Towards a Component Based Model for Database Systems

Octavian Paul ROTARU[b], Mircea PETRESCU[a], Marian DOBRE[b]

[a] *The Computer Science and Engineering Department University "Politehnica" of Bucharest, Romania*

[b] *On leave of absence from *, Currently at Amdocs Development Ltd. 141, Omonia Avenue, The Maritime Center, 3045 Limassol, Cyprus*

corresponding addresses: Octavian.Rotaru@ACM.org, MirceaStelian@yahoo.com, Marian.Dobre@Amdocs.com

Abstract

Due to their effectiveness in the design and development of software applications and due to their recognized advantages in terms of reusability, Component-Based Software Engineering (CBSE) concepts have been arousing a great deal of interest in recent years. This paper presents and extends a component-based approach to object-oriented database systems (OODB) introduced by us in [1] and [2]. Components are proposed as a new abstraction level for database system, logical partitions of the schema. In this context, the scope is introduced as an escalated property for transactions. Components are studied from the integrity, consistency, and concurrency control perspective. The main benefits of our proposed component model for OODB are the reusability of the database design, including the access statistics required for a proper query optimization, and a smooth information exchange. The integration of crosscutting concerns into the component database model using aspect-oriented techniques is also discussed. One of the main goals is to define a method for the assessment of component composition capabilities. These capabilities are restricted by the component's interface and measured in terms

of adaptability, degree of compose-ability and acceptability level. The above-mentioned metrics are extended from database components to generic software components. This paper extends and consolidates into one common view the ideas previously presented by us in [1, 2, 3].

Keywords

Aspect Oriented Programming (AOP), Component Based Software Engineering (CBSE), Object Oriented Database Systems (OODB), design reusability, software metrics, transaction processing

1. Introduction

Component-based software development is a method of software construction by assembly of prefabricated, configurable and independently evolving building blocks [18]. Developing applications using components is the key technology for the construction of reconfigurable, large software systems in a timely and affordable manner [10]. By reusing already tested and validated software components the effort and costs decrease, while the flexibility, reliability, and reusability of the final application increase [19].

Extending the use of the component paradigm to the universe of Object Oriented Databases (OODB) will bring all the benefits already proven in the Component-Based Software Engineering (CBSE) area. However, this is a challenging task. The way the component-based software is currently dealt with is inadequate for the world of database systems, in terms of both implementation and architecture. The main objective of this paper is to identify a method of using ready-made components and patterns in database design. Also, the aim is to use the components as an Object Definition Language (ODL) extension, by looking at them as intermediary logical wrappers for the classes inside a schema. This allows the concept of scope to be introduced as an escalating property of transactions.

Adding the component paradigm to the world of the OODB does not affect the way both structural and logical integrity are maintained. Maintaining the integrity of an OODB is a more complex task than maintaining the one of a relational database, because data is mapped into the application’s address space and there is no guarantee that it is not changed in an

unpredictable way. This is the price paid for the advantage of the significant performance improvement over server-only data access offered by relational databases.

Evaluating non-functional parameters of a software component is even more complicated than the same evaluation for objects, due to the lack of access to the implementation details. Our approach was to find a mathematical model starting from the practical considerations resulted from our experience in CBSD and to define it as a measurement. A quantitative metric was defined after exploiting the qualitative results obtained by analyzing existing software components. Also, our target was to find a unified model for adaptability and reusability, able to capture all the characteristics of a software system.

The reuse of already validated design components facilitates the validation of the projected database model as a whole, moving the database design to a component assembly approach.

Extending ODL with components opens a way for reusing database design elements in a consistent manner. This is intended as a first step towards the standardization of data storage.

2. Components – Database Schema Partitions

A persistent class, the basic entity in OODBMS, can contain inner classes, can inherit and can be inherited from. This flexibility of the object-oriented model can be used in order to define “partitions” or “components” of the schema.

Strongly related persistent classes can be logically grouped in components. Such schema components can be represented in ODL by using wrapper classes or by adding a new ODL element for this purpose.

As defined by Szyperski and Pfister [8], a software component represents the basic unit of composition, having a contractually specified interface and explicit context dependencies only. A software component can be integrated with other components or can be independently deployed in order to be used by third parties.

The above definition leads to the properties presented below [10]. A software component:

- Is a unit of independent deployment
- Is a unit of third-party composition
- Has no persistent state

According to the definition, each component needs to be well separated from the other components and the environment. Therefore a component encapsulates its constituent features [10]. A database component cannot always be fully independent. A particular example of independent database component is the entire schema, any other logical grouping of classes being quasi-independent. A database component can depend on other components. If a class Class2 is inheriting from a class Class1 and they are defined in different components, the component containing Class2 will depend on the component containing Class1. Even if its external relationships are weak, a component is still interconnected with the other components of the schema.

The relations pointing outside define the interface of a database component. Two database components can be composed only if they have compatible "interfaces". Practically they depend on each other as "collaborative" components.

A database component is not conceivable without a persistent state. The general definition of a software component needs to be adapted in order for it to fit in the object-oriented database model. A database component can be defined as having the following properties:

- Is a quasi-independent unit
- Can be composed by respecting its contract (external relationships and dependencies)
- Has a persistent state

The partitioning of a database schema into components should be done according to the following composition axiom:

If Component1 and Component2 have common classes, then one and only one of the following affirmations is true:

- *Component1 completely contains Component2*
- *Component2 completely contains Component1*
- *Component1 and Component2 have exactly the same content (classes)*

