From semantic to object-oriented data modeling

Haim Kilov

Bell Communications Research
MRE 1E-243
435 South Street
Morristown, NJ 07960

Abstract

The paper shows a possible path from semantic data modeling and data manipulation to greater support of information integrity by means of applying programming methodology paradigms, i.e., object-oriented data modeling and data manipulation. The reader is assumed to be aware of the main concepts (not necessarily the "theory"!) of DBMSs and, in particular, of relational and entity-relationship data modeling.

Introduction

"Most of the confusion in the topic at hand* comes from the lack of a consistent vocabulary" (T.B.Steel, Jr.) On top of that, all too often buzzwords (e.g., "structured" or "relational") trivialize important and deep computer science ideas. Currently, "object-oriented" is one of those most often used buzzwords. Many advertisements suggest that selling something is achieved by arguing that it is object-oriented.

The object-oriented paradigms have been around in programming for quite a while. They are well-known from programming methodology and include separation of concerns, abstraction, information hiding, reuse, thin and visible interfaces between modules, a system organized as a hierarchy of layers, absence of side-effects, etc. [1,2,3]. The implementation of these paradigms — from subroutine libraries to abstract data types — is also well-known. There even exist popular computer science courses for undergraduates based on the use of abstract data types [4]. However, the situation in the database (design) area seems to be somewhat different.

Databases for the latest ten or fifteen years have been — to a certain degree — outside of the mainstream of computer science. Probably, nowhere else has the contrast between academia and industry been so drastic. This had happened due to both technical and non-technical reasons which will not be discussed further. Only recently things started to change for the better: DBMSs of higher quality appear in various computing environments, good texts clearly outline the underlying principles [5], and papers present ideas and experimental systems of much more than purely academic interest**. **Programming methodology paradigms, most importantly, abstraction, start to show up in the database arena.**

Abstraction and database design

Separation of concerns in database design means that knowledge representation (i.e., data modeling) is separated both from "logical" (e.g., relational*** or hierarchical schema) design and from physical file design. Most of the commercially existing systems do not support such a separation, but this factor is of secondary importance: as Dijkstra had stressed quite some time

* data modeling (or "knowledge representation")

** e.g., POSTGRES (University of California), TAXIS (University of Toronto), IRIS (Hewlett-Packard), ORION (MCC), developments by D.Tsichritzis' group, ECRC, etc.

*** which all too often is considered as equivalent to "SQL table design"

TH0309-5/90/0000/0385$01.00 © 1990 IEEE

ago, our machines are there to execute our programs; our programs are not there to instruct our machines. This statement referred to the inadequacy of the commercially existing languages (both low-level and "high-level" ones) as well as to the inadequacy of the computer architectures. Of course, the machines can be virtual and mean commercially existing DBMSs! In the same manner as programming should be abstracted away from existing computer architectures and languages, **database design should be abstracted away from existing DBMSs.**

By defining a clear hierarchy of layers, it is possible to clearly delineate and reuse system components within each layer. Examples are: B-tree access method within the "physical" layer, query implementation strategies within the "logical" layer, and extended ER modeling elements within the "data modeling" layer.

Improving the use of abstraction in information modeling: from relational through semantic to object-oriented

The advent of the basic relational model in 1970 with its "spartan simplicity" (E.F.Codd) was probably the first successful attempt of **explicit separation of concerns in database design**. Indeed, data were modeled using a clear, formal, precise, and more powerful logical model which has been clearly separated from physical: when the relational data were defined and manipulated, the underlying physical data representation as well as the choice of access paths was irrelevant. However, it had soon become clear that the basic relational model is inadequate for dealing with real-life data: this model is still implementation-oriented to quite a large extent. Although it uses the concepts of "tables", "rows", and "columns", in most cases a table corresponds to a "dataset" or a "file" of "records". Facilities provided by the relational model both for data definition and data manipulation are very elementary. As a result, semantics of data definition and, even more often, data manipulation is taken care of by application programs, leading to incorrect and inconsistent data.

