in: Tagungsband 13. Workshop ‚Grundlagen von Datenbanken", GI-FG 2.5.1, Magdeburg, Juni 2001

# Modularity in ORDBMSs – A new Challenge

Wolfgang Mahnke, Hans-Peter Steiert
Database and Information Systems Group, University of Kaiserslautern
P.O. Box 3049, D-67653 Kaiserslautern

**Abstract:** "Modularity is the key to achieving the aims of reusability and extensibility" stated Bertrand Meyer in 1988. Relational database systems lack proper abilities supporting a modular design. There are only flat schemata and no encapsulation. The SQL:1999 standard has integrated object-oriented concepts into the relational model leading to so-called ORDBMSs. User-defined types and routines, server modules etc. introduce a new complexity but also embody new possibilities for a modular design. In this paper, we discuss how these concepts can facilitate the development of modular, ORDBMS-based applications.

## 1 Introduction

Modularity is a well known concept in the field of Software Engineering used to increase reusability and extensibility [11]. Design methods like object-oriented analysis and design (OOA&D) favour a modular design. Nowadays, software is even developed in a component-based way [7].

While component-based design is well accepted in application development, its advantages do not hold for the management of persistent data. Typically, the different components utilize a centralized data storage where all dependencies between them are reflected in the database. Further, the database is a large, global data structure, which is usually assumed to be evil because it leads to complex dependencies.

To avoid a mix of terms we undertake some definitions. A component is a modular unit outside the database. Components may need to store data persistently. Inside the database, modules divide the database into different semantic units. Normally, such a module is related to a specific component and dependencies between components are reflected between their modules. We will focus on modular database design, i.e., only modules and their dependencies are considered.

Today, relational database management systems (RDBMSs) are ubiquitous. Although RDBMSs have many powerful and useful properties, such as transactions, recovery, descriptive query language, etc. [6], they lack proper options for a modular database design. RDBMSs have been introduced to support the management of simple, structured data whereas they lack a proper support of complex data [13]. For these reasons, object-relational database management systems (ORDBMSs) have been evolved. There main design goals were, on one hand, to keep the advantages of RDBMSs and, on the other hand, to enrich data management by capabilities to manage complex data. Thus, ORDBMSs are extensions of RDBMSs. The ORDBMS standard SQL:1999 [3] is a superset of the purely relational SQL-92 [2] standard. Hence, all relational features are still available in ORDBMSs. The new, so called object-relational features of SQL:1999 are primarily user-defined types (UDT), user-defined routines (UDR), triggers, server modules, and procedural language extensions called Persistent Stored Modules (PSM). Further improvements are recursive queries, some OLAP extensions, savepoints, etc. which, however, are not relevant for this paper.

Some vendors have already offered ORDBMSs before the end of the SQL:1999 standardization process (lasting over 7 years) and have even implemented features which are not part of SQL:1999. To a certain extent, some of these features can contribute to a modular database design and are, therefore, considered later on in this paper.

We do not want to discuss whether or not object-oriented database systems (OODBMSs) are a proper alternative to ORDBMSs, because they are only used in specific application domains and did not gain general acceptance. Instead, we want to examine how the new SQL:1999 features (and some vendor-specific extensions) can be used for a modular database design. We first introduce some criteria of modularity used in software development and identify aspects of a modular database design. Using such aspects, we examine how database technology supports a modular database design, considering RDBMSs in section 3 and ORDBMSs in section 4. Section 5 finishes our paper with a conclusion and an outlook.

## 2 Aspects of Modularity

"Modularity is the key to achieving the aims of reusability and extensibility" stated Bertrand Meyer in [11]. He named five criteria to classify design methods w.r.t. modularity:

1. **Modular Decomposability:** Modular decomposability means that the design method helps to decompose a problem into several subproblems, which can be solved separately.
2. **Modular Composability:** The other way around, it should be possible to combine software elements (e.g., solutions of the subproblems) with each other to develop new systems. A method that supports this task fulfills the criterion of modular composability.
3. **Modular Understandability:** To support the maintenance of software systems, modules should be separately understandable by human readers. Ideally, it should not be needed to look at other modules, at most to a few neighboring modules.
4. **Modular Continuity:** To support modular continuity, it is necessary that the effects of changes are isolated as far as possible. A small change of the requirements should lead to a change in only one or a few modules and should not affect the relations between the modules.
5. **Modular Protection:** Modular protection is achieved if abnormal conditions (e.g., failures) inside a module are confined to the corresponding module or affect at most a few close-by modules.

