Prüfer:	o. Prof. Hon.-Prof. Dr. D. Roller
Betreuer:	Dipl.-Inform. Monika Bihler
Beginn am:	25. November 1994
Beendet am:	24. Mai 1995
CR-Nummer:	H.2, H.3, J.6

Studienarbeit-Nr. 1428

Analyse von OODBMS für die technische Modellierung

Annette Maile

Universität Stuttgart
Institut für Informatik
Lehrstuhl Grundlagen der Informatik
- Graphische Systeme -

Kurzfassung

Im Rahmen des Projekts POWER (**P**roduct modelling in **o**bject-oriented Databases **w**ith **e**fficient Methods for **R**etrieval) werden die Anforderungen an eine integrierte, umfassende Produktdatenmodellierung untersucht und richtungsweisende Konzepte für ganzheitliche, bereichsübergreifende Informationssysteme im CAD/CAM-Bereich entwickelt.

Im Rahmen dieser Studienarbeit werden objektorientierte Datenbankmanagement-systeme (OODBMS) daraufhin untersucht, inwieweit sie für die technische Modellierung geeignet sind.
Zunächst werden die Anforderungen an eine Produktmodellierung untersucht. Aufgrund der Komplexität technischer Objekte besteht die Notwendigkeit, Objekte nicht nur geometrisch zu modellieren, wie es bei herkömmlichen CAD-Systemen der Fall ist. Zusätzlich müssen die technischen Aspekte modelliert werden, wie z.B. Beziehungen und Funktionen zu anderen Objekten. Ein Lösungsansatz bietet das Konzept der technischen Modellierung, das hier vorgestellt wird.
Anhand eines Szenarios aus dem Bereich der technischen Modellierung wird eine Baugruppe konkret modelliert, dabei ist das Szenario stellvertretend für technische Anwendungen anzusehen. Die Baugruppe umfaßt eine Metallscheibe mit einer durchgehenden, mittigen Bohrung und einen symmetrischen, konzentrischen Bohrkranz, sowie eine zugehörige Achse. Trotz der Einfachheit der Baugruppe existieren Beziehungen und Abhängigkeiten der Objekte untereinander.
Für die Verwaltung der aus dem Produktmodell entstehenden komplexen Daten werden geeignete Datenbanksysteme benötigt.

Im darauffolgenden Kapitel erfolgt eine Einführung in Datenbanksysteme (DBS). Vorgestellt werden die Konzepte von relationalen und objektorientierten DBS. Relationale Datenbanken haben sich im Verwaltungsbereich bewährt. Im technischen Bereich jedoch, wo es sich meist um komplexe Objkete handelt, reichen sie zur Beschreibung dieser nicht aus. Deshalb wird nun eine neue Generation von DBS - die objektorientierten DBS - untersucht. Sie sollen die Unzulänglichkeiten der relationalen DBS beheben.

In Kapitel 4 werden die Anforderungen der technischen Modellierung an Datenbank-systeme anhand des Szenarios ermittelt. Die DBS sind u.A. auf Datenstruktur-möglichkeiten, Versions- und Variantenbildung, Darstellung von Beziehungen von Objekten, etc. zu untersuchen.

Im Anschluß daran werden diese erarbeiteten Anforderungen in den beiden ausgewählten Datenbanken - Postgres und OBST - evaluiert. Das Ergebnis dieser Analyse ergibt die Bewertung der Eignung und Leistungsfähigkeit der beiden DBS für die technische

Modellierung. Abschließend werden die beiden Datenbanken gegenübergestellt und Stärken und Schwächen der Systeme herausgestellt.

Die Durchführung der Arbeit erfolgt an HP-Workstations der Serie 9000/700. Zur Verfügung standen die Versionen Postgres 4.01 und OBST 3-4.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Markt- und Wettbewerbsverhältnisse ändern sich oft überraschend. Daher ist es für ein Unternehmen wichtig, schnell und flexibel auf diese Änderungen reagieren, oder besser schon im Vorfeld agieren, zu können. Viele spezifische Kundenbedürfnisse erfordern schnelle und kostengünstige Produktion von Gütern in vielen Varianten.

Durch steigende Variantenvielfalt, kürzere Produktlebenszyklen und der Erfordernis kürzerer Lieferzeiten besteht die Notwendigkeit einer flexiblen Fertigung [Zah93a]. Aus diesen Anforderungen resultiert ein komplexes Produktionssystem, das automatisiert und integriert sein muß. Daten aller Bereiche müssen erfaßt und in einem globalen Datenbanksystem zusammengefaßt werden.

Einige Teilbereiche im Produktionsprozeß werden von Computern unterstützt, wie **CAD** (**C**omputer **A**ided **D**esign), **CAP** (**C**omputer **A**ided **P**laning), **CAQ** (**C**omputer **A**ided **Q**uality Assurance). Jedoch hat jeder Bereich seine eigenen Systeme und eigene Datenbestände, was zu Redundanzen führt. Auch der Austausch der Daten aus verschiedenen Systemen ist meist schwierig, da die einzelnen Systeme nicht aufeinander abgestimmt sind. Ein Lösungsansatz hierfür ist **CIM** (**C**omputer **I**ntegrated **M**anufacturing). Das Ziel ist die Integration aller Informationen aus allen Bereichen eines Unternehmens.

Information ist ein wichtiger Produktionsfaktor. Die Ziele des strategischen Einsatzes von Informationstechnologien sind [Zah93b]:

- economies of scale: die Produktivitätssteigerung in der Verwaltung und der Produktion
- economies of speed: die Erhöhung der Reaktionsgeschwindigkeit des gesamten Unternehmens
- economies of scope: die Steigerung der Innovationsfähigkeit eines Unternehmens

Um die Gesamtheit aller Informationen verwalten zu können, werden geeignete Datenbanksysteme benötigt. Im verwaltungstechnischen Bereich haben sich relationale Datenbanken durchgesetzt. Jedoch im technischen Bereich, wo es sich meist um komplexe Objekte handelt, genügen diese nicht, da sie nur einfache Datentypen verwalten können. Deshalb soll nun eine neue Generation von Datenbanken, die objektorientierten Datenbanken, auf die Eignung zur Verwaltung von Daten im technischen Bereich, also im Produktionsbereich, untersucht werden.

Zur Umsetzung aller zu einem Produkt gehörenden Daten in Datenbanken besteht die Notwendigkeit der Entwicklung eines Produktmodells. In diesem Modell werden die Daten neutral beschrieben. Mit der Einführung von CAD-Systemen konnten die

Produktdaten geometrisch modelliert werden. Dies führte jedoch zu Insellösungen, da sie nicht an andere Systeme angebunden werden konnten. Außerdem ließen sich die Zusammenhänge und Beziehungen der Objekte nicht darstellen. Das Ziel ist ein integratives, durchgängiges Produktmodell, auf welches alle Bereiche eines Unternehmens zugreifen können. Dieses soll mit Hilfe der technischen Modellierung realisiert werden.
Der Vorteil der technische Modellierung besteht darin, daß die Semantik der Objekte erfaßt und im Produktmodell beschrieben wird. Die Objekte werden nicht nur als geometrische Körper wie z.B. als Zylinder angesehen, sondern vielmehr als Bestandteil des Produkts mit speziellen Eigenschaften und Funktionen. Außerdem werden Beziehungen zwischen den Objekten durch die sogenannten technisch-funktionalen Abhängigkeiten mit einbezogen. Das Produktmodell setzt sich aus drei Partialmodellen, dem technischen Modell, dem geometrischen Modell und dem technologischem Modell, zusammen. Die technische Modellierung wird im nächsten Kapitel eingehend erläutert.
Ein weiterer Vorteil der Modellierung ist, daß Fehler früh erkannt und behoben werden können. Fehler, die erst in der Implementierungsphase erkannt werden, sind sehr teuer. Außerdem kann das Modell Grundlage für Rapid-Prototyping sein. Rapid-Prototyping dient der schnellen Erstellung eines Prototypen, um die Probleme in der Arbeitsumgebung festzustellen. Dabei wird auf Benutzerfreundlichkeit und Robustheit des Systems verzichtet. Es soll lediglich Probleme und eventuelle Fehler des Modells aufzeigen.

Um einen reibungslosen Ablauf im Produktionsprozeß gewährleisten zu können, müssen die Produktdaten in einem Produktmodell modelliert werden. Für die Verwaltung aller anfallenden Daten müssen geeignete Datenbanksysteme zur Verfügung stehen. In den folgenden Kapiteln wird zunächst die technische Modellierung vorgestellt und anhand eines Szenarios eine kleine Baugruppe konkret modelliert. Dabei soll das Szenario stellvertretend für technische Anwendungen stehen. Darauf folgt eine Einführung in Datenbanken. Zum einen werden die relationalen DBMS (Datenbankmanagement-systeme) und zum anderen die objektorientierten DBMS dargestellt. Im darauffolgenden Kapitel werden die speziellen Anforderungen, die aus dem Szenario resultieren, an die DBMS herausgearbeitet. Danach sollen diese Anforderungen in den ausgewählten DBMS - Postgres und OBST - validiert werden. In einer Zusammenfassung wird die Eignung beider DBMS für die technische Modellierung beurteilt.

2 Technische Modellierung

Im zweiten Kapitel wird die technische Modellierung vorgestellt. Zunächst werden die Begrifflichkeiten der technischen Modellierung - Produktmodell, technische Objekte, technisch-funktionale Abhängigkeiten, technische Operationen - erläutert. Danach wird anhand eines Szenarios eine Baugruppe konkret modelliert.

2.1 Produktmodell

Für die technische Modellierung ist es notwendig, daß alle Daten, die im Produktionsprozeß anfallen, in einem Produktmodell zusammengefaßt werden. Zuerst werden die technischen Objekte entworfen und deren funktionaler und struktureller Aufbau ermittelt. Im Verlauf der Modellierung werden die Objekte mehr und mehr konkretisiert und verfeinert [HS93].
Das Produktmodell wird sukzessive aus folgenden Partialmodellen erstellt:

- **Technisches Modell**

  Das technische Modell ist die Abbildung eines Realweltausschnittes in einem Modell. Im Top-Down-Entwurf werden die Objekte von der Baugruppe bis hin zum Einzelteil beschrieben. Außerdem werden die Beziehungen und Abhängigkeiten zwischen den Objekten festgehalten.

- **Geometrisches Modell**

  Hier werden Formen (Quader, Zylinder etc.), Maße (Breite, Höhe, Tiefe etc.) und Topologie (Lage, Orientierung) der Objekte beschrieben.

- **Technologisches Modell**

  Im technologischen Modell werden physikalische Eigenschaften (Material, Gewicht etc.), Toleranzen, Oberfläche und Werkstoff der Objekte dargestellt.

Die Daten müssen zwischen den Partialmodellen transformiert werden, damit die Beschreibung der Objekte einheitlich erfolgt. Die Abhängigkeiten müssen erkannt und registriert werden. Das technische Modell hat die größte Bedeutung, denn in ihm wird das Objekt zuerst modelliert und daraus die beiden anderen Modelle abgeleitet und ergänzt.

2.2 Technische Objekte

Die Beschreibung der technischen Objekte erfolgt durch ihre Eigenschaften und Beschaffenheiten, sowie ihren Beziehungen zu anderen Objekten. Die Merkmale eines technischen Objekts sind z.B. seine geometrische Form, physikalische und chemische Eigenschaften, und seine Funktion.
Technische Objekte sind Baugruppen und Einzelteile einer Maschine oder Anlage, die in einer hierarchischen Struktur angeordnet sind. Baugruppen können wiederum aus Baugruppen, sogenannten Unterbaugruppen, oder Einzelteilen bestehen.
Einzelteile sind vorgefertigte Teile, wie z.B. Normteile oder Kaufteile, oder sie bestehen aus extra angefertigten Teilen, sogenannten Zeichenteilen. Ein Einzelteil besteht aus Hauptelementen, die sich wiederum aus Funktions- und Nebenelementen (siehe Abbildung 2.1) zusammensetzen. Im Top-Down-Entwurf werden die Baugruppen, Unterbaugruppen und Einzelteile ausgehend von der Maschine oder Anlage entworfen.

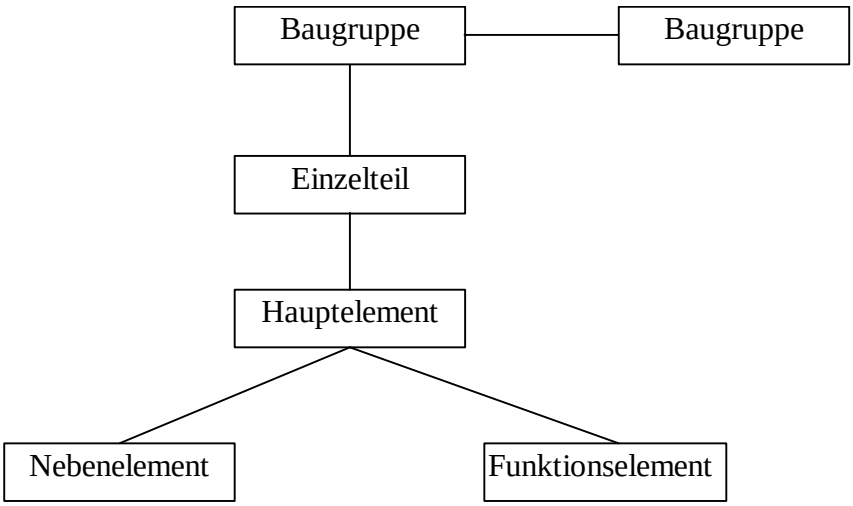


Abbildung 2.1: *Allgemeiner Aufbau einer Baugruppe*

- **Hauptelement**

  Das Hauptelement modelliert ein Formelement eines Einzelteils mit den dazugehörigen Informationen (Werkstoff, Toleranzen,...).

- **Funktionselement**

  Das Funktionselement repräsentiert den Bereich eines Einzelteils, der dessen Funktion realisiert (Übertragung eines Drehmoments,...).

- **Nebenelement**

  Das Nebenelement beschreibt das Hauptelement näher.

Bei den technischen Objekten unterscheidet man zwischen materiellen und immateriellen Objekten. Materielle Objekte sind Werkstücke, die bearbeitet werden können. Immaterielle Objekte stellen beispielsweise Vorgänge, wie Bohrungen, oder fiktive Objekte, wie z.B. ein Bohrkranz, dar.

Die technischen Objekte werden in einer Objektstruktur, dem technischen Partialmodell, angeordnet und über genau definierte Operationen, wie z.B. Erzeugen, Verändern, Löschen, manipuliert. So werden die Objekte nicht isoliert, sondern in Zusammenhang zu anderen betrachtet. Die Abhängigkeiten der Objekte untereinander können somit erkannt und explizit erfaßt werden. Diese Abhängigkeiten nennt man technisch-funktionale Abhängigkeiten.

2.3 Technisch-funktionale Abhängigkeiten

Die technisch-funktionalen Abhängigkeiten beschreiben die Beziehungen der technischen Objekte untereinander oder die Beziehungen der Objekte zu den technischen Operationen. Grundlage hierfür sind die Funktionen bzw. Teilfunktionen der Baugruppen und die Restriktionen, die durch Umgebungsvariablen aus der Konstruktionslogik oder physikalischen Prinzipien gegeben sind.

> **Beispiel**: Achse und dazugehörige Bohrung
>
> Der Außendurchmesser der Achse muß gleich dem Innendurchmesser der Bohrung sein. Bei Änderung des Außendurchmessers muß es möglich sein, daß der Innendurchmesser automatisch angepaßt wird und umgekehrt.

2.4 Technische Operationen

Die technischen Operationen dienen hauptsächlich zum Erzeugen, Verändern, Löschen und Positionieren von technischen Objekten. Hierbei müssen Abhängigkeiten der Objekte berücksichtigt werden und Konsistenzprüfungen vorgenommen werden.

Die technischen Operationen sind im wesentlichen:

- **Erzeugen**

  Beim Erzeugen eines technischen Objekts, das von einem anderen abhängt, müssen z.B. logische Abfragen, ob das andere Objekt schon erzeugt wurde, möglich sein.

- **Verändern**

  Bei der Änderung eines Parameters müssen auch alle anderen Parameter, die in Beziehung zu diesem Parameter stehen, entsprechend geändert werden.

- **Löschen**

  Beim Löschen eines Objekts sind alle Objekte, die von ihm abhängen, ebenfalls zu löschen.
- **Positionieren**

  Positionieren bedeutet, daß ein technisches Objekt an eine bestimmte Stelle eines Einzelteils plaziert wird.

- **Orientieren**

  Die Orientierung beschreibt die Position eines Objekts. Auch hierbei ist zu beachten, daß bei einer Änderung der Koordinaten eines Objekts alle Koordinaten der abhängigen Objekte geändert werden.

Nach dieser Einführung in die technische Modellierung mit allen dazugehörigen Komponenten und Zusammenhängen soll nun anhand eines Szenarios eine Baugruppe konkret modelliert werden.

2.5 Das Szenario

Zu modellieren ist eine einfache Baugruppe. Sie soll hier stellvertretend für technische Anwendungsbereiche stehen. Die Baugruppe besitzt trotz ihrer Einfachheit Beziehungen und Abhängigkeiten zwischen den einzelnen technischen Objekten.
Im Verlauf der Modellierung werden die einzelnen Komponenten der Baugruppe ausgehend von einer groben Beschreibung immer detaillierter dargestellt.

2.5.1 Entwicklung des technischen Partialmodells

Die Baugruppe umfaßt zwei Teile:

1. Eine Metallscheibe, die in ihrem Mittelpunkt eine durchgehende Bohrung sowie einen symmetrischen, konzentrischen Bohrkranz aufweist. Die Anzahl der Bohrungen auf dem Bohrkranz ist variabel. Die Tiefe der Bohrungen muß kleiner sein als die Dicke der Scheibe. Der Radius ist bei allen Bohrungen gleich.

2. Eine Achse, deren Durchmesser gleich dem Radius der mittleren Bohrung der Scheibe ist, damit sie durch die Scheibe paßt. Korrekterweise müßte der Radius der mittleren Bohrung etwas größer sein, als der Radius der Achse. Somit müßten eigentlich eine untere Toleranz für den Radius der mittigen Bohrung und eine obere Toleranz für den Radius der Achse in Abhängigkeit von Werkstoff und Funktion festgelegt werden. Dies soll hier der Einfachheit halber außer Acht gelassen werden. Außerdem soll die Länge der Achse größer sein als die Dicke der Scheibe.

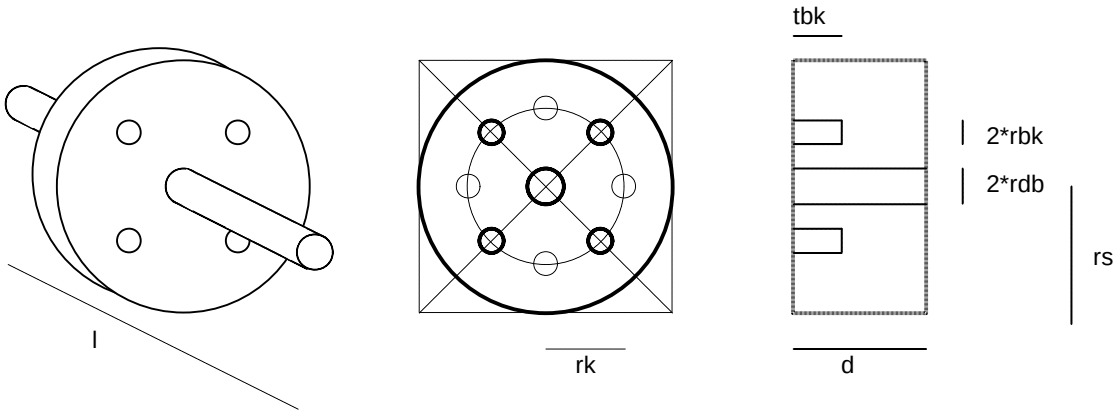


Abbildung 2.2 stellt die zu modellierende Baugruppe dar. Im folgenden werden nun die Parameter, die die Baugruppe beschreiben, erklärt.

Abbildung 2.2: *Darstellung der zu modellierenden Baugruppe*

Die Scheibe wird durch folgende Parameter definiert:

- Radius der Scheibe **rs**
- Dicke der Scheibe **d**

Die Achse wird definiert durch:

- Länge der Achse **l**
- Radius der Achse **ra**

Damit die Scheibe die gewünschte Form erhält, wird sie durch zwei Bohrvorgänge bearbeitet.

Der erste Bohrvorgang wird definiert durch:

- den Radius der Bohrung im Mittelpunkt der Scheibe **rdb**

Der zweite Bohrvorgang wird definiert durch:

- den Radius des Bohrkranzes **rk**
- den Radius der Bohrungen auf dem Bohrkranz **rbk**
- die Anzahl der Bohrungen auf dem Bohrkranz **n**
- die Tiefe der Bohrungen auf dem Bohrkranz **tbk**

Außerdem müssen die Positionen der Bohrungen genau definiert werden. Beim ersten Bohrvorgang ist die Position der Bohrung bekannt. Sie ist im Mittelpunkt der Scheibe M(x,y). Es sei M=(0,0) der Koordinatenursprung.

Beim zweiten Bohrgang ist dies nicht ganz so einfach. Die Positionen der Bohrungen hängen von mehreren Parametern ab:

- Radius des Bohrkranzes **rk**
- Anzahl der Bohrungen **n**
- Mittelpunkt Scheibe **M(x, y)**

In Abbildung 2.3 ist M der Mittelpunkt der Scheibe und **rk** ist der Radius des Bohrkranzes. Diese Werte sind gegeben. **P1** ist die Position der ersten Bohrung auf dem Bohrkranz, **P2** die Position der zweiten Bohrung.
Die Position der ersten Bohrung auf dem Bohrkranz ist willkürlich, wogegen die übrigen Positionen von dieser abhängig sind. Der Einfachheit halber wird die Position **P1** so definiert, daß sie auf der X-Achse liegt.

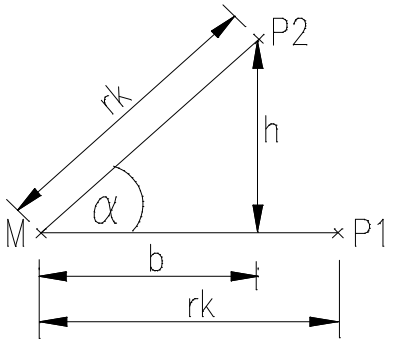


Abbildung 2.3: *Grundlage für die Berechnung der Positionen der Bohrungen*

Den Winkel α erhält man, indem man 360° durch die Anzahl n der Bohrungen teilt. Um die neuen Koordinaten der nächsten Bohrungen zu bestimmen, müssen die Werte h und b errechnet werden.

$$h=\sin(\alpha)\times rk \qquad b=\cos(\alpha)\times rk$$

Die neuen X-Werte errechnen sich aus dem X-Wert des Mittelpunkts plus b, die Y-Werte aus dem Y-Wert des Mittelpunkts plus h.

Pn (x+b, y+h)

$$P1\ (x+\cos(0°)\times rk,\ y+\sin(0°)\times rk) = (x+rk,\ y)$$
$$P2\ (x+\cos(\alpha)\times rk,\ y+\sin(\alpha)\times rk)$$
$$P3\ (x+\cos(2\alpha)\times rk,\ y+\sin(2\alpha)\times rk)$$
.
.
$$Pn\ (x+\cos((n-1)\alpha)\times rk,\ y+\sin((n-1)\alpha)\times rk)$$

Da M=(0,0) der gewählte Koordinatenursprung ist, vereinfachen sich die Positionsberechnungen auf:

$$Pn\ (\cos((n-1)\alpha)\times rk, \sin((n-1)\alpha)\times rk)$$

Diese Formel hat nur für einen symmetrischen, konzentrischen Bohrkranz Gültigkeit. Sobald keine Symmetrie gefordert wird, müssen die Positionen entweder entsprechend den Anforderungen berechnet oder eingegeben werden.
Wenn die Scheibe eine mittige Bohrung aufweist, so ist die Achse mit einem Radius derselben Größe wie der Radius der Bohrung **ra=rdb** und einer Default-Länge bzw. durch Abfragen des Parameters **l** zu erzeugen. Fällt der erste Bohrvorgang weg, so entfällt auch die Erstellung der Achse.