In order to actually support data integrity, another layer should be built on top of such a DBMS. More recently, this layer has been created and used (often manually) for semantic data modeling and for mapping of this model onto the underlying (relational) one, but in almost all cases just for data definition and not for data manipulation. The entity-relationship (ER) model by P.Chen with its extensions became the most popular among the "new" models. Often, semantic models were developed as extensions of the basic relational model [6]****. These models provide a much better, but still inadequate, support of information integrity; in particular, change, reuse, and often clarity of definitions were not dealt with.

Although semantic data models are more sophisticated than relational, they do not adequately support a very important aspect of abstraction — encapsulation, i.e., information hiding on an abstract data type level. These models deal only with state encapsulation, i.e., data definition on a sufficiently high level, but not with behavior encapsulation, i.e., high-level data manipulation operations†. As a rule, data manipulation operators for meta-types are not available, although definitions of meta-types are essentially based on their behavioral properties. Ideally semantics-independent atomic operations (e.g., relational ones) should not be available to the application. However, in "real life" low-level information carefully abstracted out at the time of data definition is provided to an application developer (all too often a not highly qualified "SQL programmer") for data manipulation because there exist no other means to interface with the data! Note that some recent commercial systems (e.g., ZIM or 4th Dimension††) acknowledge this problem and support higher-level data manipulation, but even these systems still permit the application developer to violate semantic

**** In fact, the underlying (commercial) DBMS is of less importance, if data manipulation on a higher level is supported - which all too often has not been the case.

† support of high-level data manipulation even for the basic ER model exists only at the research paper level [7].

†† The mention of particular products or vendors in this paper is as an example of a technology and is not intended to be a recommendation or endorsement of products or vendors.

integrity by means of using uninterpreted, "relational-level", operators.

High-quality data models can be built if we take into account lessons learned from programming methodology in general, and from object-oriented systems in particular. Note that current object-oriented systems based on object-oriented languages seemingly missed lessons learned from semantic data modeling. This paper will consider the path(s) from semantic data modeling toward greater support of information integrity by means of object-oriented paradigms. It will not deal with the very interesting path(s) from object-oriented programming paradigms towards object-oriented data modeling within an information system environment.

From semantic to object-oriented approach: theory

The ER model was first considered as a structural extension of the relational one. However, later it became evident that the definitions of entity meta-types themselves were based not only on structural (e.g., aggregation), but also on certain behavioral properties of the relevant objects. The introduction of data abstraction and extensibility into semantic data modeling made these observations even more apparent. As a natural next step, abstract data type support, i.e., object orientation, becomes crucial in data modeling.

Database abstractions for semantic data modeling

A "typical" semantic data model is based on the support of structural abstractions [8]: aggregation (components that are required to make up an object [9]) and sometimes generalization. In the simplest case (the basic ER model), aggregation support means uniquely identifying entity instances by means of specifying an entity as an aggregation of its "inherent" (Jay Smith) properties, as well as relationship instances by means of specifying the constituent entity instances. Note that the ER model supports (constraints governing) primitive behavioral properties of entities and relationships with respect to create, update, and delete operations.

An extended ER model supports some additional concepts. It acknowledges that **not all entities are created equal** and supports, e.g., weak (characteristic, "embedded") entities, composite entities, and subtyping. Different kinds of clusters containing interrelated entities are therefore supported. In this manner, referential integrity and cardinality are clearly seen as low-level, uninterpreted concepts used for building higher-level semantic models.

