Encapsulation or Availability

On the Combination of Objects and Relations in Systems Development

Pär J. Ågerfalk
pak@esa.oru.se
Dept. of Informatics (ESA)
Örebro University, SE-701 82 Örebro, Sweden

Abstract

This paper addresses the issue of combining object-orientation with relational database technology. It is an attempt to clarify the difficulties faced by developers when such a combination is used. The object-oriented model and the relational model, and their ontological assumptions are investigated and different approaches to combining them are described and elaborated. The discussion is mainly carried out at a conceptual level, although related technological issues are mentioned where appropriate.

Keywords: Object-orientation, relational databases, systems development, RAD-tools, object-relational databases.

BRT Keywords: FA, FC, CB06

Introduction

Nowadays it is quite common to perform enterprise modelling as well as analysis and design of information systems by use of object-oriented methodology (e.g. Jacobson *et al.*, 1995). An often-used argument for such an approach is that object-orientation is supposed to imply a transparent and seamless transformation from analysis, through design to technical implementation. Seamlessness is supposed to be achieved by recognizing that the objects identified during analysis are the same objects that exist in the produced software system. If systems are built with "true" object-oriented databases, i.e. by use of what Cattell (1994) refers to as *object-oriented database programming languages*, e.g. Objectstore (Object Design, 1990), Gemstone (Bretl *et al.*, 1988), such seamlessness might be achievable. The problem is that relational technology often serves as the foundation for business information systems. One reason for this is that relational databases constitute a well-tried and reliable technology that is supported by a vast number of popular development tools. Another problem is that new systems often need to exchange information with legacy systems that are constructed using relational databases.

Nevertheless, a new software client application might be constructed according to the principles of object-oriented programming and design, neglecting the underlying data structures at early stages of development. An example of this is the popularity (or even domination) of object-oriented tools for graphical user interface programming. This leads to the fact that developers need to change perspective at some point in the development lifecycle—from an object-oriented worldview to a relational—or at least manage to combine the two. Recent approaches to multi-tier architectures and component-based development do not seem to solve this necessary shift in perspective. Hence, developers must both understand, and be able to cope with, the differences in the two models in order to construct information systems that make use of both. This issue ought to be relevant with respect to most visual object-oriented and object-based tools (RAD-tools), such as Borland Delphi, MS Visual Basic, *et cetera,* as well as to more traditional object-oriented development with, for example, Smalltalk, C++ or Java.

Why, then, is this a problem? Is it the case that there are foundational differences between the two models' perspectives and underlying assumptions, which make them incompatible? Or is it just a question of scientific and practice maturity?

The aim of this paper is to describe, elaborate and reflect upon prevalent theories and techniques for combining object-oriented technology with relational technology. The contribution of the paper is threefold. First, it gives an overview of different techniques for combining relational and object-oriented technology, and reflects upon why the two models are hard to combine. Second, it provides a simple conceptual framework for classifying object-oriented systems that might be useful for developers when, for example, integrating new object-oriented systems with legacy relational systems. Third, and perhaps most important, it exemplifies elaboration on the theoretical foundations of technological approaches and in doing so indicates a need for developers to investigate, and ultimately understand, underlying ontological assumptions in order to be successful in practice.

Only the theories (models) as such have been studied and not, for example, different theories of information systems structuring that are closely related to each model (e.g. the relationship between Information Resource Management (IRM) and relational databases).

The paper is organized as follows. Firstly, the models are treated independent of each other. Secondly, different techniques for combining them are discussed. Finally, some conclusions and reflections are given.

The work has been performed mainly through studies of the literature. The analysis is based on what each source has emphasized and is qualitative in nature. The meta-modelling approach MA/SIMM (Goldkuhl and Fristedt, 1995) has been used as an aid during analysis and structuring of the results.

The Relational Model

This section elaborates on the relational model (RM) of data, which originates from the work by Codd (1970). The treatment is far from complete—only concepts that are of importance for this paper are presented and discussed. The chapter is based on Elmasri and Navate (1994) where not otherwise stated.

Goals and Perspective of the RM

The main goal of the RM is to offer a theory of databases, based on a solid mathematical foundation. Hence, inherent in the model is a formal and mathematical perspective. The mathematical theory underlying the RM is relational theory. Since mathematical relational theory, in turn, is based on set theory, the relational model is heavily set-oriented. The idea of the model is to decompose data into its smallest parts and to separate data completely from functionality. Strictly, functionality is not even a part of the formal model. However, when the relational model is applied, a main assumption is that data is stable and persistent, whilst the use of data is contingent and volatile. Availability and semantics-free representation of the data is therefore of great importance. Another important issue is *data independence*, i.e. allowing changes in data organization without affecting application programs (Cattell, 1994). In the RM, data are traditionally viewed at three distinct levels, referred to as the *three-level architecture* for database systems (Tsichritzis and Klug, 1978). The three levels are: 1) internal (or physical): addressing the organization of data on a physical storage medium; 2) conceptual: addressing entities and relationships; and 3) external: addressing the interpretation of data by an application.