2.5.2 Technische Objekte

Die Baugruppe besteht aus fünf technischen Objekten.

1. Metallscheibe
2. erster Bohrvorgang
3. Bohrkranz
4. zweiter Bohrvorgang
5. Achse

Die beiden Bohrvorgänge sind immaterielle technische Objekte. Sie stellen Vorgänge dar, bei denen Material abgetragen wird. Ebenso ist der (fiktive) Bohrkranz ein immaterielles Objekt, der ausschließlich zur Positionierung der Bohrungen auf dem Kranz dient.

2.5.3 Technisch-funktionale Abhängigkeiten

Im folgenden werden die im Szenario bestehenden technisch-funktionalen Abhängigkeiten, also Beziehungen zwischen den Objekten, dargestellt.

1. Die Größe des Radius der Achse hängt vom Radius des ersten Bohrvorgangs ab und umgekehrt

   $$ra = rdb$$

2. Die Positionen der Bohrungen auf dem Bohrkranz hängen vom Radius des Bohrkranzes und der Anzahl n der Bohrungen ab.

   $$a = \frac{360°}{n}$$

   $$Pn := (\cos((n-1)a) \times rk, \sin((n-1)a) \times rk)$$

3. Die Tiefe der durchgehenden, mittigen Bohrung hängt von der Dicke der Scheibe ab.

   $$tdb := d$$

Neben den technisch-funktionalen Abhängigkeiten gibt es weitere Abhängigkeiten. Beispielsweise muß der Radius des Bohrkranzes kleines als der Radius der Scheibe oder die Tiefe der Bohrungen auf dem Bohrkranz muß kleiner als die Dicke der Scheibe sein. Diese Abhängigkeiten sind nicht eindeutig definiert, sondern sind Konsistenz-

bedingungen, die die einzelnen Werte einschränken, um Fehlkonstruktionen zu vermeiden. All diese Abhängigkeiten der einzelnen Parameter untereinander werden in Kapitel 2.5.5 ausführlich erläutert.

2.5.4 Technisches Partialmodell

Im technischen Partialmodell werden nun die Objekte und ihrer Beziehungen untereinander dargestellt. Zur Darstellung des Modells wurde eine Entity-Relationship-

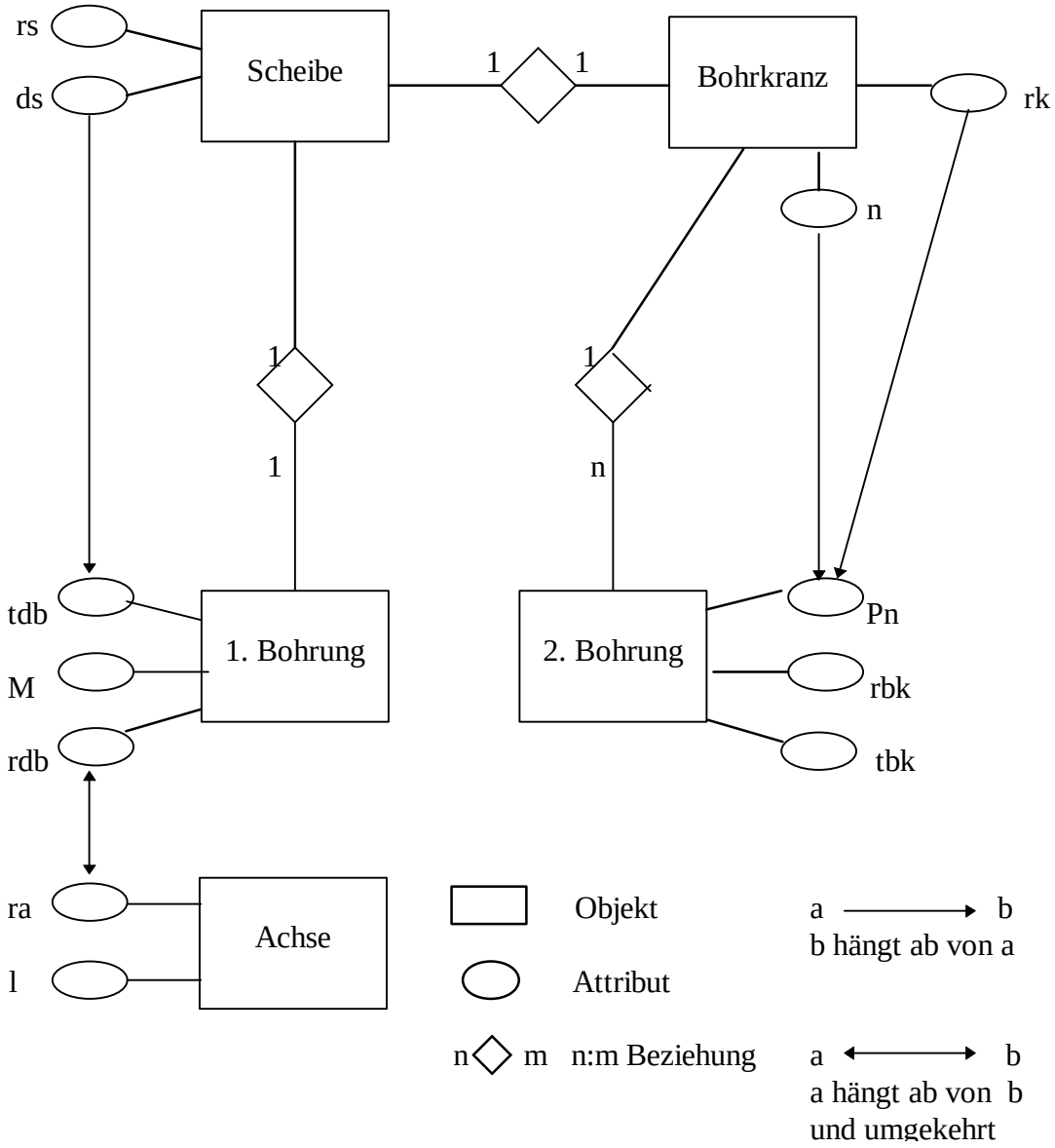


Abbildung 2.4: *Technisches Modell der Baugruppe*

ähnliche Notation gewählt.

In Abbildung 2.4 sind die technischen Objekte als Rechtecke dargestellt. Die Einträge an den Ellipsen definieren die Attribute der Objekte. Die Pfeile zwischen den Ellipsen stellen technisch-funktionale Abhängigkeiten dar, wobei die Werte der Variablen in Richtung des Pfeils an die abhängigen Variablen weitergegeben werden. Dies bedeutet, daß, wenn „a“ verändert wird, so soll automatisch auch „b“ geändert werden. Wenn die Dicke der Scheibe **ds** verändert wird, so muß die Tiefe der mittigen Bohrung **tdb** angepaßt werden. Bei Änderung der Anzahl **n** der Bohrungen auf dem Kranz oder des Radius des Bohrkranzes **rk** müssen die Positionen der Kranzbohrungen **Pn** neu berechnet werden. Falls der Radius der mittigen durchgehende Bohrung geändert wird, so muß auch eine Änderung des Radius der Achse erfolgen und umgekehrt. Diese bidirektionale Beziehung ist durch einen Doppelpfeil dargestellt.
Die Beschreibungen an den Rauten sollen die Art der Beziehung zweier Objekte darstellen. Beispielsweise stehen Scheibe und Bohrkranz bzw. Scheibe und 1. Bohrung in einer 1:1 Beziehung zueinander, d.h. jede Scheibe hat höchstens einen Bohrkranz bzw. eine mittige (erste) Bohrung und umgekehrt. Die 1:n Beziehung zwischen Bohrkranz und zweiter Bohrung beschreibt, daß ein Bohrkranz 0 bis n Bohrungen haben kann. Jede Bohrung ist genau einem Bohrkranz zugeordnet.

Nach der Beschreibung und Darstellung der technischen Objekte im technischen Partialmodell sollen im folgenden die technischen Operationen, die auf die Objekte angewandt werden können, herausgearbeitet werden.

2.5.5 Technische Operationen

Bei den technischen Operationen müssen Konsistenzprüfungen vorgenommen werden, um Fehlkonstruktionen zu vermeiden. Die Konsistenzprüfungen sind zwar auch Abhängigkeiten, jedoch werden die Parameter nicht auf konkrete Werte gesetzt oder berechnet, sondern sind nur in ihrem Wertebereich eingeschränkt. Die Konsistenzprüfungen werden im folgenden näher beschrieben.

Erzeugen der Scheibe

1. Der Radius der durchgehenden, mittigen Bohrung darf nicht größer sein als der Radius des Bohrkranzes abzüglich des Radius der Bohrungen auf dem Bohrkranz, da sich sonst die Bohrungen auf dem Bohrkranz und die mittlere Bohrung überschneiden würden (siehe Abbildung 2.5).

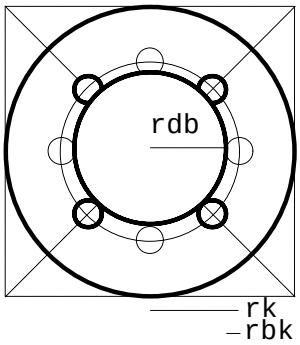


Abbildung 2.5: *Kranzbohrungen und mittige Bohrung überschneiden sich*

Zu überprüfen ist, ob

$$rdb < (rk - rbk) \qquad (1)$$

erfüllt ist. Ist das nicht der Fall, so müssen die Werte **rdb** oder **rk** oder **rbk** oder auch alle verändert werden, so daß diese Bedingung gilt.

2. Die Bohrungen auf dem Kranz dürfen sich nicht überschneiden.

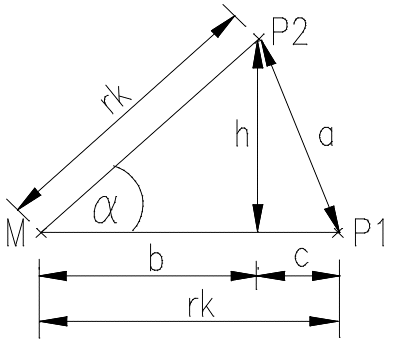


Abbildung 2.6: *Grundlage zur Berechnung des maximalen rbk*

Damit sich die Bohrungen auf dem Bohrkranz nicht überschneiden, muß der Radius der Bohrungen kleiner sein als die Hälfte der Strecke $\overline{P1P2} = \mathbf{a}$. Aus Abbildung 2.6 kann man entnehmen, daß gilt:

$$h = \sin(\alpha) \times rk = \sin\left(\frac{360^\circ}{n}\right) \times rk$$

$$b = \cos(\alpha) \times rk = \cos\left(\frac{360^\circ}{n}\right) \times rk$$

$$c = rk - b$$

$$a = \sqrt{h^2 + c^2}$$

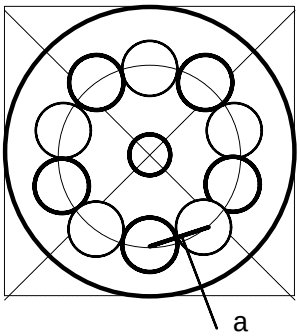


Abbildung 2.7: *Bohrungen auf dem Bohrkranz überschneiden sich*

Die Konsistenzprüfung erfolgt nun über die Bedingung

$$a \geq 2 \times rbk + \Delta x \qquad (2)$$

wobei Δx eine materialabhängige Konstante ist. Falls die Gleichung **(2)** nicht erfüllt ist, müssen entweder **rk** oder **rbk**, eventuell auch **n** solange verändert werden, bis obige Bedingung gilt. Abbildung 2.7 zeigt die Überschneidung der Kranzbohrungen.

3. Die Bohrungen auf dem Bohrkranz dürfen sich nicht zu nah am äußeren Rand der Scheibe befinden.

$$rk + rbk < rs \qquad (3)$$

Auch hier gilt: Solange diese Bedingung nicht erfüllt wird, müssen die Werte entsprechend verändert werden. Abbildung 2.8 stellt die sich zu weit außen befindlichen Kranzbohrungen dar.

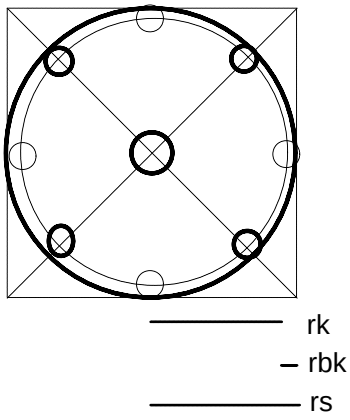


Abbildung 2.8: *Bohrungen auf dem Bohrkranz befinden sich zu weit außen*

4. Die Tiefe der Bohrungen auf dem Kranz muß kleiner sein als die Dicke der Scheibe (siehe Abbildung 2.9), da diese Bohrungen nicht durchgehend sein sollen..

$$tbk < d \qquad (4)$$

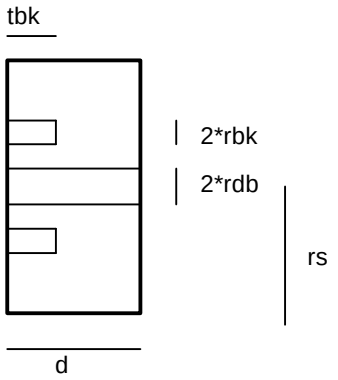


Abbildung 2.9: *Tiefe der Kranzbohrungen muß kleiner sein als die Scheibendicke*

5. Falls $rdb > 0$, so muß eine Achse erzeugt werden, die denselben Radius hat wie ***rdb***.

$$ra = rdb \qquad (5)$$

Dies ist keine Konsistenzbedingung, sondern eine Abhängigkeit von **ra** gegenüber rdb. Wenn **rdb** verändert wird, muß **ra** automatisch angepaßt werden und umgekehrt. Normalerweise müßte man **rdb** eine Toleranz nach oben und **ra** eine Toleranz nach unten gewähren. Dies soll hier jedoch außer acht gelassen werden.

Es sollten die Möglichkeiten bestehen, die Achse mit einer Default-Länge zu generieren, die verändert werden kann, oder den Längenparameter direkt abzufragen. Die Länge der Achse muß auf jeden Fall größer als die Dicke der Scheibe sein. Dabei sollte es auch möglich sein, zu überprüfen, ob sinnvolle Werte eingegeben wurden. Als sinnvoller Werte kann beispielsweise die doppelte Dicke der Scheibe gewählt werden. Dennoch sollte dem Benutzer die Möglichkeit gegeben werden, trotz Unterschreitung des sinnvollen Wertes, die Achse nach seinen Vorstellungen zu erstellen. Eventuell kann eine Warnmeldung auf die Unterschreitung des sinnvollen Wertes hinweisen.
Die Konsistenzbedingung hierfür lautet:

$$l > d \qquad (6)$$

6. Der Radius der durchgehenden, mittigen Bohrung muß kleiner sein als der Radius der Scheibe. Diese Bedingung wurde allerdings durch **(1)** und **(3)** schon beschrieben.

$$rdb < (rk - rbk) < (rk + rbk) < rs$$

7. Bei der Erzeugung der Scheibe müssen die Positionen der Bohrungen auf dem Bohrkranz berechnet werden:

$$Pn := (\cos((n-1)a) \times rk, \sin((n-1)a) \times rk) \qquad (7)$$

8. Die Tiefe der durchgehenden, mittigen Bohrung muß ebenfalls festgesetzt werden.

$$tdb := d \qquad (8)$$

Verändern der Scheibe

Bei Änderung einzelner Parameter sind wiederum Konsistenzprüfungen vorzunehmen, damit, wie oben schon erwähnt, keine Fehlkonstruktionen entstehen.

9. Bei Veränderung der Dicke der Scheibe d gilt folgende Konsistenzbedingungen:

$$tbk < d \qquad \textbf{(4)}$$

$$l > d \qquad \textbf{(6)}$$

Wenn **d** beispielsweise verkleinert wird, so muß überprüft werden, ob die Dicke der Scheibe immer noch größer ist als die Tiefe der Bohrungen auf dem Kranz. Ist dies nicht der Fall, muß entweder **tbk** oder **d** angepaßt werden. Wenn **tbk** verändert wird, so müssen die Konsistenzprüfungen für **tbk** vorgenommen werden (siehe **14.**). In diesem Fall gilt ebenfalls die Bedingung **(4)**.

Bei der Änderung von **d** muß automatisch die Tiefe des ersten Bohrvorgangs verändert werden:

$$tdb := d \qquad \textbf{(8)}$$

10. Bei Änderung des Radius der Scheibe **rs** gilt:

$$rk + rbk < rs \qquad \textbf{(3)}$$

Wenn **rs** kleiner gewählt wird, so müssen eventuell **rk** oder **rbk** oder beide verändert werden. Danach müssen die Konsistenzbedingungen von **rk** und **rbk** beachtet werden. Diese sind dann **(1)**, **(2)** und **(3)**. Wenn die Bedingung **(1)** nicht erfüllt ist, so müßte auch noch **rdb** verändert werden, was dann noch Bedingung **(5)** nach sich ziehen würde. Es ist ersichtlich, daß eine kleine Änderung viele Konsistenzprüfungen und eventuelle Änderung anderer Parameter bewirkt. Die Konsistenzprüfungen mit den entsprechenden Änderungen müssen solange durchgeführt werden, bis alle Bedingungen erfüllt sind.

11. Bei Änderung des Radius der durchgehenden, mittigen Bohrung **rdb** sind folgende Bedingungen zu beachten:

$$rdb < (rk - rbk) \qquad \textbf{(1)}$$

$$ra = rdb \qquad \textbf{(5)}$$

Bei der Änderung von **rdb** muß automatisch der Radius der Achse **ra** auf den Wert von **rdb** gesetzt werden.

12. Bei Änderung des Radius des Bohrkranzes **rk** lauten die Konsistenzbedingungen:

$$rdb < (rk - rbk) \qquad \textbf{(1)}$$

$$\sin\left(\frac{180°}{n}\right) \times rk > rbk \qquad \textbf{(2)}$$

$$rk + rbk < rs \qquad \textbf{(3)}$$

Wenn **rk** verändert wird, müssen auch die Positionen der Bohrungen auf dem Kranz angepaßt werden:

$$Pn:=(\cos((n-1)a)\times rk,\sin((n-1)a)\times rk) \quad (7)$$

13. Nach Änderung des Radius der Bohrungen auf dem Bohrkranz **rbk** sind folgende Bedingungen zu überprüfen:

$$rdb < (rk - rbk) \quad (1)$$

$$\sin\left(\frac{180°}{n}\right)\times rk > rbk \quad (2)$$

$$rk + rbk < rs \quad (3)$$

14. Die Veränderung der Tiefe der Bohrungen auf dem Bohrkranz **tbk** zieht die Konsistenzprüfung

$$tbk < d \quad (4)$$

nach sich.

15. Nach Änderung der Anzahl der Bohrungen auf dem Bohrkranz **n** folgt die Überprüfung:

$$\sin\left(\frac{180°}{n}\right)\times rk > rbk \quad (2)$$

Wenn **n** geändert wird, müssen die neuen Positionen der Bohrungen auf dem Kranz berechnet werden:

$$Pn:=(\cos((n-1)a)\times rk,\sin((n-1)a)\times rk) \quad (7)$$

Änderung der Achse

16. Bei Änderung der Länge **l** der Achse muß gelten:

$$l > d \quad (6)$$

17. Falls der Radius der Achse **ra** geändert wird, so muß der Radius der mittleren durchgehenden Bohrung **rdb** auf denselben Wert gesetzt werden.

$$ra=rdb \quad (5)$$

Löschen der Scheibe

Wenn eine Scheibe gelöscht wird, so muß auch die zugehörige Achse gelöscht werden. Auch wenn der erste Bohrvorgang „gelöscht“ wird, d.h. wenn der Radius auf 0 gesetzt wird, muß die entsprechende Achse gelöscht werden.

Löschen der Achse
Falls die Achse gelöscht wird, muß der Radius der mittigen Bohrung der Scheibe auf den Wert 0 gesetzt werden.

Positionieren und Orientieren
Die technischen Operationen „Positionieren“ und „Orientieren“ entfallen beim Szenario, da die Baugruppe nicht im Zusammenhang mit anderen Baugruppen oder Einzelteilen betrachtet wird, sondern isoliert. Normalerweise müßte die Scheibe an eine bestimmte Stelle eines Einzelteils positioniert werden. Die Orientierung würde dann die Koordinaten dieser Stelle beschreiben.
Ein großer Vorteil wäre, wenn das Datenbanksystem die Werte der Parameter, die technisch-funktional von anderen Parametern abhängen, automatisch generiert bzw. aktualisiert. So würde das Setzen und Berechnen dieser Parameter beim Erzeugen und Verändern entfallen.

2.5.6 Geometrisches Partialmodell

Aus dem technischen Partialmodell kann nun das geometrische Partialmodell entwickelt werden. Im geometrischen Modell werden die geometrischen Formen der technischen Objekte und ihre Topologie betrachtet. Bei sämtlichen Objekten des Szenarios handelt es sich um Zylinder (Scheibe und Achse), selbst die beiden Bohrvorgänge tragen Material in Form von Zylindern ab.

Ein körperorientiertes Modellierungsverfahren ist „Constructive Solid Geometry“ (kurz **CSG**). Hierbei wird beispielsweise die Scheibe mit mittlerer Bohrung als Differenzmenge aus Zylinder (Scheibe) und Zylinder (1. Bohrung) gebildet. Wenn nun die Differenzmenge aus dieser Scheibe und n Zylindern gebildet wird, so ist das Ergebnis die Scheibe mit durchgehender, mittiger Bohrung und Bohrkranz. Vereinigt man dies mit der Achse (wiederum ein Zylinder) resultiert daraus die gesamte Baugruppe. Wie man die Baugruppe aus Differenzen und Vereinigung von Zylindern erhält, ist in Abbildung 2.10 dargestellt. Das geometrische Modell wird dabei um zusätzliche Attribute ergänzt.

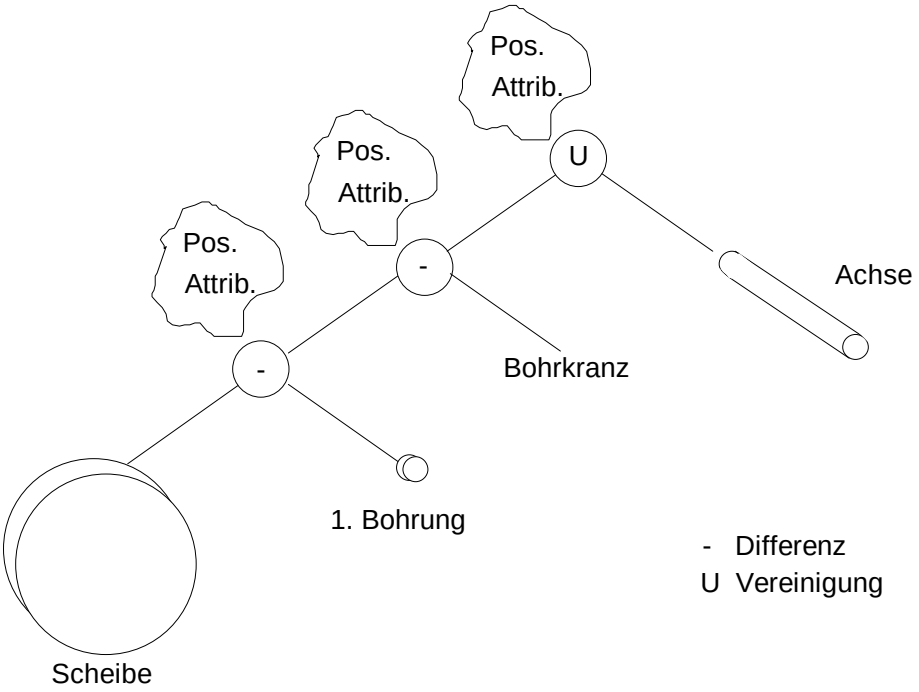


Abbildung 2.10: *Das geometrische Modell der Baugruppe*

Der Bohrkranz kann ebenso geometrisch als Differenz von Zylindern dargestellt werden (siehe Abbildung 2.11).

