Institut für Informatik
Universität Leipzig
Augustusplatz 10-11
04109 Leipzig

UNIVERSITÄT LEIPZIG
Institut für Informatik
- Abteilung DATENBANKEN -

Seminararbeit

Objektorientierte Schema-Evolution

Seminar „Schema Evolution“ (WS 05/06)

Autor: Torsten Hein (Mat-Nr. 9440522), Master-Informatik (3. FS)

Betreuer: Andreas Thor

Seminarleitung: Prof. Dr. Erhard Rahm

Eingereicht am: 08.02.2006

Inhaltsverzeichnis

1 Einleitung

In aktuellen IT-Projekten werden Entwicklungsprozesse immer häufiger mittels Agiler Methoden wie XP (Extreme Programming) oder Scrum umgesetzt. Diese Techniken werden mittlerweile von zahlreichen Modellen und Werkzeugen unterstützt, wobei sie vor allem bei der Konzeption und Softwareentwicklung eingesetzt werden. Evolutionäre Datenbankänderungen sind bisher noch nicht im adäquaten Maße möglich.

Für die Datenbankentwicklung gilt weiterhin häufig noch der klassische Ansatz, dass auf Basis des konzeptuellen systemunabhängigen Entwurfs das logische Modell entwickelt wird. Dem konzeptuellen Entwurf wird traditionell besonders große Bedeutung beigemessen. Die Anforderungen sollen möglichst vollständig erfasst und beschrieben werden, dabei wird von einem statischen Modell ausgegangen, das keine Änderung vorsieht. Veränderungen der Anforderungen können so nur durch Neuentwicklung der Datenbank berücksichtigt werden. Der logische Modellierungsschritt ist dann auf das entsprechende Datenbankmodell angepasst. Damit wird die Kategorisierung des Datenbankmanagementsystems (DBMS) vorgenommen wie beispielsweise hierarchisches Datenmodell, Netzwerkdatenmodell, relationales Datenmodell oder objektorientiertes Datenmodell.

Die größte Verbreitung haben traditionell und auch derzeit noch sicherlich die relationalen DBMS. Sie eigenen sich besonders gut für große Datenmengen von einfacher Struktur wie sie in den klassischen IT-Bereichen wie Bank-, Versicherungswesen oder Rechungs- und Buchungswesen auftreten. Eine Untersuchung der aktuellen Datenbanksysteme von IBM, Oracle und Microsoft erfolgte im Rahmen der Seminararbeit „Schema Evolution in kommerziellen DBS (dynamische Schema-Evolution)“ von Jáno Gebelein, die ebenfalls in dieser Seminarreihe „Schema Evolution“ entstand.

In den letzten Jahren hat sich der Einsatz von Datenbanksystemen auch in andere Bereiche ausgedehnt. Vor allem im ingenieurstechnischen Bereich (CAD) oder wo ansonsten komplexe Daten anfallen, kommen verstärkt objektorientierte DBMS

zur Verwendung. Ein weiterer Vorteil ist die direkte Unterstützung aktueller Entwicklungsmodelle, die überwiegend objektorientiert angelegt sind.

Ein Datenbankmodell eines Ausschnittes der realen Welt, auch als Diskurswelt bezeichnet, beschreibt die Struktur und in objektorientierten DBMS auch das Verhalten der Datenobjekte und bildet das Datenbankschema. Dieses Schema wird während des Entwurfs erstellt und wird vom DBMS verwendet, um eine Datenbank anzulegen.

Änderungen in der Diskurswelt oder wechselnde Anforderungen können Anpassungen des Schemas während des Betriebs bedingen. Dies entspricht einer Entwicklung der Datenstruktur, wobei dieser Prozess als Schemaevolution bezeichnet wird. Gründe für die Notwendigkeit einer Schemaevolution könnten sein:

- *Unvorhersehbare Änderungen in der Diskurswelt*: Die Datenbasis umfasst typischerweise Informationen, die über längere Zeiträume gesammelt wurden und überdauert häufig mehrere Generationen von verwendeten Applikationen. Da die Diskurswelt nicht immer dieser langfristigen Konstanz unterliegt, ergeben sich daraus zahlreiche Änderungsanforderungen für das Datenbankschema.

- *Fehler beim komplexen Entwurf des Datenmodells*: Da große IT-Projekte häufig komplex strukturierte Daten handhaben müssen, kann es beim Entwurf zu Fehlern kommen, die im späteren Verlauf zu Änderungen des Datenbankschemas führen können.

- *Parallele Existenz verschiedener Perspektiven der Diskurswelt*: Da auf die Datenbanken von unterschiedlichen Benutzern und über verschiedene Applikationen zugegriffen wird, kann nicht angenommen werden, dass ein Schema sämtliche Anforderungen erfüllt.

- *Angleichung und Integration weiterer Systeme*: Durch Fusionen und Übernahmen kann die Integration und Harmonisierung vormals unabhängiger Daten notwendig werden. Diese Konsolidierung kann über Schemaänderungen erfolgen.

Aktuell verfügbare Datenbanksysteme unterstützen Schemaevolutionsprozesse nur in Ansätzen und noch sehr unzureichend. Der überwiegende Anteil der Konzepte zur Unterstützung von Schema Evolution für Datenbanksysteme geht auf die 80er und 90er Jahre zurück, wobei die Ideen vor allem im Umfeld der objektorientierten Datenbanksysteme entwickelt wurden.

Im Rahmen dieser Arbeit sollen nun vor allem die Ansätze dieser objektorientierten Schema-Evolution untersucht und vorgestellt werden. Dafür werden unterschiedliche Konzepte analysiert und bewertet, wobei besonders auf die Schemaevolution mittels *Versionierungskonzepten* und *Views* eingegangen wird.

2 Theoretische Grundlagen

Schema Evolution verleiht Datenbanksystemen die Fähigkeit, auf Veränderungen äußerer Anforderungen zu reagieren. Das Datenmodell (Schema) kann sich zeitlich weiterentwickeln und verändern. Dabei besteht die Möglichkeit, die alten Zustände des Datenbankschemas zu bewahren und jederzeit auf sie zurückgreifen zu können. Damit bildet ein Schema eine Art Schnappschuss eines speziellen Datenbankmodells. Für die Anwendungsentwickler sollten die Schemaänderungen transparent sein, damit Applikationen nicht zwangsläufig jeder Schema Evolution folgen müssen.

Weiterhin darf bei der Änderung eines Datenschemas die Konsistenz der Datenbasis nicht beeinträchtigt werden. Wichtig ist, die Dauer für die Änderungen möglichst gering zu halten, bestenfalls sollten die Modifikationen im laufenden Betrieb erfolgen, d.h. das System muss nicht *offline* gehen.

Für relationale Datenbanksysteme ist das von E. F. Codd entwickelte Datenmodell allgemein bekannt und akzeptiert. Dies gilt nicht für den objektorientierten Bereich. Dort existieren mehrere objektorientierte Modelle und verschiedene Modellierungskonzepte. Daher werden im Folgenden grundlegende Elemente objektorientierter Datenbankmanagementsysteme (ooDBMS) erläutert.