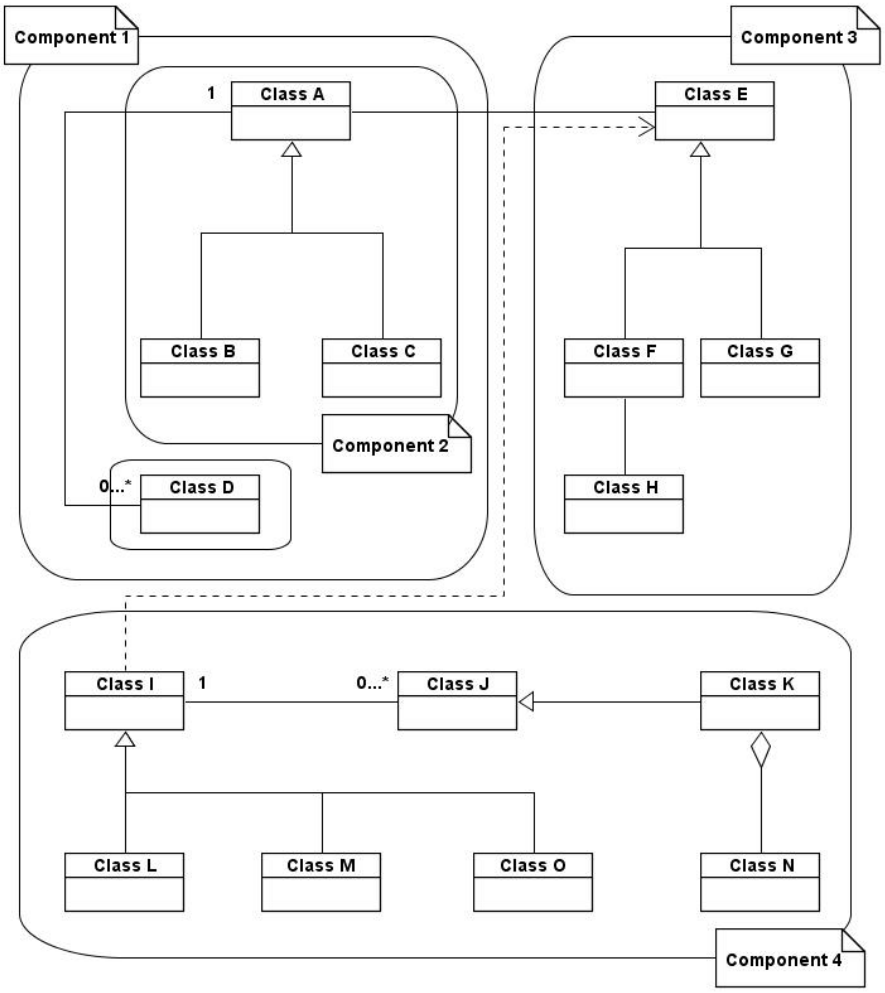

Figure 1. Database schema partitioned in components

An example of partitioning, compliant with the above-defined composition axiom, is presented in Figure 1, where Component 1 completely contains Component 2, while Component 1, Component 3 and Component 4 are disjunctive.

ODL was developed using the C++ syntax as a starting point and therefore it is natural to extend it with components by also using a C++ similar mechanism. In C++ namespaces are used to subdivide the global namespace of a project in order to avoid name clashing in large-scale projects. A similar syntax can be introduced in ODL in order to divide the schema into components. We propose the following syntax for components in ODL:

```
component / namespace [identifier]
{
      // component body
}
```

Adding two functional segments for the component's body can enrich the definition:

- Interface - containing the relationships pointing outside the namespace.
- Data – containing the class definitions.

A component can contain other components in its data segment. A new dependency literal can be added in order to clearly define the components on which the current one relies. After adding these elements the syntax of a component definition will be:

```
component [identifier]
{
// namespace body
…
interface:
      // relationships
      …
// dependencies
dependency component [comp_1];
…
dependency component [comp_n];
data:
      // class declarations
      …
}
```

3. Transactions and Scope

Usually, in a database, the scope of a transaction is the entire database schema. Adding the namespace concept to ODL as a representation of components allows the reduction of the transactions' scope.

This way it will be possible to start transactions having as scope only one component or even one class. The transaction can escalate scope-wise depending on its data processing needs.

For example, if a transaction was opened at a class level, and the same application instance tries to open another transaction at the level of a component containing that class, the scope of the class level transaction will be increased to component, without creating a new

transaction. The opposite case, when trying to start a new class level transaction while having a transaction already started at the level of a component containing the class, will produce no effect.

In object-oriented databases there are several possibilities regarding the place where to put the integrity constraints: within object instances, within object classes, or global to a set of objects. Introducing components as database partitions creates one more possibility: to have the integrity constraints stored within components.

As in every DBMS, in OODB the transactions also obey the classical ACID (Atomicity, Consistency, Isolation, and Durability) properties. In order to ensure that these properties will be respected in the perspective of having lower scope transactions, the following rules must be enforced:

- A transaction can read but cannot modify data outside its scope.
- The scope of a transaction can only be enlarged and never reduced.

The way the transactions' scope escalates is presented in Figure 2.

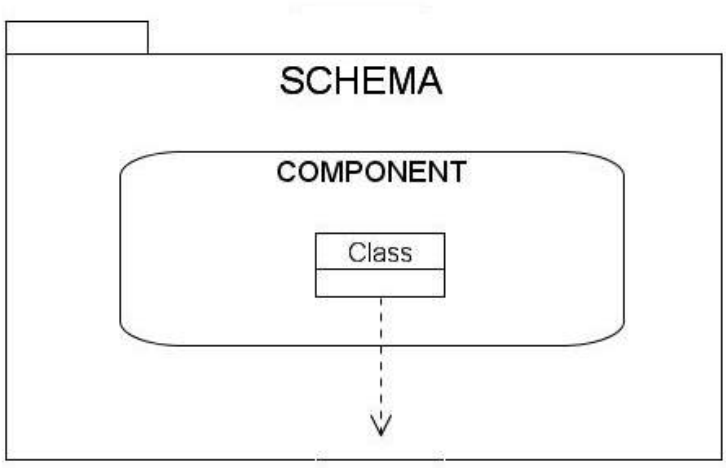

Figure 2. Transactions' Scope Evolution

4. Components and Reusability

The main purpose for splitting a database schema into components is reusability. Up to this point we presented database components only as persistent schema partitions, able to independently cope with low-scope escalating transactions. Defining the design of a database

in terms of components allows its reuse. The design of the component is the actual reusable part, so no persistent state is required. Since no persistent state is needed, a database design component comes closer to Szyperski's general definition for software components.

We consider the access statistics as part of the component design. The advantage is that the component will have good access times right from the moment of deployment. The pre-defined access statistics, validated by usage in another database, can also be used as a starting point by the optimizer of the database in which the component was deployed.