The next step: meta-types and behavioral rules

Semantic data modeling means more than support of structural abstraction. Consider the concept of meta-types, e.g., relationships, "kernel" ("strong") entities, weak entities, composite entities, reference entities, etc. A sufficiently rich set of such fundamental off-the-shelf meta-types, independent of a particular enterprise, can easily be reused in the process of actual modeling of any enterprise. These meta-types and their integrity rules and constraints will not have to be reinvented by a data modeler for each new enterprise (and new entity!) to be modeled. In order to categorize an entity as belonging to one of the meta-types — an essential data modeling activity — it is necessary to examine the structure and the (elementary aspects of the) behavior [10] of this entity with respect to (i.e., within a cluster of) other entities. **Meta-types are defined by integrity rules and constraints which may be considered as a generalization of the business procedural rules of an enterprise.** As a simple example, a weak entity instance (e.g., a dependent) may be created and entered into the database only if its corresponding "kernel" entity instance (e.g., an employee) already exists; on the other hand, when a kernel entity instance is to be deleted, no corresponding weak entity instances should exist in the database (the user may wish to relax, or otherwise replace, the latter rule). As a slightly more complex example, a reference entity type enforces a rule in accordance to which an entity instance having a certain set of properties may be created (e.g., an equipment item should be received rather than

returned) only if a reference entity instance with this set of properties exists (e.g., if a corresponding item exists in the catalog); this concept may roughly be visualized as an "extended non-atomic domain" Note that these integrity rules "express facts about aggregated object (-types) without going into too much detail on how such object (-types) are built up" [11] as aggregations of their properties. In other words, the interrelationships of the corresponding entities usually do not depend upon their properties.

The number of meta-types, unlike the number of types, is very small. However, it is very important to precisely (formally) define the properties of a meta-type object with respect to other relevant objects (in its cluster) because only in this manner a meta-type can be unambiguously understood. In quite a few object-oriented database projects meta-types are not supported, or not clearly defined, or defined in a too restrictive manner. Sometimes different concepts (e.g., composite objects and dependent objects) are merged in the same meta-type [12]. For instance, the often encountered and seemingly trivial notion of a "dependent" object may lead to confusion because it has different and often vaguely defined meanings in different systems, and therefore the semantics of the appropriate clusters is different. In order to solve this problem, distinct meta-types (with different names) should be defined as precisely and explicitly as possible.

Note that an object-oriented model based on an object-oriented programming language — rather than on a semantic data model — usually ignores coarse (i.e., meta-type) semantics of data expressed by inter-entity associations and therefore makes understanding more difficult: an additional abstraction layer is not used! This approach is against the spirit of object-oriented programming.

Semantic data models and better support of behavior

Semantic data modeling deals with both the structure and *some aspects of the behavior* of entities — information objects belonging to certain meta-types. However, this support of object behavior is weak. Although a good semantic data model can recognize an appropriate *meta-type* for an object based on properties of its primitive interactions with other objects (and therefore *start* defining clusters within which an object exists and behaves, compare [13]), this model has no means whatsoever for specifying a set of predefined non-primitive operations applicable to the object *type*! For example, it may be easy to distinguish between an employee as a "kernel" entity and a dependent as its "weak" entity, but no semantic data model can specify and support a set of abstract operations (e.g., "hire", "raise salary", "promote", "buy a computer system", ...) applicable to an instance of an employee type. Information integrity can be supported only by means of systematic use of these molecular operations rather than of atomic, semantically-devoid, record-level "CRUD" (create-read-update-delete) ones (e.g., "hire an employee" means much more that just "create an employee record", and therefore in order to support integrity only the former operation should be available to the application developer; this operation will automatically support all the relevant properties of hiring specified by means of postconditions). Note that **meta-type CRUD operations are not record-level** because they usually deal with several entities belonging to the same cluster. For instance, deleting an employee with dependents in the context of a semantic data model usually means system-supported deletion of records corresponding to both the employee and the dependents.

Generalization within the framework of a semantic data model supports only structural property inheritance for subclasses. Within this model, it is impossible to inherit behavioral properties of an object class since they are not supported. It can support the structural properties of, e.g., a secretary who "is-a" employee, but cannot support the behavioral ones, e.g., that hiring a secretary means executing the same procedure as hiring an employee plus possibly executing another procedure relevant only to a secretary.

