualisieren, Visualisieren und Erzeugen.

Aktualisieren Aktualisieren ist das Reagieren des Builders auf Ereignisse, hierzu zählen Mausbewegung, betätigen der Maustasten (rechte, mittlere, linke) und Tastatureingaben.

Visualisieren Visualisieren ist Darstellung des aktuellen Zustandes während der Konstruktion, eine Vorschau auf das neu zu erzeugende Bauteil. Da die Visualisierung eine Vorschau auf das neu zu erstellende Bauteil bietet, ist diese objektspezifisch.

Erzeugen Nach Beendigung der Eingabe erzeugt der Builder das neue Objekt, dies wird anschließend in das Gebäudemodell eingefügt.

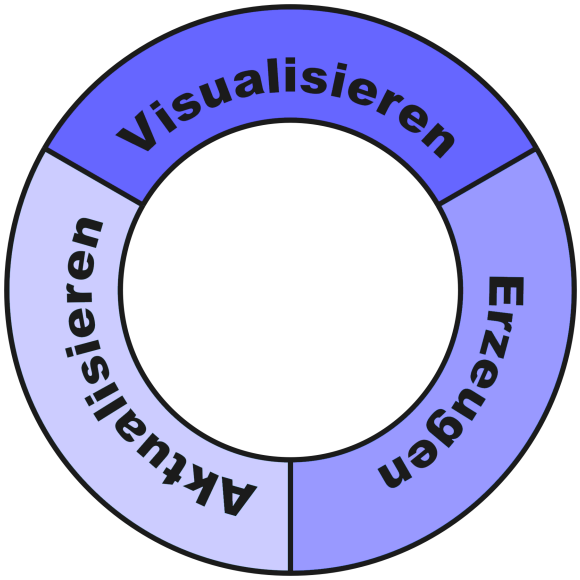


Abbildung 18.1: Darstellung der Anforderungen an den Builder

18.3 Builderarten

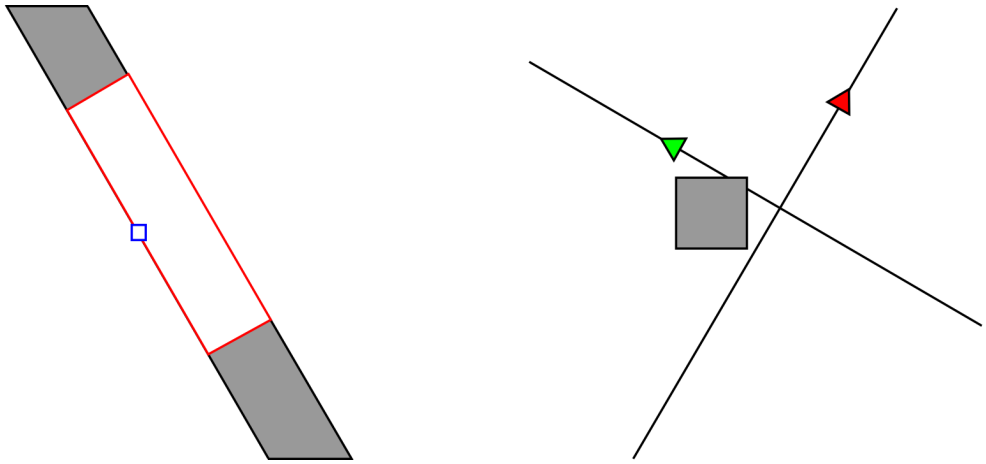
Mit Hilfe des Builderkonzeptes lassen sich verschiedene Arten von Buildern implementieren, die sich in der Anzahl der Punkte, die für die Erzeugung benötigt werden unterscheiden. Die Anzahl der benötigten Punkte reicht von einem bis zu n, grundsätzlich benötigt man zum Erzeugen von dreidimensionalen Objekten vier Punkte, die ein umschließendes Rechteck beschreiben. Die Anzahl der Punkte kann durch die Angabe ergänzender Werte reduziert werden. Für GoCAD wurden vier verschiedene Typen von Buildern verwirklicht.

18.3.1 Einpunktbuilder

Einpunktbuilder benötigen für die Erzeugung der Bauteile die Eingabe eines Punktes. Über diesen Punkt wird die Position des neuen Bauteils dem Builder übergeben. Weitere Informationen, die der Builder für die Erzeugung benötigt, müssen auf andere Art und Weise übergeben werden. Hierfür gibt es mehrere Möglichkeiten

18.3.1.1 Ermittlung der Informationen aus dem Kontext

Eine Möglichkeit, die fehlenden Informationen für die Erzeugung zu ermitteln, ist aus dem Kontext des Objektes und der Position. Einzelne Objekte stehen in Beziehung zu verschiedenen anderen Objekten, so gehören z. B. Stürze zu Öffnungen und sind Bestandteile von



(a) Erzeugung einer Tür

(b) Erzeugung einer Stütze

Abbildung 18.2: Beispiele für Einpunktbuilder

Wänden.

Setzt man als Bedingung für die Erzeugung eines Sturzes das Vorhandensein einer Öffnung bzw. einer Mauer voraus, so erhält man aus diesen Bedingungen die fehlenden Angaben. Die Stütze haben die gleiche Breite wie die Mauern somit kann diese Angabe durch den Builder von der Mauer, abgefragt werden. Als weitere Angabe wird die Länge des Sturzes benötigt. Die Länge des Sturzes kann über die Größe der Öffnung ermittelt werden, jedoch reicht dieses nicht aus, da in diesem Fall die Stürze die gleiche Länge wie die Öffnung hätte. Es wird noch die Länge des Auflagers benötigt, die jedoch als Standardangabe im Builder hinterlegt wird, wodurch daraus die Gesamtlänge ermittelt werden kann.

18.3.1.2 Eingabe der Informationen über einen Dialog

Die zweite Möglichkeit für die Erzeugung fehlender Informationen ist die Eingabe über einen Eigenschaftsdialog oder die Wieleiste. Die Erzeugung mit einem Einpunktbuilder ist möglich bei Rundstützen durch ihre Rotationssymmetrie, während er bei Stützen mit quadratischem Querschnitt nur bedingt verwendet werden kann. Über die Koordinate wird die Position der Stütze bestimmt, der Radius und die Höhe der Stütze wird über den Dialog eingegeben. Bei der Erzeugung einer quadratischen Stütze wird noch die Richtung im Raum benötigt, die bei der Rundstütze infolge der Rotationssymmetrie nicht benötigt wird. Die Richtung der Quadratstütze muss beim Einpunktbuilder entweder im Nachhinein angepasst, was einen zusätzlichen Arbeitsschritt bedeutet, oder vom Koordinatenkreuz übernommen werden, das dann im Vorhinein auf die richtige Richtung eingestellt werden muss.

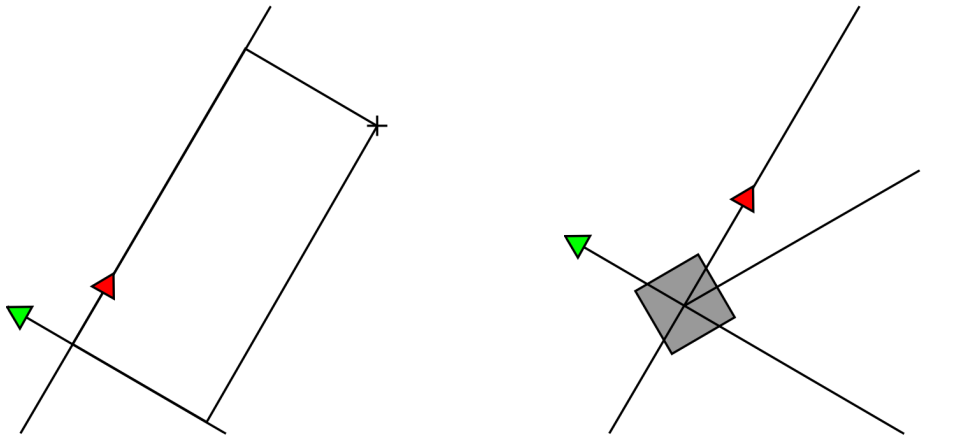
18.3.2 Zweipunktbuilder

Zweipunktbuilder benötigen für die Erzeugung von Bauteilen zwei Punkte. Aus diesen beiden Koordinaten lassen sich verschiedene Daten für die Objekte berechnen:

Abmessungen: Von den beiden Koordinaten wird der Abstand ermittelt und dieser als Maß bei der Erstellung des Objektes verwendet z. B. als Radius einer Rundstütze.

Richtung: Ermittlung einer Richtung im Raum aus den beiden Punkten, der Abstand der Punkte hat keine weitere Bedeutung (Abb. 18.3). Diese Richtung wird zur Ausrichtung des Objektes im Raum verwendet.

Maß und Richtung: Kopplung der beiden bisher vorgestellten Möglichkeiten zu einer Einzigen. Aus den beiden Punkten wird eine Länge und eine Richtung im Raum ermittelt. Verwendet wird dies u. a. für Wände als Länge und Richtung



(a) Erzeugung einer Decke

(b) Erzeugung einer Stütze

Abbildung 18.3: Beispiele für Zweipunktbuilder

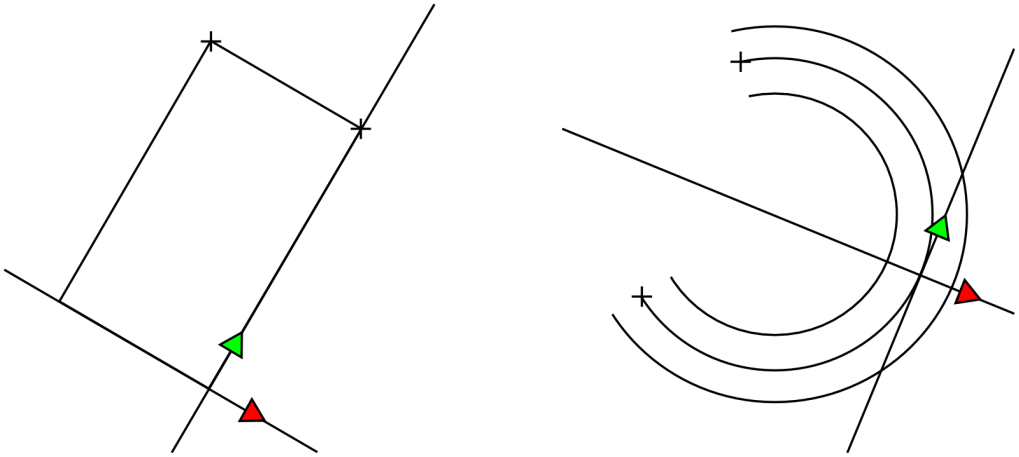
18.3.3 Dreipunktbuilder

Die dritte Variante der Builder ist der Dreipunktbuilder, dieser verwendet drei Punkte für die Erzeugung der Objekte. Aus diesen Punkten können

Maße und Richtung: Ermittlung von zwei Maßen und der Richtung im Raum für die Erzeugung von Objekten. Aus den beiden ersten Punkten wird die Richtung im Raum und das erste Maß ermittelt, der dritte Punkt ist für das zweite Maß. Ein Beispiel ist eine rechteckige Platte (Abb. 18.4(a)).

Komplexe Objekte: Erzeugung von Objekten für deren Konstruktion drei Punkte benötigt werden wie Kreisbögen durch Mittelpunkt, Anfangs- und Endpunkt oder Kreise durch drei Punkte, die auf dem Umfang liegen (Abb. 18.4(b)).

erzeugt werden.



(a) Decke (b) runde Wand

Abbildung 18.4: Beispiele für Dreipunktbuilder

18.3.4 Polygonbuilder

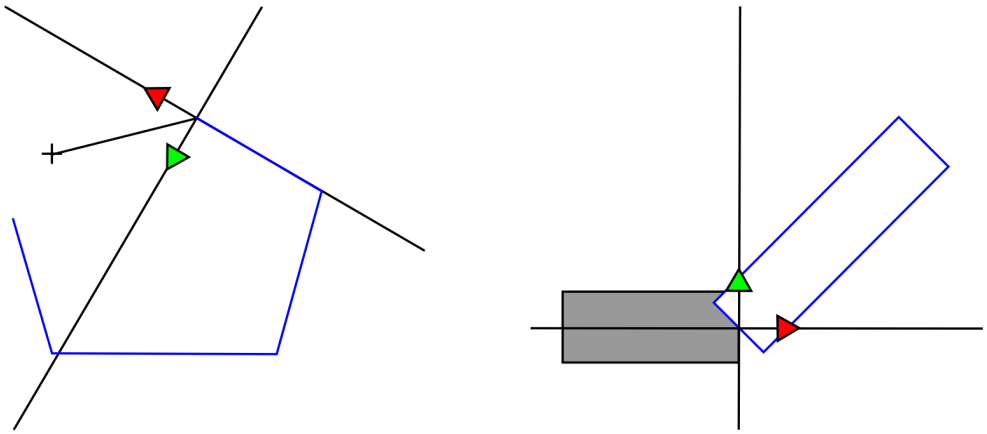
Die vierte und letzte Variante die vorgestellt wird, ist der Polygonbuilder, er wird verwendet für Objekte mit n-Punkten. Diese Objekte lassen sich in zwei Gruppen unterteilen:

Unregelmäßige Objekte Abb. 18.5(a): Erzeugung von Objekten mit unregelmäßiger Form, alle polygonalen Objekte wie Platten oder Scheiben. Das Polygon kann neben geraden Segmenten auch noch aus Elypsenabschnitten bestehen.

Fortlaufende Objekte Abb. 18.5(b): für Objekte, die aus einem fortlaufenden Polygonzug bestehen wie Wände. Der letzte Punkt einer Wand dient dabei als Anfangspunkt der nächsten Wand.

18.4 Funktionsweise der Builder

Die Arbeitsweise der Builder wird am Beispiel eines Wandbuilders erläutert, hierzu wurde ein Zweipunktbuilder ausgewählt. Durch die Auswahl der Bauteilgruppe und anschließend



(a) Decke (b) Wand

Abbildung 18.5: Beispiele für Polygonbuilder

des Bauteiltyps werden die zur Verfügung stehenden Builder angezeigt. Durch einen Klick auf den Builderbutton wird der Builder gestartet.

Nach dem Start des Builders wird die Eigenschaftsseite des Builders erzeugt und in der Wieleiste angezeigt. Über die Wieleiste werden die zusätzlichen Parameter, die für die Erzeugung der Wand benötigt und nicht über die graphische Konstruktion ermittelt werden können, eingegeben.

Die Eingabe beginnt mit dem Setzen des Startpunktes, die View reagiert auf die Nachricht OnLeftButtonUp. Die Funktion erhält die Bildschirmkoordinaten in Pixel und berechnet daraus die Position in der Ebene.

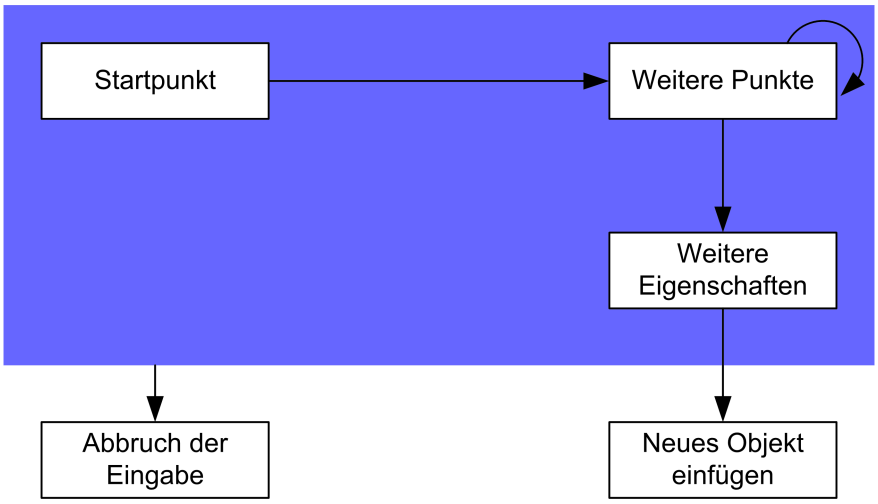


Abbildung 18.6: Schema des Konstruktionsablaufes mit einem Builder

Der Startpunkt wird an den Builder übergeben, in die Punktliste des Builders einge-

fügt und anschließend der Ursprung auf die aktuelle Position gesetzt. Der Builder wartet danach auf die Eingabe des nächsten Punktes.

Nach der Eingabe eines weiteren Punktes wird auch dieser an den Builder übergeben. Darauf folgt die Kontrolle der weiteren Randbedingungen. Hierbei wird überprüft, ob die aktuelle Position in der Nähe eines bestehenden Objektes liegt, bzw. wird an ein vorhandenes Raster angepasst und ggf. werden Konstruktionsregeln berücksichtigt. Danach wird die Position in die Punktliste eingefügt. Abschließend wird abgefragt, ob alle benötigten Punkte vorhanden sind. Ist dies nicht der Fall, so wird die Darstellung des Builders in den Views gelöscht, durch die aktuelle Darstellung ersetzt und die Richtung und Position des Ursprungs angepasst.

Die Punkteingabe wird so lange wiederholt, bis die Konstruktion vom Benutzer abgebrochen wird oder alle benötigten Punkte eingegeben wurden. Sind alle Punkte eingegeben, werden die weiteren Eigenschaften des Bauteils aus der Wieleiste ausgelesen und mit den aus den Eingaben ermittelten Parametern das neue Objekt erzeugt und zum Einfügen an das Modell übergeben.

Wird während der Konstruktion eine der Eigenschaften in der Wieleiste geändert, erhält der Builder die Nachricht „Change“. Als Reaktion auf diese Nachricht werden durch den Builder die aktuellen Einstellungen der Wieleiste abgefragt und anschließend die neue Darstellung mit den aktuellen Parametern gezeichnet.

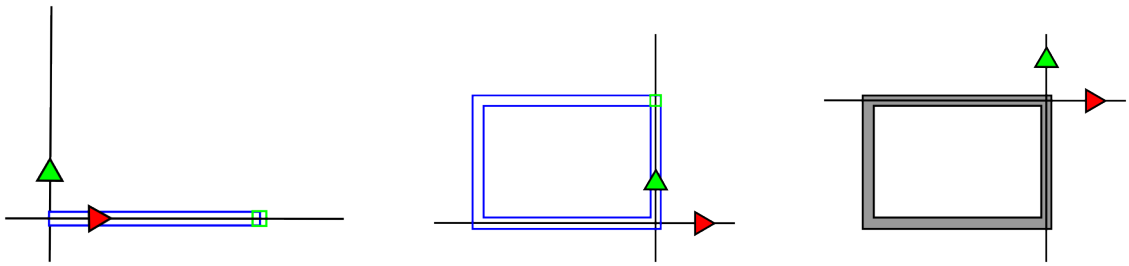
Über die Wieleiste werden die weiteren Parameter, die durch die Konstruktion in der Ebene nicht ermittelt werden können, eingegeben. Die Parameter sind vom Objekt und Buildertyp abhängig, die Höhe des Bauobjektes, die Position des Objektes in Relation zur Eingabe und das Niveau der Eingabeebene gehören in der Regel zu jedem Builder.

Die Zeichenfunktion `zeichnenView()` ist in der Basisklasse als virtuell deklariert, die Implementierung findet erst in den abgeleiteten Klassen statt. Die Zeichenfunktion ist abhängig von dem aktuellen Zustand des Builders, d. h. von den folgenden drei Punkten.

- den bis zu dem Zeitpunkt eingegebenen Punkten
- den Einstellungen in der Wieleiste
- der aktuellen Mausposition in den aktiven Views.

Ändert einer der Punkte seinen Zustand, so wird die Darstellung neu berechnet und in den Views aktualisiert. Abbildung 18.7 zeigt die Darstellung des Dreipunktwandbuilders nach der Eingabe des ersten und zweiten Punktes.

Die Funktion `bauen()` wird ebenso wie die `zeichnenView()` von der abgeleiteten Klas-



(a) 1. Punkt (b) 2. Punkt (c) 3. Punkt

Abbildung 18.7: Builderzustände während der Konstruktion

se überschrieben, da erst zu späterem Zeitpunkt bekannt ist, welches Objekt der Builder erzeugt. Die Funktion ruft den Konstruktor des zu erzeugenden Objektes auf und gibt einen Pointer auf das erzeugte Objekt zurück, der dann in das Dokument eingefügt wird. Die Punkteingabe `PunktÜbergeben()` erbt der Builder von der Klasse *CBefehl* und überschreibt diese Funktion, sofern bei der Konstruktion bestimmte geometrische Gesetzmäßigkeiten wie die Rechtwinkligkeit von Teilstrecken zueinander eingehalten werden müssen.

19 Das Painterkonzept

In Kapitel 17 wurde das Model-View-Controller-Konzept im Zusammenhang mit CAD-Programmen erläutert. Durch das MVC-Konzept kommt es zu einer Trennung zwischen den Daten und den Ansichten. Die Daten liegen in einer allgemeinen Form, unabhängig von den Views im Modell vor. In heutigen CAD-Programmen sind mehrere Viewtypen wie 2D-View und 3D-View mittlerweile standardmäßig implementiert. Für eine Darstellung des Datenmodells in einer View müssen die allgemeinen Daten aus dem Datenmodell in das Format der entsprechenden View konvertiert werden.

Bei der Konvertierung der Daten des Modells werden verschiedene Algorithmen benötigt. Diese Algorithmen sind in erster Linie abhängig von der Art der View, in der das Modell dargestellt werden soll, einen weiteren Einfluss auf den Algorithmus hat die gewünschte Darstellung. So kann ein Objekt im gleichen Viewtyp auf verschiedene Art und Weise dargestellt werden. Ein Beispiel hierfür ist eine Tür. Diese kann in einer Ansicht als Symbol oder auch als Darstellung der realen Geometrie des Objektes gezeichnet werden.

19.1 Ansprüche an die Darstellung

Die Ansprüche eines Anwenders eines CAD-Programms an die Darstellung eines Gebäudes sind sehr vielfältig und individuell. Sie hängen u. a. von der Intension des Anwenders, dem Planungsstand, Normen, und der Ansichtsart ab.

19.1.1 Planungsphasen der HOAI

Der Planungsablauf im Bauwesen wird in der HOAI beschrieben. Im Folgenden wird nur auf Teil II. der HOAI eingegangen „Leistungen bei Gebäuden, Freianlagen und raumbildenden Ausbauten“. Die anderen Teile der HOAI „Leistungen bei Ingenieurbauwerken und Verkehrsanlagen“, „Städtebauliche Leistungen“, „Leistungen bei der Tragwerkspla-

nung“ etc. sind entsprechend und werden im Moment nicht näher betrachtet. In §15 der HOAI werden neun Planungsphasen aufgelistet. Zu den folgenden vier Phasen gehören zeichnerische Darstellungen:

- 2. Vorplanung
- 3. Entwurfsplanung
- 4. Genehmigungsplanung
- 5. Ausführungsplanung

In jeder dieser Planungsphasen werden unterschiedliche Ansprüche an die Darstellung gestellt. Es wird dabei zwischen Grundleistungen und besonderen Leistungen unterschieden.

- Vorplanung

  Grundleistungen Darstellung und Bewertung, zum Beispiel versuchsweise zeichnerische Darstellungen, Strichskizzen, gegebenenfalls mit erläuternden Angaben.

  Besondere Leistungen Anfertigen von Darstellungen durch besondere Techniken, wie zum Beispiel Perspektiven, Muster, Modelle

- Entwurfsplanung

  Grundleistungen Zeichnerische Darstellung des Gesamtentwurfs, wie durchgearbeitete, vollständige Vorentwurfs- und/oder Entwurfszeichnungen (Maßstab nach Art und Größe des Bauvorhabens; bei Freianlagen im Maßstab 1:500 bis 1:100, insbesondere mit Angaben zur Verbesserung der Biotopfunktion, zu Vermeidungs-, Schutz-, Pflege- und Entwicklungsmaßnahmen sowie zur differenzierten Bepflanzung; bei raumbildenden Ausbauten: im Maßstab 1:50 bis 1:20, insbesondere mit Einzelheiten der Wandabwicklungen, Farb-, Licht- und Materialgestaltung), gegebenenfalls auch Detailpläne mehrfach wiederkehrender Raumgruppen

- Genehmigungsplanung

  Grundleistungen Erarbeiten der Vorlagen für die nach öffentlich-rechtlichen Vorschriften erforderlichen Genehmigungen.

- Ausführungsplanung

  Grundleistungen Durcharbeiten der Ergebnisse der Leistungsphasen 3 und 4 (stufenweise Erarbeitung und Darstellung der Lösung) unter Berücksichtigung

städtebaulicher, gestalterischer, funktionaler, technischer, bauphysikalischer, wirtschaftlicher, energiewirtschaftlicher (z. B. hinsichtlich rationeller Energieverwendung und der Verwendung erneuerbarer Energien) und landschaftsökologischer Anforderungen unter Verwendung der Beiträge anderer an der Planung fachlich Beteiligter bis zur ausführungsreifen Lösung, zeichnerische Darstellung des Objekts mit allen für die Ausführung notwendigen Einzelangaben, z.B. endgültige, vollständige Ausführungs-, Detail- und Konstruktionszeichnungen im Maßstab 1:50 bis 1:1, bei Freianlagen je nach Art des Bauvorhabens im Maßstab 1:200 bis 1:50, insbesondere Bepflanzungspläne mit den erforderlichen textlichen Ausführungen. Bei raumbildenden Ausbauten: Detaillierte Darstellung der Räume und Raumfolgen im Maßstab 1:25 bis 1:1, mit den erforderlichen textlichen Ausführungen

Die Zeichnungen beginnen bei einfachen Strichzeichnungen zur Darstellung der ersten Entwürfe. Die Darstellung wird immer weiter verfeinert, mehr Details werden gezeigt. Es werden verschiedene Maßstäbe verwendet, die Auswirkungen auf die Abbildung der einzelnen Bauteile haben. Die Genehmigungsplanung ist abhängig von öffentlich-rechtlichen Vorschriften für die Baugenehmigung, während die Ausführungsplanung alle für die Ausführung des Gebäudes notwendigen Details und Informationen enthalten muss.

19.1.2 Vorschriften für die Darstellung

Für die Erstellung von Bauzeichnungen gibt es eine Reihe von Vorschriften. Die zentrale Vorschrift ist die DIN ISO 128 „Technische Zeichnungen“ mit ihren verschiedenen Teilen. Die einzelnen Teile der DIN ISO 128 lösen seit Erscheinen der ersten Teile 1999 Stück für Stück die DIN 1356 „Bauzeichnungen“ ab. Die DIN ISO 128 beinhaltet allgemeine Teile die die Grundregeln enthalten und spezielle Teile für die Anwendung im Bauwesen, aber auch in der technischen Mechanik, dem Schiffbau und speziellen Anwendungen. Teil 20 beinhaltet die Grundregeln für die Linien. In Tabelle 1 der DIN werden die Grundarten der Linien aufgeführt:

———————————	Volllinie
— — — — — — — — — —	Strichlinie
— — — — — —	Strich-Abstand-Linie
——— · ——— · ——— · ——— · ——— ·	Strich-Punktlinie (Langer Strich)
——— · · ——— · · ——— · · ——— · ·	Strich-Zweipunktlinie (Langer Strich)
——— · · · ——— · · · ——— · · · ——— · · ·	Strich-Dreipunktlinie (Langer Strich)

.....	Punktlinie
——— – ——— – ——— – ——— –	Strich-Strichlinie
——— – – ——— – – ——— – – ———	Strich-Zweistrichlinie
—— · —— · —— · —— · —— · —— · —— ·	Strich-Punktlinie
—— —— · —— —— · —— —— · —— —— ·	Zweistrich-Punktlinie
—— · · —— · · —— · · —— · · —— · · —— · ·	Strich-Zweipunktlinie
—— —— · · —— —— · · —— —— · · —— —— · ·	Zweistrich-Zweipunktlinie
—— · · · —— · · · —— · · · —— · · · —— · · ·	Strich-Dreipunktlinie
—— —— · · · —— —— · · · —— —— · · · ——	Zweistrich-Dreipunktlinie

Tabelle 19.1: Linienarten nach EN ISO 128-20:2001 (D)

Beschreibung und Darstellung	Anwendung
Volllinie schmal	Kurze Mittellinien, Maßhilfslinien, Maßlinien und Maßbegrenzungen, Hinweislinien
Volllinie breit	Sichtbare Umrisse von Teilen mit Schraffur, sichtbare Umrisse von Teilen in Ansicht, vereinfachte Darstellung von Türen, Fenstern, Treppen etc.
Volllinie sehr breit	Sichtbare Umrisse von Teilen in Schnitten ohne Schraffur, Bewährungsstähle
Strichlinie schmal	Nicht sichtbare Umrisse
Strichlinie breit	Verdeckte Umrisse
Strichpunktlinie schmal	Mittellinien, Rahmen für vergrößerte Einzelteile
Strichpunktlinie breit	Schnittebene

Tabelle 19.2: Linienarten und Verwendung

Das Verhältnis zwischen den einzelnen Linienbreiten ist mit $1:\sqrt{2}$ festgelegt, hieraus ergeben sich die folgenden Abstufungen: 0,13mm, 0,18mm, 0,25mm, 0,35mm, 0,5mm, 0,7mm, 1,0mm, 1,4mm, 2,0mm. Das Verhältnis der Linienbreite in einer Zeichnung ist 4:2:1 für sehr breite : breite : schmale Linien. Teil 21 „Ausführung von Linien mit CAD-Systemen“ betrifft CAD-Programme direkt, er dient zu Vereinheitlichung der Darstellung von unter-

brochenen Linien in CAD-Systemen.

Teil 23 „Linien in Zeichnungen des Bauwesens“ spezifiziert die Linien. 19.1 des Teiles listet die Anwendung der einzelnen Linienarten auf, die hier auszugsweise wiedergegeben wird.