Similar storage requirements encountered in different applications can be addressed using components. In case of e-commerce websites such examples are shopping cart, guest book, etc. Compositional design reuse in database systems provides a uniform way to approach data organization. Despite its benefits, the reuse of architectural design was not much considered so far.

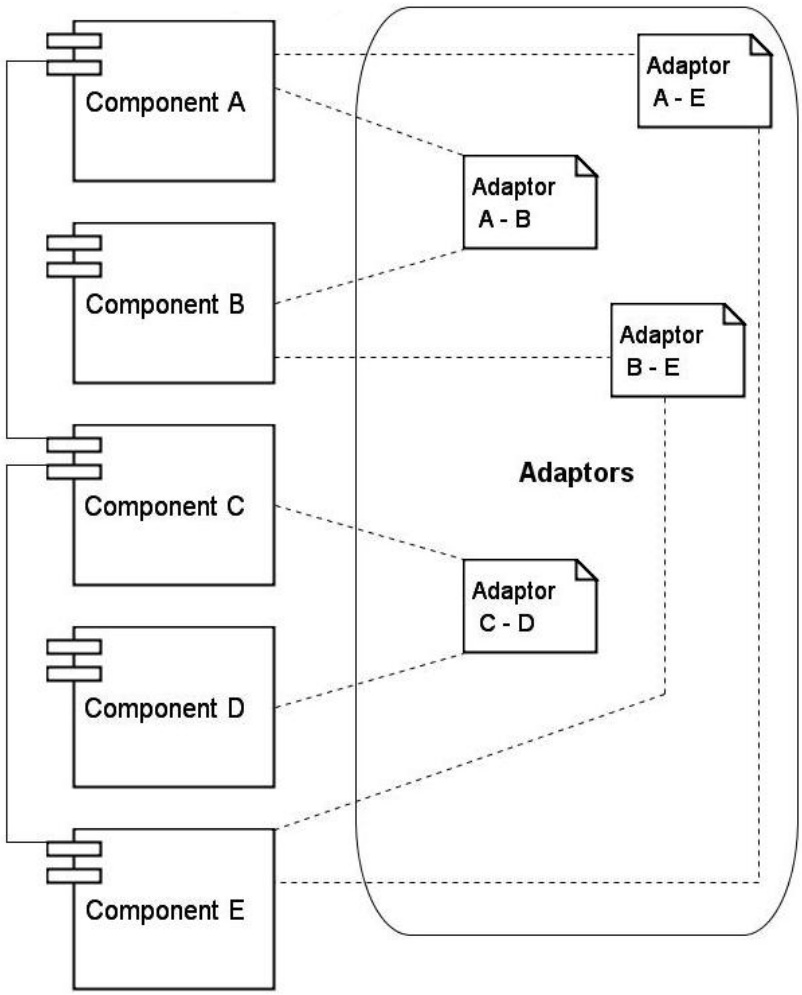

Figure 3. Adaptors Connected Components Composing a Database Schema

In the case of database components, their interfaces are defined in terms of relations. In order to assembly different components to create a database design, mediation entities have sometimes to be used to interconnect them. A mediation entity, or an adaptor, is an entity that encapsulates the native interface of a component and links it with other components or entities of the schema. The database design is therefore assembled using the interfaces provided and/or required by the components the same way Lego games are created by connecting pieces.

Figure 3 shows an example of using adaptors to connect some of the components of a database schema. Due to dependencies or compatible interfaces, some components can also be connected directly, not only through adaptors. In this example Component A is directly connected to component C, while adaptors are needed to connect it to Component B and Component E. Similarly, Component E is directly connected to Component C, while adaptors are required to connect it to Component B and Component A. A logical layer consisting of all the adaptors can be identified.

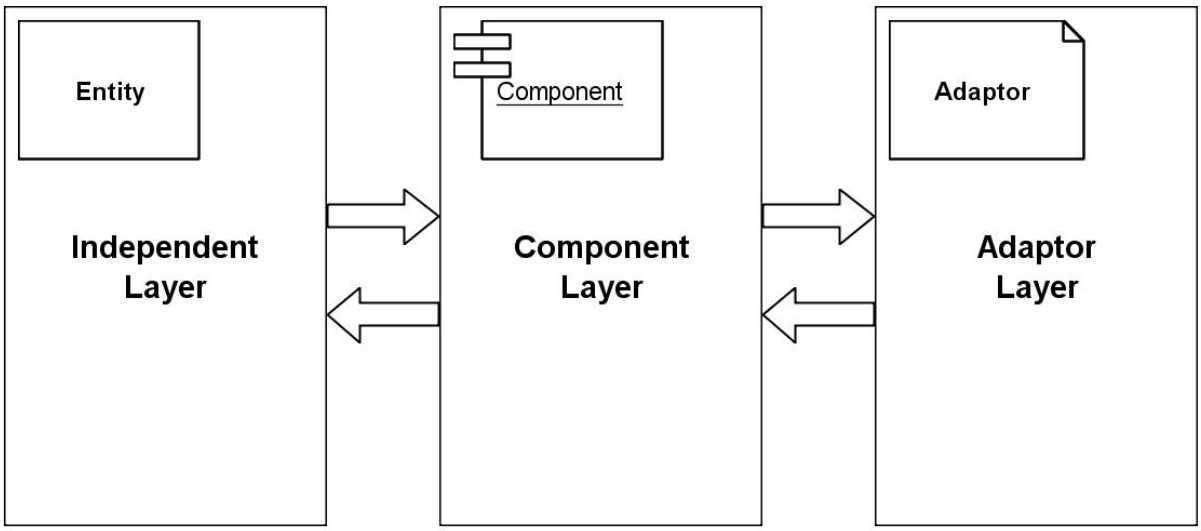

Figure 4. Multi-layer View of Component Database Schema

The database design can be logically stratified in 3 layers (as depicted in Figure 4):

- Component Layer (containing all the components)
- Adaptor Layer (containing all the adaptors)
- Independent Layer (containing all the independent entities)

Conceptually, all the other entities of the schema that are not adaptors can be logically grouped into an Independent Layer.

According to Garlan [11], the software components are composed by using connectors, which are entities determining the interaction type of the interconnected components. The consistency of the data flow between components is ensured by the connector's type specification.