From semantic to object-oriented approach: how-to

The considerations in the preceding section are not just of a theoretical, but also of a very important practical interest. In fact, they show the methods of actually integrating system support of corporate data structure and behavior, leading to semantically correct schema building as well as schema and data browsing.

Step One: From fixed to extensible meta-types

A "typical" semantic data model is based on a predetermined set of meta-types, a couple of attribute type constructors (using structure and perhaps array constructs), and possibly subtyping for "traditional" business-oriented types. In addition, some papers discuss the concept of domains (for entity properties) as sets of possible values with reasonable operations defined on them [14,15]; this approach is usually not supported, although it solves the important problem of naming (including name subsets and different names for the same thing). Naturally, the extensibility of such a type system is primitive and limited: the data modeler can define a new "record type", but this new type will not be much semantically different from the existing ones. However, **if the set of meta-types is extensible,** i.e., if it is possible to define new "entity categories", then **the information system becomes much more sophisticated** because a sizable amount of "generic" integrity rules becomes supported by the system rather than by the applications. These are usually behavioral rules and constraints dealing with atomic activities propagated within clusters of related objects when one of them is (to be) created/changed/deleted. High-level meta-type behavior is defined in this manner by the data modeler — rather than by an application — and therefore an extensible semantic data model supports more information integrity than a "usual" extended ER model.

Currently there exist many extended ER models which may differ by their meta-types (i.e., by which kinds of inter-entity predicates are supported). For instance, relationships may or may not be considered entities, may be binary or n-ary, may or may not have attributes, etc.; weak entities may have different integrity rules for deletion, may depend upon exactly one or more than one parent entity; the notion of a composite entity may or may not be supported, etc. As a rule, these models are not extensible, i.e., adding a new meta-type by the data modeler is either very difficult or impossible. In some cases, new meta-type(s) added by the model authors may mean a new and "vastly improved" (version of the) model.

The small set of widely used meta-types does not depend on the application domain. Therefore it can be included (as a "meta-type library") in any extended ER model. This set defines clusters of related objects recognizable by the model. However, it is hardly possible to freeze such a set, because in real life a need might appear for the support of new inter-entity integrity rules (possibly, but not necessarily, within the appropriate clusters) and therefore for creating new (possibly, by partially reusing old) meta-types. (As a simple example, consider the creation of a new meta-type for a weak entity with more than one parent entity.) This activity should be available to qualified data modelers rather than only to the model authors.

Step Two: From extensible meta-types to extensible types

In this manner, a decisive step is made toward the object-oriented paradigm which supports both behavior encapsulation and extensibility of the *type* system (i.e., not just meta-types, but also "user-defined" abstract types). **The "semantic" concept of extensible behavior at the meta-type level naturally leads to the "object-oriented" concept of extensible behavior at the type level.** As the number of types is much larger than the number of meta-types, it is impossible to create a universal type library. However, it is possible to create and (re)use type libraries for specific application domains, although the number of types in such a library may be very large. Both new types and new meta-types are defined not by a user (who can just place orders for them),

but rather by a data modeler ("data administrator") — an insider (B.Liskov) with respect to the system who has control over all rules governing the (behavior of) database objects. Note that the "user-defined" types should not necessarily be record-oriented — on the contrary: they may include long texts, images, sound, etc., with corresponding operations.