With such a three-level architecture, data independence means that 1) the representation at the conceptual level can be changed without affecting the external level, and 2) the representation at the internal level can be changed without affecting the conceptual level, and hence without affecting the external level. This means, for example, that entities may be added without affecting existing applications, and that the physical organization of files may be changed without affecting the conceptual model, and hence without affecting applications relying on that model.

Concepts of the RM

The four most central concepts of the relational model are, according to Elmasri and Navathe (1994): the *domain*, the *tuple*, the *attribute* and the *relation*. As indicated in the previous sub-section, the concept of relation originates from mathematics. As in mathematics, a *relation* $R \subseteq S_1 \times S_2 \times \ldots \times S_n$ is a subset of the Cartesian product of one or more sets. A relation is thus a set of *n*-tuples consisting of elements from the *n* sets participating in the relation. In the RM, such sets, with atomic elements, are referred to as *domains*. An *attribute* is the name of the role a certain domain plays in a given relation. This way, a relation can be used as a data structure representing a number of related properties that together describe some aspects of some entity (concrete or abstract). A relation might, for example, be used to describe relevant properties of people. Each single person is then represented by a tuple in the relation and, for example, (Sam, 28, Male) could be the representation of a person Sam who is a 28-year-old male. A relation can thus be viewed as the set of all tuples belonging to that relation. The term *table* is often used to talk about relations. Date (1991) clarifies this terminology by stating that "relation" is an abstract theoretical concept, while "table" is its concrete representation.

For example, assume a relation *Persons* = {(Peter, 25, Male), (Maria, 23, Female), (Linda, 24, Female)} that represents three people. *Persons* is thus defined as being a subset of the Cartesian product of the sets of all possible names, ages and sexes. This relation can be expressed as a table showing both tuples and attributes, as shown in Table 1.

Name	**Age**	**Sex**
Peter	25	Male
Maria	23	Female
Linda	24	Female

Table 1: A relational table describing persons.

The concept of relation is also used to relate relations (describing entities) to each other. Suppose that the problem domain also contains cars, described by the relation *Cars* $\subseteq$ RegNo $\times$ Colour, which are owned by people. It is now possible to define a relation

Owns $\subseteq$ *Persons* $\times$ *Cars*, such that *Owns* = {(Peter, ADB123), (Linda, FEK456)}. This relation, from *Persons* to *Cars*, is thus describing the circumstance that Peter owns a car with registration number ADB123, etc.

An important aspect of the RM is the concept of the *integrity rule*. There are two different integrity rules: *entity integrity* and *referential integrity*. The entity integrity rule states that each tuple of a relation must be uniquely identifiable. That is, there must exist a combination of one or more attributes that uniquely identifies each tuple in a relation. The collection of such uniquely identifying combinations of attributes of a relation is referred to as the relation's candidate keys, of which one is selected as the *primary key*. The referential integrity rule states that, when a relation relates relations, a *foreign key* must always have a value that is to be found as a candidate key in the related relation or be NULL. The value of a foreign key is thus selected from the same domain(s) as a candidate key (usually the primary key) in the related relation. In the example, the attribute Name is used as primary key in the *Persons* relation and RegNo in the *Cars* relation. The entity integrity rule thus implies that there can be no more than one Peter, no more than one Linda, *et cetera*. The referential integrity rule, in turn, implies that a tuple (Jane, STA789) cannot be a part of the relation *Owns*, since there is at present no tuple in *Persons* identified by the name Jane. On the other hand the tuple (Maria, NULL) $\in$ *Owns* is conformant to the rule, stating that Maria has no car. Note that this assumes that the first attribute of the relation *Persons* (i.e. Name) uniquely identifies the tuples of that relation (as stated above), otherwise this would violate the entity integrity rule since part of the primary key would be NULL. Furthermore, another tuple in *Owns* could be (Peter, NEK010), describing that Peter owns two cars.

Another central concept of the RM is that of a relation's *normal form*. The normal form concerns how attributes of a relation functionally depend on each other. An often-used rule of thumb is that each relation should be in, at least, the third normal form. If in third normal form, each attribute of the relation is functionally dependent on the key, the whole key and nothing but the key. The main purpose of *normalization* is to reduce redundancy in the data representation, or as Connolly *et al.* (1999, p. 221) put it: "*Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of an enterprise.*"

Figure 1 shows how these concepts are related to each other (the arrows in the figure are used to indicate intended direction of reading).