In a similar manner, in a database, a connector or adaptor interconnects the interfaces of two components. Also, a system can be composed only from components and adaptors if all the independent classes will be considered components. Each independent class or even the entire independent layer can be seen as a component.

5. Database Component Affinity

An adaptor is not necessarily an entity. It can also be a group of entities, depending on the interconnecting needs of the two interfaces that the adaptor is linking. The interconnecting needs of two components can be defined in terms of an adaptability level. In fact, the adaptability level is not the property of a component, but of a pair. It defines the interconnectivity affinity of two components. From this point of view 3 levels of adaptability can be identified:

- Direct interconnectivity
- Interconnected through an adaptor entity
- Interconnected through a group of entities acting as an adaptor.

Component assembly is not only a matter of reusability, but also of adaptability. If business requirements are changing, the design needs to be flexible enough to cope with this. In a component-oriented design, changing a requirement that affects a component can be implemented by replacing the old component with a new one, compatible with the new requirement. The only element of the design that is changing is the adaptor, since it is very likely that the new component will have a different interface. The general impact on the database design, and therefore on the application using it, proves to be minimal. A database design component can be doubled by a software component able to access and manage it. Together, they constitute an autonomous unit.

6. Database Components and Aspect Orientation

Despite its evident advantages, the Component-Oriented Database (CODB) model proposed in this paper is not able to cope with the crosscutting concerns generated by the common dependencies among components.

The object-oriented technology and the CBSE paradigm offer significant advantages due to their ability to clearly separate the concerns. Two concerns crosscut if their methods intersect [23]. Only one concern can be localized using the top-down object-oriented design techniques. The dominant concern shades the others that cannot be encapsulated within the main modules, and therefore are scattered across many modules. These secondary concerns are lateral additions to the top-down approach of the object-oriented design. The usage of Aspect Oriented Programming (AOP) techniques can eliminate the dictatorship of the main concern.

AOP aims to achieve a separation of concerns similar to subject-oriented and adaptive programming. It encapsulates the crosscutting code into separate constructs called aspects. The links between aspects and classes are expressed by implicit or explicit join points and are merged together by an aspect weaver. AspectJ [5] and AspectC++ [4], the aspect-oriented language extensions available, use static aspect weavers that merge the classes and aspects at build-time. However, when applying aspect-oriented techniques in database systems a dynamic weaver is needed.

The principles of both CBSE and AOP can be joined into a new model, more generic: Aspect Component Based Software Engineering (ACBSE), having important advantages in terms of reusability, adaptability, scalability and compressibility [19]. The ACBSE model can also be applied to database design by considering aspects as possible elements of an OODB schema. In order to accommodate the ACBSE techniques, ODL needs to be further extended in order to capture aspects. The aspect definition syntax for object-oriented databases can be derived from the existing aspect-oriented language extensions. Aspect Oriented Databases (AODB) [20, 21] and CODB can be joined into a new model - Aspect Component Oriented Databases (ACODB) - that combines their advantages.

7. Components in Object-Relational and Relational Databases

The component model described above can also be applied, with minor modifications, to relational and object-relational databases. Regardless of the architecture the component design of a database maintains its advantages. Actually the situation stays almost the same, the classes being replaced by tables. Therefore, database components prove to be general partitioning mechanisms for database schemas.

Nowadays there is no pure relational DBMS. All database vendors are integrating more and more object-oriented features in their products. Furthermore, the SQL-99 standard has many object-oriented features. Compared with the old SQL-92 standard, SQL-99 brings the benefits of Object-Oriented Technology (OOT) into relational databases. Actually, most of DBMS are situated somewhere at the border between object-oriented and relational.

The new object-oriented features introduced in SQL-99 rejuvenate the old relational model into the new object-relational universe. Among them the most important are:

- attributes can be arrays of values
- rows can be stored as attributes inside other rows
- user defined data types are allowed

The evolutionary approach to ever changing storage requirements offered by object-relational systems can be combined in a very useful way with component architecture.

8. Component Integrity

The main integrity problem in OODBMS architectures is that data is usually mapped directly into the application's address space. As a result, there is no guarantee that data is not inadvertently or maliciously tampered with [26]. Hence, the ability to assure the integrity of the database is limited; the advantage being the significant performance improvement over server-only data access offered by relational databases, in which this kind of behavior is impossible, data being mapped into the address space of a separate server process.

Introducing the component paradigm to the world of OODB does not help solve this integrity problem. The component, seen as a wrapper or super class containing classes and their relationships, is mapped in the memory space of the application. Preserving the integrity

of an OODB must be done not only through internal mechanisms of the database, but also through the application using it. This also proves to be a matter of programming discipline.

In case of a component, the relationships can be divided into two categories:

- Internal relationships (between objects inside component)
- Interface relationships (composition rules)

Components can contain the integrity constraints for their internal elements, preserving its internal referential integrity. A component becomes in this way an integrity unit. The referential integrity of the interface relationships can be seen as a global integrity layer for the database schema that we are referring to.

9. Database Component Compose-Ability

In order to measure the compose-ability of a component it is required to evaluate its interface relationships. From a qualitative perspective, the compose-ability degree of a component can be defined by studying the multiplicity of the relations pointing outside. The trivial case is represented by a component with no external relationships, which is definitely having the greatest degree of compose-ability. The next level of compose-ability is represented by components interfaced only by one-one relationships. The components interfaced by one-many relationships are considered to have the lowest compose-ability.

We defined the multiplicity of a component C (μC) as being the sum of the multiplicities (m) of its interface relationships:

$$\mu_C = \sum_{i=1}^{n} m_i \tag{1}$$

where n is the number of interface relationships in C.

Considering the multiplicity of a one-one relationship to be one, the multiplicity of the one-many relationships needs to be chosen. It is up to the database designer to decide how heavy a one-many interface relationship to be in terms of multiplicity compared with a one-one relationship.