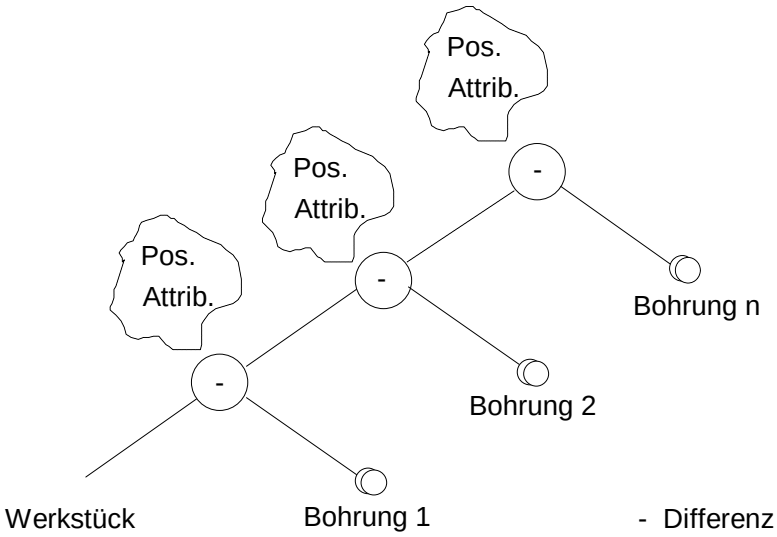


Abbildung 2.11: *Das geometrische Modell des Bohrkranzes*

An dieser Stelle würde normalerweise das technologische Partialmodell entwickelt werden. Da aber in diesem Szenario der Werkstoff, die Oberfläche etc. nicht ins Gewicht fallen und Toleranzen nicht berücksichtigt werden sollen, wurde darauf verzichtet..

2.5.7 Zusätzliche Attribute des Modells

Für die Verwaltung der Baugruppe sind nicht nur die technischen Daten, die für die Fertigung notwendig sind, von Interesse sondern beispielsweise auch Daten, die die Baugruppe beschreiben. Zusätzliche Attribute sind:

- Name der Baugruppe
- Beschreibung
- Bearbeiter
- Konstruktionsdatum
- Material
- spezifisches Gewicht des Materials
- Zeitstempel pro Bohrung
- Boolsches Flag pro Bohrung, das angibt, ob die Bohrung schon durchgeführt wurde
- Version
- Variante

Version und Variante könnten wie folgt definiert werden:
Wird nur ein Parameter einer bestehenden Baugruppe verändert, so könnte dies zu einer neuen Version führen. Wenn hingegen mehrere Parameter verändert werden, könnte eine neue Variante entstehen. Es soll jedoch dem Benutzer freigestellt werden, wann eine neue Version und wann eine neue Variante vorliegt.
Außerdem soll noch eine Möglichkeit zur Abfrage von Volumen und Gewicht zur Verfügung gestellt werden.
Das Volumen der Baugruppe errechnet sich aus dem Volumen der „rohen“ Scheibe abzüglich des Volumens der mittigen Bohrung und des Volumens des zweiten Bohrvorgangs und zuzüglich des Volumens der Achse.

1. Volumen „rohe“ Scheibe: $\mathbf{P}(rs)^2 \times d$
2. Volumen mittige Bohrung: $\mathbf{P}(rdb)^2 \times d$
3. Volumen der Bohrungen auf dem Kranz $\mathbf{P}(rbk)^2 \times tbk \times n$
4. Volumen der Achse: $\mathbf{P}(rdb)^2 \times l$

Volumen der gesamten Baugruppe:

$$V = \mathbf{P} \times (((rs)^2 - (rdb)^2) \times d - (rbk)^2 \times tbk \times n + (rdb)^2 \times l)$$

Gewicht der gesamten Baugruppe:

$$G = V \times spez.Gewicht$$

Das spezifische Gewicht entnimmt man den zusätzlichen Attributen.

Die Aufgabe besteht nun darin, geeignete Datenbanken zu finden, die technisch-funktionale Abhängigkeiten darstellen können. Die konventionellen relationalen Datenbanken können die Objekte nur in begrenztem Maße verwalten. Sobald die Objekte eine komplexe Struktur aufweisen, wird dies sehr schwierig oder es ist gar nicht möglich. Beziehungen können nur über teilweise sehr komplizierte Verknüpfungen hergestellt werden.

Im nächsten Kapitel erfolgt eine Einführung in Datenbanken. Vorgestellt werden die Konzepte relationaler und objektorientierter Datenbanksysteme.

3 Einführung in Datenbanken

In diesem Kapitel werden Datenbanken und ihr Aufbau beschrieben. Nach einer allgemeinen Einführung in Datenbanken (auch Datenbanksysteme = DBS genannt) werden die Strukturen der Daten, die auf Datenmodellen basieren und die Aufgaben des Datenbankmanagementsystems (DBMS) behandelt. Vorgestellt werden relationale Datenbanksysteme (RDBS) und objektorientierte Datenbanksysteme (OODBS).

3.1 Aufgaben und Strukturen von Datenbanken

Datenbanken dienen dazu, die Menge aller gespeicherten Daten zu strukturieren und zu verwalten. Eine Datenbank besteht aus einer Datenbasis, in der die eigentlichen Daten abgespeichert sind und einem Datenbankmanagementsystem, welches für die Verwaltung der Daten zuständig ist. Das DBMS stellt die Schnittstelle zwischen der Datenbasis und dem jeweiligen Anwendungsprogramm dar, d.h., es greift auf die Daten in der Datenbasis zu und stellt sie dem Anwendungsprogramm zur Verfügung und speichert die aktualisierten Daten wieder in der Datenbasis. Durch Trennung von Daten und Programmen wird Datenunabhängigkeit erreicht, was zum Vorteil hat, daß Anwenderprogramme unabhängig von den Daten verändert werden können. Dies bedeutet eine erhebliche Erleichterung bei der Modifikation von Programmen und somit Bereitschaft für Innovationen und Veränderungen.

Dadurch, daß alle Anwendungsprogramme über das DBMS auf dieselbe Datenbasis zugreifen, werden Redundanzen vermieden. Redundanzen entstehen durch mehrfache, meist uneinheitliche Speicherung der gleichen Daten in mehrere Dateien. Dies ist der Fall, wenn unterschiedliche Anwendungsprogramme auf ihre speziellen Datenbasen zugreifen und vom jeweiligen DBMS nicht in allen Datenbasen die Änderungen durchgeführt werden oder die gleichen Daten unterschiedlich eingegeben und somit uneinheitlich abgespeichert werden. Abhilfe kann hier die Datenintegration schaffen. Sie soll durch die Zugriffe verschiedener Anwendungsprogramme auf eine zentrale Datenbank sichergestellt werden, wie in Abbildung 3.1 dargestellt wird.

Die folgende Abbildung soll verdeutlichen, wie verschiedene Bereiche eines Unternehmens und damit unterschiedliche Anwendungsprogramme auf dieselbe globale Datenbank zugreifen, in der alle Daten einheitlich gespeichert sind. Werden Daten aktualisiert, so geschieht dies nur einmal und jeder Anwender hat danach die richtigen Daten zur Verfügung. Wenn die Daten redundant wären, müßten sie in jeder Datenbank, in der sie vorkommen, verändert werden.

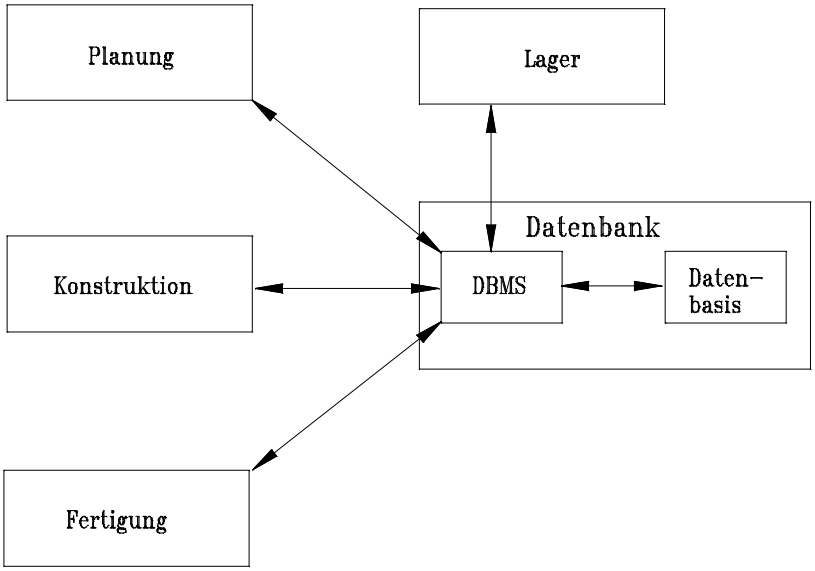


Abbildung 3.1: *Zugriffskonzept für eine Beispiel-Datenbank*

Anwendungsprogramme fungieren als Schnittstellen zwischen DBMS und Benutzer, über die Daten neu eingegeben, verändert, gelöscht oder abgefragt werden können. Dabei ist nicht nur die Abfrage der in der Datenbasis stehenden einzelnen Daten möglich, sondern es können durch geeignete Methoden zusätzliche Informationen gewonnen werden. Diese Methoden werden nur einmal bestimmt, können aber beliebig oft angewandt werden. Sie spielen eine große Rolle bei sich häufig ändernden Daten.

> **Beispiel**:
>
> Wenn das Volumen eines Körpers und das spezifische Gewicht des Materials, aus dem der Körper besteht, als Daten vorliegen, so kann das Gewicht des Körpers bestimmt werden, ohne daß dies explizit in die Datenbasis eingegeben werden muß. Es wird bei jeder Anfrage neu berechnet, nimmt aber keinen zusätzlichen Speicherplatz in Anspruch.

Nach dieser Einführung in Datenbanken wird im folgenden auf die Strukturierung der Daten eingegangen. Die formale Beschreibung der Daten und ihrer Beziehungen untereinander erfolgt in Datenmodellen.

3.2 Datenmodelle

In Datenmodellen (DM) werden die Eigenschaften von Objekten und deren Beziehungen untereinander beschrieben. Es wird ein Realweltausschnitt auf ein Modell abgebildet. In den nächsten Abschnitten werden syntaktische DM, das hierarchische und das Netzwerkmodell, und ein semantisches DM, das Entity-Relationship-Modell, behandelt. Das zu den syntaktischen DM gehörende relationale Datenmodell wird in Kapitel 3.4

ausführlich erläutert. Diese Datenmodelle werden nun kurz eingeführt, um die bekannten Konzepte vorzustellen.

3.2.1 Hierarchisches Datenmodell

Dieses Modell besitzt die Struktur eines Baumes und kann effizient, z.B. in einer linearen Liste, implementiert werden. Problematisch ist allerdings, daß sich nicht alles in einer Hierarchie darstellen läßt.

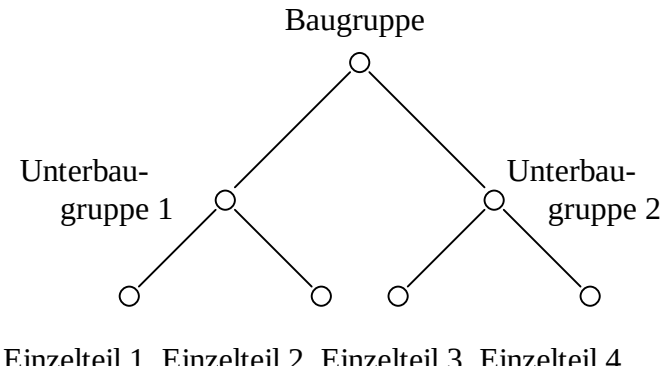


Abbildung 3.2: *Struktur des hierarchischen Modells am Beispiel einer Baugruppe*

Abbildung 3.2 zeigt beispielhaft den Aufbau einer Baugruppe. Diese besteht aus zwei Unterbaugruppen, die wiederum jeweils aus zwei Einzelteilen zusammengesetzt sind. Die Komplexität des hierarchischen Datenmodells steigt rasch an, sobald z.B. Einzelteile aus anderen Unterbaugruppen Bestandteil von Unterbaugruppe 2 werden. Im obigen Beispiel müßten Einzelteil 1 und 2 bei Unterbaugruppe 2 angehängt werden. Dies führt zu Unübersichtlichkeit und Mehrfacheingabe der Daten.

3.2.2 Netzwerkmodell

Ein weiteres syntaktisches Modell ist das Netzwerkmodell, das als Graph dargestellt wird. Die Knoten repräsentieren die Objekte und die Kanten deren Beziehungen zu anderen Objekten. Die Implementierung ist jedoch aufwendiger als die eines Baumes, da bei n Knoten eine $n \times n$ -Adjazenzmatrix zur Beschreibung der Kanten erforderlich ist. Hierbei steigt die Komplexität durch die Zunahme der Objekte, während beim Baum die vermehrte Anzahl von Beziehungen für die größere Komplexität verantwortlich ist.

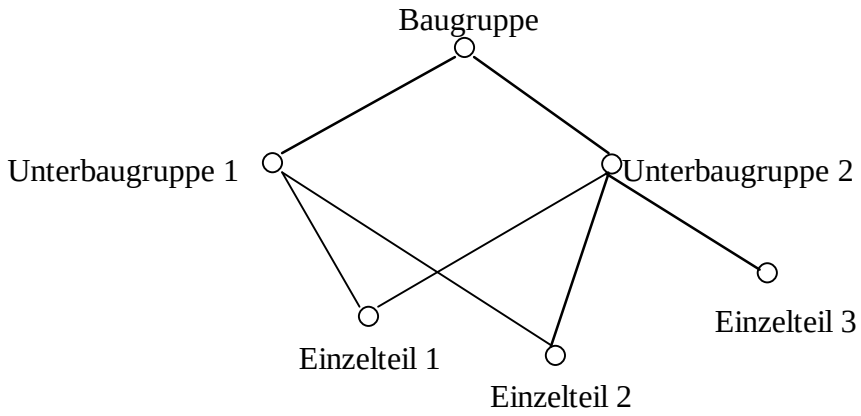


Abbildung 3.3: *Struktur des Netzwerkmodells am Beispiel einer Baugruppe*

Abbildung 3.3 zeigt wiederum die Baugrupppe mit den Unterbaugruppen 1 und 2, jedoch ist hier zu beiden Unterbaugruppen die Teile 1 und 2 und zu Unterbaugruppe 2 zusätzlich noch Einzelteil 3 hinzugefügt worden. Jegliche Beziehungen oder Abhängigkeiten der Objekte untereinander sind darstellbar.

3.2.3 ER-Modell

Das **E**ntity-**R**elationship-Modell ist ein semantisches Datenmodell. Es ist unabhängig von einer Anwendung. In diesem Modell werden Objekte mit ihren Eigenschaften und den Beziehungen zu anderen Objekten dargestellt. Die sogenannten Entities sind genau definierte identifizierbare Objekte des abzubildenden Realweltausschnittes, wie z.B. Personen oder Dinge. Die Entities werden durch Rechtecke, Beziehungen (Relationships) durch Rauten und die Eigenschaften der Objekte durch Ellipsen graphisch beschrieben.

Abbildung 3.4 zeigt die Beziehungen der Objekte am Beispiel Auftragsabwicklung. Ein Kunde erteilt einen Auftrag. Dieser Auftrag erhält eine Auftragsnummer und wird mit dem Datum der Bestellung versehen und besteht aus mindestens einem Artikel.

Im ER-Modell werden **drei** Arten von Beziehungen unterschieden:

1:1 Beziehung
Ein Objekt steht mit genau **einem** anderen Objekt in Beziehung.
Beispiel :

1 1
Schraube ——— Mutter

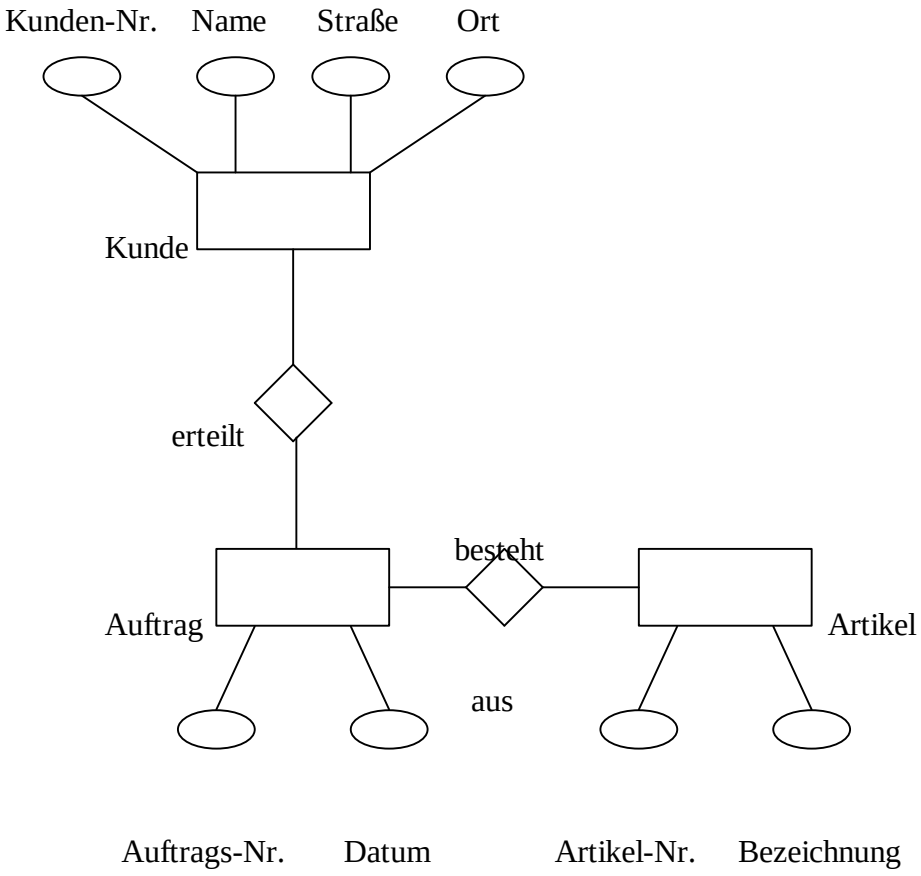


Abbildung 3.4: *ER-Modell für die Auftragsbearbeitung*

1:M Beziehung
Ein Objekt steht mit mehreren anderen Objekten in Beziehung.
Im Beispiel aus Abbildung 3.4 besteht eine 1:M Beziehung zwischen Kunde und Auftrag.

1 M
Kunde ——— Auftrag

Ein Kunde kann **mehrere** Aufträge erteilen, jedoch wird jedem Auftrag nur **ein** Kunde zugeordnet.

M:N Beziehung
Ein oder mehrere Objekte stehen in Beziehung zu einem oder mehreren Objekten.
Eine M:N Beziehung zeigen Auftrag und Artikel im Beispiel aus Abbildung 3.4.

M N
Auftrag —— Artikel

Ein Auftrag besteht aus einem oder mehreren Artikeln und umgekehrt kann ein Artikel in einem oder mehreren Aufträgen vorkommen.
Zusammenfassend kann gesagt werden, daß hierarchische DM einfach in der Handhabung sind, jedoch die Komplexität rasch ansteigt, wenn mehrere Beziehungen der Objekte untereinander bestehen. Netzwerkmodelle sind zwar in der Lage, Beziehungen und Abhängigkeiten von Objekten darzustellen, aber die Implementierung erfordert einen größeren Aufwand. ER-Modelle können Beziehungen gut darstellen, müssen aber auf ein syntaktisches Datenmodell abgebildet werden, da sie anwendungs- und rechner-unabhängig sind.
Wie oben schon erwähnt, bestehen Datenbanken nicht nur aus der Datenbasis; ein wichtiger Bestandteil einer Datenbank ist das Datenbankmanagementsystem, das die Schnittstelle zwischen den Daten und den Anwendungsprogrammen darstellt. Im folgenden werden anwendungsunabhängige Anforderungen an DBMS definiert.

3.3 Anforderungen an Datenbankmanagementsysteme

DBMS sind für die Realisierung von Mehrbenutzerbetrieb notwendig. Sie stellen den einzelnen Anwendern die jeweils angeforderten Daten zur Verfügung. Dabei hat jeder Benutzer den Eindruck, daß er alleinigen Zugriff auf die Datenbasis hat, d.h. er wird nicht durch Anfragen anderer Benutzer beeinträchtigt. Hierbei spielt die Erhaltung der *Konsistenz* der Daten eine wichtige Rolle. Daten werden inkonsistent, wenn sie von mehreren Benutzern geändert werden und die jeweiligen anderen Änderungen nicht berücksichtigt werden.

Vorgang 1

Artikel-Nr.	**Bezeichnung**	**Bestand**
12345	Schraube	12500
Abbuchung von 500 Stück		
12345	Schraube	12000

Vorgang 2

Artikel-Nr.	**Bezeichnung**	**Bestand**
12345	Schraube	12500
Zubuchung von 500 Stück		
12345	Schraube	13000

Abbildung 3.5: *Konsistenzproblem bei gleichzeitigem Zugriff auf dieselben Daten*

Wenn die in Abbildung 3.5 dargestellten Vorgänge gleichzeitig stattfinden, ergibt sich ein falscher Bestand und die Datenbank befindet sich in einem inkonsistenten Zustand. Das DBMS muß in der Lage sein, Vorgang 2 den Zugang auf die Daten solange zu verwehren bis der Schreibzugriff von Vorgang 1 beendet ist. Somit wäre der Anfangsbestand 12000 und mit der Zubuchung von 500 Stück wieder 12500 und nicht 13000. Um einen konsistenten Zustand einer Datenbank zu erhalten, muß der Schreibzugriff exklusiv sein, d.h., wenn ein Datum verändert wird, erhält kein anderer Benutzer Schreibzugriff. Lesezugriff, sogar mehrfach, ist dagegen problemlos.

Um die Konsistenz einer Datenbank zu gewährleisten, sollte das DBMS Transaktionen nach dem ACID-Prinzip (**A**tomicity - **C**onsistency - **I**solation - **D**urability) durchführen.