2.1 Besonderheiten objektorientierter Datenbanksysteme

Objekte stellen den zentralen Teil eines objektorientierten Systems dar. Sie repräsentieren die Konzepte der zu modellierenden realen Welt. Die Datenbasis ist also nicht satzorientiert wie in relationalen Datenbanksystemen. Damit bieten sich ooDBMS häufig für die Speicherung komplexer Datenstrukturen, wie sie beispielsweise in CAD-Systemen anfallen, an. Die Hauptparadigmen der objektorientierten Programmierung gelten auch für ooDBMS. Der entscheidende Unterschied besteht in der Persistenz der Objekte. Probleme bei der Schema Evolution entstehen durch die Möglichkeit zur Vererbung, denn es muss genau bedacht werden, ob sich Änderung auch auf abgeleitete Unterklassen auswirken sollen. Dadurch sind dynamische Änderung der Zugehörigkeit von Objekten zu Klassen sowie Änderungen der Klassendefinitionen häufig nur mit Einschränkungen möglich. Die Abgrenzung der ooDBMS gegenüber evolutionärer

Softwareentwicklung ohne Persistenzunterstützung besteht in den Abhängigkeiten zu anderen Teilen des Schemas, zu bereits bestehenden Daten oder zu existierenden Applikationen.

2.2 Operationen bei der Schema-Evolution

Für eine Schemaänderung sind verschiedene Operationen notwendig. Allgemein können folgende fundamentale Ziele zusammengefasst werden:

- jederzeitige Durchführung beliebiger Schemaänderungen und Sicherstellung der Konsistenz durch korrigierende Maßnahmen
- fortlaufende Ausführbarkeit vorhandener Applikationen ohne Anpassung
- fortlaufende Zugreifbarkeit vorhandener Objekte der Datenbank mit der Möglichkeit zur Kooperation von Applikationen verschiedener Schemazustände
- effiziente Realisierung evolutionärer Schemaänderungen und unterbrechungsfreier Betrieb insbesondere auch bei großen Datenmengen

Mittels Schemaänderungsprimitiven sollen diese Ziele erreicht werden. Eine Taxonomie dafür gilt als vollständig, wenn sich jedes beliebige Ausgangsschema durch Anwendung der Schemaänderungsprimitiven in jedes beliebige Zielschema überführen lässt.

Eine der ersten Taxonomien wurde vom *Department of Computer Sciences* der University of Texas in [BCG+87] für das Datenbanksystem ORION verfasst, dass die elementaren Elemente enthält:

- Modifikation einer Klasse
  - Änderungen an einem Attribut
    - Hinzufügen
    - Löschen
    - Ändern des Namens
    - Ändern des Wertebereichs

        - Ändern der Oberklasse (bei Mehrfachvererbung und hierdurch entstehenden Attributkonflikten)
        - Ändern des Default-Wertes
        - Ändern des Speicherverhalten eines Attributes (shared values)
        - Ändern der Referenzeigenschaft eines Attributes

    - Änderungen an einer Methode
        - Hinzufügen
        - Löschen
        - Ändern des Namens
        - Ändern der Implementierung
        - Ändern der Oberklasse (bei Mehrfachvererbung und hierdurch mehrfach geerbten Attributen)

- Modifikation der Klassenhierarchie

    - Hinzufügen einer Klasse zu der Liste der Oberklassen einer anderen

    - Entfernen einer Klasse aus der Liste der Oberklassen einer anderen

    - Ändern der Reihenfolge der Liste der Oberklassen

- Modifikation der Klassenmenge

    - Hinzufügen einer Klasse

    - Entfernen einer Klasse

    - Ändern des Namens einer Klasse

Bei allen Schemaänderungen ist die Erhaltung der Konsistenz von großer Bedeutung: Dafür werden sogenannte Schemainvarianten definiert, die eine Art formale Verifikationen der Änderungen darstellen und in die Bereiche *Eindeutigkeit*, *Vollständigkeit*, *Vererbungsbeziehungen* und *Verhalten* kategorisiert werden können.

2.3 Invarianten bei der Schema-Evolution

Die *Invarianten der Eindeutigkeit* stellen sicher, dass diese unverwechselbar angesprochen werden können. Dabei betreffen die Namen mehr die Sicht des Benutzers eines Datenbanksystems, während die Identitäten intern Verwendung finden.

Die *Invariante der Vollständigkeit* garantiert, dass alle benötigten Komponenten eines Schemas vorhanden sind. Dabei unterscheiden wir die für die Beschreibung eines Datenbankzustandes notwendige strukturelle Notwendigkeit von der für die Ausführung von Methoden benötigten verhaltensmäßigen Notwendigkeit.

Invarianten der Vererbungsbeziehungen garantieren eine wohldefinierte Semantik zwischen den Klassen eines Schemas. Dabei sind insbesondere zweierlei Konflikte zu berücksichtigen. Zum einen können sich Überschneidungen zwischen den lokal definierten und den von ihren Oberklassen geerbten Eigenschaften ergeben (Überschreibungen), zum anderen können sich Diskrepanzen zwischen den Eigenschaften mehrerer direkter Oberklassen ergeben (typischer Fall eines Konfliktes bei Mehrfachvererbung).

Invarianten bezüglich des Verhaltens regeln das Überladen und das Überschreiben von Methoden.

In [BCG+87] werden folgende konkrete Invarianten definiert:

Invariante des Klassenverbands (Class Lattice Invariant)

Der Klassenverband ist ein verbundener, gerichteter, azyklischer Graph mit eindeutig markierten Knoten und Kanten. Der Graph hat genau einen Wurzelknoten, eine vom System definierte Klasse *OBJECT*, von der aus jeder Knoten erreichbar ist.

Invariante der Eindeutigkeit der Namen (Distinct Name Invariant)

Alle Attribute einer Klasse, ob vererbt oder in der Klasse definiert, haben unterschiedliche Namen. Ebenso alle Methoden.

Invariante der Eindeutigkeit der Identität (Distinct Identity (Origin) Invariant)

Alle Attribute und Methoden einer Klasse, die durch Mehrfachvererbung doppelt erscheinen, werden nur von einer Oberklasse geerbt.

Invariante vollständiger Vererbung (Full Inheritance Invariant)

Eine Klasse erbt von ihren Oberklassen alle Attribute und alle Methoden. Ausnahme ist der Fall der Erbung zweier gleichnamiger Attribute oder Methoden egal ob gleicher oder unterschiedlicher Herkunft. Hier wird nur jeweils ein Merkmal vererbt.

Invariante kompatibler Wertebereiche (Domain Compatibility Invariant)

Wertebereiche von Attributen können bei der Vererbung eingeschränkt, nicht jedoch verallgemeinert werden.

2.4 Regeln zur Implementierung der Operationen bei der Schema-Evolution

Die in 2.2 aufgeführten Operationen dürfen ein Datenbankschema nicht derart verändern, dass es zu einer Verletzung der Invarianten kommt. Dafür werden in [BCG+87] 12 verschiede Implementierungsregel formuliert, die in unterschiedliche Bereiche unterteilt sind.

Standard-Konfliktauflöse-Regeln (Default Conflict Resolution Rules)

(1) Namensgleiche Attribute oder Methoden einer Klasse überschreiben diejenigen aus Oberklassen.

(2) Bei namensgleichen geerbten Attributen oder Methoden, die in unterschiedlichen Klassen definiert wurden, hat das Attribut oder die Methode aus derjenigen Oberklasse Vorrang, die zuerst in der Liste der Oberklassen erscheint.