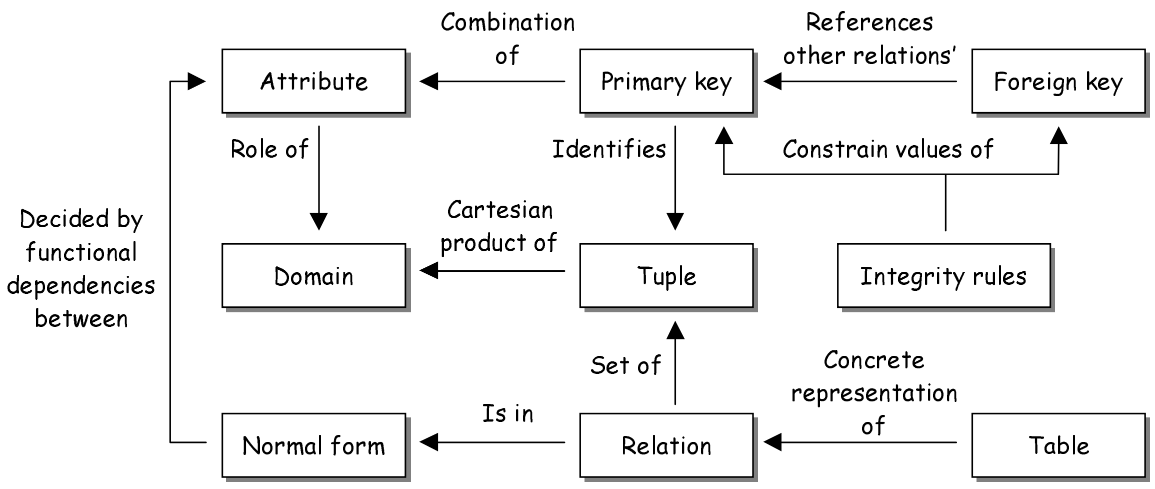

Figure 1: Central concepts of the RM.

Relational Databases and Query Languages

Any database that is founded on the RM is referred to as a *relational database*. Thus, a relational database supports definitions of domains and relations as well as the implementation of integrity rules. Most relational databases are defined and manipulated by use of the query language SQL (Structured Query Language). SQL is a (almost) declarative language based on the relational calculus, which is a mathematical theory for manipulation of relations. Declarative means that a programmer is allowed to express what is to be done without specifying how. Instead, query optimization and construction of the actual algorithm used to fetch data from the database are performed by a relational database management system (RDBMS). A query is formulated as logical statements in SQL and the system returns data that match the query. A query might concern several tables and is not necessarily pre-specified (compiled). This facilitates formulation of so called *ad hoc* queries, useful when information requirements change in the supported business.

Summary of the RM

The basic ontological assumptions of the RM seem to be that information can be represented as data and that data can be decomposed and normalized into atomic terms constituting domains. How these terms are actually used to combine information is an application-specific property and consequently not part of the model *per se*.

To summarize, the relational model:

- Is theoretically founded on relational theory and set theory.
- Implies decomposition of data into its smallest constituents.
- Separates data from functionality.
- Emphasizes flexibility and availability of data.

The object-oriented model

This section elaborates on the object-oriented model (OOM). As in the previous treatment of the relational model, only concepts that are of importance for this paper are presented and discussed.

Goals and perspective of the OOM

Object-orientation lacks a rigorous theoretical foundation corresponding to the relational theory of the RM. Instead, practitioners have forced the development of object-orientation, which has resulted in a vast number of different pragmatic solutions (Britts, 1997). The most related theory is probably that of Langefors (1966), since a sharp distinction is made between systems' (objects') internal and external properties (cf. Langefors, 1995). Another similarity with Langefors' theory is the top-down approach of decomposing systems into sub-systems.

The OOM encompasses a strong descriptive perspective. That is, the objects of the system shall have counterparts in the problem domain (the business) that the system describes. This perspective might be related to the fact that the OOM originated in object-oriented programming, which in turn has been influenced by the language Simula, a language specially designed for simulations (Dahl *et al.*, 1968). Thus, the software system is viewed as a simulation of reality. It is important to notice that the descriptions (system counterparts) of real world occurrences correspond to both data and functionality (sometimes referred to as knowledge and behaviour).

Increased reuse is often stressed as an advantage of object-orientation. The argument is that since the software objects correspond to real world occurrences, these ought to be reusable in all systems within the same domain (e.g. Sims, 1994).