Focusing on modular database design, we have identified five aspects of modularity which are outlined in the following.

**Modules:** A module groups together several elements, e.g. tables, views, triggers, etc. within a single semantic unit. In particular, a module can contain other modules, i.e., modules can be nested. Further, modules provide a namespace for the elements contained and a failure model, which allows to confine the effects of failures. Also, modules are administrative units. They can be installed and deinstalled as units. In addition, multiple instances of a module can be installed. This means, the name of each instance has to be user-defined and used as root of the corresponding namespace.

**Interfaces:** In order to specify which elements of a module are intended to be used by others, modules offer interfaces. The intention is to guarantee that interfaces are stable even if the internal structure changes. To handle different levels of dependencies between modules, a module can provide different interfaces, because a strongly dependent module may need more information than a less dependent.

**Visibility:** The visibility concept guarantees that only interfaces of a module are used by other modules and that the internal structure of a module is kept private. Note that visibility is different from authorization. While visibility is related to the relationship between modules, authorization is related to user privileges.

**Dependencies:** Modules need interfaces in order to let other modules utilize their services. This results in dependencies between the modules. The intention of modularity is to reduce dependencies. All dependencies have in common that there is a server which offers some service and a client which uses the service. We give a short classification of the different forms of dependencies:

- A *reference dependency* is given if the client references the server (e.g., a client-side foreign key).
- In a *reverse reference dependency* the client logically references a server, but this is implemented by the server (e.g., a foreign key in the server of a (n:1)-relation initiated by the client).
- If a client derives something from the server, this is a *derivation dependency.* For example, if an assertion in the client is evaluated on elements of the server, we call this a derivation dependency. Such dependencies can be further classified depending on the kind of access:
  - *Query dependencies* exist if a query is executed on a server element.
  - *Call dependencies* exist if the client calls a routine of the server.
- *Observer dependencies* are given if the client listens to server events.
- In a *structural dependency* the client uses some structural elements defined by the server, e.g., a data type.

A *transitive dependency* may occur, e.g., if a client A depends on server B and server B is client of server C, a change in C may result in a change in B and, therefore, become visible to A.

**Module Relationships:** Typically, dependencies occur implicitly and are hidden in the mud of module implementations. Hence, we have to treat them from a more administrative point of view. For this purpose, only explicit relationships allow dependencies between modules. The design environment must prevent dependencies between modules which are not explicitly related to each other. So far, we have identified two kinds of such relationships:

- *Import relationship*s express that a module imports structural elements of another module, i.e., only structural dependencies are allowed.
- *Use relationships* express that a module utilizes functionality provided by another module, i.e., all dependencies mentioned above except structural dependency.

# 3 Modularity and Relational Database Management Systems

As mentioned in the introduction, RDBMSs generally lack appropriate support for modular database design. Nevertheless, we examined the concepts offered by RDBMSs to support modularity.

**Modules:** RDBMSs offer two concepts which have properties related to the notion of modules: catalogs and schemas. Each catalog (which can be regarded as a database [5]) consists of several schemas and each schema hosts several SQL objects. Catalogs and schemas can not be nested.

Of course, a catalog can be used to improve modularity. However, this is only suitable for applications using components which are independent w.r.t. their persistent data. Since we focus on modular database design this aspect is outside the scope of this paper.

So, what about schemas? A schema is an administrative unit for the objects contained. Each schema provides a namespace for SQL objects within a catalog. Further, a unique name is assigned to each schema. Unfortunately, schemas can not be nested and do not offer a failure model.