(3) Bei Mehrfacherbung eines Attributes gleicher Herkunft, jedoch mit unterschiedlich eingeschränktem Wertebereich, wird dasjenige mit dem kleinsten Wertebereich geerbt. Sind jedoch die Wertebereiche gleich oder nicht vergleichbar (einer ist nicht Obermenge des anderen), so wird das Attribut derjenigen Oberklasse vererbt, die zuerst in der Liste der Oberklassen erscheint.

Regeln zur Eigenschaftsvererbung und Änderung (Property Propagation and Change Rules)

(4) Werden Eigenschaften (Name, Wertebereich, Default-Wert, Speicherart, Composite-Link) eines geerbten Attributes geändert, so werden diese neuen Eigenschaften weitervererbt.

(5) Neu hinzugefügte oder geänderte Attribute, werden nur dann weitervererbt, wenn dies konfliktfrei geschehen kann.

(6) (Domain Change Rule) Wertebereiche von Attributen können bei der Vererbung nur generalisiert werden.

DAG Manipulationsregeln (DAG Manipulation Rules)

(7) (Edge Addition Rule) Bei Hinzuname einer Klasse als Oberklasse, wird diese an das Ende der Liste der Oberklassen angefügt.

(8) (Edge Removal Rule) Bei Entfernen der letzten Oberklasse A aus der Oberklassenliste einer Klasse B, werden alle Oberklassen der Klasse A direkte Oberklassen von Klasse B.

(9) (Node Additon Rule) Wird bei Hinzufügen einer Klasse zu dem Schema keine Oberklasse angegeben, so wird die Wurzelklasse OBJECT automatisch einziger Vater.

(10) (Node Removal Rule) Beim Löschen einer Klasse wird für jede Unterklasse zum Löschen der Kante Regel 8 angewendet.

Regeln für komplexe Objekte (Composite Object Rules)

(11) (Composite Link Rule) Die Composite-Link Eigenschaft kann von einem Attribut entfernt, nicht jedoch hinzugefügt werden.

(12) Das Entfernen der Composite-Link Eigenschaft von einem Attribut löst die IS-PARTOF-Beziehung, so dass die Lebenszeit des referenzierten Objekts nun nicht mehr von dem referenzierenden Objekt abhängt.

In [BCG+87] werden weiterhin alle Schemaänderungen, die von ORION unterstützt werden, beispielhaft erklärt.

2.5 Vollständigkeit und Korrektheit bei der Schema-Evolution

Ein entscheidender Faktor bei der Schema-Evolution ist die Erfüllung der beiden Bedingungen *Vollständigkeit* und *Korrektheit* (siehe [BCG+87] → („Completeness and Correctness“). Diese sind wie folgt definiert:

- **Korrektheit**: Die Anwendung eines Operators auf ein korrektes Schema muss wiederum zu einem korrekten Schema führen.
- **Vollständigkeit**: Ein beliebiges Schema muss durch sukzessive Anwendung der Operationen jedes beliebige andere Schema transformierbar sein.

Grundsätzlich wird durch [BCG+87] eine nahezu vollständige formale Beschreibung für die Schema-Evolution in objektorientierten Datenbanken gegeben.

3 Objektorientierte Schemaevolution

In diesem Kapitel wird Schemaevolution mittels *Versionierung* bzw. über *Object Views* betrachtet.

3.1 Versionierungskonzepte für die objektorientierte Schemaevolution

Der Begriff der *Version* eines Objektes bezeichnet einen Zustand, in dem sich das Objekt befindet, eine Ausprägung des Objektes oder auch eine Variante desselben. Änderungsoperationen, die sich als unvorteilhaft herausstellen, sollten ein Rollback unterstützen und auch Applikationen sollten auf unabhängige Schemaversionen zugreifen können. Damit existieren unabhängige Schemata die durch Ableitungsbeziehungen verbunden sind. Eine Ableitungsbeziehung von einer Version ov_1 zu einer Version ov_2 drückt aus, dass ov_2 von ov_1 abgeleitet wurde. Anstatt eine anstehende Änderung direkt auf der Objektversion ov_1 durchzuführen und deren vorherigen Zustand zu überschreiben, bleibt ov_1 unverändert erhalten. Stattdessen wird bei der Ableitung zunächst eine Kopie unter der Beziehung ov_2 angelegt und nur diese Kopie wird verändert. Damit existieren nun zwei Versionen, die den Zustand des Objektes vor und nach der Veränderung repräsentieren und die durch die Ableitungsbeziehung verbunden sind. Damit kann die Veränderung durch Rüchsetzen auf ov_1 rückgängig gemacht werden, wenn dies erwünscht ist. Im Allgemeinen kann ein versioniertes Objekt natürlich aus beliebig vielen Versionen bestehen.

Je nach Grad der Ordnung entstehen unterschiedliche Ableitungsgraphen. Bei *Vollordnung* (Abbildung 1a) existiert genau eine neuste Version und nur diese darf geändert werden. Bei *Halbordnung* (Abbildung 1b) bestehen Alternativen da auch ältere Versionen modifiziert werden dürfen. Dadurch entstehen mehrere logische neuste Versionen. Eine weitere Verallgemeinerung wird durch *Verschmelzung* (Abbildung 1c) erreicht. Dabei können gewünschte Elemente von zwei oder mehr Versionen zu einer neuen Version zusammengeführt werden.

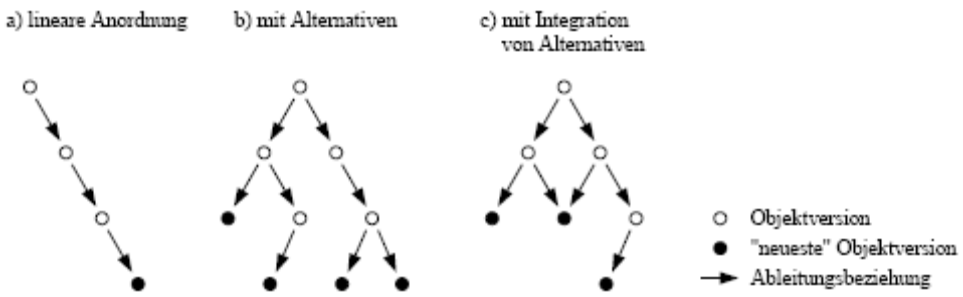

Abbildung 1: Arten von Ableitungsbeziehungen zw. Versionen eines Objektes

Bei der Auswahl der bei einem Zugriff zu benutzenden Objektversionen wird zwischen statischem und dynamischen Binden unterschieden. Beim statischen Binden wählt die zugreifende Applikation bereits zur Übersetzungszeit explizit eine bestimmte Version anhand eines Versionsidentifikators fest aus. Dagegen findet bei dynamischem Binden keine explizite Auswahl der zuzugreifenden Version statt. Die Auswahl geschieht stattdessen für alle unspezifisch zugreifenden Applikationen auf dieselbe Art und Weise nach einem dynamischen Kriterium. Dies kann beispielsweise immer die neuste Version sein.

