in: Proc. Int. Workshop "Component-based Application Engineering in Research and Practice", Wien, Sept. 2001

Es ist an der Zeit, die Prinzipien komponentenbasierten Software Engineerings im objekt-relationalen Datenbankentwurf anzuwenden!

Wolfgang Mahnke, Hans-Peter Steiert
AG Datenbanken und Informationssysteme
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern

Abstract

Bei der Anwendungsentwicklung werden komponentenbasierte Methoden bereits erfolgreich eingesetzt. Beim objekt-relationalen Schemaentwurf ist dies allerdings noch nicht der Fall. Dies liegt u.a. daran, dass Konzepte, die einen solchen Entwurf unterstützen, noch nicht im Standard SQL:1999 enthalten sind. In diesem Aufsatz stellen wir Ansätze zur Klassifikation verschiedener Module in objekt-relationalen Datenbanksystemen vor. Um die impliziten Abhängigkeiten zwischen den Modulen explizit zu machen, definieren wir verschiedene Beziehungen zwischen den Modulen und geben Definitionsbeispiele für Module und deren Beziehungen. Anhand dieser Definitionen soll ein objekt-relationales Datenbanksystem in die Lage versetzt werden, die Module und die Beziehungen dazwischen zu verwalten und somit die Grundlage für einen komponentenbasieren Schemaentwurf zu bieten.

Keywords: *ORDBMS, schema design, modularity*

1. Einleitung

Die Anhänger der Komponententechnologie sehen in ihr das nächste Paradigma der Softwareentwicklung. Hinter Komponenten steckt die Idee, Teilfunktionalität unabhängig vom späteren Einsatzkontext realisieren zu können und große Systeme durch Kombination vorgefertigter Komponenten zu erstellen. Komponenten fördern daher die Wiederverwendung und diese führt wiederum dazu, dass Komponenten schneller ausreifen. So genannte Komponentenmodelle [8, 9, 11] bieten die technologische Plattform, um Komponenten auch in heterogenen Umgebungen zu Informationssystemen zu kombinieren. Komponentenbasierte Entwicklungsmethoden [1, 3] fördern die Erstellung von Komponenten und die Entwicklung großer Systeme mit Hilfe von Komponenten.

Wenn wir aber die Entwicklung großer, datenbankgestützter Informationssysteme betrachten, so müssen wir feststellen, dass komponentenbasierte Methoden hier kaum Einzug gehalten haben. Während auf der Anwendungsebene Komponenten und komponentenbasierte Entwicklungsmethoden durchaus zum Einsatz kommen, verlieren sich deren Vorteile oft auf der Ebene der Datenbanksysteme (DBS). Dies hat hauptsächlich vier Gründe. Erstens ist in datenintensiven Anwendungen das DBS meist der Flaschenhals. Um ein ausreichendes Leistungsverhalten zu erreichen, werden die lose gekoppelten Komponenten der Anwendung im Datenbankschema wieder enger gekoppelt. Zweitens erfordert die Integritätssicherung in der Datenbank, dass Integritätsbedingungen über die Grenzen logischer Komponenten hinweg formuliert werden müssen. Drittens überlappen sich häufig Komponenten auf Anwendungsebene in Schemaausschnitten. Um eine redundante Speicherung zu vermeiden, werden diese dann vereinigt. Viertens haben Datenbanken eine lange Lebensdauer, während der ihr Schema ständig gewartet, verändert und optimiert wird. Dabei gehen die Grenzen der Anwendungskomponenten leicht verloren. Letztlich entsteht in der Datenbank ein kaum strukturiertes, stark verflochtenes, flaches Netzwerk von Abhängigkeiten zwischen Schemaelementen. Dadurch wird der Austausch oder die Veränderung von Komponenten auf Anwendungsebene erschwert oder unmöglich gemacht.