In 19.2 sind die Liniengruppen angegeben, die Bezeichnung richtet sich nach der Breite für die breite Linie.

Liniengruppe	Schmale Linie	Breite Linie	Sehr breite Linie	Linienbreite graph. Symbole
0,25	0,13	0,25	0,5	0,18
0,35	0,18	0,35	0,7	0,25
0,5	0,25	0,5	1	0,35
0,7	0,35	0,7	1,4	0,5
1	0,5	1	2	0,7

Tabelle 19.3: Linienstärken

Teil 50 „Grundregeln für Flächen in Schnitten und Schnittansichten“ befasst sich u. a. mit Schraffuren von Flächen, in ihr sind in Bild NB2 Schraffuren für Schnittflächen und Kennzeichnungen von Naturstoffen festgelegt.

Die DIN ISO 7519 „Zeichnungen für das Bauwesen - Allgemeine Grundlagen für Anordnungspläne und Zusammenbauzeichnungen“ dient als Ergänzung zur DIN ISO 128 für Bauzeichnungen, vorwiegend im Bereich der Architekturzeichnungen. Wesentlicher Bestandteil sind die Linienarten in den Ansichten und Schnitten. Sie beinhaltet auch Symbole für die Darstellung Türen, Fenster, Treppen, Durchbrüche, Nischen. Diese sind zum Teil abhängig vom Maßstab der Zeichnung.

Es existiert noch eine Reihe weiterer Normen, die Einfluss auf Zeichnungen haben. Viele Bereiche des Bauwesens wie der Stahlbau, Straßenbau, etc. haben ihre eigenen, zusätzlichen Anforderungen an die Bauzeichnungen.

19.1.3 Ansichtsarten

In den aktuellen CAD-Systemen haben sich verschiedene Viewarten etabliert, hierzu zählen:

- 2D-View - Grundriss, Schnitte, Ansichten
- Textview
- 3D-View - Animationen

Jede dieser Viewarten (Abb.17.5) benötigt einen eigenen, speziellen Algorithmus für die Darstellung. Diese Darstellungsalgorithmen sind voneinander unabhängig.

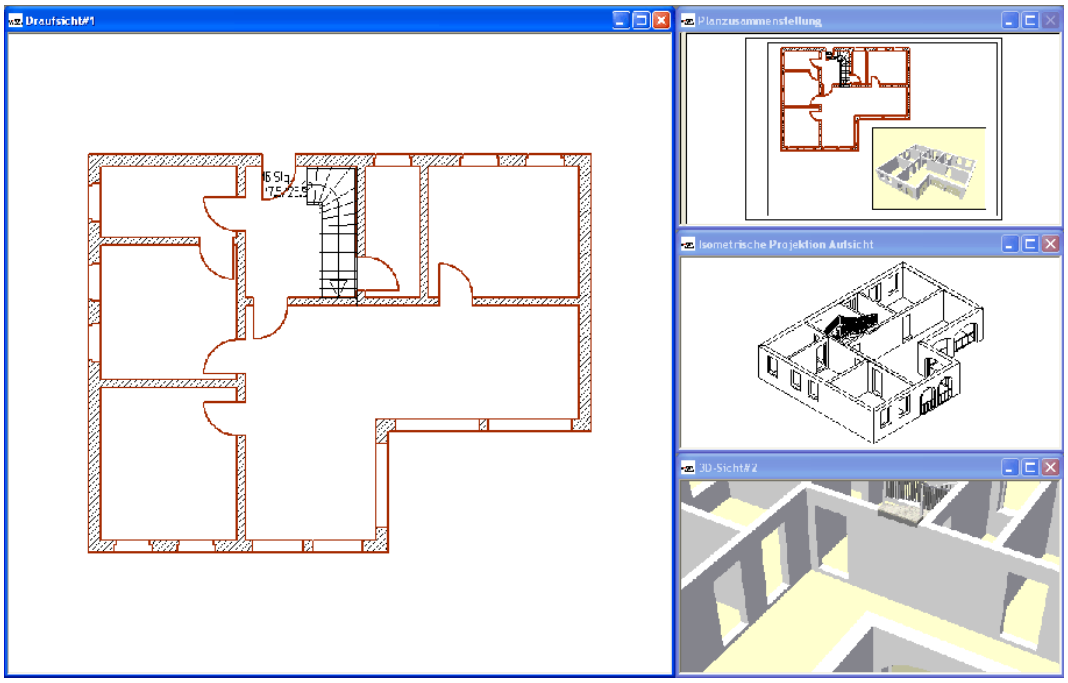


Abbildung 19.1: Ansicht verschiedener Viewarten

Die Algorithmen für die einzelnen Views benötigen individuelle Informationen für die Darstellung. Bei den Textviews stehen bei der Darstellung die Objekteigenschaften im Vordergrund, die Geometrie wird nur indirekt verwendet. Bei den graphischen Views wird die Geometrie des Objektes benötigt, um aus dieser die geforderte Ansicht zu errechnen.

19.1.4 Objekttypen

Der Objekttyp bestimmt auch mit dem Viewtyp den Darstellungsalgorithmus. Meist reicht ein Darstellungsalgorithmus für einen Viewtyp und alle Objekte aus, wenn die Darstellung

aus der Geometrie des Objektes errechnet werden kann. Es gibt auch Bauobjekte, deren Darstellung nicht aus der Geometrie errechnet wird, hierzu zählen Türen und Fenster mit ihrer symbolischen Darstellung in Grundrissen, die in der Detailgenauigkeit auch mit dem Maßstab variieren.

19.1.5 Zweck der Zeichnung

Das Aussehen einer Zeichnung wird stark von ihrem Zweck bestimmt. Im Folgenden werden zwei verschiedene Zwecke näher betrachtet, die ich Architekten- und Ingenieurzeichnungen nenne. Bei den Ingenieurzeichnungen liegt das Augenmerk auf der Ausführung. Diese Zeichnungen sollen alle für die Ausführung des Bauwerkes notwendigen Informationen enthalten, sie müssen eindeutig sein und jeder soll sie verstehen können. Das Hauptaugenmerk bei den Architektenzeichnungen liegt auf einem anderen Schwerpunkt, vielmehr handelt es sich um eine Präsentationszeichnung. Mit diesen Plänen soll der Entwurf des Architekten verkauft werden. Hierbei geht es weniger um die korrekte Darstellung des Objektes, vielmehr sollen die Ideen, die hinter dem Entwurf stehen herausgearbeitet werden, wofür auch von den Darstellungsvorschriften abgewichen wird, wenn es dafür zweckmäßig ist.

19.1.6 Zusammenfassung der Anforderungen

Das Datenmodell muss an die verschiedenen Viewarten „angepasst“ werden können. Die Darstellung folgt im Allgemeinen vorgegebenen Regeln, jedoch muss die Möglichkeit gegeben sein, von den vorgegeben Regeln abzuweichen, um die Zeichnung individuell zu gestalten. Die Detailgenauigkeit der Zeichnung muss auf einfache Art und Weise gesteuert werden können. Besonderheiten von einzelnen Objekttypen müssen beim Zeichnen berücksichtigt werden.

19.2 Die Painter

Um den Ansprüchen an die Darstellung gerecht zu werden, wird die Klasse *CPainter* entwickelt. Bei ihrer Entwicklung wurden verschiedene Entwurfsmuster verwendet.

Der Algorithmus für die Darstellung der einzelnen Objekttypen in einem Viewtyp ist nicht in der Bauobjektklasse implementiert sondern wird in eine eigene Klasse, genannt Painter,

ausgelagert.

Um die Unabhängigkeit der Bauobjekte von den einzelnen Views zu gewährleisten, wurde das Strategiemuster verwendet. Der Darstellungsalgorithmus wurde vom Bauobjekt gelöst und in eine eigene Klasse, der Klasse *CPainter* ausgelagert und bildet den Kern der Klasse. Durch die Auslagerung des Algorithmus werden die Bauobjekte unabhängig von den Views, die Darstellung eines Objektes wird aus den Daten des Bauobjektes durch den Painter errechnet und an die View weitergegeben.

Für jeden Viewtyp existiert mindestens ein Painter der für eine allgemeine Darstellung der Bauobjekte zuständig ist. Weicht die Darstellung eines Bauobjektes von der allgemeinen Darstellung der Objekte in der View ab, so benötigt man für die Bauobjekt-Viewkombination einen eigenen Painter, hierzu gehören z. B. die Darstellung von Türen und Fenster als Symbol in den Grundrissen.

Eine Änderung von Bauobjekten hat keine Auswirkung auf die Views, es muss nur ggf. der Painter an das neue Objekt angepasst werden. Es besteht auch die Möglichkeit die Views zu ändern oder eine neue View einzufügen, ohne dass das Datenmodell geändert werden muss. Durch die Unabhängigkeit der Objekte von den Views ist es nur nötig das Programm um einen neuen viewspezifischen Painter zu ergänzen, der dann die an die View angepasste Darstellung übernimmt. Würde der Darstellungsalgorithmus beim Bauobjekt angesiedelt sein, müssten z. B. bei der Einführung einer neuen View in das Programm alle Objekte für diese View modifiziert werden, damit die Objekte in der neuen View dargestellt werden können. Für alle Objekte, deren Darstellung aus der Geometrie ermittelt werden, ist nur ein Painter pro Viewtyp notwendig.

Für die Abbildung werden neben dem Algorithmus auch Darstellungsparameter wie u. a. Linienstärke, Farbe und Art benötigt. Diese Parameter werden vom Algorithmus benötigt, sind aber auch objekt- bzw. objekttypspezifisch. Eine Ansiedlung der Parameter beim Objekt würde diese mit den Views verknüpfen und die Trennung von Dokument und View aufheben, dies würde auch die schon erläuterten Nachteile, die bei Änderung der Views auftreten, mit sich bringen.

Eine andere Variante ist die Speicherung der Darstellungsparameter beim Painter. Der Painter wird in diesem Fall gemäß dem Fliegengewichtsentwurfsmuster genutzt, der intrinsische Zustand sind die Darstellungsparameter, der extrinsische Zustand die beim Objekt gespeicherte Geometrie.

Die Speicherung der Darstellungsparameter beim Painter ermöglicht die Reduzierung des Datenaufkommens gegenüber der Variante der Speicherung beim Objekt (siehe Abb. 19.2).

Die Auslagerung ermöglicht es verschiedenen Objekten die gleichen Parameter zu verwenden, es müssen nicht für jedes Objekt und jede View die Parameter gespeichert werden.

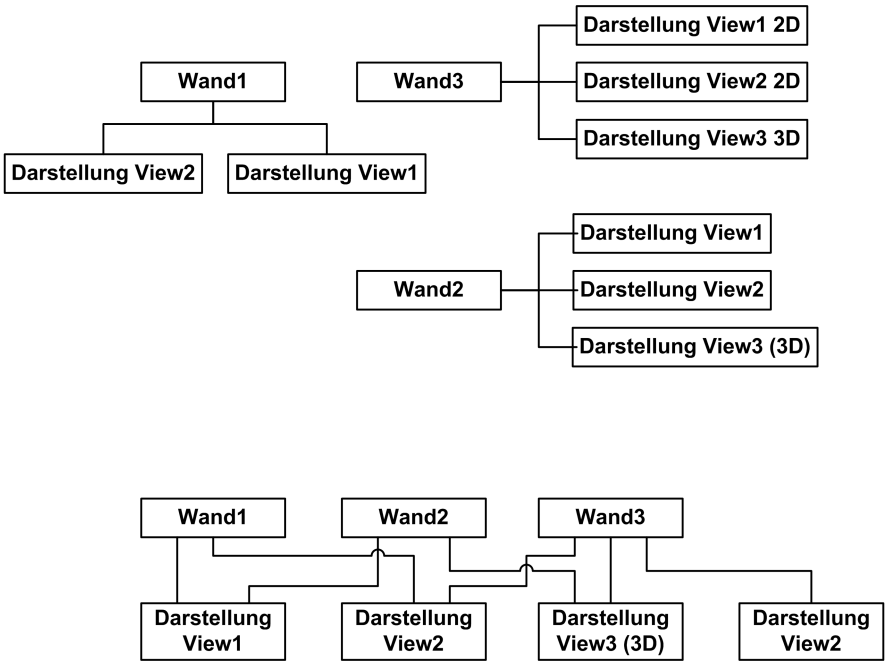


Abbildung 19.2: Darstellung des Painteraufkommens

In der „Grundform“ ist ein Painter zuständig für einen Objekttyp und einen Viewtyp, sodass nur wenige verschiedene Painter benötigt werden. Wird die Darstellung, z. B. die Linienstärke geändert, so wird für die Speicherung der neuen Darstellungsparameter ein neuer Painter benötigt. Für die Erzeugung des neuen Painters dient der alte Painter als Prototyp. Die Painter besitzen entsprechend dem Entwurfsmuster Prototyp eine Funktion für die Erzeugung eines neuen Painterobjektes, wobei der Vorhandene als Schablone dient.

Nach der Erzeugung des neuen Painterobjektes werden die Darstellungsparameter der Vorlage in den neuen Painter kopiert und dieser anschließend unter Angabe der Objekt- und Viewkombination, für welche er zuständig ist, abgespeichert.

20 Das Referenzkonzept

Das Referenzkonzept beruht auf Beobachter-Entwurfsmuster (siehe Kapitel 3.5). Mit dem Referenzkonzept werden Verbindungen zwischen einzelnen Bauteilen erzeugt. Durch die Verknüpfungen reagieren einzelne Bauteile auf Änderungen am Objektmodell. Die Referenzen können auf einzelne Bauteile z. B. Wände und Fenster oder auf Teilen des Geometriemodells wie Flächen, Kanten zeigen.

20.1 Implementierung von Referenzen

20.1.1 Identifizierung der Objekte

Für die Implementierung der Referenzen müssen die Bauteile eindeutig identifizierbar sein. Die Bauteile können auf zwei verschiedene Arten identifiziert werden; die Identifizierung über einen Pointer auf das Bauteil oder über die ID des Objektes.

Bei der Verwendung der Pointer für die Identifizierung der Bauteile kann direkt auf die anderen Objekte zugegriffen werden, was einen Geschwindigkeitsvorteil mit sich bringt. Der Nachteil der Pointer ist, dass diese nicht konstant sind. Beim Schließen des Projektes werden die Objekte zerstört. Beim Einlesen des Projektes aus der Datei, werden die Objekte neu erzeugt, wobei die einzelnen Objekte neue Speicheradressen bekommen. Die neuen Speicheradressen stimmen nicht mit den alten überein, wodurch die Referenzen auf die falschen Objekte oder ins Leere zeigen. Die Referenzen müssen nach dem Einlesen des Datenmodells korrigiert werden.

Bei der Verwendung der Objekt-ID für die Identifizierung der Bauteile entfällt die Korrektur der Referenzen nach dem Einlesen, da die IDs in jedem Projekt eindeutig vergeben werden und auch nach dem Speichern und Einlesen konstant bleiben. Nachteilig wirkt sich bei der Verwendung der IDs aus, dass die Objekte nicht direkt angesprochen werden können, sondern erst anhand der ID aus dem Objektmodell herausgesucht werden müssen,

was die Geschwindigkeit beeinträchtigt.

20.1.2 Speicherung der Relationen

Die Abhängigkeiten zwischen den einzelnen Objekten müssen im Programm gespeichert werden. Dies kann durch die Abspeicherung der Referenzen direkt beim Objekt geschehen. Das Objekt erhält hierfür eine Klasse, die die Referenzen verwaltet und mit virtuellen Funktionen pflegt. Für die Abspeicherung beim Objekt gibt es drei Möglichkeiten:

1. Verweise am untergeordneten Objekt
2. Verweise im übergeordneten Objekt
3. Gegenseitige Verweise

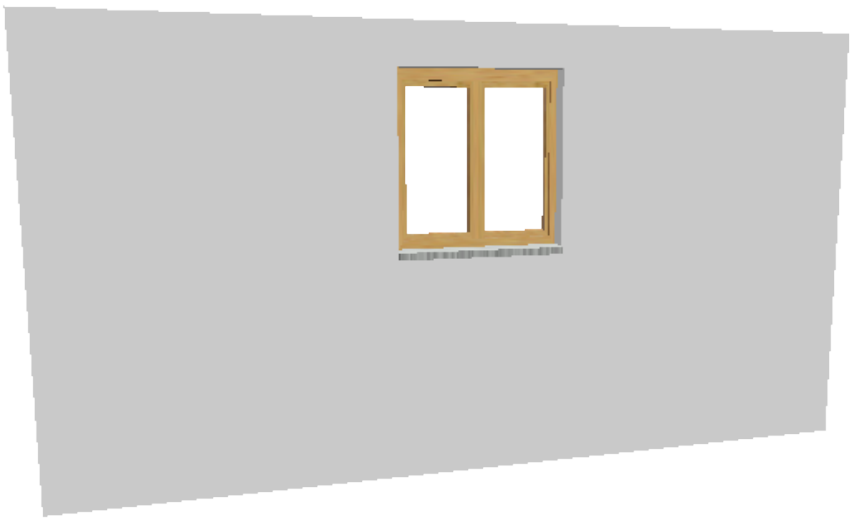


Abbildung 20.1: Wand mit Fenster

Abb. 20.1 zeigt eine Wand mit einem Fenster. Das Fenster ist ein der Wand untergeordnetes Bauteil. Bei der Referenzierung 1 wird beim Fenster gespeichert, dass es ein Bestandteil der Wand ist. Bei der Art 2 wird bei der Wand gespeichert, dass sie das Fenster enthält. Die beiden Methoden haben den Nachteil, dass die Verweise einseitig sind. Bei 1 kann das Fenster die Wand über Änderungen informieren, bei 2 kann die Wand das Fenster informieren. Informationen über Veränderungen in der jeweils anderen Richtung können nur mit großem Aufwand weitergegeben werden. Um Informationen in umgekehrter Richtung weiterzugeben, muss das gesamte Datenmodell durchsucht und die Referenzen aller Bauteile überprüft werden. Dieser Vorgang muss nach jeder Änderung beim Datenmodell

erneut vorgenommen werden, was einen hohen Zeitaufwand bedeutet. Bei der dritten Art werden die Referenzen beim über- und untergeordneten Objekt gleichzeitig abgespeichert, dies bedeutet den doppelten Speicherbedarf für die Referenzen gegenüber 1 und 2. Der Vorteil ist, dass das Durchsuchen des Datenmodells entfällt, da die Referenzen gleich zur Verfügung stehen.

20.2 Aufbau der Relationen

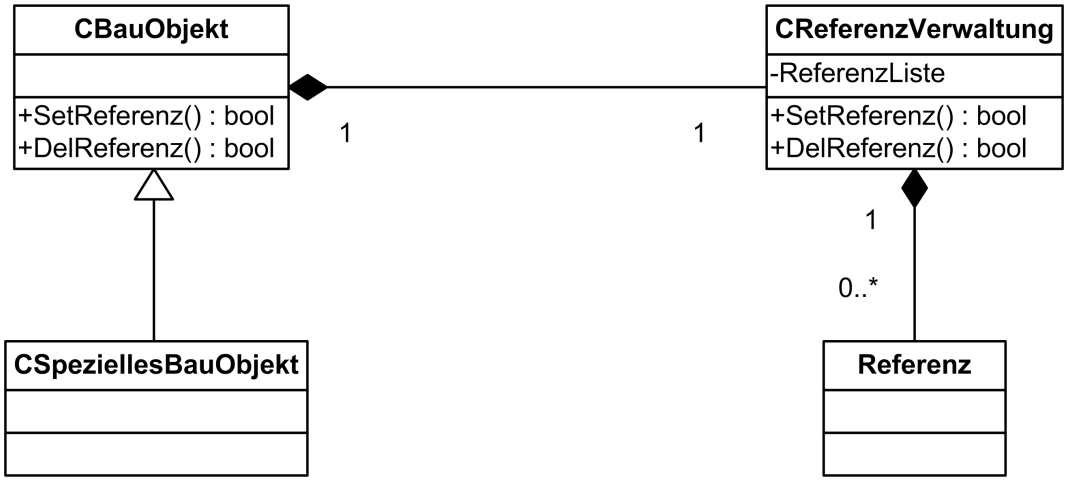


Abbildung 20.2: Klassendiagramm eines Bauobjektes mit Referenz

Das Klassendiagramm eines Bauobjektes mit der Verwaltung der Referenzen zeigt Abb. 20.2. Die einzelnen Bauobjekte erben von der Basisklasse *CBauobjekt* die Funktionalität zur Verwaltung der Referenzen. Die Basisklasse besitzt ein Objekt der Klasse *CReferenzverwaltung*, welches für die Verwaltung der Referenzen zuständig ist.

Die Klasse *CReferenzverwaltung* beinhaltet die Referenzliste für die Speicherung der Referenzen. Die Referenzen werden beim Abspeichern des Datenmodells mit abgespeichert. Des Weiteren beinhaltet die Klasse die Methoden zum Einfügen und Entfernen von Referenzen aus der Referenzliste. Die Anfragen an das Bauobjekt werden direkt an die Referenzverwaltung weitergeleitet.

Besteht zwischen zwei Objekten A und B eine Beziehung, so ruft Objekt A die Funktion `SetReferenz(A)` des Objektes B auf. Die Funktion `SetReferenz()` überprüft, ob schon eine Referenz auf das Objekt vorhanden ist und stellt so sicher, dass jedes Objekt nur einmal referenziert wird. Ist das Objekt schon vorhanden oder kann die Referenz nicht in die Liste eingefügt werden, so gibt die Funktion ein False an Objekt A zurück. Wenn ein False zurückgegeben wird, fügt das Objekt A die Referenz auf das Objekt B nicht in die

eigene Referenzliste ein und stellt so sicher dass keine einseitigen Referenzen vorhanden sind.

20.3 Aktualisierung des Systems

Eine Änderung des Basisobjektes löst das Ereignis „Update aller abhängigen Objekte“ aus.

```
pBase->PostUpdateDependingObjects()
```

Die abhängigen Objekte aktualisieren sich daraufhin über eine, von jeder abgeleiteten Klasse zu überschreibenden, Funktion

```
pDepObj⇒PostUpdateMasterObject(pMaster...)
```

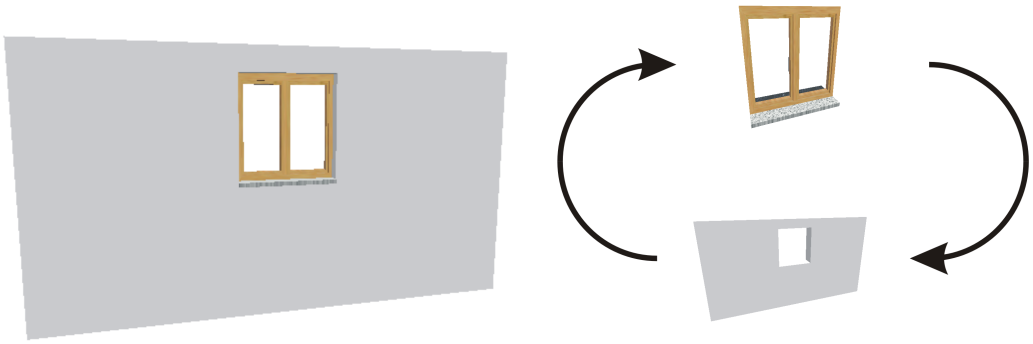


Abbildung 20.3: Updatezyklus infolge Änderung am Modell

Die Funktion `PostUpdate()` kann als SendMessage im Sinn von Windows implementiert werden, das bedeutet, dass die Ausführung des Updates sofort erfolgt. Da keine Instanz existiert, die die Updates sammelt, kann es zu mehrfachen Updates und Updatezyklen kommen. Eine Änderung der Wand 20.3 bewirkt ein Update an dem dazugehörigen Fenster. Das Fenster reagiert auf das Update mit einem SentUpdate an die Wand, der Updatezyklus ist geschlossen. Durch die Mehrfachupdates und die Update-Zyklen erweist sich das Update in Form einer SendMessage für diese Anwendung als unbrauchbar. Als besser erweist sich eine SendUpdate-Funktion in Form einer PostMessage. Im Gegensatz zu den SendMessage-Funktionen werden die PostMessage-Funktionen nicht direkt ausgeführt, sondern in einer Liste gesammelt. Ist die Sammlung der PostMessage-Nachrichten abgeschlossen, werden die in der Liste gesammelten Updates ausgeführt.

Die Klasse *CS_BuilderXXX* meldet vor dem Senden des „PostUpdates“ eine Transaktion beim Dokument an. Alle darauffolgenden „PostUpdates“ werden in der Transaktion in einem Graphen gesammelt. Nach Beendigung der Transaktion wird der weitere Aufbau und die Abarbeitung des Graphen gestartet.

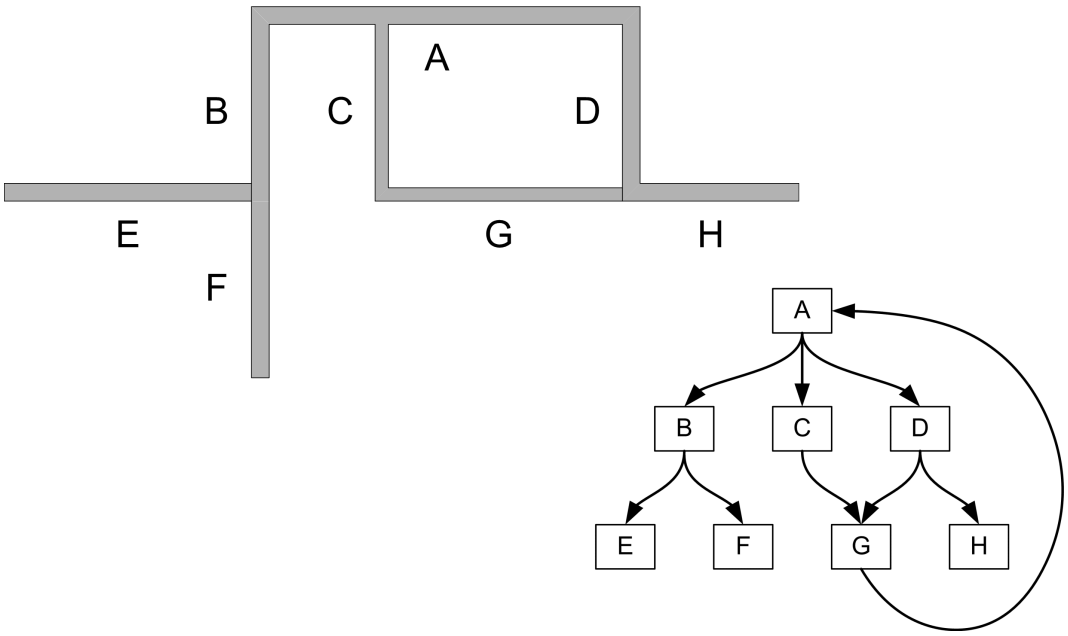


Abbildung 20.4: Postupdatezyklus infolge Änderung am Modell

In Bild 20.4 ist ein Ausschnitt aus einem Grundriss dargestellt. Nach einer Änderung der Wand A wird der Baum (rechts in der Abbildung) aufgebaut, indem jedes Objekt nur einmal vorhanden ist. Nach Beendigung des Aufbaues wird der Baum durchlaufen und für jedes Objekt die Updatefunktion aufgerufen und die Objekte aktualisieren sich.

Teil IV

Implementierung in GoCAD

21 Implementierte Programmarchitektur

21.1 Allgemeine Programmarchitektur

GoCAD besitzt eine mehrschichtige Programmarchitektur. Der Kern von GoCAD beruht auf dem von der MFC bereitgestellten Document-View-Konzept (DV-Konzept). Hierbei handelt es um eine Vereinfachung des Architekturmusters Model-View-Controller (vgl. Kapitel 17 und 4), der Controller ist hierbei kein eigenständiges Modul, sondern wurde mit der View kombiniert.

