June 2, 2008

Editor Roberto V. Zicari- ODBMS.ORG http://www.odbms.org

Object Database Systems: Quo vadis?

I wanted to have an opinion on some critical questions related to Object Databases:

Where are Object Database Systems going? Are Relational database systems becoming Object Databases? Do we need a standard for Object Databases? Why ODMG did not succeed?

I have therefore interviewed three of our Experts, **Mike Card, Jim Paterson,** and **Kazimierz Subieta,** on their view on the current State of the Union of object database systems.

Mike works with Syracuse Research Corporation (SRC) and is involved in object databases and their application to challenging problems, including pattern recognition. He chairs the ODBT group in OMG to advance object database standardization.
Jim is a Lecturer in the School of Computing and Mathematical Sciences at Glasgow Caledonian University.
Kazimierz is professor at the Polish-Japanese Institute of Information Technology and active in the ODBT group in OMG.

Question1:
It has been said (See Java Panel II) that an Object Database System in order to be a suitable solution to the object persistence problem needs to support not only a richer object model, but it also has to support set-oriented, scalable, cost-based-optimized query processing, and high-throughput transactions.
Do current ODBMS offer these features?

Mike Card:
In my opinion, no though the support for true transactional processing varies between vendors. Some products use "optimistic" concurrency control, which is suitable only for environments where there is very little concurrent access to the database, such as single-threaded embedded applications. In my opinion, a database engine is not "scalable" (at least in the enterprise sense of the word) if it is based on optimistic concurrency control. This is because most truly large-scale applications will require optimal performance with many concurrent transactions, and this cannot be achieved when updates have to be rolled back at transaction commit time and re-attempted due to access conflicts.

Jim Paterson:
Different ODBMS products offer these features to greater or lesser degrees. Transactional processing, for example, depends more on the design and intended purpose of the individual product than on the data model.

Kazimierz Subieta:
Functionality of query languages in existing ODBMS is limited. Our ODRA system and its query/programming language SBQL is the first case when a query language offers the full algorithmic power. In this respect SBQL follows SQL-2003, but it is much simpler. The potential of a query optimizer depends on functionality of the query language. Usually complex functionality cuts opportunities for query optimization. A query language without query optimization is not usable for large databases. This is the main reason for limited functionality of query languages is current ODBMS. Again, SBQL implemented in ODRA is an exception: due to formal theory (Stack-Based Architecture) we have developed very powerful and general query optimization methods (rewriting rules, removing dead subqueries, query modification, methods based on indices, pipelining, cached queries, etc.). Next methods are being developed. Concerning transactions, we have implemented algorithms that are known from relational systems, but currently we work on new algorithms for distributed transactions. I believe proper qualities of ODBMS are possible only in case when a new object model and a new languages addressing the model are developed. Databases are enough important in developing business-oriented applications to be worth their own datamodel and a programming language. Any bottom-up approach that extends existing programming tools (e.g. Java or C#) by database capabilities must result in functional limitations and exotic (if not chaotic) solutions. The model and the language must integrate a query language with a (database) programming language into a homogeneous whole. This is just done in SBQL. From the very beginning

we have adopted the UML object model, consequently turning it from analytical to the programming model and adding to it a query/programming language SBQL.

Any form of impedance mismatch should be avoided and this is impossible on the ground of the bottom-up approach. In particular, the mechanisms serving (nested) collections and persistence must be build-in into the integrated language as fundaments. Any kind of libraries, API-s, mappers, queries expressed in native syntax, etc. that are used in the bottom-up approach must result in limitations and violating principles of language's construction. Although they can enjoy some local success, from the strategic point they present a mistake that decreases the acceptance of ODBMS by wide software industry communities. Extending the network of small and narrow paths is a bad strategy for constructing highways.

Question2:
Relational systems are rapidly becoming object database systems (See Java Panel II). Do you agree or disagree with this statement? Why?

Mike Card:
I would disagree, because relational databases still fundamentally access objects as rows of tables and do not offer seamless integration into a host programming language's type system. It is true that there are some good ORMs out there, but these will never offer the performance or seamlessness that is available with a good ODBMS. I would agree that ORMs are getting better, but relational databases themselves are not becoming object databases.

Jim Paterson:
The DBMSs themselves are not generally becoming object databases, but they are becoming increasingly used as databases for storing objects, a purpose for which they are not necessarily well-suited even with the help of ORMs.