Das heißt, daß eine Transaktion entweder ganz oder gar nicht ausgeführt wird. Im Fehlerfall, z.B. bei einem Systemabsturz, muß der ursprüngliche Zustand, der vor Beginn des abgebrochenen Prozesses bestand, durch das DBMS wieder hergestellt werden (Recovery). Andernfalls befinden sich die Daten in einem inkonsistenten Zustand. Zwar können einige Daten während einer Transaktion inkonsistent sein, jedoch muß die Datenbank bei Beendigung der Transaktion wieder konsistent sein. Die Datenbank darf von mehreren Benutzern gleichzeitig bearbeitet werden, jedoch müssen alle Transaktionen isoliert in einer bestimmten Reihenfolge ausgeführt werden. Die durch Transaktionen entstehenden Änderungen müssen selbst nach einem Systemabsturz Gültigkeit haben.
Eine weitere Aufgabe des DBMS besteht darin, Datenschutz zu gewährleisten, d.h. jeder Anwender kann nur die für ihn relevanten Daten abfragen oder bearbeiten, z.B. durch die Vergabe von Zugriffsrechten oder Verschlüsselung von Daten. Beispielsweise dürfen nicht alle Abteilungen auf Personaldaten, geheime Forschungsergebnisse oder ähnliches Zugriff haben. Ausführliche Informationen zu Datenverschlüsselung und Versiegelung von Dokumenten sind in [Anc94] zu finden.

Nach diesen allgemeinen Informationen zu DBMS werden nun zwei Ansätze, relationale und objektorientierte DBMS, dargestellt.

3.4 Relationale Datenbankmanagementsysteme

Grundlage für relationale Datenbankmanagementsysteme (RDBMS) ist das relationale Datenmodell. Im folgenden wird deshalb das Konzept dieses Modells vorgestellt, da RDBMS ausschließlich der Realisierung desselben dienen.

3.4.1 Relationales Datenmodell

Eine Relation wird als Menge von Daten definiert, auf die bestimmte Operationen angewandt werden können. Die Darstellung der Relationen erfolgt in Form einer Tabelle. Die Eigenschaften der Objekte werden durch sogenannte **Attribute** beschrieben. Sie definieren die Spalten der Tabelle. Die Gesamtheit aller Attribute, die zu einer Relation gehören, nennt man **Schema** der Relation. Die Zeileneinträge sind die sogenannten **Tupeln**, die Attribute nehmen hier konkrete Werte an.
Die Anzahl der Attribute wird **Grad** einer Relation genannt. Jedes Attribut hat einen Typ wie z.B. Integer, Real, Char, Strings, Datum, Zeit, Boolean (die sogenannten Standarddatentypen). Außerdem existiert zu jedem Attribut eine **Domäne**. Diese ist der Wertebereich des Attributtyps.

Dies läßt sich anhand des folgenden Beispiels anschaulich darstellen:

Artikel	Bezeichnung
1	Schraube
2	Mutter
3	Unterlegscheibe
4	Abdeckkappe

Abbildung 3.6: *Darstellung der Beispiel-Relation Artikel*

Im Beispiel aus Abbildung 3.6 besteht das **Schema** aus den Attributen *Artikel-Nr.* und *Bezeichnung*. Der **Grad** der Relation ist zwei, da sie aus zwei Attributen besteht. Die Zeileneinträge der Tabelle sind:

Tupel 1 = (1, Schraube)
Tupel 2 = (2, Mutter)
Tupel 3 = (3, Unterlegscheibe)
Tupel 4 = (4, Abdeckkappe)

Die **Domäne** des ersten Attributs ist eine Integer-Zahl, die beispielsweise 6 Stellen begrenzt ist. Die zweite Domäne ist ein auf z.B. 40 Zeichen begrenzter String.

Zur Identifizierung eines Tupels wird der sogenannte Primärschlüssel benötigt. Dieser besteht aus einem Attribut bzw. aus einer Kombination von Attributen, falls ein Attribut alleine nicht ausreicht, und darf nicht NULL sein. Hierbei ist NULL nicht zu vergleichen mit der Zahl 0 oder Blank bei Zeichenketten, sondern soll angeben, daß der Wert nicht bekannt ist. In Abbildung 3.6 ist der Primärschlüssel „Artikel-Nr“. Jede Artikel-Nummer wird nur einmal vergeben. Im Gegensatz dazu existieren natürlich viele Schrauben, Muttern etc. .

Bei einer Relation „Kunde“ mit dem Schema (Name, Straße, Ort) genügt als Primärschlüssel z.B. „Name“ nicht. Deshalb besteht die Notwendigkeit, eine Kombination der Attribute zu bilden. Name und Ort genügen noch nicht, deshalb wäre in diesem Fall der Primärschlüssel „Name + Straße + Ort“. Bei Einführung eines Attributs „Kunden-Nr.“ ist dies der Primärschlüssel, da, wie bei der Relation „Artikel“ die „Artikel-Nr“, jeder Wert auch nur einmal vorkommt. Zudem müssen sich zwei Tupeln in mindestens einem Attributwert unterscheiden, da sie sonst wiederum nicht eindeutig identifizierbar sind.
Das einmalige Vorkommen eines Objekts in einer Relation und die eindeutige Identifizierbarkeit wird auch als **Objektintegrität** bezeichnet.

Um Beziehungen zwischen Objekten darzustellen, wird neben dem Primärschlüssel ein Fremdschlüssel benötigt. Dieser ist ein Attribut einer Relation, das in einer anderen Relation Primärschlüssel ist.

Beispiel: Fremdschlüssel

Relation 1: Artikel (Artikel-Nr., Bezeichnung)
Relation 2: Kunde (Kunden-Nr., Name, Straße, Ort)
Relation 3: Auftrag (Auftrags-Nr., Datum, Kunden-Nr., Artikel-Nr.)

In Relation 3 ist der Primärschlüssel „Auftrags-Nr." und ein Fremdschlüssel ist „Kunden-Nr". (Primärschlüssel von Relation „Kunde") und ein anderer Fremdschlüssel ist Artikel-Nr. (Primärschlüssel von Relation „Artikel"). In diesem Beispiel ist zu erkennen, daß Beziehungen im relationalen Datenmodell durch eine Verknüpfung von Tabellen mit Hilfe der Fremdschlüssel dargestellt wird. Ein Auftrag hat also den Bezug zu genau einem Kunden und zu einem oder mehreren Artikeln.

Wenn der Primärschlüssel einer Relation zu kompliziert wird, da viele Attribute zu einer Kombination benötigt werden, kann ein künstlicher Schlüssel eingeführt werden. Es wird ein zusätzliches Attribut definiert, meist eine fortlaufende Nummer, das dann die Identifizierung der einzelnen Tupeln erleichtert.

Beispiel:
Die Relation „Kunde" hat das Schema (Name, Straße Ort). Der Primärschlüssel ist „Name + Straße + Ort". Mit der Einführung eines zusätzlichen Attributs „Kunden-Nr." ist nun der Primärschlüssel „Kunden-Nr.", der vollkommen ausreicht, die Tupel eindeutig zu identifizieren.

Jeder Fremdschlüsselwert muß in einer anderen Relation Primärschlüssel sein oder den Wert NULL haben. Die referentielle Integrität stellt ein Art Existenzabhängigkeit dar [Kra94]. In Abbildung 3.4 wäre diese Abhängigkeit: „Es darf kein Auftrag ohne Kunde oder ohne Artikel bestehen". Der Wert NULL ist nur möglich, wenn keine solche Existenzabhängigkeit besteht.
Bei Einfüge-, Änderungs- und Löschoperationen muß ebenfalls die abhängige Relation aktualisiert werden. Wenn in Beispiel 1 die Kunden-Nummer eines Kunden verändert wird, so muß dies auch in der Relation „Auftrag" in *all* seinen Aufträgen geschehen.

Relationale Datenbanksysteme sind für den Bereich der Verwaltung sehr gut geeignet. Wenn es sich dagegen um komplexe Objekte, die hauptsächlich im technischen Bereich vorkommen, handelt, so stößt man auf die Grenzen von RDBMS. Die Beschreibung dieser Objekte ist meist nicht oder sehr erschwert möglich. Deshalb ist nun eine neue Generation von DBMS, die objektorientierten DBMS, in Entwicklung, um diese Mängel zu beheben. Im nächsten Kapitel werden die OODBMS, die im Gegensatz zu RDBMS nicht auf einem einheitlichen Konzept beruhen, vorgestellt.

3.5 Objektorientierte Datenbankmanagementsysteme

Objektorientierte Datenbankmanagementsysteme (OODBMS) sollen spezielle Anforderungen der Anwender erfüllen. So soll es z.B. möglich sein, eigene Typen zu definieren, da sich komplexe Objekte meist nur unvollständig auf Standarddatentypen abbilden lassen.
Es gibt zwei Konzepte für die Implementierung von OODBMS. Die erste Möglichkeit ist, daß objektorientierte Programmiersprachen um Datenbankfunktionen erweitert werden. Dies bietet den Vorteil, daß keine zusätzliche Sprache erlernt werden muß. Die zweite Möglichkeit besteht darin, relationale Datenbanksysteme um objektorientierte Features zu erweitern, wie z.B. Vererbung und Definition eigener Typen.
In dieser Studienarbeit wird je ein Vertreter der beiden Ansätze untersucht. Zum ersten Ansatz gehört OBST und zum zweiten Postgres. Beide DBMS werden in Kapitel 5 bzw. Kapitel 6 eingehend erläutert.

3.5.1 Objektidentität

Bei OODBMS erhält jedes Objekt bei der Erstellung eine Objektidentität (OID), die bis zur Löschung bestehen bleibt und nie an ein anderes Objekt vergeben wird oder verändert werden kann. Realisiert wird dies durch ein sogenanntes Surrogat. Ein Surrogat ist ein global eindeutiges Attribut, d.h. im gesamten System gibt es keinen doppelten oder mehrfachen Eintrag. Zur Adressierung von Objekten sind Surrogate jedoch nicht besonders komfortabel, da sie aus beliebigen Zahlen und Buchstabenkombinationen bestehen, die für den Benutzer nicht unbedingt einen Sinn ergeben. So können auch bei OODBMS Schlüssel notwendig sein. Im Gegensatz zu RDBMS ist jedoch das Problem der eindeutigen Identität von Objekten gelöst.

Bei relationalen DBS erhält jedes Tupel, auch Datensatz genannt, eine Satz-Nr., die sich jedoch bei Löschoperationen verändert. Wird ein Satz gelöscht, wird die Satz-Nr. der nachfolgenden Sätze um eins verringert. Somit ist die Satz-Nr. zwar teilweise eindeutig, jedoch nicht immer. Durch die Einführung eines künstlichen Schlüssels kann dem abgeholfen werden, falls der Primärschlüssel aus einer großen Kombination von Attributen besteht. Jedoch muß dieser vom Anwender gepflegt werden, während die Objektidentität in die Zuständigkeit des Systems fällt. Dadurch sinkt die Fehlerquote, da das System nicht „vergißt“, bestimmte Dinge zu tun.

RDBMS

Satz-Nr.	Kunden-Nr.	Kunde
1	258	Krüger
2	746	Moser
3	332	Faber
4	776	August

OODBMS

OID	Kunden-Nr.	Kunde
1	258	Krüger
2	746	Moser
3	332	Faber
4	776	August

Löschen des Kunden „Moser“ ergibt folgendes:

Satz-Nr.	Kunden-Nr.	Kunde
1	258	Krüger
2	332	Faber
3	776	August

OID	Kunden-Nr.	Kunde
1	258	Krüger
3	332	Faber
4	776	August

Abbildung 3.7: *Satz-Nr. bei RDBMS und Objektidentität bei OODBMS*

Wird in Abbildung 3.7 im RDBMS Kunde „Moser“ mit der Satz-Nr. „2“ gelöscht, so erhält Kunde „Faber“ die Satz-Nr. „2“ und Kunde „August“ die Satz-Nr. „3“. Wohingegen beim OODBMS die Objektidentität „3“ bei „Faber“ und „4“ bei „August“ erhalten bleibt. Die OID „2“ wird nun für einen neuen Eintrag freigegeben.

3.5.2 Komplexe Objekte und ihre Strukturen

Komplexe Objekte lassen sich in der Regel nicht durch die einfachen Standarddatentypen wie Integer, Real, String, Date etc. beschreiben. Daher besteht die Notwendigkeit, eigene Typen definieren zu können. Objektorientierte Datenbanken stellen dafür sogenannte Typkonstruktoren zur Verfügung, mit denen aus Standarddatentypen komplexe Typen erstellt werden können [Heu92]:

1. **Tupelkonstruktor: Tuple of**

   setzt sich aus mehreren Komponenten unterschiedlichen Typs zusammen

2. **Mengenkonstruktor: Set of**

   setzt sich aus mehreren Komponenten eines Typs zusammen. Die Elemente in einer Menge kommen nie mehrfach vor und stehen in keiner Ordnung, wie z.B. einer Reihenfolge.

3. **Listenkonstruktor: List of**

   setzt sich wie der Mengenkonstruktor aus Komponenten eines Typs zusammen, jedoch können Elemente mehrmals vorkommen und die Liste ist geordnet. Bei einer Matrix beispielsweise ist die Reihenfolge sehr wichtig.

Beispiele für Typkonstruktoren:

```
Typ: Kunden
Set of (Tuple of (Name: Tuple of    (Vorname: String
                                     Nachname: String)
                  Adresse: Tuple of  (Straße: String
                                      PLZ: Integer
                                      Ort: String)
                  Kunden-Nr.: Integer)
```

In Abbildung 3.8 wird verdeutlicht, daß sich ein Kunde aus zwei Tupelkonstruktoren und einem einzelnen Attribut zusammensetzt. Der Name besteht aus den beiden Komponenten *Vorname* und *Nachname*. Die Adresse wird aus den drei Attributen *Straße*, *PLZ* und *Ort* gebildet. Die Gesamtheit aller Kunden wird mit dem Mengenkonstruktor gebildet, der sich aus den Tupeln aller Kunden zusammensetzt.

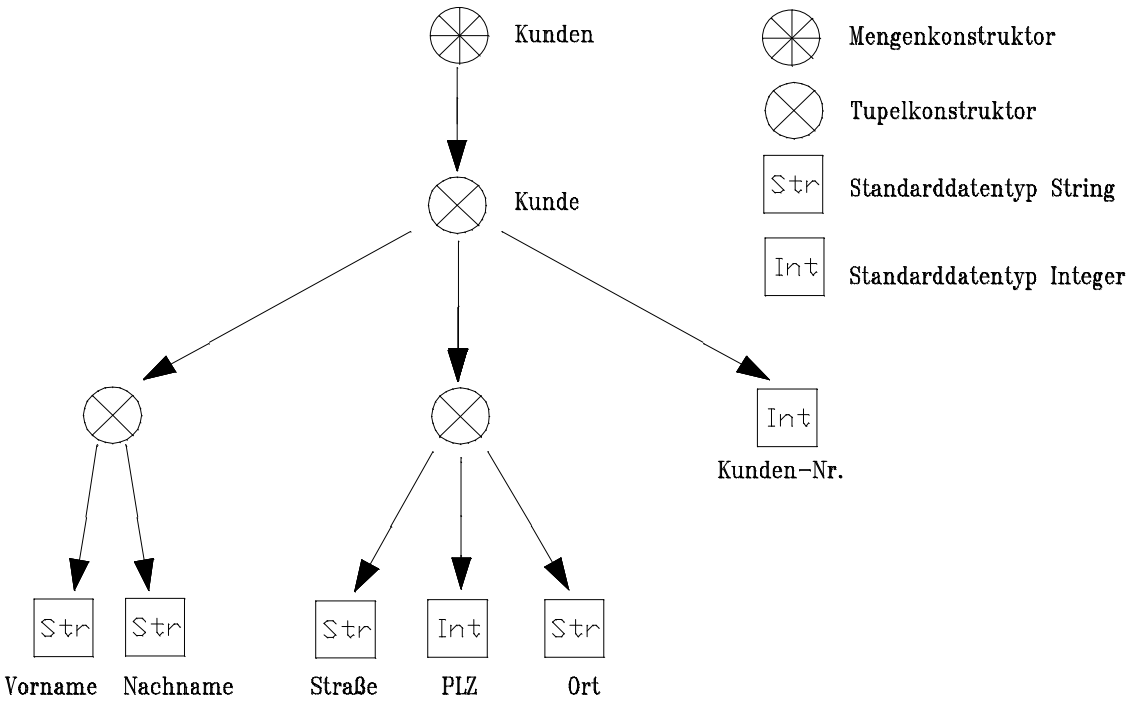


Abbildung 3.8: *Darstellung der Typkonstruktoren am Beispiel Kunde*

Es ist erforderlich, daß die obigen Typkonstruktoren orthogonal angewendet werden können. Das bedeutet, daß sie beliebig untereinander angewendet werden, wie im obigen Beispiel:

Set of (Tuple of(Tuple of)).

Die Definition eines einfachen geometrischen Körpers, wie z.B. dem Kreis, ist mit Standarddatentypen nicht möglich. Man benötigt einen Radius und einen Mittelpunkt mit dazugehörige Koordinaten.

Beispiel: Typ Kreis in *Postgres*

```
typedef struct {
     double x, y;
} POINT;

typedef struct {
     POINT center;
```

```
        double r;
    } CIRCLE;
```

Zuerst wird ein Typ “Punkt“ definiert, der aus einer Koordinate X und Y besteht. Danach definiert man den Typ “Kreis“, der aus dem Mittelpunkt vom Typ “Punkt“ besteht und dem Radius vom Typ “double“ (reelle Zahl).
Man sieht leicht, daß solch simple Typen zu einem fast unlösbaren Problem bei relationalen DBS wird. Man kann zwar die Koordinaten und den Radius als Real-Zahlen verwalten, jedoch ist keine Darstellung der Zusammengehörigkeit der Daten möglich.

Zur Durchführung von Operationen auf komplexe Typen benötigen die Typkonstruktoren bestimmte Operatoren.

- Der Tupelkonstruktor kann auf seine Komponenten zugreifen und Elemente auf Gleichheit oder Ungleichheit untersuchen.

- Der Mengenkonstruktor kann mittels Operatoren auf Elemente in der Menge zugreifen und testen, ob ein Element in der Menge enthalten ist. Außerdem sind die üblichen Mengenoperationen, wie Gleichheit, Teilmenge, echte Teilmenge, Vereinigung, Durchschnitt, Differenz, möglich.

- Der Listenkonstruktor kann die Liste mit Hilfe eines Iterators durchlaufen oder auf bestimmte Teile konkret zugreifen:

  first: Zugriff auf erstes Element
  next: Zugriff auf nächstes Element
  last: Zugriff auf letztes Element
  tail: Erstellung einer Teilliste

In objektorientierten Systemen werden Objekte gekapselt, d.h., daß Teile des Objekts nach außen hin nicht sichtbar sind. Der Grund dafür ist, daß ein Objekt nicht über die Datenstruktur, sondern auf abstrakter Ebene beschrieben wird. Man nennt dies auch abstrakter Datentyp (**ADT**).
Der Datentyp und die auf ihn definierten Operationen werden formal beschrieben, jedoch wird nichts über die Implementierung ausgesagt. Dies ist sinnvoll, da es für den Benutzer nur von Bedeutung ist, welche Art von Daten und zugehörige Operationen zur Verfügung stehen. Die interne Struktur ist nicht von Belang.

3.5.3 Klassen

In Klassen werden Objekte mit ähnlichen Eigenschaften und den auf ihnen definierten Operationen (Methoden) zusammengefaßt. Ein Objekt nennt man auch oft Instanz einer Klasse. Eine Instanz ist vergleichbar mit einem Tupel bei RDBMS. Ein Objekt wird durch die Instanzvariablen (entsprechen Attributen bei RDBMS) beschrieben. Eine

Klasse wird auch als Behälter angesehen, der Objekte beinhaltet, die dasselbe "Verhalten" aufweisen.
In OODBMS ist es möglich, daß Struktur und Methoden einer Klasse an eine Unterklasse durch die sogenannte Vererbung "übergeben" werden können. Dies bedeutet, daß Klassen, die ähnliche Eigenschaften haben, nicht erst neu definiert werden müssen, sondern Teile aus höheren Klassen genutzt werden können. In diesen Subklassen können dann zusätzlich neue Attribute und Operationen definiert werden. Beispielsweise haben Kunden und Mitarbeiter einer Firma eine Adresse. So wäre dann eine Klasse "Adresse" mit den Attributen *Straße, Postleitzahl* und *Ort*. Diese Attribute können nun sowohl an die Klasse "Kunde" als auch an die Klasse "Mitarbeiter" vererbt werden. Bei der Klasse "Kunde" käme dann z.B. noch *Kunden-Nr.* dazu. Bei der Klasse "Mitarbeiter" wären *Gehalt, Einstellungsdatum* etc. vorstellbar.
Manche OODBMS bieten neben der Einfach- auch Mehrfachvererbung. Bei der einfachen Vererbung kann eine Klasse nur von **einer** Oberklasse erben. Dies führt zu einer Baumstruktur. Bei der Mehrfachvererbung ist es möglich, von mehreren Oberklassen zu erben. Daraus resultiert ein gerichteter azyklischer Graph, der genau eine Wurzel besitzt.

In den letzten beiden Abschnitten wurden die Konzepte von relationalen und objektorientierten Datenbanksystemen vorgestellt. Um Datenbanken erstellen und auf die Daten zugreifen zu können, werden Datenbanksprachen benötigt.

3.6 Datenbanksprache

Eine Datenbanksprache besteht aus einer Datenbeschreibungssprache (Data Definition Language = DDL) und einer Datenmanipulationssprache (Data Manipulation Language = DML). Datenbanksprachen findet man bisher meist nur bei den relationalen DBMS. Bei einigen objektorientierten DBMS erfolgen Anfragen nur über die Programmiersprache, was umständlich und unkomfortabel ist. Deshalb wird eine Standardisierung von OODBMS in Form eines einheitliches Datenmodells und einer einheitlichen Anfragesprache gefordert.

Mit der DDL können Relationen definiert werden, z.B. die Attribute und ihre Domänen. Mit der DML können Daten eingegeben, verändert, gelöscht und abgefragt werden. Als Standard hat sich SQL (**S**tructured **Q**uery **L**anguage), entwickelt von IBM, herauskristallisiert.

Es gibt deskriptive und navigierende Datenmanipulationssprachen. Bei den deskriptiven DML müssen die Attribute bekannt sein und erfolgen meist im Dialog.

Beispiel: Gesucht sind alle Kunden, die in Stuttgart wohnen.

In SQL sieht die Anfrage folgendermaßen aus:

```
Select Name
from Kunde
where Ort="Stuttgart";
```

Bei den navigierenden DML wird nicht nach konkreten Werten der Attribute gefragt, sondern z.B. nach dem Artikel mit dem kleinsten Bestand.

3.7 Zusammenfassung

Relationale DBMS sind von der Struktur und den auf die Daten definierten Operationen her einfach in der Handhabung. Für den kaufmännischen Bereich, wie z.B. Auftragsbearbeitung, sind sie vollkommen ausreichend. Im technischen Bereich jedoch, wo es sich oft um komplexe Objekte handelt, genügen sie nicht. Komplexe Objekte lassen sich meist nicht durch Standarddatentypen beschreiben. Daher ist es erforderlich, eigene Typen definieren zu können.
Neben den unten aufgeführten, allgemeinen Anforderungen an Datenbanken, muß es auch möglich sein, individuelle Anforderungen zu erfüllen, die aus einer Anwendung heraus entstehen.

Diese allgemeinen Anforderungen sind:

- Vermeidung von Redundanzen - keine Mehrfachspeicherung von Daten
- Datenintegration - globale Datenbank für alle
- Datenunabhängigkeit - Daten getrennt von Anwendungsprogrammen
- Mehrbenutzerbetrieb
- Konsistenz der Daten - Richtigkeit der Daten
- Recovery - im Fehlerfall Wiederherstellung des alten Zustands

Die speziellen Anforderungen an DBMS bezüglich der Eignung für die technische Modellierung werden anhand des Szenarios im nächsten Kapitel konkretisiert.