Unserer Meinung nach liegt dies vor allem daran, dass die Standardsprache SQL auch in ihrer neuesten Version SQL:1999 [2] keine Konzepte zur Modularisierung von Datenbankschemata und keine Mechanismen zur Verwaltung der Abhängigkeiten bietet [7]. Dies ist jedoch Voraussetzung für einen komponentenbasierten Schemaentwurf. Die Situation hat sich durch das Aufkommen der objekt-relationalen Datenbanksysteme (ORDBS) [10] eher noch verschlechtert, da durch die vielen neuen Konzepte, insbesondere durch das Konzept der Erweiterbar-

keit, das Potenzial für gegenseitige Abhängigkeiten weiter gestiegen ist.

Mit einem komponentenbasierten Schemaentwurf könnte auch beim logischen Datenbankentwurf die Wiederverwendung gefördert werden. Da bei ORDBS Teile der Anwendungslogik in das DBS integriert werden, um datenintensive Operationen nahe an den Daten ausführen zu können, erhöht sich der Aufwand des logischen Datenbankentwurfs stark, und damit auch der Nutzen der Wiederwerwendung.

In diesem Aufsatz präsentieren wir erste Ansätze zur Erweiterung von SQL:1999 um Konzepte zum komponentenbasierten Schemaentwurf. Ähnlich wie auf Anwendungsebene sollten Schemata aus vorgefertigten Bausteinen kombiniert werden. Weiterhin wollen wir Abhängigkeiten zwischen Komponenten eines Datenbankschemas durch das ORDBS kontrollieren und verwalten lassen. Zunächst zeigen wir in Abschnitt 2 anhand eines Beispiels die Probleme des objekt-relationalen Schemaentwurfs auf. In Abschnitt 3 führen wir dann unseren Modulbegriff und drei Klassen von Modulen ein, die wir identifiziert haben. Diese semantischen Einheiten sind Voraussetzung für einen komponentenbasierten Schemaentwurf. Auf die Beziehungen zwischen den Modulen gehen wir in Abschnitt 4 ein. Abschnitt 5 erläutert die Bestandteile der drei Klassen genauer und gibt Beispiele von Moduldefinitionen. Wir schließen diesen Beitrag mit einer Zusammenfassung und einem Ausblick ab.

2. Probleme beim objekt-relationalen Schemaentwurf ohne Modularität

Die Voraussetzung für einen komponentenbasierten Schemaentwurf sind lose gekoppelte Module, wobei möglichst wenig Abhängigkeiten zwischen diesen Modulen bestehen sollen. Die notwendigen Abhängigkeiten zwischen den Modulen müssen explizit gemacht werden, andere Abhängigkeiten vom System unterbunden werden. Der aktuelle Standard SQL:1999 bietet aber keine semantischen Einheiten von Schemaelementen, dies kann bestenfalls in der Dokumentation geschehen. Die Abhängigkeiten entstehen implizit zwischen den Schemaelementen der semantischen Einheiten. Eine Klassifikation dieser Abhängigkeiten kann [7] entnommen werden (siehe auch Abbildung 1). Ein ORDBS bietet keine Möglichkeiten, diese Abhängigkeiten zu unterbinden.

Im Folgenden wollen wir an einem Beispiel erläutern, welche Abhängigkeiten sich beim objekt-relationalen Schemaentwurf ergeben. Das Beispiel ist ein vereinfachter Ausschnitt aus der SFB-501-Erfahrungsdatenbank (EDB) [4].

Dort haben wir einen benutzerdefinierten Typ (engl. user-defined type, UDT) `ObjektID` implementiert. Ein `ObjektID`-Wert dient als eindeutiger Identifikator und enthält Informationen bezüglich des Speicherungsortes (Tabelle) und des Typs eines Objekts. Neben der `ObjektID` ist eine Tabelle (`Wurzel_ta`, siehe Abbildung 1) definiert, die ein Primärschlüsselattribut vom Typ `ObjektID` sowie Trigger zu dessen Wartung enthält. Jede Tabelle, die eine `ObjektID` als Primärschlüssel enthalten soll, muss von dieser Tabelle erben. Für die Vererbung zwischen Tabellen in einem ORDBS ist es notwendig, diesen Tabellen UDTs zuzuweisen, die ebenfalls in einer identischen Vererbungshierarchie stehen. Entsprechend hat die Wurzeltabelle den Typ `Wurzel_ty`.