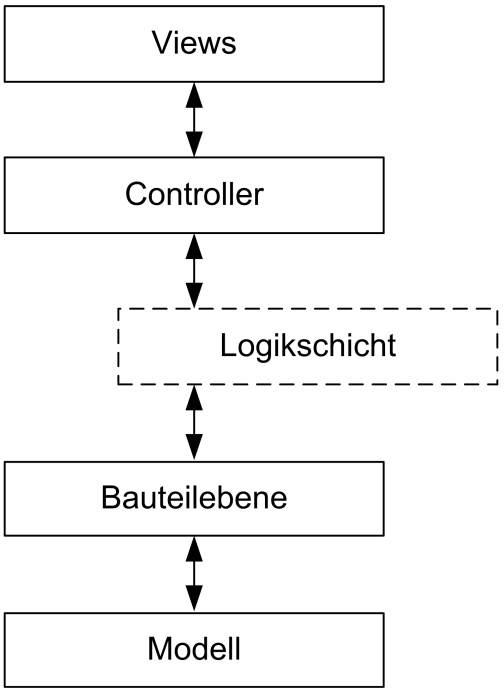


Abbildung 21.1: Schichtmodell von GoCAD

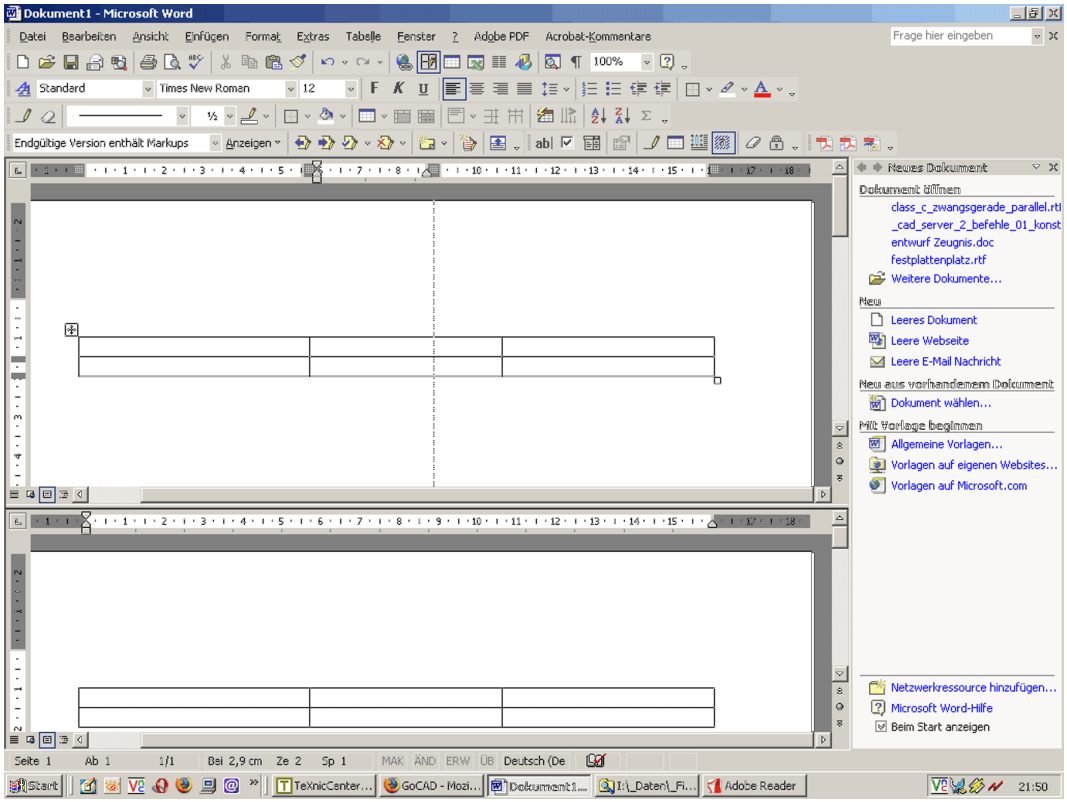


Abbildung 21.2: Multiview bei Word

21.1.1 Verwendung des Document View Konzeptes

Durch die Kopplung von View und Kontroller ist das Document-View Konzept nur bedingt verwendbar für CAD-Programme, es ist ausgelegt für Microsofteigene Programme wie Word und Excel. Die Eingaben in Word erfolgen entweder über die Tastatur, Buchstaben und Zahlen oder Maustastenclick, Mousebuttondown bzw. -up, ein Beispiel ist in Abb. 21.2 dargestellt. Dieses zeigt ein Word-Dokument, das in zwei verschiedenen Sichten dargestellt wird. Im oberen Teil der Abbildung wird die Größe der zweiten und dritten Spalte geändert. Die Konstruktion ist nur in der oberen Hälfte sichtbar, die zweite Ansicht bleibt unverändert. Nach Beendigung der Konstruktion (Maustaste hoch) wird die Änderung an das Dokument weitergegeben, anschließend die Nachricht UpdateView gesendet, was eine Aktualisierung der Ansichten auslöst. Ein simultanes Arbeiten in den verschiedenen Views ist bei Word nicht vorgesehen, da dies für Officeprogramme nicht von Nutzen ist.

Moderne CAD-Programme verfügen über die Möglichkeit des Wechsels der Views während der Eingabe. Dies erforderte eine Modifikation des von der MFC gelieferten DV-Konzepts, da bei diesem der Kontroller in die View integriert ist. Hierdurch ist ein Wechsel zwischen

den Views nicht möglich, da die eingegebenen Daten lokal in der View gespeichert werden und bei einem Wechsel zwischen den Views nicht weitergereicht werden.

Das Document-View-Konzept dient deshalb nur als Basis, welche weiter ausgebaut wird. Es stellt die erste und fünfte Schicht des Aufbaus von GoCAD.

21.1.2 Die weiteren Schichten in GoCAD

Durch die Kopplung des Kontrollers mit der View im DV-Konzept ist dieses Konzept nur bedingt verwendbar, es wurde deshalb eine zusätzliche Steuerebene in GoCAD integriert. Diese Steuerebene ist Bestandteil der Klasse *CControlCenter*, welche in der Klasse *CMainFrame* angesiedelt ist. Sie ist die vierte Ebene von GoCAD.

Die zweite Ebene von GoCAD bildet die Bauteilebene, sie stellt die Objekte für die Anwendung zur Verfügung. Die dritte Schicht, die spezifische Anwendungsschicht (Architektur, Ingenieurbau etc.) ist in GoCAD nicht implementiert.

21.2 Implementierung des Dokumentes in GoCAD

Das Dokument beruht auf der Klasse *CDocument*, die von der MFC bereitgestellt wird. Die Klasse *CGoCADDoc* wurde public von *CDocument* abgeleitet und hat von dieser Klasse die Grundfunktionalität geerbt.

Neben der Bauobjektliste existiert noch eine weitere Liste, die BauObjektAuswahlListe. Diese enthält Pointer auf die vom Anwender ausgewählten Objekte, welche in den Views markiert angezeigt werden.

Die Methoden der Klasse *CGoCADDoc* lassen sich in die folgenden Bereiche einteilen:

- Funktionalität der Klasse
- Bauobjekte
- Darstellung der Objekte
- Auswahl der Objekte
- Sonstige Funktionen

Die Bauobjekte werden nach der Erzeugung in der BauObjekteListe (CBauObjektListe) gespeichert. Nach dem Einfügen in die Bauobjektliste bekommt das Bauobjekt eine programmeindeutige ID zugewiesen, die zur Identifizierung der Objekte dient. Die ID für die

Bauobjekte wird in der Variablen m_iAktuelleNummer gespeichert. Nach dem Einfügen eines Objektes wird die Variable um eins nach oben gezählt. Die Variable ist so implementiert, dass sie nicht reduziert werden kann. Beim Speichern des Modells wird die Nummer mit gespeichert, so dass nach dem Laden, ausgehend von der alten Anzahl, weitergezählt werden kann.

21.2.1 Darstellung der Objekte in GoCAD

Für die Darstellung der Objekte werden die Painter benötigt. Die Painter erzeugen aus dem Datenmodell die Objekte für die viewspezifische Darstellung. Die Objekte für die Darstellung werden in der View gespeichert, so dass die Darstellung nicht jedes Mal neu berechnet werden muss, wenn die View neu gezeichnet wird.

Die Darstellung der Objekte wird vom Dokument gesteuert, da nur dieses die verschiedenen Views kennt. Die Methode `DisplaylistenAktualisieren(CBauObjekt *pBauObjekt)` wird aufgerufen, nachdem ein neues Objekt in das Dokument eingefügt wurde. Die Funktion geht über alle Views und sucht für das Objekt den passenden Painter für die Darstellung. Die Methode `DisplaylistenAktualisieren()` arbeitet auf die gleiche Weise, es wird jedoch nicht ein spezielles Objekt sondern alle Objekte dargestellt.

Die Methode `DisplaylistenEntfernen(CBauObjekt *pBauObjekt)` wird verwendet, um die Darstellung eines Objektes aus den Views zu entfernen. Sie ruft die Methode `Entfernen(*CBauObjekt)` jeder View auf. Die Methode sucht anhand der Speicheradresse des Objektes alle zum Objekt gehörenden Darstellungsobjekte und löscht diese aus den Displaylisten.

Klasse CGoCADDoc	
Öffentliche Methoden	***Beschreibung***
virtual ~CGoCADDoc()	Destruktor
virtual BOOL OnNewDocument()	Erzeugen eines neuen Dokumentes
virtual BOOL OnOpenDocument(LPCTSTR lpszPathName)	Öffnen eines Dokumentes
virtual void Serialize(CArchive &ar)	Speichern und Laden des Dokumentes
void OnSave()	Zusatzfunktion beim Speichern
void BO_einfuegen(CBauObjekt *pBO)	Bauobjekt in Dokument einfügen
CBauObjektListe *GetBOL()	Gibt Zeiger auf Objektliste zurück
void ShowObjektEigenschaften()	Abfragen von Objekteigenschaften
void Trans(CMatrix *pmat_Transformation)	Transformieren von Objekten
void BoundBox()	Ermittlung der BoundBox
CBauAuswahlListe *GetBOAL()	Liste der ausgewählten Objekte
void AuswahlListeLeeren()	Leeren der Auswahlliste
int AnzAusgewaehlteElemente()	Anzahl der ausgewählten Objekte
void BoundingBoxAuswahl()	Boundingbox um Auswahl
void AuswahlLoeschen()	Löscht ausgewählte Objekte

Fortsetzung Klasse CGoCADDoc	
void markieren(CBauAuswahlListe *pAusObjekte, int Auswahl)	Markiert ausgewählte Objekte
void DisplaylisteView(CGoCADView *pView)	Baut Displaylisten auf
void DisplaylistenAktualisieren()	Aktualisiert Displayliste
void DisplaylistenAktualisieren (CBauObjekt *pBauObjekt)	Aktualisiert bestimmtes Objekt in den Displaylisten
void DisplaylistenEntfernen(CBauObjekt *pBauObjekt)	Löscht Objekt aus den Displaylisten
void DisplaylistenLoeschen()	Löscht alle Displaylisten
void DisplaylistenAufraeumen()	Entfernt NULL_Zeiger aus den Displaylisten
void KSUeberzeichnen()	Neuzeichnen des Koordinatensystems
void KSZeichnen(bool invers)	Zeichnen des Koordinatensystems
void KSsetzen (CVektorf US, CVektor2D RI)	Position des Koordinatensystems setzen
void RiSetzen (CVektor2D RI)	Richtung des Koordinatensystems setzen
void USsetzen (CVektor3D US)	Ursprung des Koordinatensystems setzen
CString AddViewAnzahl()	Setzen der Anzahl der Sichten
void verschneiden()	Verschneiden von Objekten
CVektor3D GetSetzpunkt()	Abfrangen des Setzpunktes
void Trans (CMatrix *pmat_Transformation)	Transformieren von Objekten
Geschützte Methoden	***Beschreibung***
CGoCADDoc()	Konstruktor
afx_msg void OnFileSave()	Funktion für Nachricht Speichern
afx_msg void OnFileOpen()	Funktion für Nachricht öffnen
Öffentliche Attribute	***Beschreibung***
int m_iBefehl	Variable für den aktuellen Befehl
Geschützte Attribute	***Beschreibung***
int ViewAnzahl	Anzahl der zu dem Dokument gehörenden Views, wird durchgezählt
Private Attribute	***Beschreibung***
CBauObjektListe BauObjektListe	Liste aller Bauobjekte
CBauAuswahlListe BauObjektAuswahlListe	Liste der markierten Bauobjekte
int m_iAktuelleNummer	Anzahl der eingegebenen Bauobjekte
CBauAuswahlListe BauAuswahlListe	Liste der markierten Bauobjekte

Tabelle 21.1: Methoden und Attribute der Klasse CGoCADDoc

21.3 Implementierte Views in GoCAD

Für GoCAD wurden verschiedene, in CAD-Programmen standardmäßig vorkommende Views entwickelt. Die implementierten Views sind:

2D-View stellt Schnitte durch das Datenmodell dar. Die Schnitte können in jeder beliebigen Ebene verlaufen.

3D-View ist eine 3D-Ansicht auf Basis von OpenGL zur Visualisierung des Datenmodells

Text-View ist eine Listenansicht der verwendeten Objekte.

Die Klassenhierarchie der Viewklassen ist in Abb. 21.3 dargestellt. Die Basisklasse der Views ist die Klasse *ICViewEigenschaften*, sie ist eine virtuelle Basisklasse, von ihr ist die

Klasse *CGoCADView* abgeleitet, die die Basis der drei implementierten Viewarten ist.

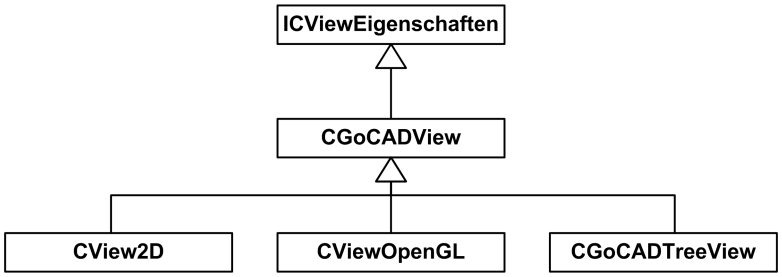


Abbildung 21.3: Viewklassen von GoCAD

21.3.1 Virtuelle Basisklasse ICViewEigenschaften

Die Klasse *ICViewEigenschaften* ist die virtuelle Basisklasse aller Views. Es existiert von dieser Klasse kein Objekt, sie ist eine Schnittstellenklasse für die Views. Sie stellt die Funktionen zum Berechnen von Koordinaten und Abfragen bzw. Setzen der Vieweigenschaften zur Verfügung. Die Implementierung der Methode erfolgt in den abgeleiteten Klassen.

Klasse ICViewEigenschaften	
Öffentliche Methoden	***Beschreibung***
virtual CGraphListe *GetDisplayListe ()=0	Gibt Zeiger auf Displayliste zurück
virtual CVektor2D Koord3Din2D (const CVektor3D &GeoPunkt)=0	Koordinaten umrechnen 3D in 2D
virtual CVektor3D Koord2Din3D (const CVektor2D &GeoPunkt)=0	Koordinaten umrechnen 2D in 3D
virtual CVektor3D WeltZuAnsichtsKoordinaten(const CVektor3D &v) const =0	Koordinaten umrechnen Welt in Bildschirm
virtual CVektor2D AnsichtsZuBildschirmKoordinaten(const CVektor3D &v) const =0	Koordinaten umrechnen Bildschirm in Welt
virtual void SetzeKamera const CVektor3D &kameraposition, const CVektor3D &objektposition, const CVektor3D &orientierung)=0	Setzen des Sichtursprungs
virtual void HoleKamera (CVektor3D &kameraposition, CVektor3D &objektposition, CVektor3D &orientierung) const =0	Abfragen des Sichtursprungs
virtual void SetViewTyp (ViewType viewtyp)=0	Setzen des Sichttypes
virtual ViewType GetViewTyp() const =0	Abfragen des Sichttypes
virtual void SetMasstab(double masstab)=0	Setzen des Zeichnungsmaßstabes
virtual double GetMasstab()=0	Abfragen des Zeichnungsmaßstabes
virtual void SetUmrechnungsfaktor(double faktor)=0	Setzen des Umrechungfaktors
virtual double GetUmrechnungsfaktor()=0	Abfragen des Umrechnungsfaktors
virtual void SetFangRadius(double fangradius)=0	Setzen des Fangradius
virtual double GetFangRadius()=0	Abfragen des Fangradius
virtual void SetBoundingBox(CBoundingBox *boundingbox)=0	Setzen der Boundingbox
virtual CBoundingBox *GetBoundingBox()=0	Abfragen der Boundingbox

Fortsetzung Klasse ICViewEigenschaften	
virtual void SetMittelpunkt(const CVektor2D &mittelpunkt)=0	Setzen des Zeichnungsmittelpunktes
virtual const CVektor2D &GetMittelpunkt() const =0	Abfragen des Zeichnungsmittelpunktes
virtual void SetAusschnitt(CDRechteck *ausschnitt)=0	Setzen des Zeichnungsausschnittes
virtual CDRechteck *GetAusschnit()=0	Abfragen des Zeichnungsausschnittes
virtual ClokKS *GetLokalesKoordinatenSystem()=0	Abfragen des lokalen Koordinatensystems
virtual CGeoEbene * GetViewEbene()=0	Abfragen der Sichtebene
virtual const AnsichtsRaum *HoleAnsichtsRaum() const =0	Abfragen des Ansichtshauses

Tabelle 21.2: Methoden und Attribute der Klasse ICViewEigenschaften

21.3.2 Die Klasse CGoCADView

Die Klasse *CGoCADView* ist von der virtuellen Basisklasse *ICViewEigenschaften* abgeleitet und ist die Basisklasse für alle Viewklassen. Von der Klasse *CGoCADView* existiert, wie von ihrer Basisklasse, auch keine konkrete Instanz in der Anwendung. Der Hauptzweck von *CGoCADView* ist die Implementierung aller Methoden und Attribute, welche die abgeleiteten Klassen gemeinsam besitzen. Dies sind die Funktionen zur Painterverwaltung, Positionsbestimmung, Objektauswahl, Objektdarstellung und die Funktionen zum Setzen und Abfragen der Eigenschaften. Die wichtigsten Attribute, welche die Klasse einführt sind die Displayliste, welche die aus dem Objektmodell berechneten graphischen Primitiven zur Darstellung enthält und die Painterliste mit den viewspezifischen Paintern.

Klasse CGoCADView	
Öffentliche Methoden	***Beschreibung***
virtual void XMLSave(ofstream &datei)	Speichern der viewspezifischen Daten
virtual void XMLLoad(fstream &datei)	Laden der viewspezifischen Daten
virtual void NaechsteGraphObjekte(CVektor3D *pPosition, CGraphListe *pListe)	Suche alle Objekte im Fangradius
virtual CGraphObjekt *NaechstesGraphObjekt(CVektor3D *pPosition)	Suche des am nächsten gelegenen Objektes
void PainterZurueckssetzenOTV(CBauObjekt *pBauObjekt)	Entfernen des objekttypspezifischen Painters
void PainterZurueckssetzenOB(CBauObjekt *pBauObjekt)	Entfernen des objektspezifischen Painters
void SetPainterObjektSpezifisch(int ID, CPainter *pPainter)	Setzen des objektspezifischen Painters
void SetPainterObjektTypSpezifisch(CString Objektklasse, CPainter *pPainter)	Setzen des objekttypspezifischen Painters
CPainter *GetPainterObjektspezifisch(CBauObjekt *pBauObjekt)	Abfragen des objektspezifischen Painters
CPainter *GetPainterObjekttypspezifisch(CBauObjekt *pBauObjekt)	Abfragen des objekttypspezifischen Painters
CVektor2D DKtoWK(CPoint point)	Umrechnung von Weltkoordinaten in Viewkoordinaten
CGEO_Punkt Position3D(CDPunkt Position2D)	Funktion zur Umrechnung der 2D in 3D Koordinaten
virtual CVektor3D WeltZuAnsichtsKoordinaten(const CVektor3D &v) const	Umrechnung Welt zu Ansichtskoordinaten
virtual CVektor2D AnsichtsZuBildschirmKoordinaten(const CVektor3D &v) const	Umrechnung Ansicht zu Bildschirmkoordinaten

Fortsetzung Klasse CGoCADView	
CDPunkt DKtoWK(CPoint point)	Umrechnung von Weltkoordinaten in Viewkoordinaten
virtual void KoordsysSetzen(CDC *pDC)	Setzt den Ursprung des Koordinatensystems der View
virtual ClokKS *GetLokalesKoordinatenSystem()	Abfragen des lokalen Koordinatensystems
virtual void HoleKamera(CVektor3D &kameraposition, CVektor3D &objektposition, CVektor3D &orientierung) const	Abfragen der Kameraposition
virtual void SetzeKamera(const CVektor3D &kameraposition, const CVektor3D &objektposition, const CVektor3D &orientierung)	Setzen der Kameraposition
virtual void RegenList()	Benötigt für OpenGL View
virtual CGraphListe *GetDisplayListe()	Abfragen der Displayliste
virtual void Entfernen(CBauObjekt *BauObjekt)	Entfernt Bauobjekt aus Viewdarstellung
void AuswahlListeAuswahl (CVektor2D Position)	Sucht am nächsten liegendes Objekt
void BereichAuswaehlen (CAuswahlbox Box)	Sucht Objekte in übergebenen Bereich
void OnPrint()	Funktion für Nachricht Print
afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)	Funktion für Nachricht OnChar
double Fangradius()	Umrechnung des Fangradius von Bildschirm Koordinaten in Weltkoordinaten
void Maßstab()	Ermittlung des Maßstabes
virtual void SetViewTyp(ViewType viewtyp)	Setzen des Viewtypes
virtual ViewType GetViewTyp() const	Abfragen des Viewtypes
virtual void SetMasstab(double masstab)	Setzen des Zeichnungsmaßstabes
virtual double GetMasstab()	Abfragen des Zeichnungsmaßstabes
virtual void SetUmrechnungsfaktor(double faktor)	Setzen des Umrechnungsfaktors
virtual double GetUmrechnungsfaktor()	Abfragen des Umrechnungsfaktors
virtual void SetFangRadius(double fangradius)	Setzen des Fangradius
virtual double GetFangRadius ()	Abfragen des Fangradius
virtual void SetBoundingBox(CBoundingBox *boundingbox)	Setzen der Boundingbox
virtual CBoundingBox *GetBoundingBox()	Abfragen der Boundingbox
virtual void SetAusschnitt (CDRechteck *ausschnitt)	Setzen des Zeichnungsausschnittes
virtual CDRechteck *GetAusschnit()	Abfragen des Zeichnungsausschnittes
virtual void SetMittelpunkt(const CVektor2D &mittelpunkt)	Setzen des Zeichnungsmittelpunktes
virtual const CVektor2D &GetMittelpunkt() const	Abfragen des Zeichnungsmittelpunktes
void SetzeAnsichtsRaum(const CVektor3D &v1, const CVektor3D &v2, const CVektor3D &v3)	Setzen des Sichtraums
virtual const AnsichtsRaum *HoleAnsichtsRaum() const	Abfragen des Sichtraums
int Typ()	Funktion gibt den Typ der View zurück
void ZeigerZustand(bool flag)	Abfrage des Zeigerzustandes
void ZeigerPos(CVektor3D Position)	Setzen der Zeigerposition
virtual CGeoEbene *GetViewEbene()	Abfrage der Sichtebene
virtual void ErzeugeViewEbene()	Erzeugen der Sichtebene
void BoundingBoxAuswahlreset()	Auswahlbox zurücksetzen
void BoundingBoxAuswahl(CBauObjekt *pBauObjekt)	Ermittlung einer Boundingbox um die ausgewählten Elemente
virtual void BoundBox()	Ermittlung der Bounding Box
Geschützte Methoden	***Beschreibung***
CGoCADView()	Konstruktor
virtual ~CGoCADView()	Destruktor
virtual void OnDraw(CDC *pDC)	Funktion für Nachricht OnDraw
virtual BOOL OnPreparePrinting(CPrintInfo *pInfo)	Funktion für Nachricht OnPreparePrinting
Private Methoden	***Beschreibung***
void ErzeugeAnsichtsRaum(const CVektor3D &v1, const CVektor3D &v2, const CVektor3D &v3)	Erzeugen des Ansichtraumes

Fortsetzung Klasse CGoCADView	
void ZerstoereAnsichtsRaum()	Zerstören des Ansichtraumes
void BerechneModelViewMatrix()	Berechnung der Umrechnungsmatrix
void BerechneTransformationsMatrix()	Berechnung der Transformationsmatrix
Öffentliche Attribute	***Beschreibung***
CBauAuswahlListe AuswahlListe	Liste der ausgewählten Objekte
int Art	Variable für die Art der View
double m_dUmrechnungsfaktor	Variable für den Umrechnungsfaktor für den Maßstab
double m_iMassstab	Maßstab der View
double Umrechnungsfaktor	Umrechnungsfaktor für den Maßstab
CVektor2D m_Mittelpunkt2D	Koordinate Viewmittelpunkt
ClokKS m_lokKS	Lokales Koordinatensystem
CZeiger Zeiger	Instanz des Zeigers
CDRechteck m_drAusschnitt	Ausschnitt der angezeigt wird
CBoundingBox m_BoundingBox	BoundingBox der View
CBoundingBox m_BoundBoxAuswahl	Boundingbox um die Ausgewählten Objekte
Geschützte Attribute	***Beschreibung***
CGraphListe m_DisplayListe	Liste mit den darzustellenden graphischen Objekten
CObjektViewPainterListe m_ObjektspezifischePainterliste	Liste für objektspezifische Painter
CObjekttypViewPainterListe m_ObjekttypspezifischePainterliste	Liste für objekttypspezifische Painter
AnsichtsRaum m_Ansichtsraum	Variable für Ansichtsraum
CReduktionsmatrix Reduktionsmatrix	Matrix zur Umrechnung der 3D-Koordinaten auf die View
CVektor2D MausPos	Position der Maus
bool MausPosNeu	Flag für geänderte Mausposition
bool m_bCursor	Variable für Cursorzustand
bool MausUeberObjekt	True wenn Maus über Objekt
ViewType m_ViewTyp	Typ des aktuellen Views
double m_dFangRadius	Fangradius für Objekte unter der Maus
CGeoEbene m_Ebene	Ebene der View
CRect m_crFenster	Variable für die Fenstergröße
CVektor2D m_Mausrunter	Koordinate Maus gedrückt
CVektor2D m_Mausrauf	Koordinate Maus losgelassen
HCURSOR lhCursor1	Cursor Typ 1
HCURSOR lhCursor2	Cursor typ 2
HCURSOR lhCursor_Zoom	Zoomcursor
CMatrix4x4 m_ModelView	Variable für Umrechnungsmatrix
CMatrix4x4 m_Projection	Variable für Projektionsmatrix
CMatrix4x4 m_Transform	Variable für Transformationsmatrix
CMatrix4x4 m_InvModelView	Variable für Umrechnungsmatrix
CVektor4D m_VRP	Vektor für Raumkoordinaten
CVektor4D m_VFP	Vektor für Raumkoordinaten
CVektor4D m_VUP	Vektor für Raumkoordinaten

Tabelle 21.3: Methoden und Attribute der Klasse CGoCADView

21.3.3 Implementierung 2D-View

Die 2D-View ist die typische View eines CAD-Programms. In GoCAD wurden verschiedene 2D-Views implementiert. Über die Toolbar kann ausgewählt werden, welche der drei

folgenden Views erzeugt wird:

- x-y Ansicht, Blick in z-Richtung
- x-z Ansicht, Blick in y-Richtung
- y-z Ansicht, Blick in x-Richtung

Über einen weiteren Button in der Toolbar kann ein Dialog zur Erzeugung verschiedener Sichten aufgerufen werden. Dieser Dialog bietet die Möglichkeit, zusätzlich die Lage der Schnittebene in den Sichten anzugeben.

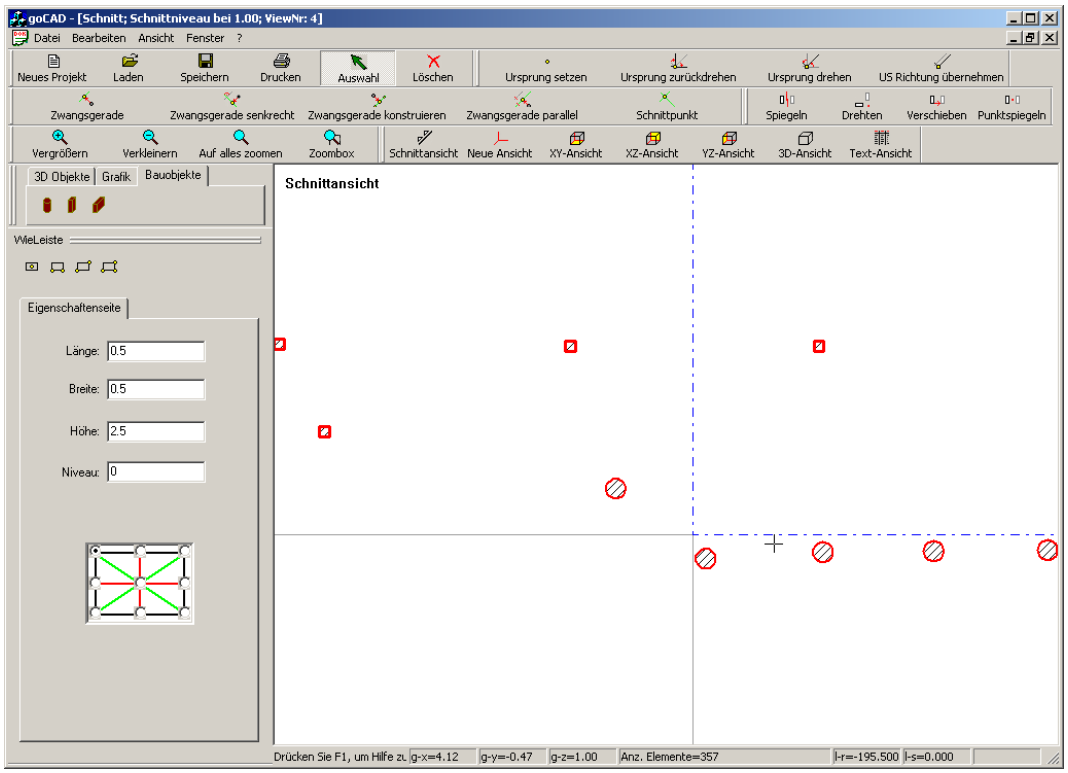


Abbildung 21.4: 2D-View von GoCAD

Die dritte Möglichkeit, die GoCAD zur Erzeugung einer Sicht bietet, ist ein Ausschnitt aus einer bestehenden View. Hierfür werden mit der Maus drei Punkte eingegeben, die ein Rechteck bilden. Die beiden ersten Punkte bilden die vordere Schnittebene des Ausschnittes, der dritte Punkt gibt die Sichtrichtung und die Lage der hinteren Schnittebene an.

Der für die Schnittdarstellung notwendige Painter wurde im Rahmen einer Diplomarbeit (Sci06) entwickelt. Dieser ermittelt die im Ansichtsraum liegenden Objekte und berechnet die Projektion der Objekte im Ansichtsraum. Bei Objekten, die auf den Grenzen des Ansichtsraumes liegen, werden die Schnittflächen berechnet und entsprechend dargestellt. Die wichtigsten Attribute der 2D-View sind:

- Schnittniveau
- vordere Schnittebene
- hintere Schnittebene
- Matrizen zur Ansichtsberechnung

Beim Beenden des Programms werden die Attribute der View mit abgespeichert, damit sie beim Laden des Projektes wieder erzeugt werden können.