Concepts of the OOM

It is quite common to talk about the “three cornerstones of object-orientation”: *encapsulation, inheritance* and *polymorphism* (e.g. Olsson, 1991). Encapsulation means that data and functions operating on those data are bundled together to form coherent objects. Data can thus be protected from misuse, but are also hard to access and restructure if information requirements change. Inheritance means that adding new data or functionality to existing object-types (*classes*) can be used to create new ones, co-existing with the originals. The new class (the *sub-class*) thus inherits the properties of the existing class (the *super-class*). Note that this is not inheritance in the legal sense when an ancestor dies, but in the sense that a son, for example, inherits some physical attributes of his father without physically removing them from him. In the OOM, a system is viewed as a set of collaborating objects. Objects sending messages to each other, and thus synchronizing their behaviour, constitute such collaboration. Polymorphism means that every given object (*instance*) of a sub-class understands messages intended for its super-class, and hence is able to receive and react upon them. The communication of messages is usually implemented as method calls, i.e. the sending object calls one of the functions associated with the receiving object.

Figure 2 shows how these concepts are related to each other (similarly to Figure 1, the arrows in the figure are used to indicate the intended direction of reading).

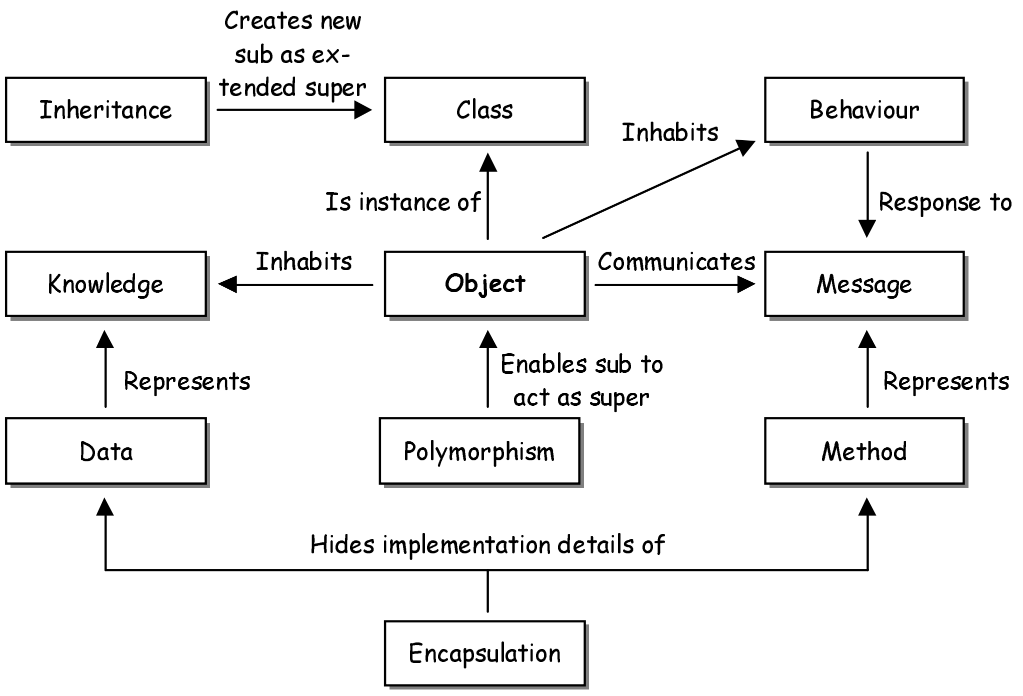

Figure 2: Central concepts of the OOM.

Object databases and Query Languages

There are two different types of object databases (Cattell, 1994). The first type is usually referred to as an *object-oriented database* or *object-oriented database programming language*. An Object-oriented database is basically a programming language (usually C++ or Smalltalk) that has been extended with database functionality. By explicitly stating in the source code that an object shall be persistent, its data is stored in the database. Storing and retrieving of data is then handled by an object-oriented database management system (OODBMS). This way, persistent objects can be treated in the same way as transient objects and referenced by using traditional pointers. Conversions between main memory addresses and disk representations are thus handled transparently by the OODBMS.

A problem of object-oriented databases is the lack of declarative query languages. Instead, queries must be coded in the particular programming language used, without explicit support from the system. Another problem is that of insufficient performance of current object databases for typical business applications. This problem is partly related to the previous in that *ad hoc* queries, common in business information systems, are hard to support efficiently. However, Cattell (1994) provides evidence based on benchmarking that an OODBMS might perform as well as, or even better than, an RDBMS for engineering applications such as computer aided design (CAD), although he admits that the results can be attributed to architecture-based, rather than model-based, differences.