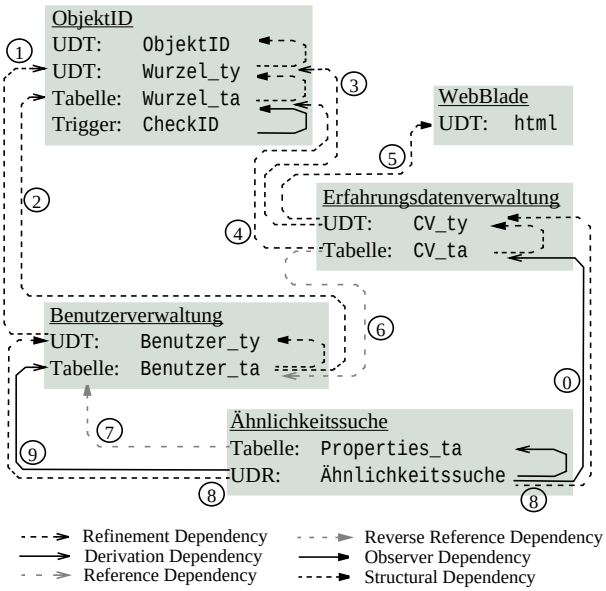

Abbildung 1: Ausschnitt aus dem EDB-Schema

Ein weiterer Aspekt der EDB ist die Benutzerverwaltung, vereinfacht durch die Tabelle `Benutzer_ta` und den UDT `Benutzer_ty` repräsentiert. Da wir für den Benutzer eine `ObjektID` benötigen, muss `Benutzer_ty` von `Wurzel_ty` (1) und `Benutzer_ta` von `Wurzel_ta` (2) erben. Diese Art der Abhängigkeit nennen wir *Refinement Dependency*.

Die Erfahrungsdatenverwaltung spielt in der EDB eine zentrale Rolle. Wir beschränken uns hier darauf, dass Erfahrungsdaten als sogenannte Charakterisierungsvektoren (CV) gespeichert werden. Diesen soll wie den Benutzern eine `ObjektID` zugewiesen werden. Entsprechend wird der Tabelle der CVs (`CV_ta`) ein Typ `CV_ty`

zugewiesen. Dieser erbt von `Wurzel_ty` (3), die Tabelle `CV_ta` von `Wurzel_ta` (4). Für die Definition der CVs reichen die Built-in-Typen des ORDBS nicht aus. Deshalb verwenden wir den UDT `html` des Informix WebBlade [5] in `CV_ty` (5). Die Verwendung von Datentypen anderer semantischer Einheiten führt zu einer *Structural Dependency*. Im CV wird gespeichert, welcher Benutzer die Erfahrung angelegt hat. Dies ist über eine Fremdschlüsselbeziehung zwischen `CV_ta` und `Benutzer_ta` realisiert (6), was *Reference Dependency* genannt wird.

Zum Auffinden von Erfahrungen haben wir eine ähnlichkeitsbasierte Suche in einer benutzerdefinierten Routine (engl. user-defined routine, UDR) `Ähnlichkeitssuche` implementiert. Die Maße der Ähnlichkeit können dabei durch Parameter spezifiziert werden. Diese Parameter werden in der Tabelle `Properties_ta` gespeichert. Jedem Benutzer ist genau ein solcher Satz an Parametern zugewiesen, wobei auch mehrere Benutzer die gleichen Parameter verwenden können. Diese Beziehung ist mit einem Fremdschlüssel realisiert (7), wobei das Fremdschlüsselattribut in `Benutzer_ta` eingetragen werden muss. Diese Art der Abhängigkeit, bei der der Fremdschlüssel in der referenzierten Einheit realisiert werden muss, wird *Reverse Reference Dependency* genannt. Die `Ähnlichkeitssuche` bekommt beim Aufruf einen `Benutzer_ty` und eine Vergleichsinstanz `CV_ty` als Parameter (8). Daraufhin liest die UDR aus `Benutzer_ta` (9) und `Properties_ta` die zu verwendenden Ähnlichkeitsmaße ein. Anschließend werden eine Anfrage auf `CV_ta` (0) ausgeführt und die Treffer nach Ähnlichkeit geordnet zurückgegeben. Anfragen auf Tabellen anderer semantischer Einheiten nennen wir *Derivation Dependency*.

