Grundlegende Konzepte des Datenbankmodells von PCTE

Udo Kelter

23.4.2002

Zusammenfassung dieses Lehrmoduls

Dieses Lehrmodul stellt die wichtigsten Merkmale des Datenbankmodells von PCTE vor. Nach Durcharbeiten dieses Moduls soll man in der Lage sein, mit dem Standardbrowser in einer PCTE-Objektbank zu navigieren und im Browser einfache Datenmanipulationen auszuführen.

Beschrieben werden die Konzepte Objekt, Attribut, Link, Typname und Schema Definition Set. Das Sichtenkonzept von PCTE wird ausführlich vorgestellt und mit dem relationalen verglichen.

Vorausgesetzte Lehrmodule:

empfohlen:
- Datenmodellierung mit ER-Modellen
- Einordnung und Historie von PCTE

Stoffumfang in Vorlesungsdoppelstunden: 1.6

Inhaltsverzeichnis

1 Einordnung von PCTE

Dieses Lehrmodul stellt die grundlegenden Konzepte des Datenbankmodells von PCTE vor.

PCTE ist konzipiert worden als ein Nichtstandard-DBMS, das Softwareentwicklungsdaten verteilt verwalten kann und die Konstruktion von Software-Entwicklungsumgebungen (SEU) unterstützt. Derartige Systeme werden auch **Objektmanagementsystem** oder **Repository** (vgl. [IRA]) genannt. Die PCTE-Spezifikationen wurden als ECMA-Standard 149 [PCTE90] und ISO-Standard 13719 [PCTE94] verabschiedet. **H-PCTE** ist eine hochperformante (partielle) Implementierung von PCTE, die durch die Fachgruppe Praktische Informatik an der Universität Siegen realisiert wurde.

Das Datenbankmodell von PCTE hat folgende Hauptmerkmale:

- Es ist auf die *Dokumentverwaltung* und nicht auf die Verwaltung großer Mengen tabellarischer Daten ausgerichtet.
- Seine Grundkonzepte basieren auf dem *Entity-Relationship-Modell*. Eine Objektbank enthält also Objekte und Beziehungen.

  In einigen Details ist es stark beeinflußt von der Denkwelt von UNIX-Dateisystemen.
- Es ist *navigierend*. Die PCTE-Standards definieren keine mengenorientierte Abfragesprache. Man kann aber ohne weiteres Abfragesprachen für PCTE-Objektbanken definieren; ein Beispiel ist NTT, das Teil des Systems H-PCTE ist.
- PCTE unterstützt ein *Verteilung*skonzept, bei dem die Objektbank in mehrere Segmente geteilt ist, die auf mehrere Rechner verteilt werden können, bei dem die Segmentierung und Verteilung von Daten für die Applikationen transparent ist und bei dem einzelne Objekte transparent von einem Rechner auf einen anderen verlagert werden können. Ferner sind nicht nur die Nutzdaten verteilbar, sondern auch die Schemadaten.
- PCTE ist strukturell objektorientiert, d.h. es gibt *komplexe Objekte* und Operationen, die komplexe Objekte als Ganze verarbeiten.

PCTE ist nicht verhaltensmäßig objektorientiert, d.h. man kann Objekte (bzw. Beziehungen) nicht mit Hilfe von benutzerdefinierten Operationen kapseln.

Das vollständige Datenbankmodell von PCTE ist relativ komplex. Diese Komplexität resultiert aus einer Vielzahl von Sonderfällen, die für die eine oder andere wichtige Anwendungssituation benötigt werden, die aber für ein erstes Verständnis von PCTE eher störend sind. Wir vernachlässigen daher diese Sonderfälle zunächst. Wir geben hier nur eine erste, z.T. vergröberte Darstellung der PCTE-Konzepte und konzentrieren uns auf die wichtigen und häufigen Fälle.

2 Objekte, Beziehungen und Attribute

Eine Objektbank besteht aus einer Menge von Objekten und Beziehungen, die ein Netzwerk zwischen diesen Objekten bilden.

2.1 Objekte

Ein **Objekt** entspricht in gewisser Weise einem Record in einer Programmiersprache oder einem Tupel in einer relationalen Datenbank. Jedes Objekt hat einen **Objekttyp**. Dieser Typ legt fest:

1. einen oder mehrere direkte **Elterntypen** (*multiple inheritance*)
2. eine Menge von direkten Attributen
3. eine Menge von zulässigen **ausgehenden Linktypen** (s.u.)

Wenn OT1 **Elterntyp** von OT2 ist, dann nennt man OT2 auch **Subtyp** von OT1, und es gilt dann folgendes: 1. OT2 erbt alle Attribute und zulässigen Linktypen von OT1, 2. OT2 ist typkompatibel zu OT1, d.h. immer da, wo eine Instanz von OT1 erforderlich ist, kann auch eine Instanz von OT2 verwendet werden. Dies ist das normale Prinzip objektorientierter Systeme, wonach Instanzen eines bestimmten Typs immer durch Instanzen eines Subtyps ersetzt werden können.

Die Objekttypen bilden infolge des mehrfachen Erbens zunächst eine Halbordnung. Diese wird weiter dahingehend eingeschränkt, daß sie

genau eine Wurzel hat. Dieser vordefinierte Wurzeltyp heißt `object` , d.h. alle anderen Objekttypen sind stets Subtyp von `object` . Hieraus folgt, daß alle Attribute von `object` auch bei allen anderen Objekttypen vorhanden sind. Diese Attribute werden daher **Standardattribute** genannt. Beispiele für Standardattribute sind diverse Zähler für ankommende oder ausgehende Links und diverse Datumsstempel[1].

Es gibt eine ganze Reihe von Operationen mit Objekten, die wir hier nur andeuten, d.h. wir erklären ihre Parameter und Wirkung erst später im Detail:

`OBJECT_CREATE (objekttyp, ...)`
: legt ein Objekt vom angegebenen Typ an.

`OBJECT_DELETE (...)`
: löscht ein angegebenes Objekt.

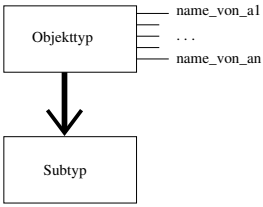
`OBJECT_COPY (...)`
: erzeugt eine Kopie des angegebenen Objekts.

`OBJECT_SET_ATTRIBUTE (...., Attributname, wert)`
: setzt das Attribut mit dem angegebenen Namen auf den angegebenen Wert. Dies entspricht beim Denken in Records einer Wertzuweisung der Form: `recordvariable.Feldname := Wert`

`OBJECT_GET_ATTRIBUTE (...., Attributname) : wert`
: liest den Wert des angegebenen Attributes und gibt ihn zurück. Dies entspricht beim Denken in Records einer Zuweisung in der Form `x := recordvariable.Feldname`

Objekttypen stellen wir in folgender Weise graphisch dar:

[1] Weitere Hinweise hierzu folgen in Abschnitt 4.4.



2.2 Beziehungen, Links und Linktypen

In PCTE gibt es nur binäre Beziehungen.

Diese sind insofern eine Besonderheit von PCTE, als im Gegensatz zu anderen Datenbankmodellen eine Beziehung zwischen zwei Objekten als ein Paar von zwei gegenläufigen gerichteten **Links** realisiert wird. Das folgende Bild zeigt eine Beziehung zwischen zwei Objekten O1 und O2.