Durch das Versionierungskonzept liegen nun beliebig viele Objekte der Datenbank in mehreren Versionen vor. Aus der damit entstehenden kombinatorischen Vielfalt beschriebener Zustände der Diskurswelt ergeben sich im Allgemeinen Inkonsistenzen. Da Konsistenz aber wie bereits herausgearbeitet ein entscheidendes Kriterium darstellt, müssen Mechanismen eingeführt werden, die die Konsistenz wahren. Solche Mechanismen werden unter der Bezeichnung Konfigurationsverwaltung zusammengefasst. Eine Konfiguration selektiert für eine Menge von Objekten untereinander konsistente Versionen. Damit reduziert sich das Problem der Auswahl konsistenter Objektversionen auf die Auswahl konsistenter Konfigurationen.

Bei der Schemaevolution mittels Versionierung wird allgemein zwischen drei verschiedenen Ansätzen unterschieden:

- **restriktiver (konvertierender) Ansatz**: Die Änderungen sind irreversible. Somit handelt es sich um keine ‚wirkliche' Evolution im Sinne von Version Management.
- **historischer Ansatz**: Jede bedeutende Änderung bedingt eine neue Schema Version. Somit ist jede Version mit ‚seinen' entsprechenden Daten verbunden („eingefroren") und in einem separatem Speicherbereich abgelegt. Damit ist die Anzahl der Schemata gleich der Anzahl der Datenbanken.
- **paralleler Ansatz**: Es gibt unterschiedliche Schemeta in einer gemeinsamen Zone, d.h. alle Schemata verwenden dieselbe Datenbasis. Somit gibt es eine Datenbank mit einer unbestimmten Anzahl von Schemata.

Vor allem der parallele Ansatz bringt entscheidende Vorteile, da mit unterschiedlichen Schemata auf der gleichen Datenbasis gearbeitet wird. Dieser Ansatz wird auch in [ALP91] untersucht und vorgestellt. Dabei werden vier verschiedene Änderungseinheiten unterschieden, nämlich Versionen von:

- Schemata
- Klassen
- Objekten mit Schema
- Views

Grundsätzlich ist eine Version ein stabiler und kohärenter Zustand, den ein Administrator oder Datenbankdesigner dauerhaft erhalten möchte. In [ALP91] werden zwei Typen von Versionen unterschieden: *working* und *stable* Versions. Änderungen können dabei immer nur an einer working Version vorgenommen werden, d.h. eine stable Version kann weder gelöscht noch geupdatet werden. Stable Version müssen immer erst in working Version transformiert werden. Queries können dabei sowohl auf working als auch auf stable Version angewendet

werden. Weiterhin gibt es eine default Version, die verwendet wird, wenn vom User kein konkretes Schema angegeben wird. Dieses kann individuell festgelegt werden.

In [ALP91] wird für jede Version ein *generic content* angelegt. Dieser setzt sich folgendermaßen zusammen:

```
[id, name, first_version, default, [working_versions],
[stable versions], next_version, root_class]
```

Id ist der vom System berechnete Identifier, wohingegen *name* der externe Name ist, der vom Administrator vergeben wird.

Eine Version wird hingegen wie folgt beschrieben:

```
[id, gen_id, name, number, [successors], [previous], date,
state, [[nodes], [edges]]]
```

Dabei ist *id* der intern berechnete Identifier und *gen_id* der Identifier des dazugehörigen *generic content*.

Bei der Datenbankänderung können nun folgende Operationen unterschieden werden:

- Veränderung des Datenbankschemas
  - Hinzufügen bzw. Unterdrücken einer Version
- Veränderung einer Version
  - Veränderung des Graph-Verlaufes
    - Hinzufügen bzw. Unterdrücken eines Knotens
    - Hinzufügen bzw. Unterdrücken einer Kante
  - Veränderung des Graph-Kontexts
    - Veränderung eines Knotens
    - Veränderung einer Kante

Zu all diesen Operationen gibt es in [ALP91] insgesamt 7 Transformationsregeln, die nicht verletzt werden dürfen.

Zusammenfassend bleibt zu sagen, dass in [ALP91] ein *Version Model* beschrieben wurde, welches die Schema-Evolution einer Datenbank unterstützt. Die Versionsverwaltung basiert dabei auf der Beschreibung von *Contents*, der eine Art Erweiterung zum View-Konzept darstellt. Transformationen werden dabei auf Teilen des Datenbankschemas angewendet, wobei eine unbestimmte Anzahl von *Contents* für die Datenbank festgelegt werden kann. Jeder *Content* beschreibt dabei einen bestimmten Teil des Datenbankschemas.

Eine ausführliche Betrachtung des *Version Management* erfolgt in der Seminararbeit „Versionierung von Schemas“ von Christian Lehmann im Rahmen dieser Seminarveranstaltung.

Grundsätzlich gilt, dass die parallele Schema Versionisierung eine gewisse Unabhängigkeit von Daten und Applikationen ermöglicht. Eine bestehende Applikation kann weiterhin auf einem bekannten Schema arbeiten, auch wenn es zwischenzeitlich zu Veränderungen gekommen ist. Somit entfallen aufwendige Softwareänderungen.

Weiterhin wird der sogenannte *Object Life Cycle* unterstützt, d.h. die verschiedenen Phasen eines Objektes werden mit den dazugehörigen Integritätsbedingungen erfasst, beschrieben und persistent verwaltet.

3.2 Objektorientierte Schemaevolution mittels Object Views

Leider gibt es im Gegensatz zu rDBMS für ooDBMS kein einheitliches View-Konzept. Vielmehr existieren unterschiedliche Ansätze oder auch überhaupt keine Unterstützung.

Wenn ein objektorientiertes Datenbanksystem einen View-Mechanismus anbietet, so bestehen weitere Möglichkeiten der Schemaevolution. Die Schemaänderungen müssen nicht mehr am Basisschema durchgeführt werden, sondern können lediglich auf den Views, die auf das Basisschema aufsetzen, definiert werden. Wenn alle Applikationen auf Sichten aufsetzen und nicht direkt auf das darunterliegende Basisschema zurückgreifen, so können Schemaänderungen gleichsam simuliert werden. Damit wären idealerweise keine Änderungen des Basisschemas notwendig und es würden lediglich die Views modifiziert werden.

Grundsätzlich unterteilt man Schemaänderungen häufig in folgende Arten [AH88]:

- capacity preserving
- capacity reducing
- capacity augmenting

Transformationen können notwendig werden, wenn sich die abgebildete reale Welt ändert – eventuell müssen neue oder zusätzliche Daten erfasst werden. Die Transformationen basieren auf den Konzepten von unabhängigen Views als virtuelle Klassen und Subschemata also Subsets von Klassen und Views. Damit lassen sich nach [TS93] *capacity preserving* und *capacity reducing* Transformationen durchführen. Diese erfolgen in zwei Schritten:

- **Schritt 1**: Erweiterung des bestehenden Schemas durch ein Set von Views, das die veränderte Situation entsprechend simuliert. Integration der Views an den entsprechenden Stellen im Schema.
- **Schritt 2**: Definition eines Subschemas des erweiterten Schemas mittels Auswahl der entsprechenden Klassen und Views, die Struktur des neuen Subschemas beschreiben. Subschema als vollständig deklarieren.

Bei [TS93] sind keine *capacity augmenting* Transformationen möglich. Das View-Konzept der objektorientierten Datenbanken unterstützt diese Art von Änderungsoperationen nicht. In [RR97] wird ein Konzept beschrieben, welches das bestehende View-Konzept erweitert. Das vorgestellte *Transparent Schema-Evolution System (TSE)* erlaubt auch *capacity augmenting* Transformationen.