4 Anforderungen der technischen Modellierung an Datenbanken

In Kapitel 3 wurden die allgemeinen Anforderungen an Datenbanken herausgearbeitet. Jede Anwendung hat ganz individuelle Eigenschaften, aus denen spezielle Anforderungen an Datenbanksysteme resultieren. In folgendem Kapitel sollen nun die besonderen Anforderungen des Szenarios an DBMS ermittelt werden. Das Szenario steht dabei beispielhaft für technische Anwendungen. Es enthält technische Objekte und Abhängigkeiten und Beziehungen der Objekte untereinander.

4.1 Konzept des Datenmodells

In diesem Kapitel wird Struktur des Datenmodells des Szenarios beschrieben. Dargestellt werden soll, welche Klassen benötigt werden und welche hierarchischen Zusammenhänge und Beziehungen der Klassen untereinander bestehen.

Die einzelnen Objekte, die in Kapitel 2 bei der Entwicklung des technischen Partialmodells herausgearbeitet wurden, sollen in separaten Klassen gespeichert sein. Dies ist notwendig, damit die Daten redundanzfrei vorliegen. Jedes Objekt wird nur einmal in der entsprechenden Klasse gespeichert. Die Zuordnung zu einer Baugruppe soll dann nur durch Referenzierung erfolgen. Im folgenden werden die Klassen **fett** und die Attribute *kursiv* dargestellt.

Das Datenmodell enthält folgende Klassen:

- Scheibe (**scheibe**) mit den Attributen *Radius, Dicke*
- mittlere Bohrung (**mbohr**) mit den Attributen *Radius, Tiefe, Mittelpunkt*
- Bohrkranz (**kranz**) mit den Attributen *Radius, Anzahl der Bohrungen*
- Kranzbohrungen (**kbohr**) mit den Attributen *Radius, Tiefe, Positionen, Kranz*. Das Attribut *Kranz* stellt die Zuweisung eines Bohrkranzes dar. Diese ist notwendig, da die Positionen der Bohrungen vom Radius des Bohrkranzes und von der Anzahl der Bohrungen abhängt.
- Achse (**achse**) mit den Attributen *Radius, Länge*

Die Klassen **scheibe**, **mbohr** und **achse** haben dieselbe Struktur, nämlich die eines Zylinders. Deshalb sollen diese Klassen Struktur und Methoden von einer Klasse Zylinder (**zyl**) mit den Attributen *Radius, Tiefe* erben.

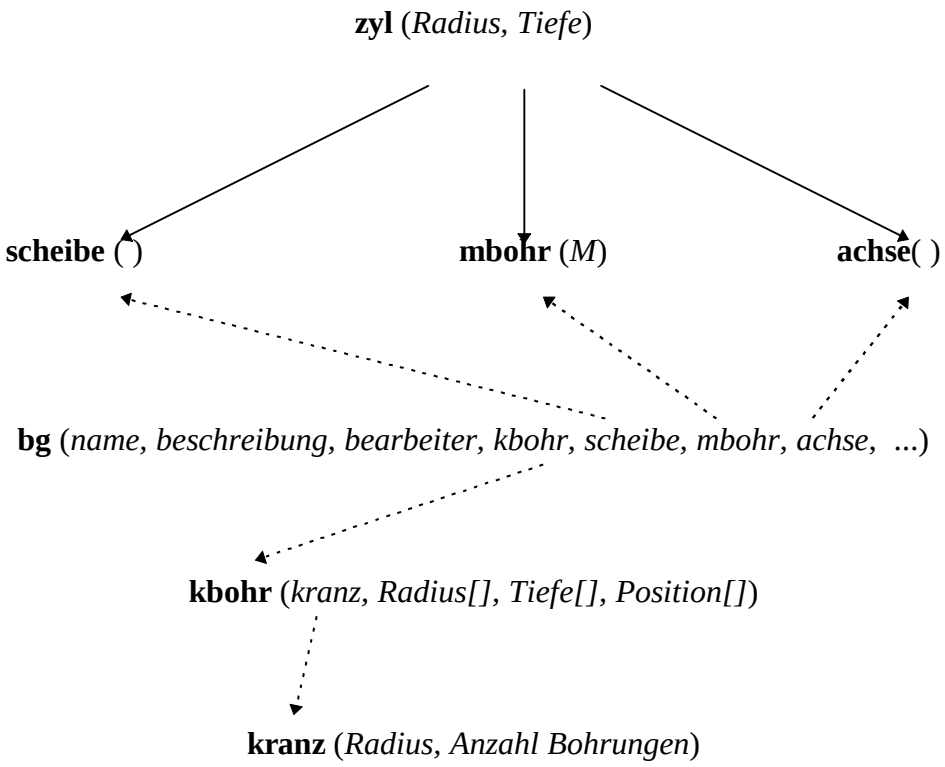
Die Klasse Baugruppe (**bg**) besitzt die Attribute:

- *Name* der Baugruppe
- *Beschreibung*
- *Bearbeiter*
- *Konstruktionsdatum*
- *Version*
- *Variante*
- *scheibe* (Referenz auf Instanz der Klasse **scheibe**)
- *mbohr* (Referenz auf Instanz der Klasse **mbohr**)
- *kbohr* (Referenz auf Instanz der Klasse **kbohr**)
- *achse* (Referenz auf Instanz der Klasse **achse**)
- *mbohrflag* (gibt an, ob mittlere Bohrung schon durchgeführt wurde)
- *mbohrzeit* (gibt die Zeit an, zu der die Bohrung durchgeführt wurde)
- *kbohrflag* (gibt an, ob die einzelnen Kranzbohrungen schon durchgeführt wurden)
- *kbohrzeit* (gibt die Zeiten an, zu denen die Kranzbohrungen durchgeführt wurden)
- *spezifisches Gewicht* des Materials
- *Gewicht*
- *Volumen*
- *Rasterbild*

Dabei sollen die Attribute *scheibe*, *mbohr*, *kbohr*, und *achse* nur Verweise auf die entsprechenden Instanzen der einzelnen Klassen darstellen. Auf den Bohrkranz muß in dieser Klasse nicht verwiesen werden, da dies in der entsprechenden Instanz von **kbohr** geschieht.

Durch diese Art und Weise der Verknüpfung von Klassen werden Redundanzen vermieden. Jede Scheibe wird nur einmal in der Instanz der Klasse **scheibe** abgespeichert, jedoch kann sie von mehreren Instanzen der Klasse Baugruppe (**bg**) referenziert werden. Dies gilt analog für die mittlere Bohrung, den Bohrkranz, die Kranzbohrungen und die Achse.

Abbildung 4.1 stellt das Konzept des Datenmodells des Szenarios dar. Zwischen der Klasse **zyl** und den Klassen **scheibe**, **mbohr** und **achse** besteht eine eindeutige hierarchische Abhängigkeit, d.h. **scheibe**, **mbohr** und **achse** erben die Attribute *Radius* und *Tiefe* von **zyl**. Dies wird durch die Pfeile mit durchgezogener Linie dargestellt. Die Klasse **mbohr** hat zusätzlich noch das Attribut *M* (Mittelpunkt). Die in der Graphik dargestellten Pfeile mit gestrichelter Linie von den Attributen zu den entsprechenden Klassen stellen die Verweise auf die jeweiligen Instanzen dar.



a ——→ b bedeutet: a vererbt an b

a ·····→ b bedeutet: a referenziert b

Abbildung 4.1: *Konzept des Datenmodells*

Auf der Basis dieses Konzepts sollen nun die speziellen Anforderungen an Datenbanksysteme herausgearbeitet werden.

4.2 Spezielle Anforderungen an DBMS

In diesem Kapitel werden die aus dem Szenario resultierenden speziellen Anforderungen an DBMS aufgezeigt. Diese Anforderungen der technischen Modellierung sind nicht an ein bestimmtes DBMS gebunden, sondern sollen allgemeingültig sein. Die Datenbanken sind anhand der folgenden Kriterien auf ihre Eignung für die technische Modellierung zu untersuchen:

Erstellung der Klassen

- **Vererbung**

  Bei der Erstellung der Klassen soll Mehrfachvererbung möglich sein. Im Szenario sollen Struktur und Methoden der Klasse **zyl** an die Klassen **scheibe**, **mbohr** und **achse** vererbt werden können. Im Szenario ist zwar nur Einfachvererbung gefordert, jedoch kann in anderen technischen Anwendungen Mehrfachvererbung notwendig sein.

- **Datenstrukturmöglichkeiten**

  Um das Szenario darstellen zu können, muß es möglich sein, im DBMS die benötigten Datentypen zu definieren bzw. auf vordefinierte Typen zugreifen zu können.

  Folgende Datentypen sind im Szenario vorhanden:

  - Reelle Zahlen (Radien, Tiefen, Gewicht und Volumen)
  - Integer Zahlen (Anzahl der Bohrungen)
  - unbegrenzter Text (Name, Beschreibung und Bearbeiter)
  - logischer Typ (mbohrflag und kbohrflag)
  - Koordinaten (Positionen der Bohrungen)
  - Datum (Konstruktionsdatum)
  - Zeit (mbohrzeit, kbohrzeit)
  - unbegrenzte Listen (In der Klasse **kbohr** sind die Attribute *Radius, Tiefe* und *Position* unbegrenzte Listen von Radien, Tiefen, Positionen, da die Anzahl der Bohrungen variabel ist. Ebenso ist *kbohrflag* eine unbegrenzte Liste mit logischen Einträgen und *kbohrzeit* eine unbegrenzte Liste mit Zeiteinträgen.)

  Zu untersuchen sind auch die Datenstrukturmöglichkeiten für die Attribute *scheibe, mbohr, kbohr, achse, kranz*, die die Referenzen zu den Instanzen in den jeweiligen Klassen darstellen. Zudem sind die Möglichkeiten zur Erstellung eines Rasterbildes, wie z.B. durch Aufruf eines externen Programmes, und der Definition komplexer Datentypen zu überprüfen.

- **Darstellung von Beziehungen von Objekten**

  Die Beziehung der Klasse **bg** (Baugruppe) zu den einzelnen Objekten soll durch Verweise auf die einzelnen Instanzen realisiert werden können. Falls dies nicht möglich ist, müssen die Beziehungen zwischen den Objekten anderweitig erfolgen.

Erstellung und Manipulation der Daten

- **Default-Belegungen**

  Bei der Erstellung von Objekten sollen Belegungen von Attributen mit Default-Werten möglich sein. Beispielsweise soll bei Anlegung der Achse eine Default-Länge vorgegeben werden können. Außerdem soll festgestellt werden können, ob dieser Default-Wert verändert wurde. Eine Möglichkeit der Realisierung könnte ein boolsches Flag sein, das "true" gesetzt wird, wenn der Wert verändert wird.

- **Festlegung und Überprüfung von sinnvollen Werten**

  Sinnvolle Werte für die Länge der Achse sollen festgelegt und überprüft werden können. Die Länge der Achse sollte beispielsweise mindestens doppelt so groß sein wie die Dicke der Scheibe. Falls dies nicht der Fall ist, sollte eine Warnmeldung ausgegeben werden, die den Benutzer darauf hinweist, daß der Wert unterschritten wurde. Eventuell könnte der Benutzer solange gezwungen werden, diesen Wert zu verändern, bis er in den richtigen Grenzen liegt.
  Wenn noch nicht alle Daten spezifiziert sind, sollen Gewicht und Volumen sinnvolle Werte liefern, d.h., wenn beispielsweise die Maße der Scheibe nicht vorhanden sind, dagegen die Werte für die Bohrungen schon vorliegen, kämen bei der Berechnung von Gewicht bzw. Volumen negative Werte heraus. In diesem Fall sollen Gewicht und Volumen noch nicht berechnet werden bzw. auf den Wert 0 gesetzt werden.

- **Konsistenzprüfungen**

  Zur Vermeidung von Fehlkonstruktionen müssen die in Kapitel 2.5.5 herausgearbeiteten Konsistenzprüfungen durchgeführt werden. Diese müssen solange erfolgen, bis alle Maße sinnvolle Werte haben, um das Werkstück produzieren zu können.

Abfrage von Daten

- **Eindeutige Objektidentität (OID)**

  Das DBMS soll OID`s (siehe Kapitel 3.5.1) zur eindeutigen Identifizierung der Instanzen zur Verfügung stellen. Bei RDBMS werden Instanzen über den Primärschlüssel identifiziert, dessen Attribute nicht NULL sein dürfen. Falls aber doch ein Attribut nicht bekannt ist, so ist eine Instanz nicht mehr eindeutig. Um dem abzuhelfen, sollte eine Objektidentität vergeben werden.

- **Abfragesprache**

  Neben einer Datendefinitionssprache, mit der Klassen erstellt werden können, sollte auch eine Datenmanipulationssprache zur Verfügung gestellt werden. Damit können Daten erstellt, verändert und gelöscht werden. Dies ermöglicht die Interaktion des Benutzers mit der Datenbank ohne aufwendige Implementierung eines Anwendungsprogramms. Zudem soll die Möglichkeit bestehen, sowohl alle als auch einzelne, bestimmte Instanzen abzufragen.

- **Zugriff aus einer Programmiersprache heraus**

  Neben einer Abfragesprache sollte es auch den Weg geben, aus einer Programmiersprache heraus auf die Daten zugreifen zu können. Dies ermöglicht beispielsweise die Implementierung spezieller und komplexer Funktionen, die das DBMS nicht zur Verfügung stellt.

- **Benutzerinteraktion**

  Um die Interaktion des Benutzers mit der Datenbank zu ermöglichen, ohne daß dieser alle dafür notwendigen Befehle wissen muß, sollten Mittel zur Erstellung von z.B. Eingabemasken und Abfrageformularen existieren.

Zusätzliche Anforderungen an DBMS

- **Realisierung der technisch-funktionalen Abhängigkeiten**

  Bei Änderung der Werte bestimmter Attribute, sollen auch die Werte der davon abhängigen Attribute automatisch angepaßt werden (aktive Datenbank).

  Abhängigkeiten dieser Art sind:

  - Dicke der Scheibe $\longrightarrow$ Tiefe der mittleren Bohrung
  - Radius Achse $\longleftrightarrow$ Radius der mittlere Bohrung
  - Radius Kranz, Anzahl der Bohrungen $\longrightarrow$ Positionen der Kranzbohrungen

  Bei Eingabe oder Änderung der Dicke der Scheibe muß die Tiefe der mittleren Bohrung auf denselben Wert gesetzt bzw. geändert werden. Die Positionen der Bohrungen auf dem Kranz werden durch den Radius des Kranzes und der Anzahl der Bohrungen bestimmt. Dies gilt jedoch nur für einen symmetrischen Bohrkranz. Zusätzlich sollen die Positionen der Bohrungen frei editierbar sein.
  Im Gegensatz dazu existiert eine bidirektionale Abhängigkeit zwischen den Radien der Achse und der mittigen Bohrung. Bei Änderung eines Wertes muß das jeweils andere Attribut angepaßt werden. Dies wird oben durch einen Doppelpfeil verdeutlicht.

- **Versions- und Variantenbildung**

  Ein Produkt wird im allgemeinen nicht sofort komplett spezifiziert. Infolgedessen soll die Datenbank verschiedene Versionen und Varianten von Instanzen verwalten können. Versions- und Variantenbildung liegt beim Anwender. Beispielsweise kann bei geringfügigen Änderungen von Maßen einzelner Parameter eine neue Version definiert werden. Dagegen kann bei strukturellen Änderungen, wie z.B. Wegfallen der Achse, eine neue Variante festgelegt werden.

Die speziellen Anforderungen der technischen Anwendungen an DBMS beinhalten die Verwaltung und Bereitstellung von Produktdaten. Dabei spielt die Verwaltung komplexer Daten eine große Rolle.

4.3 Zusammenfassung

In diesem Kapitel wurden die internen Strukturen des Datenmodells des Szenarios aufgezeigt. Zur Umsetzung des Modells in ein Datenbanksystem wurden die daraus resultierenden Anforderungen an DBMS beschrieben. Bei der Erstellung der Klassen soll Mehrfachvererbung, Definition komplexer Datentypen und die Realisierung von Beziehungen von Objekten möglich sein. Bei der Erstellung und Manipulation von Daten sollen Default-Belegungen von Attributwerten, Feststellung der Änderung dieser Default-Belegungen, Überprüfung sinnvoller Werte und Konsistenzprüfungen realisierbar sein. Um die Daten abfragen zu können, sollte das DBMS eine OID zur eindeutigen Identifizierung der Objekte zur Verfügung stellen. Neben einer Abfragesprache sollte auch der Zugriff aus einer Programmiersprache heraus ermöglicht werden. Zusätzlich sollten Tools zur Erstellung von Eingabemasken und Abfrageformularen für die Benutzerinteraktion im Bereich des Möglichen sein. Ferner wird von den DBMS die Realisierung der technisch-funktionalen Abhängigkeiten und die Verwaltung von Versionen und Varianten gefordert.

In den nächsten beiden Kapiteln sollen nun die ausgewählten DBMS daraufhin untersucht werden, inwieweit sie diese Anforderungen ganz, bedingt oder nicht erfüllen. Das Ergebnis der Analyse soll aufzeigen, wie geeignet die DBMS für die technische Modellierung sind.

5 Eignungsanalyse von Postgres für die technische Modellierung

In Kapitel 4 wurden die speziellen Anforderungen an Datenbanken erarbeitet. Nach einer kurzen Einführung in Postgres sollen diese Anforderungen validiert und prototypisch implementiert werden.

5.1 Einführung

Postgres wurde an der University of Berkeley, Kalifornien unter der Leitung von Michael Stonebraker entwickelt. Es ist der Nachfolger des relationalen Datenbanksystems Ingres (**Post** In**gres**). Postgres wurde um abstrakte Datentypen (**ADT**), benutzerdefinierte Funktionen und Vererbung erweitert. Postgres kann sowohl als erweitertes relationales als auch als objektorientiertes DBS bezeichnet werden.

Postgres beruht weitgehend auf dem Relationenmodell. Zusätzlich ist es mit folgenden objektorientierten Merkmalen ausgestattet:

- Vergabe eines eindeutiger Object-Identifier vom System
- abstrakte Datentypen
- Klassen
- Vererbung
- benutzerdefinierte Funktionen

Wenn „Klasse“ durch „Relation“ und „Instanz“ durch „Tupel“ ersetzt wird, erhält man ein relationales DBS.
Es existiert keine eindeutige Aussage, ob Postgres nun objektorientiert oder erweitert relational ist. Nach Meinung der Entwickler ist es jedoch eher als erweitertes relationales DBMS anzusehen [SRH94].

Zu Postgres wurde die SQL-ähnliche Datenbanksprache namens **Postquel** entwickelt. Postquel ist sowohl Datendefinitions- als auch Datenmanipulationssprache, d.h. mit ihr können Daten definiert, verändert und abgefragt werden.

Im folgenden Unterkapitel wird untersucht, inwieweit Postgres den allgemeinen Anforderungen an DBMS aus Kapitel 3 gerecht wird.

5.2 Untersuchung von Postgres anhand der allgemeinen Anforderungen

In Kapitel 3 wurden die relationalen und objektorientierten Datenbanken vorgestellt und die Anforderungen, wie Vermeidung von Redundanzen, Konsistenz der Daten, Recovery etc., an sie herausgearbeitet. Postgres soll nun darauf hin untersucht werden, ob und wie diese Anforderungen erfüllt werden.

Die Vermeidung von Redundanzen erfolgt durch das in Kapitel 4 dargestellte Konzept des Datenmodells. Alle Objekte, wie Scheibe, mittlere Bohrung, Bohrkranz und Achse, werden in separaten Klassen gespeichert. Verweise auf die entsprechenden Instanzen werden in der Klasse Baugruppe abgelegt. Somit kann auf eine Scheibe beispielsweise mehrmals verwiesen werden, aber sie wird nur einmal in der dazugehörigen Klasse gespeichert.

Postgres bietet Datenunabhängigkeit, d.h. Anwendungsprogramme und Daten sind getrennt voneinander. Applikationen können verändert werden, ohne daß dies Einfluß auf die Daten hat. Anwendungen können in der Programmiersprache C implementiert werden. Postgres stellt eine Schnittstelle zu C (**LIBPQ**) zur Verfügung, mit der auf die Daten zugegriffen werden kann.

Die Prozeßstruktur von Postgres beruht auf dem “process-per-user-model“ [SR94]. Das heißt, es existiert genau ein DBMS-Prozeß pro Anwendungsprogramm. Dieser DBMS-Prozeß heißt **postmaster**. Der **postmaster** stellt die Verbindung zwischen der Applikation und der Datenbasis her.

Die Konsistenz der Daten wird dadurch gewährleistet, daß Postgres nach dem ACID-Prinzip arbeitet. Alle Befehle, die zwischen einem “begin“ und einem “end“ stehen, werden erst dann vollständig durchgeführt, wenn die Transaktion durch ein “commit“ beendet wird. Das heißt, die Datenbank wird von einem konsistenten Zustand in einen anderen konsistenten Zustand geführt. Wird jedoch die Transaktion abgebrochen (“abort“), wird die Datenbank wiederum in einen konsistenten Zustand geführt, nämlich in den, der vor der Transaktion bestand.

Die Wiederherstellung (Recovery) des ursprünglichen Zustands bei einem Fehlerfall ist ebenfalls gewährleistet. Alle während eines “crashs“ laufenden Transaktionen, die in einen “begin“ - “end“ - Block eingebettet sind, werden als “abgebrochen“ markiert und deshalb nicht ausgeführt. Somit wird der ursprüngliche Zustand wieder hergestellt.

Es ist deutlich geworden, daß Postgres die allgemeinen Anforderungen erfüllt, die an DBMS gestellt werden. In den nächsten Punkten sollen nun die Datenstruktur-möglichkeiten von Postgres und die speziellen Anforderungen, die aus dem Szenario (und damit aus der technischen Modellierung) resultieren, untersucht werden.

5.3 Untersuchung von Postgres anhand der speziellen Anforderungen

Postgres soll nun auf die in Kapitel 4 beschriebenen Anforderungen technischer Anwendungen an DBMS untersucht werden. Die speziellen Anforderungen sind unterteilt in Anforderungen bei der Erstellung von Klassen, Erstellung und Manipulation von Daten, Abfrage von Daten und Zusätzliche Anforderungen an DBMS.

5.3.1 Erstellung von Klassen

Bei der Erstellung von Klassen wird Mehrfachvererbung gefordert. Zu untersuchen sind auch die Datenstrukturmöglichkeiten und die Darstellung von Beziehungen von Objekten.

- **Vererbung**

  In Postgres ist lediglich Einfachvererbung möglich. Beispielsweise definiert man eine Klasse **zyl** mit den Attributen *Radius* und *Tiefe* folgendermaßen:

  ```
  create zyl (radius=float8, tiefe=float8)
  ```

  Die Klassen **scheibe**, **mbohr** und **achse** können von **zyl** wie folgt erben:

  ```
  create scheibe () inherits (zyl)
  create mbohr (M=point) inherits (zyl)
  create achse () inherits (zyl)
  ```

  Bei **mbohr** kommt noch der Mittelpunkt als zusätzliches Attribut hinzu.

- **Datenstrukturmöglichkeiten**

  Postgres stellt eine Reihe von Datentypen zur Verfügung:

  - Für reelle Zahlen existieren die Datentypen `float4` und `float8`, die sich nur in der Genauigkeit unterscheiden.

  - Ganze Zahlen werden durch `int2` oder `int4` definiert. Sie unterscheiden sich ausschließlich in der Größe.

  - Für einzelne Zeichen existiert der Datentyp `char` und für Zeichenketten `char16` oder `text`. In `char16` können 16 Zeichen eingetragen werden, wogegen `text` von variabler Länge ist.

- Logische Attribute werden durch `bool` dargestellt.