Die Bezeichnung Link kommt aus der POSIX-Denkwelt, wo Links Verbindungen zwischen Verzeichnissen und Dateien herstellen.

Jeder Link hat eigene Attribute. Dies klingt zunächst etwas seltsam, da aus einer abstrakten Sicht Attribute normalerweise einer Beziehung zugeordnet werden und die Zuordnung zu einem der beiden involvierten Links eher willkürlich erscheint. Die Ursache hierfür liegt in der Verteilung und wird später diskutiert werden.

Links sind ebenso wie Objekte typisiert. Der **Linktyp** legt folgende Merkmale der Links fest:

1. eine Folge von **Schlüsselattributen**; ein Linktyp kann kein, ein oder mehrere Schlüsselattribute haben
2. eine Menge von Nicht-Schlüsselattributen

3. eine Menge zulässiger **Zielobjekttypen** (dies spielt nur eine Rolle beim Erzeugen von Links)
4. eine **Kategorie**; diese drückt bestimmte semantische Eigenschaften aus; beispielsweise kann ausgedrückt werden, daß das Zielobjekt als Komponente des Ausgangsobjekts zu behandeln ist
5. die **Duplikationseigenschaft** (`DUPLICATED` / `NON-DUPLICATED` ; diese Eigenschaft legt fest, ob der Link mitkopiert wird, wenn das Ausgangsobjekt kopiert wird)
6. diverse andere Eigenschaften, die wir erst später kennenlernen werden
7. den **Umkehrlinktyp**, d.h., wenn man einen Link vom Typ X hat und Y der Umkehrlinktyp von X ist, dann hat dieser Link vom Typ X einen Umkehrlink vom Typ Y.

Wenn X den Umkehrlinktyp Y hat, dann hat umgekehrt Y den Umkehrlinktyp X. Da der Umkehrlinktyp eindeutig ist, ist der Fall, daß zwei Linktypen X1 und X2 den gleichen Umkehrlinktyp Y haben, nicht möglich, denn Y hätte dann ja keinen eindeutigen Umkehrlinktyp.

Auch für Links gibt es eine Reihe von Operationen, die wir hier nur andeuten und später genauer kennenlernen werden:

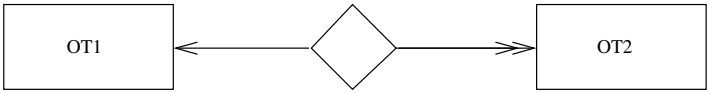
`LINK_CREATE (....)`
: erzeugt einen Link und den Umkehrlink

`LINK_DELETE (....)`
: löscht einen Link mitsamt seinem Umkehrlink

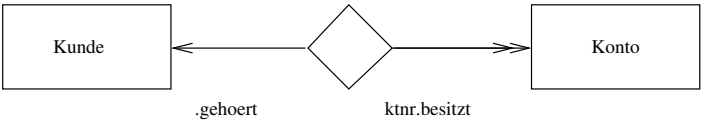
`LINK_GET_ATTRIBUTE (....)`
: liest ein Attribut eines Links

`LINK_SET_ATTRIBUTE (....)`
: überschreibt ein Attribut eines Links

Auch **Linktypen** werden graphisch dargestellt. Diese Darstellung ist im wesentlichen eine der vielen Varianten von Entity-Relationship-Diagrammen.



In dieser Darstellung werden der Linktyp und sein Umkehrlinktyp durch ein einziges Symbol, nämlich durch eine Raute dargestellt. Von dieser Raute aus gehen Pfeile zu den beiden beteiligten Objekttypen. Die Pfeile werden beschriftet in der Form: Schlüsselattribute.Linktypname. Beispiel:



2.3 Attribute

Ein **Attribut** ist definiert durch:

1. seinen Namen (Attributname)
2. seinen Wertebereich (Attributtyp)
3. einen Initialwert
4. eine Duplikationseigenschaft, die festlegt, ob der Wert dieses Attributes beim Kopieren des Objekts mitkopiert wird. Wenn ein Attribut nicht duplizierbar ist, bekommt es in der Kopie eines Objekts den Initialwert.

Als Attributtypen sind nur wenige elementare Datentypen vorgesehen, und zwar: boolean, integer, natural, float (d.h. reelle Zahlen),

string (long field), time, und benutzerdefinierte Aufzählungen. Strings können fast beliebig lang werden und können bereits als eine Realisierung des Konzepts langer Felder angesehen werden. Strukturierte Attributwerte (Felder, Verbunde, Listen usw.) sind nicht vorgesehen[2].

[2] An dieser Stelle sei die Warnung angebracht, daß wir in diesem Text die Begriffe Attribut und Attributtyp wie in der allgemeinen Datenbankliteratur üblich benutzt haben. Diese beiden Begriffe werden in ECMA- bzw. ISO-PCTE völlig anders (und leider nicht sehr sinnvoll) definiert. Das Problem ist, daß eine saubere Unterscheidung von Typ und Instanz beim Attributbegriff nicht ganz einfach ist. Eine saubere Definition des Begriffs Attribut muß so vorgehen, daß man ein Attribut als eine Abbildung auffaßt, die die Entitäten, die ein Attribut haben können (genaugenommen die Menge I der Identifizierer für solche Identitäten) abbildet auf einen Wertebereich W, der dem Attributtyp entspricht, etwa wie folgt:

$$attribut : I \rightarrow W$$

Dieser Wertebereich muß noch durch einen speziellen Wert 'undefiniert' ergänzt werden für den Fall, daß ein Attribut an einer solchen Entität gar nicht definiert ist.

Betrachtet man nun die in Programmiersprachen übliche Begriffswelt, und setzt man Objekte mit Records gleich und Attribute mit Recordkomponenten, so treffen folgende Beobachtungen zu: Es gibt im allgemeinen eine klare Trennung zwischen Typen und Instanzen. Ein Typ ist etwas, was instanziiert werden kann bzw. von dem Instanzen existieren. Ein Attribut (oder eine Recordkomponente) ist hingegen ein Begriff, der sowohl auf der Typ- wie auch auf der Instanzenebene benutzt wird! Es kann jeweils nur im Kontext entschieden werden, welche Ebene hier gemeint ist, d.h. ob hier ein Teil einer Recordtypdefinition gemeint ist oder Teil einer Recordvariablen. Mit einem Ausdruck der Form "Attribut eines Objekttyps" meinen wir daher die Typebene, und ein Attribut auf dieser Ebene ist gegeben durch

- einen Attributnamen und
- einen Attributtyp (also einen Datentyp, der den Wertebereich und die Operationen auf den Werten festlegt).

Da wir ja bei einem Datenbanksystem nur mit der Speicherung von Werten befaßt sind und da Operationen auf den Werten innerhalb der Applikation stattfinden, spielen diese Operationen für unsere Betrachtungen keine weitere Rolle. Im Gegensatz dazu verstehen wir unter einem Ausdruck der Form "Attribut eines Objekts O" die Instanzenebene. Ein Attribut ist auf dieser Ebene definiert durch

- den Attributnamen und
- den Wert, den dieses Attribut bei dem Objekt O hat.