3.2.1 Capacity Preserving Transformations

Darunter werden Transformationen zusammengefasst, die keine Änderungen am Umfang der Datenbasis vornehmen, d.h. die Anzahl der Klassen ändert sich nicht. Dazu zählen Operationen wie:

- Umbenennung von Klassen und Funktionen
- Änderung der Klassenhierarchie

Derartige Änderungen können anfallen, wenn beispielsweise für Applikationen Schnittstellen nach außen bekannt gemacht werden sollen und dabei eine veränderte Nomenklatur gewünscht wird. Im folgenden Beispiel aus [TS93] soll die Änderung der Klassenhierarchie beschrieben werden.

Die Ausgangssituation ist wie folgt:

Die Klasse *person P* ist die Oberklasse von *youngsters Y* und *workers W*, dagestellt mittels weißer Kreise. Dabei sind *youngsters* jünger als 30 Jahre und *workers* jünger als 65. Da die Gruppe der *youngsters* im Prinzip eine Untermenge von *workers* bildet (age<30 $\Rightarrow$ age<65), kann in der Klassenhierarchie *youngsters* auch als Unterklasse von *workers* angesehen werden. Ein entsprechender View wäre wie folgt definiert:

```
define view Y' as select [age<30] (W)
```

Damit bildet *Y'* die Unterklasse von W. Das resultierende neue Subschema ist in Abbildung 2 grau hinterlegt.

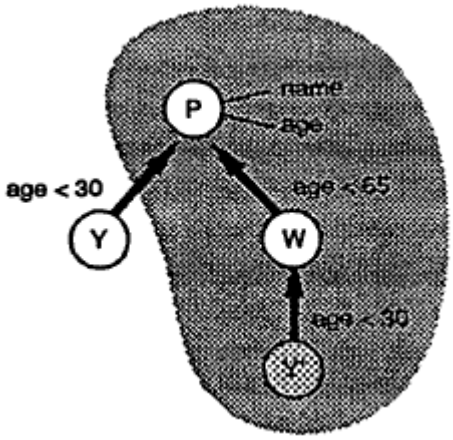

Abbildung 2: Beispiel für eine capacity preserving Transformation aus [TS93]

3.2.2 Capacity Reducing Transformation

Wenn Transformationen die Anzahl der Klassen verringern, werden sie als *capacity reducing* Transformations bezeichnet. Darunter fallen Transformationen, die ein objekt-orientiertes Modell in ein werte-orientiertes Modell umwandeln. Dies kann günstig sein, wenn strukturierte Informationen in vereinfachter, kompakterer Form zusammengefasst werden sollen.

In Abbildung 3 ist folgende Ausgangssituation (weiße Kreise) dargestellt:

Es gibt die Klassen *person P* und *address A*. Zu jedem *P* gehört ein *A*, das die Attribute *city*, *street*, *no* besitzt. Um von Person auf die indirekten Attribute zugreifen zu können, ist eine *unnest*-Operation notwendig. Eine *capacity reducing* Transformation wäre die Umwandlung der indirekten Attribute in direkte Attribute von *P*. Ein entsprechender View lautet wie folgt:

```
define view P' as extend
   [city:=city(addr(p));
    street:=street(addr(p));
    no:=no(addr(p))](p:P);
```

Die object-to-value Transformation ist in Abbildung 3 dargestellt. Dabei bildet *P'* das neue Subschema.

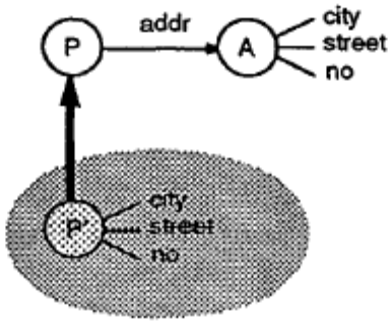

Abbildung 3: Beispiel für eine capacity reducing Transformation aus [TS93]

Eine umgekehrte *value-to-object* Transformation bedingt eine *capacity augmenting* Transformation.

3.2.3 capacity augmenting Transformations

Die herkömmlichen View-Konzept objektorientierter Datenbanksysteme unterstützen perse keine capacity augmenting Transformationen. Zur Realisierung müssen die Konzepte erweitert werden.

Bei TSE wird der View-Ansatz der ooDBMS um das MultiView-Konzept [RR97] erweitert. Damit können die in [RR97] definierten Anforderungen an das TSE-System realisieren werden:

- Die Schemaänderungen, die für ein View vorgenommen werden, dürfen bestehende Views nicht beeinflussen.
- Neu erstellte Views müssen updateable sein.
- Beide, alte und neue Version eines Schemas, müssen auf die gleichen persistenten Daten gemeinsam zugreifen können.
- Schemaänderungen müssen transparent für den Anwender sein
- Das View-Konzept muss neben *capacity preserving* und *capacity reducing* Transformationen auch *capacity augmenting* Transformationen unterstützen.

Als ooDBMS wurde GemStone verwendet. Die gesamte Architektur des TSE-Systems ist in Abbildung 4 dargestellt.

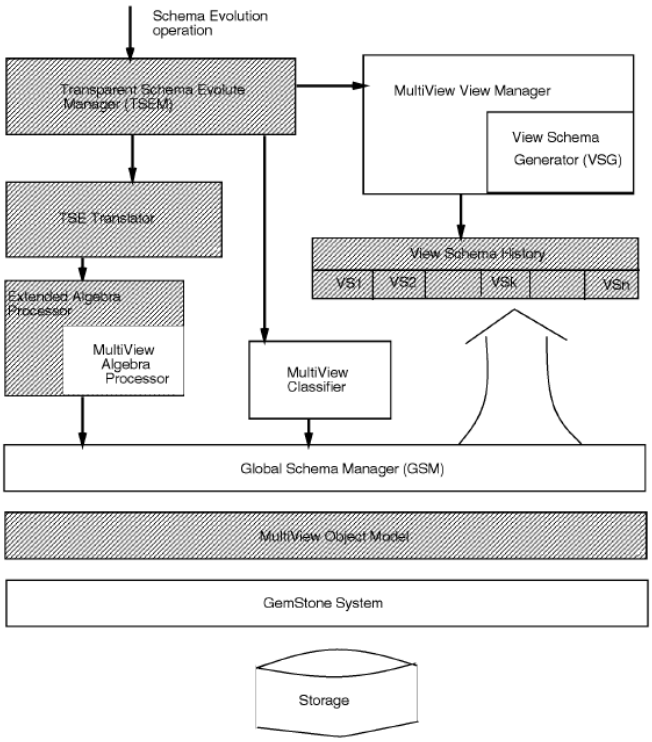

Abbildung 4: TSE-System Architektur aus [RR97]

In Abbildung 5 wird das Anlegen eines neuen Schemas im TSE-System dargestellt. Nach der Schemaerweiterung können zwei User bzw. Applikationen auf unterschiedliche Schemata zugreifen, wobei die Datenbasis jeweils identisch ist.