  Beispiel: `wahr = “T“`

- Datum und Zeit werden durch den Typ `abstime` repräsentiert.

  Monat Tag [Stunde : Minute : Sekunde] Jahr [Zeitzone]

wobei	Monat	Jan, Feb, ..., Dec	ist,
	Tag	1, 2, ..., 31	ist,
	Stunde	00, 01, ..., 24	ist,
	Minute	00, 01, ..., 59	ist,
	Sekunde	00, 01, ..., 59	ist,
	Jahr	1970, 1971, ..., 2038	ist.

  Beispiel: `Datum = “May 24 00:00:00 1995 GMT“`

- Koordinaten lassen sich durch `point` darstellen.

  Beispiel: Punkt = “(4.0,0.5)“

- Es lassen sich unbegrenzte Listen von jedem Typ definieren. In der Klasse **kbohr** können die Attribute wie folgt definiert werden:

  Radien: `float8[]`
  Tiefen: `float8[]`
  Positionen: `point[]`

  In der Klasse **bg** werden unbegrenzte Listen für die Attribute *kbohrflag* und *kbohrzeit* benötigt.

  kbohrflag: `bool[]`
  kbohrzeit: `abstime[]`

  Beispiel:
  ```
  kbohrflag = “{“t“,“f“,“f“,“f“}“
  kbohrzeit = “{“Jan 5 1995“}“
  ```

- Die Attribute *scheibe, mbohr, kbohr* und *achse* in der Klasse **bg** und das Attribut *kranz* in der Klasse **kbohr** können unterschiedlichen Typs sein.

  Postgres stellt den Typ `oid` zur Verfügung, der die Objektidentität einer Instanz repräsentiert. Der Wert des Attributs *scheibe* in der Klasse **bg** ist die Objekt-ID der Instanz der Klasse **scheibe**, die den gewünschten Radius und die Dicke beschreibt. Durch geeignete Abfragen kann auf diese Werte zugegriffen werden.

Die zweite Möglichkeit besteht darin, den Typ `postquel`[1] in Anspruch zu nehmen. Der Wertebereich dieses Typs beinhaltet jegliche Abfrage, die in Postgres existieren kann. Somit könnte die oben erwähnte, geeignete Abfrage direkt im Attribut abgespeichert werden. Infolgedessen wird automatisch die richtige Scheibe durch Zugriff auf das Attribut *scheibe* in **bg** in der Klasse **scheibe** gesucht.

Beispiel:

```
create bg (name=text, scheibe=postquel ...)
append bg (name="Baugruppe1",scheibe="retrieve
          scheibe.all where scheibe.oid="123" ")
```

Die Scheibe mit der Objektidentität "123" hat den gewünschten Radius und die gewünschte Dicke. Um den Inhalt der Scheibe "123" zu erhalten, existieren zwei Möglichkeiten:

```
retrieve (bg.scheibe) where bg.name="Baugruppe1"
```

oder

```
execute (bg.scheibe) where bg.name="Baugruppe1"
```

- Die Erzeugung des Rasterbildes der Baugruppe soll von einer externen Routine erfolgen. In Postgres gibt es die Möglichkeit, eine C-Prozedur mit dem Typ `cproc`[2] zu definieren.

  Beispiel:

  ```
  create bg (name=text, rasterbild=cproc...)
  append bg (name="Baugruppe1", rasterbild="..source
            code..")
  ```

  Im Attribut *rasterbild* ist der source-code der Prozedur gespeichert, die die entsprechenden Objekte zeichnet. Das Problem besteht darin, daß in jeder Instanz die Prozedur eingegeben werden muß. Dem kann abgeholfen werden, indem man die Prozedur in einer eigenen Klasse abspeichert.

  ```
  create zeichnung (name=text, proc=cproc)
  append zeichnung (name="raster", proc="..source
                   code..")
  ```

  Diese Prozedur kann durch den Befehl

  ```
  execute (zeichnung.proc) with ("Baugruppe1") where
          zeichnung.name = "raster"
  ```

  ausgeführt werden.

---

[1] in der Version 4.01 noch nicht realisiert (siehe auch Kapitel 7)

[2] in der Version 4.01 noch nicht realisiert (siehe auch Kapitel 7)

- In Postgres können eigene Datentypen definiert werden. Notwendig hierfür sind eine Input- und eine Output-Funktion, welche die externe Repräsentation (Zeichen) in die interne Repräsentation (z.B. Integer) umwandeln und umgekehrt. Ein ausführliches Beispiel hierfür findet der interessierte Leser im Postgres User Manual.

Neben den hier vorgestellten existieren weitere vordefinierte Typen, die allerdings für das vorgegebene Szenario nicht benötigt werden. Im Referenzhandbuch sind alle zur Verfügung stehenden Datentypen aufgeführt.

Das Szenario kann beinahe vollständig durch die von Postgres mitgelieferten Datentypen realisiert werden. Die Erzeugung des Rasterbildes erfordert allerdings die Programmierung einer geeigneten Prozedur.

- **Darstellung von Beziehungen von Objekten**

  Es gibt mehrere Möglichkeiten, Beziehungen zu realisieren. Wie unter Punkt **Datenstrukturmöglichkeiten** schon erwähnt, erfolgt dies über den Typ `oid` und entsprechende Abfragen oder direkt über den Typ `postquel`. Eine weitere Möglichkeit bieten die sogenannten **rules**. In einer rule kann folgendes definiert werden:

  Bei der Ausführung eines Postquel-Befehls soll statt dessen oder zusätzlich ein anderer Postquel-Befehl ausgeführt werden.

  Beispiel:

  Der Benutzer greift auf das Attribut *scheibe* in der Klasse **bg** zu. Die entsprechende Regel sorgt dafür, daß statt dessen auf die Instanz in der Klasse **scheibe** mit der OID gleich der OID, die im Attribut *scheibe* steht, zugegriffen wird.

5.3.2 Erstellung und Manipulation von Daten

Bei der Erstellung und Manipulation von Daten wird die Möglichkeit der Festlegung von Default-Werten, der Festlegung und Überprüfung von sinnvollen Werten und der Durchführung von Konsistenzprüfungen gefordert.

- **Default-Belegungen**

  Default-Belegungen, wie z.B. die Vorgabe der Länge bei der Erstellung der Achse, sind in Postgres nicht vorgesehen. Wenn dies gewünscht oder benötigt wird, so muß das über ein Anwendungsprogramm erfolgen. Jedoch kann mit Hilfe einer

Regel ein boolsches Flag realisiert werden, das “true“ gesetzt werden soll, sobald der Default-Wert geändert wird.

- **Festlegung und Überprüfung von sinnvollen Werten**

  Da keine vordefinierten Größen, wie unter Punkt **Default-Belegungen** bereits erwähnt, möglich sind, ist auch die Festlegung von sinnvollen Werten in Postgres nicht vorgesehen. Ebenso wird die Überprüfung dieser Werte von Postgres nicht unterstützt, sondern muß vom Anwendungsprogramm erledigt werden.

- **Konsistenzprüfungen**

  Die Konsistenzprüfungen, die Fehlkonstruktionen verhindern sollen, können nur im Anwendungsprogramm vorgenommen werden. Dazu werden zunächst die Daten in C eingelesen. Daraufhin werden die Konsistenzprüfungen durchgeführt. Der Anwender wird solange „gezwungen“, die Werte zu verändern, bis alle Bedingungen der Konsistenzprüfungen erfüllt sind. Danach werden die Werte in entsprechende Formate umgewandelt und in die einzelnen Klassen geschrieben.

5.3.3 Abfrage von Daten

Zur eindeutigen Identifizierung eines Objekts wird die Vergabe einer Objektidentität gefordert. Zur Abfrage der Daten soll eine Abfragesprache zur Verfügung stehen. Außerdem soll zusätzlich die Möglichkeit bestehen aus einer Programmiersprache auf die Daten zugreifen zu können. Zudem sind die Möglichkeiten für die Benutzerinteraktion zu untersuchen.

- **Eindeutige OID**

  Postgres vergibt bei der Erstellung eines Objekts eine eindeutige Objektidentität in Form einer Nummer, die aber als Zeichenkette vorliegt. Somit liegt jede Instanz global eindeutig in der Datenbank vor.

- **Abfragesprache**

  Die Datendefinitions- und Datenmanipulationssprache von Postgres ist **Postquel**. Durch sie werden die Klassen erstellt, Instanzen eingetragen, verändert, gelöscht und abgefragt.
  Postgres stellt den sogenannten **Monitor** zur Verfügung, mit dem interaktiv Postquel-Befehle ausgeführt werden können.
  Postquel wird in den Kapiteln 5.5.1 und 5.5.2 eingehend erläutert.

- **Zugriff aus einer Programmiersprache heraus**

  Die Schnittstelle zwischen Postgres und C heißt LIBPQ. Sie stellt Bibliotheken zur Verfügung, die es ermöglichen, von C aus auf Instanzen der Datenbank zuzugreifen, diese zu verändern und dann wieder zurückzuschreiben.

  Mit dem Befehl:

  ```
  PQexec (query)
  ```

  kann jeglicher Postquel-Befehl ausgeführt werden.
  Neben LIBPQ existiert eine Schnittstelle zu LISP: **CLOS** (**C**ommon **L**ISP **O**bject **S**ystem). Diese soll hier jedoch nicht weiter betrachtet werden.

  Postgres ist nicht an eine bestimmte Programmiersprache gebunden. Es bietet vielmehr die Möglichkeit, von vielen Programmiersprachen genutzt werden zu können. Dazu ist jedoch die Implementierung der Schnittstellen nötig.

- **Benutzerinteraktion**

  Die Benutzeroberfläche für Postgres heißt PICASSO. Derzeit existiert aber lediglich eine Implementierung für SUN-Rechner. Daher stand PICASSO der Autorin nicht zur Verfügung. Eine Portierung auf andere Rechner ist geplant.

5.3.4 Sonstige Anforderungen an DBMS

Die sonstigen Anforderungen beinhalten die Realisierung der technisch-funktionalen Abhängigkeiten und die Versions- und Variantenverwaltung.

- **Realisierung der technisch-funktionalen Abhängigkeiten**

  Die technisch-funktionalen Abhängigkeiten können durch die schon oben erwähnten rules verwirklicht werden. Im Szenario soll die Tiefe der mittleren Bohrung automatisch bei Änderung der Dicke der Scheibe angepaßt werden. Hierfür könnte eine rule definiert werden, die die Tiefe der mittleren Bohrung ändert, sobald die Dicke der Scheibe geändert wird. Da jedoch im Szenario in der Baugruppe nur Verweise auf die einzelnen Komponenten der Baugruppe bestehen, muß in diesem Fall in der Klasse **scheibe** eine vorhandene Scheibe mit der gewünschten Dicke gesucht, oder bei nicht Vorhandensein eine neue erstellt, und deren OID in das Attribut *scheibe* in der Klasse **bg** eingetragen werden. Daraufhin muß in der Klasse **mbohr** eine neue Bohrung gesucht werden, die denselben Radius hat, wie die ursprüngliche mittlere Bohrung, jedoch die neue gewünschte Tiefe. Die OID dieser Instanz muß dann ebenso in **bg**, und zwar in das Attribut *mbohr*, eingetragen werden.

  Da die rules solche komplexen Aktionen nicht bewerkstelligen können, müssen im Szenario die technisch-funktionalen Abhängigkeiten über das Anwendungsprogramm realisiert werden.

- **Versions- und Variantenbildung**

  Postgres ermöglicht die Bildung der Version einer Klasse mit dem Befehl:

  ```
  create version classname1 from classname2 [[abstime]]
  ```

  classname1 ist eine Version von **classname2**. Wenn Änderungen in **classname1** vorgenommen werden, unterscheiden sich die Inhalte der beiden Klassen. Werden jedoch Änderungen in **classname2** durchgeführt, so werden diese an die Versionsklasse **classname1** weitergeleitet.
  Im Szenario werden jedoch Versionen und Varianten von Instanzen gefordert. In Postgres gibt es indirekt die Möglichkeit, Versionen und Varianten zu verwalten. Bei Änderungen von Instanzen überschreibt Postgres die Werte nicht, sondern speichert die alten Werte, um die sogenannte „Zeitreise“ zu ermöglichen. Das heißt, der Inhalt der Klasse bleibt von der Erstellung bis hin zur Löschung erhalten. Wird nun bei jeder Änderung einer Instanz eine neue Versions- oder Variantennummer vergibt, erhält man die gewünschten Versionen und Varianten.

  Beispiel:

  ```
  create test (name=text, radius=float8, vers=int2,
              variante=int2)
  append test (name="Teil1", radius=2.5, vers=1,
              variante=1)
  replace test (radius=2.0, vers=2) where name="Teil1"
  replace test (radius=10.0, vers=1, variante=2) where
               name="Teil1"
  ```

  (Anmerkung: Version ist ein Schlüsselwort und darf deshalb nicht als Attributname verwendet werden. Daher die Namensgebung “vers“.)

  Mit dem Standardbefehl für Abfragen
  `retrieve (test.all)` erhält man:

name	radius	vers	variante
Teil1	10.0	1	2

Abbildung 5.1: *Inhalt der Klasse test*

Abbildung 5.1 zeigt den aktuellen Inhalt der Klasse **test**.
Um alle Versionen und Varianten zu erhalten, ist es nötig, dem `retrieve`-Befehl Zeitparameter hinzuzufügen. Der Befehl

```
retrieve (T.all) from T in test ["Jan 1 1970","now"]
          where name="Teil1"
```

erzeugt die Ausgabe:

name	radius	vers	variante
Teil1	2.5	1	1
Teil1	2.0	2	1
Teil1	10.0	1	2

Abbildung 5.2: *Inhalt der Klasse test mit Zeitangabe*

Abbildung 5.2 zeigt den Inhalt der Klasse **test** von der Erstellung bis zum Zeitpunkt des Aufrufs des Befehls mit allen Änderungen.
Die Zeitspanne [“Jan 1 1970“,“now“] kann mit [,] abgekürzt werden. Es können natürlich auch jegliche andere Zeiträume abgefragt werden. In der der Autorin zur Verfügung stehenden Version 4.01 muß bei Inanspruchnahme von Zeitspannen ein Surrogat (in obigem Beispiel: retrieve (**T**.all) **from T in test** ...) verwendet werden. Werden nur die Werte einer bestimmten Version gewünscht, erfolgt dies beispielsweise über die Anfrage:

```
retrieve (T.all) from T in test [,] where
          T.name=“Teil1“ and T.vers=2 and T.variante=1
```

Dies bedingt die Ausgabe:

name	radius	vers	variante
Teil1	2.0	2	1

Abbildung 5.3: *Konkrete Instanz der Klasse test*

Abbildung 5.3 zeigt den Inhalt der konkret abgefragten Instanz.
Die „Zeitreise“ bietet somit eine effiziente Versions- und Variantenverwaltung. Ohne diese Möglichkeit müßte bei jeder neuen Version oder Variante zuerst der Inhalt der jeweiligen Instanz kopiert werden, um dann die entsprechenden Werte verändern zu können.

Zusammenfassend kann gesagt werden, daß Postgres den meisten Anforderungen genügt. Die Realisierung einiger weniger Punkte der Anforderungen muß aber individuell implementiert werden.
Im nächsten Schritt sollen nun die wesentlichen Anforderungen implementiert werden. Die Implementierung erfolgt nur prototypisch, da das gesamte Modell zu umfangreich wäre.

5.4 Prototypische Implementierung

Zunächst werden die grundsätzlichen Befehle zur Erstellung von Klassen und Objekten, sowie zur Manipulation der Daten vorgestellt. Anschließend werden beispielhaft die Möglichkeiten der Schnittstelle zwischen Postgres und C (**LIBPQ**) untersucht.

5.4.1 Erstellung der Klassen

In Postgres werden die Klassen wie folgt erstellt:

```
create zyl (radius=float8, dicke=float8)
create kranz (radius=float8, n=int4)
create kbohr (kranz=oid, radius=float8[], tiefe=float8[],
              p=point[])
create bg (name=text, beschreibung=text, bearbeiter=text,
           konstrdat=abstime, spezGewicht=float8,
           vers=int4, variante=int4, scheibe=oid,
           mbohr=oid, kbohr=oid, achse=oid, mbohrflag=bool,
           kbohrflag=bool[], mbohrzeit=abstime,
           kbohrzeit=abstime[], volumen=float8,
           gewicht=float8)
```

Für Klassen, die von einer anderen Klasse erben, lautet der Befehl:

```
create scheibe () inherits (zyl)
create mbohr (M=point) inherits (zyl)
create achse () inherits (zyl)
```

Wenn zu einer Klasse noch Attribute hinzugefügt werden sollen, so geschieht dies durch den Befehl[3]:

```
addattr (attname1 = type1 {,attnamei = typei}) to
         classname{*}
```

Zur Darstellung der Beziehungen zwischen den Objekten wurde der Typ `oid` gewählt, da der Typ `postquel` in der der Autorin zur Verfügung stehenden Version 4.01 nicht verfügbar war.

Nach der Beschreibung zur Erstellung der Klassen soll nun die Vorgehensweise zur Eingabe, Änderung, Löschung und Abfrage von Daten aufgezeigt werden.

[3] Zur genauen Syntax der Befehle wird auf das Referenz-Handbuch verwiesen.

5.4.2 Eingabe, Änderung, Löschen und Abfrage von Daten

Die einzelnen Befehle werden im folgenden beispielhaft anhand der Klasse **scheibe** dargestellt.

Daten werden mit dem Befehl `append` eingegeben.

```
append scheibe (radius=5.4, tiefe=1.0)
```

Verändert werden die Daten mit dem Befehl `replace.`

```
replace scheibe (radius=5.2) where scheibe.radius=5.4
```

Daten können mittels des Befehls `delete` gelöscht werden.

```
delete scheibe where scheibe.radius=5.2
```

Dieser Befehl löscht alle Instanzen aus der Klasse **scheibe** mit dem Radius 5.2 .

```
delete scheibe
```

löscht alle Instanzen aus der Klasse **scheibe**.

Mit dem Befehl `retrieve` können konkrete oder auch alle Instanzen abgefragt werden.

```
retrieve (scheibe.all)
```

listet alle Instanzen der Klasse **scheibe** auf.

```
retrieve (scheibe.dicke) where scheibe.radius=5.2
```

liefert als Resultat alle Dicken der Scheiben mit dem Radius 5.2 .

Um den Klassennamen nicht immer ausschreiben zu müssen, gibt es die Möglichkeit, sogenannte Surrogate zu benutzen. In folgendem Beispiel wird die Klasse **scheibe** durch das (kürzere) Surrogat "s" ersetzt. Die Definition des Surrogats erfolgt durch den Zusatz "`from s in scheibe`".

```
retrieve (s.dicke) from s in scheibe where s.radius=5.2
```

Alle Postquel-Befehle können mit Hilfe des Postgres-Monitors interaktiv ausgeführt werden. Jedoch besteht nur die Möglichkeit der Eingabe und Abfrage konkreter Werte. Wenn die Inhalte von Variablen in die Datenbank geschrieben werden sollen, muß auf ein Anwendungsprogramm zurückgegriffen werden. Im nächsten Abschnitt wird nun LIBPQ - die Schnittstelle zu C - untersucht.

5.4.3 LIBPQ - Die Schnittstelle zu C

Die Bibliothek LIBPQ beinhaltet eine Vielzahl von Befehlen, die es ermöglichen, auf die Datenbasen aus C-Programmen zuzugreifen. In diesem Kapitel werden exemplarisch die zwei für das Szenario wichtigen Befehle erklärt. Ausführliche Informationen über alle Befehle können im Referenz-Handbuch [Pos90 S.93 ff] nachgelesen werden.

Mit dem Befehl

```
PQexec(query)
```

kann jeder Postquel-Befehl ausgeführt werden.

Beispiel:

```
PQexec(“append scheibe (radius=4.5, tiefe=2.2“)
```

Mit dem Befehl

```
PQputline (String)
```

kann ein String in eine Klasse geschrieben werden.

Beispiel:

```
PQexec (“copy scheibe from stdin“);/*Klasse angeben*/
PQputline (“4.5<TAB[4]>2.2“); /*Radius und Tiefe übergeben*/
PQputline (“.\n“);          /*“.“ und Zeilenende anhängen*/
PQendcopy();
```

Um Inhalte von Variablen in Klassen zu schreiben, müssen zunächst alle Datentypen in Zeichenketten umgewandelt werden. Anschließend ist es notwendig, alle Strings getrennt durch ein <TAB> zu **einem** String zusammenzufügen. Dieser String kann dann in eine Klasse geschrieben werden.

Beispiel:

```
char s[],t[],u[];
float a,b;      /* a = Radius Scheibe, b = Dicke Scheibe */
.
.
t=ftos (a);         /* Umwandlung float in string */
u=ftos (b);         /* dto. */
strcpy (s,t);       /* Wert von a in s kopieren */
strcat (s,“\t“);    /* Tabulator anhängen */
strcat (s,b);       /* Wert von b in s kopieren */
strcat (s,“\n“)     /* Zeilenende anhängen */
PQexec (“copy scheibe from stdin“);
```

[4] Betätigung der Tabulatortaste

```
PQputline (s);      /* s in Klasse scheibe schreiben */
PQputline (“.\n“);  /* “.“ und Zeilenende anhängen */
PQendcopy();
```

Beispielhaft wurde die Eingabe von zwei Werten in der Klasse **scheibe** implementiert. Um dies zu verwirklichen, mußten relativ viele Schritte unternommen werden. Es ist ersichtlich, daß dies z.B. bei der komplexen Klasse **bg** umfangreich und unübersichtlich wird.

5.5 Zusammenfassung

Postgres ist prinzipiell für den technischen Bereich gut geeignet. Ein Vorteil ist die große Anzahl von vordefinierten Typen, die hauptsächlich für den technischen Bereich ausgelegt sind. Außerdem können zusätzlich benötigte Typen und ihre zugehörigen Operationen definiert werden. Durch die Datenbanksprache Postquel können schnell Klassen erstellt und Daten manipuliert werden. Über LIBPQ können aus Anwendungsprogrammen auf die Datenbasen zugegriffen werden. Ein weiterer großer Vorteil ist die Versionsverwaltung, die über die „Zeitreise“ einfach und effizient erfolgen kann.

Nachteile von Postgres sind unter anderem die Benutzerfreundlichkeit. Der Monitor ist nicht komfortabel gestaltet. Befehle können beispielsweise nicht einfach wiederholt werden. So muß bei fehlerhafter Eingabe der gesamte Befehl neu eingegeben werden. Für Postgres existiert eine Benutzeroberfläche namens **PICASSO**. Diese ist aber derzeit nur für SUN-Rechner implementiert.

Die Schnittstelle zu C hat den Nachteil, daß nur Strings an die Datenbank übergeben werden können. Dazu müssen zuerst sämtliche Datentypen in Zeichenketten umgewandelt werden.

In der Version 4.01 sind noch nicht alle in den Handbüchern beschriebenen Features, wie z.B. die Datentypen `postquel` und `cproc` realisiert. Wenn jedoch alle Unzulänglichkeiten behoben sind, ist Postgres ein geeignetes DBMS für die technische Modellierung.

6 Eignungsanalyse von OBST für die technische Modellierung

Nach der in Kapitel 5 in Postgres durchgeführten Validierung anhand der allgemeinen und speziellen Anforderungen an DBMS, soll dies nun gleichermaßen für das zweite ausgewählte DBMS - OBST - erfolgen. Nach einer allgemeinen Vorstellung von OBST werden diese Anforderungen evaluiert.