Starting from the multiplicity of a component, we propose the compose-ability degree (σC) to be defined in the following way:

$$\sigma_C(\mu_C) = \lim_{x \to \mu_C} \frac{1}{x+1} \tag{2}$$

The compose-ability degree of a component (σC) is inversely proportional with its multiplicity (μC). As illustrated in Figure 2, when the multiplicity tends to infinity, the compose-ability tends to zero. Also, the compose-ability degree is decreasing when the multiplicity is increasing. For small values of μC, the value of σC is decreasing sharply.

$$\begin{aligned} &\mu_C \in [0,\infty) \Rightarrow \sigma_C \in (0,1] \\ &\mu_C \to \infty \Rightarrow \sigma_C \to 0 \end{aligned} \tag{3}$$

Let the multiplicity of the one-many relationships be (νm). An *acceptability level* (α) for the compose-ability degree of a component, specific to the application, must be chosen. The *acceptability level* has a relationship of inverse proportionality with (νm):

$$\alpha \sim \frac{1}{\nu_m} \tag{4}$$

The relation (4) of inverse proportionality can be extended in order to properly define the *minimum acceptability level*, (α) of a component as being:

$$\alpha = \frac{1}{C \cdot \nu_m} \tag{5}$$

where C is constant. From now on (α) will be referred to as the acceptability level.

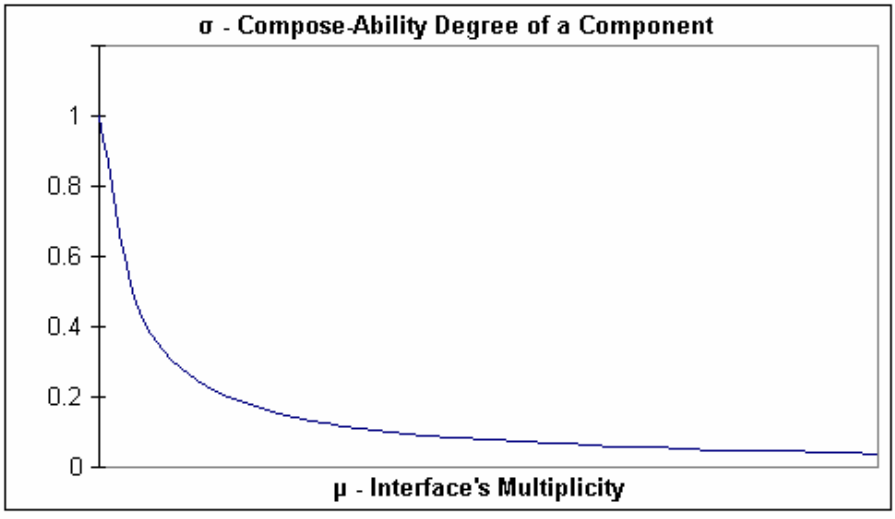

Figure 5. Compose-Ability Degree vs. Interface's Multiplicity for a Component

A component is considered as being compose-able in a context if its compose-ability degree is at least the acceptability level.

The constant number C used to calculate the acceptability level can be defined as being the maximum number of one-many relationships a compose-able component, with only one-many relationships in the interface, can have.

It is not recommended to have one-many relationships as part of an interface. Hence, if C is less than 1, than this recommendation is enforced.

10. Component Database Consistency

A database contains entities that are structured based on constraints or assertions about data. A database is consistent if all its assertions are satisfied. In some situations, if the database management system allows it, data can pass through temporary inconsistent states in order to end-up in a new consistent one. By rolling back everything if any of its actions is failing, a transaction is preserving the consistency of the database.

Choosing the component as being the elementary consistency unit and treating it as a whole, a transaction having a component as scope can pass through temporary inconsistent states. Since transactions are atomic operations, the database will finally end in a consistent state. In order to achieve this, the constraints defined inside a component must be disabled from the moment the transaction starts until the moment it ends. If the database is not in a consistent state in the moment the transaction ends and the constraints are again enforced, the transaction will fail, rolling back all its actions.

This is an optimistic consistency assurance mechanism, based on the assumption that the transactions will rarely fail or end in an inconsistent state. If the constraints are disabled during the transaction's execution, then they are checked only once, at the end of the transaction, instead of the relevant ones being checked every time data is updated.

The main advantage of this method is that the transaction is not aborted even when the component being its scope passes through inconsistent states during transaction's execution. In this way a more flexible mechanism for updating data is provided.

Such an approach is not favorable for very small transactions. Also, if a long update-intensive transaction will eventually break one of the consistency rules, than the system will still process the entire transaction, even if this situation could be potentially detected after only a few statements.

In general, the total execution time for a transaction is defined as follows:

$$tT = te + tc + to \quad (6)$$

Where

te = execution time (data access and update only)

tc = consistency time (time spent for consistency and integrity checks)

to = overhead time (i.e. for enforcing locks)

In the optimistic case of checking the consistency rules only at commit time, the execution time for the same transaction is defined as:

$$tT = te + tfc + to \quad (7)$$

Where

tfc is the time used for the final consistency checks.

The difference between the total execution times in these two cases is:

$$\Delta t_T = \begin{cases} t_{fc} - t_c, if\_commited \\ \Delta t_e + t_{fc} - t_c, if\_rollbacked \end{cases} \quad (8)$$

This result (8), in conjunction with the probability of a rollback situation to occur, can be used to decide if an optimistic approach is appropriate or not for consistency assurance.

Figure 6 depicts a situation in which the database is passing through a temporary inconsistent state.

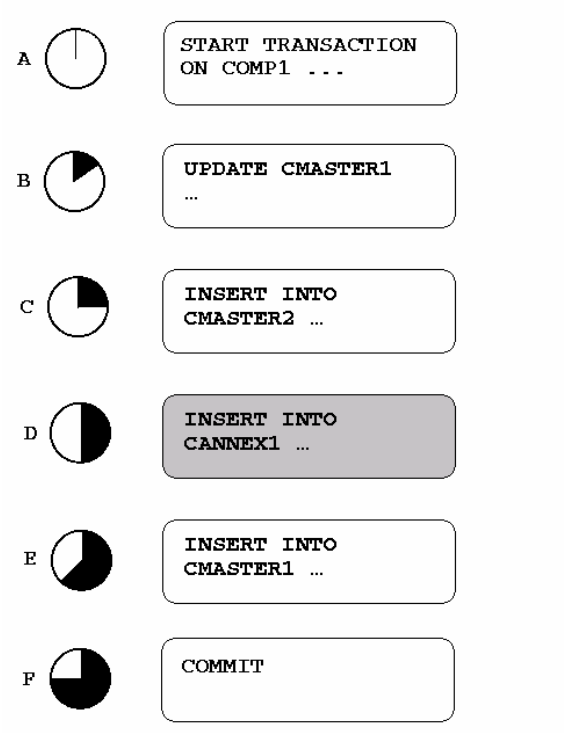

Figure 6. A transaction passing through temporary inconsistent states

At step D, the detail data is inserted into annex CAnnex1 before the master data is inserted in CMaster1. CAnnex1 has a one-many relationship with CMaster1. For this reason, the state after step D is temporary inconsistent. Finally the transaction is committed successfully, the master class of this one-many relationship being updated at step E, just before the transaction commit.