The second type of object database is the *object-relational database*, also referred to as the *hybrid database* (Berild, 1996) or the *extended relational database system* (Cattell, 1994). Such a database is founded on the relational model but has several object-oriented extensions, e.g. support for object identifiers, inheritance, and methods on objects. To support object-relational databases, new generations of SQL (SQL3 and SQL4) have been developed (Connolly *et al.*, 1999).

Stonebraker (1990) shows the relation between different kinds of database systems as shown in Figure 3.

	Simple data	Complex data
Query	Relational DBMS	Object-relational DBMS
No query	File system	Object-oriented DBMS

Figure 3: Classification of database systems according to Stonebraker (1990).

Summary of the OOM

The basic ontological assumption of the OOM seems to be that things (entities), possessing knowledge and behaviour, constitute the world. Consequently, there is no point in decomposing information into smaller atoms than those corresponding to such things, since the behaviour is related to the thing and not to the atomic data. To summarize, the object-oriented model:

- Lacks a theoretical foundation, which has led to pragmatic solutions.
- Implies decomposition of systems into sub-systems where each sub-system corresponds to a simple concept in the problem domain.
- Treats data and functionality as non-separable wholeness.
- Encourages control and encapsulation of data.

Techniques to Combine Objects and Relations

In this section some prevalent techniques and approaches to combining objects and relations are presented and elaborated.

Using both a relational and an object-oriented database

One obvious way to obtain the advantages of both object-oriented databases and traditional relational databases would be to use one of each. An object-oriented database could then be used for object-oriented systems and a relational database for *ad hoc* queries and legacy systems. However, to manage this, replication is needed to maintain consistency between the two systems, which might be embedded in object- or component-oriented middleware. This does not, however, solve the problem of perspective change.

Mapping classes to relations

If an object-oriented model is to be represented in a relational database, a conversion to a relational database schema must be performed. The easiest way to perform such a conversion is to let each class be represented by a table, i.e. a one-to-one mapping. The data members of the class thus become attributes of the corresponding table. In some cases, the resulting tables must then be further normalized. When the object-model consists of inheritance, there are some alternative designs that must be considered (cf. Blaha and Premerlani, 1998). One approach is to map an entire inheritance hierarchy into a single table. Such a solution is straightforward, but yields many NULL values in tuples representing super-class objects. Another approach is to create one table for each class and let sub-class tables contain only attributes introduced at that level in the hierarchy. In the latter case, all tables representing the inheritance hierarchy must have the same attribute as primary key to enable simple joins. An alternative to the latter approach is to “push down” super-class attributes and let each sub-class table consist of all attributes belonging to that class, including inherited ones. In all cases, some inheritance semantics are lost in the transformation and must be handled by an application or, for example, as stored procedures or triggers. The same counts for the methods of the classes, whether inherited or not.

Architectural Styles

It is possible to distinguish two main principles for the design of object-oriented systems.

The first principle is the one used by most 4GL tools. It is characterized by the use of an object-oriented interface between the application and the relational database (cf. “Microsoft Data access objects” subsequently). With this architectural style, the objects corresponding to the problem domain (domain objects for short) are not represented in clients' address spaces at all. Instead, the objects that execute in the clients' address spaces represent the database's tables, stored procedures etc. These objects are, typically, manipulated by the same functions and event handlers that are used to control the graphical user interface (GUI), e.g., with Borland Delphi the GUI objects are directly connected to these object-oriented representations of the database.

The second principle is to handle the domain objects in the application and maintain a mapping to these objects' representations in the database. How the mapping is implemented depends on the kind of database being used. As mentioned in section 3, with a pure object-oriented database the mapping is handled transparently by the OODBMS. If a relational database is used, on the other hand, the objects in the client must be able to fetch information about themselves from the database. An object-oriented interface encapsulating the tables of the database might also be used for this purpose (cf. the section "Object interfaces" subsequently).

As mentioned above, an alternative to relational databases and object-oriented databases is the object-relational database. With an object-relational database some of the domain objects, whole or in part, might execute in the database or in the clients' address space. However, the problem of mapping programming language objects to their query language representation, usually referred to as *impedance mismatch* (Cattell, 1994), exists with the use of either relational or object-relational databases.

Microsoft Data access objects

As an example of encapsulation of the relational database structure, we present Microsoft's "Data access objects" (DAO), used by, for example, MS Visual Basic and MS Visual C++. Our description is brief, aiming to show the basic principles. The description is taken from "Database developers guide with Visual Basic 4" (Jennings, 1996) and is based on version 3.0 of DAO. The data access objects are used by data-aware user interface components to directly connect themselves to the underlying data. This is the same architectural style used by, for example, Borland Delphi, although the actual designs differ.

