Generic Components in Object-Relational Database Systems

Wolfgang Mahnke

University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
mahnke@informatik.uni-kl.de

Abstract. Modularizing programs and developing systems in a component-based way is state-of-the art in application development. Object-relational database management systems (ORDBMS) are not supporting these concepts appropriately. We present our approach bringing these concepts into the object-relational schema design. Genericity is another concept to foster reuse, widely used in application development. An ORDBMS can by seen as powerful generic component, parameterizable at schema creation time and, to some degree, at run time (by the so-called schema evolution). Nevertheless, using genericity on top of this generic system to parameterize schema components is a promising way to develop more general schema components and thereby increase the opportunities to reuse them. Finally, we enhance our approach to support generic schema components.

1. Introduction

Grouping together programming units to modules or similar constructs is a common feature in programming languages or environments [Me88]. Nowadays, even a component-based design is state-of-the-art in application development ([ABB+02], [DW98]). Both enable reuse of parts of the software, structural programming, information hiding, etc.; modules normally at source code and components at machine code level. Looking at the object-relational schema design, there is a lack of such concepts to group schema elements like tables, views, user-defined types (UDT) or user-defined routines (UDR) to higher constructs. Neither the standard SQL:1999 [ISO99a] nor the vendors offer adequate concepts. In SQL:1999, there are only flat schemata and server modules [ISO99b], which mainly contain UDRs and do not deal with other schema elements. Both concepts do not offer interfaces or explicitly defined relationships between them (see [MS01], [Ma02] for details). The vendors just offer initialization units called Cartridges [Ora01] or Datablades [IBM01a], placing their elements in a schema. Nevertheless, grouping schema elements together in components, offering interfaces to these components and supporting relationships between the components would make the benefits common to component-based application design also feasible at the database schema level. Particularly w. r. t. the object-relational features of SQL:1999, like UDRs and UDTs, this would be extremely beneficial. These object-relational features allow adding much more functionality to the database schema that, in turn, leads to a much more complex and time-consuming schema design.

Genericity is another concept to foster reuse. In component-based development, it is used to create components that are more general and adapt them to different scenarios with generic parameters. This allows an easier evolution of the generic components, since common code is not replicated in several components and reduces the number of components and thereby the complexity of managing those components [Be00]. Whereas component-based design is currently not part of database schema design, genericity is a foundation of database management systems (DBMS).

More precisely, creating a database schema is parameterizing the DBMS, the probably most frequently used component in information technology. Although the DBMS itself is a generic component, it is useful to support genericity on top of it in a component-based schema design. Thus, the DBMS is parameterized using schema components, which again are parameterized themselves.

In this paper, we introduce generic, object-relational schema components. First, we motivate the need for database schema components. Afterwards, our approach of schema components is presented. After taking a closer look on common techniques supporting genericity, we complete our approach with generic schema components. The paper finishes with a conclusion and an outlook.

2. Motivation for a component-based schema design

There are several reasons to support a component-based schema design. We will discuss the main issues briefly and present an example of an object-relational schema design to demonstrate these issues in a real-live example.

2.1 Main advantages of a component-based schema design

The main advantages of a component-based schema design are in detail:

Reuse of parts of the schema. By grouping semantically interrelated schema elements in components, they can easily be reused in other schemata. For example, developing an XML data type and UDRs for managing XML documents inside the database is a cost- and time-intensive task. Moreover, this functionality is probably needed by many different schemata. By grouping the schema elements for managing XML documents in a single schema component, they can easily be reused by other schemata. Without the concept of a component, different schema elements, like UDTs and UDRs, are hidden in the schema making it unclear which elements are actually needed for a given functionality.

Easy and rapid schema design by assembling off-the-shelf components. If there is a sufficient number of (off-the-shelf) schema components, new schemata can be developed mainly by combining existing schema components. This does not only reduce development costs, but also decrease the time-to-market.

Quality and robustness of a schema. By (re)using high quality, well-tested schema components, both quality and robustness of a schema increase. Since most testing can be done on a small excerpt of the schema (the schema component), testing is much simpler compared to testing the whole schema at a time. The complexity of the object-relational technology, especially the UDRs, makes testing necessary.