11. Practical Considerations

Let's consider an example of a very simple database containing information about products, customers and their orders. Designing its schema can be easily done in a component approach by assembling three ready-made components, if they are available: Orders, Products and Customers. Considering the above three components as being independent, two adaptors will be required for interconnecting them: Products - Orders and Customers – Orders (as presented in Figure 7).

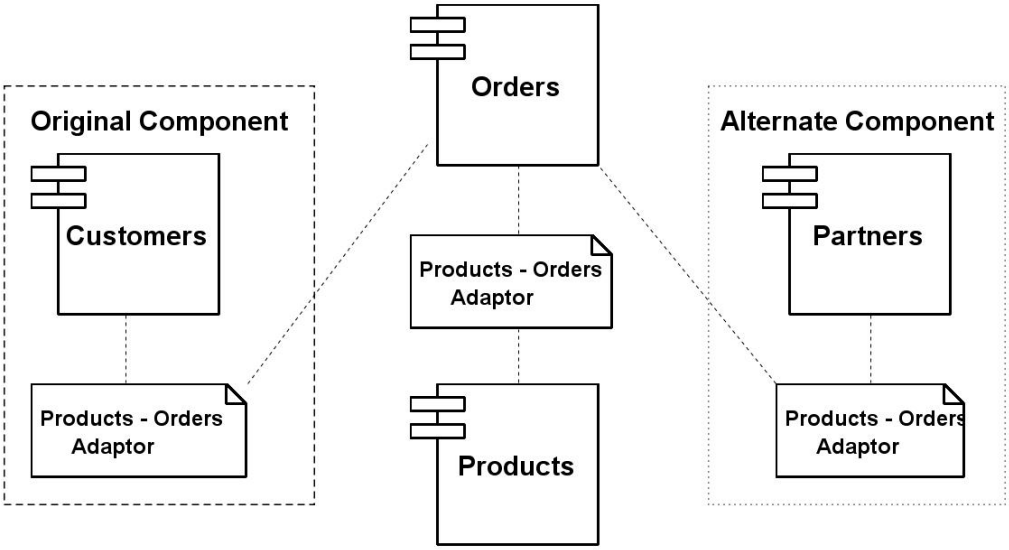

Figure 7. Example of Database Schema Composed from Components

If the business model keeps changing, the structure of the database can be easily adapted to support the new requirements. Suppose that the organization is growing, and apart from customers other types of business relations will appear. The Customers component can be replaced with one having a much larger spectrum, in this case Partners. This possible replacement is also depicted in Figure 7.

In order to be reusable, a component needs to be extendable. For example, in case of the Customers component mentioned above, no matter how hard the designer will try to cover all the aspects, situations in which the component will not fully cover the business model will still occur. In order to achieve a high degree of extend-ability, our proposal is to add a new annex to the component, related to its main entity and containing the excess details needed by each particular business.

This excess detail annex can be in a one-one relation with its master, and contains a LOB column in which the details are stored as an XML or INI-style file, or can be in a more classical one-many relation, every excess detail having its own entry. Since storing and retrieving LOB columns will highly impact the performance such an approach is not recommended. However, because the type and size of an excess detail cannot always be estimated when designing the system, in some situations the performance trade-off represented by the extra LOB column is acceptable. Using a table or a class containing parameter name, type and value as storage for excess details introduces a processing overhead caused by the type conversions, which in most of the cases will be lower compared with the overhead introduced by using LOBs.

In case the database component is also doubled by a software component providing access services to it, if the user requires a certain detail that is not part of the design, then the software component will check the excess details and will pass the information to the caller or will raise an “information not found” exception.

Designing a database in component terms introduces an overhead of extra relations that puts its mark on performance. The performance trade-off is mainly induced by the join operations that are needed in order to consolidate data. This is why a component-based modeling approach is not recommended in real-time applications or in case of applications having very strict performance constraints.

We have run performance tests in order to compare the component database schema described above with its classical equivalent, having no redundant entities for interconnection and extend-ability. The tests were run on different data volumes of up to 100 MB. In order to cover all the usage scenarios, three rounds of testing were performed: a retrieve-intensive round (results presented in Figure 8), an update-intensive round (Figure 9) and a balanced one (Figure 10).