Kazimierz Subieta
Pure relational database systems cannot be considered object-oriented. The question probably concerns extensions of relational systems by object-oriented features. Such extensions imply a lot of very hard issues. If objects are added on top of relational tables, then the complexity of the database programming environment significantly grows. New concepts added to the data model increase it complexity in square, as they must be combined with already introduced concepts. Every such concept must be served by query and programming capabilities, which grow in square as well. Assuming orthogonal combination of query and programming capabilities, every new concept must result in explosion of new concepts in queries and programs. Query optimization adds its own overhead: combinations of concepts must be served by a query optimizer, which implies further complexity of the implementation. This is one of the reasons of the volume of the SQL 2003 standard. The users of object-relational systems must cope with complexity and competence mismatch: a concrete business task can be accomplished in many ways, but some of them could be inefficient.

There is little evidence that object-oriented features added on top of relational tables are used by anybody. Big relational companies send to the user (consciously or not) a message "you see, we have implemented all object-oriented features, but you cannot use them because they are inadequate to your business". In effect, users are more convinced that object-orientation in databases makes little sense and they should stay with relational systems. A good side effect from the position of marketing offices of relational databases vendors, but very bad effect concerning the progress in the domain. For this reason we start to build a pure object-oriented database management system ODRA, discarding all fundamentals of the relational model. However, we adopt good experience of relational systems, including architectural assumptions, query operators, query optimization, transaction processing, etc.

The object-oriented model includes the relational model by definition: every relational tuple can be considered an object. In SBQL, if object model is reduced to the relational model, then many SBQL queries becomes SQL queries, up to minor syntactic sugar. However, the SBQL object model includes many features that 1:1 map object-oriented conceptual modeling visions of the analysts, designers and programmers. Thus, SBQL looses nothing from the relational model but enables database programming on the conceptual level that is close or identical to object-oriented analysis and design models.

We are dealing with collections of objects, just as in our common understanding of the real world.

Classes, methods and inheritance can be features of the server, which implies great reusability, hard to achieve in relational systems. Encapsulation concerns a client and a server, hence a lot of features (including some server attributes) can be hidden from the client conceptual view. The concept of "sub-table" that could be a problem of potential SQL 2003 programmers is redundant.

Question3:
A lot of the worlds systems are built on relational technology and those systems need to be extended and integrated.
That job is always difficult. An ODBMS should be able to fully participate in the enterprise data ecosystem as well as any other DBMS for both new development as well as enhancing existing applications. How this can be achieved?
What is your opinion on this issue?

Mike Card:
As many vendors have noted, this is to some extent a marketing problem in terms of making enterprise customers aware of what object databases can do. It is also a technology issue, however, as engines based on “small-scale” concepts like optimistic concurrency control are not suitable to many enterprise environments.

Jim Paterson:
The ODBMS may occupy a particular specialized role within the ecosystem, and may need to be able to work with other producers and consumers of data. For example, transactional data or data gathered on mobile or partially disconnected devices may need to be made available for other purposes, like data mining and decision making. The ODBMS itself may not be the best data source for these activities, and so the ability, in db4o for example, to replicate data to other types of data store can play a big part in allowing the ODBMS to participate fully.

Kazimierz Subieta:
There are two possible approaches. The first one is obvious: a relational database is 1:1 virtually mapped as a database in a given database model X, in particular, object oriented. Then, queries addressing the datamodel X are 1:1 mapped into SQL. This approach was accomplished by me in 1992 in the DBPL system built at the Hamburg University. The advantage of this approach is that SQL optimizers are fully utilized. The disadvantage concerns the slavery w.r.t. the relational model: the user must work with relational structures, although perceiving them as structures of X, and a query language addressing X must be constructed in such a way that mapping a query into SQL is straightforward. Concerning very large relational databases, many ORM-s that are developed today actually must apply this approach, even if they offer more sophisticated mappings from object-oriented structures into relational ones. In database terms such mappings are views and it is commonly known that sophisticated views lead to performance problems and to view updating problems. The problems may cut the possibility of sophisticated mappings, although for some particular cases (databases of moderate size, retrieval only, etc.) they can be efficient.