Obwohl bei der EDB bereits auf einen modularen Entwurf geachtet wurde, sind die Module auf Schemaebene nicht zu erkennen. Die Module sind nur in der Dokumentation wiederzufinden. Aber selbst dies hilft bei der Wiederverwendung der Module wenig, da die Abhängigkeiten zwischen den Modulen nicht explizit gemacht wurden und die impliziten Abhängigkeiten schwer zu erkennen sind. Einige Abhängigkeiten machen die Wiederverwendung von Modulen unmöglich und sollten deshalb nicht zugelassen werden. Beispielsweise verändert die Reverse Reference Dependency die Struktur der Benutzerverwaltung, obwohl die Abhängigkeit von der Ähnlichkeitssuche ausgeht. Damit kann die Benutzerverwaltung in einem anderen Kontext nicht verwendet werden.

Für einen modularen Schemaentwurf fordern wir:

- Semantisch zusammenhängende Einheiten müssen als solche verwaltet werden können.
- Implizite Abhängigkeiten auf Ebene der Schemaelemente müssen explizit auf Modulebene definiert werden.
- Das ORDBS muss Module und die Beziehungen dazwischen als Teil des Schemas verwalten und unerlaubte Abhängigkeiten unterbinden.

Verschiedene Arten von semantisch zusammenhängenden Einheiten werden wir im Folgenden vorstellen. Die Beziehungen zwischen den Modulen werden erläutert, und exemplarisch Definitionen für die Module und Beziehungen angeben, nach denen das ORDBS die Abhängigkeiten warten soll.

3. Semantisch zusammenhängende Einheiten

Anhand der EDB haben wir verschiedene Arten von Modulen identifiziert, die wir in diesem Abschnitt beschreiben werden.

Die Module unterscheiden sich beispielsweise nach der Art der Schemaelemente, die sie enthalten können, oder der Möglichkeit, ob und wie oft ein Modul instanziert werden kann. Als potenzielle Elemente eines Moduls eines ORDBS sind Tabellen, Sichten, UDTs, UDRs, verschiedene Arten von Integritätsbedingungen, Trigger, Rollen und Benutzerrechte zu nennen. Als unterschiedliche Modulklassen haben wir Components, Packages, und Frameworks identifiziert.

Ein *Component* ist ein Modul, das eine eigene Verarbeitungseinheit darstellt und eigenständig existieren kann. Ein Component wird einmal definiert und kann mehrfach instanziert werden. Beispielsweise kann das Component Benutzerverwaltung in einer Datenbank mehrfach verwendet werden, einmal zur Verwaltung der Benutzer der Erfahrungsdatenverwaltung, ein anderes Mal für Benutzer der Produktdatenverwaltung. Die Definition eines Component enthält Definitionen von Schemaelementen. Erst bei der Instanzierung des Component werden die Schemaelemente angelegt. Jede Instanz eines Component spannt einen eigenen Namensraum auf. Dies ist nicht nur erforderlich, um ein Component mehrfach zu instanzieren, sondern auch, um Namenskonflikte zwischen Elementen verschiedener Components zu vermeiden. Bei Components besteht keine Einschränkung bzgl. der Schemaelemente. Die Erfahrungsdatenverwaltung und die Ähnlichkeitssuche aus Abbildung 1 sind weitere Beispiele für Components.