Exchange and extension of parts of the schema. Using interfaces and, thereby, achieving *information hiding*, schema components implementing the same interface can be replaced in a schema. This allows the optimization of schema parts and the inclusion of new code into an existing schema. Furthermore, the schema can be extended with new functionality by replacing an old schema component with a new one offering more functionality, but still complying with the old interface. We have to mention that most schema components have a persistent state, e.g., stored in the tuples of a table. Exchanging the stateful schema components in a running system requires that the state of the old schema component is transferred to the new one. Nevertheless, exchanging schema components seems to be a promising way to handle schema evolution.

Structural and distributed schema design. Modularity allows structural design by dividing different tasks of the schema in different schema modules and only defining interfaces and relationships between them. The schema components can be developed independently of each other by distributed (groups of) programmers.

Continued, component-based design. Nowadays, a component-based development of applications is often blurred as far as the data storage component is concerned. The application components are separated at the application layer, but to manage their persistent data, they use a global database schema with overlapping parts. If each application component offers its own schema component and if only well-defined relationships between the schema components are used, separation and isolation are preserved at the schema level.

The main objective of a component-based schema design is to control the dependencies between schema elements of different schema components. The dependencies may emerge from a foreign key definition between two tables, but also a UDR call or a trigger definition can lead to a dependency. In [MS01], a description and classification of the different dependencies is given in detail. Our example in the following section will give a brief overview about the dependencies. Using narrow interfaces and defining relationships between different schema components, the number of possible dependencies is restricted and other dependencies (unknown to the DBMS) are prohibited.

2.2 An example of an object-relational schema design

As stated in the introduction, there is no sufficient mechanism to support a component-based, object-relational schema design. Nevertheless, we introduce an example of an object-relational schema considering groups of schema elements as components. The example is used to denote all dependency types between the schema elements and illustrate them in an explicit way. It is a simplified excerpt derived from the SFB-501-Reuse-Repository[8] [FGM+00, MR02, SFB02].

The Reuse-Repository is designed to support all phases of a reuse process and the accompanying improvement cycle of the Quality Improvement Paradigm [BR91, FGM+00] by providing adequate functionality. To gain more experience with the new object-relational technology, we have chosen the, as we call it, *extreme extending* (X^2) approach, i.e., almost everything is integrated into the DBMS by using the extensibility infrastructure of the object-relational DBMS (ORBDMS). Thus, X^2 means that not only the entire application logic runs within the DB-server, but also major parts of the presentation layer (GUI) reside within the DB-server, since the Reuse-Repository dynamically generates its HTML pages used for user interaction within the DBMS. In this context, we do not want to describe the functionality of the Reuse-Repository in detail. Briefly summarized, its main functionality is to manage experience data and to support similarity-based search on such data.

Because our ORDBMS does not support a reference type, we have implemented a UDT called `ObjectID`. An `ObjectID` value serves as a unique identifier and stores information about the storage location (table) and the type of the object. In addition to the `ObjectID`, there is a typed table called `root_ta` (see Fig. 1) of the type `root_ty` including an `ObjectID` as primary key and some triggers maintaining the `ObjectID` (e.g., keeping values unique). All tables using the `ObjectID` inherit from `root_ta`. Together these schema elements build a component

[8] The SFB-501-Reuse Repository is part of the subproject A3 "Supporting Software Engineering Processes by Object-Relational Database Technology" of the Sonderforschungsbereich 501, *"Development of Large Systems by Generic Methods"*, funded by the German Science Foundation.

providing an object identifier. There are already some dependencies between the schema elements of the component, for example, an *observer dependency* of the triggers observing the `root_ta`.

Another aspect of the Reuse-Repository is related to user management, which is simplified here as a typed table `user_ta` of the type `user_ty`. Because a user needs an `ObjectID`, the type inherits from `root_ty` (1) and the table from `root_ta` (2). This kind of dependency is called *refinement dependency*. Note that these dependencies leave the borders of the user management component and affect the component providing the object identifier.