Die Klasse *CView2D* überschreibt einige Methoden der Basisklasse *CGoCADView*. Die Methoden, welche überschrieben werden, sind viewspezifisch, hierzu gehören Methoden zum Laden und Speichern der Viewattribute und die Methoden zur Berechnung der Koordinaten.

Neu zur Viewklasse hinzugekommen sind die Methoden, die die View benötigt um auf die Eingaben der Tastatur und Maus zu reagieren. Die Methoden, die auf die Maus reagieren ermitteln im Allgemeinen die Position der Maus im 3D-Raum und geben diese an den aktiven Befehl zur Weiterverarbeitung.

Klasse CView2D	
Öffentliche Methoden	***Beschreibung***
CView2D()	Konstruktor
virtual ~CView2D()	Destruktor
virtual void OnInitialUpdate()	Funktion Nachricht OnInitialUpdate
CGoCADDoc *GetDocument()	Erzeugt Zeiger auf Dokument
virtual void XMLLoad(fstream &datei)	Speicherfunktion
virtual void XMLSave(ofstream &datei)	Ladefunktion
virtual void ErzeugeViewEbene()	Erzeugt die Viewebene
void Statusleiste()	Einträge in die Statusleiste
void Paint(CBauObjekt *BauObjekt)	Erzeugen der Viewdarstellung
virtual CVektor2D Koord3Din2D(const CVektor3D &GeoPunkt)	Umrechnungsfunktion 3D in 2D
virtual CVektor3D Koord2Din3D(const CVektor2D &GeoPunkt)	Umrechnungsfunktion 2D in 3D
CDPunkt KoordInView(CGEO_Punkt GeoPunkt)	Koordinatenumrechnung
void Position(CPosition Pos)	Abfragen der aktuellen Mausposition
CAntwort Position(CVektor2D Punkt)	Berechnung der aktuellen Postion unter Berücksichtigung der anderen Objekte
CAntwort Position(CGEO_Punkt P, CDPunkt Punkt, double Delta)	Berechnung der aktuellen Postion
void Loeschen(CGraphObjekt *Objekt)	Löscht das angegebene Objekt aus View/Auswahlliste
void Entfernen(CBauObjekt *BauObjekt)	Entfernung der für das BauObjekt erzeugten Objekte
void BoundBox()	Ermittlung der Bounding Box für die View
CGraphObjekt *NaechstesGraphObjekt(CGEO_Punkt *pPosition)	Suche des nächst gelegenen Objektes
Berücksichtigung der anderen Objekte	
Geschützte Methoden	***Beschreibung***
virtual void OnDraw (CDC *pDC)	Funktion für Nachricht OnDraw
afx_msg void OnChar (UINT nChar, UINT nRepCnt, UINT nFlags)	Funktion für Nachricht OnChar

Fortsetzung Klasse CView2D	
afx_msg void OnKeyUp (UINT nChar, UINT nRepCnt, UINT nFlags)	Funktion für Nachricht OnKeyUp
afx_msg void OnMouseMove (UINT nFlags, CPoint point)	Funktion für Nachricht OnMouseMove
afx_msg void OnLButtonDown (UINT nFlags, CPoint point)	Funktion für Nachricht OnLButtonDown
afx_msg void OnLButtonUp (UINT nFlags, CPoint point)	Funktion für Nachricht OnLButtonUp
afx_msg BOOL OnSetCursor (CWnd *pWnd, UINT nHit-Test, UINT message)	Funktion für Nachricht OnSetCursor
afx_msg void OnRButtonDown (UINT nFlags, CPoint point)	Funktion für Nachricht OnRButtonDown
Öffentliche Attribute	***Beschreibung***
CAuswahlbox Box	Box zum Auswählen und Zoomen
CPosition Positionen	Positionen in verschiedenen Systemen

Tabelle 21.4: Methoden und Attribute der Klasse CView2D

21.3.4 Implementierung 3D-View

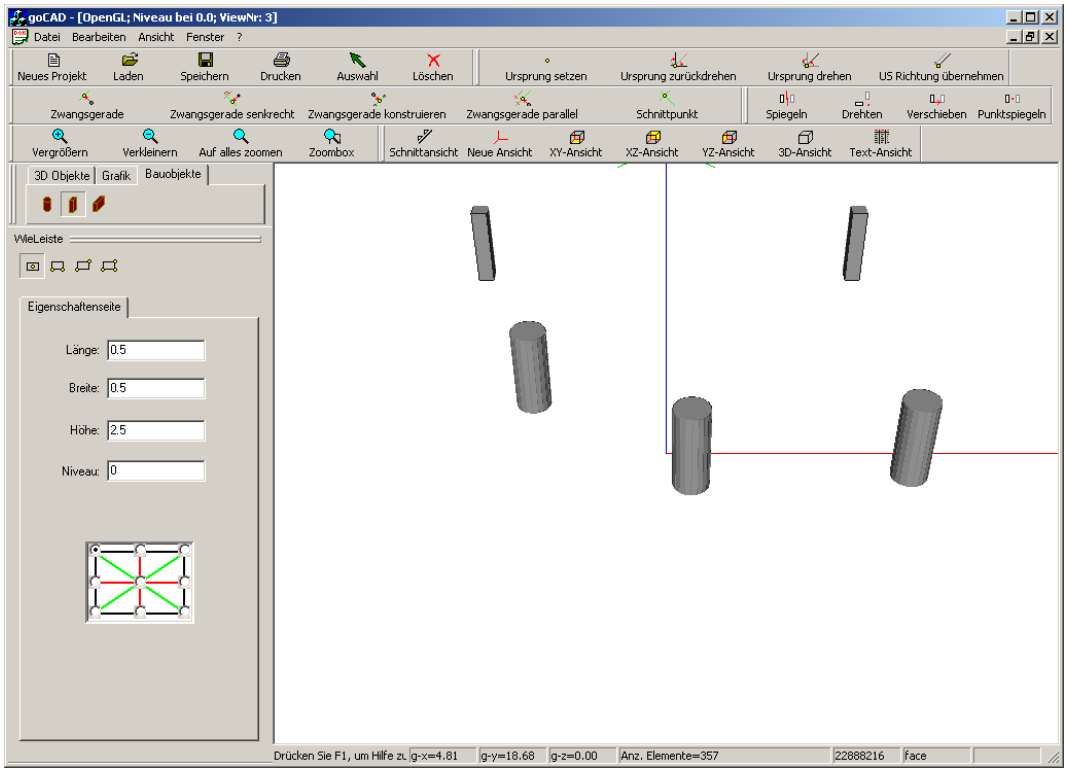


Abbildung 21.5: 3D-View von GoCAD

Die Grundlagen der 3D-View von GoCAD wurden im Rahmen einer Studienarbeit[1] entwickelt. Für die Visualisierung des Objektmodells wurde OpenGL[2] verwendet, hierdurch

[1] (Wei05a)

[2] OpenGl wurde 1992 von der Firma SGI als plattform- und programmiersprachenunabhängiges API zur Entwicklung von 3D-Graphik vorgestellt. OpenGL wird hardwaremäßig von Grafikkarten unterstützt. Der Einsatzbereich von OpenGL liegt heute hauptsächlich im professionellen CAD-Bereich und wird unter

wurde es ermöglicht, die Hardwarebeschleunigung von Grafikkarten zu verwenden, gleichzeitig kann die Renderfunktionalität von OpenGl verwendet werden, was die eigene Entwicklung erspart.

Für die Visualisierung in der 3D-View wird das Brep-Modell der Bauteile verwendet, indem die einzelnen Polygone des Objektes in der View dargestellt werden.

Dem Anwender wird die Möglichkeit zur Verfügung gestellt durch die Ansicht zu manövrieren. Über die Pfeiltasten lässt sich die Ansicht drehen, durch Drehen das Mausrades kann herein- und wieder herausgezoomt werden.

Mit der rechten Maustaste kann ein Objekt ausgewählt werden und über das erscheinende Menü die Textur und Farbe des Objektes in der Ansicht geändert werden

Klasse CViewOpenGL	
Öffentliche Methoden	***Beschreibung***
virtual ~CViewOpenGL()	Destruktor
void OnCreateGL()	Erzeugung einer neuen OpenGL-View
virtual BOOL PreCreateWindow(CREATESTRUCT &cs)	Wird vor Erzeugen der View aufgerufen
CGoCADDoc *GetDocument()	Abfragen des Dokumentes
virtual void OnDraw(CDC *pDC)	Zeichenfunktion
void OnDrawGL(CDC *pDC)	Zeichenfunktion
virtual CVektor2D Koord3Din2D (const CVektor3D &GeoPunkt)	Koordinatenumrechnung 3D in 2D
virtual CVektor3D Koord2Din3D(const CVektor2D &GeoPunkt)	Koordinatenumrechnung 2D in 3D
virtual void XMLSave(ofstream &datei)	Speicherfunktion
virtual void ErzeugeReduktionsmatrix()	Funktion zum Erzeugen der Reduktionsmatrix
virtual void ErzeugeViewEbene()	Erzeugen der Viewebene
void Paint(CBauObjekt *pBauObjekt)	Erzeugung der Viewdarstellung
void RegenList()	Benötigt für OpenGL View
const CString GetInformation(InfoField type)	Benötigt für OpenGL
void VideoMode(ColorsNumber &c, ZAccuracy &z, BOOL &dbuf)	Benötigt für OpenGL
void Entfernen(CBauObjekt *pBauObjekt)	Entfernt Bauobjekt aus Viewdarstellung
void SetMouseCursor(HCURSOR mcursor=NULL)	Setzen des Cursors
GLuint MoveSelection(CPoint Point, int *ptr)	Benötigt für OpenGL
CBauObjekt *ObjektunterMaus(CPoint point)	Funktion für Objektauswahl
Geschützte Methoden	***Beschreibung***
CViewOpenGL()	Konstruktor
void PaintScreen(GLenum mode)	Benötigt von OpenGL
afx_msg void OnSize(UINT nType, int cx, int cy)	Funktion für Nachricht OnSize
afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)	Funktion für Nachricht OnKeyDown
afx_msg void OnLButtonDown(UINT nFlags, CPoint point)	Funktion für Nachricht OnLButtonDown
afx_msg void OnRButtonDown(UINT nFlags, CPoint point)	Funktion für Nachricht OnRButtonDown
afx_msg void OnLButtonUp(UINT nFlags, CPoint point)	Funktion für Nachricht OnLButtonUp
afx_msg void OnMouseMove(UINT nFlags, CPoint point)	Funktion für Nachricht OnMouseMove
afx_msg int OnMouseWheel(UINT nFlags, short zDelta, CPoint pt)	Funktion für Nachricht OnMouseWheel

UNIX eingesetzt. Der OpenGL-Standard wird mittlerweile von der OpenGL ARB gepflegt.

Fortsetzung Klasse CViewOpenGL	
afx_msg BOOL OnEraseBkgnd(CDC *pDC)	Funktion für Nachricht OnEraeBkgnd
afx_msg void OnLinien()	Funktion für Nachricht OnLinien
afx_msg void OnFarbe()	Funktion für Nachricht OnFarbe
afx_msg void OnTextur()	Funktion für Nachricht OnTextur
afx_msg int OnCreate(LPCREATESTRUCT lpCreate-Struct)	Funktion für Nachricht Create
afx_msg void OnDestroy()	Funktion für Nachricht OnDestroy
afx_msg BOOL OnSetCursor(CWnd *pWnd, UINT nHitTest, UINT message)	Funktion für Nachricht OnSetCursor
afx_msg void OnOpenglTextur()	Funktion für Nachricht onOpenglTextur
Private Methoden	***Beschreibung***
void PaintViewpoint()	Zeichnen des Ursprungs
int MyErrFun(char *Fname, int Lnumber, char *FuncName)	Funktion zur Fehlerbearbeitung
unsigned charComponentFromIndex (int i, UINT nbits, UINT shift)	Benötigt für OpenGL
void CreateRGBPalette()	Erzeugung der Farbpalette
BOOL bSetupPixelFormat()	Benötigt für OpenGL
Öffentliche Attribute	***Beschreibung***
CBauAuswahlListe AuswahlListe	Liste der ausgewählten Objekte
Geschützte Attribute	***Beschreibung***
int OpenGLListe	OpenGL Objektliste
HGLRC m_hRC	OpenGl handle
Private Attribute	***Beschreibung***
CDC *m_pCDC	DC auf Zeichenfläche
HCURSOR m_hMouseCursor	Mauscursor
CPalette m_CurrentPalette	Farbpalette
CPalette *m_pOldPalette	Farbpalette
CRect m_ClientRect	Fensterauschnitt
int m_Fangradius	Fangradius
int m_DispListVector[20]	Displayliste
CPoint MouseDownPoint	Mausposition
GLdouble gldAspect	Benötigt für OpenGL
GLsizei glnWidth	Breite (OpenGL)
GLsizei glnHeight	Höhe (OpenGL)
double X_Angle	Winkel (OpenGL)
double Y_Angle	Winkel (OpenGL)
double depth	Tiefe (OpenGL)
double TranslationX	Wert für x Transformation (OpenGL)
double TranslationY	Wert für y Transformation (OpenGL)
COLORREF m_Farbe	Farbe
CString m_Textur	Texturname

Tabelle 21.5: Methoden und Attribute der Klasse CViewOpenGL

21.3.5 Implementierung der Textview

Die Textview von GoCAD ist eine Listenansicht auf das Datenmodell. Das Datenmodell wird ausgewertet und ausgewählte Eigenschaften in einer Liste dargestellt. In der Liste werden die folgenden Daten angezeigt:

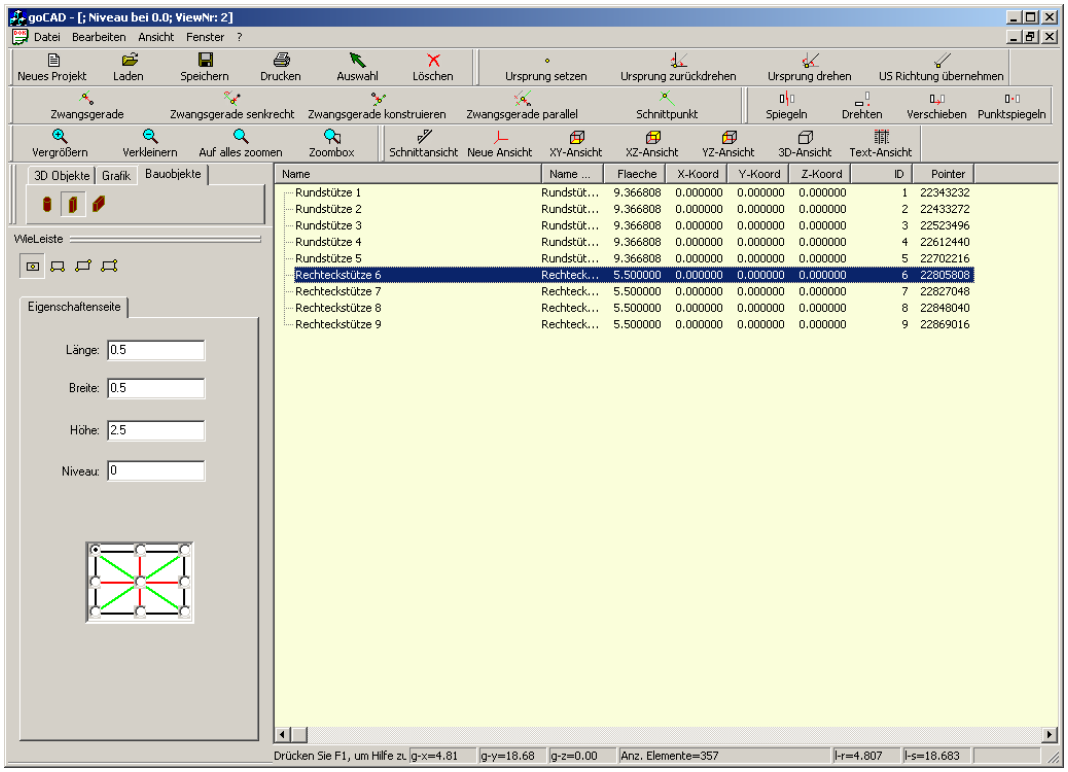


Abbildung 21.6: Textview von GoCAD

- Bezeichnung des Objektes
- Name der Klasse
- Oberfläche
- X, Y, Z-Koordinate des Setzpunktes
- Objekt ID
- Pointer

Eine Bearbeitung der Daten ist in der View nicht möglich, es ist nur eine Textsicht auf das Datenmodell. Erweiterungen hierfür beginnen mit Ändern der Eigenschaften und Löschen aus der Liste und gehen bis zur Erzeugung neuer Objekte und hierarchischer Anzeige der Modellstruktur.

Als Auswertung der Objekte ist exemplarisch die Oberfläche des Objektes angegeben. Weitere nicht implementierte Möglichkeiten der Auswertung sind das Volumen, die Unterteilung der Oberfläche nach der Bedeutung der Teilfläche für die Bauausführung.

Klasse CGoCADTreeView	
Öffentliche Methoden	***Beschreibung***
CGoCADTreeView()	Konstruktor
virtual ~CGoCADTreeView()	Destruktor

Fortsetzung Klasse CGoCADTreeView	
void InitListe()	Initialisierung der Liste
virtual void OnInitialUpdate ()	Funktion für Nachricht OninitialUpdate
void Initialize()	Initialisierungsfunktion
virtual void XMLSave(ofstream &datei)	Speicherfunktion
void SortTree(int nCol, BOOL bAscending, HTREEITEM hParent)	Sortieren der Liste
void ResetScrollBar()	Funktion für Scrollen
BOOL VerticalScrollVisible()	Abfrage Scrollbarzustand
BOOL HorizontalScrollVisible()	Abfrage Scrollbarzustand
int StretchWidth (int m_nWidth, int m_nMeasure)	Änderung der Breite
Geschützte Methoden	***Beschreibung***
CGoCADTreeView()	Konstruktor
virtual void OnDraw(CDC *pDC)	Funktion für Nachricht OnDraw
virtual BOOL OnNotify(WPARAM wParam, LPARAM lParam, LRESULT *pResult)	Funktion für Nachricht OnNotify
virtual BOOL PreCreateWindow(CREATESTRUCT &cs)	Funktion für Nachricht PreCreateWindow
virtual ~CGoCADTreeView()	Destruktor
afx_msg void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar *pScrollBar)	Funktion für Nachricht OnHScroll
afx_msg void OnSize(UINT nType, int cx, int cy)	Funktion für Nachricht OnSize
afx_msg void OnContextMenu(CWnd *pWnd, CPoint point)	Funktion für Nachricht OnContextMenu
CGoCADDoc *GetDocument()	Abfragen des Dokumentes
CVektor2D Koord3Din2D(const CVektor3D &GeoPunkt)	Umrechnung 3D in 2D
CVektor3D Koord2Din3D(const CVektor2D &GeoPunkt)	Umrechnung 2D in 3D
Private Methoden	***Beschreibung***
void DisplayToolTip(CPoint point, TVHITTESTINFO hInfo, CString colText)	Funktion für Tooltips
Öffentliche Attribute	***Beschreibung***
CImageList m_cImageList	Liste der Treebilder
BOOL m_RTL	Zustandsvariable
CFont m_headerFont	Schrifttyp
CTreeListCtrl m_tree	Liste
CScrollBar m_horScrollBar	Scrollbar
Private Attribute	***Beschreibung***
CTreeToolTipCtrl m_tooltip	tooltip
int m_IntervalTime	Variable für Zeit
BOOL m_TLInitialized	Zustandsvariable
int m_iAnz	Anzahl der Einträge

Tabelle 21.6: Methoden und Attribute der Klasse CGoCADTreeView

21.4 Implementierung des Controllers

Als Kontroller für das DV-Konzept wurde die Klasse *CControllCenter* entwickelt. Die Klasse *CControllCenter* ist sowohl Controller für das DVC-Konzept als auch zentrale Kontrollinstanz des Programms. Der Kontrollcenter hat die folgenden Funktionen

- Controller für die Views
- Verwaltung der Befehle

- Verwaltung der Builder
- Verwaltung der Toolbars
- Steuerung des Programms

Die Views reichen die Eingaben (Maus/Tastatur) über das Hauptfenster an das Kontrollcenter weiter. Das Kontrollcenter übergibt die Informationen an den aktiven Befehl (Builder), der diese dann weiterverarbeitet und auswertet.

21.4.1 Der Kontroller

Das Kontrollcenter dient nicht nur als Kontroller für das DVC-Konzept sondern auch als zentrale Steuereinheit für das gesamte Programm.

Die Eingaben, die der Anwender in den einzelnen Views vornimmt, werden zum Kontrollcenter weitergeleitet. Das Kontrollcenter überprüft, welcher Befehl bzw. Builder aktiviert ist und übergibt ihm die Eingabe. Die Vieweingaben, die vom Kontrollcenter verarbeitet werden sind:

- afx_msg void OnLButtonDown(UINT nFlags, CPoint point)
- afx_msg void OnLButtonUp(UINT nFlags, CPoint point)
- OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
- afx_msg void OnMouseMove(UINT nFlags, CPoint point)

Diese Eingaben werden von der View aufgearbeitet, z. B. die Position der Maus im Raum bestimmt und anschließend über MainFrame und das Kontrollcenter an den Befehl/Builder geleitet.

21.4.1.1 Befehlsverarbeitung

Nachdem der Anwender über die Toolbars einen Befehl oder Builder (abgeleitet von Befehl) gewählt hat, sendet der Button die Nachricht OnCommand, diese wird vom Mainframe entgegengenommen und an das Kontrollcenter weitergeleitet. Das Kontrollcenter sucht über die mitgeteilte Kenn-ID den Befehl aus der Befehlsliste bzw. Builderliste heraus und führt diesen aus.

Klasse CControlCenter	
Öffentliche Methoden	***Beschreibung***
CControlCenter()	Konstruktor

Fortsetzung Klasse CControlCenter	
virtual ~CControlCenter()	Destruktor
void DllLaden()	Laden der DLLs
void Init(CGoCADApp *pApp, CMDIFrameWnd *pMainFrame)	Initialisieren des Kontrollcenters
void Destroy()	Sichern von Werten beim Beenden des Programms
void InitGUI()	Initialisieren der Oberfläche
void regBefehl(CBefehlBasis *pBefehl)	Registriert Befehl
void AnhaengenAuszufuehrenderBefehl(CBefehlBasis *pBefehl)	Hängt den übergebenen Befehl an die Liste an
void AnhaengenAuszufuehrenderBefehl(int &Command)	Sucht den Befehl zur ID und hängt ihn an die Liste an
void AnhaengenLoeschen(CBefehlBasis *pBefehl)	Löscht die Befehlsliste und hängt einen Befehl an
CBefehlBasis *GetBefehl_zu_Command (UINT &CommandBefehl)	Suchen des Befehls zur Kennzahl
void Aktiviere_Befehl_zu_Command(UINT &CommandBefehl)	Aktiviert den Befehl zur Kennzahl
CVektor3D *PunktUebergebenLBDown(CVektor3D &Punkt, CGoCADView *pView)	Übergibt einen Punkt an den Befehl bei linker Taste runter
CVektor3D *PunktUebergeben(CVektor3D &Punkt, CGoCADView *pView)	Übergibt einen Punkt an den Befehl
bool BefehlAktiv()	Überprüft ob ein Befehl aktiv ist
CBefehlBasis *GetAktuellerBefehl()	Gibt einen Zeiger auf den aktuellen Befehl zurück
int GetIDAktuellerBefehl()	Fragt die ID des aktuellen Befehles ab
void Anhaengen(CBefehlBasis *pBefehl)	Hängt einen Befehl an die Liste der möglichen Befehle an
CBefehlBasis *GetBefehl (int &Nr)	Sucht den Befehl zur ID
int AnzahlBefehle()	Fragt die Anzahl der Befehle ab
bool Punkt_an_Befehl(CVektor3D &pPosition)	Übergibt ein Punkt an den aktuellen Befehl
void ueberzeichnenAlleViews(CBefehlBasis *pBefehl, CVektor3D AktuellerPunkt)	Überzeichnet die Befehle in allen Views
void zeichnenAlleViews(CBefehlBasis *pBefehl)	Zeichnet die Befehle in allen Views
void BefehlslisteZuruecksetzen()	Befehle aus Liste entfernen
void BefehlDeaktivieren()	Befehle deaktivieren
void UrsprungGeaendert(CBefehlBasis *pBefehl)	Abfragen ob Ursprung geändert wurde
CGoCADView *ErzeugeNeueAnsicht(const CZeichenattribute *pAttr, void *tag=NULL, CGoCADDoc *pDoc=NULL)	Neue Ansicht erzeugen
LRESULT OnTasteGedrueckt(WPARAM wParam, LPARAM lParam)	Funktion reagiert auf drücken einer Taste
LRESULT BuilderPropertyChanged(WPARAM wParam, LPARAM lParam)	Funktion reagiert auf Änderung der Eigenschaften
LRESULT OnButton(WPARAM wParam, LPARAM lParam)	Reaktion auf PSButton
void OnUpdateCommandRange(CCmdUI *pCCmdUI)	Reaktion auf Toolbar
void OnCommandRange(UINT nID)	Reaktion auf Toolbar
CBuilderBasis *GetBuilder(CString &Objekttyp)	Builder suchen
void DoOptionenDialog()	Optionsdialog ausführen
void BuilderDeaktivieren()	Deaktivieren des aktiven Builders
void BuilderEingabeBeenden()	Beenden der Buildereingabe
void ueberzeichnenAlleViews(CBuilderBasis *pBuilder, CVektor3D AktuellerPunkt)	Überzeichnen der Builder
void zeichnenAlleViews(CBuilderBasis *pBuilder)	Zeichnen der Builder
CBauObjekt *Punkt_an_Builder(CVektor3D &pPosition)	Punkt an Builder weiterreichen
bool PunktUebergebenAnBuilder(CVektor3D &AktuellePosition, CGoCADView *pView)	
CBuilderBasis *GetAktiverBuilder()	Aktiven Builder abfragen
Öffentliche Attribute	***Beschreibung***

Fortsetzung Klasse CControlCenter	
CBuilderBasis *m_pAktiverBuilder	Pointer auf den aktiven Builder
CBefehlsliste li_Befehle	Liste der vorhandenen Befehle
Private Typen	***Beschreibung***
typedef vector<HINSTANCE> DllListe	Liste für die geladenen DLLs
typedef DllListe::iterator iterDLL	Iterator für die DLLs
Private Attribute	***Beschreibung***
CMDIFrameWnd *m_pMainFrame	Pointer auf das Hauptfenster
CGoCADApp *m_pApp	Pointer auf die Anwendung
CIni m_Inidatei	Eingelesene Inidatei
DllListe m_Dll_Liste	Liste der DLLs
CBefehlsliste li_AktiverBefehle	Aktive Befehle
CBuilderVerwaltung m_BuilderVerwaltung	Verwaltung der Builder
CBefehlsVerwaltung m_BefehlesVerwaltung	Verwaltung der Befehle
CToolBarVerwaltung m_ToolbarVerwaltung	Verwaltung der Toolbars
CPropSheetBar m_pbWasleiste	WasLeiste
CPropSheetBar m_pbWieleiste	Wieleiste
CPPWieSeite m_pPage	Leere Seite für die Wieleiste
CPPWasLeiste *Leerseite	Leere Eigenschaftsseite
CPSButton *AktiverBFButton	Aktiver Befehlsbutton
CPSButton *AktiverBuilderButton	Aktiver Builderbutton
UINT m_AktiveBuilderFamilie	Nummer der aktiven Buildergruppe
bool m_bzeichnen	Gibt an ob der Builder gezeichnet werden soll

Tabelle 21.7: Methoden und Attribute der Klasse CControlCenter

21.4.2 Die Befehlsverwaltung

Die Verwaltung der Befehle in GoCAD ist in der Klasse *CBefehlsVerwaltung* implementiert. Die Befehlsverwaltung bekommt nach dem Einlesen eines Befehls aus einer DLL den Zeiger auf den Befehl übergeben (`regBefehl(CBefehlBasis*)`). Die Methode fügt den Befehl in die Befehlsliste ein und erzeugt die zum Befehl gehörende ID. Die ID des Befehls setzt sich zusammen aus der Summe von 42000 und der Eintragsnummer des Befehles in der Liste. Die IDs der Befehle befinden sich im Bereich 42000-42999. Die Methode `regBefehl()` übergibt den Befehl anschließend an die Toolbarverwaltung, die den Toolbutton für den Befehl erzeugt. Des Weiteren verfügt die Klasse noch über zwei Funktionen, die den passenden Befehl anhand der ID bzw. des Befehlsnamens aus der Befehlsliste heraussucht.

Klasse CBefehlsVerwaltung	
Öffentliche Methoden	***Beschreibung***
CBefehlsVerwaltung()	Konstruktor
virtual ~CBefehlsVerwaltung()	Destruktor
void reset()	Zurücksetzen der Befehlsliste
void regBefehl (CBefehlBasis *Befehl)	Befehl in die Liste einfügen
CBefehlBasis *GetBefehl_zu_Command(int CommandBefehl)	Sucht den Befehl zum Command
CBefehlBasis *GetBefehl_zu_Name(CString &Name)	Sucht den Befehl zum Namen

Fortsetzung Klasse CBefehlsVerwaltung	
void SetToolBarVerwaltung(CToolBarVerwaltung *pTBVerwaltung)	Funktion zum Setzen des Pointers der Toolbarverwaltung
void SetControlCenter(CControlCenter *pControlCenter)	Funktion zum Setzen des Pointers auf das Kontrollcenter
Private Typen	***Beschreibung***
typedef vector<CBefehlBasis*> BefehlsListe	Liste der Befehle
typedef BefehlsListe::iterator iterBefehl	Iterator
Private Attribute	***Beschreibung***
BefehlsListe m_BefehlsListe	Befehlsliste
CControlCenter *m_pControlCenter	Pointer auf des ControlCenter Parent"
CToolBarVerwaltung *m_pToolBarVerwaltung	Pointer auf die Toolbarverwaltung

Tabelle 21.8: Methoden und Attribute der Klasse CBefehlsVerwaltung

21.4.3 Die Builderverwaltung

Die unterschiedlichen Builder sind mit den Objekten in DLLs ausgelagert. Nach dem Einlesen der Builder aus den DLLs werden diese in der Builderverwaltung abgelegt. In der Builderverwaltung sind die Builder hierarchisch geordnet.