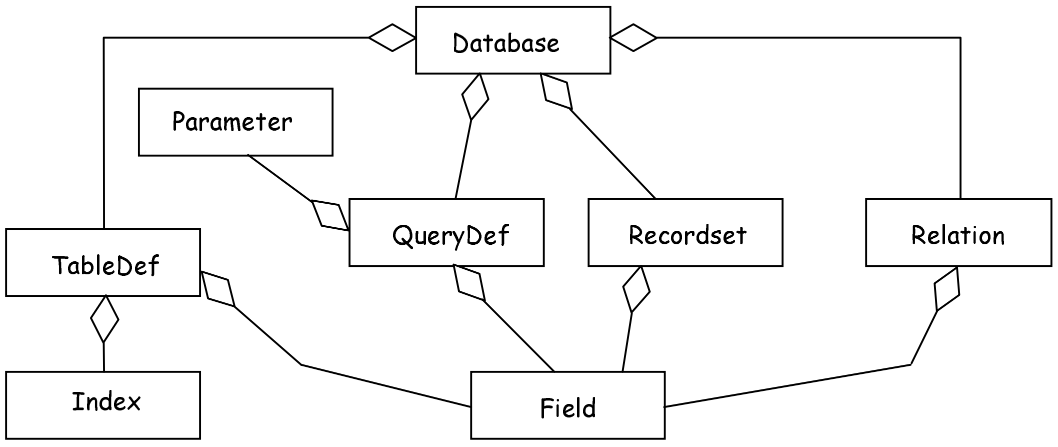

Figure 4: The structure of (part of) Microsoft DAO 3.0.

Figure 4 shows the structure of the most important parts of DAO expressed as a UML class diagram (the diamonds denote aggregates). For each class, there is also a corresponding collection, i.e. a class representing the set of objects of the class. For the class Recordset there is, for example, a corresponding class Recordsets. The meaning of the most important classes is shown in Table 2.

To query the database, objects of the class Recordset are used. Each Recordset object is based on an existing Tabledef object or Querydef object and is used to browse the resulting set of records (tuples). Optimization of network traffic is built into the model because a given tuple is not physically retrieved from the database until the corresponding row in the Recordset object is actually used.

Tabledef	The definition of a table with fields (attributes) and possibly indexes.
Querydef	Stored (precompiled) queries or procedures.
Recordset	A table or answer of an SQL query, which can have parameters.
Relation	Referential integrity between two tables.
Database	The whole database.

Table 2: Some of the most important classes of Microsoft's DAO 3.0.

It is worth noticing that what DAO offers is merely object-oriented handling of the primitives in the relational model. A domain object that has been distributed across many tables, perhaps due to normalization, will thus not constitute a single DAO object.

Object interfaces

There are many possible designs to maintain a mapping from objects in a client to tables in the database. The OOA method by Coad and Yourdon (1991) recommends a dedicated database object to be created for each persistent domain object. The database objects are then responsible for retrieving and storing their domain objects' data in the database by use of, for example, SQL. Agarwal and Keller (1998) present an object interface architecture where each domain object (called a business object) communicates with one or more table objects, each representing a row in a database table. Such an architecture can, of course, be combined with that of Coad and Yourdon and is actually an application of encapsulation of the relational database structure as described previously, though not necessarily based on DAO.

A problem with these kinds of mappings (sometimes referred to as *procedure interfaces*) is that DBMS support is not optimally utilized. It is the application that must handle, for example, query optimizing and transaction processing. An obvious risk with this approach has been referred to as *the server trap*, i.e. that an object-oriented DBMS is built from scratch in every new project.

Multi-tiered architectures

One approach to overcoming some of the obstacles with mappings from object-orientation to relational databases is to use a *"Multi-tiered client/server architecture"* (MTCS). An example of MTCS, using three tiers, is that used by a Swedish company developing systems for the financial market, shown in Figure 5 (Björnstedt, 1997).

In this three-tier architecture, clients (in the client services tier) use an object-oriented database that takes care of transactions and maintains an object-cache (in the business services tier). The OODBMS acts as a mediator and retrieves actual data from a relational database (in the data services tier). One advantage of this architecture is that client programs need not bother with transactions since these are handled by the OODBMS. Another advantage is that code (objects) can be transferred between clients and server for performance optimization, if the same language (typically Java or Smalltalk) is used in both tiers. The object cache also minimizes the need for frequent (and time consuming) SQL joins. The relational database can be changed to an object-relational database if needed. With a competent OODBMS, data might actually be fetched from any source, for example web servers, and be presented as objects to the clients.