The second approach is much more sophisticated and till now accomplished only in the ODRA system. In this approach we tried to free from the slavery by making a wrapper from SBQL to SQL with full algorithmic power and without loosing SQL optimizers. The idea is that firstly an internal wrapper is built (as in the first approach) that virtually maps every relational tuple into a primitive ODRA object. Then, such objects are virtually mapped into complex and linked objects by SBQL updatable views. In contrast to SQL, SBQL views have full algorithmic power. Such complex objects, although virtual, are non-distinguishable from regular ODRA objects and can be addressed by SBQL queries and updates. However, the processing of such queries and updates is significantly changed. SBQL queries addressing complex virtual objects are mapped via views into huge SBQL queries addressing primitive tuple-like objects. Then, in such queries we look for biggest subqueries that can be 1:1 mapped into SQL. Such subqueries are mapped into SQL through JDBC. So far we have positive tests of this method for several cases. The biggest problem with this method is caused by SBQL features that have no counterparts in SQL. And vice versa, we cannot generate SQL operators that are absent in SBQL. So the problem of mapping between object and relational data structures to some extent will persist.

Question4:
Object databases vary greatly from vendor to vendor. Is a standard for object databases (still) needed? If yes, what needs to be standardized in your opinion?

Mike Card:
Yes, I believe it is. The APIs for creating, opening, deleting, and synchronizing/replicating databases as well as the native query APIs should be standardized to allow application portability. Any APIs needed to insert objects into the database, remove them from the database, or create an index on them should also be standardized, again for the sake of application portability. I would also like to see a standard XML format for exporting object database contents to allow for data portability. I am not sure our current OMG effort can achieve all of these standardization goals, but I would like to.

Jim Paterson:
"Good standards can provide interoperability and portability. Bad standards can stifle innovation. 'supports XXX standard' is not a real user requirement... *De facto* standards are usually a much better fit to user requirements than *a priori* ones." That quote comes from the developers of Hibernate, arguing that Hibernate has become such a de facto standard. Similarly, SQL was a de facto standard before it became an ANSI standard. SQL was not originally successful because it was standardized: it was standardized (well, kind of) because it was successful in meeting user requirements. So, it could be argued that ODBMS standards should be based on what is actually being used successfully and is meeting those user requirements for creating and opening databases, schema definition, querying, inserting, updating and deleting objects and creating indexes.

Kazimierz Subieta:
It is said that the goals of a database standard include portability of applications and interoperability between different systems. However, this effect didn't happen within relational DBMS (despite several SQL standards) and didn't happen after publishing versions of the ODMG standard. Moreover, till now and perhaps in the nearest future big vendors of database systems present no interest in the standardization of object databases.

Many people have problems with defining the goals of an object database standard and many of them doubt if the standard is necessary and has chances for success. What kind of benefits one can expect from the standard if portability and interoperability are hopeless anyway and the biggest players are out of the game?

I think that the biggest benefit is rising the authority of object-oriented database idea in the community. The standard, with proper conceptual and technical quality, may become a pivot of discussions, comparisons and new developments, including especially the authority within influential academic communities. The standard of proper quality may also rise interest of big players. For OMG, the object database standard is the biggest gap in promoting object-oriented technologies.

Question5:
How would this new standard would different to the previous effort in ODMG? And what relationships this new standard would have with standards such as SQL?

Mike Card:
Unlike the previous ODMG standard, the new standard should have a conformance test suite that anyone can download and run against a candidate product. The standard itself should also be unambiguous and use precise language as is done in ISO standards for things like programming languages, e.g. ISO/IEC 8652 (Ada programming language standard).

The primary focus of an object database standard should be its support of a native programming language, so I would expect that an object database standard might be more closely tied to an ISO standard for an object programming language (Ada, C++, other ISO-standardized languages that may appear) than to SQL, though perhaps if a LINQ-like native query capability were included in the object database standard would also reference the SQL standard due to the use of SQL-like verbs and semantics in LINQ.

Jim Paterson:
See my reply to Question 4.

Kazimierz Subieta:
The ODMG standard was too early and too much oriented towards quick marketing effect. The effect, however, didn't happen. Just otherwise, the standard to some extent spoiled the opinion on object-oriented databases. Prof. Suad Alagic asked "does it make sense?" and this question is still relevant. Many people believe that the ODMG standard is no standard but an intermediate proposal of some research group. Technically, the standard is underspecified, inconsistent and incomplete. For instance, despite explanation, we don't know what is "object" in the sense of a formal definition. The typing system proves lack of competence concerning the state of the art. Specification of the semantics of OQL through rough explanations and simple examples is far from the rigorous specification discipline required for standards.