In der 1. Ordnungsebene sind die Buildergruppen, in diesen Gruppen sind thematisch zusammengehörende Builderfamilien zusammengefasst. Zu den Buildergruppen gehören u. a.:

- Graphische Objekte
- Bauteile
- 3D Objekte

Die Builderfamilien bilden die zweite Ebene der Verwaltung. Sie fassen die einzelnen Builder eines Bauobjekttyps zusammen. Die Builderfamilien der Bauteile umfassen:

- Wandbuilder
- Stützenbuilder
- Deckenbuilder

Die dritte und unterste Ebene sind die einzelnen Builder eines Bauteils, die in einer Builderfamilie zusammengefasst sind. Zu jedem Bauobjekt gibt es in der Regel mehrere Builder für die Erzeugung, welche sich in der Anzahl der Punkte und der weiteren Angaben, die die Builder für die Erzeugung der Bauobjekte benötigen, unterscheiden. Es sind Varianten der folgenden Builder vorhanden:

- Einpunktbuilder
- Zweipunktbuilder
- Dreipunktbuilder
- Polygonbuilder

21.4.3.1 Registrierung der Builder

Nach dem Einlesen eines Builders wird dieser in der Builderverwaltung registriert. Im ersten Schritt wird die Buildergruppe des neuen Builders mit den vorhandenen Buildern verglichen. Dies geschieht durch die Methode `Gruppenzugehoerigkeit()` der Klasse *CBuilderGruppe*. Die Methode `Gruppenzugehoerigkeit()` vergleicht die Gruppe des Builders mit den vorhandenen Gruppen. Die Überprüfung geschieht anhand des Gruppennamens der als String in der Basisklasse *CBuilderFamilie* des Builders gespeichert wird.

Ist die benötigte Buildergruppe nicht vorhanden, wird eine neue Instanz der Klasse *CBuilderGruppe* erzeugt und in der BuilderGruppenListe der Builderverwaltung eingefügt. Für die neue Gruppe wird eine Registrierkarte in der WasLeiste angelegt. Der Gruppenname der neuen Gruppe wird auf den benötigten Namen gesetzt, es wird mit dieser Gruppe weitergearbeitet.

Im nächsten Schritt wird die zum Builder gehörende Builderfamilie gesucht. Hierfür wird überprüft, ob in der BuilderFamilienListe der Buildergruppe die entsprechende *CBuilderFamilienVerwaltung* für die Familie vorhanden ist. Wenn diese vorhanden ist, wird der Builder in die BuilderListe der entsprechenden *CBuilderFamilienVerwaltung* eingetragen. Fehlt die benötigte *CBuilderFamilienVerwaltung*, so wird diese neu erzeugt und in die BuilderFamilienListe eingetragen. In die WasLeiste der BuilderGruppe wird der Button für die BuilderFamilie eingetragen.

Die Überprüfung der BuilderFamilie geschieht in der Methode `Gruppenzugehoerigkeit(CBuilderFamilienVerwaltung *pBuilderFam)` über einen dynamic_cast. Da alle zu einer Familie gehörenden Builder von derselben BuilderFamilie abgeleitet sind, liefert die Funktion bei Übereinstimmung der Familie ein TRUE zurück ansonsten ein FALSE.

Klasse CBuilderVerwaltung	
Öffentliche Methoden	***Beschreibung***
CBuilderVerwaltung()	Konstruktor
virtual ~CBuilderVerwaltung()	Destruktor
CBuilderBasis *GetBuilder(CString &Objekttyp)	Builder abfragen
CBuilderBasis *GetBuilder(UINT nID)	Builder abfragen
void reset()	Listen zurücksetzen

Fortsetzung Klasse CBuilderVerwaltung	
void SeitenEinfuegenweg(CPropSheetBar *pPropertyBar)	Seite in Eigenschaftsdialog einfügen
void regBuilder (CBuilderBasis *pBuilder)	Fügt einen Builder in die Builderliste ein, vergibt die IDs
void InitBuilderBar(CPropSheetBar *pBar, UINT ID)	Initialisieren der Builderleiste
bool BuilderVorhanden()	Fragt das Vorhandensein von Buildern ab
void InitSeiten(CTabCtrl *ptabCtrl)	Baut die Seite der Gruppe auf
void SeitenEinfuegen(CPropSheetBar *pPropertyBar)	Fügt die Propertypages der Buildergruppen in die Wasleiste ein
Private Typen	***Beschreibung***
typedef vector<CBuilderGruppe*>BuilderGruppenListe	Liste der Buildergruppen
typedef BuilderGruppenListe::iterator iterBuilderGruppe	Iterator
typedef vector<CBuilderFamilienVerwaltung*> BuilderFamilienListe	Liste der BuilderFamilien
typedef BuilderFamilienListe::iterator iterBuilderFam	Iterator
typedef vector<CBuilderBasis*> BuilderListe	Liste der Builder
typedef BuilderListe::iterator iterBuilder	Iterator
Private Attribute	***Beschreibung***
BuilderGruppenListe m_BuilderGruppenListe	Liste für die Buildergruppen
BuilderFamilienListe m_BuilderFamilienListe	Liste für die Builderfamilien
BuilderListe m_BuilderListe	Liste für die Builder

Tabelle 21.9: Methoden und Attribute der Klasse CBuilderVerwaltung

21.4.4 Die Toolbarverwaltung

Die Toolbarverwaltung organisiert das Erscheinungsbild der Befehle in den Toolbars. Nach der Registrierung eines Befehls in der Befehlsverwaltung bekommt die Toolbarverwaltung einen Zeiger auf den Befehl übergeben. Die Toolbarverwaltung überprüft zuerst, ob für die Befehlsgruppe, zu der der Befehl gehört, eine Toolbar vorhanden ist. Wenn die passende Toolbar vorhanden ist, wird der Befehl in die Toolbar eingefügt. Wenn keine Toolbar vorhanden ist, wird eine Toolbar erzeugt und anschließend der Button für den Befehl eingefügt.

Die Funktion `InitEigenschaftenDlg()` bekommt eine Eigenschaftsseite mit einer Liste übergeben. In dieser Liste sind die vorhandenen Toolbars eingetragen. Über die Checkbox am Anfang des Listeneintrages kann der Anwender auswählen, ob die Toolbar angezeigt oder ausgeblendet wird.

Klasse CToolBarVerwaltung	
Öffentliche Methoden	***Beschreibung***
CToolBarVerwaltung()	Konstruktor
virtual ~CToolBarVerwaltung()	Destruktor
void SetTBParent (CMDIFrameWnd *pMainFrame)	Pointer auf das Hauptfenster, Parent der Toolbars
void DeInitEigenschaftenDlg()	Deinitialisieren des Eigenschaftsdialog
void InitEigenschaftenDlg(TreePropSheet::CTreePropSheet &PropSheet)	Initialisieren des Eigenschafsdialoges
void ReadToolBarPos(CIni &Ini)	Einlesen der Toolbarposition

Fortsetzung Klasse CToolBarVerwaltung	
void WriteToolBarPos(CIni &Ini)	Schreiben der Toolbarposition
int ButtonInToolBar(CBefehlBasis *pBefehl)	Einfügen eines Buttons
Private Typen	***Beschreibung***
typedef vector<CPsToolbar*> ToolBarListe	Liste der Toolbars
typedef ToolBarListe::iterator iterToolbar	Iterator
Private Attribute	***Beschreibung***
ToolBarListe m_Toolbar_Liste	Liste für die Toolbars
CMDIFrameWnd *m_pMainFrame	Zeiger auf das Hauptfenster
CPP_ToolBar *m_pPageToolBar	Eigenschaftsseite der Toolbars
CTreeToolBarEintraege *m_pEintraege	Toolbareintrag

Tabelle 21.10: Methoden und Attribute der Klasse CToolBarVerwaltung

22 Implementierung des Datenmodells

Für das CAD-Programm GoCAD wurde ein vereinfachtes Datenmodell entwickelt. Das Ziel beim Entwurf des Datenmodells war nicht der Entwurf eines vollständigen Datenmodells für ein CAD-Programm, sondern die Demonstration des Aufbaus und der Funktionsweise eines Datenmodells.

Die Basisklasse des Datenmodells ist die Klasse *CBauobjekt*. Sie beinhaltet die grundlegenden Eigenschaften und Methoden des Bauobjektes. Die Methoden sind, sofern sie bauobjektspezifisch sind, virtuell deklariert und werden später von den abgeleiteten Klassen überschrieben.

Zur Speicherung der Geometrie der Bauteile wird ein Brep-Modell verwendet. Dieses Modell wurde im Rahmen einer Diplomarbeit (Jen03) an der TU Kaiserslautern von Jochen Jenter entwickelt und implementiert. In diesem Brep-Modell sind sowohl die Volumenobjekte als auch die graphischen Objekte gespeichert. In Abbildung 22.1 wurde das Klassendiagramm der Basisklasse dargestellt, es beinhaltet nur die wichtigsten Eigenschaften, deren Bedeutung im Folgenden näher erläutert werden. Die Variable BrepObjekt ist

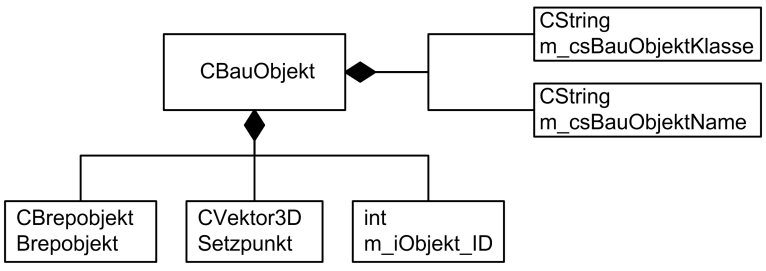


Abbildung 22.1: BauobjektKlassendiagramm

ein Brep-Modell, das die Geometrie des Objektes beschreibt. Der Setzpunkt ist die 3D-Koordinate des Mittelpunktes des Objektes. m_iObjekt_ID ist eine Kennnummer des Objektes, diese wird direkt nach dem Erzeugen des Objektes vergeben und ist so implementiert, dass sie nur einmal vorkommt und nicht geändert werden kann. Die ID bleibt auch bei der Speicherung und dem Laden des Objektes erhalten. m_csBauObjektName

beinhaltet den Namen des Bauobjektes, er kann frei vergeben und jederzeit geändert werden, ist jedoch vorbelegt mit einer Kombination aus dem Objekttyp und der ID. m_csBauObjekteKlasse enthält den Namen der Objektklasse, sie wird mit der ID vergeben und zur Feststellung der Zugehörigkeit eines Objektes zur Klasse verwendet. Die Klasse CBauObjekt hat die folgenden Methoden:

Klasse CBauObjekt	
Öffentliche Methoden	***Beschreibung***
CBauObjekt()	Konstruktor
virtual ~CBauObjekt()	Destruktor
void XMLSave(ofstream &datei)	Speichern des Objektes
virtual const char *GetObjektTyp()	Abfrage des Objekttypes der Klasse
CPropertyPage *GetPropertyPage()	Erzeugen der Eigenschaftsseite
void SetBauObjektName(CString &Objektname)	Setzen des Objektnamens
virtual void Trans(CMatrix *pmat_Transformation)	Transformieren des Objektes
COLORREF GetFarbe()	Abfragen der Objektfarbe (für 3D)
void SetFarbe(COLORREF Farbe)	Setzen der Objektfarbe (für 3D)
CString GetTextur()	Abfragen der Textur (für 3D)
void SetTextur(CString Textur)	Setzen der Textur (für 3D)
bool Ausgewaehlt()	Abfrage des „Auswahlzustandes“ des Bauobjektes
void Ausgewaehlt(bool auswahl)	Änderung des Auswahlzustandes
int GetObjektID()	Abfrage der ObjektID
CVektor3D GetSetzpunkt()	Gibt den Setzpunkt des Objektes zurück
void SetSetzpunkt(CVektor3D &Punkt)	Setzen des Setzpunktes
virtual CGeoObjekt *GetGeoObjekt()	Abfrage des zum Bauobjekt gehörigen geometrischen Objektes
CBauObjekt *Get_BauObjekt()	Liefert Zeiger auf sich selbst
CBrepObjekt *Get_BrepObjekt()	Liefert eingebundenes BrepObjekt
void SetObjektID(int ID)	Setzen der ID des Objektes, nur einmal möglich
CString GetBauObjektKlassenName()	Abfragen des Klassennamens des Bauobjektes
CString GetBauObjektName()	Abfragen des Namens des Bauobjektes
int GetPointerInt()	Abfragen des Pointers als Integerwert
double GetFlaeche()	Funktion berechnet die Fläche des Brep-Modells
bool IstBrepObjekt()	Abfragen ob das Brep-Modell verwendet wird
bool IstBrepObjekt2D()	Abfragen ob das Brep-Modell-2D verwendet wird
virtual void Trans(CMatrix *pmat_Transformation)	Transformieren des Bauobjektes
bool Ausgewaehlt()	Abfrage des „Auswahlzustandes“ des Bauobjektes
void Ausgewaehlt(bool auswahl)	Änderung des Auswahlzustandes
int ObjektID()	Abfrage der ObjektID
CGEO_Punkt GetSetzpunkt()	Gibt den Setzpunkt des Objektes zurück
CBauObjekt *Get_BauObjekt()	Liefert Zeiger auf sich selbst
CBrepObjekt Get_BrepObjekt()	Liefert eingebundenes BrepObjekt
Öffentliche Attribute	***Beschreibung***
COLORREF m_crFarbe	Objektfarbe
CString m_csTextur	Texturname
Geschützte Attribute	***Beschreibung***
COLORREF Schnittfarbe	Farbe von Schnittlinien
bool m_bBrep2D	Brep 2DModell verwendet?
bool m_bBrep	Brep Modell verwendet?
bool m_bID_Gesetzt	Schon eine ID vergeben?
CBrepObjekt BrepObjekt	Enthält BrepStruktur
bool m_bausgewaehlt	Zustandsvariable, True oder False
CString m_csBauObjektName	Name des Bauobjektes

Fortsetzung Klasse CBauObjekt	
CString m_csBauObjektKlasse	Name der Bauobjektklasse
CVektor3D Setzpunkt	Setzpunkt/Zentrum des Bauobjektes
int m_iObjekt_ID	Kennzahl des Objektes, wird vom Programmierer vergeben

Tabelle 22.1: Methoden und Attribute der Klasse CBauObjekt

22.1 Methoden des Datenmodells

Die meisten der Methoden der Klasse CBauObjekt sind zum Abfragen und Setzen der Eigenschaften. Zur Auswertung des Objektes wurde exemplarisch die Methode `GetFlaeche()` implementiert, sie ermittelt über das Brep-Modell die Oberfläche des Objektes, indem die Teilflächen des Brep-Modells berechnet und aufaddiert werden.

Über Funktion `Trans()` wird das Objekt transformiert, sie bekommt die Transformationsmatrix übergeben und wendet sie anschließend auf das Brep-Modell des Objektes an und berechnet die transformierte Geometrie. Bei einer Lageänderung des Objektes wird sie auch noch auf den Setzpunkt angewandt und die aktuelle Lage im Raum bestimmt.

Die Funktion `SaveXML()` speichert das Objekt in einer XML-Datei. Sie wird in den abgeleiteten Klassen überschrieben, damit die objektspezifischen Eigenschaften gespeichert werden.

Die Methode `CPropertyPage *GetPropertyPage()` erzeugt eine Eigenschaftsseite für das Objekt und gibt einen Pointer auf diese zurück. Diese dient zur Anzeige der Eigenschaften des Objektes auf Aufforderung des Anwenders.

22.2 Objektklassen des Datenmodells

Das Datenmodell umfasst drei verschiedene Objekttypen; diese sind:

1. Graphische Objekte
2. 3D-Objekte
3. Bauteile

Die Klassenhierarchie des Objektmodells ist in Abbildung 22.2 dargestellt. Die graphischen Objekte sind der einfachste Objekttyp. Er umfasst die typischen Elemente eines

2D-CAD-Systems wie Gerade, Kreis, Polygon etc. Es handelt sich bei den graphischen Objekten um 2D-Objekte, zur Ergänzung der Viewdarstellung, sie können auch als Hilfslinien verwendet werden. Für GoCAD wurden die folgenden drei Objekttypen implementiert: Punkt, Linie, Kreis.

Die 3D-Objekte sind allgemeine 3D-Objekte aber noch keine Bauteile. Sie dienen der Erzeugung beliebiger Objekte. Zu den vorhandenen 3D-Objekten gehören: Würfel, Rotationskörper, Translationskörper.

Abschließend wurden noch einige Bauteile implementiert, es sind die Objekte eines 3D-CAD-Systems für Bauwesen. In GoCAD vorhanden sind die Rundstütze, die Rechteckstütze und die Wand.

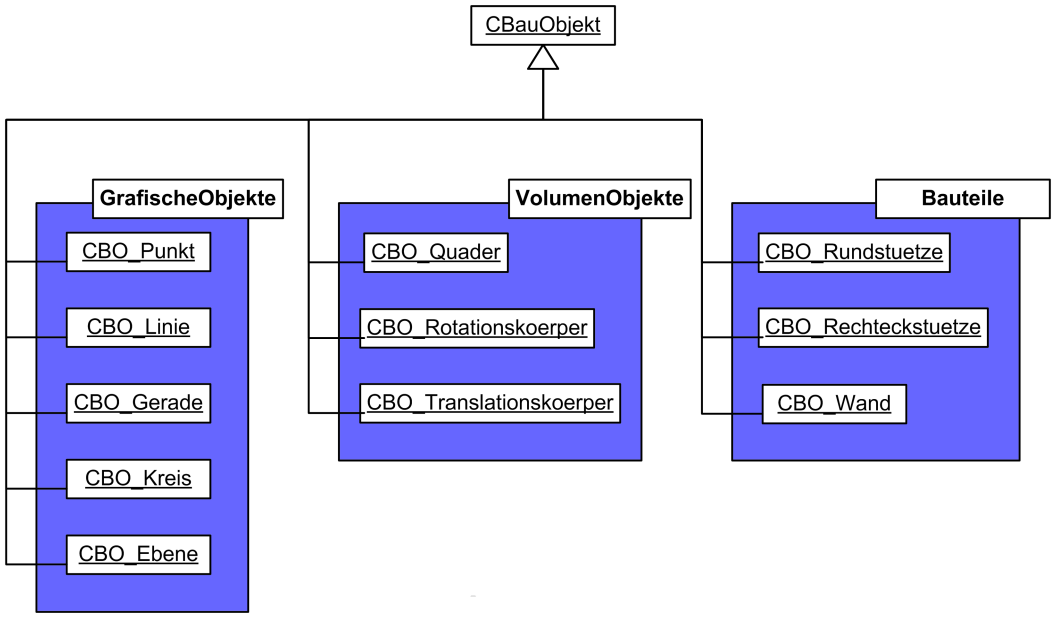


Abbildung 22.2: Klassenhierarchie des Objektmodells

23 Implementierung des Builderkonzeptes

Bei den Buildern handelt es sich um eine spezielle Form eines Befehles zum Erzeugen von Bauobjekten.

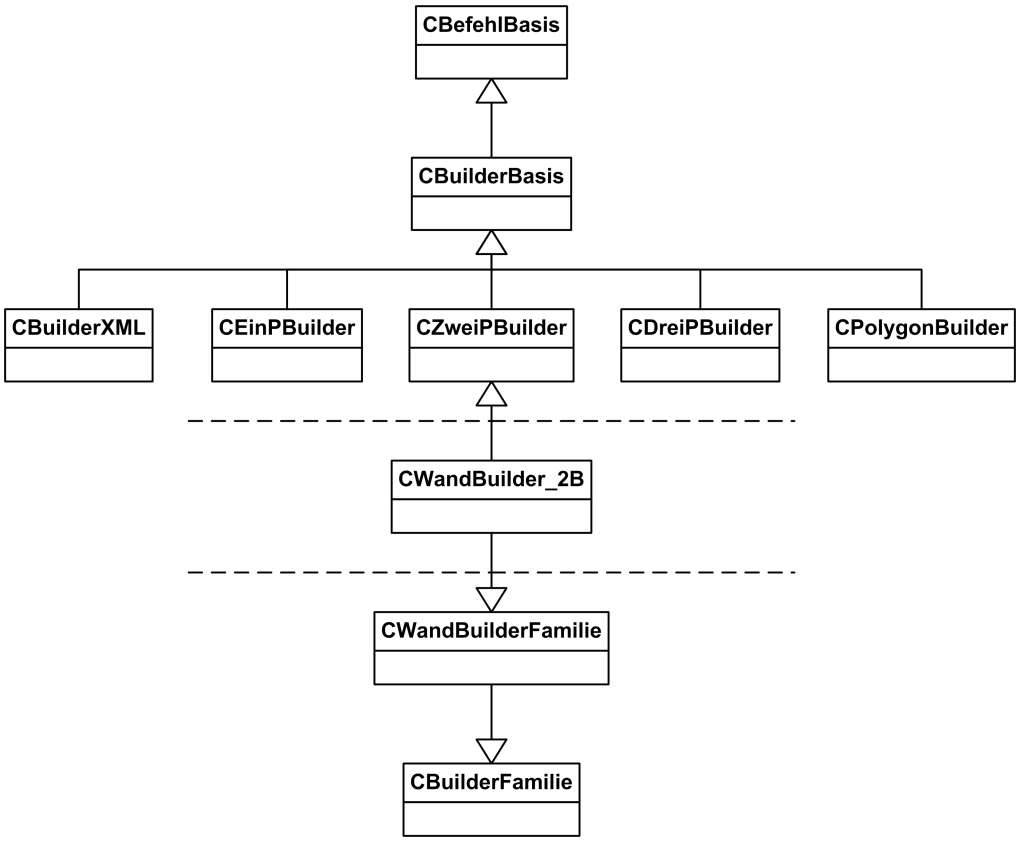


Abbildung 23.1: Hierarchie der Klasse CWandBuilder_2P

Die Basisklasse der Builder *CBuilderBasis* wurde deshalb von *CBefehlBasis* abgeleitet. Im nächsten Schritt wurden von *CBuilder* die einzelnen Buildergrundformen *CEinPBuilder*, *CZweiPBuilder*, *CDreiPBuilder*, *CPolygonBuilder* und *CBuilderXML* abgeleitet. Die Punktbuilder werden verwendet für die interaktive, graphische Konstruktion der Bauobjekte. Sie unterscheiden sich in der Anzahl der Punkte, die für die Konstruktion verwendet werden. Der XML-Builder erzeugt die Objekte aus den Daten einer XML-Datei beim Ein-

lesen des gespeicherten Objektmodells. Von diesen Klassen wurden die weiteren Builder für die Erzeugung eines Bauobjektes abgeleitet.

Die zweite Basisklasse der Builder ist die Klasse *CBuilderFamilie*, sie dient zur Verwaltung der Builder eines Bauobjektes.

23.1 Die Builderklassen

23.1.1 Die Klasse CBuilderBasis

Die Klasse *CBefehl* bildet die Basisklasse der Klasse *CBuilderBasis*. Sie enthält die Erweiterungen des Builders gegenüber dem Befehl. Das Klassendiagramm von *CBuilderBasis* ist in Abbildung 23.2 dargestellt.

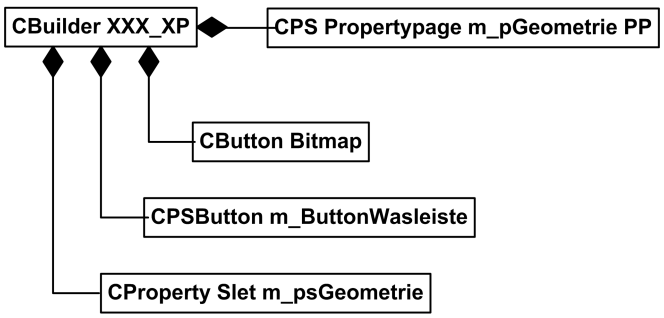


Abbildung 23.2: Klassendiagramm der Klasse CBuilderBasis

Die Klasse *CBuilderBasis* deklariert alle Methoden und Eigenschaften, die die Builder für die Objekterzeugung benötigen.

Klasse CBuilderBasis	
Öffentliche Methoden	***Beschreibung***
CBuilderBasis()	Konstruktor
virtual ~CBuilderBasis()	Destruktor
virtual CBauObjekt *bauen()	Übergeben eines Punktes an den Builder
virtual CBauObjekt *bauen(fstream &datei)	Kommentar
void reset()	Builder zurücksetzten.
void reset(CPropSheetBar *pPropertySheet)	Zurücksetzen der Propertypage, Leerseite wieder einfügen
void GetBuildPropertySet()	Abfragen des Propertysets.
virtual void PropertyDialogErstellen(CPropSheetBar *pPropertySheet)	Propertypage erstellen
CPSButton *GetBuilderButton()	Abfragen des Builderbuttons
virtual void BilderLaden()	Bilder für den Button laden
void SetIDButton(int iD)	Setzen der ID für den Button

Fortsetzung Klasse CBuilderBasis	
bool VergleicheID(int ID)	Vergleicht die übergebenen ID mit der Button ID
virtual CVektor2D Position(ICViewEigenschaften *pViewEigen, CVektor2D Position)	Abfragen der Position.
virtual LRESULT Taste(WPARAM wParam, LPARAM lParam)	Reaktion auf Tastatureingabe
virtual CString GetObjektTyp()	Abfragen des Objekttyps der den Builder erzeugt
CString GetBuilderTyp()	Abfragen des Typs des Builders
Geschützte Attribute	***Beschreibung***
CString m_csBuilderName	Name der Builderfamilie
CString m_csBuilderTyp	Art des Builders
CPSButton m_ButtonWasLeiste	Button für die Wasleiste
CBitmap * m_pBuilderBildUp	Bild für Buttonbuilder
CBitmap * m_pBuilderBildDown	Bild für Buttonbuilder
CBitmap * m_pBuilderBildOver	Bild für Buttonbuilder
CBitmap * m_pBuilderBildDisable	Bild für Buttonbuilder
UINT m_idBuilderBildUp	ID des Buttons
UINT m_idBuilderBildDown	ID des Buttons
UINT m_idBuilderBildOver	ID des Buttons
UINT m_idBuilderBildDisable	ID des Buttons
int m_idBuilderButton	ID der Familie - Button
CVektor3D ZwischenPunkt	Aktuelle Zwischenpunkte
CPsPropertyPage * m_pGeometriePP	Eigenschaftsseite mit den Daten für die Erstellung
CPropertyPage * m_pLeerSeite	Pointer auf die Leerseite
CPropertySet * m_psGeometrie	Eigenschaften der Linie

Tabelle 23.1: Methoden und Attribute der Klasse CBuilderBasis

Die Methode `bauen()` ist die eigentliche Erzeugungsfunktion des Builders für die Objekte. Sie ermittelt aus den eingegebenen Punkten und dem Propertyset alle für die Erzeugung benötigten Werte, ruft den Konstruktor für das neue Objekt auf und gibt abschließend das dynamisch erzeugte Objekt zurück.

Über die Methode `virtual CVektor2D Position(ICViewEigenschaften* pViewEigen, CVektor2D Position)` kann die aktuelle Position der Maus in Abhängigkeit des Builders während der Konstruktion ermittelt werden. Mit `virtual void PropertyDialogErstellen(CPropSheetBar* pPropertySheet)` wird eine Eigenschaftsseite für den Builder erzeugt und anschließend in die übergebene *CPropSheetBar* eingefügt. Die Methode `void GetBuildPropertySet()` liest den aktuellen Propertyset aus der Propertypage aus. Die `reset()`-Funktion wird aufgerufen, wenn der Konstruktionsvorgang vorzeitig abgebrochen wird. Der Builder wird auf die Ausgangswerte zurückgesetzt und die Konstruktionsvorschau in den verschiedenen Views gelöscht.

Die Methoden `CPSButton* GetBuilderButton()`, `virtual void BuilderLaden()`, `void SetIDButton(int iD)` und `bool VergleicheID(int ID)` werden für die Button des Builders in der Builderauswahl des Wiedialoges benötigt.

23.1.2 Die Klasse CBuilderFamilie

Die Klasse *CBuilderFamilie* ist die zweite Basisklasse der Builder. Diese Klasse wurde zur Verwaltung der Builder eingeführt.

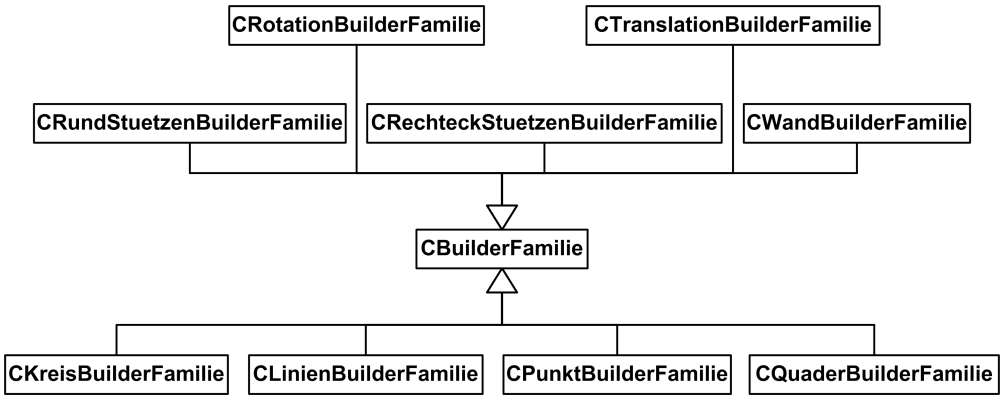


Abbildung 23.3: Klassendiagramm der Klasse CBuilderFamilie

Fast alle Eigenschaften und Methoden der Klasse *CBuilder* werden benötigt für die Darstellung und die Verwaltung der Builder.