Ein *Package* dient dazu, anderen Modulen Schemaelement-Definitionen – wie vordefinierte Datentypen und Routinen – zur Verfügung zu stellen. Beispielsweise stellt das WebBlade einen HTML-Datentyp zur Verfügung. Im

Gegensatz zu einem Component kann ein Package nicht eigenständig instanziert werden. Die Elemente eines Package werden aber bei jeder Instanzierung eines Component, das das Package verwendet, in dessen Namensraum angelegt. Ein Package kann UDTs, UDRs, Sichten, und Tabellen enthalten, sowie einfache Integritätsbedingungen und Trigger, die sich auf diese Tabellen beziehen. In einem Package können keine Rollen und Benutzerrechte spezifiziert werden, dies gilt auch für Integritätsbedingungen, die sich auf das gesamte Datenbankschema beziehen (Assertions).

Ein *Framework* bietet bereits ein vorgegebenes Gerüst für ein Component, das aber für seine Verwendung noch um konkrete Eigenschaften erweitert werden muss. Die ObjektID ist ein Beispiel für ein Framework. Obwohl ein Framework kein eigenständiges Component ist, kann sich die Anzahl der erlaubten Instanzen eines vervollständigten Framework unterscheiden. Im Fall der ObjektID muss genau eine Instanzierung existieren, falls alle `ObjektID`-Werte in der gesamten Datenbank eindeutig sein sollen. In diesem Fall darf das Framework lediglich einmal in seinem eigenen Namensraum instanziert werden. Falls der `ObjektID`-Wert jedoch lediglich im Kontext eines Component eindeutig sein soll, muss das Framework jeweils im Namensraum des entsprechenden Component instanziert werden. Um das ObjektID-Framework zu vervollständigen, muss mindestens eine Subtabelle unter der Wurzeltabelle definiert werden. Ein Framework kann bereits alle Schemaelemente enthalten.

4. Beziehungen zwischen Modulen

Je nach Art eines Moduls kann es auf unterschiedliche Weise Gebrauch von anderen Modulen machen. Die Abhängigkeiten zwischen Modulen ergeben sich dabei immer auf der Ebene der Schemaelemente. In Abbildung 1 werden Beispiele dieser Abhängigkeiten aufgezeigt (0-9). Um diese Abhängigkeiten besser in den Griff zu bekommen, definieren wir verschiedene Beziehungen zwischen den Modulen, die mögliche Abhängigkeiten zwischen den Elementen der Module festlegen. Ein Modul stellt dafür eine Menge von Konnektoren zur Verfügung, die von anderen Modulen verwendet werden können. Außerhalb der durch die Beziehungen der Module erlaubten Abhängigkeiten auf den Konnektoren dürfen keine weiteren Abhängigkeiten zwischen Elementen der Module auftreten. In Abbildung 2 werden die verschiedenen Beziehungen zwischen den Modulklassen aufgezeigt.

Ein Package kann lediglich andere Packages verwenden. Dabei ergeben sich zwei verschiedene Beziehungsarten. Ein Package P_1 kann ein anderes Package P_2 *einbinden*, d.h., die eingebundenen Elemente von P_2 werden in den Namensraum von P_1 aufgenommen. Um Namenskonflikte zu vermeiden, können bei der Beziehungsdefinition die einzubindenden Elemente aufgezählt und ein Mapping auf andere Namen angegeben werden. Dies bedeutet, dass im Prinzip Kopien der Elemente von P_2 in P_1 angelegt werden. Wird ein Package P_4 von einem Package P_3 *importiert*, kann P_3 zwar Elemente von P_4 verwenden, muss diese aber im Namensraum von P_4 ansprechen. Hier werden die Elemente lediglich referenziert.