Unter "Attributtyp" verstehen wir stets einen Datentyp, z.B. integer, string usw. Diese Doppeldeutigkeit des Attributbegriffs auf der Typ- und Instanzenebene tritt bei allen Programmiersprachen sowie auch in der normalen Datenbankliteratur auf

Operationen mit Attributen treten nur im Zusammenhang mit Objekten oder Links auf. Von daher sind bereits oben Beispiele angegeben worden. Zusätzlich kann man anstatt mit einem Attribut sofort mit mehreren Attributen an einem Objekt oder Link arbeiten. Die zugehörigen Operationsnamen sind:

```
OBJECT_SET/GET_ATTRIBUTE (...)
OBJECT_SET/GET_SEVERAL_ATTRIBUTES (...)
LINK_SET/GET_ATTRIBUTE (...)
LINK_SET/GET_SEVERAL_ATTRIBUTES (...)
```

---

und kann dadurch erklärt werden, daß Attribute von nicht alleine instanziierbar sind, sondern nur im Kontext eines Objekts, bei dem sie auftreten.

Die in den diversen PCTE-Spezifikationen benutzten Begriffe sind leider nicht kompatibel mit der normalen Begriffswelt (und leider noch nicht einmal völlig durchgängig innerhalb der PCTE-Spezifikation benutzt): Unter "*attribute*" versteht man die Instanzenebene, d.h. den Namen und den Wert eines Attributs an einem Objekt oder Link. Unter "*attribute type*" versteht man den Namen und den Typ des Attributs in einer Typdefinition. Bildlich gesprochen entspricht dies einer Zeile innerhalb einer Recorddefinition, in der der Name und der Typ einer Komponente stehen. Diese Definition des Begriffs Attributtyp stellt ein absolutes Kuriosum dar und ist in keiner Weise konsistent mit dem normalen Begriff Typ, denn Typen können immer instanziiert werden. Eine derartige Zeile in einer Recorddefinition ist natürlich nicht instanziierbar. Nachdem nun der Begriff Attributtyp bereits verbraucht ist, muß eine andere Bezeichnung für das gefunden werden, was normalerweise als Attributtyp bezeichnet wird. Hierfür wird in den Standard-Dokumenten der Begriff "*attribute value type*" definiert. Hiermit ist ein Datentyp gemeint, und zwar derjenige eines Attributes. Der Begriff *value type* ("Wertetyp") ist ein völliger Mißgriff, denn ein Typ impliziert immer bereits die Definition eines Wertebereichs. Von daher müßte man *value type* als Wertebereichs-Wertebereich übersetzen.

Ein weiteres Beispiel für die stellenweise etwas verkorkste Begriffswelt in den PCTE-Standards stellt der Begriff "enumeral type" dar. Hierbei handelt es sich um eine Konstante innerhalb einer Aufzählung, d.h. um einen konstanten Datenwert. Der Begriff Typ ist hier eigentlich deplaziert, er wurde verwendet, weil schlicht alle elementaren Teile von Typdefinitionen als Typen bezeichnet werden.

Im folgenden werden wir weiterhin bei der gängigen Begriffswelt bleiben, was leider dazu führt, daß die in diesem Text verwendeten Begriffe manchmal nicht kompatibel mit den Begriffen sind, die in den Standard-Dokumenten verwendet werden und die zum Teil sogar in den Operationsnamen auftreten.

Standardattribute. Wie schon früher erwähnt worden ist, erben alle Objekttypen vom Objekttyp `object` eine Reihe von Standardattributen, von denen wir hier einige erläutern wollen. Angegeben ist jeweils der Name, der Typ und eine Beschreibung der Bedeutung.

```
exact_identifier : string
```

Der in diesem Attribut gespeicherte String identifiziert ein Objekt eindeutig innerhalb einer Objektbank (bzw. PCTE-Installation); ein einmal benutzter Wert wird nicht wiederverwendet, d.h. der `exact_identifier` hat die Surrogateigenschaft.

```
last_access_time : time
last_modification_time : time
last_change_time : time
```

Hierbei handelt es sich um Zeitstempel, die den Zeitpunkt des letzten lesenden, des letzten verändernden bzw. des letzten "stark verändernden" Zugriffs auf ein Objekt festhalten. Ein Reihe von Administrationsoperationen, wie z.B. eine Änderung der Zugriffsrechte, gelten hierbei nur als eine "leichte" Veränderung und führen nicht zu einer Veränderung des Zeitstempels für "starke" Veränderungen[3].

```
num_incoming_links : integer
```

Dieses Attribut zeigt die Zahl der in einem Objekt endenden Links an.

Die Standardattribute dienen vor allem dazu, den internen Zustand der Objekte bzw. Objektbank sichtbar und für Applikationen lesbar zu machen. Hieraus folgt, daß diese Attribute zwar wie normale Attribute lesbar sind (auch das mit Ausnahmen; es gibt Fälle, in denen diese Attribute noch nicht einmal lesbar sind). Diese Attribute sind aber nie mit normalen Operationen schreibbar; ihr Wert ändert sich nur indirekt als Seiteneffekt von anderen Operationen.

[3] Diese Zeitstempel sind in H-PCTE nicht realisiert, sind aber in den Typdefinitionen aus Kompatibilitätsgründen vorhanden!

2.4 Ein Beispiel

Mit den bisher kennengelernten Konzepten können wir nun bereits ein Beispiel für die Typen einer Applikation betrachten (in einer pseudocode-artigen Syntax):

```
Objekttyp Person
Attribute:
   Nachname: string, initial: ''
   Vorname: string, initial: ''

Objekttyp Student
Subtyp von Person
zusaetzliche Attribute:
   Matrikelnummer: natural, initial: 0
Links:
   wird_betreut_von [natural] nach Mitarbeiter

Objekttyp Mitarbeiter
Subtyp von Person
zusaetzliche Attribute:
   Personalnummer: string, initial: ''
Links:
   betreut [natural] nach Student
```

3 Identifizierung von Objekten

Jedes DBMS (bzw. genauer gesagt jedes Datenbankmodell) muß irgendwelche Möglichkeiten vorsehen, auf einzelne Objekte zuzugreifen und sie hierzu zunächst einmal zu identifizieren. Man unterscheidet hier zwei grundlegende Vorgehensweisen:

1. Nichtnavigierende Sprachen: Das beste Beispiel hierfür sind relationale Abfragesprachen, in denen die Objekte, mit denen man arbeiten

möchte, deskriptiv spezifiziert werden (siehe auch [DBI]).
2. Navigierende Zugriffsverfahren: Bei diesen wird ausgehend von bereits lokalisierten Objekten mit Hilfe von gewissen Operationen zu anderen Objekten hin navigiert.

PCTE gehört zur zweiten Art von DBMS. PCTE kennt keinen Direktzugriff zu Mengen von Objekten anhand von deskriptiven Merkmalen, sondern nur Navigationen über Links, wobei die Links anhand ihrer Linknamen spezifiziert werden.

3.1 Linknamen

Wie schon früher erwähnt ist das Konzept der Links stark angelehnt an das Konzept der Links innerhalb von POSIX (oder sonstigen Dateisystemen). Jeder Link hat einen Linknamen, und dieser ist lokal eindeutig innerhalb der Menge der Links, die vom gleichen Objekt ausgehen. Mit anderen Worten liegt hier eine *lokale Schlüsseleigenschaft* vor, während im Vergleich dazu in relationalen Systemen Schlüsseleigenschaften stets global sind. Der Unterschied wird einem klar, wenn man die Menge der Links eines Typs betrachtet. Es kann durchaus mehrere Links des gleichen Typs mit dem gleichen Linknamen in der Objektbank geben, allerdings müssen diese Links dann von verschiedenen Objekten ausgehen.