Die CString-Variable m_csBuilderGruppenName beinhaltet den Namen der Buildergruppe zu der die Builderfamilie gehört. Die Zugehörigkeit zur Buildergruppe wird durch den Vergleich des Gruppennamens überprüft. Zwar besitzt die Builderfamilie eine CString-Variable für den Namen, für die Überprüfung der Builder zur Familie wurde jedoch ein anderes Verfahren gewählt als bei der Zugehörigkeit zur Buildergruppe. Um sicherzustellen, dass die Zuordnung zur Builderfamile eindeutig ist, findet die Überprüfung mit einem dynamic-cast statt. Die Methode `GleicheFamilie()` bekommt einen Builder übergeben und castet diesen auf die Klasse der Builderfamilie. Dieser Cast gelingt nur, wenn der Builder von dieser Builderfamilie abgeleitet ist. Als Ergebnis liefert er, wenn er gelingt, ein TRUE zurück, ansonsten ein FALSE.

Die weiteren Funktionen der Klasse werden für das Einfügen der Button der Builderfamilie in der Wasleiste benötigt. Die Variablen des Typs CBitmap, UINT und CPSButton werden für die Darstellung der BuilderFamilie in der Was-Leiste benötigt.

Klasse CBuilderFamilie	
Öffentliche Methoden	***Beschreibung***
CBuilderFamilie()	Konstruktor
virtual ~CBuilderFamilie()	Destruktor
virtual bool GleicheFamilie(CBuilderBasis *pBuilder)	Überprüft, ob der Builder zur Familie gehört
int GetIDButton()	Abfragen der ID für den Builder/Button
bool SetIDButton(int ID)	Setzen der Button ID
CPSButton * GetFamilienButton()	Erzeugt einen Button für die Builderfamilie

Fortsetzung Klasse CBuilderFamilie	
CString GetBuilderFamilienName()	Abfragen des Namens der Builderfamilie
CString GetBuilderGruppenName()	Abfragen der Buildergruppe
Geschützte Attribute	***Beschreibung***
CString m_csBuilderGruppenName	Name der Gruppe, zu der die Familie gehört
CString m_csBuilderFamilienName	Name der Builderfamilie
CPSButton m_ButtonWasLeiste	Button für die Wasleiste
CBitmap * m_pButtonBildUp	Bild für Button
CBitmap * m_pButtonBildDown	Bild für Button
CBitmap * m_pButtonBildOver	Bild für Button
CBitmap * m_pButtonBildDisable	Bild für Button
UINT m_idBitmapUp	ID des Buttons
UINT m_idBitmapDown	ID des Buttons
UINT m_idBitmapOver	ID des Buttons
UINT m_idBitmapDisable	ID des Buttons
int m_idFamilienButton	ID der Familie - Button.

Tabelle 23.2: Methoden und Attribute der Klasse CBuilderFamilie

23.1.3 Implementierte Builder

Objekt	XML	1 Punkt	2 Punkt	3Punkt	Polygon
Wand	X		X	X	X
Rundstütze	X	X	X		
Rechteckstütze	X	X	X	X	
Kreis	X	X	X		
Gerade	X	X	X		
Würfel	X	X	X	X	
Translationskörper	X				X
Rotationskörper	X				X

Abbildung 23.4: Übersicht über die implementierten Builder

Für die Erstellung der verschiedenen Bauteile wurden unterschiedliche Builder, Einpunktbuilder, Zweipunktbuilder, Dreipunktbuilder und Polygonbuilder entwickelt. In Tabelle 23.4 ist eine Übersicht der verschiedenen Builder dargestellt. Es wurde nicht für jedes Bauobjekt jede Art von Builder implementiert sondern für verschiedene Objekte ausgesuchte Builder. Die Builder die verwendet werden, sind von den Bauteilen abhängig, da nicht für jedes Bauteil jeder Buildertyp sinnvoll ist. Es kommt auch vor, dass von einem Buildertyp wie 2 Punkt Builder, 3 Punkt Builder für einen Objekttyp mehrere Varianten existieren, die sich in der erzeugten Objektgeometrie unterscheiden.

23.1.4 Die Klasse CXMLBuilder

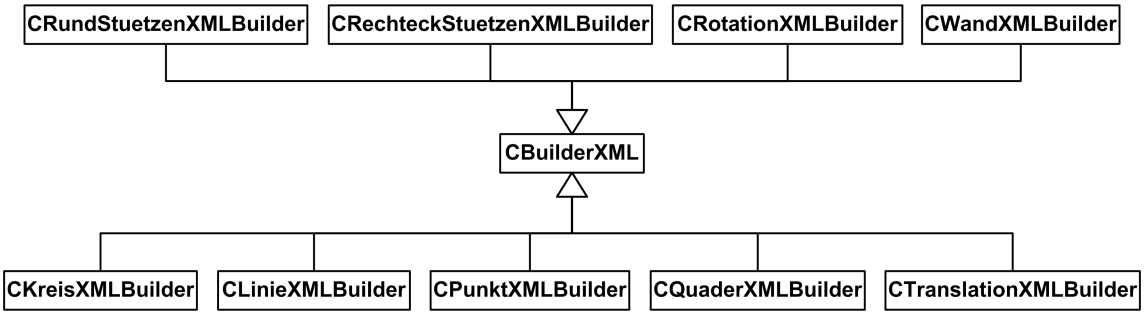


Abbildung 23.5: Klassendiagramm der von CBuilderXML abgeleiteten Builder

Die Klasse *CXMLBuilder* ist von der Klasse *CBuilder* abgeleitet. Sie dient als Basisklasse für alle XML-Builder. Die XML-Builder finden Verwendung bei der Erzeugung von Bauobjekten aus einer XML-Datei. Die XML-Builder besitzen nur einen Konstruktor und Destruktor und sonst keine weiteren Methoden. Die abgeleiteten Klassen sind in Abbildung 23.5 dargestellt.

Klasse CXMLBuilder	
Öffentliche Methoden	***Beschreibung***
CBuilderXML()	Konstruktor
virtual ~CBuilderXML()	Destruktor

Tabelle 23.3: Methoden und Attribute der Klasse CXMLBuilder

23.1.4.1 Die Funktionsweise der XML-Builder

Klasse CxxxXMLBuilder	
Öffentliche Methoden	***Beschreibung***
CBauObjekt *bauen (fstream &datei)	Objekt erzeugen
CString GetObjektTyp ()	Abfragen des Objekttyps, den der Builder erzeugt
CKreisXMLBuilder()	Konstruktor
virtual ~CKreisXMLBuilder()	Destruktor

Tabelle 23.4: Methoden und Attribute der Klasse CxxxXMLBuilder

Die XML-Builder sind von der Klasse *CBuilderBasis* und den verschiedenen Klassen *CXXXBuilderFamilie* abgeleitet. Es existiert für jedes Bauteil ein XML-Builder. GoCAD verwendet zur Speicherung der Daten eine Datei im XML-Format. Die XML-Builder werden verwendet, um beim Einlesen des Datenmodells die Bauteile aus den Daten zu erzeugen.

Die Einleseroutine von GoCAD in CGoCADDoc arbeitet in der Methode `OnOpenDocu-`

`ment()` den XML-Baum der Datei durch. Trifft die Funktion auf ein Bauobjekt (XML-Tag </Bauobjekte>), wird zunächst der Typ des Bauobjektes (XML-Tag </Objekttyp>) bestimmt, der beim Speichern angegeben wurde. Dies geschieht mit Hilfe der Methode `typeid(*this).name()`. Der Typ des Bauobjektes ist gleich dem Namen der Klasse.

Über diesen Namen wird der zum Bauobjekt gehörige XML-Builder aus der Builderliste gesucht, dieser wird im weiteren Verlauf benutzt.

Die Methode `bauen()` des XML-Builders bekommt einen Zeiger auf den Einlesestream übergeben. Die Funktion erzeugt zuerst ein Objekt der entsprechenden Klasse, danach werden die einzelnen Attribute des Objektes eingelesen und dem neu erzeugten Objekt übergeben. Abschließend wird das Brep-Modell des Objektes eingelesen. Die Methode `bauen()` wurde in der Basisklasse als virtual deklariert und in den abgeleiteten Klassen mit der konkreten Funktion überschrieben.

23.1.5 Die Klasse CEinPBuilder

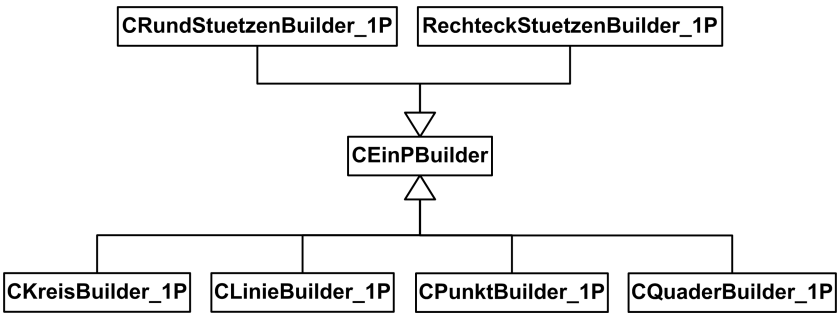


Abbildung 23.6: Klassendiagramm der Klasse CxxxBuilder_1P

Neben dem Konstruktor und Destruktor überschreibt die Klasse noch die Methode `Punkt-Uebergeben()` der Basisklasse. Die Funktion regelt die Eingabe der für die Konstruktion benötigten Punkte, gibt TRUE zurück wenn die benötigten Punkte in diesem Fall ein Punkt eingegeben sind, ansonsten FALSE. Abbildung 23.6 zeigt alle von *CEinPBuilder* abgeleiteten Klassen.

Klasse CEinPBuilder	
Öffentliche Methoden	***Beschreibung***
CEinPBuilder()	Konstruktor
virtual ~CEinPBuilder()	Destruktor
bool PunktUebergeben(CVektor3D *pPunkt)	Fügt neuen Punkt in Liste ein

Tabelle 23.5: Methoden und Attribute der Klasse CEinPBuilder

23.1.5.1 Funktionsweise der Einpunktbuilder

Klasse CxxxBuilder_1P	
Öffentliche Methoden	***Beschreibung***
CxxxBuilder_1P()	Konstruktor
virtual ~CxxxBuilder_1P()	Destruktor
void BilderLaden()	Laden der Bilder für die Toolbar
void zeichnenView(CDC *pDC, ICViewEigenschaften *pViewEigenschaften)	Zeichnen während der Eingabe
CBauObjekt* bauen()	Erzeugen des Bauobjektes

Tabelle 23.6: Methoden und Attribute der Klasse CxxxBuilder_1P

Die abgeleiteten Einpunktbuilder überschreiben die Methoden `BilderLaden()`, `zeichnenView()` und `bauen()`. Die Methode `BilderLaden()` dient zum Laden der Bilder der Builder für die Toolbars. `Bauen()` erzeugt das neue Bauobjekt aus den eingegebenen Daten, nachdem die Position des Objektes eingegeben wurde. Die Methode `zeichnenView()` sorgt für die Darstellung während der Konstruktion des Objektes. Die Übergabeparameter für das Zeichnen sind ein Zeiger auf den Gerätekontext und auf die Eigenschaften der View. Die Darstellung erfolgt immer in allen Ansichten gleichzeitig. Bei den Einpunktbuildern hängt das Erscheinungsbild der Geometrie nur von den Angaben in der Wieleiste ab, so dass es sich während der Konstruktion nicht ändert.

23.1.6 Die Klasse CZweiPBuilder

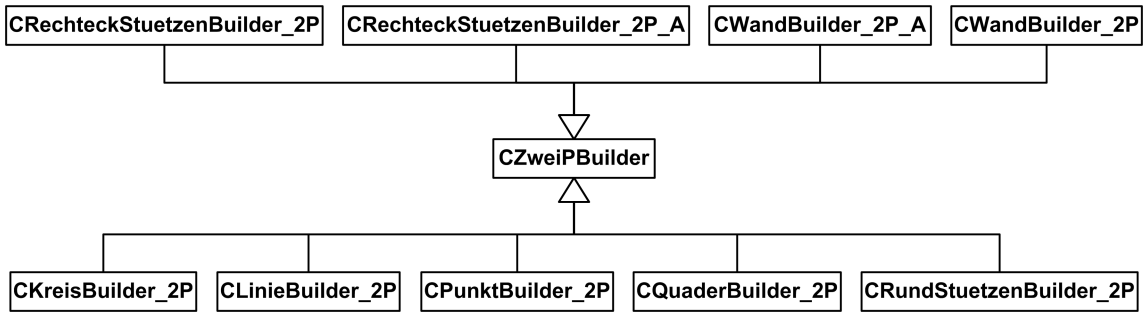


Abbildung 23.7: Klassendiagramm der Klasse CxxxBuilder_2P

Die Klasse ist gleich der Klasse der Einpunktbuilder aufgebaut, der Unterschied ist die Anzahl der für die Konstruktion benötigten Punkte, in diesem Fall zwei Punkte. Die Klassenhierarchie der Zweipunktbuilder stellt Abbildung 23.7 dar.

Klasse CZweiPBuilder	
Öffentliche Methoden	***Beschreibung***
CZweiPBuilder()	Konstruktor
virtual ~CZweiPBuilder()	Destruktor

Fortsetzung Klasse CZweiPBuilder	
bool PunktUebergeben(CVektor3D *pPunkt)	Fügt neuen Punkt in Liste ein

Tabelle 23.7: Methoden und Attribute der Klasse CZweiPBuilder

23.1.6.1 Die Zweipunktbuilder

Klasse CxxxBuilder_2P	
Öffentliche Methoden	***Beschreibung***
CxxxBuilder_2P()	Konstruktor
virtual ~CxxxBuilder_2P()	Destruktor
void BilderLaden()	Laden der Bilder für die Toolbar
void zeichnenView(CDC *pDC, ICViewEigenschaften *pViewEigenschaften)	Zeichnen während der Eingabe
CBauObjekt *bauen()	Erzeugen des Bauobjektes

Tabelle 23.8: Methoden und Attribute der Klasse CxxxBuilder_2P

Die Methode `bauen()` des Builders erstellt das neue Objekt aus zwei Punkten und dem Eingeben in der Wasleiste. Aus den zwei Punkten werden durch die Funktion entweder die Ausrichtung im Raum (Quadratstütze) oder die Länge des Objektes (Wand) berechnet. Eine neue Konstruktionsrichtung kann, muss aber nicht, durch die Funktion berechnet werden.

23.1.7 Die Klasse CDreiPBuilder

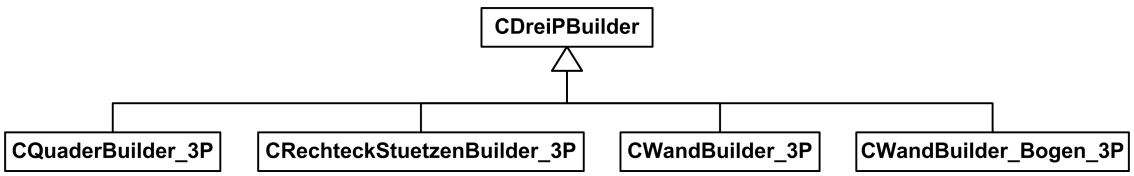


Abbildung 23.8: Klassendiagramm der Klasse CxxxBuilder_3P

Die Klasse der *Dreipunktbuilder* entspricht der der Ein- und Zweipunktbuilder. Die Methode `PunktUebergeben()` arbeitet mit drei Punkten, nach dem dritten Punkt wird TRUE zurückgegeben. Bild 23.8 zeigt die von CDreiPBuilder abgeleiteten Klassen.

Klasse CDreiPBuilder	
Öffentliche Methoden	***Beschreibung***
CDreiPBuilder()	Konstruktor
virtual ~CDreiPBuilder()	Destruktor
bool PunktUebergeben(CVektor3D *pPunkt)	Fügt neuen Punkt in Liste ein

Tabelle 23.9: Methoden und Attribute der Klasse CDreiPBuilder

23.1.7.1 Die Dreipunktbuilder

Die Dreipunktbuilder besitzen neben den Funktionen der Ein- und Zweipunktbuilder noch die Methode `Position()`. Diese Methode berechnet die aktuelle Position des Cursors in Abhängigkeit von der bisherigen Eingabe.

Klasse CxxxBuilder_3P	
Öffentliche Methoden	***Beschreibung***
CxxxBuilder_3P()	Konstruktor
virtual ~CxxxBuilder_3P()	Destruktor
void BilderLaden()	Laden der Bilder für die Toolbar
void zeichnenView(CDC *pDC, ICViewEigenschaften *pViewEigenschaften)	Zeichnen während der Eingabe
CBauObjekt *bauen()	Erzeugen des Bauobjektes
CVektor2D Position(ICViewEigenschaften *pViewEigen, CVektor2D Position)	Bestimmung der aktuellen Position in Abhängigkeit vom Builder

Tabelle 23.10: Methoden und Attribute der Klasse CxxxBuilder_3P

23.1.8 Die Klasse CPolygonBuilder

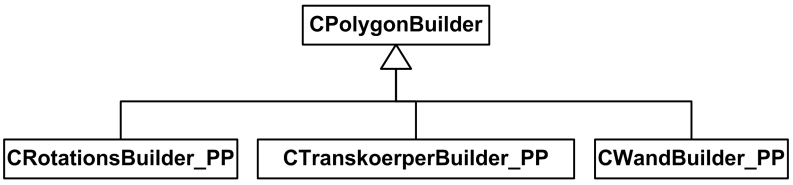


Abbildung 23.9: Klassendiagramm der Klasse CxxxBuilder_PP

Die Klasse *CPolygonBuilder* unterscheidet sich von den anderen Punktbuildern vor allem dadurch, dass die Anzahl der Punkte nicht begrenzt ist. Der Builder besitzt die Methode `NeueRichtung2()`. Diese Funktion wird in der Methode `PunktUebergeben()` verwendet und berechnet nach jeder Punkteingabe die neue Konstruktionsrichtung.

Klasse CPolygonBuilder	
Öffentliche Methoden	***Beschreibung***
CPolygonBuilder()	Konstruktor
virtual ~CPolygonBuilder()	Destruktor
void NeueRichtung2(int Anfang, int Ende)	Berechnung der neuen Konstruktionsrichtung
void reset()	Builder zurücksetzen
bool AllePunkte()	Gibt an, ob alle Punkte des Builders eingegeben wurden
int MaxAnzPunkte()	Abfrage der Anzahl der benötigten Punkte
virtual LRESULT Taste (WPARAM wParam, LPARAM lParam)	Reaktion auf Tastatureingabe
bool PunktUebergeben (CVektor3D *pPunkt)	Fügt neuen Punkt in Liste ein
Öffentliche Attribute	***Beschreibung***
CGEOPunktListe GPListe_2	2. Liste der eingegebenen Punkte

Fortsetzung Klasse CPolygonBuilder	
bool m_bListe_1_voll	True wenn Liste 1 voll
bool m_bListe_2_voll	True wenn Liste 2 voll
int m_iMaxAnzPunkte_2	Maximalanzahl der Punkte in Liste 2
int m_iMinAnzPunkte_1	Mindestanzahl der Punkte in Liste 1
int m_iMinAnzPunkte_2	Mindestanzahl der Punkte in Liste 2

Tabelle 23.11: Methoden und Attribute der Klasse CPolygonBuilder

Im Gegensatz zu den anderen Buildern wird beim Polygonbuilder immer nach jeder Punkteingabe die neue Richtung bestimmt. Der Builder reagiert auch auf Tasteneingabe, mit der bei den Rotations- und Translationsbuildern die Eingabe beendet wird. Die Klasse *CPolygonBuilder* besitzt als einziger der Builder zwei Punktlisten. Diese beiden Punktlisten werden für den Rotations- und Polygonbuilder verwendet, da diese zwei Polygone, das eine für die Form, das andere für die Translationsstrecke bzw. Rotationsachse, benötigen. Die Klassenhierarchie der Polygonbuilder ist im Diagramm 23.9 zu sehen.

23.1.8.1 Die Polygonbuilder

Die Polygonbuilder sind die komplexesten Builder. Die Funktionalität entspricht der der anderen Builder, jedoch ist die `bauen()`-Funktion die umfangreichste. Die Methode erzeugt entweder ein ebenes Bauteil mit unregelmäßiger Form (Decke) oder ein fortlaufendes Objekt mit konstantem Querschnitt (Wand).

Klasse CxxxBuilder_PP	
Öffentliche Methoden	***Beschreibung***
CxxxBuilder_PP ()	Konstruktor
virtual ~CxxxBuilder_PP()	Destruktor
void BilderLaden()	Laden der Bilder für die Toolbar
void zeichnenView(CDC *pDC, ICViewEigenschaften *pViewEigenschaften)	Zeichnen während der Eingabe
CBauObjekt *bauen()	Erzeugen des Bauobjektes

Tabelle 23.12: Methoden und Attribute der Klasse CxxxBuilder_PP

24 Implementierung des Painterkonzepts

Durch den Schichtaufbau von GoCAD kommt es auch zu einer strikten Trennung des Datenmodells und der Sichten. Die Daten werden neutral im Dokument gespeichert, hieraus müssen die für die Darstellung benötigten Informationen gewonnen und entsprechend aufgearbeitet an die View weitergegeben werden. Dieser Vorgang wird durch das implementierte Painterkonzept realisiert.

In GoCAD sind verschiedene Painter realisiert. Dies sind zum einen die drei viewtypspezifischen Painter für die OpenGl-, Text- und 2D-View, die für alle 3D-Objekte mit Brep-Modell zuständig sind, zum anderen die objekt- und viewspezifischen Painter der graphischen Objekte, welche speziell für die Objekte und die 2D- bzw. Textview sind.

24.1 Painterklassen

24.1.1 Basisklasse CPainter

Die Klasse *CPainter* dient als Basisklasse, alle weiteren Painter sind von ihr abgeleitet. In ihr ist die Basisfunktionalität der Painter implementiert. Für GoCAD wurden die folgenden Klassen von *CPainter* abgeleitet:

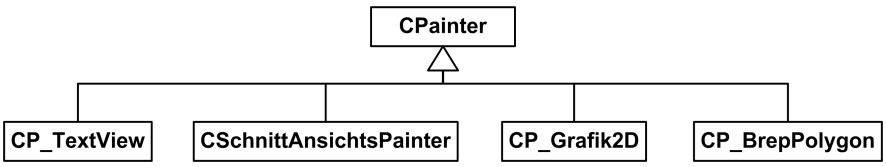


Abbildung 24.1: Klassenhierarchie der Painter

Die Painter lassen sich nach verschiedenen Gesichtspunkten einteilen:

- Objektunabhängige Painter für einen bestimmten Viewtyp

  - Painter für die 2D-Views und Brepobjekte CPainterAnsichtSchnitt
  - Painter für die 2D-Views und Graphikelemente CP_Grafik2D
  - Painter für die Textview - CP_TextView
  - Painter für die OpenGLView, nur für Brepobjekte - CP_BrepPolygon

- Die zweite Gruppe sind die objektabhänigen Painter. Implementiert wurden diese für verschiedene Objekte und die Viewklasse CView2D.

  - Painter für graphische Objekte und CView2D
    - CP_BOEbene
    - CP_BOPunkt

- Allgemeine Geometrie

  - CP_BOQuader
  - CP_BORotationskoerper
  - CP_BOTranslationskoerper

- Bauteile

  - CP_BORechtStuetze
  - CP_BORundStuetze
  - CP_BOWand

Klasse CPainter	
Öffentliche Methoden	***Beschreibung***
CPainter()	Konstruktor
CPainter(CPainter &Painter)	Konstruktor
virtual ~CPainter()	Destruktor
void LoadXML(std::fstream &datei)	Ladefunktion des Painters
void SaveXML(std::ofstream &datei)	Speicherfunktion des Painters
virtual BOOL Painter(CBauObjekt* pBauObjekt, ViewType Viewtyp)return false;	Suchen des passenden Painters
virtual void Paint(ICViewEigenschaften *pView, CBauObjekt* pBauObjekt)	Objekt darstellen
CPropertyPage* GetPropertyPage()	Abfragen der Eigenschaftsseite
CPainter* GetCopy()	Painter kopieren
CPainterDarstellungsEigenschaften &GetDarstellungsEigenschaften()	Abfragen der Darstellungseigenschaften
void SetDarstellungsEigenschaften(CPainterDarstellungsEigenschaften &Eigenschaften)	Setzen der Darstellungseigenschaften
void SetViewTyp(ViewType Typ)	Setzen des Viewtyps
ViewType GetViewTyp()	Abfragen des Viewtyps
CString GetPainterName()	Abfragen des Painternamens
void SetPainterName(CString Name)	Setzen des Painternamens
operator =, operator ==, operator !=	Kopier- und Vergleichsoperatoren
virtual CString BO_Klasse()	Name der zugehörigen Bauobjektklasse
int PainterID()	Abfrage der PainterID

Fortsetzung Klasse CPainter	
Geschützte Attribute	**Beschreibung**
CPainterDarstellungsEigenschaften m_PD_Eigenschaften	Variable für die Paintereigenschaften
CString m_csPaintername	Name des Painters
ViewType Viewtyp	Variable für Viewtyp
int Painter_ID	KennID des Painters
int m_iLinienTyp	Variable für den Linientyp
double m_dLinienBreite	Variable für die Linienbreite
COLORREF m_crFarbe	Variable für die Farbe

Tabelle 24.1: Funktionen und Variablen der Klasse CPainter

Der Painter besitzt drei Membervariablen. Eine Variable der Struktur ViewType, sie beinhaltet die Viewart, für die der Painter verwendet werden kann. Eine CString-Variable m_csPaintername, für den Namen des Painters. Der Name wird verwendet für den Eintrag des Painters in die Listbox des Verwaltungsdialoges. Die Variable m_PD_Eigenschaften der Klasse *CPainterDarstellungsEigenschaften* beinhaltet die Informationen für die Darstellung des Objektes.

Die Funktionen der Klasse *CPainter* werden im Folgenden näher erläutert. Die Methode `Painter()` überprüft die Zuständigkeit des Painters für das Objekt in der aktuellen View. Die Zuständigkeit ist abhängig von dem Objekttyp und dem Viewtyp. Der Rückgabewert ist ein boolescher Wert, in der Basisklasse auf FALSE gesetzt. Die Funktion Painter ist als virtual deklariert und wird in den abgeleiteten Klassen überschrieben.

Die Methode `Paint()` beinhaltet den Algorithmus für die Darstellung des Objektes in der View. Sie ist in der Basisklasse als virtuell deklariert. Die abgeleiteten Klassen überschreiben diese Funktion mit einem konkreten Algorithmus. Die Übergabeparameter dieser Funktion sind ein Zeiger auf die Schnittstellenklasse ICViewEigenschaften und ein Zeiger auf das Bauobjekt, das dargestellt werden soll.

Die Funktion `CPropertyPage* GetPropertyPage()` erzeugt die Eigenschaftsseite des Painters für den Eigenschaftsdialog des Objektes. Über diese Seite können die Darstellungsparameter des Objektes verändert werden.

Die `MethodeCPainter* GetCopy()` erzeugt eine Kopie des Painters und gibt einen Zeiger auf diese Kopie zurück. Diese Funktion wird in der Painterverwaltung für die Erzeugung neuer Painter verwendet.

24.1.2 Class CPainterDarstellungsEigenschaften

Die Klasse *CPainterDarstellungsEigenschaften* beinhaltet die Parameter, die der Painter für die Darstellung des Objektes benötigt. Sie besitzt drei Variablen der Klasse *CStifte-Eigenschaften* m_SE_SchnittKanten, m_SE_SichtbareKanten, m_SE_Systemachsen. Diese sind die Linienarten für die Darstellung der sichtbaren Kanten, der geschnittenen Kanten des Modells mit der Schnittebene und die Systemachsen des Objektes.

Klasse CPainterDarstellungsEigenschaften	
Öffentliche Methoden	***Beschreibung***
CPainterDarstellungsEigenschaften()	Konstruktor
CPainterDarstellungsEigenschaften(const CPainterDarstellungsEigenschaften &mEigenschaften)	Konstruktor
virtual ~CPainterDarstellungsEigenschaften()	Destruktor
CPainterDarstellungsEigenschaften &operator =(const CPainterDarstellungsEigenschaften &mEigenschaften)	Kopieroperator
BOOL Painter(CBauObjekt *pBauObjekt, ViewType Viewtyp)	Funktion zur Paintersuche
void Paint(ICViewEigenschaften *pViewEigen, CBauObjekt *pBauObjekt)	Darstellungsfunktion
bool operator ==(const CPainterDarstellungsEigenschaften &mEigenschaften)	Vergleichsoperator
bool operator !=(const CPainterDarstellungsEigenschaften &mEigenschaften)	Vergleichsoperator
Öffentliche Attribute	***Beschreibung***
CStiftEigenschaften m_SE_SchnittKanten	Stifteigenschaft
CStiftEigenschaften m_SE_SichtbareKanten	Stifteigenschaft
CStiftEigenschaften m_SE_Systemachsen	Stifteigenschaft
CBrushEigenschaften m_BE_VorderesPolygon	Pinseleigenschaft
CBrushEigenschaften m_BE_HinteresPolygon	Pinseleigenschaft

Tabelle 24.2: Funktionen und Variablen der Klasse CPainterDarstellungsEigenschaften

Die Klasse *CBrushEigenschaften* ist zwei Mal vorhanden als m_BE_VorderesPolygon; m_BE_HinteresPolygon. Diese Klasse beinhaltet die Beschreibung des hinteren und vorderen Schnittpolygons von geschnittenen Objekten.