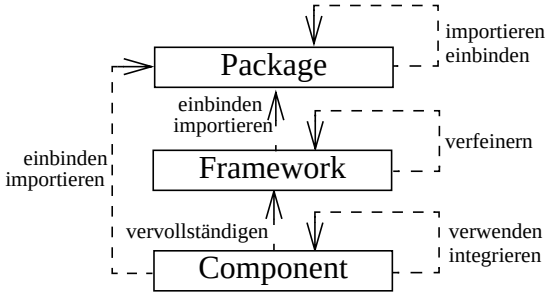

Abbildung 2: Beziehungen zwischen Modulen

Ein Framework F_1 kann ein Package P_1 *einbinden* oder *importieren*. Es ergibt sich die gleiche Semantik wie bei den Beziehungen zwischen Packages. Ein Framework F_1 kann ein anderes Framework F_2 *verfeinern*. Dabei wird F_1 bereits um Eigenschaften erweitert, das Ergebnis ist aber noch kein eigenständiges Component. Die Elemente von F_2 werden in den Namensraum von F_1 kopiert.

Ein Component kann Packages *einbinden* oder *importieren*. Dies entspricht der oben angesprochenen Semantik. Falls ein Component instanziert wird, werden neben den Tabellendefinitionen des Component auch alle Tabellen der eingebundenen Packages im Namensraum des Component angegeben. Für die Tabellendefinitionen eines importierten Package wird zunächst der Namensraum des Package im Namensraum des Component angelegt und anschließend die Tabellen in diesem Namensraum. Ein Component kann ein Framework *vervollständigen*. Bei der Vervollständigung eines Framework ist zu unterscheiden, ob das Framework genau einmal oder mehrfach instanzierbar ist. Im ersten Fall spannt das Framework seinen eigenen Namensraum auf, im zweiten Fall werden die Elemente des Framework in den Namensraum des Component integriert. Bei der Instanzierung des Component werden im ersten Fall neben den Elementen des Component auch die Elemente des Framework im Namensraum des Framework angelegt, falls dies noch nicht

geschehen ist. So können verschiedene Components das gleiche Framework vervollständigen und über den gleichen Namensraum darauf zugreifen. Im zweiten Fall werden die Elemente des Framework bei jeder Instanzierung des Component im Namensraum des Component angelegt. Schließlich kann ein Component C_1 ein anderes Component C_2 *integrieren* oder *verwenden*. Im ersten Fall wird eine Instanz von C_2 in jeder Instanz von C_1 angelegt und ist nur über diese Instanz adressierbar. Im zweiten Fall liegen C_1 und C_2 in verschiedenen Namensräumen und können darüber angesprochen werden.

5. Definitionen von Modulen und Beziehungen

Nachdem wir zunächst die verschiedenen Modulklassen beschrieben und anschließend die Beziehungen zwischen den Klassen aufgezählt haben, wollen wir jetzt beispielhaft Definitionen der verschiedenen Module und der Beziehungen zwischen den Modulen vorstellen.

```
1 DEFINE PACKAGE WebBlade
2  IMPORTING IfmxClobs
3  INCLUDING XmlBlade
4  (DEFINE TYPE HTML (...),
5   DEFINE TYPE XHTML OF XML,
6   DEFINE TABLE html_tags(...),
7   ...)
8  EXPORTING HTML,XHTML;
```

Abbildung 3: Beispiel einer Package-Definition

Die Definition eines Package ist in Abbildung 3 zu sehen. Zunächst wird ein eindeutiger Name angegeben (Zeile 1). Anschließend werden die Beziehungen spezifiziert, die von dem Package ausgehen (Zeile 2-3). Daraufhin werden die Elemente definiert, die in dem Package enthalten sind (Zeile 4-7). Dabei können auch die in den Beziehungen spezifizierten Elemente verwendet werden. Beispielsweise wird in Zeile 5 der Typ XML aus dem eingebundenen Package XmlBlade verwendet. Schließlich wird festgelegt, welche Elemente des Package von anderen Modulen verwendet werden dürfen (Zeile 8). Diese Konnektoren des Package sind die einzigen Elemente, die außerhalb des Package angesprochen werden können. Nichtsdestotrotz bedeutet die Instanzierung eines Component, die das Package einbindet, dass alle Tabellen des Package angelegt werden müssen, auch wenn lediglich die Elemente des Package darauf zugreifen dürfen.