Die Syntax von Linknamen wurde so gewählt, daß die typischen Linknamen in Dateisystemen unverändert übernommen werden können. Die Syntax eines Linknamens ist:

```
Schluessel '.' Linktypname
```

Der Schlüssel (*key*) eines Links besteht aus den Werten der Schlüsselattribute, die durch Punkte getrennt werden[4]. Es sei daran erinnert, daß ein Linktyp eine *Folge* (nicht Menge) von Schlüsselattributen definiert. Entsprechend dieser Folge sind die Schlüsselattributwerte im Schlüssel enthalten.

[4]Diese Begriffswahl kann zu Mißverständnissen führen, denn nur der *gesamte* Linkname ist ein Identifizierungsschlüssel für die Links, die von einem Objekt ausgehen.

Beispiele für Linknamen mit verschieden vielen Schlüsselattributen sind:
0 Schlüsselattribute: `.doku .headers ....`
1 Schlüsselattribute: `einfuehrung.tex einfuehrung.toc main.c ..`
2 Schlüsselattribute: `einfuehrung.v7.tex main.v5.c ...`
3 Schlüsselattribute: `einfuehrung.v7.5.tex main.v5.2.c ...`

Wenn ein Linktyp kein Schlüsselattribut hat, dann kann von einem Objekt maximal *ein* Link dieses Typs ausgehen, d.h. die **Kardinalität** dieses Linktyps ist [0,1], was in der PCTE-Terminologie auch als "*one*" bezeichnet wird. Dargestellt werden Links dieses Typs in den Schema-Diagrammen mit einem einfachen Pfeil $\rightarrow$.

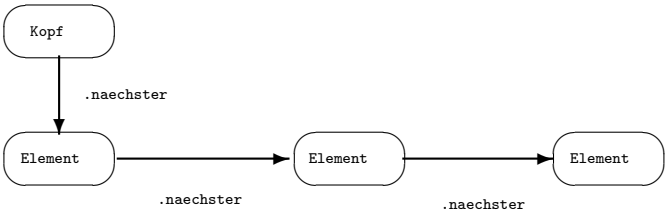
Wenn ein Linktyp wenigstens ein Schlüsselattribut hat, können beliebig viele Links dieses Typs von einem Objekt ausgehen (natürlich mit verschiedenen Schlüsselattributwerten). Links dieses Typs haben Kardinalität [0,∞], was in der PCTE-Terminologie auch als "*many*" bezeichnet wird, und werden mit einem Pfeil mit doppelter Pfeilspitze dargestellt: $\twoheadrightarrow$

Im folgenden geben wir einige Beispiele von Objekten und Links, die diese Objekte verbinden, an. Wir stellen diese Beispiele graphisch als ein Netzwerk dar. Kästen stellen Objekte dar. Innerhalb der Kästen steht der Name des Typs dieses Objekts. Links werden als Pfeile dargestellt, die mit dem Linknamen beschriftet sind. Die Umkehrlinks werden in den meisten Fällen nicht dargestellt.

Beispiel 1: Verzeichnisse in POSIX-Dateisystemen

Dateistrukturen in POSIX (oder anderen Dateisystemen) lassen sich fast immer direkt in Netzwerke von PCTE-Objekten und Links umsetzen. Für einen Dateinamen wie `main.c` muß ein Linktyp namens `c` definiert werden, der ein Schlüsselattribut hat, dessen Wertebereich Strings sind. Zulässiger Zielobjekttyp für `c` sollte naheliegenderweise nur der Objekttyp sein, der C-Programme repräsentiert. Analog muß man für `main.o` , `papier.tex` , `papier.dvi` usw. jeweils eigene Linktypen definieren.

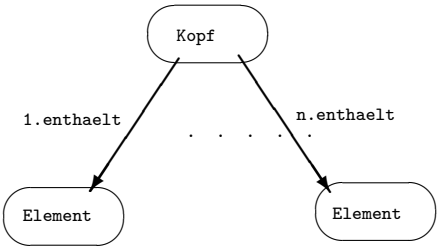
Beispiel 2: listenartige Strukturen



Umkehrlinktyp zu `naechster` : `voriger`

Beispiel 3: eindimensionale Felder

Arrays sind in der Programmierung eine sehr häufige Form der Datenstrukturierung. PCTE hat kein Konzept, durch das Arrays direkt nachgebildet werden können. Stattdessen muß man ein Array als Ganzes durch ein Kopfobjekt modellieren, von dem aus Links zu den Elementen des Arrays gehen.

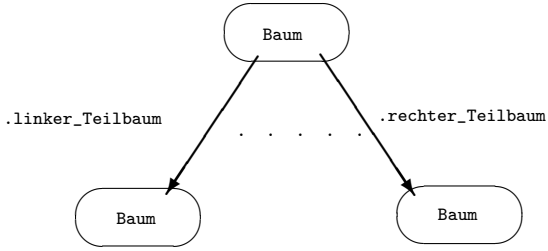


Die Links werden am einfachsten mit einem numerischen Schlüssel, der den Array-Positionen entspricht, durchnumeriert. Im folgenden Beispiel heißt der Linktyp `enthaelt` (der Umkehrlinktyp könnte z.B. `ist_enthalten_in_feld` heißen), und es sind Links mit Schlüssel 1 bis n angedeutet. Auf diese Weise kann ein `array [1..n]` simuliert werden. Es ist hier allerdings nicht garantiert, daß nicht z.B. ein Link mit Schlüssel n+1 erzeugt wird oder daß Links mit Schlüsseln zwischen

1 und n fehlen. Es ist also nicht garantiert, daß das Array vollständig vorhanden ist oder daß die Array-Grenzen nicht überschritten werden.

Beispiel 4: rekursive Datenstrukturen

Die direkte Modellierbarkeit rekursiver Datenstrukturen ist eine der wesentlichen Anforderungen an Objektmanagementsysteme. Als Beispiel betrachten wir hier einen binären Baum.



Von einem Objekt des Typs Baum gehen zwei Links mit Linknamen `.linker_Teilbaum` und `.rechter_Teilbaum` aus. Die beiden Links haben kein Schlüsselattribut und verschiedene Linktypen, die Kardinalität der Linktypen ist also "eins". Auf diese Weise kann sichergestellt werden, daß maximal genau zwei Teilbäume enthalten sind. Die Umkehrlinktypen könnten z.B. `ist_linker_Teilbaum_von` und `ist_rechter_Teilbaum_von` heißen. Es ist *nicht* möglich, für beide Links den gleichen Umkehrlinktyp zu nehmen, denn dieser Linktyp hätte dann keinen eindeutigen Umkehrlinktyp.

Alternativ hätte man einen einzigen Linktyp `Teilbaum` definieren können und für den linken und rechten Teilbaum verschiedene Schlüssel verwenden können, z.B. 1 und 2. Bei einem numerischen Schlüssel allerdings gibt es dann keine Garantie, daß nicht ein Link mit Linknamen `3.Teilbaum` erzeugt wird. Dies bedeutet also, daß das Konsistenzkriterium eines Binärbaums, daß maximal zwei Teilbäume vorhanden sind, verletzt werden könnte.