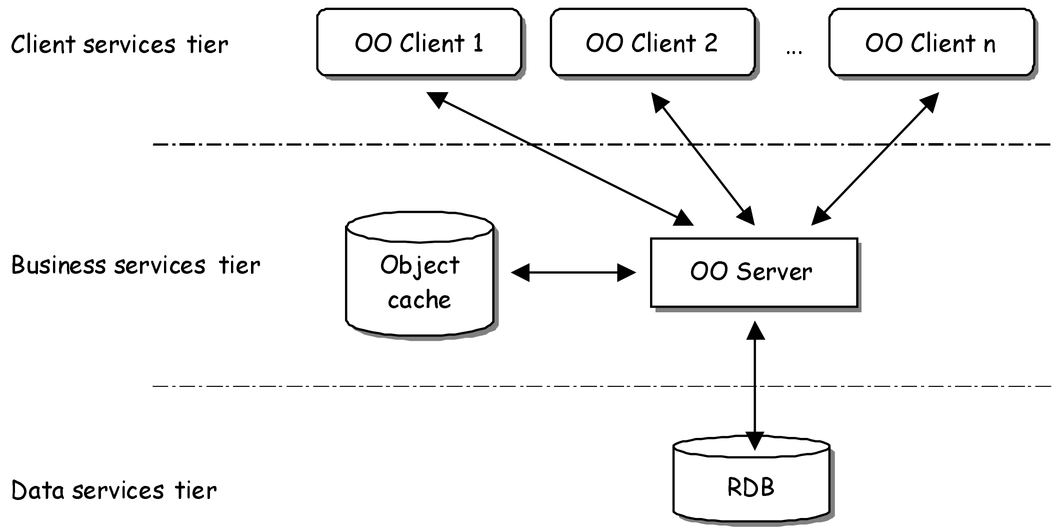

Figure 5: A three-tiered architecture (Björnstedt, 1997).

Conclusions

This paper has elaborated on the problem of mapping an object-oriented analysis to a relational design. It has been shown that such mappings can be performed in two primarily different ways, yielding the classification of object-oriented systems shown in Figure 6.

The first approach is to use an object-oriented interface to a relational database (i.e. the objects model the RM, not the problem domain). That is, to not implement the domain objects (business objects) in the client, represented by the "No domain objects in clients" leaf of Figure 6. Instead, data and functionality are separated; with data in the database and functionality distributed across user interface objects. This is a simple and convenient solution with an environment such as MS Visual Basic. At the same time, it is far from optimal from an object-oriented point of view. This is the most common approach in current rapid development tools, which typically handle the RDB interface transparently by letting developers use predefined database components.

The second approach is to let domain objects execute in clients' address spaces and maintain mappings to their representation in the database, represented by the "Domain objects in clients" leaf of Figure 6. A problem with such an approach is that it abandons the support given by client tools' database components (e.g. Borland Delphi), as mentioned above. This in turn leads to increased complexity and time consumption during technical implementation. Another implication is that it does not take full advantage of the capabilities of the database management system. Developers must, for example, implement their own query optimizer and cannot use the benefits of a declarative query language such as SQL. The mappings between client domain objects and their database counterparts are typically implemented by the use of some object-oriented encapsulation of the relational database, such as DAO (the same technique as used in the first approach mentioned above).

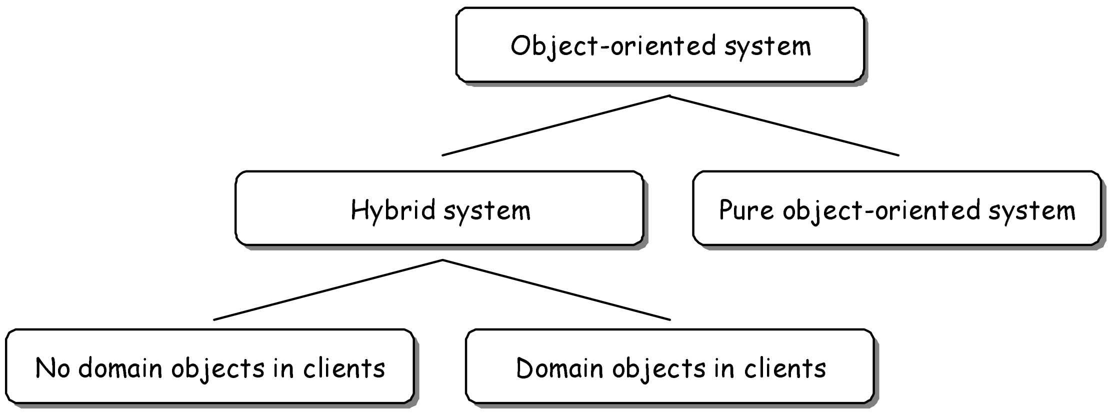
Object-oriented system
Hybrid system
Pure object-oriented system
No domain objects in clients
Domain objects in clients

Figure 6: Classification of object-oriented systems.