The experience management plays a central role in our Reuse-Repository. In our example, experience data is stored in the form of characterization vectors (CV). Similar to the user, a CV should have an `ObjectID`. Therefore, the typed table `cv_ta` inherits from `root_ta` (3) and its type `cv_ty` from `root_ty` (4). Because the build-in types of the ORDBMS are not expressive enough to represent a CV, we used the UDT `html` (5) of the Informix WebBlade [IBM01b]. The use of data types establishes *structural dependencies*. In the CV, its creator is recorded, which is realized with a foreign key relationship to `user_ta` (6). The corresponding dependency is called *reference dependency*.

To retrieve experience data we have implemented a similarity-based search by a UDR called `SimSearch`. The measures of similarity are specified by parameters. These parameters are stored in a table called `properties_ta`. To apply a specific similarity function, each user refers to a set of parameters, where several users can use the same set of parameters (e.g., the default set). A foreign key between `user_ta` and `properties_ta` represents this relationship. Hence, the foreign key attribute is included in the `user_ta` (7). Although the dependency is caused by the similarity search, the foreign key is realized in the referenced component, the user management. Such a kind of dependency is called *reverse reference dependency*. Calling `SimSearch`, occurrences of a `user_ty` and of a `cv_ty` as comparison instance have to be specified as parameters (8). `SimSearch` selects the parameters for the similarity function by reading the `user_ta` and the `property_ta` (9). Afterwards a query is evaluated on `cv_ta` (10) and the results are returned ordered by the similarity value. Evaluating a query on tables of other semantic units leads to a *derivation dependency*.

Although we have observed a component-based schema design in the Reuse-Repository, these structures cannot be seen at the schema level. It can only be found in the documentation. Because the implicit, hard-to-find dependencies are not made explicit, reuse of single schema components is impossible. Even if we would have an explicit component structure, some kinds of dependencies would prevent the reuse of schema components and, therefore, have to be avoided. For example, the *reverse reference dependency* changes the structure of the referenced component. Therefore, the referenced component cannot be reused in another context.