3.2 Referenzobjekte

Die Linknamen, die wir bisher kennengelernt haben, ermöglichen es uns, von einem Objekt durch Angabe eines Linknamens über den so bezeichneten Link zu einem anderen Objekt zu navigieren. Wir können also ausgehend von bereits erreichten Objekten andere Objekte erreichen. Die Frage bleibt dabei offen, wie wir überhaupt zum ersten Objekt hinkommen, d.h. wo wir mit der Navigation anfangen. Um dies zu ermöglichen, gibt es eine Reihe von direkt referenzierbaren Objekten, sog. **Referenzobjekte**, von denen wir hier nur die wichtigsten tabellarisch angeben:

Kurz- / Langname	Bedeutung
`_` `$common_root`	allgemeines Wurzelobjekt einer Installation (analog zum Verzeichnis '/' in POSIX)
`~` `$home_object`	Heim-Objekt des Benutzers des laufenden Prozesses

Von POSIX-Shells ist man daran gewöhnt, mit '`.`' ein aktuelles Objekt (Verzeichnis) des laufenden Prozesses angeben zu können. Ein solches Referenzobjekt ist aber nur innerhalb eines Kommandointerpreters (also nicht für andere Prozesse) gültig und muß daher in der Applikation realisiert werden.

Der PCTE-Standard definiert noch einige weitere Referenzobjekte, die allerdings weniger häufig benutzt werden und in H-PCTE nicht realisiert sind.

3.3 Pfadnamen von Objekten

Objekte können in PCTE mit Hilfe von Pfadnamen bezeichnet werden. Ein **Pfadname** realisiert die Idee, daß man ausgeht von einem Referenzobjekt und von dort aus über beliebig viele Links zu dem gewünschten Objekt hinnavigiert. Der Referenzobjektname steht links in einem (absoluten) Pfadnamen. Ihm folgen durch '/' getrennt beliebig viele Linknamen.

Anwendungsprozesse können auch dynamisch "Zeiger" auf Objekte (sog. Objektreferenzen; diese lernen wir später genauer kennen) anlegen; auch von so referenzierten Objekten aus kann weiter navigiert werden, in diesem Fall durch Angabe eines **relativen Pfadnamens**, in dem der Referenzobjektname fehlt. Die Syntax von Pfadnamen ist insgesamt wie folgt:

```
Pfadname = Referenzobjektname [ '/' relativer Pfadname ]
         | relativer Pfadname
relativer Pfadname = Linkname { '/' Linkname }
```

Beispiele für Pfadnamen (in Klammern ist der naheliegende Typ des Zielobjekts angegeben):

```
1) ~/uni_beispiel.x/meier.ist_uni_angehoeriger      (person)
2) ~/uni_beispiel.x/schmidt.ist_uni_angehoeriger    (person)
3) ~/uni_beispiel.x/schmidt.ist_uni_angehoeriger/
   meier.betreut                                   (student)
4) ~/uni_beispiel.x/schmidt.ist_uni_angehoeriger/
   meier.betreut/schmidt.wird_betreut_von      (mitarbeiter)
```

Die Pfadnamen 1 und 3 sind zwar textuell verschieden, können aber ohne weiteres zum gleichen Objekt führen. Gleiches gilt für die Pfadnamen 2 und 4. Dies ist ein erheblicher Unterschied zur Denkweise, die man aus Dateisystemen gewohnt ist[5]. In Dateisystemen ist die Struktur der Verzeichnisse fast immer baumartig. Lediglich auf der untersten Ebene können z.B. in POSIX einzelne Dateien in mehreren Verzeichnissen enthalten sein. Derartige Restriktionen bestehen in PCTE nicht, d.h. die Links zwischen den Objekten bilden ein vollkommen beliebiges Netzwerk, bei dem im allgemeinen viele Pfade zum gleichen Objekt

[5] Wobei wir symbolische Links ausnehmen. Links in PCTE entsprechen "harten" Links in POSIX-Dateisystemen, es gibt in PCTE kein den symbolischen Links entsprechendes Konzept.

führen können und bei dem innerhalb von Pfaden ohne weiteres Schleifen auftreten können. Hieraus folgt als erste Erkenntnis, daß es für das gleiche Objekt im allgemeinen beliebig viele Pfadnamen geben kann.

Umgekehrt ist keineswegs sicher, daß mit dem gleichen Pfadnamen immer das gleiche Objekt bezeichnet wird. Ein Link, der innerhalb eines Pfadnamens auftritt, kann nämlich ohne weiteres gelöscht werden und es kann zu einem späteren Zeitpunkt ein Link mit dem gleichen Namen, der zu einem anderen Objekt führt, wieder angelegt werden, so daß die Bedeutung des gesamten Pfadnamens indirekt mitgeändert worden ist.

Pfadnamen sind somit kein Konzept oder Mittel, mit dessen Hilfe man Objekte längerfristig eindeutig bezeichnen kann. Wir werden später andere Mittel kennenlernen, mit denen dies möglich ist. Um Beispiele für Pfade und Links kennenzulernen, kann man z.B. mit einem der Browser von H-PCTE in einer Objektbank navigieren. Eine erste *Übungsaufgabe* ist daher, mit einem dieser Browser in der Beispieldatenbank herumzufahren und ein paar Objekte und Links anzusehen[6].

Hinweise zur Benutzung von Referenzobjekten. Wenn man von _ oder ~ zu bestimmten Objekten einer Applikation navigiert, können relativ lange Pfade entstehen, die unhandlich sind und auch nur ineffizient verarbeitbar sind. Im Vorgriff auf Lehrmodul [PRF] sei hier schon erwähnt, daß eine Applikation beliebige Objekte zu zusätzlichen "Stützpunkten" machen kann, auf die dann direkt zugegriffen und von denen aus effizient weiternavigiert werden kann.

4 Schemaverwaltung

4.1 Selbstreferentialität und die Metadatenbank

Wir haben oben bereits Möglichkeiten kennengelernt, Objekte bzw. Links eines bestimmten Typs zu erzeugen. In den Operationen muß der gewünschte Typ in Form eines Parameters bzw. als letzter Teil des

[6] Hinweise zur Installation des H-PCTE-Systems sind über `http://pi.informatik.uni-siegen.de` erhältlich.

Linknamens angegeben werden. Das OMS muß nun bei Angabe eines solchen Typs prüfen, ob er existiert und welche Eigenschaften er hat. Mit anderen Worten müssen als Teil jeder Objektbank nicht nur die eigentlichen Instanzen der benutzerdefinierten Typen gespeichert werden, sondern auch Informationen über die Typen selbst. Derartige Daten nennt man auch **Metadaten**, also Daten über Daten.

Wie wir auch schon gesehen haben, muß es für Anwendungen irgendwie möglich sein, neue Typen zu erzeugen. Darüber hinaus wird man natürlich auch fordern, daß man Informationen über die vorhandenen Typen auf irgendeine Weise abfragen kann. Mit anderen Worten muß jedes OMS intern ein Typverwaltungssystem haben sowie Schnittstellen, um Typen anzulegen und Informationen über Typen abzufragen. Für die Gestaltung der Schnittstelle zu diesem Typverwaltungssystem gibt es zwei prinzipielle Ansätze:

1. Man definiert geeignete Operationen, durch die alle Eigenschaften der Typdefinition abgefragt werden oder soweit zulässig verändert werden können. Beispielsweise könnte es eine Operation geben, die zu einem Objekttyp die Menge seiner Elterntypen liefert.
2. Alternativ kann man die Metadaten als ganz normale Daten ansehen und demzufolge durch Objekte und Beziehungen darstellen. Derartige OMS nennt man **selbstreferentiell**. Das bedeutet, daß die Datenmodellierungseigenschaften des OMS dazu benutzt werden, Metadaten oder relevante interne Verwaltungsstrukturen als ganz normale Daten darzustellen.