The importance and role of precise specifications in meta-type and type creation and reuse

The definition of types and meta-types should be governed by predefined specifications ("assertions") precisely formulating the properties of the relevant clusters within the information system. This is not a new idea: axiomatic definition of abstract data types [16,17] has been proposed more than ten years ago. The information system environment is more complex than the "usual programming" one, and therefore disciplined software development within this environment is of even greater importance. Clear and precise specifications of abstract data type operations and their properties by means of assertions (predicates, i.e., "boolean expressions" with quantifiers, that should be valid for certain states of the computation, i.e., at certain points of the program text†††) and invariants (predicates that should be always true for a certain environment) provide a rigorous basis for writing correct software. The level of formalism of these specifications may be adjusted towards the practical needs of software developers. If, however, the specifications are imprecise, i.e., expressed by means of various sorts of handwaving, then the information system may well become inconsistent (as an aside, note how the informal and therefore imprecise character of certain standards in the database area had negatively influenced its "state-of-the-art").

††† "Program text" can and should be written on different levels of abstraction. In particular, "design" and "implementation" texts are just two of the possibilities - compare [16].

Ramifications

The concepts presented in the preceding sections and their actual implementation are reasonably straightforward. It should be noted, however, that they have important ramifications for the database community. Both system development and user interface become more simple and elegant.

Encapsulation and intelligent browsing through the database schema. Clusters

One of the most important goals of an object-oriented DBMS is to provide facilities for encapsulation and reuse of types and meta-types both by data modelers and by application developers. In this manner, a solution may be "configured" instead of "programmed" in the more traditional sense; note that the **methodology is the same for both the "programming" and the "configuring"**. Data modelers should be able (with care!) to create new types or meta-types based on the existing ones (possibly, by overriding some of the properties or by applying partial inheritance), whereas application builders should be able to use the existing types only through the facilities offered in their specifications (i.e., never have any access to underlying implementations). In both cases there exists a very important necessity of finding the relevant type within a certain context. This software engineering problem of intelligent context-sensitive browsing through the database schema is currently the subject of extensive research [18,19].

Browsing within an object-oriented model is easier than within a semantic one because clusters of related objects are more diverse and convey more information than clusters of related meta-types within a semantic model. Indeed, the latter clusters are incomplete because they are based only on the (structural and behavioral) properties of meta-types and on structural inheritance, ignoring behavioral properties of types. For instance, changing the balance of a savings account may include opening a loan account if the balance becomes less than a certain amount,

and therefore "loan account" and "savings account" should belong to the same cluster, although neither inheritance nor the (aggregation-based) properties of corresponding meta-types suggest this. **In order to completely define a cluster, properties of both meta-types and types should be taken into account.** This step can be done when a semantic data model is extended with object-oriented primitives. In this example, "loan account" will be included in the specification (assertion) of the behavior of "savings account", and therefore both object types will belong to the same cluster. (Compare with the notion of a reference object in [20].)

Abstraction and discovering types and meta-types

In order to (re)use a type or meta-type, its properties should be precisely formulated — otherwise the too well-known problem of using an almost black box will have to be solved. The job of finding (identifying) an object type or meta-type and formulating its properties is usually done by an "abstractor" who "comprehends a problem space, isolates a portion of the space for which a general solution is possible, and provides a reusable solution". In order to avoid "TILI", i.e., the take-it-or-leave-it syndrome, "the abstractor has to communicate the context within which an abstraction solves a problem to potential users" [21].

The abstractor's job of **discovering object types and meta-types cannot be formalized** because mapping of the "real world" onto a formal system is an informal activity. In the same manner, there is no algorithmic way of creating a database schema, and of programming in general! However, in all these cases certain guidelines are available [1,2,3]. An abstractor can consider a set of existing object types and meta-types that might be at least partially applicable to the problem at hand. Another useful heuristic (proposed for use in the Ada environment by G.Booch) is to take an appropriate "noun" out of the requirements document and use it as a first approximation of a possibly new object type. Note that discovering meta-types is an activity of greater importance, responsibility, and complexity than discovering types: meta-types (unlike types) are independent of the application domain and will therefore be reused much more often. If the meta-types are not adequately defined, i.e., type properties are not properly abstracted out, then the complexity of the type system will drastically increase or else some of the type properties would have to be "supported" by application programs — a completely unacceptable solution. If the number of different types in the database is small then, after some effort, the type system can be understood even without the use of meta-types. If, however, this number is large(r) then mastering complexity of the database schema becomes very non-trivial, to say the least, and in these cases the existence of an additional abstraction layer (i.e., meta-types) can be the decisive factor in accepting the data model.