**Interfaces:** If we would treat a schema as a module, what would be its interface? In SQL-92 there is no way to define an interface. Hence, everything is accessible by other modules and the 'implicit' interface consists of all schema elements. Because only base tables and views can reasonably be used outside the schema, we only have to consider them in the following.

**Visibility:** SQL-92 has no concept of private information. Instead of hiding information, each catalog provides an information schema which describes all SQL objects contained. As already mentioned, user privileges can not substitute visibility markers like 'private' or 'protected', because they are related to users and not to modules. This is not the intention of the visibility concept. Nevertheless, paying the price of organizational overhead, user privileges can be abused to simulate information hiding, but 'real' visibility is preferable.

**Dependencies:** If only base tables and views are used by other modules, dependencies can only cover those schema elements. Therefore, only reference and reverse reference dependencies (using foreign keys) and query dependencies (in views or constraints) can occur and may also lead to transitive dependencies.

**Module Relationships:** Without the notion of modules, interfaces, and visibility, explicit module relationships do not make sense.

Lets take a closer look at the five criteria of Bertrand Meyer w.r.t. RDBMS and modularity. Modular decomposability is rudimentally supported by schemas, but those can not be nested. The same applies to modular composability. Because of the simple data model, modular understandability is given and can be improved using comments. Since the implicit interface consists of the whole internal structure, modifying the internal structure always leads to a modification of the interface. Therefore, modular continuity is not supported. Furthermore, also modular protection is not supported, because there is no failure model in SQL-92 which allows to keep the effects of failures inside a module.

# 4 Modularity and Object-Relational Database Management Systems

ORDBMSs offer the possibility of extending the database by new functionality. Lets look at how the extensions of SQL:1999 help w.r.t. modularity.

**Modules:** Because SQL:1999 is a superset of SQL-92, schemas are still supported. An SQL:1999 schema may also include UDTs, UDRs and triggers. UDRs can be external routines written in a programming language different from SQL (like Java [1]) whereas SQL routines are written using SQL including PSM. PSM are procedural language extensions of SQL. What we have said about schemas in section 3 does also apply here.

Further, there is a new concept related to modules, called server module [4]. Server modules may contain UDRs and temporary tables, but no UDTs, base tables, views or triggers. Only UDRs contained in the same server module can access temporary tables assigned to that module. A server module is part of a schema [8] and adds its elements to the schema. Hence, a server module does not provide its own namespace, i.e., the name of a UDR defined in the server module can conflict with the name of any other UDR defined in the schema.

ORDBMS vendors come along with proprietary mechanisms to manage modules called DataBlades [9] or Cartridges [12]. Such modules enclose a number of UDRs, UDTs, base tables, views and triggers. This combination seems to be more natural than server modules, because it encompasses also UDTs and base tables. But those modules are also only administrative units, which do only extend the schema and do not provide a namespace.

SQL:1999 still does not offer a failure model to keep failures inside modules. Only UDRs have the possibility to handle errors.

**Interfaces:** In SQL:1999 there is still no interface concept. Hence, everything is accessible by other modules, including UDRs and UDTs. Triggers defined in the server can not be used by a client. However, a client can define a trigger in order to subscribe to database events on base tables (insert, update, delete) in the server.

**Visibility:** SQL:1999 offers no concept for visibility. However, information hiding has been added in a few places. First, temporary tables inside a server module can only be accessed by UDRs defined in the same server module. Second, routines, which are called by external routines, can reside outside the scope of the database and are, therefore, hidden.

Another way of hiding information is given by so-called opaque types [10] which are (unfortunately) not part of the SQL:1999 standard. The ORDBMS manages values of such types only through a predefined interface. The internal structure is hidden to the ORDBMS.

**Dependencies:** UDTs of a server can be used in base tables, views, UDTs, and UDRs of the client, i.e., structural dependencies can occur. As in SQL-92, reference, reverse reference, and query dependencies can occur, too. Using the reference data type introduced by SQL:1999, reverse reference dependencies can be avoided. This is a great advantage, because a server does not need to be changed if a client module tries to refer it in a (n:1)-relationship. Call dependencies occur, if a UDR of another module is called in constraints, views, triggers, and UDRs of the client. Therefore, both kinds of derivation dependencies can happen. In addition, client-side triggers can lead to observer dependencies.