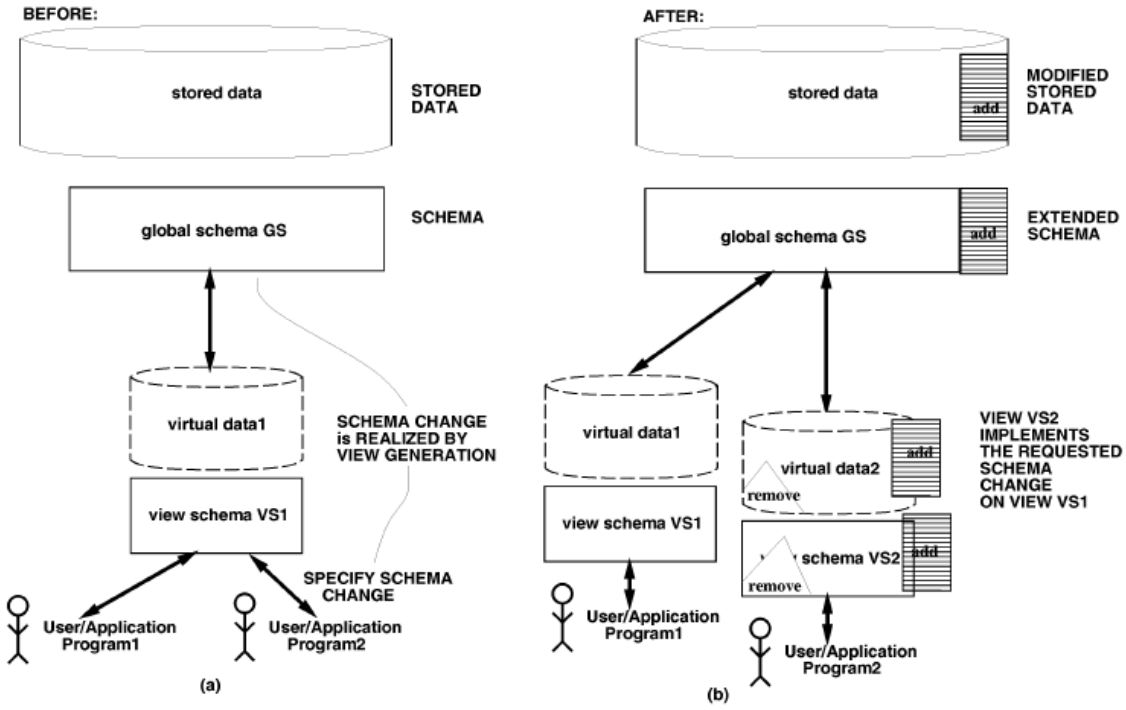

Abbildung 5: Anlegen eines neuen Schemas im TSE-System aus [RR97]

Ein konkretes Beispiel einer Schemaänderung wird in Abbildung 6 gegeben. Dabei wird ein zusätzliches Attribut eingefügt.

Das Schema könnte die Datenbank einer Universität darstellen. Angenommen mit dem ursprünglichen Schema in (a) arbeiten mehrere Applikationen. Nun wird aus verwaltungstechnischer Sicht für jeden Studenten noch das Attribut *register* angelegt. Das TSE-System erlaubt es nun, dass die anderen Applikationen weiterhin das ursprüngliche Schema verwenden können, welches parallel zum neu angelegten Schema weiter besteht. Dabei ist die Datenbasis dieselbe.

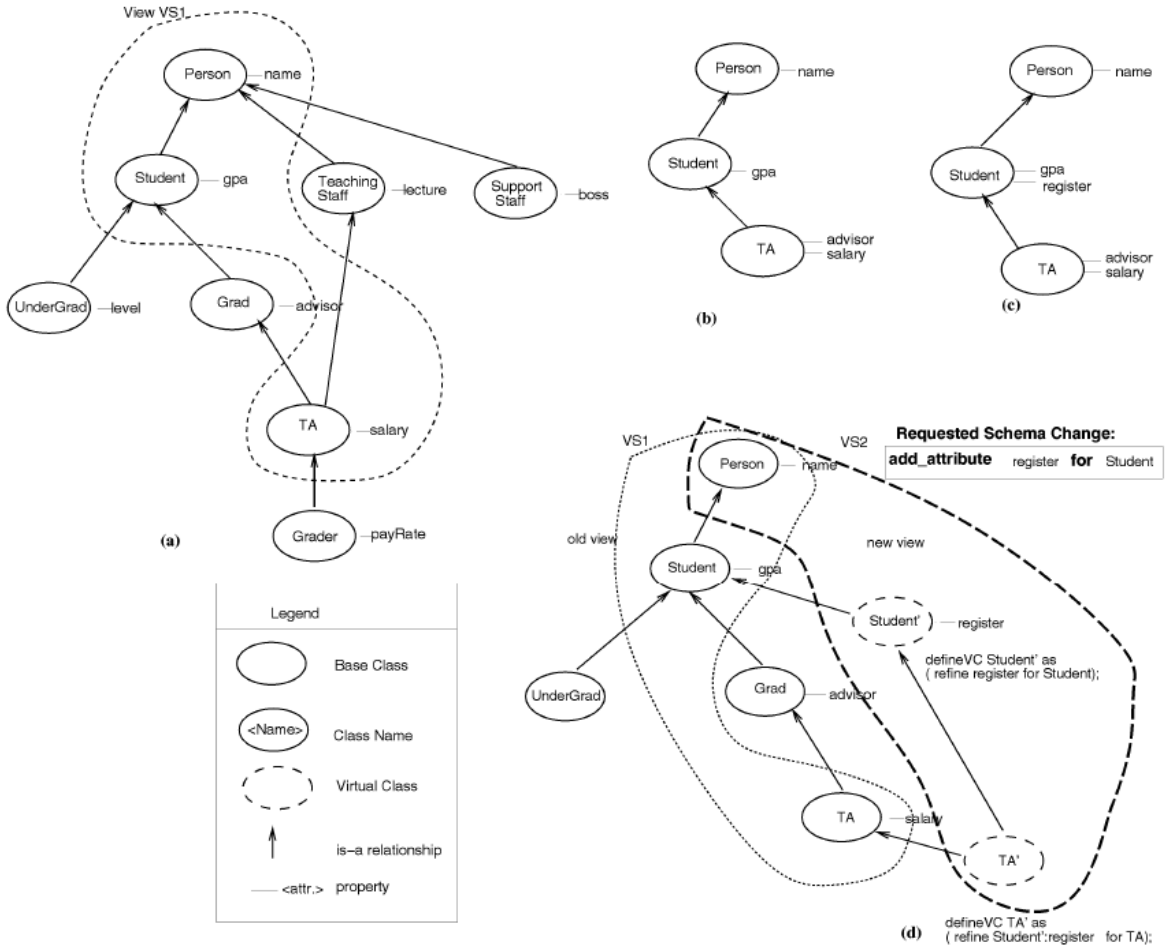

Abbildung 6: Hinzufügen eines Attributes im TSE-System aus [RR97]

Allgemein werden in [RR97] drei Vorteile des TSE-Systems gegenüber konquentionellen ooDBMS benannt:

(1) it allows all instances to be directly shared by each schema,

(2) it integrates the restructring power of views and schema evolution and

(3) it simplifies version merging.

Der Vorteil des Instance-Sharings wird an einem konkreten Beispiel beschrieben. Die beiden anderen Punkte können in [RR97] nachgelesen werden.