Die Definition eines Framework erfolgt bei den Beziehungen (Zeile 2-3 in Abbildung 4) und Elementen (Zeile 4-8) wie bei einem Package. Auch bei einem Framework müssen die Konnektoren spezifiziert werden, die eine Verfeinerung oder Vervollständigung des Framework ermöglichen. Da die Konnektoren auf verschiedene Art verwendet werden können, kann bei der Spezifikation auch die Benutzungsart angegeben werden. In Zeile 10 wird beispielsweise `wurzel_ta` sowohl für die Erstellung von Subtabellen als auch für eine Sicht definierende Anfrage freigegeben. Ausgeschlossen sind damit z.B. Fremdschlüsselreferenzen auf diese Tabelle. Da ein Framework noch ein unvollständiges Gerüst eines Component ist, muss spezifiziert werden, an welchen Stellen das Framework vervollständigt werden muss. Das Framework aus Abbildung 4 kann beispielsweise nur dann von einem Component vervollständigt werden, falls eine Subtabelle unter der Tabelle `wurzel_ta` angelegt wird (Zeile 11-12).

```
1 DEFINE FRAMEWORK ObjektID
2  INCLUDING TriggerFunctions
3  REFINING  Object
4  (DEFINE TYPE wurzel_ty (.. ) UNDER root_ty,
5   DEFINE TABLE wurzel_ty(...) OF wurzel_ty
6   UNDER root_ty,
7   DEFINE TRIGGER CheckID(...),
8   ...)
9  EXPORTING wurzel_ty FOR INHERITANCE,
10  wurzel_ta FOR INHERITANCE, VIEW DEFINITION
11 NEEDS COMPLETION
12  OF wurzel_ta THROUGH INHERITANCE;
```

Abbildung 4: Beispiel einer Framework-Definition

Da ein Component mehrfach instanziert werden kann, ist es sinnvoll, zunächst eine Definition des Component anzugeben. Sonst würde die mehrfache Instanzierung eines Component jedesmal die Spezifikation des Component erfordern. Abbildung 5 zeigt einen Ausschnitt der Definition des Erfahrungsdaten-Component (Zeile 1-10). Bei der Vervollständigung eines Framework muss spezifiziert werden, ob ein globaler oder lokaler Namensraum verwendet wird (Zeile 3).

```
1 DEFINE COMPONENT Erfahrungsdatenverwaltung
2  INCLUDING WebBlade
3  COMPLETING ObjektID FOR MULTIPLE USE
4  USING Ähnlichkeitssuche AS SimSearch
5  REFERENCING Benutzerverwaltung
6  (DEFINE TYPE cv_ty (Beschreibung HTML...)
7   UNDER wurzel_ty,
8  ...)
9  EXPORTING cv_ty FOR REFERENCING
10 WITH INTERFACE Ähnlichkeitsfunktion, cv_ta;

11 CREATE COMPONENT edv1
12  OF Erfahrungsdatenverwaltung
13  REFERENCING Benutzerverwaltung bv1;
```

Abbildung 5: Beispiel einer Component-Definition

Da ein Component instanziert wird und damit von einem Anwendungsprogramm auf diese Instanz zugegriffen werden kann, muss hierfür eine Schnittstelle definiert werden (Zeile 10). Dies ist bisher in SQL nicht vorgesehen, d.h., zur Zeit darf auf jedes Schemaelement zugegriffen werden, falls dies nicht durch Benutzerrechte untersagt ist. Bei einem komponentenbasierten Schemaentwurf ist eine solche Schnittstelle aber sehr wichtig, damit Components ausgetauscht werden können.