**Module Relationships:** Again there is no notion of modules, interfaces, and visibility, i.e., module relationships do not make sense.

So, do ORDBMSs offer a better modularity support than RDBMSs? To answer this question we use again the five criteria of Bertrand Meyer. ORDBMSs offer server modules to collect UDRs resp. proprietary mechanisms to collect UDRs, UDTs, etc. However, both concepts do not offer an own namespace so that only the administration gets slightly simpler. Because of the missing namespace, these units can not be combined freely due to name conflicts. Therefore we think that modular composability and decomposability are slightly better supported.

The modular understandability depends on how the system is modularized. The object-relational extensions increase the risks to impair the modular understandability but this does not necessarily have to occur. The modular continuity is improved, because UDRs can be used as interface and encapsulate the data. However, remember that there is still no interface concept in SQL:1999 and consequently, we have to abuse the user privileges to realize visibility. Modular protection in ORDBMSs mainly covers failures in UDRs.

Recapitulating, it can be said that modularity is better supported in ORDBMSs compared to RDBMSs but ORDBMSs are still weak w.r.t. a modular database design. We are convinced that ORDBMSs should do a much better job in this concern.

# 5 Conclusions and Outlook

A modular database design is needed, because in a component-based application development environment dependencies between components may also relate their persistent data. Therefore, the persistent data of a component is put into a module and different modules have to reflect the dependencies of the components. Furthermore, the increasing complexity of the upcoming ORDBMSs urgently requires a modular database design.

Using the object-relational extensions of ORDBMSs, modularity can be enhanced in comparison to RDBMSs. But proper concepts for a modular database design are still missing in ORDBMSs. In particular, the interface and visibility concepts are essential defects.

In addition to an interface concept and a visibility concept, ORDBMSs should provide a better module concept. A module has to be able to host UDRs, UDTs, base tables, views, and other modules. Furthermore, host modules should span their own namespace so that name conflicts between elements of different modules are avoided.

Another obstacle is that some dependencies have to occur on base tables. Base tables should be encapsulated by views. But foreign keys and references can only be defined on base tables. Also triggers can only listen to events on the base tables. So, some base tables and with it (parts of) the internal structure has to be offered in an interface, even if we would have an interface concept.

To manage the implicit dependencies between different modules, a module relationship concept is needed so that only dependencies are allowed if they are made visible by module relationships.

Because of the problems mentioned, we plan to develop an environment that supports a modular database design in ORDBMSs. We first have to refine the concepts needed for a modular database design. Then, we want to develop suitable design methods and guidelines supporting a modular database design in ORDBMSs. Based on these methods, we plan to develop a module management language.

# 6 Literature


[1] ANSI X3H2-99-284: SQLJ Part 1: SQL Routines using the Java Programming Language. American National Standard Institute, Inc., 1999.

[2] ANSI X3.135-1992: Database Language SQL. American National Standard Institute, Inc., 1992. Also available as ISO Document ISO/IEC 9075:1992.

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

[4] ANSI/ISO/IEC 9075-4-1999: Database Languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM). American National Standard Institute, Inc., 1999

[5] Date, C.J., Darwen, H.: A Guide to the SQL Standard, 3rd edition, Addison-Wesley Publishing, 1993

[6] Date, C.J.: An introduction to database systems, Addison-Wesley Publishing, 2000

[7] D'Souza, D.F., Wills, A.C.:Objects, Components, and Frameworks with UML - The Catalysis Approach, Addison-Wesley Publishing, 1999.

[8] Gulutzan, P.,Pelzer, T.: SQL-99 Complete, Really - An Example-Based Reference Manuel of the New Standard. R&D Books, 1999.

[9] Informix DataBlade Module Development Overview. Informix Software, Inc, September 1999.

[10] Informix DataBlade API Developer‘s Guide. Informix Software, Inc, September 1999.

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

[12] Oracle 8i Data Cardridge Developer´s Guide. Oracle Cooperation, December 1999.

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