6.1 Einführung

OBST (**OB**ject Management System of **ST**one) ist ein objektorientiertes Datenbanksystem, das speziell für das Projekt STONE (**ST**ructured **O**pe**N** **E**nvironment) am Forschungszentrum für Informatik (FZI) in Karlsruhe entwickelt wurde. Das Ziel von STONE ist eine offene Software-Engineering-Umgebung im Lehrbereich. Software-Engineering-Methoden sollen erlernt und trainiert werden. Bei den Objekten von STONE handelt es sich hauptsächlich um Dokumente. Zu Beginn des Projektes existierte kein OODBMS, das die Anforderungen von STONE erfüllte. Daher wurde OBST entwickelt, das auf die speziellen Bedürfnisse von STONE zugeschnitten wurde.

Die Hauptmerkmale von OBST sind:

- OBST ist nicht an eine bestimmte Programmiersprache gebunden. OBST soll von vielen Sprachen genutzt werden können. Dazu ist die Implementierung einer geeigneten Schnittstelle notwendig. Derzeit existiert lediglich eine Schnittstelle zu C++. Eine weitere zu LISP ist geplant.
- OBST bietet Mehrfachvererbung.
- Das Datenstrukturkonzept von STONE beruht auf dem eines Dokuments. Es existieren Datenmodelle, die zwischen großen und kleinen Objekten unterscheiden. Da der Umfang von Dokumenten erheblich variieren kann, spielt die Größe eines Objekts bei OBST keine Rolle,

OBST besitzt keine Anfragesprache, d.h. Daten können nur mit Hilfe einer Programmiersprache manipuliert und abgefragt werden. Es existiert lediglich eine Datendefinitionssprache, mit der Klassen in einem sogenannten *Schema* definiert werden.. Neben der Erstellung der Klassen müssen zusätzlich ihre Methoden in der entsprechenden Programmiersprache implementiert werden. Um auf die Klassen zugreifen zu können, muß das OBST-Schema vom OBST-Compiler in die entsprechende Programmiersprache übersetzt werden. Abbildung 6.1 zeigt die Entwicklung einer OBST-Applikation.

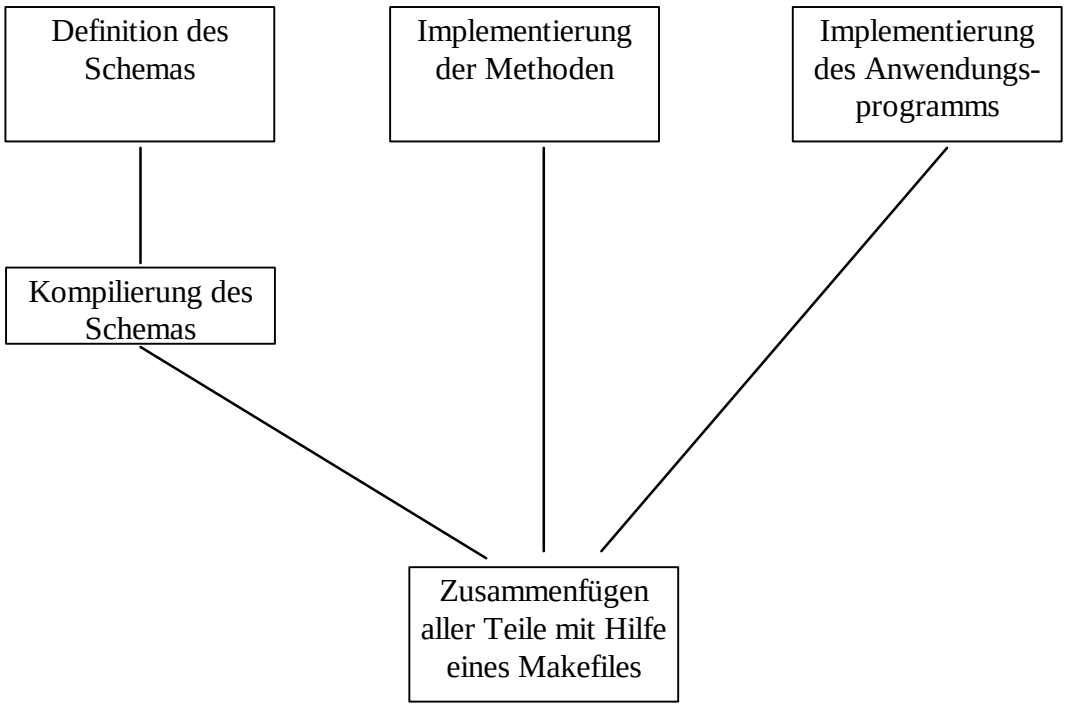


Abbildung 6.1: *Entwicklung einer OBST-Anwendung*

Da OBST unabhängig von einer Programmiersprache ist, kann prinzipiell ein Schema in mehrere Programmiersprachen übersetzt werden. Zur Zeit ist dies aber nur für C++ möglich.

Daten werden in sogenannten *Containern* gespeichert. Hierbei wird zwischen temporären und persistenten Containern unterschieden. Bei temporären Containern gehen die Daten nach Beendigung des Anwendungsprogramms verloren. Bei persistenten Containern bleiben die Daten solange erhalten, bis sie gelöscht werden.

Nach dieser kurzen Einführung in OBST sollen nun die allgemeinen Anforderungen an DBMS aus Kapitel 3, wie Datenintegration, Datenunabhängigkeit, Konsistenz der Daten und Recovery, untersucht werden.

6.2 Untersuchung von OBST anhand der allgemeinen Anforderungen

Das Ziel von STONE ist die Integration aller Tools, die zur Software-Engineering-Umgebung gehören. Da OBST speziell für STONE entwickelt wurde, ist deshalb die

Datenintegration realisiert, d.h. alle dazugehörigen Tools können auf dieselbe Datenbank zugreifen.
Die Synchronisation des Multi-User-Betriebs basiert auf dem Konzept der Container. Die Interprozeß-Synchronisation läßt zu, daß ein Container entweder von mehreren Prozessen zum Lesen oder von einem zum Schreiben geöffnet werden kann, d.h. Schreibzugriff ist exklusiv. Es besteht aber immer die Möglichkeit, den Inhalt der letzten Version des Containers zu lesen. Wird ein Container zum Schreiben geöffnet, so legt OBST zunächst eine Kopie des Containers an. In dieser Kopie werden dann die Änderungen vorgenommen. Mit der Operation "commit" wird der ursprüngliche Container durch die Kopie ersetzt. Mit der Operation "reset" werden alle Änderungen ignoriert und der Inhalt des Original-Containers bleibt erhalten. Somit ist die Konsistenz der Daten gewährleistet.

Das Transaktionsmanagement beruht auf dem ACID-Prinzip. Alle Manipulationen auf Daten zwischen einem "begin" - "end" - Block werden atomar und isoliert behandelt. Anderen Anwendungen dürfen auf die modifizierten Daten erst dann zugreifen, wenn die Transaktion abgeschlossen ist. Wird die Transaktion unterbrochen, so wird der ursprüngliche Zustand wieder hergestellt.
Dadurch, daß zunächst eine Kopie eines Containers bei Schreibzugriff erstellt wird, ist auch Recovery beim Auftritt eines Fehlers gewährleistet. Die ersten Bytes eines Containers sind für spezielle Ereignisse, wie z.B. Recovery, reserviert.

Durch das Konzept der Container ist auch Datenunabhängigkeit realisiert. Anwendungsprogramme können verändert werden, ohne daß dies Einfluß auf die Daten, d.h. Inhalte der Container, hat.

Um einen Verlust der Performance von Containern zu vermeiden, sollten nicht alle Daten in **einem** Container gespeichert werden. Wenn die Daten modifiziert werden, so wird ein weiterer Schreibzugriff auf den Container gesperrt. Dadurch werden andere Anwendungen blockiert und müssen warten bis die sperrende Anwendung den Container wieder frei gibt. Daher ist es sinnvoll, nur die zu einer Anwendung gehörenden Daten in einem Container zu speichern werden.

OBST erfüllt die allgemeinen Anforderungen an Datenbankmanagementsysteme. Im nächsten Schritt sollen nun die speziellen Anforderungen, die aus dem Szenario resultieren, untersucht werden.

6.3 Untersuchung von OBST anhand der speziellen Anforderungen

Die speziellen Anforderungen der technischen Modellierung aus Kapitel 4 sind in vier Gebiete gegliedert:

Anforderungen an DBMS bei:

- Erstellung der Klassen
- Erstellung und Manipulation der Daten
- Abfrage von Daten
- Sonstige Anforderungen an DBMS

In diesem Kapitel soll OBST nun daraufhin untersucht werden, inwieweit das OODBMS die gestellten Anforderungen erfüllt.

6.3.1 Erstellung der Klassen

Dieses Kapitel beinhaltet die Analyse von Vererbungs-, Datenstrukturmöglichkeiten und Darstellung von Beziehungen zwischen Objekten.

- **Vererbung**

  OBST bietet Mehrfachvererbung, d.h. eine Klasse kann von mehreren Klassen Attribute und Methoden erben und zusätzlich Komponenten und Methoden hinzufügen. Die Vererbung ist transitiv, d.h. eine Klasse erbt nicht nur die Merkmale seiner Superklasse, sondern auch die Merkmale deren Superklasse. Somit kann eine Klasse indirekt von einer anderen erben.
  Falls zwei Klassen ein Attribut oder eine Methode mit demselben Namen haben und eine andere Klasse von diesen beiden Klassen erbt, kommt es zu einem Namens-Konflikt. Um dieses Problem zu lösen, muß in der erbenden Klasse das entsprechende Attribut oder die Methode neu definiert werden.

- **Datenstrukturmöglichkeiten**

  Die Auswahl an vordefinierten Datentypen ist in OBST nicht sehr umfangreich. Im folgenden werden alle für das Szenario benötigten Typen auf Vorhandensein bzw. Möglichkeit ihrer Definition untersucht.

  - Es existiert kein vordefinierter Typ für reelle Zahlen. Dies ist für technische Anwendungen unerläßlich. Die Definition eines Typs “Real“ mit all seinen Operationen wie Addition, Subtraktion, Multiplikation, Division, Gleichheit, Ungleichheit, größer als, kleiner als etc. ist sehr umfangreich.

  - Für Integer-Zahlen existiert der Datentyp `sos_Int`. Damit kann eine 4-Byte lange Integer-Zahl in Form eines 2er-Komplements dargestellt werden.

  - Einzelne Zeichen können mittels des Typs `sos_Char` dargestellt werden. Der Typ `sos_Cstring` enthält einen Pointer, der auf eine Folge von

Zeichen des Typs `sos_Char` verweist, deren Ende durch das ASCII-Kontrollzeichen **NUL** angezeigt wird. Somit kann unbegrenzter Text dargestellt werden.

- Für logische Attribute existiert der Typ `sos_Bool`. Der Wertebereich von `sos_Bool` ist {FALSE, TRUE}.

- Für Datum und Zeit existiert kein vordefinierter Datentyp. Datum kann beispielsweise folgendermaßen definiert werden:

```
class Date
{
protected:
    sos_Int day;
    sos_Int month;
    sos_Int year;

public:
    sos_String get_date();
    void       set_date(sos_String);
}
```

Die Klasse **Date** besteht aus den Attributen *Tag*, *Monat* und *Jahr*, die jeweils vom Standarddatentyp `sos_Int` sind. Die Methode „Lesen“ eines Datums hat keine Eingabe, jedoch ist der Rückgabewert eine Zeichenkette vom Typ `sos_String`. Das Schreiben eines Datums hat als Eingabe “Datum“ vom Typ `sos_String` und keine Rückgabe. Allerdings fehlen hier die Einschränkungen des Tags von 1..31, des Monats von 1..12 ,des Jahres von beispielsweise 19??..20?? und die Berücksichtigung von Schaltjahren.

- Für die Koordinaten der Bohrungen existiert kein vordefinierter Datentyp. Er kann aber ebenso wie Datum mit der Schema-Sprache definiert werden, nachdem ein Typ für reelle Zahlen definiert wurde.

- In OBST ist die Definition von unbegrenzten Arrays nicht möglich. Bei der Erstellung eines Arrays muß dessen Größe immer angegeben werden.

Neben den Arrays gibt es weitere Aggregationen von vordefinierten oder selbst definierten Typen:

- **Set:** In einer Menge darf ein Element nur einmal vorkommen. Die Reihenfolge spielt dabei keine Rolle.
- **Bag:** `Bag` ähnelt der Menge, nur dürfen Elemente mehrfach vorkommen.
- **List:** Eine Liste ist beliebig lang und geordnet.
- **Mapping:** `Mapping` realisiert die Darstellung der Beziehung zwischen zwei Objekten. In `Mapping` stehen keine einzelnen Elemente, sondern immer Paare.

Mit dem Typ `sos_Cursor` können die Aggregationen vorwärts und rückwärts durchlaufen werden.

- Die Attribute eines Objekts können nicht nur Werte, sondern auch Verweise auf Objekte derselben oder auch anderer Klassen enthalten [UTSRZA94 S.4 ff]. In OBST werden nur bei einfachen Standarddatentypen, wie Integer, Char etc. Werte gespeichert. Ansonsten enthalten die Komponenten Referenzen.

- Die Realisierung eines Rasterbildes, durch Aufruf eines externen Programms, kann über die Schnittstelle zu C++ erfolgen.

- OBST bietet die Möglichkeit, eigene Typen zu definieren. Mit der Schema-Sprache können aus den vordefinierten Typen neue Typen entwickelt werden. Außerdem können externe Typen mit einer Programmiersprache definiert werden. Die Größe des externen Typs ist auf 4 bis 8 Bytes begrenzt. Beispiele für solch externe Typen sind `sos_Int`, `sos_Char`, `sos_Bool` etc.

Die OBST-Bibliothek beinhaltet einige wichtige Schemata. Das Kernel-Schema (`knl`) beschreibt die Standarddatentypen `sos_Bool`, `sos_Char`, `sos_Cstring`, `sos_Int`, `sos_Pointer`. Das Aggregations-Schema (`agg`) enthält die Zusammenfassung einiger Standarddatentypen wie `Array`, `Bag`, `List`, `Set`, `Mapping`. Das Directory-Schema (`dir`) ist für die hierarchische Strukturierung aller Instanzen zuständig. Die Bedeutsamkeit dieses Schemas resultiert aus der Ermöglichung des Zugriffs auf alle Instanzen, die in der Datenbank gespeichert sind. Das Meta-Schema (`mta`) wird von allen Methoden benötigt, um auf Objekte zugreifen zu können.

Es ist ersichtlich, daß OBST nicht allzu viele für die technische Modellierung nötige Standarddatentypen zur Verfügung stellt. Mit der Schema-Sprache können

aus den Standarddatentypen eigene Typen definiert werden. Um komplexe Datentypen für den technischen Bereich definieren zu können, muß zuallererst der Typ “Real“ mit all seinen Operationen festgelegt werden. Dies ist aufwendig und daher als erheblicher Nachteil anzusehen.

- **Darstellung von Beziehungen von Objekten**

  Wie in Kapitel 6.3 schon erwähnt, existiert der Aggregationstyp `Mapping`, mit dem Beziehungen zwischen genau zwei Objekten dargestellt werden können.
  Eine weitere Möglichkeit besteht darin, Referenzen auf andere Objekte direkt im Attribut zu speichern. OBST speichert nur Werte von einfachen Datentypen, ansonsten enthalten die Attribute Referenzen.

6.3.2 Erstellung und Manipulation von Daten

In diesem Kapitel werden die Möglichkeiten für Default-Belegungen, Überprüfung sinnvoller Werte und Konsistenzprüfungen ermittelt.

- **Default-Belegungen**

  Default-Belegungen, wie beispielsweise die Länge der Achse, können in den Methoden definiert werden.

- **Festlegung und Überprüfung von sinnvollen Werten**

  Die Festlegung von sinnvollen Werten kann in den Methoden erfolgen. Die Überprüfung dieser Werte bei Änderungen wird entweder in den Methoden oder im Anwendungsprogramm realisiert.

- **Konsistenzprüfungen**

  Die Konsistenzprüfungen müssen in Applikationen erfolgen. Die Werte werden in Variablen eingelesen und solange geprüft, bis alle Restriktionen erfüllt sind. Danach können die Werte in die entsprechenden Klassen geschrieben werden.

6.3.3 Abfrage von Daten

Zur eindeutigen Identifizierung von Objekten wird eine OID gefordert. Neben einer Abfragesprache sollte der Zugriff auf die Daten aus einer Programmiersprache heraus durchführbar sein. Zusätzlich sind die Möglichkeiten zur Benutzerinteraktion zu untersuchen.

- **Eindeutige OID**

  Jedes Objekt in OBST hat eine eindeutige Objektidentität. Zwei Objekte sind identisch, wenn sie auf dasselbe Objekt verweisen. Dagegen sind zwei Objekte gleich, wenn ihre Attribute dieselben Werte haben.

- **Abfragesprache**

  OBST stellt keine Abfragesprache zur Verfügung. Die Schema-Sprache ist lediglich eine Datendefinitionssprache. Auf Objekte kann nur aus einer Anwendung heraus zugegriffen werden. Zugriff auf OBST-Objekte wird nur durch eine Referenz eines anderen Objekts ermöglicht. Direkter Zugriff existiert nur für das sogenannte *root object*. Alle Objekte sind in der Klasse **Directory** hierarchisch strukturiert. Über das *root object* kann auf ein beliebiges Objekt mit Hilfe seines Schlüssels zugegriffen werden, den jedes Objekt beim Einbinden in die Hierarchie erhält.
  OBST wurde nicht für Endbenutzer, sondern für bereichsspezifische Umgebungen entwickelt. Die Gewichtung liegt daher bei Datenbankmanagement und Integration und nicht bei interaktiven Abfragen oder automatischer Listenerstellung.

- **Zugriff aus einer Programmiersprache heraus**

  Über eine Schnittstelle kann derzeit von C++ auf OBST zugegriffen werden. Das Schema wird in C++ übersetzt und mit den definierten Methoden und dem Anwendungsprogramm zusammengelinkt.

- **Benutzerinteraktion**

  Derzeit existieren keine Tools zur Erstellung von Eingabemasken oder Ähnlichem. Jedoch ist eine Benutzeroberfläche zur Erstellung von Schemata in Entwicklung. Der Name der Benutzeroberfläche ist **USE** und basiert auf Tcl und tclOBST (Schnittstelle zwischen Tcl und OBST).

6.3.4 Sonstige Anforderungen an DBMS

Bei den sonstigen Anforderungen wird die Realisierung der technisch-funktionalen Abhängigkeiten, sowie der Versions- und Variantenverwaltung geprüft.

- **Realisierung der technisch-funktionalen Abhängigkeiten**

  Die technisch-funktionalen Abhängigkeiten müssen im Anwendungsprogramm realisiert werden. Beispielsweise muß beim Aufruf der Methode zur Änderung der Dicke der Scheibe auch die Methode zur Anpassung der Tiefe der mittigen Bohrung mit dem entsprechenden Wert aufgerufen werden.

- **Versions- und Variantenbildung**

  OBST stelle keine direkte Versions- und Variantenverwaltung zur Verfügung. Es besteht aber die Möglichkeit, eine Kopie eines Objekts mittels des Befehls:

  ```
  static C clone (C source, sos_Container ct);
  ```

  zu erstellen [UTSRZA94 S.30]. Dieser Kopie wird dann eine neue Versions- bzw. Variantennummer vergeben und die Attribute können verändert werden.

Eigentlich erfüllt OBST fast alle Anforderungen, die aus Sicht der technischen Modellierung notwendig sind. Ein entscheidender Nachteil besteht allerdings darin, daß kein vordefinierter Typ für reelle Zahlen mit den entsprechenden Operationen existiert. Gerade im technischen Bereich sind reelle Zahlen unerläßlich. Das Fehlen des Typs "Real" resultiert aus den Anforderungen des Projekts STONE, die hauptsächlich aus dem Bereich **CASE** (**C**omputer **A**ided **S**oftware **E**ngineering) gestellt wurden.

Im Verlauf dieser Studienarbeit wurde eine neue Version von OBST veröffentlicht. In dieser aktuellen Version 3-4.3 wurden einige der erwähnten Unzulänglichkeiten behoben. Die Neuerungen sollen nun im nächsten Unterkapitel erläutert werden.

6.4 Neuerungen der Version 3-4.3

Objektorientierte Datenbanken unterliegen einer ständigen Weiterentwicklung. Anfangs werden sie für einen speziellen Zweck entwickelt. Im Laufe der Zeit stellen sich dann die Unzulänglichkeiten des Systems heraus. Dies gilt auch für den Verlauf der Entwicklung von OBST. Von der speziellen Datenbank für das Projekt STONE, wird OBST Schritt für Schritt zu einem DBMS weiterentwickelt, das in vielen Bereichen genutzt werden kann.

Eine wohl entscheidende Innovation ist die Einführung des Datentyps `sos_float`. Mit diesem Typ können reelle Zahlen dargestellt werden. Die Länge der Zahlen ist auf vier Bytes beschränkt. Falls größere Genauigkeit erforderlich ist, kann der Typ `sos_double` in Anspruch genommen werden, dessen Länge auf acht Bytes limitiert ist. Durch diese beiden Typen können nun mit relativ wenig Aufwand komplexe Datentypen definiert werden. Dies bedeutet auch einen wesentlichen Schritt in Richtung Eignung für die technische Modellierung.

6.5 Zusammenfassung

OBST bietet nur wenige Standarddatentypen an. Durch das Fehlen des Typs für reelle Zahlen ist die Version 3-4 nur eingeschränkt für die technische Modellierung geeignet, da im technischen Bereich reelle Zahlen die Grundlage für viele geforderte Datentypen sind. Infolge der Einführung der Datentypen `sos_float` und `sos_double` in der aktuellen Version 3-4.3 wird nun die Definition von komplexen Typen erheblich erleichtert. Ein Vorteil ist der objektorientierte Ansatz, so daß Objekte direkt über ihre zugehörigen Methoden manipuliert werden. Im Gegensatz dazu können in RDBMS Objekte nur über allgemeingültige, d.h. für alle Klassen gleiche Befehle angesprochen werden.

Der Nachteil von OBST liegt im Fehlen einer Datenmanipulationssprache. Somit können keine Eingaben, Änderungen, Löschungen und Abfragen interaktiv durchgeführt werden. Dies muß über ein Anwendungsprogramm erfolgen. Der Grund dafür liegt in der Konzipierung von OBST. Das OODBMS wurde nicht für Endbenutzer entwickelt, sondern für bereichsspezifische Umgebungen.
Ein weiterer Nachteil ist der Mangel an Tools für die Erstellung von Applikationen. Um dies zu beheben, ist derzeit eine graphische Benutzeroberfläche namens **USE** zur Erstellung von OBST-Schemata in Entwicklung. Die Implementierung von USE erfolgt mittels Tcl und tclOBST, der Schnittstelle zwischen Tcl und OBST.

Im folgenden Kapitel werden die beiden ausgewählten DBMS in einer Gegenüberstellung auf ihre Eignung für die technische Modellierung bewertet.

7 Fazit

In Kapitel 5 und 6 wurden die Anforderungen, die an DBMS aus der technischen Modellierung resultieren anhand von Postgres bzw. OBST evaluiert. In diesem Kapitel sollen nun beide in einer Gegenüberstellung auf Eignung für die technische Modellierung bewertet werden.
Zunächst erfolgt ein tabellarischer Überblick (siehe Abbildung 7.1), inwieweit die beiden DBMS die gestellten Anforderungen erfüllen.