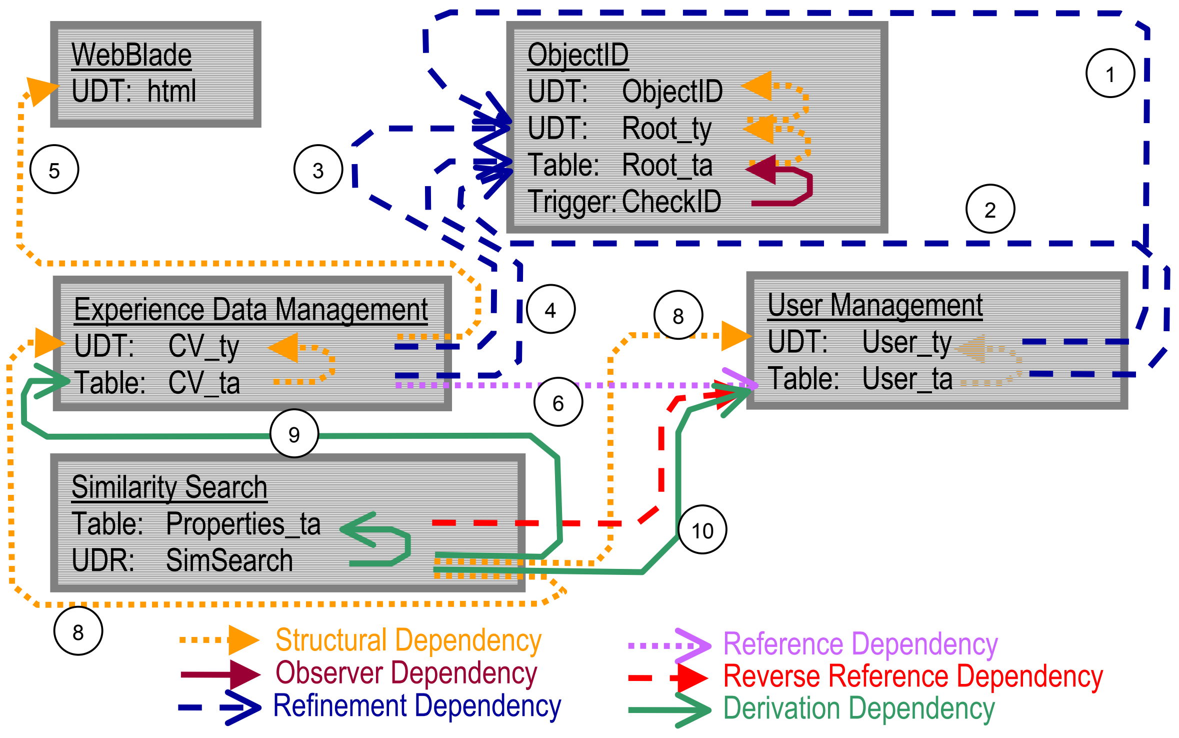

Figure 1. Excerpt of the database schema of the Reuse-Repository

Consequently, for a component-based schema design we demand:

- semantically interrelated schema elements must be managed within schema components;
- implicit dependencies between schema elements have to be made explicit at the component level;
- schema components and their relationships have to be managed as part of the schema and disallowed dependencies have to be prevented;
- interfaces should be used as an abstraction from the implementation of a component;

The last point is obvious, since components should be developed independently of each other, but often depend on other components.

3. A framework for a component-based schema design

In this section, we introduce a framework for a component-based schema design without considering genericity. Genericity will be added to the framework in section 5, after regarding general generic techniques in section 4.

3.1.1 Design Decision

To support schema components it is important to realize that there has to be a separation of the declaration and the initialization of a component. A DDL statement like a "`CREATE TABLE` …" directly[9] declares a schema element and deploys it. Using a schema component the same way would make it

[9] Of course you have to observe the transaction context of the statement.

- impossible to use the same component twice in a schema and
- much more complicated to use schema components with other schema components.

Therefore, there are two possibilities how to integrate a component-based design into the schema design. The first one is to use a development environment for schema component declarations and only use a small language extension of SQL:1999 providing interfaces, a namespace, etc. when deploying a schema component and at run time. We call such an approach *weak extension*. The other possibility is to integrate the declaration of the schema components into the language extension, too, which leads to the *strong extension* approach. The main advantage of the weak extension approach is that the language extension is smaller; nevertheless, you have to provide a language extension to proper support schema components, especially information hiding by observing the dependencies. Using the weak extension approach would mean that you either standardize the development environment or depend on one tool. Furthermore, information about the assembly of the schema would get lost in the metadata of the schema. This would make the maintenance of the schema much more complicated and burden the reuse and exchange of schema components. The strong extension approach, in contrast, leads to a component repository inside the database schema, making it easy to interpret, reuse, and exchange schema components. It serves as a standard environment, development tools can be build on top of it. Therefore, we choose the strong extension approach, making the schema component-aware without additional tools.

3.2 Interfaces

Interfaces are a powerful concept to enable information hiding and to allow the replacement of implementations. Interfaces are needed for a component-based schema design, since schema components either use or are even build up on other schema components. During the declaration, interfaces abstract from concrete component implementations used by the component.

Interfaces in programming languages mainly consist of function signatures, whereas this aspect differs in schema design. First, there are two kinds of interfaces, an *API* to the applications outside the database system, and a *connector* to other schema components. Second, there are additional elements in schema interfaces. They contain descriptions of different kinds of schema elements like tables, UDTs, UDRs, etc. Those descriptions must hold all information needed to use the schema elements, e.g., a signature of a UDR or the attributes and name of a table. The way to access those schema elements can be restricted. For example, it can be specified that a table can be read only, or that a trigger or a foreign key can be defined on it. We call those different possibilities *apply modes*. An API can only offer a few apply modes, such as read, write, delete, or update a table, but no mode for the definition of a trigger or a foreign key on a table. Connectors outrange APIs, since they can contain all apply modes. Interfaces cannot only contain schema elements that are directly used by other schema components or applications, but also constraints specifying the properties of an interface more accurately. Allowing constraints on UDRs, design by contract [Me88] can be supported. Each schema component can implement many interfaces, at least one. Alike object-oriented programming languages like Java, interfaces support multiple inheritance. This avoids specifying too many interfaces for one component.