PCTE bietet beide Ansätze parallel an. Zum einen wird eine Vielzahl von Operationen zum Abfragen und Modifizieren von Typeigenschaften angeboten. Gleichzeitig ist PCTE sehr weitgehend selbstreferentiell. Nicht nur alle Typdefinitionen, sondern auch Datenträger, Benutzer, Prozesse und viele andere Dinge werden durch Objekte und Beziehungen repräsentiert. Die Objekte und Beziehungen, die die Typdefinitionen repräsentieren, bilden die sogenannte **Metadatenbank**. Die Metadatenbank ist über den Pfad `_/.schemas` erreichbar.

Die Metadatenbank enthält alle in einer PCTE-Installation bekannten Typen und entspricht daher dem konzeptuellen Schema in der

ANSI/SPARC-Schichtenarchitektur. Interne Schemata sind in PCTE nicht vorgesehen; dies liegt hauptsächlich am Verteilungskonzept von PCTE (unterstellt werden große, heterogene Workstation-Netze, keine mengenorientierte Abfragesprache). Das Sichtenkonzept besprechen wir im folgenden Abschnitt.

Die Metadatenbank hat eine recht komplexe Struktur, da sie direkt und feinkörnig die komplette Typwelt von PCTE wiedergibt. Für einen PCTE-Anfänger ist sie zu komplex; sinnvollerweise sollte man zunächst mit einfacheren Strukturen üben. Details der Metadatenbank werden daher erst in Lehrmodul [PMDB] erklärt.

4.2 Arbeitsschemata

Eine der wesentlichen Leistungen eines DBMS besteht darin, unterschiedlichen Anwendungsprogrammen eigene Sichten auf die Daten zu realisieren. In PCTE entspricht der Begriff **Arbeitsschema** (*working schema*) dem Begriff Sicht (oder externes Schema). Das Arbeitsschema eines Prozesses definiert eine Menge von Typen, die für diesen Prozeß sichtbar sein können, ferner ggf. noch bestimmte Einschränkungen bzgl. der Zugriffsart zu Instanzen dieser Typen. Ein Prozeß kann also nur auf denjenigen Typen operieren, die in seinem Arbeitsschema enthalten sind. Alle anderen Typen sind für ihn "unsichtbar".

Der Sichtenmechanismus von PCTE unterscheidet sich infolge der Objektorientierung und anderer Gründe erheblich vom Sichtenmechanismus, den man aus relationalen Datenbanken kennt:

- Eine Sicht in einem relationalen System wird definiert als virtuelle Relation. Diese Relation hat einen eigenen Namen und entspricht daher einem eigenen neuen Objekttyp. In PCTE entstehen durch Sichten *keine* neuen Typen, stattdessen wird für den laufenden Prozeß die Definition vorhandener Typen modifiziert. Hieraus folgt:
- Die Definition eines Relationentyps ist in relationalen Systemen für alle Applikationen gleich, in PCTE können verschiedene Prozesse verschiedene Definitionen des gleichen Objekttyps "sehen". In PCTE kann ein Prozeß diese Definition sogar zur Laufzeit ändern.

- In einem relationalen System wird bei jedem DML-Kommando die zu benutzende Sicht durch Angabe der zu benutzenden Relationen explizit angegeben. In PCTE arbeiten die Operationen nur mit den "wirklichen" Typen, die zu benutzende Sicht ist eine Eigenschaft des Prozesses.
- Sichten in PCTE sind identitätserhaltend, Sichten in relationalen Systemen nicht. Ein Tupel in einer virtuellen Relation wird z.B. dann, wenn die Sicht als Verbund definiert wird, aus mehreren anderen Tupeln aus realen Relationen abgeleitet. Die Identität der Ausgangs"objekte" geht dabei verloren. Eine Konsequenz hieraus ist, daß in solche virtuellen Relationen nicht ohne weiteres Tupel eingefügt oder vorhandene Tupel geändert werden können.

  In PCTE können derartige Sichten nicht gebildet werden. In PCTE entspricht ein Objekt unter einer Sicht immer einem realen Objekt, es hat unter jeder Sicht immer die gleiche Identität (Surrogat) und den gleichen Typ – letzteres entspricht der objektorientierten Denkweise viel besser als der relationale Ansatz. Erzeugende, löschende und ändernde Operationen sind in PCTE, sofern sie überhaupt zulässig sind, auch unter jeder beliebigen Sicht zulässig. In PCTE besteht daher gar kein Anlaß, einen Begriff wie virtuelles Objekt zu bilden.

  Dies ist sehr wichtig, denn im Gegensatz zu konventionellen Anwendungen ist der überwiegende Anteil der Applikationen in einer SEU schreibend. Der Sichtenmechanismus in relationalen Systemen erlaubt zwar komplexere Sichtendefinitionen, wenn Verbundoperatoren oder ähnlich mächtigen Operatoren benutzt werden, dies ist aber von beschränktem Nutzen, wenn diese Sichten nicht "schreibbar" sind.

4.3 Globale und schemaspezifische Typeigenschaften

Eine typische Anwendung eines Sichtenmechanismus besteht z.B. darin, einige Attribute eines Objekttyps auszublenden. Die Menge der Attribute eines Objekttyps kann daher in verschiedenen Arbeitsschemata verschieden sein. Umgekehrt wäre es sehr problematisch, wenn z.B. die

Folge der Schlüsselattribute eines Linktyps in verschiedenen Arbeitsschemata variieren könnte. Es gibt daher Typeigenschaften, die *nicht* variieren können, d.h. die in allen Arbeitsschemata gleich sind[7], die somit *global* gültig sind. Diese Eigenschaften können auch nicht mehr verändert werden, nachdem die Typdefinition einmal in der Objektbank angelegt worden ist. Globale Eigenschaften sind:

- bei Objekttypen: die Menge der Elterntypen
- bei Linktypen: die Folge der Schlüsselattribute, die Kategorie, der Umkehrlinktyp und die Duplikationseigenschaft
- bei Attributen: der Initialwert und die Duplikationseigenschaft.

Alle anderen Typeigenschaften, u.a. die Menge der Attribute eines Objekttyps, sind abhängig vom Arbeitsschema. Diese Eigenschaften der Typdefinitionen sind auch veränderbar; sie können sogar noch nachträglich modifiziert werden, nachdem bereits Instanzen der Typen erzeugt worden sind. Beispielsweise kann bei einem existierenden Objekttyp `person` ein neues Attribut `sprachkenntnisse` hinzugefügt werden. Bei Instanzen von `person` , die vor dieser Schemaänderung erzeugt worden sind, kann das Attribut `sprachkenntnisse` ohne explizite Vorbereitungen gelesen werden; man erhält dann den Initialwert. Wenn man es überschreibt, wird die interne Speicherstruktur automatisch konvertiert, um den neuen Attributwert aufnehmen zu können.