I think, however, it is much easier to improve things than invent them from the beginning. In this sense the standard fulfills its role and on its ground we can think on further standards. The ODMG standard can be improved and actually our research on the stack-based architecture and SBQL can be considered as such an improvement. So, in general my attitude to ODMG is still positive. This job must be completed, simply. But not in the style that was represented by ODMG. The standard must be much more rigorous, with clearly defined formal semantics. The stack-based architecture offers such a semantic specification method. Moreover, the standard should not be published only as the result of voting. It must be supported by a reference implementation, even non-optimized. Enough to say that two my (indeed, brilliant) MSc students have made a bigger database system in comparison to the ODMG proposal. The system was a non-optimized prototype, but operating, hence for sure its specification was checked for correctness. Not a big deal, but absolutely necessary. Independently how many and how much competent professionals work on a specification, implementation always discovers a lot of hidden flaws.

I don't consider the recent SQL standard as a basis for my work. As an academic researcher, I have my own sense of aesthetic. I don't like huge, eclectic and redundant constructs. Things should be as simple as possible (but not simpler). The SQL-2003 standard is far more complex than necessary. As an engineer, I have my own feeling of usability. I don't believe that SQL-2003 will be usable in common practice, event if it will be implemented. Probably, people will still use some small subset of SQL-2003. As a teacher I have doubts if more than 1000 pages of the specifications (plus thousands of pages of various appendices) that are not implemented yet are worth attention. As many my academic colleagues, I observe the SQL scene and quietly wait what will happen next.

What needs to be standardized? After implementation of the ODRA system with SBQL as a query and programming language the answer to this question seems to be easier. I think that ideologically the features should be close to SQL-2003, but technically based on a pure, non-redundant object model. I list the features with comments.

1. The concepts of an object and an object store on a very high abstraction level (i.e. an object model). In ODRA an object is a triple <*i,n,v*>, where *i* is an internal object identifier, *n* is an external name, and *v* is a value, which can be atomic, can be a pointer and can be a set of objects. In this style we can define an object store, collections, classes and inheritance, dynamic object roles, etc. The object relativism is an important principle. A database consists of objects. An object is atomic or consists of objects. General properties of objects on any hierarchy level are the same. The relativism much simplifies further specifications and implementations, including query optimization.
2. A gross architecture of processing that subdivides the programming environment into modules responsible for database and program development, clients and servers, volatile and persistent data, query processing and optimization, etc. Specification of the architecture is necessary for unified common understanding of other concepts specified in the standard.
3. A semi-strong typing system that allows to specify types of objects, types of collections, inheritance, classes, interfaces, specification of variables (including database variables), etc. "Semi-strong" means that the typing system works with collections with the cardinalities [0..1] (optional objects), [0..*] (repeating objects) and with automatic type coercions that change, if necessary, an element e into a collection {e}, and vice-versa. Semi-strong typing is an answer to the common complain that the strong typing is too strong. The typing system should be unified for persistent (server side) and volatile (client side) data.
4. A database schema language and a metamodel internally representing a schema. In ODRA we were much influenced by ODMG ODL and by UML class diagrams. The specification of a metamodel is necessary for generic programming with reflection (c.f. the CORBA Interface Repository).
5. A query language that addresses all features of the introduced data model and a schema language. Queries and programming expressions should be unified. For instance, 2+2 should be a regular

query. The most primitive queries are literals and data names. More complex queries should be built by orthogonal combination of simpler ones, following the typing system. Hence, specification of syntax should avoid big syntactic aggregates (such as *select…from…where… group by…having…*). Specification of semantics should be explicit and formal (but not necessarily mathematical).

6. Imperative (programming) constructs based on queries. Providing queries are unified with programming expressions, queries can return references to stored objects (to be used on the left-hand-side of assignments, as arguments of delete operations, as *call-by-reference* parameters, etc.).
7. Programming abstractions known as modules, procedures, functions, classes, methods, etc., having queries as parameters. Parameters can be passed in the *call-by-value* and *call-by-reference* modes, with some possible variants. Modules and classes should follow encapsulation, e.g. they can exist in two forms: as interfaces and as implementations.
8. Exceptions and exception handling integrated with all the programming environment.
9. Facilities for programming and synchronization of parallel processes (threads).
10. Persistent database abstractions. They include all programming abstractions: procedures, functions and classes can be stored at the database side and there should be no differences in specification and access in comparison to their volatile versions. Persistent database abstraction should include updatable database views, triggers and business rules.
11. Constructs for transaction processing, including optimistic and pessimistic algorithms, and protocols for distributed transactions.
12. Programming interfaces for database administration that include administrative management of schema, indices, access privileges, views, triggers, business rules, security options, transactions, and so on.