In Fig. 2, an example of an interface declaration is shown. The interface describes the functionality needed for the user management of the example in section 2.2. It offers a type `user_ty` (lines 2-4) inheriting from another type `root_ty` provided by the interface ObjectID (not shown in Fig. 2) and a table `user_ta` (lines 5-6). The apply modes of the table are restricted (line 6). Note that the interface does not say that the table must inherit from `root_ta`, although it is implemented that way in section 2.2. A constraint in the interface (line 7) specifies that a component implementing this interface guarantees the constraint.

```
1. DECLARE INTERFACE UserManagementInterface
2.  TYPE user_ty AS (nname CHARACTER(20),
3.                  vname CHARACTER(20))
4.              UNDER ObjectID.Root_ty;
5.  TABLE user_ta OF user_ty
6.        FOR SELECT, INSERT, UPDATE, DELETE, REFERENCE;
7.  CONSTRAINT singlename ON user_ta UNIQUE(nname, vname);
8.  ...
9. END INTERFACE;
```

Figure 2. Example interface declaration

3.3 Different kinds of semantic units

We have identified different kinds of semantic units to group schema elements, helpful to support a component-based schema design. Since the term component is used in the literature mostly for a binary, independently executable unit, we use the notion of a *schema module* as a comprehensive term to group schema elements. Those schema modules can be distinguished in schema packages, schema components, and schema frameworks.

A *schema package* serves as a kind of library for other schema modules. It only offers UDTs and UDRs. It cannot be used directly from outside the database schema, i.e., it does not offer an API. A schema package cannot be deployed independently; it can only be deployed in the context of other schema modules. Although a schema package only offers UDTs and UDRs, it is possible that inside the schema package other schema elements are used. A UDR may need a temporary table to compute its results or even a base table, e.g., to store some values for performance reasons. Therefore, we distinguish stateful and stateless packages. A *stateless package* contains no persistent data, whereas a *stateful package* does. The differentiation is important for the internal management of the packages, a stateless package only needs one deployment in a schema, and different namespaces can be mapped to the same package, whereas a stateful package is deployed for every namespace. The WebBlade in section 2.2 is an example of a stateless schema package.

A *schema component* can contain all schema elements and is independently deployable. A schema component can offer APIs and connectors and is multiple deployable in a single schema. The namespace of a schema component is specified at deployment time. The Experience Data Management, the User Management, and the Similarity Search in section 2.2 are examples of schema components.

A *schema framework* provides the foundations for a schema component, but it is not a complete schema component. That is, it first needs to be completed and therefore cannot be deployed independently. Therefore, you can say a schema framework is a kind of abstract schema component similar to abstract classes in object-oriented programming languages. An example of a schema framework is the ObjectID in section 2.2.

In Fig. 3, you can see the declaration of the user management component used in section 2.2.

```
DECLARE COMPONENT UserManagement
IMPLEMENTING UserManagementInterface
COMPLETING ObjectID FOR MULTIPLE USE;
DEFINE TYPE user_ty AS (nname CHARACTER(20) DEFAULT NULL,
                        vname CHARACTER(20) DEFAULT NULL)
                       UNDER ObjectID.Root_ty;
DEFINE TABLE user_ta
 (singlename UNIQUE(nname, vname))
 OF user_ty UNDER ObjectID.Root_ta;
END COMPONENT;
```

Figure 3. Example component declaration

3.4 Relationships

Depending on the kind of the schema modules, different relationships can be defined between them. A relationship is always declared on an interface during the declaration of a schema module. When deploying the schema module, the interface is bound to a corresponding schema module. Depending on the relationship, the bound schema module is deployed, too, or an already deployed schema module is used.