Neben den Konstruktoren besitzt die Klasse noch die beiden Operatoren == und != zum Vergleichen der Darstellungseigenschaften verschiedener Painter.

24.1.3 CStiftEigenschaften

Die einzelnen Stifte werden über Objekte der Klasse *CStiftEigenschaften* definiert. Diese Klasse beschreibt die Eigenschaften einer Linie, dies umfasst die Farbe, den Linientyp und die Linienstärke.

Klasse CStiftEigenschaften	
Öffentliche Methoden	***Beschreibung***
CStiftEigenschaften(const CStiftEigenschaften &Eigenschaften)	Konstruktor
CStiftEigenschaften &operator =(const CStiftEigenschaften &Eigenschaften)	Kopieroperator
CStiftEigenschaften()	Konstruktor
virtual ~CStiftEigenschaften()	Destruktor
bool operator ==(const CStiftEigenschaften &Eigenschaften)	Vergleichsoperator
bool operator !=(const CStiftEigenschaften &Eigenschaften)	Vergleichsoperator
Öffentliche Attribute	***Beschreibung***
COLORREF Farbe	Farbvariable
int Art	Variable für die Linienart
int Breite	Variable für die Linienbreite

Tabelle 24.3: Funktionen und Variablen der Klasse CStiftEigenschaften

Die Variable Farbe beschreibt die Farbe der Linie als COLORREF-Variable. Die Integervariable Art beschreibt die Linienart. Es sind fünf verschiedene GDI-Linienarten implementiert.

- PS_SOLID - durchgezogene Linie
- PS_DASH - gestrichelte Linie
- PS_DOT - Punkt Linie
- PS_DASHDOT - Strich Punkt Linie
- PS_DASHDOTDOT - Strich Punkt Punkt Linie

Die Variable Breite steht für die Breite der Linie, diese wird in 1/100 mm angegeben. Es sind die folgenden acht Linienstärken implementiert:

- 13 - 0,13mm
- 18 - 0,18mm
- 25 - 0,25mm
- 35 - 0,35mm
- 50 - 0,50mm
- 100 - 1,00mm
- 140 - 1,40mm
- 200 - 2,00mm

An Funktionen besitzt diese Klasse neben den Konstruktoren, Destruktor und Kopierkonstruktor noch die zwei Vergleichsoperatoren == und !=.

24.1.4 CBrushEigenschaften

CBrushEigenschaften ist das Äquivalent von *CStiftEigenschaften* für die Darstellung der Schnittflächen.

Klasse CBrushEigenschaften	
Öffentliche Methoden	***Beschreibung***
CBrushEigenschaften(const CBrushEigenschaften &Eigenschaften)	Konstruktor
CBrushEigenschaften()	Konstruktor
virtual ~CBrushEigenschaften()	Destruktor
CBrushEigenschaften &operator =(const CBrushEigenschaften &Eigenschaften)	Kopieroperator
bool operator ==(const CBrushEigenschaften &Eigenschaften)	Vergleichsoperator
bool operator !=(const CBrushEigenschaften &Eigenschaften)	Vergleichsoperator
Öffentliche Attribute	***Beschreibung***
int m_iMuster	Variable für das Füllmuster
COLORREF m_crFarbe	Variable für die Füllfarbe

Tabelle 24.4: Funktionen und Variablen der Klasse CBrushEigenschaften

Diese Klasse besitzt die beiden Variablen m_crFarbe und m_iMuster. Die m_crFarbe ist eine COLORREF-Variable für die Farbe des Musters, die Intergervariable m_iMuster gibt den Mustertyp an. Es ist das folgende Muster implementiert:

- HS_BDIAGONAL (Schraffur diagonal rechts oben nach links unten)

CBrushEigenschaften besitzt die beiden Operatoren == und != für den Vergleich von verschiedenen Objekten der Klasse *CBrushEigenschaften* sowie den Zuweisungsoperator =.

24.2 Implementierte Painter

In GoCAD wurden verschiedene Views implementiert, für die Darstellung des Datenmodells in diesen Sichten werden verschiedene Painter verwendet. Hierfür wurden die folgenden Painter entwickelt:

- CSchnittAnsichtPainter
- CP_TextView
- CP_BrepPolygon
- verschiedene objektspezifische Painter

24.2.1 CSchnittAnsichtPainter

Die Klasse *CSchnittAnsichtPainter* wurde im Rahmen einer Diplomarbeit von Johann van Sciver ((Sci06)) entwickelt und löste eine Reihe von Paintern ab. Die Klasse kapselt einen Algorithmus zur Berechnung von Schnitten durch das Geometriemodell und der anschließenden Darstellung in einer 2D-View. Durch diesen Painter können Grundrisse aber auch Schnitte durch das Geometriemodell berechnet werden.

Klasse CSchnittAnsichtPainter	
Öffentliche Methoden	***Beschreibung***
CSchnittAnsichtPainter()	Konstruktor
~CSchnittAnsichtPainter()	Destruktor
BOOL Painter(CBauObjekt *pBauObjekt, ViewType Viewtyp)	Suche des passenden Painters
void Paint(ICViewEigenschaften *pViewEigen, CBauObjekt *pBauObjekt)	Objekt darstellen

Tabelle 24.5: Funktionen und Variablen der Klasse CSchnittAnsichtPainter

24.2.2 CP_BrepPolygon

Im Rahmen einer Studienarbeit wurde von Markus Weiler (Wei05a) eine 3D-View für das Programm GoCAD entwickelt. Die Darstellung in der 3D-View beruht auf der Graphikbibliothek OpenGL. Zu der 3D-View wurde auch der für die Darstellung benötigte Painter programmiert.

Klasse CP_BrepPolygon	
Öffentliche Methoden	***Beschreibung***
CP_BrepPolygon	Konstruktor
~CP_BrepPolygon	Destruktor
BOOL Painter(CBauObjekt *pBauObjekt, ViewType Viewtyp)	Suche des passenden Painters
void Paint(ICViewEigenschaften *pViewEigen, CBauObjekt *pBauObjekt)	Objekt darstellen

Tabelle 24.6: Funktionen und Variablen der Klasse CP_BrepPolygon

Der Painter *CP_BrepPolygon* greift auf das Geometriemodell der Bauobjekte zu. Die Methode `CP_BrepPolygon::Paint()` liest aus dem Brep-Modell die Kanten und Flächen aus und erzeugt daraus *CBrepPolygon*-Objekte, diese werden an die 3D-View übergeben.

Die Methode `CP_BrepPolygon::Painter()` überprüft zwei Punkte. Zuerst wird der übergebene Viewtyp auf Übereinstimmung mit dem Typ **CViewOpenGL** überprüft. Anschließend erfolgt die Kontrolle, ob die Geometrie des Bauobjekts über ein Brep-Modell beschrieben wird. Wenn beide Kontrollen positiv sind, gibt die Funktion TRUE zurück,

der Painter kann für die Kombination View/Bauobjekt verwendet werden, ansonsten wird FALSE zurückgegeben.

24.2.3 CP_TextView

Der Painter *CP_TextView* ist zuständig für den Typ *CGoCADTreeView*. In dem Viewtyp *CGoCADTreeView* werden die Bauobjekte mit mehreren ihrer Eigenschaften in einem Baum aufgelistet. Der Painter ist unabhängig vom Objekttyp, er liest die folgenden Eigenschaften aus dem Bauobjekt für die Darstellung in der View aus:

- Die Oberfläche (sie wird aus dem Brep-Modell ermittelt)
- Der Name des Objekts
- Der Name der Objektklasse
- Die ID des Objekts
- Die Speicheradresse des Objekts
- Die Position im Raum (x, y und z Koordinate des Objektmittelpunktes)

Klasse CP_TextView	
Öffentliche Methoden	***Beschreibung***
CP_TextView()	Konstrukor
~CP_TextView()	Destruktor
BOOL CP_TextView::Painter(CBauObjekt *pBauObjekt, ViewType Viewtyp)	Suche des passenden Painters
void Paint(ICViewEigenschaften *pViewEigen, CBauObjekt *pBauObjekt)	Objekt darstellen

Tabelle 24.7: Funktionen und Variablen der Klasse CP_TextView

Die Funktion `CP_TextView::Paint()` erzeugt ein *CGRA_TreeEintrag*-Objekt und übergibt dies anschließend der *CGoCADTreeView* für die Darstellung.

24.2.4 Weitere Painter

Die Painter *CP_BOEbene*, *CP_BOGerade*, *CP_BOKreis*, *CP_BOPunkt* sind die Painter für die Darstellung der graphischen Objekte Ebene, Gerade, Kreis und Punkt in der 2D-View. Da die graphischen Objekte für die Speicherung der Geometrie nicht das Brep-Modell verwenden, sondern jeder Objekttyp seine eigene Struktur besitzt, kann für diese Objekte kein allgemeiner Painter verwendet werden, jeder Objekttyp benötigt einen eigenen Painter, der auf die objektspezifische Geometrie zugreifen kann.

Klasse CP_BO...	
Öffentliche Methoden	***Beschreibung***
CP_BO...	Konstruktor
~CP_BO...	Destruktor
BOOL Painter(CBauObjekt *pBauObjekt, ViewType Viewtyp)	Suche des passenden Painters
void Paint(ICViewEigenschaften *pViewEigen, CBauObjekt *pBauObjekt)	Objekt darstellen

Tabelle 24.8: Funktionen und Variablen der Klasse CP_BO...

24.3 Organisation der Painterverwaltung

Der Painter ist zuständig für die Darstellung eines Objektes in einer View. Hierfür muss ein Painter dem Objekt und der View zugeordnet sein. Der einfachste Fall ist, wenn es für einen Viewtyp nur einen Painter gibt, der für die Darstellung aller Objekttypen verantwortlich ist.

Bei verschiedenen Objekten, erfolgt die Darstellung nicht nach dem allgemeinen Algorithmus durch Berechnung aus dem Datenmodell. Diese Objekte benötigen ihren eigenen viewspezifischen Darstellungsalgorithmus. Als Beispiel kann eine Tür angeführt werden, da in einer 2D-View ihre Darstellung nicht vom Geometriemodell abhängig ist, sondern durch ein Symbol erfolgt.

Durch die Speicherung der Darstellungsattribute beim Painter entstehen neue Painter/Objekte/-View Kombinationen. Ein Painter kann zuständig sein für die individuelle Zeichnung eines Objektes in einer bestimmten View oder auch für ein Objekt in einem speziellen Viewtyp. Eine dritte Möglichkeit ist die Zuordnung des Painters zu einem Objekt- und Viewtyp.

Aus den fünf aufgeführten Zuordnungen muss der richtige Painter für den benötigten Fall herausgesucht werden. Hieraus ergibt sich die Forderung nach der Erstellung einer Painterhierarchie, die die verschiedenen Zuständigkeiten berücksichtigt. Die unterste Ebene der Hierarchie enthält die allgemeinsten, die oberste Ebene die speziellsten Painter. Die folgende Liste enthält die Painterhierarchie, angefangen beim speziellsten und endend bei der allgemeinsten Kombination.

1. Objekt und View
2. Objekt und Viewtyp
3. Objekttyp und View
4. Objekttyp und Viewtyp

5. Allgemeine Painter

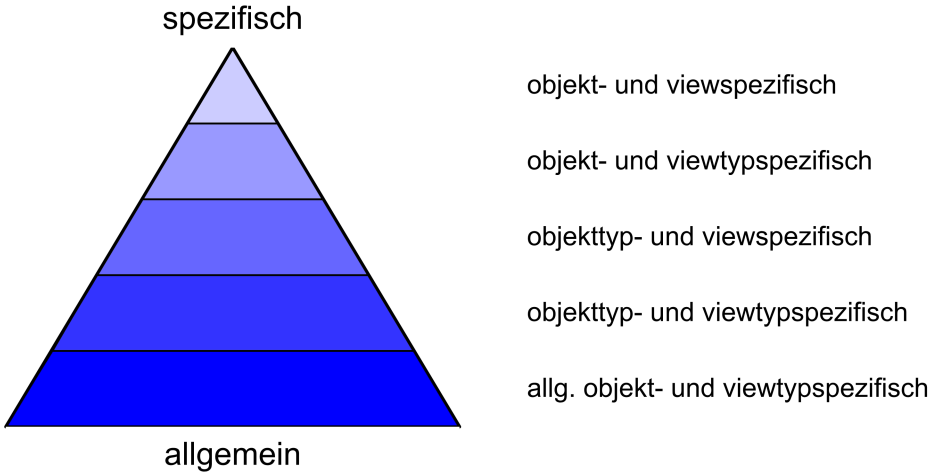


Abbildung 24.2: Darstellung der Painterhierarchie

Die Suche nach dem richtigen Painter beginnt beim speziellsten und endet bei der allgemeinsten Kombination.

24.4 Implementierung einer Painterverwaltung

Die zentrale Klasse zum Verwalten der Painter ist die Klasse *CPainterverwaltung*. Die Klasse *CPainterverwaltung* ist für die Suche nach dem richtigen Painter zuständig. Wenn ein Objekt neu gezeichnet werden soll, wird vom Document unter Angabe des Objektes und der View bei der Painterverwaltung nach dem korrekten Painter angefragt. Die Painterverwaltung durchsucht die Painterhierarchie, angefangen oben beim speziellsten Painter und endet unten beim allgemeinsten Painter.

Die einzelnen Painterebenen sind in verschiedenen Listen gespeichert. Diese Listen sind entweder Bestandteil der Painterverwaltung oder, wenn es sich um viewspezifische Painter handelt, Bestandteil der View.

24.4.1 Klasse CObjektViewPainterListe

Die oberste Ebene sind die objekt- und viewspezifischen Painter, deren Verwaltung wurde mit der Klasse *CObjektViewPainterListe* implementiert. Diese Klasse ist Bestandteil der View.

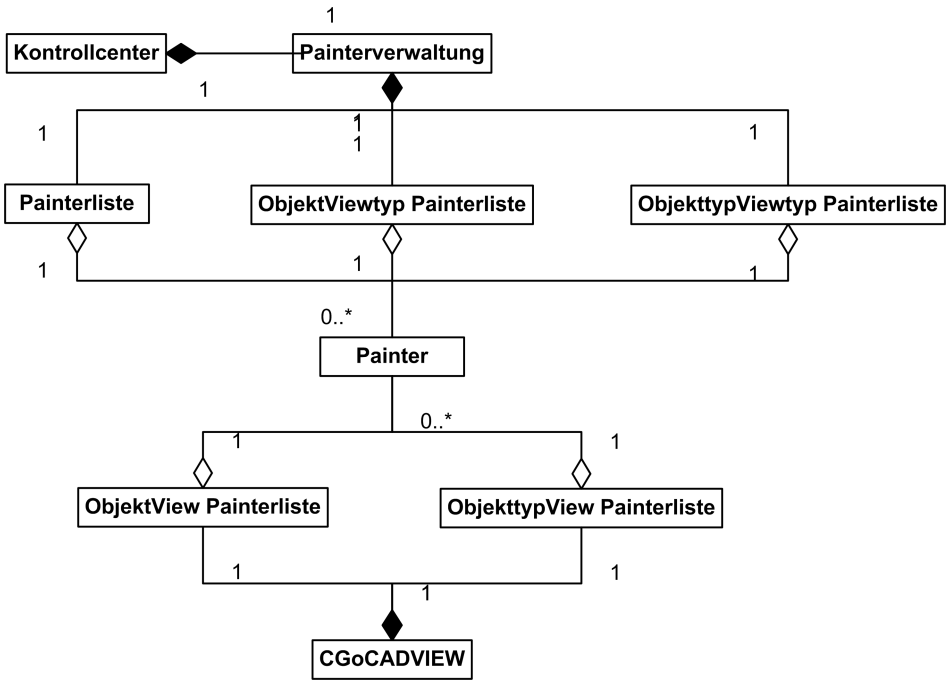


Abbildung 24.3: Darstellung des Aufbaus der Painterverwaltung

Für die Verwaltung der objekt- und viewspezifischen Painter werden zwei Schlüssel benötigt. Der erste Schlüssel ist das Objekt, dies wird über die ID des Objektes realisiert, da sich der Pointer auf das Objekt nach jedem Speichern ändert. Die ID ist eine Integerzahl und entspricht der Nummer des Eintrages im Document. Diese Nummer wird bei jedem Hinzufügen eines Objektes zum Dokument um eins hochgezählt, das Löschen eines Objektes verändert die Nummer nicht, somit wird sichergestellt, dass keine ID doppelt vorkommt. Der zweite Schlüssel ist die View, für die der Painter bestimmt ist. Dieser Schlüssel wird realisiert durch die Implementierung der *CObjektViewPainterListe* als Membervariable der View. Eine View verwaltet nur Painter, die auch für diese benötigt werden.

Die Liste wurde als assoziativer Container in Form einer Map implementiert. Die Map beinhaltet die Daten als Datenpaare in der Form pair<const Key, Data>. Der erste Teil des Paares ist der Schlüssel für die Identifikation der Daten, der zweite Teil sind die eigentlichen Daten. In Maps können Schlüssel nur einmal vorkommen, somit ist sichergestellt, dass es für ein Objekt nur einen zugehörigen Painter gibt. Die Map wurde mithilfe der STL erstellt.

Klasse CObjektViewPainterListe	
Öffentliche Methoden	***Beschreibung***
CObjektViewPainterListe()	Konstruktor
virtual ~CObjektViewPainterListe()	Destruktor

Fortsetzung Klasse CObjektViewPainterListe	
void SetPainter(CBauObjekt *pBauObjekt, CPainter* pPainter)	Painter einfügen
CPainter* GetPainter(CBauObjekt *pBauObjekt)	Painter abfragen
void PainterZuruecksetzen(CBauObjekt *pBauObjekt)	Painter entfernen
Private Attribute	***Beschreibung***
typedef std::map<int, CPainter*> Map	Map für die Painter
Map PainterMap	map für Painter
Map::iterator iter	Iterator

Tabelle 24.9: Funktionen und Variablen der Klasse CObjektViewPainterListe

Die Klasse *CObjektViewPainterListe* besitzt neben dem Konstruktor und Destruktor noch drei Methoden. Die Funktion `void SetPainter(CBauObjekt *pBauObjekt, CPainter* pPainter)` fügt einen neuen Painter in die Painterliste ein. `SetPainter()` fragt zuerst die ID des BauObjekts ab und überprüft auf das Vorhandensein eines zugehörigen Painters. Wenn ein zugehöriger Painter vorhanden ist, wird dieser zerstört und der Eintrag in der Liste durch den Neuen ersetzt. Ist kein Painter in der Liste vorhanden, so wird der Painter mit der ID als Schlüssel direkt in die Liste eingefügt.

`CPainter* GetPainter(CBauObjekt *pBauObjekt)` sucht in der Liste nach dem passenden Painter. Nach der Übergabe des BauObjekts wird von diesem die ID abgefragt und mit dieser als Schlüssel nach einem Painter gesucht. Wird ein Painter gefunden, so gibt die Funktion den Zeiger darauf zurück, andernfalls einen NULL-Pointer.

`Void PainterZuruecksetzen(CBauObjekt *pBauObjekt)` ist für das Entfernen eines Painters aus der Liste zuständig. Wird ein Painter für die Darstellung nicht mehr benötigt, kann dieser über diese Funktion aus der Liste entfernt werden. Der Liste wird ein Pointer auf das Objekt übergeben, die Funktion ermittelt die ID des Objektes, sucht den zur ID gehörenden Painter und entfernt diesen aus der Liste. Anschließend wird der Painter mittels delete zerstört.

24.4.2 Klasse CObjektViewtypPainterListe

Die zweite Ebene sind die objekt- und viewtypspezifischen Painter, die durch die Klasse *CObjektViewtypPainterListe* verwaltet werden. Diese Painter sind es, die für die Darstellung eines bestimmten Objektes in einer bestimmten View zuständig sind. Da diese Painter von dem Bauobjekt und der View abhängig sind, werden zwei Schlüssel benötigt. Als erster Schlüssel dient die ID des Objektes, als zweiter Schlüssel der Viewtyp.

Die Liste ist wiederum als Map implementiert. Das Datenpaar besteht aus pair<ID, Painterpointer>, der zweite Schlüssel kann nicht direkt in der Map gespeichert werden, da die

Map nur über einen Schlüssel verfügt. Da diese Painter den Views übergeordnet sind, kann der Key nicht wie bei der *CObjektViewPainterListe* einer View zugeordnet werden. Um dieses Problem zu lösen wurde der zweite Schlüssel direkt dem Painter zugeordnet. Die Painterklasse hat eine Variable vom Typ struct ViewTyp. In dieser Variablen wird der Viewtyp gespeichert, für den dieser Painter verwendet werden kann. Der Painter weiß somit, für welchen Viewtyp er verwendet werden kann.

Klasse CObjektViewtypPainterListe	
Öffentliche Methoden	***Beschreibung***
CObjektViewtypPainterListe()	Konstruktor
virtual ~CObjektViewtypPainterListe()	Destruktor
CPainter* GetPainter(CBauObjekt *pBauObjekt, ViewType Typ)	Painter abfragen
void SetPainter(CBauObjekt *pBauObjekt, ViewType Typ, CPainter* pPainter)	Painter setzen
void PainterZuruecksetzen(CBauObjekt *pBauObjekt, ViewType Typ)	Painter entfernen
Private Attribute	***Beschreibung***
Map PainterMap	Map für Painter
typedef std::multimap<int, CPainter*> Map	Map für Painter
Map::iterator iter	Iterator
Map::iterator iterAnfang	Iterator
Map::iterator iterEnde	Iterator

Tabelle 24.10: Funktionen und Variablen der Klasse CObjektViewtypPainterListe

Die Funktionalität der Klasse entspricht der der Klasse *CObjektViewPainterListe*. Der Hauptunterschied liegt im inneren Aufbau der Methoden `CPainter* GetPainter(CBauObjekt *pBauObjekt, ViewType Typ)` und `void SetPainter(CBauObjekt *pBauObjekt, ViewType Typ, CPainter* pPainter)`.

Die Übergabeparameter der Methode `SetPainter` sind das Bauobjekt, der Viewtyp und der Painter. Im ersten Schritt überprüft die Funktion die Liste auf das Vorhandensein eines Painters für das Bauobjekt und den Viewtyp. Ist ein Painter vorhanden, so wird dieser zerstört und der Eintrag in der Map gelöscht. Im zweiten Schritt wird ein Eintrag für das Paar(ID-Bauobjekt/Painterpointer) erzeugt und die Variable Viewtyp des Painter mit der Methode `SetViewTyp` auf den Wert Typ gesetzt. Die Funktion `GetPainter()` durchsucht die Liste nach dem zum Bauobjekt gehörenden Painter. Da für das Bauobjekt mehrere Painter in der Liste vorhanden sein können, wird anschließend mit der Funktion `GetViewTyp()` des Painters abgefragt und überprüft, ob der ViewTyp auch zum vorgegebenen Viewtyp passt.

24.4.3 Klasse CObjekttypViewPainterListe

Die dritte Ebene ist durch die Klasse *CObjekttypViewPainterListe* verwirklicht worden. Die Klasse *CObjekttypViewPainterListe* verwaltet die Painter, die für die Darstellung eines Bauobjekttypes in einer bestimmten Ansicht zuständig sind. Für die Verwaltung dieser Painterart werden zwei Schlüssel benötigt. Der erste Schlüssel ist die View, in der das Bauobjekt gezeichnet werden soll, der zweite ist der Objekttyp.

Die Painterliste in der Klasse *CObjekttypViewPainterListe* wurde wiederum mit einer STL-Map implementiert. Der erste Schlüssel wurde verwirklicht, indem die Klasse als Variable der Klasse *CGoCADView*, der Basisklasse aller Views, implementiert wurde. Eine View verwaltet dadurch nur die objekttypspezifischen Painter, die zu der View gehören. Der zweite Schlüssel wurde über den Klassennamen implementiert, der als Schlüssel für die Map dient. Mit dem Operator typeid (RTTI) wird eine Referenz auf das zum Objekt gehörende type_info Objekt ermittelt und von diesem anschließend mit der Funktion `name()` der Klassenname ermittelt.

Klasse CObjekttypViewPainterListe	
Öffentliche Methoden	***Beschreibung***
CObjekttypViewPainterListe()	Konstruktor
virtual ~CObjekttypViewPainterListe()	Destruktor
void PainterZuruecksetzen(CBauObjekt *pBauObjekt)	Painter entfernen
void SetPainter(CBauObjekt *pBauObjekt, CPainter *pPainter)	Painter einfügen
CPainter* GetPainter(CBauObjekt *pBauObjekt)	Painter abfragen
Private Attribute	***Beschreibung***
typedef std::map<CString, CPainter*> Map	Map für Painter
Map PainterMap	Map für Painter
Map::iterator iter	Iterator

Tabelle 24.11: Funktionen und Variablen der Klasse CObjekttypViewPainterListe

Die Funktionalität der Klasse *CObjekttypViewPainterListe* ist gleich der Funktionalität der Klasse *CObjektViewPainterListe*. Die Methoden haben die gleichen Übergabeparameter, die Objekttypermittlung findet erst innerhalb der Funktionen statt.

24.4.4 Klasse CObjekttypViewtypPainterListe

Die Klasse *CObjekttypViewtypPainterListe* ist für die Verwaltung der objekttyp- und viewtypabhängigen Painter zuständig, dies ist die vierte Ebene der Painterhierarchie. In dieser Klasse ist es möglich, mehrere Painter für den gleichen Objekt- und Viewtyp zu speichern. Einer der Painter ist als Standardpainter für die Kombination Objekt- und Viewtyp definiert.

Für die Auswahl des richtigen Painters für die Typ-/Viewkombination werden drei Schlüssel benötigt. Der erste Schlüssel ist der Name des Objekttyps, hierfür wird mit dem Operator typeid eine Referenz auf das type_info-Objekt ermittelt und über die Funktion `name()` den Namen der Klasse. Der zweite Schlüssel ist der Viewtyp. Jedem Painter ist der Viewtyp zugeordnet, für den er verwendet werden kann. Der dritte Schlüssel ist die Markierung als Standardpainter. Die Markierung als Standardpainter wurde durch eine eigene Map map<pointer, bool> realisiert. Als Schlüssel für diese Map dient der Pointer auf den Painter, das Ergebnis ist ein boolescher Wert, True, wenn es sich um den Standardpainter handelt, ansonsten False.

Klasse CObjekttypViewtypPainterListe	
Öffentliche Methoden	***Beschreibung***
CObjekttypViewtypPainterListe()	Konstruktor
virtual ~CObjekttypViewtypPainterListe()	Destruktor
void SetPainter(CBauObjekt *pBauObjekt, ViewType Typ, CPainter *pPainter)	Painter einfügen
CPainter* GetPainter(CBauObjekt *pBauObjekt, ViewType Typ)	Painter abfragen
void SetStdPainter(int PainterNr)	Painter als Standard setzen
CString GetStdPainterName()	Paintername abfragen
int GetStdPainterNr()	Painternummer abfragen
bool NameDoppelt(CString &Name)	Überprüfen auf doppelten Paternamen
void PropDialogFuellen(CPropertySheet *pPropSheet, int Nr)	Eigenschaftsdialog mit Einträgen füllen
void ListBoxFuellen(CListBox *pLBox, CString Name, ViewType Typ)	Listbox mit Einträgen füllen
Private Attribute	***Beschreibung***
typedef std::multimap<CString, CPainter*> Map	Map für Painter
Map PainterMap	Map für Painter
Map::iterator iter	Iterator
Map::iterator iterAnfang	Iterator
Map::iterator iterEnde	Iterator
typedef std::map<CPainter*, bool> StdMap	Map für Painter
StdMap StandardPainterMap	Map für Painter
StdMap::iterator stditer	Iterator
int m_StdPainterNr	Nummer für den Standardpainter
CString m_StdPainterName	Name des Standardpainters
ViewType m_vtTyp	Variable des Viewtypes
CString m_csBOTyp	Name des Bauobjekttypes

Tabelle 24.12: Funktionen und Variablen der Klasse CObjekttypViewtypPainterListe

24.4.5 Klasse CPainterListe

Die Klasse *CPainterListe* ist die unterste Stufe in der Painterhierarchie, sie ist die allgemeinste Liste der Painterverwaltung. Sie beinhaltet die Painter, die aus den DLLs in das Programm geladen werden. Die Klasse besitzt eine Liste in Form eines Vektors, in dem die Pointer auf die Painter gespeichert werden.

Für die Identifizierung des Painters werden zwei Schlüssel verwendet. Der erste Schlüssel ist das Bauobjekt, der zweite der Viewtyp. In der Funktion `CPainter* GetPainter(CBauObjekt *pBauObjekt, ViewType Viewtyp)` wird überprüft, ob der Painter für diesen Objekttyp und den Viewtyp zuständig ist. Bei Zuständigkeit wird ein Pointer auf diesen Painter für die weitere Verwendung zurückgegeben.

Klasse CPainterListe	
Öffentliche Methoden	***Beschreibung***
CPainterListe()	Konstruktor
virtual ~CPainterListe()	Destruktor
void Anhaengen(CPainter *pPainter)	Painter in Liste einfügen
CPainter* GetPainter(CBauObjekt *pBauObjekt, ViewType Viewtyp)	Painter aus Liste suchen
int GetAnzPainter()	Abfragen der Anzahl der Painter
void Leeren()	Einträge der Painterliste entfernen
void Loeschen()	Painter in der Liste löschen
Private Attribute	***Beschreibung***
typedef vector <CPainter*> Painterliste	Vektor für die Painter
typedef Painterliste::iterator iter	Iterator
Painterliste Liste	Liste für die Painter

Tabelle 24.13: Funktionen und Variablen der Klasse CPainterListe

24.4.6 Ablauf der Paintersuche

Wenn ein Objekt neu dargestellt wird, muss der entsprechende Painter herausgesucht werden. Die Suche fängt beim speziellsten Fall an und endet beim allgemeinsten Fall.