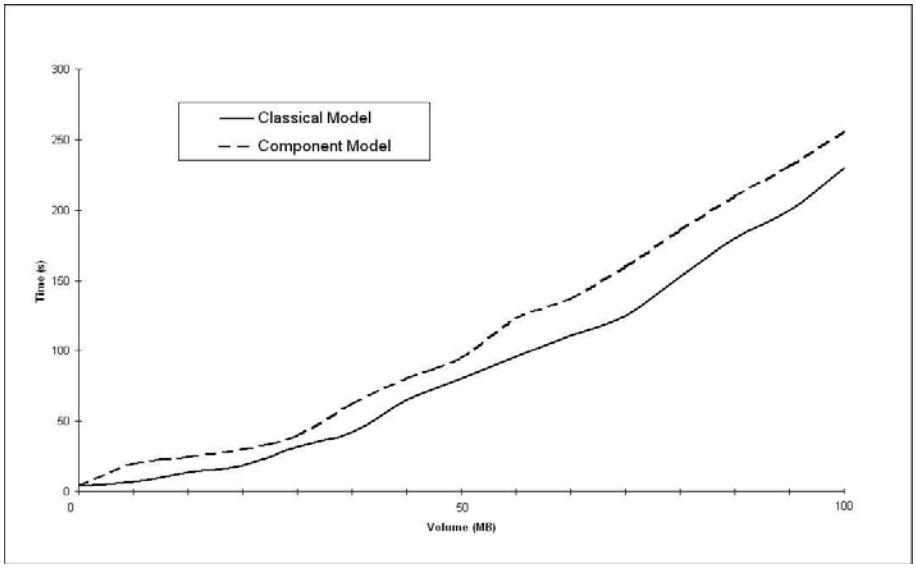

Figure 8. Retrieve-Intensive Operations

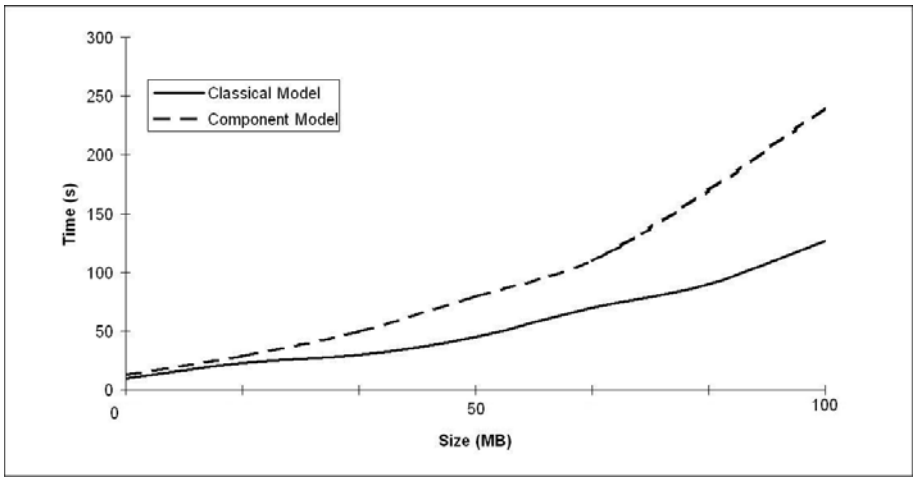

Figure 9. Update-Intensive Operations

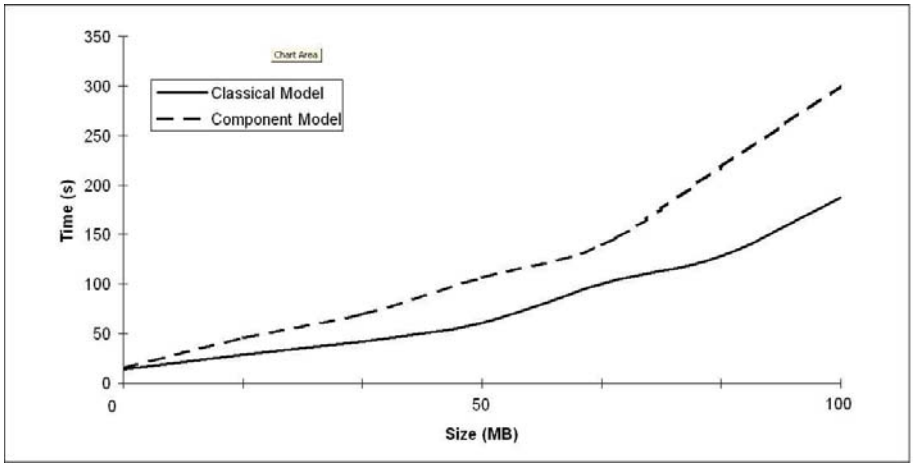

Figure 10. Balanced Operations

As shown in the graphics, the difference in performance becomes considerable for big volumes of data in case of update-intensive operations (inserts and updates). Also, in case of balanced update and retrieve operations the difference in performance is still important.

The conclusion drawn from the performance measurements is that the component model for databases is good performance-wise if the retrieve operations are preponderant. It is required to find a solution to improve the performance of the model in case of update intensive operations.

12. Towards General Applicability Metrics for Software Components

We believe that the degree of compose-ability metric for database components defined by us in [2] and also presented here in section 9, is possible to be used for software components. Similarly, the compose-ability degree of a software component can be qualitatively defined by studying the parameters and return values of its interface methods. A software component interfaced only by methods with no parameters and no return value has the biggest compose-ability degree because it does not have any external data dependencies. From the same perspective, software components which have only methods with no parameters in their interface, but which return a value, can be considered to have a lower compose-ability degree, while components that have methods with both parameters and return values in the interfaces have the lowest compose-ability degree. The multiplicity of a component C (μ_C) is defined as the sum of the multiplicities (m_i) of its interface methods (1).The multiplicity of an interface method M of a software component (m_M) is the sum of its return and signature multiplicities:

$$m_M = c_r \cdot m_r + c_s \cdot m_s \tag{9}$$

Where:

- m_r is the return multiplicity,
- m_s is the signature multiplicity
- c_r and c_s are constants.

The return multiplicity is zero when the method does not return anything and one when the method has a return value:

$$m_r = \begin{cases} 0, \ no\ return\ value \\ 1, return\ value \end{cases} \tag{10}$$

The signature multiplicity is the sum of the multiplicities of the method's parameters:

$$m_s = \sum_{i=1}^{k} p_i \tag{11}$$

Where:

- p_i is the multiplicity of a parameter,
- k is the total number of parameters.

Since references and pointers are possible external data dependencies, their multiplicity is considered to be heavier than the multiplicity of a parameter passed by value:

$$p = \begin{cases} 1, value \\ r, reference / \ pointer \end{cases} \tag{12}$$

where r > 1.

(1), (9), (10), (11) and (12) define the multiplicity of a software component. Since the multiplicity of a software component is measurable, the compose-ability metric defined in [2] for database components can now be applied for general software components:

$$\sigma_C(\mu_C) = \lim_{x \to \mu_C} \frac{\varepsilon}{\varepsilon \cdot x + 1} \tag{13}$$

Where $\varepsilon \in (0,1]$.

13. Conclusions

Defining the design of a database in terms of components allows their reuse. The main benefits of CODB are: clear separation of the functional requirements, increased efficiency in application development and shorter time-to-market. This also offers a solution for data storage standardization. The use of the above described component model for database systems allows a smooth information exchange between organizations using the same components. It gives the possibility of creating a library of ready-made database design components and patterns that can be composed in order to create a database schema.