Schema packages can only *import* other schema packages; a stateless schema package can of course only import other stateless schema packages. Importing means, that the elements defined by the interface of the imported package can be used in the other schema module, either using an additionally namespace or directly using the namespace of the schema module. Schema frameworks and schema components can import schema packages, too.

A schema framework can *refine* other schema frameworks, i.e., the schema elements declared in the interface of the refined schema framework are usable in the namespace of the refining schema framework and are supplemented by other schema elements. To a certain degree, this is comparable to an inheritance relationship between classes. The used interface of the refined framework has to be provided by the refining framework, either directly or by using interface inheritance a more specialized one. The elements declared in the interface are inherited, too. Nevertheless, since a schema framework can offer more than one interface, it is not the same as class inheritance.

A schema component can *complete* a schema framework. Similar to the refinement relationship, the elements offered by the interface of the schema framework are usable in the namespace of the completing schema component. However, the used interface of the schema framework has not to be provided by the schema component.

Schema components can *utilize* other schema components. All elements provided by the interface of the utilized component can be used by the utilizing component, either in its own namespace or directly in the namespace of the utilizing component.

Only schema components can be directly deployed. When deploying the component, it has to be specified which other schema modules are used; the declaration of the schema component only refers to interfaces. Except for stateless schema packages, which are only deployed once in a schema, it has to be specified, whether the used schema modules are deployed exclusively for the schema component or not. Although it would be sufficient to declare this at deployment time, we think it is better doing it at declaration time, since the intention to use another schema module exclusively or not is better understood by the person declaring the component than the person deploying it. If the other schema module is used exclusively, it has to be deployed in the context of the schema component. If it is not used exclusively, either an existing schema component has to be specified or a new global schema module has to be deployed.

4. Generic Techniques

To a certain degree, genericity can always be seen as parameterizing something (e.g., a class or a schema component) with generic parameters. There is a wide range of techniques when to apply the parameters and how the parameters look like.

In programming languages, generic parameters can be applied either before compilation, normally using a pre-processor, e.g., macros in C, or during compilation. The most popular example for using genericity at compilation time is the template mechanism of C++. There is already a large library of standard templates widely used [Au99]. There are approaches bringing similar mechanisms to other programming languages like Java [BOS+98]. According to Czarnecki and Eisenecker, generic parameters are type parameters or value parameters of types [CE00].

In component-based development environments like EJB [SUN01], there is another point in time after the compilation: the deployment. Generic components require parameterization at deployment time. This is normally described in a deployment descriptor, containing the generic parameters but also information about the environment. Similar to programming languages, generic parameters are type or value parameters. Baum and Becker however, demand code fragments as generic parameters in the context of generic components [BB00]. Since you can include code in type parameters via methods, this is not necessary in object-oriented environments. Even if this is not feasible in the component environment used, it is possible to either implement the code in another component and compose both to the needed one or use an adapter as proposed in [KAZ98].

Looking at DBMS it is hard to compare it with the lifecycle of a program (compile – [deploy] – run). The DBMS serves as large component that is already running when parameterizing certain aspects of it. Therefore, the DDL statements, i.e. the generic parameters, are executed at run time and the results outlast the reboot of the DBMS. Hence, the DBMS offers the most general way to parameterize it. Since SQL:1999 is computationally complete, you can specify everything you like while the system is running, even changing your parameterization (e.g., `ALTER TABLE` …). Beside type parameters (creating UDTs) and value parameters (e.g., inserting data into a table) you can create tables, UDRs, etc.

5. Generic schema components

Which generic techniques are useful for schema components? The most powerful technique is already offered by ORDBMS, and schema components become a part of it. Nevertheless, since those techniques are already part of an ORDBMS, they are too powerful for generic schema modules. To keep the parametrization simple, we think it is sufficient to provide type and value parameters. Functionality that needs to be added to a schema module can be provided either by the methods of a type parameter or by another schema module. The parameters of a generic schema module have to be provided at deployment time.