Die Suche beginnt mit einer Anfrage des Dokumentes bei der Painterverwaltung. Die Painterverwaltung ist Bestandteil der Klasse CMainFrame.

Der Mainframe besitzt eine Methode `CPainter* CMainFrame::GetPainter(CBauObjekt *pBauObjekt, CGoCADView *pView)`, die die Anfrage an die Painterverwaltung weiterreicht. Die Parameter der Funktion sind das BauObjekt, für das der Painter benötigt wird, und die View, in der das Objekt dargestellt werden soll. Die Painterverwaltung fragt zuerst über die Funktion `CGoCADView::GetPainterObjektspezifisch(CBauObjekt *pBauObjekt)` bei der View an, ob diese einen Painter besitzt, der für die Darstellung dieses Objektes in dieser View verwendet werden soll. Ist dies der Fall, wird ein Pointer auf diesen Painter zurückgegeben und die weitere Suche abgebrochen. Wenn kein geeigneter Painter vorhanden ist, wird ein NULL-Pointer zurückgegeben und die Suche fortgesetzt.

Ist kein objekt- und viewspezifischer Painter vorhanden, wird die Suche mit der Suche nach einem objekttyp- und viewspezifischen Painter fortgesetzt. Die Painterverwaltung fragt bei der ObjektViewtypListe nach einem entsprechenden Painter an, diese Liste ist Bestandteil

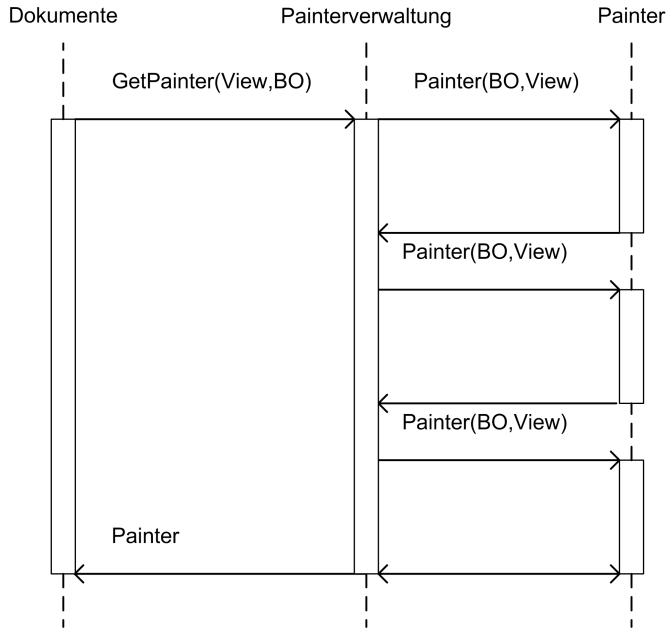


Abbildung 24.4: Ablauf der Paintersuche

der Painterverwaltung. Die Methode `CPainter* CObjektViewtypPainterListe::GetPainter(CBauObjekt *pBauObjekt, ViewType Typ)` benötigt das Bauobjekt und den Typ der View. Der Viewtyp wird mit der Methode `ViewType CGoCADView::GetViewTyp()` const von der View abgefragt. Der Rückgabewert ist ein Pointer auf den Painter oder ein NULL-Pointer, wenn kein passender Painter gefunden wurde.

Anschließend wird von der Painterverwaltung nach einem objekttyp-, viewspezifischen Painter gesucht. Für die Suche wird über die Methode `CPainter* CGoCADView::GetPainterObjekttypspezifisch(CBauObjekt *pBauObjekt)` bei der View angefragt, ob ein Painter für diesen Fall definiert ist. Der Übergabeparameter ist ein Pointer auf das darzustellende Bauobjekt. Der Rückgabewert ist wiederum ein Pointer auf den Painter, bzw. ein NULL-Pointer, wenn kein entsprechender Painter vorhanden ist.

In der nächsten Stufe wird nach einem Objekttyp- und viewtypspezifischen Painter gesucht. Hierfür fragt die Painterverwaltung bei der ObjekttypViewtypPainterListe an. Die Methode `CPainter* CObjekttypViewtypPainterListe::GetPainter(CBauObjekt *pBauObjekt, ViewType Typ)` der ObjekttypViewtypPainterListe bekommt einen Pointer auf das Bauobjekt und den Typ der View als Parameter übergeben. Zurückgegeben wird ein Pointer auf den gefundenen Painter oder ein NULL-Pointer wenn kein Painter vorhanden ist.

Sollte in diesen vier Listen kein speziell definierter Painter gefunden worden sein, findet die letzte Anfrage nach einem Painter bei der Painterliste statt. In dieser Liste sind alle aus

den DLLs geladenen Painter abgespeichert. Die Methode `CPainter* CPainterListe::GetPainter(CBauObjekt *pBauObjekt, ViewType Viewtyp)` bekommt einen Pointer auf das Bauobjekt und den Viewtyp übergeben. In dieser Liste sind auch allgemeine Painter, vom Objekttyp unabhängig, vorhanden. Diese Painter sind nur vom objektinternen Datenmodell abhängig, so dass sichergestellt ist, dass für jedes Objekt mindestens ein Painter vorhanden ist.

24.4.7 Änderung der Darstellung

Die Änderung der Darstellungsparameter geschieht über die Seiten Darstellung 1 Abb. 24.5 und Darstellung 2 Abb.24.6 des Eigenschaftsdialogs des Objekts. Dieser lässt sich nach Markieren des Objektes im Menü der rechten Maustaste aufrufen. Nach dem Aufruf des Dialoges wird der für das Objekt zuständige Painter gesucht. Anschließend wird für den weiteren Ablauf eine Kopie des Painters erzeugt und über die Methode `CPropertyPage* CPainter::GetPropertyPage()` wird die Eigenschaftsseite des Painters erzeugt. Über die Eigenschaftsseite können die Parameter für die verschiedenen Linien und Schraffuren geändert werden. Mit der Combobox am unteren Ende der Eigenschaftsseite kann die Art des Painters ausgewählt werden, es werden hierbei zwischen vier Arten unterschieden:

- objekt- und viewspezifischer Painter
- objekt- und viewtypspezifischer Painter
- objekttyp- und viewspezifischer Painter
- objekttyp- und viewtypspezifischer Painter

Standardmäßig ist bei Start des Dialoges der aktuelle Typ des Painters ausgewählt. Nach Beendigung des Eigenschaftsdialoges mit OK findet eine Überprüfung der Standardeigenschaften statt. Haben sich die Parameter des Painters geändert, werden die geänderten Parameter beim Originalpainter angepasst und anschließend die Kopie zerstört. Wurde der Typ des Painters geändert, so wird die Kopie als neuer Painter in die entsprechende Liste eingetragen.

Die Eigenschaftsseite Darstellungsseite 2 Abb. 24.6 wird von der Painterverwaltung erzeugt und in den Eigenschaftsdialog eingefügt. Die Seite ist in zwei Teile aufgeteilt. Im ersten Bereich sind drei Checkboxen, durch diese kann der

- objekt- und viewspezifische Painter

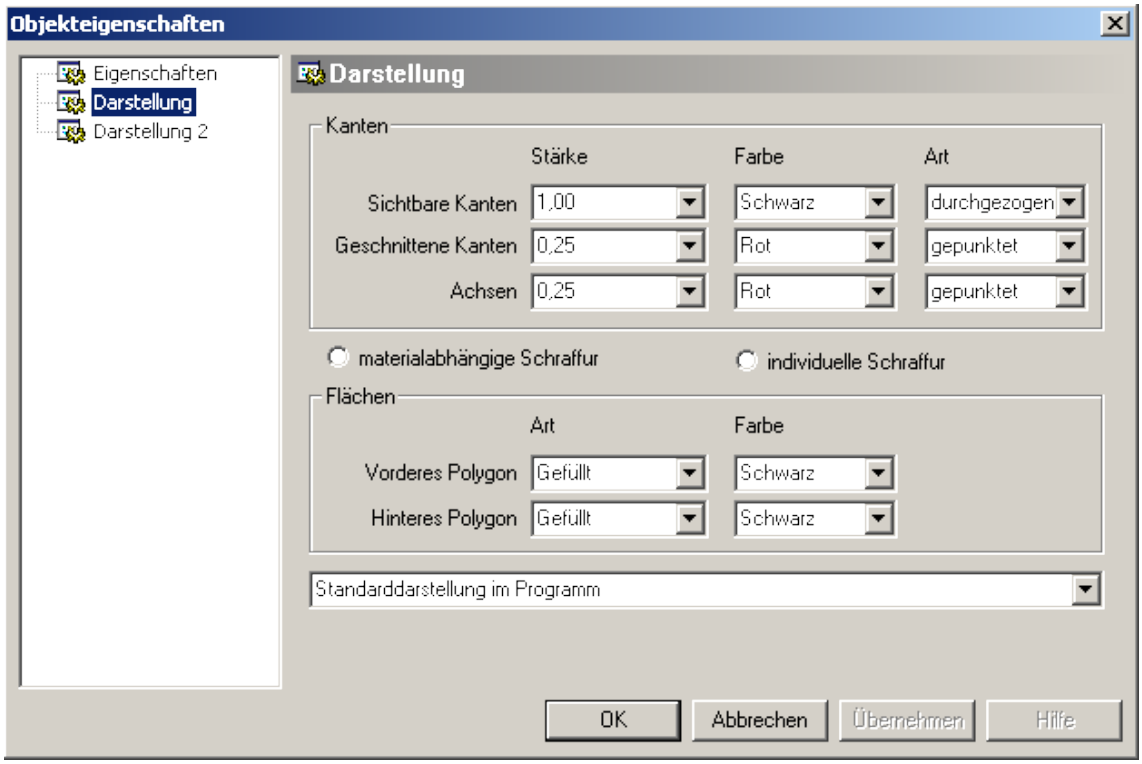


Abbildung 24.5: Dialog zur Einstellung der Darstellungseigenschaften

- objekt- und viewtypspezifische Painter
- objekttyp- und viewspezifische Painter

zurückgesetzt werden. Diese Painter werden gelöscht und aus den entsprechenden Painterlisten entfernt. Damit wird die Darstellung des Objektes auf den in der Hierarchie folgenden Painter gesetzt. Der zweite Teil der Eigenschaftsseite betrifft die viewtyp- und objekttypspezifischen Painter. In einer Listbox werden alle für das Objekt vorhandene Painter aufgeführt. Der aktuelle Standardpainter wird in einem darüber angeordneten Textfenster angezeigt. Unterhalb der Listbox sind drei Knöpfe angeordnet, mit diesen kann der aktuell ausgewählte Painter als Standardpainter gesetzt, gelöscht oder bearbeitet werden. Der Knopf „Bearbeiten“ erzeugt einen neuen Eigenschaftsdialog, in dem die Darstellungseigenschaften des ausgewählten Painters geändert werden können.

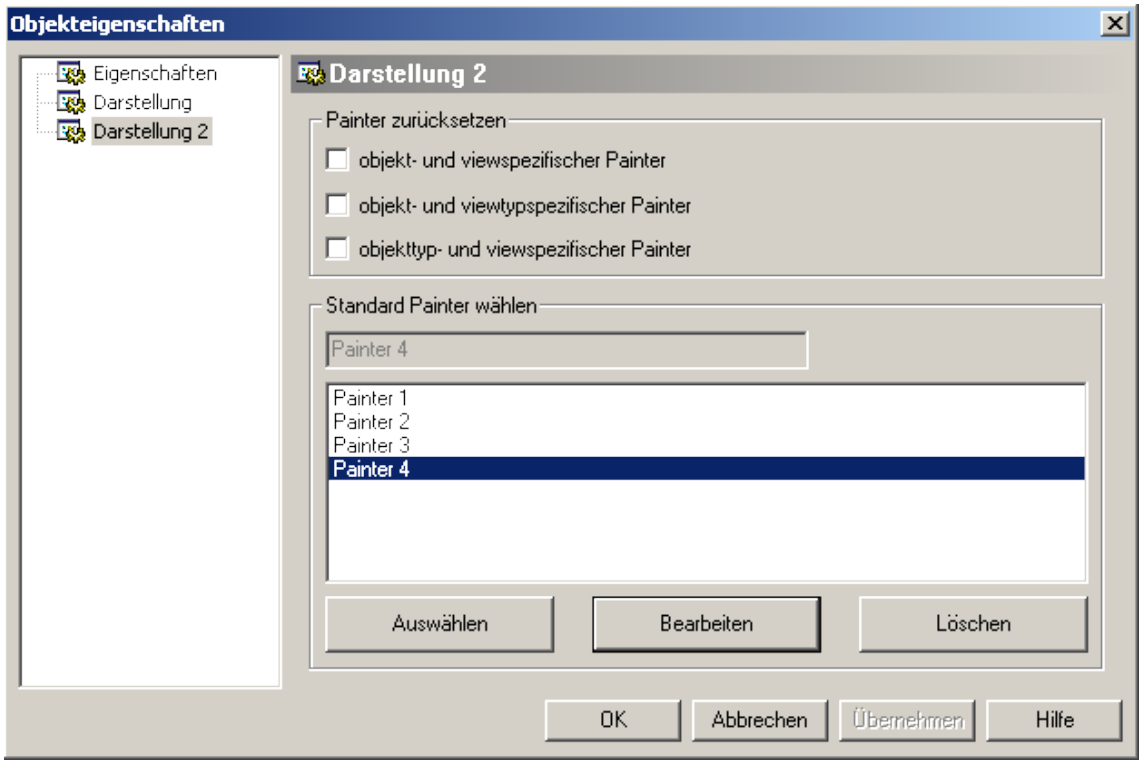


Abbildung 24.6: Dialog zur Painterauswahl

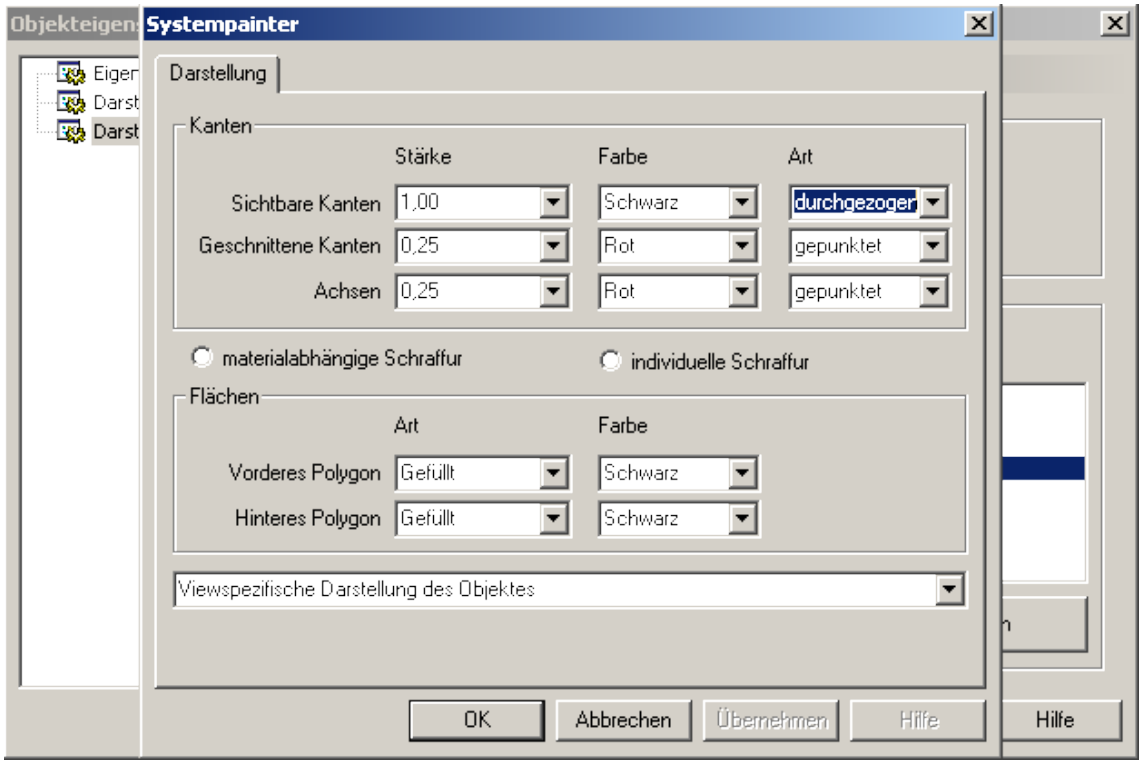


Abbildung 24.7: Dialog zur Bearbeitung eines Painters

Teil V

Zusammenfassung

25 Zusammenfassung

Der Fortschritt in der Hard- und Softwareentwicklung der letzten Jahre sorgte für große Veränderungen bei CAD-Anwendungen. Aus einfachen Zeichenprogrammen wurden bauteilorientierte, intelligente Anwendungen. Aus der Kombination von CAD mit anderen Programmen, wie z. B. FEM-Anwendungen, entwickelte sich Computer Aided Engineering (CAE). Viele Schlagworte wie Objekte, Intelligenz und Gebäudemodell prägen die Broschüren der Programmhersteller. In dieser Arbeit werden die Anforderungen, Bestandteile und die Architektur moderner objektorientierter 3D-CAD-Programme im Bauwesen beschrieben.

Anhand des Programms Vicado (Kapitel 7) wird die Funktionalität einer modernen Bau-CAD-Anwendung erläutert. Dies umfasst nicht nur die Funktionen zur Konstruktion des Bauwerkmodells und die Erstellung der Pläne, sondern auch die Möglichkeiten der Auswertung und Visualisierung.

Auf den Aufbau und die Elemente der graphischen Oberfläche eines CAD-Programmes wird in Kapitel 8 eingegangen. Die einzelnen Bestandteile der Oberfläche werden vorgestellt und erläutert.

Die Grundzüge eines Datenmodells für ein Bau-CAD-Programm werden in Kapitel 10 behandelt. Hierbei wird auf die Objekthierarchie für ein Datenmodell und die Grundfunktionalität der Objekte eingegangen.

Die Referenzen (Kapitel 12 und 20) ermöglichen die Verknüpfung einzelner Bauteile untereinander. Änderungen an einem Bauteil bewirken ein automatisches Anpassen der verknüpften Bauteile. Hierdurch wird den Bauteilen und dem Programm eine gewisse Intelligenz verliehen.

Mit den Rezepten (Kapitel 13) wird eine weitere praktische Anwendung der Referenzen vorgestellt. Die Rezepte sind eine Methode neue Bauteile mit Hilfe abgespeicherter Regeln zu erzeugen. Die Geometrie der neuen Bauteile wird hierbei durch im Rezept abgespeicherte Parameter und existierende Bauteile bestimmt.

Die Möglichkeiten des Austausches der Daten von CAD-Programmen werden in Kapitel 15 aufgezeigt. Hierbei wird unterschieden zwischen Datenformaten für den Modellaustausch (IFC), Formaten zum Austausch und zur Weiterverarbeitung der Pläne (DWG, DXF) bzw. zum Weiterreichen der Pläne (PDF).

Die mehrschichtige Architektur eines modernen CAD-Programmes wird in Kapitel 16 erläutert. Sie ermöglicht eine flexible Erweiterung der Anwendung mit weiteren Bauteilen, aber auch den Einbau von weiteren neuen Sichten oder spezifischen Anwendungen. Der Kern einer CAD-Anwendung beinhaltet die Datenhaltung, die Programmsteuerung und die Darstellung. Als weitere Schicht kommen die Bauteil- und Anwendungsebene hinzu.

Auf einem allgemeinen Datenmodell können unterschiedliche Anwendungen aufbauen. Diese werten das Datenmodell unter den verschiedensten Gesichtspunkten wie Architektur, Ingenieurbau, oder FEM aus.

Die Anwendung des Model-View-Controller Konzeptes als Grundstock für ein CAD-Programm wird in Kapitel 17 betrachtet. Dieses Konzept trennt die Daten und Ansicht und bildet die gute Grundlage für CAD-Programme mit einem digitalen Bauwerksmodell.

Das Builderkonzept aus Kapitel 18, trennt die Objekterzeugung vom Objekt. Durch die Trennung der Konstruktion vom Objekt, kann die Konstruktion vereinheitlicht werden. Die verschiedensten Bauteile werden mit Builder durch ein einheitliches Schema erzeugt.

Das Painterkonzept in Kapitel 19 dient zur Darstellung der Bauteile in den verschiedensten Sichten; es trennt den Darstellungsalgorithmus vom Objekt. Der Painter ermittelt aus dem Datenmodell des Bauteils die für die Sicht entsprechende Darstellung. Eine zentrale Verwaltung der Painter ermöglicht die flexible Gestaltung der Darstellung des Datenmodells eines CAD-Programms. Über eine Painterverwaltung kann die Darstellung des Datenmodells für eine einzelne Ansicht oder die gesamte Anwendung gesteuert werden.

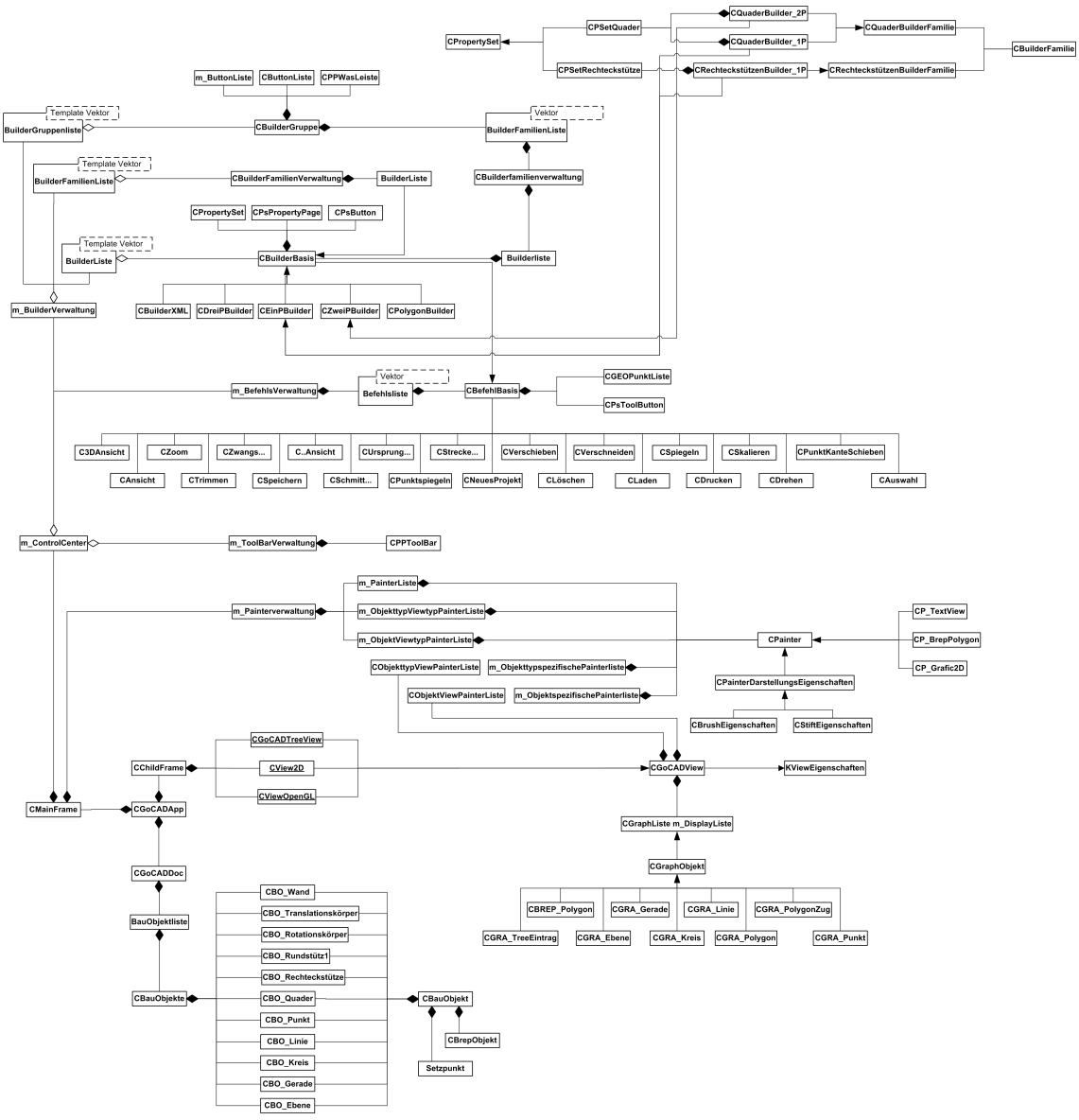
Im Rahmen dieser Arbeit wurde aufgezeigt wie moderne und zukünftige CAD-Programme aufgebaut sind. Die Programme bieten mehr als nur die Erstellung von Zeichnungen, durch die Schnittstellen dienen sie zur Eingabe und Auswertung vieler anderer Anwendungen. Die graphische Oberfläche bietet viele Möglichkeiten zur Unterstützung des Anwenders bei der Arbeit. Der Einzug moderner Technologien steht in der Praxis jedoch hinter der Entwicklung zurück, da diese oftmals tiefe Eingriffe in das bestehende Programm erfordern.

Teil VI
Anhang

A Lebenslauf

seit 2006	Mitarbeiter im Ingenieurbüro Karl Jagsch, Kaiserslautern
2001 - 2006	Wissenschaftlicher Mitarbeiter im Fachgebiet EDV-gestütztes Entwerfen, Berechnen und Konstruieren im Bauingenieurwesen (Bauinformatik) an der TU Kaiserslautern
1996 - 2001	Studium Bauingenieurwesen an der Universität Kaiserslautern, Vertiefungsfächer Stahlbau, Statik und Grundbau
1996 - 2000	Studentische Hilfskraft am Fachgebiet Bodenmechanik und Grundbau
1992 - 1996	Studium Bauingenieurwesen an der Fachhochschule Rheinland-Pfalz, Abt. Kaiserslautern, Studienschwerpunkt Konstuktiver Ingenieurbau
1991 - 1992	Wehrdienst im Mannheim, Mayen und Daun
1989 - 1991	Fachoberschule Neunkirchen
1983 - 1989	Kreisrealschule Homburg
1982 - 1983	Hauptschule in Waldmohr
1978 - 1982	Grundschule in Waldmohr
1972	geboren in St. Ingbert

B Klassendiagramm von GoCAD



Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AI90] Alfred Iwainsky, Prof. Dr. sc. n.: *Computergrafik in CAD/CAM-Prozessen.* Verlag Technik Berlin, 1990

[BHK04] Born, Marc ; Holz, Eckhard ; Kath, Olaf: *Softwareentwicklung mit UML 2.* Addison-Wesley, 2004

[Fle04] Flesch, Florian: *Entwicklung und Implementierung Geometrischer Grundklassen in GoCAD unter Verwendung von C++*, TU Kaiserslautern, Diplomarbeit, 2004

[GHJV04] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, 2004

[Grä89] Grätz, Joachim-F.: *Handbuch der 3D-CAD-Technik.* Siemens AG, 1989

[Ham00] Hamwi, Dr.-Ing. S.: *Entwurf, Realisierung und Test eines Objektorientierten CAD-Datenmodells für die Tragwrksplanung*, Universität Kaiserslautern, Diss., Februar 2000

[Hec98] Heck, Dr.-Ing. P.: *Ein objektorientiertes CAD-Modell für die raum- und bauteilorientierte Bearbeitung von Gebäuden in der Vorplanung*, Universität Kaiserslautern, Diss., 1998

[hoa06] *HOAI, Honorarordnung für Architekten und Ingenieure.* 6. Kohlhammer, 2006

[Hor94] Horst, Kretschmar: *Computergestützte Bauplanung.* Verlag für Bauwesen, 1994

[HS93] Horn, Erika ; Schubert, Wolfgang: *Objektorientierte Software-Konstruktion, Grundlagen - Modelle - Methoden - Beispiele.* Carl Hanser Verlag, 1993

[Jen03] Jenter, Jochen: *Entwicklung eines Boundary Representation Modells am Beispiel eines Quaders unter Verwendung von C++*, TU Kaiserslautern, Diplomarbeit, 2003

[Kin98] Kinzler, Andreas: Klassenarbeit, MFC-Programmierung, Teil 3: das Document-View-Konzept. In: *c't* (1998), April

[Oes97] Oestereich, Bernd: *Objektorientierte Softwareentwicklung: Analyse und Design.* R. Oldenbourg Verlag, 1997

[OHJ+90] Oestereich, Bernd ; Hruschka, Dr. P. ; Josuttis, Nicolai ; Kocher, Dr. H. ; Krasemann, Dr. H. ; Reinhold, Markus: *Erfolgreich mit Objektorientierung.* Oldenburg Verlag, 1990

[Rie97] Riehle, Dirk: *Entwurfsmuster für Softwarewerkzeuge, Gestaltung und Entwurf von Anwendungen mit graphischer Benutzeroberfläche.* Addison-Wesley, 1997

[Sch94] Scheer, Christian: Volksbibliotheken, Borlands OWL 2.0 versus Microsofts MFC 2.5. In: *c't* (1994), Juli

[Sci06] van Sciver, Johann: *Entwicklung und Implementierung von Algorithmen zur Berechnung und Visualisierung von Schnitten durch das BRep-Modell des CAD-Programmes GoCAD*, TU Kaiserslautern, Diplomarbeit, 2006

[Wei05a] Weiler, Markus. *Entwicklung einer 3D-View unter Verwendung von OpenGL.* 2005

[Wei05b] Weiler, Markus: *Import und Export in CAD Systemen*, TU Kaiserslautern, Diplomarbeit, 2005

Sachregister