Bei der Instanzierung des Component müssen Angaben bezüglich des Namensraums gemacht werden. In Zeile 11-13 wird der Aufruf für eine Instanzierung dargestellt.

6. Zusammenfassung und Ausblick

In diesem Aufsatz haben wir Ansätze für eine Klassifikation von Modulen in ORDBS vorgestellt. Die verschiedenen Klassen bieten unterschiedliche Möglichkeiten der Wiederverwendung an. Anschließend wurden verschiedene Beziehungen zwischen den Klassen definiert. Mit diesen Beziehungen sollen Abhängigkeiten zwischen den Modulen sichtbar gemacht und eingeschränkt werden. Wir haben bereits erste Vorschläge für eine Erweiterung von SQL:1999 skizziert, um einen komponentenbasierten Schemaentwurf zu gewährleisten. Anhand der Definition von Modulen und der Beziehungen soll das ORDBS in die Lage versetzt werden, Module zu verwalten und die Abhängigkeiten zwischen den Modulen zu überwachen.

In unserer weiteren Arbeit wollen wir die vorgestellten Modulklassen und die Beziehungen dazwischen genauer spezifizieren. Dazu ist es insbesondere notwendig, die Möglichkeiten der Konnektorenspezifikation genauer zu durchleuchten. Die Verwendung von Schnittstellen zu den Anwendungsprogrammen sowie die Aggregation verschiedener Components zu neuen Components muss ebenfalls untersucht werden. Zur Spezifikation wollen wir zunächst eine Metamodellierung der verschiedenen Modulklassen auf Basis von UML durchführen und schließlich einen Vorschlag für eine Spracherweiterung von SQL:1999 einbringen, der den unserer Meinung nach dringend notwendigen komponentenbasierten Schemaentwurf ermöglicht. Unsere Erfahrungen in verschiedenen Projekten [4, 6] haben gezeigt, dass ansonsten die Handhabung von ORDBS und insbesondere die Einbettung von Anwendungslogik in die Datenbank extrem schwierig sind.

7. Literatur

[1] Atkinson, C., Bayer, J., Laitenberger, O., Zettel, J.: Component-Based Software Engineering: The KobrA Approach. 3rd International Workshop on Component-based Software Engineering, Limerick, Ireland, 2000

[2] ANSI/ISO/IEC 9075-2-1999: Database Languages - SQL - Part 2: Foundation (SQL/Foundation). American National Standard Institute, Inc., 1999

[3] D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML - The Catalysis Approach. Addison Wesley, 1998

[4] Feldmann, R.L., Geppert, B., Mahnke, W., Ritter, N., Rößler, F.: An ORDBMS-based Reuse Repository Supporting the Quality Improvement Paradigm - Exemplified by the SDL-Pattern Approach. TOOLS USA 2000, Santa Barbara, CA, July, 30 - August, 3, 2000.

[5] Informix Web DataBlade Module - Application Developer's Guide, Version 4.0, Informix Software, Inc., 1999

[6] Mahnke, W., Ritter, N., Steiert, H.-P.: Towards Generating Object-Relational Software Engineering Repositories. 8. GI-Fachtagung „Datenbanken in Büro, Technik und Wissenschaft" (BTW'99), Freiburg, March 1999

[7] Mahnke, W., Steiert, H.-P.: Modularity in ORDBMSs - A new Challenge. 13. Workshop „Grundlagen von Datenbanken", GI-FG 2.5.1, Magdeburg, Juni 2001

[8] Microsoft: The Component Object Specification, Version 0.9, Oktober 1995, Microsoft Coperation, 1995

[9] OMG 2001-02-33: The Common Object Request Broker: Architecture and Specification, Revision 2.4: Oktober 2000. Object Management Group, 2000

[10] Stonebraker, M., Brown, M.: Object-Relational DBMSs - Tracking the Next Great Wave. Morgan Kaufmann, 1999

[11] Sun Microsystems: Enterprise JavaBeans Specification, Version 2.0, Sun Microsystems, April 2001