Since a schema module is only accessible using its interfaces, the interfaces have to be generic, too. In Fig. 4, an example of a generic interface for the user management is shown. Two types and one value are expected as generic parameters (lines 2-3). You can specify the kind of type expected (structured, distinct or built-in) and additional constraints. Whereas T1 can be of any type, T2 has to be a structured type without an attribute called `gender`. This is necessary, since T2 is used as a structured type and an attribute gender is added in a subtype of it (line 4). Not using parameter constraints could lead to an error during the deployment.

```
1    DECLARE INTERFACE UserManagementInterface
2    <TYPE T1, STRUCTURED TYPE T2(CHECK NO ATTRIBUTE gender),
3     INT length (CHECK 10<length<100)>
4     TYPE description_ty AS (gender T1) UNDER T2;
5     TYPE user_ty AS (nname       CHARACTER(length),
6                      vname       CHARACTER(length),
7                      description decription_ty)
8                   UNDER ObjectID.Root_ty;
9    ...
10   END INTERFACE;
```

Figure 4. Example of a generic interface declaration

A non-generic schema module has to specify the parameter of any used generic interface during its declaration; a generic schema module can hand over the parameters to any used generic interface at deployment. Fig. 5 illustrates the declaration of a generic user management. Note, that some parameters (`T1` and `length`) are passed to the interface, whereas other parameters are already utilized at declaration time (`CHARACTER(10)`). That is, the schema component only implements a restricted interface compared to the one specified in Fig. 4. It is reasonable that constraints on the parameters of the interface can only be restricted in the constraints of the parameters of the schema module (e.g. line 3 vs. line 13).

```
11  DECLARE COMPONENT UserManagement
12   <STRUCTURED TYPE T1 (CHECK NO ATTRIBUTE gender),
13    INT length (CHECK 10<length<50)>
14   IMPLEMENTING UserManagementInterface
15               <CHARACTER(10), T1, length>
16   COMPLETING ObjectID FOR MULTIPLE USE;
17   DEFINE TYPE description_ty AS (gender CHARACTER(10))
18               UNDER T1;
19   DEFINE TYPE user_ty AS
20         (nname       CHARACTER(length) DEFAULT NULL,
21          vname       CHARACTER(length) DEFAULT NULL,
22          description description_ty)
23         UNDER ObjectID.Root_ty;
24  DEFINE TABLE user_ta
25   (singlename UNIQUE(nname, vname))
26   OF user_ty UNDER ObjectID.Root_ta;
27  END COMPONENT;
```

Figure 5. Example of a generic component declaration

The parameters of the generic schema modules have to be passed at deployment time. Therefore, we offer a DDL statement that either contains the parameters and used schema modules directly or refers to a deployment descriptor. The deployment descriptor can contain references to deployment descriptors of the used schema modules. Allowing a deployment descriptor hierarchy makes the deployment process clear and allows the developer of a schema module to offer a default deployment descriptor for its module.

6. Conclusion and Outlook

In this paper, we have introduced a framework for a component-based schema design. The framework allows the easy reuse of parts of a schema, a structural design, and a continued, component-based design at application and database level. Different kinds of schema modules are needed: packages, frameworks, and components; offering different kinds of relationships between them. Interfaces abstract from concrete implementations of the schema modules.

Genericity can be used to develop components that are more general and adapt them with generic parameters. Similar to concepts in programming languages, we provide generic parameters of types and values for the schema modules to enhance our approach to generic components. We use constraints on the generic parameters to disallow a misuse of the schema modules. Using deployment descriptors, we avoid a too complex deployment statement and enable default configurations for the schema modules.

We demand an adequate use of the generic technique and forbid a too excessive use like specifying parts of the code. Since you can use other schema modules as some kind of parameterization, too, this seems to be a more adequate solution for complex parameters.

As further work, we will make a formal specification of the language extension of SQL:1999 supporting generic schema modules. Afterwards, we want to develop a system supporting the language extensions based on an existing ORDBMS.