(1) Instance Sharing Among Schema Versions

Die verteilte Nutzung von Instanzen zwischen unterschiedlichen Schema Versionen ist in Abbildung 7 abgebildet.

Angenommen der Nachfolger einer Applikationen arbeitet mit einem veränderten Schema, dass zusätzliche Attribute aufweist. Mittels TSE-System ist es jetzt möglich, dass neu angelegte Instanzen des erweiterten Schemas weiterhin vom vorherigen Tool verwendet werden können und selbst der Schreibzugriff – natürlich nur auf die ursprünglichen Attribute – ist möglich.

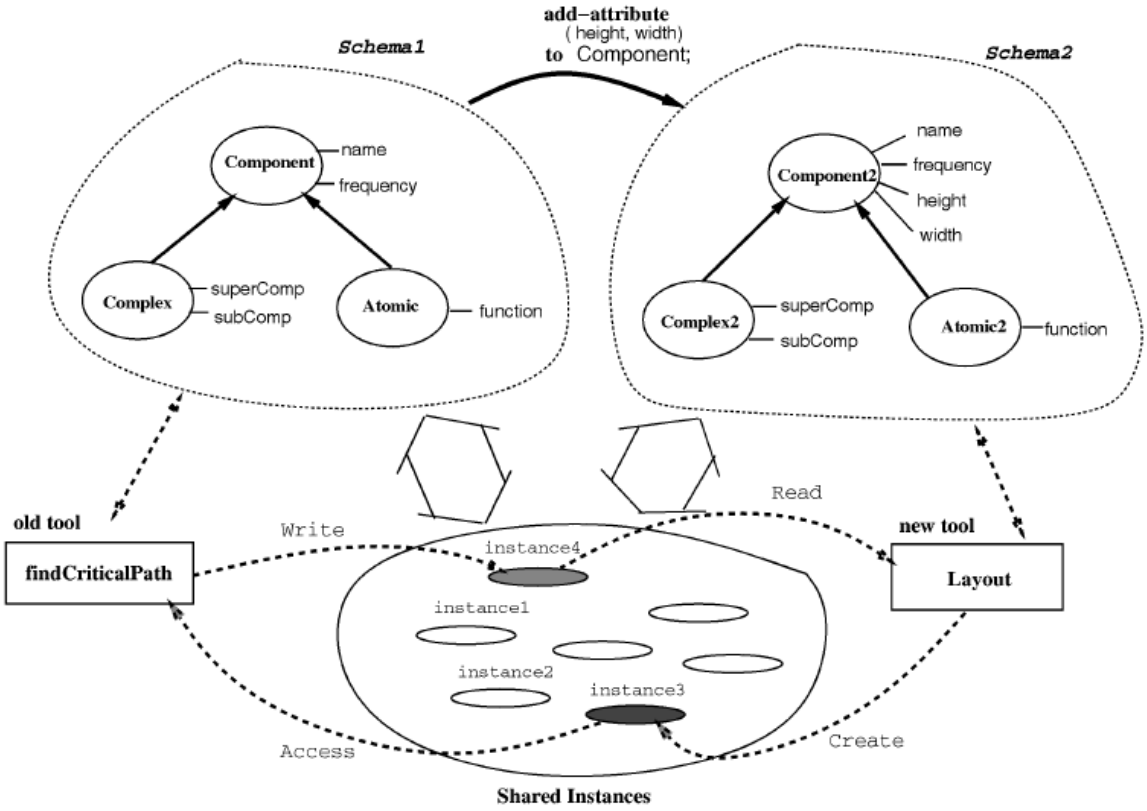

Abbildung 7: Verteilte Instanzennutzung unterschiedlicher Schema Versionen

Anhand der gegebenen Beispiele zeigen sich die Vorteile und Möglichkeiten bei der Verwendung des TSE-Systems. Weitergehende detaillierte Ausführungen können in [RR97] nachgelesen werden.

3.3 Toolgestütze Schema-Evolution

Da die Änderungen komplexer Schemata sehr aufwendig und umfangreich sein können, wird versucht dem Administrator durch Softwaretools bei dieser Aufgabenstellung zu unterstützen. Dabei gibt es einen ausführlichen Paper der University of Pittsburgh, wo ihr Softwaretool OTGen beschrieben wird [LH90].

OTGen (Object Transformer Generator)

OTGen ist eine Weiterentwicklung von TransformGen. Damit ist die automatisierte Code-Erzeugung bei Strukturänderungen in objektorientierten Datenbanken möglich. OTGen unterstützte dabei folgende Änderungsoperationen [LH90]:

- Hinzufügen einer Variablen in einer Klasse
- Löschen einer Variablen in einer Klasse
- Umbenennen einer Variablen in einer Klasse
- Ändern des Typs einer Variablen in einer Klasse
- Hinzufügen einer Klasse in die Liste der Oberklassen einer Klasse
- Löschen einer Klasse in der Liste der Oberklassen einer Klasse
- Hinzufügen einer neuen Klasse
- Löschen einer Klasse
- Umbenennen einer Klasse

Dabei dürfen die folgenden Anforderungen nicht verletzt werden [LH90]:

- **Vollständigkeit**: Jedes Objekt muss in seine neue Version transformiert worden sein, bevor man darauf zugreifen kann.
- **Korrektheit**: Nach der Transformation muss jedes Objekt zu einer Klassen-definition in der neuen Version passen. Insbesondere muss der Wert einer Variablen zu dem Typ der Variablen in der entsprechenden Klassendefinition passen.
- **Erhaltung gemeinsamer Daten**: Wenn zwei Variablen vor der Transformation auf ein gemeinsames Objekt zeigten, und wenn:
  - keine der Variablen gelöscht wurde,
  - keine der Standardtransformationen der Variablen überschrieben wurde,
  - und der Typ des transformierten gemeinsamen Objekts zu d. Variablen passt, dann müssen sie auch nach der Transformation auf ein gemeinsames Objekt zeigen

Das Hinzufügen einer Variablen ist als Beispiel für eine Standardtransformation in Abbildung 8 dargestellt.

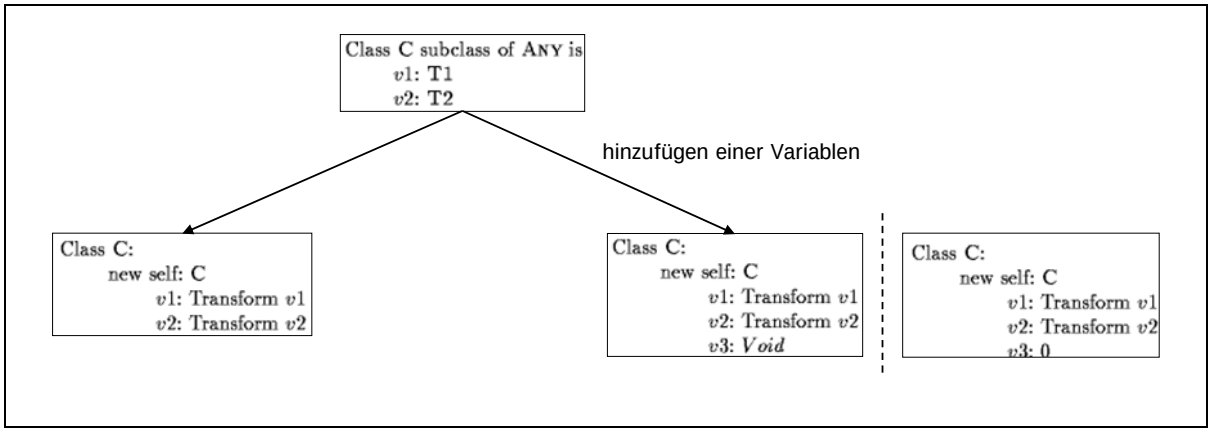

Abbildung 8: Hinzufügen einer Variablen in OTGen aus [LH90]