4.4 Schema Definition Sets

In PCTE basiert die Verwaltung aller Typen, die in einer Installation vorhanden sind, auf dem Konzept des **Schema Definition Set** (**SDS**): Ein SDS gruppiert eine Menge zusammengehöriger Typdefinitionen und hat einen systemweit eindeutigen **SDS-Namen**. Die Metadatenbank besteht also aus einer Reihe von SDS. Ein SDS mit Namen XX wird durch das Objekt mit dem Pfadnamen `_/.schemas/XX.known_sds` repräsentiert. SDS ermöglichen es, die Verwaltung von Typen zu modularisieren. Hierfür gibt es zwei wichtige Motive:

[7] Wie dies technisch erreicht wird, ist hier zunächst belanglos.

- Zum einen soll es möglich sein, die Typen, die zu einem bestimmten neuen Werkzeug gehören, separat in ein oder mehrere SDS zu verkapseln, die zu diesem Werkzeug gehören, und diese SDS einzeln in der Objektbank zu installieren.
- Ein weiterer wichtiger Grund zur Modularisierung der Typdefinitionen ist die Verteilung. Es soll möglich sein, Teile der Typdefinitionen auf verschiedenen Rechnern zu speichern.

Vordefinierte SDS. Es gibt mehrere vordefinierte SDS, die in jeder Objektbank initial vorhanden sind und die nicht modifiziert werden sollten:

1. Das SDS `system` enthält diverse grundlegende Typen, z.B. `object, process` usw., die für die Selbstmodellierung des Systems wichtig sind und als Grundlage für weitere Typen benötigt werden. Die meisten Typen im SDS `system` sind allerdings für normale Applikationen vollkommen irrelevant, z.B. die meisten Standardattribute des Objekttyps `object` . Daher ist es sinnvoll, aus den häufig gebrauchten Typen aus `system` ein neues SDS zu bilden und nur dieses zu verwenden. Wie dies geht, werden wir später kennenlernen.
2. Das SDS `metasds` enthält die Typen, die in der Metadatenbank auftreten. Diese nennt man auch Metatypen, d.h. es sind Typen, deren Instanzen Typen repräsentieren.
3. Das SDS `discretionary_security` enthält diverse Typen, die für die Repräsentation und Handhabung der Zugriffskontrollen benötigt werden, z.B. Typen für die Gruppenverwaltung, für die Zugriffskontrollisten usw.
4. Das SDS `mandatory_security` enthält analog dazu Typen, die für die Informationsflußkontrollen (Typen für *mandatory access controls*, MAC) benötigt werden.
5. Das SDS `accounting` enthält Typen, die für die Abrechnungsfunktionen von PCTE benötigt werden.

Von den im Standard enthaltenen SDS sind in H-PCTE nicht alle

komplett enthalten oder z.T. mit Modifikationen enthalten:

- `system` , `metasds` und `discretionary_security` sind wie im Standard definiert enthalten.
- `mandatory_security` und `accounting` sind nicht enthalten, weil diese Funktionsbereiche in H-PCTE nicht realisiert sind.
- `hpcte` ist ein zusätzliches SDS, das für Interna von H-PCTE benötigt wird.

Überschneidungen von SDS. Ein SDS "enthält" Mengen von Typdefinitionen. Ein wichtiger Punkt ist nun, daß diese Mengen i.a. *nicht disjunkt* sind, d.h. es kann sein, daß der gleiche Typ in mehreren SDS vorhanden ist[8]. Dabei wird sichergestellt, daß die globalen Typeigenschaften in allen SDS gleichartig definiert sind; beispielsweise hat ein Linktyp in allen SDS, in denen er auftritt, die gleiche Kategorie und den gleichen Umkehrlinktyp.

4.5 Das Setzen des Arbeitsschemas eines Prozesses

Ein Prozeß kann sein Arbeitsschema jederzeit verändern. Hierzu gibt es eine Operation zum Setzen des Arbeitsschemas

`process_set_working_schema (..,Folge_von_SDSnamen)`.

Ein Beispiel eines Aufrufs könnte sein

```
process_set_working_schema (..,(uni, system))
```

Die Operation `process_set_working_schema` erzeugt folgendes Arbeitsschema:

1. Es enthält alle Typen, die in wenigstens einem der aufgeführten SDS vorkommen.

[8] Mittel, wie man einen solchen Zustand herstellt, werden wir erst später kennenlernen (s. Abschnitt 3.2 in [PDDL]).

2. Die Eigenschaften dieser Typen werden folgendermaßen bestimmt:
   Sofern ein Typ nur in einem SDS definiert ist, ergeben sich seine Eigenschaften aus dieser Definition. Ist er in mehreren SDS enthalten, so sind nach Voraussetzung die *globalen* Typeigenschaften in allen SDS identisch definiert, so daß auch im Arbeitsschema diese einheitliche Festlegung gilt.

   Für die *schemaspezifischen* Typeigenschaften gilt i.w., daß die Eigenschaften aus den verschiedenen SDS "vereinigt" werden: Beispielsweise hat ein Objekttyp in einem Arbeitsschema die Vereinigung aller Attribute, die er in irgendeinem der SDS hat, aus denen das Arbeitsschema besteht. Ein Objekttyp O kann z.B. in einem SDS S1 ein Attribut A1 haben und in einem SDS S2 ein Attribut A2. In dem Arbeitsschema bestehend aus den beiden SDS hat O die Attribute A1 und A2. Diese Vereinigungsregel gilt analog:

   - bei Objekttypen: für Attribute und die ausgehenden Linktypen
   - bei Linktypen: für Nichtschlüsselattribute und die Zielobjekttypen

   Attribute haben keine schemaspezifischen Typeigenschaften.

Initiales Arbeitsschema. Das initiale Arbeitsschema eines Prozesses ist anfangs leer, d.h. es enthält 0 Typen. Mit anderen Worten sieht der Prozeß zunächst überhaupt nichts. Hieraus folgt, daß der Prozeß als erstes immer ein geeignetes Arbeitsschema setzen muß, um arbeiten zu können. Wenn man in einem Prozeß (z.B. einem Browser) nachsehen möchte, welche SDS überhaupt verfügbar sind, muß man allerdings die Metadatenbank lesen können. Dies ist nur dann möglich, wenn man das `metasds` im Arbeitsschema hat.

4.6 Typnamen

Bei der bisherigen Behandlung der Schemamechanismen haben wir einen wichtigen (und leider etwas komplizierten) Aspekt ausgeklammert, nämlich Typnamen. In früheren Beispielen haben wir bereits

Typnamen verwendet (insb. die Linktypnamen innerhalb von Linknamen).

Die Namen eines Typs hängen vom Arbeitsschema ab. Ein Typ hat im Arbeitsschema i.a. mehrere Namen, die unterschiedslos benutzt werden können. Es gibt drei verschiedene Arten von Typnamen, die wir i.f. erläutern.

Lokale Typnamen in SDS. Innerhalb eines SDS hat ein darin enthaltener Typ einen eindeutigen lokalen Typnamen[9]. Das SDS bildet einen einzigen Namensraum, d.h. ein Objekttyp und ein Attribut mit gleichem Namen sind nicht erlaubt.

Die Namenseindeutigkeit gilt nur lokal innerhalb eines SDS und nicht global innerhalb der Menge aller SDS. In zwei verschiedenen SDS können also durchaus unterschiedliche Typen enthalten sein, die den gleichen lokalen Namen in den beiden SDS haben.