Anforderungen	**Postgres Version 4.01**	**OBST Version 3-4**
ACID-Prinzip	ja	ja
Recovery	ja	ja
Vererbung	Einfachvererbung	Mehrfachvererbung
Reelle Zahlen	`float4, float8`	nein [5]
Integer Zahlen	`int2, int4`	`sos_Int`
Text (unbegrenzt)	`text`	`sos_Cstring`
logisch	`bool`	`sos_Bool`
Koordinaten	`point`	nein muß selbst definiert werden
Datum	`abstime`	nein muß selbst definiert werden
Array (unbegrenzt)	ja Bsp.: `float8[]`	Array ja, unbegrenzt nein Bei der Erstellung eines Arrays muß die Größe angegeben werden.
Referenz auf andere Objekte	nein	ja
Rasterbild	C-Prozedur mit Hilfe des Typs `cproc (`in Version 4.01 noch nicht verfügbar)	Aufruf eines externen Programms über Schnittstelle zu C++
Def. eigener Typen	ja	ja

[5] Version 3-4.3: `sos_float, sos_double`

Darstellung von Beziehung	• Typ `oid` und geeignete Abfrage • Typ `postquel` (in Version 4.01 noch nicht verfügbar)	`Mapping` oder Referenz auf Objekte
Default-Werte	möglich über Anwendungsprogramm	ja Definition in den Methoden
Überprüfung sinnvoller Werte	in Applikation	in den Methoden
Konsistenzprüfungen	in Applikation	in Applikation
eindeutige Objektidentität	ja	ja
Datendefinitionssprache	**Postquel**	**OBST**
Datenmanipulationssprache	**Postquel**	nein
Schnittstelle zu Programmiersprache	**LIBPQ**: Schnittstelle zu C **CLOS**: Schnittstelle zu LISP	Schnittstelle zu C++ Schnittstelle zu LISP geplant
Benutzerinteraktion: Erstellung von Masken, etc.	**PICASSO**: Benutzeroberfläche für Postgres (derzeit nur für SUN-Rechner)	**USE**: Benutzeroberfläche zur Erstellung von OBST-Schemata (in Entwicklung)
Realisierung von technisch-funktionalen Abhängigkeiten	• rules • in Applikation (z.B. im Szenario)	in Applikation
Versions- und Variantenverwaltung	über "Zeitreise"	über Kopie eines Objekts

Abbildung 7.1: *Gegenüberstellung von Postgres und OBST*

Vorteile von Postgres

Bei der Bereitstellung vordefinierter Datentypen bietet Postgres weitaus mehr als OBST. Insbesondere existieren Datentypen, die im technischen Bereich benötigt werden, wie beispielsweise reelle Zahlen (`float4, float8`), Koordinaten (`point`), Rechtecke (`box`) und Kreise (`circle`). Ein weiterer Vorteil ist die Existenz des vordefinierten Typs `abstime`, mit dem Datum und Zeit verwaltet werden können. Zusätzlich zu den vordefinierten Datentypen können aus diesen eigene, komplexe Typen definiert werden.
Ebenso ist die Möglichkeit, unbegrenzte Arrays jeglichen Typs zu erstellen vorteilhaft, da die Anzahl der Elemente eines Objekts variabel sein kann, wie beispielsweise die variable Anzahl der Bohrungen auf dem Bohrkranz im vorgestellten Szenario.
Die Vergabe einer eindeutigen Objektidentität ist ebenfalls gewährleistet, so daß jedes Objekt in der Datenbank global eindeutig vorliegt.

Eine weitere Stärke von Postgres ist die Datendefinitions- und Datenmanipulationssprache **POSTQUEL**. POSTQUEL erlaubt die interaktive Eingabe, Änderung, Löschung und Abfrage von Daten. Außerdem existieren Schnittstellen zu Programmiersprachen. Derzeit bestehen Schnittstellen zu C (LIBPQ) und zu LISP (CLOS).
Die Realisierung der technisch-funktionalen Abhängigkeiten kann prinzipiell über geeignete Regeln erfolgen. Regeln beschreiben Aktivitäten, die zusätzlich oder anstatt eines eintretenden Ereignisses ausgeführt werden. Da Regeln in der Version 4.01 nur tupelbezogen angewandt werden können, muß die Darstellung der Abhängigkeiten mitunter, wie im Szenario, in den Applikationen erfolgen.
Eine wohl entscheidende Stärke von Postgres ist die Versionsverwaltung über die "Zeitreise". Bei Änderung von Daten müssen nur neue Versions- bzw. Variantennummern vergeben werden, da Postgres den Inhalt einer Datenbasis nicht überschreibt, sondern die ehemaligen Werte speichert. Dadurch kann der Inhalt der Datenbasis, und somit alle Versionen und Varianten der Instanzen, von der Erstellung bis zur Löschung mit einer Zeitangabe abgefragt werden.

Nachteile von Postgres

In Postgres ist lediglich Einfachvererbung möglich. Dies ist für diverse technische Anwendungen unzureichend. Eine Referenzierung von Objekten ist nicht möglich, da Postgres keine Zeiger zur Verfügung stellt. Dies wäre ein geeignetes Mittel, um die einzelnen Objekte (Scheibe, mittlere Bohrung, Bohrkranz und Achse) von der Baugruppe aus zu referenzieren. Die Implementierung von Zeigern ist jedoch geplant [SJGP94]. Ebenso ist die Vorgabe von Default-Werten in Postgres nicht möglich. Dies kann nur über ein Anwendungsprogramm realisiert werden.
Die Schnittstelle zu C hat den Nachteil, daß nur Strings an die Datenbank übergeben werden können. Dazu müssen zuerst sämtliche Datentypen in Zeichenketten umgewandelt werden, was unkomfortabel und aufwendig ist.
Ein weiterer Mangel von Postgres ist die Benutzerfreundlichkeit. Der Postgres-Monitor, in dem Postgres-Befehle interaktiv ausgeführt werden können, erlaubt nicht die Wiederholung einer Eingabe. Wenn beispielsweise bei der Eingabe eines Befehl ein Fehler unterläuft, so muß der gesamte Befehl nochmals eingegeben werden. Zudem sollte die Möglichkeit bestehen, Eingabemasken für den Benutzer zu erstellen. Es existiert zwar eine Benutzeroberfläche für Postgres (PICASSO), welche aber lediglich auf SUN-Rechnern zur Verfügung steht.
Ferner besteht ein Nachteil darin, daß noch nicht alle in den Handbüchern beschriebenen Features realisiert sind. Beispielsweise standen die Typen `postquel` und `cproc` in der Version 4.01 nicht zur Verfügung. Die Erstellung von Regeln unterliegt Einschränkungen, da lediglich auf bestimmte Tupeln bezogene Regeln möglich sind. Außerdem können bidirektionale Beziehungen nicht durch rules beschrieben werden. Eine solche bidirektionale Abhängigkeit liegt im Szenario bei dem Radius der mittigen Bohrung und dem Radius der Achse vor. Bei Änderung eines Radius soll jeweils der andere Radius angepaßt werden können. Dies würde bei Postgres mittels zweier Regeln zu einer Endlosschleife führen.

Vorteile von OBST

Ein Vorzug von OBST ist die Möglichkeit der Mehrfachvererbung im Gegensatz zur Einfachvererbung bei Postgres. Ebenso ist die Referenz auf andere Objekte vorteilhaft. Bei fast allen Datentypen (außer einfachen Typen wie `sos_Int` oder `sos_Char`) werden die entsprechenden Werte nicht gespeichert, sondern referenziert. Daher existiert auch die Möglichkeit, andere Objekte zu referenzieren.
Die Erzeugung eines Rasterbildes der Baugruppe kann direkt über die Schnittstelle zu C++ mit Hilfe einer geeigneten Prozedur verwirklicht werden.
Vorteilhaft ist die Möglichkeit, in OBST eigene Typen definieren zu können. Mit der OBST-Schema-Sprache können aus den vordefinierten Typen neue Typen entwickelt werden. Zusätzlich können externe Typen mit einer Programmiersprache definiert werden.
Die Darstellung von Beziehungen zwischen Objekten kann durch den aggregierten Typ `Mapping`, dessen Werte immer Paare von Objekten umfassen, oder über Referenzierung erfolgen.
Ein weiterer Vorteil besteht darin, Default-Werte in den Methoden definieren zu können.

Nachteile von OBST

Ein entscheidender Nachteil ist das Fehlen eines vordefinierten Typs für reelle Zahlen, denn bei technischen Anwendungen sind reelle Zahlen meist die Basis für viele Datentypen. Die Definition eines Typs “Real“ mit allen dazugehörigen Operationen wäre sehr umfangreich. In der aktuellen Version 3-4.3 wurden die Typen `sos_float` und `sos_double` hinzugefügt.
Ebenso nachteilig ist die geringe Anzahl von vordefinierten Datentypen. Zwar ist in OBST die Definition eines Typs “Datum“ möglich, jedoch ist die Implementierung der Beschränkungen der Werte für Tag (1..31), Monat (1..12), Jahr (19??..20??)und Berücksichtigung der Schaltjahre umfangreich.
Unbegrenzte Arrays sind in OBST nicht möglich. Bei der Erstellung von Arrays muß immer die Größe angegeben werden. Dieser Nachteil wiegt allerdings nicht so schwer wie das Fehlen einer Abfragesprache. Mit der Definitionssprache OBST können zwar Schemata erstellt werden, jedoch kann die Manipulation und Abfrage von Daten nur über eine Applikation erfolgen, d.h. interaktive Abfragen sind nicht möglich. Dies und eine fehlende Benutzeroberfläche erschweren die Anwendung von OBST. Eine Benutzeroberfläche zur Erstellung von OBST-Schemata ist allerdings in Entwicklung.
Eine weitere Schwachstelle ist die nicht vorhandene Versionsverwaltung. Hierfür muß im Anwendungsprogramm zuerst eine Kopie eines Objekts angelegt werden, die daraufhin verändert werden kann. Außerdem kann die Realisierung technisch-funktionaler Abhängigkeiten nur in Applikationen erfolgen. Dazu muß beim Aufruf der Methode zur Änderung eines Parameters auch die Methode zur Änderung des abhängigen Parameters aufgerufen werden.

Die folgende Bewertung von Postgres und OBST bezieht sich auf die Eignung für die technische Modellierung ausschließlich auf der Grundlage der in dieser Arbeit

entwickelten Anforderungen. Sie soll aufzeigen, wie gut bzw. schwach die DBMS die einzelnen Anforderungen erfüllen.

Anforderungen	**Bewertung Postgres 4.01**	**Bewertung OBST 3-4**
ACID-Prinzip	+	+
Recovery	+	+
Vererbung	o	++
Reelle Zahlen	++	-- [6]
Integer Zahlen	++	++
Text (unbegrenzt)	++	++
logisch	++	++
Koordinaten	++	-
Datum	++	-
Array (unbegrenzt)	++	o
Referenz auf andere Objekte	--	++
Rasterbild	- [7]	+
Def. eigener Typen	++	++
Darstellung von Beziehungen	o	++
Default-Werte	-	++
Überprüfung sinnvoller Werte	o	o
Konsistenzprüfungen	o	o
eindeutige Objektidentität	++	++
Datendefinitionssprache	++	+
Datenmanipulationssprache	++	--

[6] Bewertung ++ in der Version 3-4.3
[7] Bewertung + nach der Realisierung des Typs `cproc`

Schnittstelle zu Programmiersprache	o	+
Benutzerinteraktion: Erstellung von Masken, etc.	-	-
Realisierung von technisch-funktionalen Abhängigkeiten	o[8]	-
Versions- und Variantenverwaltung	++	-

++ sehr gut + gut o weniger gut - schwach -- sehr schwach

Abbildung 7.2: *Bewertung von Postgres und OBST*

In Abbildung 7.2 wird die Bewertung der beiden DBMS bezüglich ihrer Erfüllung der Anforderungen für die technische Modellierung aufgezeigt. Eine genaue Analyse beider DBMS erfordert eine Gewichtung der einzelnen Anforderungen. Diese Gewichtung hängt jedoch stark von der jeweiligen Anwendung ab. In der technischen Modellierung liegt beispielsweise eine hohe Gewichtung auf dem Vorhandensein eines Typs für reelle Zahlen. Dagegen ist die Existenz eines Typs "Datum" oder "Koordinaten" nicht von erheblicher Wichtigkeit, wenn diese Typen definiert werden können.

Die Version 3-4 von OBST ist durch das Fehlen des Typs `float` nur bedingt für die technische Modellierung geeignet. Dieser Mangel wurde in der aktuellen Version behoben. Das Hinzufügen der beiden Typen `sos_float` und `sos_double` ist ein entscheidender Schritt in Richtung Eignung für die technische Modellierung. OBST ist jedoch für interaktive Nutzung ungeeignet. Um OBST anwenden zu können, ist Programmiererfahrung erforderlich. Außerdem mangelt es an geeigneter Benutzerführung durch die Handbücher.

Postgres ist prinzipiell gut für die technische Modellierung geeignet. Durch die Abfragesprache können auch Benutzer mit geringen Programmierkenntnissen Postgres nutzen. Jedoch fehlen in der Version 4.01 einige wichtige Features, wie beispielsweise die Datentypen `postquel` und `cproc`. Außerdem ist die Erstellung von Regeln noch nicht vollständig realisiert. Ein wichtiger Aspekt ist auch die Möglichkeit der Referenzierung von Objekten, die aber von den Entwicklern geplant ist.

Beide DBMS sind noch nicht vollständig ausgereift, jedoch gehen die Entwicklungen in die richtige Richtung. Postgres 4.01 kann mit Einschränkungen in der technischen Modellierung eingesetzt werden. Werden alle in den Handbüchern beschriebenen und geplanten Features realisiert, ist es für die technische Modellierung gut geeignet.
Die Version 3-4 von OBST ist für technische Anwendungen noch nicht ausgereift. Dagegen ist die aktuelle Version 3-4.3 durch Hinzunahme der Typen `sos_float` und `sos_double` der Eignung für die technische Modellierung einen großen Schritt näher gekommen.

[8] Bewertung + nach Erweiterung der rules

Wünschenswert ist eine Weiterentwicklung beider DBMS durch Implementierung von Tools, die die Anwendung von Postgres bzw. OBST benutzerfreundlicher gestalten. Insbesondere besteht bei OBST die Notwendigkeit der Realisierung einer geeigneten Abfragesprache.
An dieser Stelle möchte ich mich bei meiner Betreuerin Dipl.-Inform. Monika Bihler für ihre sehr engagierte Betreuung und Unterstützung während meiner Studienarbeit bedanken.

Des weiteren danke ich Dipl.-Inform. Markus Stolpmann für seine Anregungen und ergänzenden Kommentare.

Glossar

ACID-Prinzip

Atomicity - **C**onsistency - **I**solation - **D**urability. Eine Transaktion wird entweder ganz oder gar nicht ausgeführt.

ADT

Abstrakter **D**aten**T**yp wird formal beschrieben, jedoch ist die Implementierung nach außen hin nicht sichtbar, da dies für den Anwender nicht relevant ist.

Attribut

Attribute definieren die Spalten der Tabelle im relationalen Datenmodell.

CSG

Constructive **S**olid **G**eometry ist ein körperorientiertes Modellierungsverfahren.

Datenbasis

Teil der Datenbank, in der die Daten stehen.

Datenintegration

Daten befinden sich in einer zentralen Datenbank, auf die alle Anwendungsprogramme zugreifen können.

DM

In **D**aten**M**odellen werden die Eigenschaften von Objekten und deren Beziehungen untereinander beschrieben.

DBMS

Das **D**aten**B**ank**M**anagement**S**ystem ist für das Auffinden und Speichern von Neueinträgen, Änderungen und Löschungen von Daten zuständig.

DDL

Die **D**ata **D**efinition **L**anguage benötigt man zur Definition der Attribute einer Relation.

DML

Die **D**ata **M**anipulation **L**anguage ist dazu da, Neueinträge, Löschungen und Veränderungen von Daten und Anfragen auf diese zu erledigen.

Domäne

Wertebereich eines Attributtyps.

ER-Modell

Im **E**ntity-**R**elationship-Modell werden Objekte mit ihren Eigenschaften und den Beziehungen zu anderen Objekten unabhängig von einer Anwendung beschrieben.

Fremdschlüssel

Attribut einer Relation, das in einer anderen Relation Primärschlüssel ist.

Grad

Die Anzahl der Attribute wird Grad einer Relation genannt.

Instanz

Instanzvariablen nehmen konkrete Werte an. (Vergleichbar mit Tupeln bei RDBMS).

Integrität

Jeder Anwender erhält nur die für ihn relevanten Daten.

Klasse

Zusammenfassung von Objekten mit ähnlichen Eigenschaften.

Konsistenz

Korrektheit und Vollständigkeit der Daten in der Datenbank.

Künstlicher Schlüssel

Definition eines zusätzlichen Attributs, meist fortlaufende Nummer, das die Identifizierung der Tupeln erleichtert, wenn der Primärschlüssel zu kompliziert wird.

Mehrfachvererbung

Strukturen und Methoden mehrerer Klassen können an eine Subklasse vererbt werden.

Objektintegrität

Ein Objekt kommt in einer Relation nur einmal vor und ist eindeutig identifizierbar.

OID

Objektidentität wird bei der Erstellung eines Objekts vergeben und ist global eindeutig in der Datenbank. Sie bleibt bis zur Löschung des Objekts erhalten und wird nie an ein anderes Objekt vergeben.

OODBMS

Objekt**O**rientiertes **D**aten**B**ank**M**anagement**S**ystem

OODBS

Objekt**O**rientiertes **D**aten**B**ank**S**ystem

Primärschlüssel

Attribut oder Attributkombination zur eindeutigen Identifizierung eines Tupels bei relationalen Datenbanksystemen.

RDBMS

Relationales **D**aten**B**ank**M**anagement**S**ystem

RDBS

Relationales **D**aten**B**ank**S**ystem

Recovery

Herstellung des ursprünglichen Zustandes einer Datenbank bei Eintritt eines Fehlers (z.B. Systemabsturz).

Redundanz

Vorkommen derselben Daten in einer oder verschiedenen Datenbanken.

Referentielle Integrität

Bei Änderung eines Primärschlüssels müssen alle Relationen aktualisiert werden, in denen dieser als Fremdschlüssel vorkommt.

Schema

Schema beschreibt die Gesamtheit aller Attribute einer Relation.

Surrogat

Um den Klassennamen nicht immer ausschreiben zu müssen, gibt es in Postgres die Möglichkeit, sogenannte Surrogate zu benutzen.
Beispiel:

```
retrieve (s.all) from s in scheibe where s.radius=5.4
```

Technisches Modell

Abbildung eines Realweltausschnittes in einem Modell. Im Top-Down-Entwurf werden die Objekte von der Baugruppe bis hin zum Einzelteil beschrieben.

Technische Objekte

Baugruppen und Einzelteile einer Maschine oder Anlage, die in einer hierarchischen Struktur angeordnet sind.

Technische Operationen

Die technischen Operationen dienen hauptsächlich zum Erzeugen, Verändern, Löschen und Positionieren von technischen Objekten.

Technisch-funktionale Abhängigkeiten

Beschreibung der Beziehungen zwischen technischen Objekten oder der Beziehungen der Objekte zu technischen Operationen.

Tupel

Zeileneintrag der Tabelle im relationalen Datenmodell. Attribute nehmen konkrete Werte an.

Vererbung

Strukturen und Methoden einer Klasse werden an Subklassen vererbt, d.h. sie müssen nicht nochmals definiert werden.

Literaturverzeichnis

[ACSST93] J. Alt, E. Casais, B. Schiefer, S. Sirdeshpande, D. Theobald: *The OBST Tutorial*, ftp://ftp.fzi.de/pub/OBST/OBST3-4/psfiles, 1993

[Anc94] M. C. Ancutici: *Datenverschlüsselung und Versiegelung von Dokumenten,* Diplomarbeit Nr. 1166, Universität Stuttgart, 1994

[CRSTZ92] E. Casais, M. Ranft, B. Schiefer, D. Theobald, W. Zimmer: *OBST - an Overview,* ftp://ftp.fzi.de/pub/OBST/OBST3-4/psfiles, 1992

[Dud93] *Duden Informatik*, Dudenverlag, Mannheim, Leipzig, Wien, Zürich, 1993

[Heu92] A. Heuer: *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme,* Addison-Wesley, Bonn, München, Paris [u.a.], 1992

[Hil94] V. Hillmann: *Objektorientierte Programmierung in C++: das ganz andere C++-Buch*, Markt und Technik, Buch- und Software-Verlag GmbH, Haar bei München, 1994

[HPS89] C. Hübel, R. Paul, B. Sutter: *Technische Modellierung und DB-gestützte Datenhaltung - ein Ansatz für ein durchgängiges, integriertes Produktmodell*, Report, Universität Kaiserslautern, 1989

[HS93] C. Hübel, B. Sutter: *Datenbankintegration von Ingenieuranwendungen Modelle, Werkzeuge, Kontrolle*, Vieweg, Braunschweig, Wiesbaden, 1993

[Kra94] U. Kracke: *Datenbank-Management*, Fachverlag für EDV, Augsburg, 1994.

[Pos90] POSTGRES Group: *Reference Manual 4.0*, ftp://s2k-ftp.Berkeley.edu/pub/postgres, 1990

[PS91] R. Paul, B. Sutter: *Technisches Modellieren - Ein Zugang zur integrierten Produktdatenverwaltung,* in H.-J. Appelrath (Herausgeber), *Datenbanksysteme in Büro, Technik und Wissenschaft*, Seiten 288-307, Springer-Verlag, Kaiserslautern, 1991

[PST94] C. Popp, B. Schiefer, D. Theobald: - *USE - The OBST Support Environment for Schema Evolution,* ftp://ftp.fzi.de/OBST/OBST3-4/psfiles, 1994

[Reu92] A. Reuter: *Vorlesung Informationssysteme / Datenbanken WS 92/93*, Universität Stuttgart, 1992

[RKP94] J. Rhein, G. Kemnitz and the POSTGRES Group: *User Manual*, ftp://s2k-ftp.cs.Berkeley.edu/pub/postgres, 1994

[RS94] L.A. Rowe, M. Stonebreaker: *The POSTGRES Data Model*, ftp://s2k-ftp.cs.Berkeley.edu/pub/postgres/papers, 1994

[SJGP94] M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos: *On rules, procedures, caching and views in data base systems*, ftp://s2k-ftp.cs.Berkeley.edu/pub/postgres/papers, 1994

[SR94] M. Stonebreaker, L.A. Rowe: *The Design of Postgres*, ftp://s2k-ftp.cs.Berkeley.edu/pub/postgres/papers, 1994

[SRH94] M. Stonebreaker, L.A. Rowe, M. Hirohama: *The Implementation of Postgres,* ftp://s2k-ftp.cs.Berkley.edu/pub/postgres/papers, 1994

[STUCF94] B. Schiefer, D. Theobald, J. Uhl, E. Casais, A. Freyberg: *User´s Guide - OBST Release 3-4* , ftp://ftp.fzi.de/pub/OBST/OBST3-4/psfiles, 1994

[Unl95] R. Unland: *Objektorientierte Datenbanken: Konzepte und Modelle*, International Thomson Publishing, Bonn, 1995

[UTSRZA94] J. Uhl, D. Theobald, B. Schiefer, M. Ranft, W. Zimmer, J. Alt: *The Object Management System of STONE - OBST Release 3-4 - ,* ftp://ftp.fzi.de/pub/OBST/OBST3-4/psfiles, 1994

[Zah93a] E. Zahn: *Vorlesung Operative Planung SS 93*, Universität Stuttgart, 1993

[Zah93b] E. Zahn: *Vorlesung Planungs- und Informationstechnologie SS93*, Universität Stuttgart, 1993

[ZG92] E. Zahn, J. Greschner: *Strategischer Erfolgsfaktor Information*, in H. Krallmann, J. Papke, B. Rieger (Hrsg.): *Rechnergestützte Werkzeuge für das Management,* Berlin, 1992