Proposed by us as a new abstraction level for database systems, components are the needed support for enriching the transaction processing mechanism by adding a new property for transactions: scope. A transaction can now start at component level, escalating only if required. An application will be able to run at the same time two transactions having as scope disjunctive components, increasing in this way the parallelism.

The component model introduced in [1] and [2] and refined here is flexible enough to be applied not only to object-oriented databases, but to relational and object-relational database systems design as well.

Adding the component paradigm to the world of the OODB does not affect the way both structural and logical integrity are maintained. Components become the integrity, coherency and consistency units for databases, having the integrity constraints encapsulated within. There are now two levels of integrity checks: one at the component level, checking the local relations, and a global one, checking the interactions between components.

Introducing the components into the picture as the fundamental unit of consistency for transactions improves the way the potential conflict areas can be detected in case optimistic mechanisms for maintaining consistency of the database and controlling the concurrency are used. They proved to be the best choice in the case of an OODB.

CBSD is more and more used today for the development of large-scale applications and therefore system integrators need to choose between different components providing the same functionality. Choosing the right components when integrating an application brings important savings in time and cost.

From both qualitative and quantitative perspectives, we defined the degree of compose-ability of a database component by studying the multiplicity of its interface relations. In our effort to address the market needs for uniform metrics and methods of assessment of component characteristics like adaptability and compose-ability, we extended the compose-ability metric defined for database components to software components.

References

1. Octavian Paul Rotaru, Marian Dobre, *Component Aspects in Object Oriented Databases*, Proceedings of the International Conference on Software Engineering Research and Practice

(SERP'04), Volume II, ISBN 1-932415-29-7, pages 719-725, Las Vegas, NV, USA, June 2004.

2. Octavian Paul Rotaru, Marian Dobre, Mircea Petrescu, *Integrity and Consistency Aspects in Component-Oriented Databases*, Proceedings of the International Symposium on Innovation in Information and Communication Technology (ISIICT'04), pages 131-137, Amman, Jordan, April 2004.

3. Octavian Paul Rotaru, Marian Dobre, Mircea Petrescu, *Reusability Metrics for Software Components*, to appear in the Proceedings of the 3rd ACS / IEEE International Conference on Computer Systems and Applications (AICCSA-05), Cairo, Egypt, January 2005.

4. *AspectC++ Home Page*, http://www.aspectc.org

5. *AspectJ Home Page*, http://www.aspectj.org

6. Bertrand Meyer, *Object-Oriented Software Construction* (2nd Edition), Prentice Hall, 2000.

7. Bjarne Stroustrup, *Why C++ is not just an Object-Oriented Programming Language*, Addendum to OOPSLA95 Proceedings, ACM OOPS Messenger, October 1995.

8. C. Pfister and C.Szyperski, *Why Objects Are Not Enough*, Proceedings of Component Users Conference, Munich, Germany, 1996.

9. Clemens Szyperski, *Component-Oriented Programming – A Refined Variation on Object-Oriented Programming*, The Oberon Tribune, Vol. 1(2), 1995.

10. Clemens Szyperski, *Component Software – Beyond Object-Oriented Programming*, ACM Press/Addison-Wesley, 1998.

11. D. Garlan, *Software Architecture*, Proceedings of the 22nd International Conference on Software Engineering (ICSE-00), ACM Press, pages 91-102, June 2000, New York, USA.

12. Dirk Riehle, *Framework Design: A Role Modeling Approach*, Ph.D. Thesis, No. 13509, Zurich, Swiss, ETH Zurich, 2000.

13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

14. Eun-Sun Cho, Sang-Yong Han and Hyoung-Joo Kim, *A New Data Abstraction Layer Required For OODBMS*, Proceedings of 1997 International Database engineering and Applications Symposium (IDEAS'97), pages 144-150, Montreal, Canada, August 1997.

15. Frank Buschmann, et al., *Pattern-Oriented Software Architecture*, John Wiley & Sons Ltd., 1996.

16. Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom, *Database Systems – A complete book*, Prentice Hall, 2002.

17. Jens Palsberg and Michael I. Schwartzbach, *Object-oriented Type Systems*, John Wiley and Sons Ltd, 1994.

18. Paul Clements, *From subroutines to subsystems: Component-Based Software Development*, Allen Brown, Ed., Component-Based Software Engineering: Selected Papers from Software Institute, pages 3-6, 1996.

19. Pedro J. Clemente and Juan Hernandez, *Aspect Component Based Software Engineering*, Proceedings of The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), March 2003.

20. Rashid A. and E. Pulvermueller, *From Object-Oriented to Aspect-Oriented Databases*, Proceedings of 11th International Conference on Database and Expert Systems Applications (DEXA), 2000, Springer-Verlag LNCS, Volume 1873, pages 125-134.

21. Rashid A. and P. Sawyer, *Aspect-Orientation and Database Systems: An Effective Customisation Approach*, IEE Proceedings – Software, 2001, Volume 148, pages 156 – 164.

22. R. Green, A. Rashid, *An Aspect-Oriented Framework for Schema Evolution in Object-Oriented Databases*, 1st Workshop on Aspects, Components and Patterns for Infrastructure Software (held with AOSD 2002).

23. Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, *Discussing Aspects of AOP*, Communications of the ACM, vol. 44, no. 10, pages 33-38, October 2001.

24. Wolfgang Keller, *Object/Relational Access Layers – A Roadmap, Missing Links and More Patterns*, Proceedings of the Third European Conference on Pattern Languages of Programming and Computing, Bad Irsee, Germany, July 1998.

25. Xiaong Lu, J. Wenny Rahayu and David Taniar, *ODMG Extension of Composite Objects in OODBMS*, Proceedings of the Fortieth International Conference on Tool Pacific: Objects for internet, mobile and embedded applications – Volume 19, pages 133-142, Sydney, Australia, 2002.

26. Gregory McFarland, Andres Rudmik, and David Lange *Object-Oriented Database Management Systems Revisited – An Updated DACS State-of-the-Art Report*, Modus Operandi Inc., January 1999.

27. Andrew C. Myers, *Resolving the Integrity / Performance Conflict*, MIT Laboratory for Computer Science, 1993.

28. J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, in Michael Stonebraker *Readings in Database Systems*, 3rd edition, Morgan Kaufmann Publishers, Inc., 1998.