I have listed the most important features for the core of the standard. It may also include features aiming at interoperability with other environments, such as bindings to popular programming languages, data externalization and exchanging formats (perhaps in XML), facilities for Web Services, facilities for dynamic Web pages, etc. Because now there are hundreds of various languages and technologies that may be interfaced with the standard, the choice is a matter of good prediction of their importance in the future.

Question6:
LINQ is leading in database API innovation, providing native language data access. Is this a suitable standard for ODBMS? Why?

Mike Card:
LINQ looks like it has a lot of promise in this area. We (the Object Database Technology Working Group in OMG) are currently evaluating LINQ vs. the stack-based query language (SBQL) developed at the Polish-Japanese Institute for Information Technology to see how these technologies compare for handling complex queries. SBQL has proven to be very good for complex queries and is being deployed in several EU projects, though it is unknown to most American developers. We are doing this evaluation to ensure LINQ is a good foundation for developers of applications that require complex queries, and is not too "small-scale" in its current form. We also want to hear from the LINQ community on plans (if any) to include update capability in LINQ and we need to be sure there are no surprises for parallel transaction execution.

Jim Paterson:
LINQ has a lot of momentum, and is becoming a widely accepted way of interacting with a database from a .NET app. If LINQ is your query mechanism, it becomes relatively easy to slot any kind of data store into the app with very little recoding - useful if you want to try out an ODBMS to check performance against an existing relational database, or to prototype with something like db4o while leaving yourself the option to switch to another database for production. LINQ is arguably becoming a de facto standard for ODBMS in that if a vendor wants people to use his ODBMS product in .NET apps then that product had better support LINQ. Then we need LINQ for Java also…

Kazimierz Subieta:
LINQ presents a valuable research result and a commercial product. It has many fans and success stories. For this reason it is worth its own standardization. LINQ is based on the idea of extending of functionality and syntax of programming language to process collections via queries. For this reason is presents substantial progress in comparison to the idea to embed queries in programs as strings, or to

the idea of adopting native Java syntax to represent queries. Only integrated query and programming language, with new syntax, semantics and functionality developed from scratch, is able to overcome the infamous impedance mismatch. From this point of view LINQ and SBQL are on the same positions.

However looking at points that I have listed in the answer to Question 4, LINQ is far from fulfilling all expectations behind the object database standard. Moreover, our recent comparison of LINQ with SBQL shows some shortcoming of LINQ. Below I list some of them.

1. LINQ object model is much limited in comparison to e.g. ODMG, UML and SBQL models
2. The syntax of LINQ is much more complex than the syntax of SBQL. In average, a LINQ query is two times longer (in terms of lexical units) than an equivalent SBQL query. Simple example ("Get departments together with the average salaries of their employees"):

   SBQL: Department join avg(employs.Employee.salary)

   LINQ: from d in Department select new {
       dpt = d,
       avg = (from e in d.employs select e.salary).Average()}

   A specific syntactic convention (a mix of SQL clauses with lambda expressions and with postfix method calls) causes that many LINQ queries are totally illegible.
3. LINQ forces the use of auxiliary variable names (such as *d, dpt, avg, e* in the above example). In SBQL, similarly to SQL, they are optional.
4. Lambda expressions that are used in LINQ are perceived as too complex. To a big extent, the genericity implied by lambda expressions and higher-order methods could be difficult to utilize in languages such as C# and Java, because these languages are not fully type polymorphic. Lambda expressions introduce substantial syntactic overhead with no functional gains for 99% of queries. Query optimization may reduce the genericity at all.
5. LINQ is not a stand-alone language. In .NET it is a syntactic extension of C# and Visual Basic. Married with Java it requires changing Java syntax. Using native Java syntax for representing LINQ queries will result in limitations of LINQ functionality and/or modification of LINQ syntax, making queries even more illegible.
6. Till now, specification of LINQ includes the syntax and examples. There is no formal semantics. Lack of formal semantics makes inferences on query optimization methods much more difficult or impossible.
7. Concerning very large databases LINQ (LINQ to SQL) is a slave of the relational model. Complex mappings between relational and object-oriented models can be much limited due to two fundamental problems: performance and view updating. A complex mapping may give no chances to SQL optimizers. A mapping is equivalent to a database view, leading to the well-known (and not fully solved) view updating problems. It can be anticipated that for processing very large databases future LINQ programmers will use data structures and queries that are 1:1 syntactically compatible with relational structures and SQL.
8. Versions of LINQ (LINQ to SQL, LINQ to entities, LINQ to XML) show a bit doubtful design where the front-end query interface is not separated from back-end middleware for accessing external resources. In this way in the nearest future we can expect dozens of LINQ variants (LINQ to CORBA, LINQ to J2EE, LINQ to RDF, etc. ?)
9. Many LINQ operators exist in several syntactic and semantic versions (aggregate functions, join, etc.) and with variable number of parameters, making LINQ much more complex than necessary. This design style is in contrast with SBQL, which is based on a minimal set of operators that can be orthogonally combined.
10. LINQ still makes the subdivision between programming expressions and queries. This is illogical, because the typing system is the same. In particular, LINQ queries cannot be used as left-hand-side parameters of updating statements (at least, I didn't find an example) and cannot be passed as *call-by-reference* parameters of methods.