Nachfolgend wird ein weiteres Beispiel für eine kontextabhängige Änderung gegeben:

> Angenommen, man fügt eine Klasse D als Unterklasse der Klasse C in dem Beispiel wie folgt ein:
>
> Class D subclass of C is
> $v2$: T4
>
> Will man nun die Objekte der Klasse C, bei denen der Variablenwert von $v2$ vom Typ T4 ist, in Objekte des Typs D umwandeln, gibt man folgende Transformationstabelle an:
>
> Class C:
> if TypeCheck (Transform($v2$), T4)
> new self: D
> $v1$: Transform $v1$
> $v2$: Transform $v2$
> else
> new self: C
> $v1$: Transform $v1$
> $v2$: Transform $v2$

Auch die Umstrukturierung von Hierarchien ist möglich und wird in [LH90] ausführlich beschrieben.

Einen anderen toolgestützten Ansatz wurde an der Universität Frankfurt entwickelt. Das Projekt COAST versucht mit Hilfe von aufwendigen objektorientierten Frameworks, idealerweise im laufenden Betrieb, auf Modifikationen des Datenmodells geeignet zu reagieren Das wird in der Regel durch das Propagieren von Änderungsmeldungen im Framework, an Komponenten die auf Daten operieren, erreicht [@COAST].

4 Zusammenfassung

Schemaevolution ist ein aktuell intensiv verfolgtes Tätigkeitsfeld. Vor allem in den relationalen DBMS wird verstärkt versucht, entsprechende Techniken zu unterstützen. Sowohl IBM, Oracle und auch Microsoft unterstützen momentan nur relativ simple Evolution-Prozesse. Ein formal vollständiger Ansatz, der das Data Refactoring ähnlich dem Code Refactoring kochrezeptartig beschreibt, wird in [Amb05] gegeben.

Schemaevolution in ooDBMS war ein von Anfang an berücksichtigter Punkt. Allgemein wurden bereits in den 80er Jahren Konzepte zur Realisierung der Problematik vorgestellt. Grundsätzlich wurden die Aspekte der Schemaevolution formal beschrieben und an einigen Prototypen gezeigt.

Die Umsetzung in konkreten freien oder kommerziellen ooDBMS ist bisher nur in sehr begrenztem Rahmen erfolgt, was einerseits an der noch immer relativ geringen Nachfrage nach ooDBMS liegen kann, aber weiterhin auch der Komplexität des Problems geschuldet ist.

Das View-Konzept eignet sich vor allem für Änderungen, die die Kapazität eines Schemas reduzieren oder erhalten. Um auch kapazitätenerweiternde Operationen ausführen zu können, sind bisher individuelle Erweiterungen wie beispielsweise das MultiView-Konzept aus [RR97] notwendig. Ein weiterer Nachteil des View-Ansatzes bei objektorientierten Datenbanksystemen ist, dass es immer noch kein einheitliches und standardisiertes View-Konzept für ooDBMS gibt. Auch die Problematik von Update-Operationen auf Views ist nicht geklärt. Schwierig gestaltet sich auch die Konsistenzwahrung bei Änderungen im Klassenmodell.

Der View-Ansatz weisst generell Gemeinsamkeiten mit dem Versionierungsansatz auf. Auf eine gemeinsame Datenbasis kann über unterschiedliche Schnittstellen zugegriffen werden. Allerdings ist das Versionierungskonzept wesentlich flexibler und klarer, da sowohl Vorwärts- als auch Rückwärtskonvertierungen unterstützt werden. Außerdem gehen keine Informationen verloren, da Ausgangngs- und Zielschema über die Ableitungsfunktion miteinander verbunden sind. Das Versionierungsmanagement eignet sich auch sehr gut, um unterschiedliche

Entwicklungsstränge eines Projektes zu verfolgen, die verschiedene Auffassungen der Diskurswelt wiederspiegeln können. Weiterhin unterstützen Schemaversionen Änderungstransparenz, was kostenintensive Apllikationsanpassungen überflüssig macht. Allerdings wird durch die Verwaltung unterschiedlicher Schemata und Ableitungsgraphen auch die Komplexität erhöht und die Effizienz beim Datenzugriff verringert.

Grundsätzlich zeigt sich aber an allen Ausführungen, dass die Unterstützung von Schemaevolution deutliche Gewinne für die Datenbankentwicklung bringt. Die reale dynamische Welt kann damit auch in der Datenverarbeitung umgesetzt werden und auch Life-Cycle-Prozesse lassen sich besser darstellen.

Abbildungsverzeichnis

Literaturverzeichnis

[AH88] Abiteboul, S. & Hull, R. (1988). *Restructuring hierarchical database objects.* Theoretical Computer Science, 62

[ALP91] Andany, J., Léonard, M., & Palisser, C. (1991) Management Of Schema Evolution In Databases. In Proceedings of the 17th international Conference on Very Large Data Bases, 1991, 161-170

[Amb05] Ambler, S. & Sadalage, P. J. (2005). *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley Professional

[@Amb05] Ambler, S. (2005). Database Refactoring. (http://www.agiledata.org/essays/databaseRefactoring.html; zuletzt besucht: 12.12.2005)

[BCG+87] Banerjee, J., Chou, H.-T., Garzy, J., Kim, W., Woelk, D. Ballou, N. & Kim, H.-J.: *Data model issue for object-oriented applications.* ACM Transactions on Office Information Systems, Band 5, Nr. 1, 1987.

[BKKK87] Banerjee, J., Kim, W., Kim, H., & Korth, H. F.(1987). *Semantics and implementation of schema evolution in object-oriented databases.* Proc. ACM SIGMOD 1987, 311-322.

[@COAST] http://www.dbis.informatik.uni-frankfurt.de/~coast/

[FMZFM95] Ferrandina, F., Meyer, T., Zicari, R, Ferran, G. & Madec, J. (1995). *Schema and Database Evolution in the O2 Object Database System.* VLDB 1995: 170-181

[Heu97] Heuer, A. (1997). *Objektorientierte Datenbanken. Konzepte, Modelle, Standards und Systeme.* Addison-Wesley

[LH90] Lerner, B. & Habermann, A.N. (1990). *Beyond Schema Evolution to Database Reorganization*. OOPSLA 1990

[RR97] Ra, Y. & Rundensteiner, E. (1997). *A Transparent Schema-Evolution System Based on Object-Oriented View Technology.* TKDE 1997

[Scholl92] Scholl, M. H., Laasch, C., Rich, C., Schek, H.-J. & Tresch, M. (1992). The COCOON Object Model. Technical Report 211, Dept of Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland. (http://citeseer.ist.psu.edu/219515.html)

[TS93] Tresch, M & Scholl, M. H. (1993). *Schema Transformation without Database Reorganization.* Sigmod Record 1993