Additional, we have to think about the following problems:

- How do we exchange or modify schema modules if they are already deployed? What will happen to the current state, i.e. the persistent data of the schema module? Nevertheless, exchanging schema modules seems to be a promising way to handle schema evolution easier.
- Do measures exist for a good schema design? Can those measures be calculated to help the designer to provide a good modular design?
- Can we reengineer existing schemata to get a modular design, thereby getting a better understanding of the schema and gaining reusable schema components?

Our experience in various projects using object-relational technology [MRS99, MR02] has shown that handling ORBDMS is extremely hard without the benefits of a component-based schema design.

References

[ABB+02] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Peach, B., Wust, J., Zettel, J.: *Component-Based Product Line Engineering with UML*. Addison Wesley, 2002.

[Au99] Austern, M. H.: Generic Programming and the STL – Using and Extending the C++ Standard Template Library. Addison Wesly, 1999.

[BB00] Baum, L., Becker, M.: *Generic Components to Foster Reuse*. Proc. 4th IEEE International Conference on the Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2000), Sydney, Australia, 2000.

[Be00] Becker, M.: *Generic Components: A Symbiosis of Paradigms.* 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00), Erfurt, October 2000.

[BOS+98] Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: *Making the future safe for the past: Adding Genericity to the Java Programming Language*. 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98), Vancouver, Canada, October 1998.

[BR91] Basili, V. R., Rombach, H. D.: *Support for comprehensive reuse*. In IEE Software Engineering Journal, 6(5):303–316, September 1991.

[CE00] Czarnecki, K., Eisenecker, U. W.: *Generative Programming, Methods, Tools and Applications*. Addison-Wesley, 2000.

[DW98] D'Souza, D. F., Wills, A. C.: Objects, Components, and Frameworks with UML - The Catalysis Approach. Addison Wesley, 1998.

[FGM+00] 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, 2000, pp. 125-136.

[IBM01a] IBM: *IBM Informix DataBlade Module Development Overview*, Version 4.0. IBM Corporation. August, 2001.

[IBM01b] IBM Corp.:IBM Informix Web DataBlade Module Application Developers Guide. Version 4.13, IBM Corp., December 2001.

[ISO99a] ANSI/ISO/IEC 9075-1:1999, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework). ISO, 1999.

[ISO99b] ANSI/ISO/IEC 9075-2:1999, Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM). ISO, 1999.

[KAZ98] Kücük, B., Alpdemir, M. N., Zobel, N.: Customizable adapters for black-box components. Proc. 3rd Int. Workshop on Component Oriented Programming (WCOP 98), 1998, pp. 53-60.

[Ma02] Mahnke, W.*: Towards a modular, object-relational schema design*, in: Proc. 9th Doctoral Consortium at CAiSE'2002, Toronto, May, 2002, pp. 61-71.

[Me88] Meyer, B.: *Object-oriented Software Construction*. Prentice Hall, 1988.

[MR02] Mahnke, W., Ritter, N.: The *ORDB-based SFB-501-Reuse Repository*. VIII. International Conference on Extending Database Technology (EDBT 2002), Demo Presentation Session, Prague, 2002, pp. 745-748.

[MRS99] Mahnke, W., Ritter, N., Steiert, H.-P.: Towards *Generating Object-Relational Software Engineering Repositories*. Proc. 8. GI-Fachtagung 'Datenbanksysteme in Büro, Technik und Wissenschaft' (BTW'99), Freiburg, March 1999, pp. 251-270.

[MS01] Mahnke, W., Steiert, H.-P.: *Modularity in ORDBMSs - A new Challenge*, in: Proc. 13. Workshop "Grundlagen von Datenbanken", GI-FG 2.5.1, Magdeburg, Juni 2001, pp. 83-87.

[Ora01] Oracle Cop.: *Oracle 9i Data Cardridge Developer´s Guide*. Oracle Cooperation, June 2001.

[SFB02] SFB-501 Reuse Repository, https://edb.sfb501.uni-k.de.

[SUN01] Sun Microsystems: *Enterprise JavaBeans Specification*. Version 2.0, Sun Microsystems, Inc., April 2001.