Summing up, LINQ is an interesting proposal promoted by rich and successful company, but I have doubts if it able to satisfy the goals of object database standardization. From the point of view of very large databases, LINQ is an addition to the family of ORM-s. The time will show if it wins in this competition.

Question7:
When object databases are a suitable solution for an Enterprise and when they are not?

Mike Card:
They are not suitable when the engine is intended primarily for use in single-threaded embedded systems (optimistic concurrency control is a good indicator of this as I mentioned earlier).

An object database would be suitable for use in an enterprise system if it was really good at large-scale data management, i.e. the engine was designed to handle large volumes of data and many parallel transactions. Some object databases are not built like this, they are designed for use primarily in single-threaded embedded applications with fairly small data volumes and as such they would not be good candidates for enterprise applications.

Besides the technology used in the database engine itself, a good enterprise object database would need database maintenance tools (e.g. taking database A offline and replacing it with database B, updating or fiddling with database A and then bringing it back on-line, scheduling backups of databases and replicating databases between sites etc.).

Jim Paterson:
Object databases are not all alike, in the same way that relational databases are not all alike. Some products provide high performance, scalability and concurrency, while others focus on small footprint, rapid development and zero administration. Each has its own place within or at the edge of the enterprise. What they do have in common is that when an enterprise application is based on a complex object model, the object database becomes a more compelling solution. On the other hand, if data needs to be shared between legacy applications, then the tight language integration of an object database is less attractive.

Kazimierz Subieta:
It depends on a kind of business to be supported, on responsibility of the software for this business, on the scale of a database, and so on. There is no reason that object databases have less potential than relational databases, just otherwise. All conceptual advantages of relational and object-relational databases over pure object-oriented databases (sound mathematical background, proper scalability, possibility and efficiency of query processing, possibilities for distributed databases, and so on) are fantasies of some people, mostly from marketing departments of big relational vendors, who has direct interests in promoting such fantasies. Technical capabilities, however, depend on the scale of investment into the software development and for this reason big and rich relational vendors have substantial advantage.

Currently the biggest advantage of relational systems concerns many non-technical aspects, such as understanding the technology by wide software development community, academic and non-academic teaching programs, maturity and stability of the technology, market position and popularity, etc. For such reasons I do not recommend to use object database systems for typical business applications. There are however, new areas of applications such as Web portals, CASE tools, scientific data processing, multimedia databases, etc. that object databases can be applied. They can also be efficient in situations when a project is not well defined and we can expect a lot of changes during the design and operation of an application. In my opinion, the biggest advantage of object databases is reducing the mismatch between object-oriented modeling tools (e.g. UML) and databases and the mismatch between object-oriented programming and databases. The mismatches are ugly from the aesthetic point of view, but the have also severe consequences for the productivity of programmers. Impedance mismatches have especially negative impact on the maintenance of big applications. A single, universal, homogeneous and non redundant object model for software analysis and design, databases and programming reduces the need for complex mapping between different models, supports smooth transition between analysis and design models into implementation models; thus in consequence has big potential to reduce the complexity of software development and maintenance.

Question 8:
Future direction of object databases. Where do they go?