Bottom-up design in the changing world vs. *ad hoc* relational queries

The meta-type layer is very useful both for developing a system from existing tools in a bottom-up manner and for decomposing an application into objects in a top-down manner. Although both approaches are used, some authors convincingly suggest that the bottom-up one is certainly preferable within vaguely defined, ill-structured, and rapidly changing environments because it takes into account the engineering needs for change and reuse [16,17,22]. Many if not most database design activities belong to these environments.

Within the realm of relational DBMSs, the need for change materialized in the very popular concept of *ad hoc* queries (applied to a fixed schema!) which were even considered as one of the most important merits of the basic relational model in comparison with navigational ones. However, it is easy to see that **"arbitrary" relational joins have no semantic value** and should therefore (as a rule) not be permitted. Usually queries are not *ad hoc*: on the contrary, they are predefined within the object type definitions stored in the high-level schema (even if these definitions exist only in the head of the data modeler), and the correct pattern is some-

thing like "*ad hoc* query number 3". The almost only possibility for "unanticipated" relational queries materializes when the user wants to do selections and projections rather than joins, but this does not require knowledge of any access paths. Moreover, ***ad hoc* queries do not reflect the semantics of the changing "real world" because they are formulated with respect to a fixed database schema**. On the other hand, system development in a bottom-up manner takes care of the changes both in data definitions and, consequently, in corresponding operations and therefore reflects the inevitable changes in the real world.

Generic and application-specific development

Libraries of meta-types can be used for creating generic applications, whereas libraries of types are usually created for particular application areas and are necessary for application-specific development. In particular, users will see only the terminology of their own application (i.e., of the appropriate types) and will not be aware of the generic (i.e., the meta-type and the object-oriented) terminology. The paradigm of information hiding ("need-to-know") will help them to be comfortable both with their old and with their new applications which may be delivered in an incremental manner by means of adding relevant types to their type libraries.

Summary: visualizing another abstraction layer

Traditionally the development and use of databases has been considered within the framework of using the physical layer at the bottom, the logical layer on top of physical, and the conceptual layer on top of logical. This visualization helped to understand the most general approach to information systems. Prepackaged components of the physical layer (e.g., the B-tree access method) and less often of the logical one (e.g., relational join optimization algorithms) have been reused in actual system design and actual applications. Currently, however, it became understood that the layer of reusability could and should be higher than for "classical" systems: both meta-types and types should be reusable.

As a consequence, **the notion of one conceptual layer becomes too coarse.** Indeed, the "lower conceptual layer" takes care of *meta-types*, i.e., of extended ER model primitives, whereas an object-oriented model adds extensible behavior of abstract data *types* built on top of these meta-types and corresponds to an "upper conceptual layer" built on top of the "lower conceptual layer". Each type belongs to some meta-type within the context of a corresponding cluster. In this manner, the additional layer facilitates controlling complexity, i.e., better understanding of and browsing through the database schema†††† and consequently the reuse of existing components of this schema. This layer takes into account the existing **valuable experience in semantic data modeling too often ignored when object-oriented database systems are built upon object-oriented programming languages.** Intelligent browsing becomes easier because some of the common properties of types within their clusters are abstracted away and presented as properties of the corresponding meta-types.

Conclusion

The dissimilarity between the semantic and the object-oriented models is "not as sharp as it may seem" [9]. The trend for current experimental and even commercial database models is to have both facilities: both structural and behavioral properties of data should belong to the database schema, i.e., to a "data definition language". Extensibility of the sets of available meta-types and of available types as well as explicit support of high-level behavior at the meta-type and type level lead to the integration of these two models, and therefore, as this paper has hopefully shown, the path from an extensible semantic model to an object-oriented one is within the

†††† in order to decrease the "certain amount of black magic" needed for problem decomposition into objects and for browsing through a very large object software base, see [22].

reach of the current generation of data modelers and even users.