Furthermore, the problem is not eliminated by the use of three or more tiers instead of two. Assuming that we have good middleware support to handle communication between clients and application servers, as well as between application servers and databases, we must still deal with separation of data and functionality. Although we have more options when deciding where to put what, we must still make the decision. With a three-tier architecture, the problem is merely pushed down from client development to the development of business objects in the business services tier (executing on application servers).

The lack of a rigorous formal theoretical foundation of the object-oriented model might be a reason for the lack of standardized methods and architectures. Current tools for rapid application development do not seem to be suitable for pure object-oriented development, due to a strong bias towards the relational model.

The mismatch between the models seems to be derived from differences in perspective. The relational model regards information and information processing as two different aspects of the world that should be treated independently. In the object-oriented model these two aspects are non-separable. The relational model encourages availability in order to cope with changing environments. The object-oriented model encourages information hiding and control. The relational model regards atomic data terms as constituents of the world. The object-oriented model views the world as constituted by things inhabiting knowledge (data term aggregates with associated semantics) and behaviour. Although they are different, it is possible to combine them if we are prepared to relinquish some benefits from one in order to gain some from the other.

References

Agarwal S and Keller A M (1998). *Architecting Object Applications for High Performance with Relational Databases*. Persistence Software, Inc. San Mateo, CA.

Berild S (1996). *En introduktion till hybrid-DBMS*. In Swedish. SISU publikation 96:11, rapport – juni 1996.

Björnestedt N (1997). *Hur utveckla objektorienterade system mot en relationsdatabas?* Presentation given in Swedish at the Neotech AB seminar "Seminarium om objektorienterad systemutveckling med relationsdatabaser". Royal Viking hotell, Stockholm, Sweden, 1997-04-22.

Blaha M, and Premerlani W (1998). *Object-oriented modeling and design for database applications*. Prentice Hall, Inc.

Bretl R, Maier D, Otis A, Penney J, Schuchardt B, Stein J, Williams H, and Williams M (1988). The Gemstone Data Management System. In Kim W, and Lochovsky F H (Eds., 1988): *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley. Reading, Massachusetts.

Britts S (1997). *Databaser – historik och framtidsvisioner*. Presentation given in Swedish at the Neotech AB seminar "Seminarium om objektorienterad systemutveckling med relationsdatabaser". Royal Viking hotell, Stockholm, Sweden, 1997-04-22.

Cattell R G G (1994). *Object data management: object oriented and extended relational database systems*. Addison-Wesley publishing company, Inc.

Coad P, and Yourdon E (1991). *Object-Oriented Design*. Object international, Inc. Published by Prentice Hall, Inc.

Codd E (1970). A relational model for large shared data banks. *CACM*, 13:6, June 1970.

Connolly T, Begg C, and Strachan A (1999). Database Systems: Practical Approach to Design, Implementation, and Management. 2nd edition. Addison Wesley Longman Limited, England.

Dahl O J, Myrhaag B, Nyygard K (1968). *Simula 67 Common Base Language*. Norwegian Computing Center. Revised in 1970, 1972 and 1984.

Date C J (1991). *An introduction to database systems*, volume I, 5th ed. Addison-Wesley publishing company, Inc.

Elmasri R, and Navate S B (1994). *Fundamentals of database systems*, 2nd ed. The Benjamin/Cummings publishing company, Inc.

Goldkuhl G, and Fristedt D (1994). *Metodanalys: En beskrivning av metametoden SIMM*. In Swedish. Research report, IDA, Linköping University.

Jacobson I, Ericsson M, and Jacobson A (1995). *The object advantage: business process reengineering with object technology*. ACM-press. Addison-Wesley publishing company, Inc.

Jennings R (1996). *Database developers guide with Visual Basic 4,* 2:nd ed. Sams publishing, USA.

Langefors, B (1966). *Theoretical Analysis of Information Systems*. Studentlitteratur, Lund, Sweden.

Langefors, B (1995). *Essays on Infology*. Dahlbom B (Ed.). Studentlitteratur, Lund, Sweden.

Object Design (1990). *Objectstore Reference Manual*. Object Design, Inc. Burlington, Massachusetts.

Olsson I (1991). *Fortsätt med Pascal*. In Swedish. Liber, Stockholm, Sweden.

Sims O (1994). *Business Objects – delivering co-operative objects for client-server*. McGraw-Hill, Berkshire, England.

Stonebraker M (1990). Third-generation database system manifesto. *ACM SIGMOD* record 19, 3, September 1990.

Tsichritzis, D and Klug A (Eds., 1978). The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems. *Information Systems*, 3, 1978.