Wir hatten schon früher erwähnt, daß ein bestimmter Typ in mehreren SDS enthalten sein kann. In jedem SDS kann er einen anderen Namen haben, d.h. in unserer oben etablierten Begriffswelt würde man den lokalen Namen als eine sichtenspezifische Eigenschaft eines Typs ansehen.

In einem Arbeitsschema hat ein Typ im einfachsten Fall *jeden* lokalen Namen, den er in einem der beteiligten SDS hat. Wenn das gleiche Attribut beispielsweise im SDS `X` den lokalen Namen `Nachname` hat und im SDS `Y` den lokalen Namen `Familienname` und wenn das Arbeitsschema die beiden SDS enthält, kann das Attribut mit beiden lokalen Namen unterschiedslos angesprochen werden.

Eine Ausnahme von der vorstehenden Regel wird bei einem Namenskonflikt notwendig. Ein Namenskonflikt liegt vor, wenn der gleiche lokale Typname in mehreren SDS des Arbeitsschemas auftritt und in diesen SDS unterschiedliche Typen bezeichnet. In diesem Fall ist die *Reihenfolge* der SDS, die beim Setzen des Arbeitsschemas angegeben

[9] Es kann auch namenlose Typen geben, was an dieser Stelle aber nicht interessiert.

wurde, relevant[10]: der lokale Name bezeichnet dann den Typ gemäß dem *ersten* SDS, in dem dieser Name auftritt. Andere Typen, die in einem der folgenden SDS vorkommen und die dort den gleichen lokalen Namen haben, haben somit im Arbeitsschema diesen lokalen Namen nicht, haben also u.U. gar keinen lokalen Namen mehr! Die vorderen SDS überdecken also die lokalen Namen der hinteren SDS.

Volle Typnamen. Weil der lokale Name eines Typs innerhalb eines SDS ggf. nicht ausreicht, um den Typen eindeutig innerhalb der Objektbank zu identifizieren, gibt es eine lange Form: diese besteht aus dem Namen des SDS, gefolgt von '-' und dem lokalen Namen, und wird auch als der **volle Typname** bezeichnet. Da SDS-Namen global eindeutig sind, sind auch die vollen Typnamen global eindeutig.

In unseren obigen Beispielen für Pfadnamen trat unter anderem der Linktypname `betreut` auf. Hierbei handelte es sich um einen lokalen Namen. Wenn das SDS, in dem dieser Linktyp definiert ist, `uni` heißt, dann ist der volle Typname dieses Linktyps `uni-betreut` .

In einem Arbeitsschema hat ein Typ immer *alle* vollen Namen gemäß den SDS des Arbeitsschemas, in denen er vorkommt.

Typidentifizierer. Es ist bei bestimmten Gelegenheiten notwendig, auch mit Objekten bzw. Links umgehen zu können, deren Typ nicht im aktuellen Arbeitsschema enthalten ist, die also eigentlich unsichtbar sind[11]. Für diesen Fall sind die sogenannten **Typidentifizierer**

[10] Dies ist übrigens der einzige Punkt, bei dem diese Reihenfolge relevant ist.

[11] Dies widerspricht natürlich dem Begriff (Un-) Sichtbarkeit.

Das simpelste Beispiel sind Daten, deren Typ gelöscht worden ist. Hierzu ist anzumerken, daß Typdefinitionen in der Metadatenbank gelöscht werden können, auch wenn noch Instanzen dieser Typen in der Objektbank vorhanden sind. Wegen der Verteilungsannahmen können diese Instanzen i.a. nicht sofort mit der Löschung des Typ beseitigt werden.

Ein anderes Beispiel für eine solche Situation ist das Auflisten der Links, die von einem Objekt ausgehen. Um ein Objekt löschen zu können, darf es keine "störenden" Links geben. Wenn unsichtbare Links überhaupt nicht anzeigbar wären, könnte man nicht den geringsten Hinweis herausfinden, warum ein Objekt nicht gelöscht werden kann. Dies ist in der Praxis nicht akzeptabel.

vorhanden. Ein Typidentifizierer ist eine Zeichenkette, die mit einem Tiefstrich '_' beginnt; der Tiefstrich ist bei benutzerdefinierten Typnamen nicht als erstes Zeichen zugelassen. Die Syntax dieser Zeichenkette ist ansonsten in den Standards undefiniert, kann also von jeder PCTE-Implementierung selbst festgelegt werden.

Hintergrund dieses Typidentifizierers ist, daß das System immer dann, wenn ein neuer Typ angelegt wird, für diesen Typ ein Surrogat vergibt. Ein **Surrogat** ist ein Bezeichner, der nur einmal während der Lebensdauer des gesamten Systems vergeben wird, der also nicht wiederverwendet wird und der damit den bezeichneten Gegenstand innerhalb der gesamten Lebensdauer des Systems eindeutig identifiziert. Intern arbeitet das System nur mit diesen Typidentifizierern.

Typidentifizierer sind systemweit eindeutig; jeder in einem Arbeitsschema enthaltene Typ hat daher genau einen Typidentifizierer.

Bei der Rückgabe von Typnamen an ein Benutzerprogramm – z.B. bei der Anzeige von Linknamen im Browser – wird

- irgendein lokaler Typname benutzt, sofern wenigstens einer vorhanden ist;
- andernfalls, sofern der Typ im Arbeitsschema enthalten ist, irgendeiner der vollen Typnamen benutzt;
- andernfalls der Typidentifizierer.

Übungsaufgaben. Die Wirkung von externen Schemata lernt man am besten kennen, indem man mit verschiedenen externen Schemata über die gleichen Datenbestände navigiert. Machen Sie hierzu am Rechner[12] folgende Übungsaufgaben:

- mit einem Arbeitsschema bestehend aus SDS `system` in der Objektbank herumwandern und aufpassen, was passiert (also normalerweise nach Starten der Applikation; der Browser setzt initial ein Arbeitsschema bestehend aus den SDS `system` , `security` und `hpcte`)

[12] Ggf. müssen Sie hierzu erst H-PCTE installieren; eine Anleitung befindet sich in [HINS].

- dito, mit einem Arbeitsschema bestehend aus den SDS `allgemein` und `uni`
- dito mit einer selbstgewählten Kombination von SDS

Literatur

[PCTE90] Portable Common Tool Environment - Abstract Specification (Standard ECMA-149); European Computer Manufacturers Association, Geneva; 1990

[PCTE94] Proceedings of the PCTE '94 Conference, San Francisco, 29.11.-1.12.1994; 1994/11

[DBI] Kelter, U.: Lehrmodul "Einführung in Datenbanksysteme"; 1999/10

[HINS] Kelter, U.: Lehrmodul "Einführung in die Benutzung des H-PCTE-Systems"; 1999/12

[IRA] Kelter, U.: Lehrmodul "Integrationsrahmen für Software-Entwicklungsumgebungen"; 1999/11

[PDDL] Kelter, U.: Lehrmodul "Definition und Manipulation von Schemata"; 1999/11

[PHIS] Kelter, U.: Lehrmodul "Einordnung und Historie von PCTE"; 1999/11

[PMDB] Kelter, U.: Lehrmodul "Die Metadatenbank und das Typverwaltungs-API von PCTE"; 1999/11

[PRF] Kelter, U.: Lehrmodul "Referenzen"; 1999/12

[PSV] Kelter, U.: Lehrmodul "Segmentierung und Verteilung"; 1999/11

Index