Acknowledgements

Thanks for very interesting and fruitful discussions and comments go to Harry Goldberg, Dave Luber, Jim Ross, and Amit Sheth.

References

[1] E.W.Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[2] E.W.Dijkstra. The teaching of programming, i.e., the teaching of thinking. *Lecture Notes in Computer Science, Vol. 46*. Springer Verlag, 1976.

[3] D.Gries. *The science of programming*. Springer Verlag, 1981.

[4] D.D.McCracken. *A second course in computer science with Pascal*. J.Wiley & Sons, 1987.

[5] J.Hughes. *Database technology: a software engineering approach*. Prentice-Hall, 1988.

[6] E.F.Codd. Extending the database relational model to capture more meaning. *Transactions on Database Systems*, Vol.4 (1979), No.4.

[7] C.Parent and S.Spaccapietra. Enhancing the operational semantics of the entity-relationship model. In: *Database Semantics* (ed. by T.Steel, Jr., and R.Meersman). North-Holland, 1986, pp. 159-173.

[8] J.Smith and D.Smith. Database abstractions: aggregation and generalization. *Transactions on Database Systems*, Vol. 2 (1977), No.2.

[9] R.King. My cat is object-oriented. In: *Object-oriented concepts, databases, and applications* (Ed. by Won Kim and Frederick H. Lochovsky). Addison-Wesley, 1989, pp. 23-30.

[10] H.Kilov. An approach to conceptual schema support in a relational DBMS environment. *Proceedings of the IV Jerusalem Conference on Information Technology*. IEEE Computer Society Press, 1984, pp. 347-349.

[11] R.Meersman. Preface to *Data and Knowledge (DS-2)*. (Ed. by R.Meersman and A.Sernadas.) North-Holland, 1988, p. vii-xii.

[12] G.T.Nguyen and D.Rieu. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering*, Vol. 4(1989), pp. 43-67.

[13] T.Teorey, G.Wei, D.Bolton, J.Koenig. ER model clustering as an aid for user communication and documentation in database design. *Communications of the ACM*, Vol. 32, No. 8 (August 1989), pp. 975-987.

[14] H.Kilov. Domains and semantic integrity. *Computer Standards and Interfaces*, Vol.9 (1989).

[15] E.Laenens and D.Vermeir. A language for object-oriented database programming. *Journal of Object-Oriented Programming*, Vol.1, No.5 (1989), pp. 18-27.

[16] B.Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming*, Vol. 1, No. 1 (1989), pp. 19-39, especially section 4.

[17] B.Meyer. *Object-oriented software construction*. Prentice-Hall, 1988.

[18] C.Arapis, G.Kappel. Organizing objects in an object software base. In: *Active object environments* (Ed. by D.Tsichritzis). University of Geneva, 1988, pp. 32-50.

[19] X.Pintado, D.Tsichritzis. An affinity browser. In: *Active object environments* (Ed. by D.Tsichritzis). University of Geneva, 1988, pp. 51-60.

[20] R.Unland and G.Schlageter. An object-oriented programming environment for advanced database applications. *Journal of Object-Oriented Programming*, Vol. 2, No. 1 (May/June 1989), pp. 7-19.

[21] W.Cunningham and K.Beck. Constructing abstractions for object-oriented applications. *Journal of Object-Oriented Programming*, Vol. 2, No. 2 (July-August 1989), pp. 17-19.

[22] O.Nierstrasz and D.Tsichritzis. Integrated office systems. In: *Active Object Environments* (Ed. by D.Tsichritzis), University of Geneva, 1988, pp. 187-201.