Mike Card:
The answer to this question depends on where object programming languages themselves go. Up to this point, programming languages have not included the concept of persistence, it is always included

as a “foreign” thing to be dealt with using APIs for things like file I/O etc. This is a very 1960s view of persistence, where programs were things that lived in core memory and persistent things were data files written out to tape or disk.

The closest thing to true integration of persistence I have seen is in Ruby with its “PStore” class. I would like to see persistence integrated even more fully, where objects can be declared persistent or made persistent a la

```
public class myClass {

 persistent Integer[] myInts = new Integer[5];
 Integer[] myOtherInts = new Integer[2];

 public void aMethod() {
 myOtherInts.makePersistent();
 }

}
```

and the programming language itself would take care of maintaining them in files and loading them in at program start-up etc. without any additional work from the programmer.

Now there are obviously challenges with this as this small example shows. What does it mean to initialize a persistent object in a class declaration? Is the object re-initialized when the program starts up? Or is the persisted value retained, rendering the initialization clause meaningless on a subsequent run of the program? Should persistent objects be allowed to have initialization clauses like this? What are the rules about inter-object access? Must persistence by reachability be used to ensure referential integrity? Can a “stack” variable (i.e. a variable declared in a method) be declared or made persistent, or must persistent variables be at the class level or even “global” (static)? Are these questions different for interpreted languages like Ruby which do not have the same notions of class as languages like Java? These are computer science/discrete math questions that will be answered during the language design process which will in turn determine how much “database” functionality ends up in the language itself.

If persistence were fully integrated into an object programming language in this way, then the role of an object database for that language might be to just provide an efficient way to organize and search the program’s persistent variables. This would reduce the scope of what an object database has to do, since today an object database not only has to provide efficient organization and search (index and query) capability, but it also has to make objects persistent as seamlessly as possible. Of course, this “reduction in scope” would only be possible if the default persistence mechanism for the programming language was implemented in a way that was efficient and fast for large numbers of objects.

Jim Paterson:
Object databases have been around for quite a long time, and in that time they have not been able to establish themselves as a mainstream data storage solution. Nevertheless, some vendors have been, and continue to be, successful within fairly specific markets. More recent entrants have pushed object database technology into new markets, embedded and mobile systems for example, and these are exciting areas for growth. In the mainstream of object persistence, there are signs of a growing recognition that object databases can provide a useful alternative to relational databases and mappers. To continue growing, it’s important that object databases support the techniques, platforms and tools that developers want to use, while offering unique advantages in terms of simplicity, performance and application integration. Re-engagement of the academic research community can also play an important part driving the technology forward through innovative developments.

Kazimierz Subieta:
My vision of the future of object databases that we are trying to accomplish within the ODRA system and SBQL is the following:

1. The unified and conceptually homogeneous object-oriented application development environment that includes tools for object-oriented analysis and design, with smooth transition to object-oriented prototyping and then to object-oriented programming that integrates object-

oriented database features. All the tools follow the same object data model that is non-redundant, complete and not burdened by past database theories and ideologies.

2. Rising the level of abstraction in programming through a minimal and powerful object datamodel, orthogonal persistence, programming via a query language smoothly integrated with programming capabilities, powerful programming and database abstractions such as modules, procedures, functions, classes, updatable object oriented views, triggers and business rules.
3. Distributed architectures (data intensive grids) enabling any configurations (client-server, P2P, etc.), efficient distributed transaction processing and distributed query optimization.
4. Integration of external heterogeneous service and data resources via dynamic run-time wrappers (mappers) rather than by language bindings or by API-s that assume inserting queries as strings into programs. No impedance mismatch due to subdivision of applications on the business layer and the middleware layer.
5. New architectures of applications that includes 3 kinds of specializations in programming: (1) database programmer that prepares a database schema and implements persistent database abstractions such as stored classes, stored procedures, updatable views, triggers and business rules; (2) application programmers that prepare application code using all the published database features in the integrated query and programming language; (3) administrative programmer that acts during application operation and dynamically manages access and architecture options such as accepting a new external resource, providing access rights for applications and users, performance and integration facilities such as resolving fragmentations of data, transparent replicas, preparing sub-schemas for users and applications, etc.
6. Easy web programming that includes access, composition and construction of Web Services, generating dynamic web pages, etc.
7. Easy and manageable generic programming via linguistic reflection based on a metamodel, with the possibility of dynamic generation of a program and executing it immediately within the same application.
8. Semi-strong typechecker that can assure full strong typing of the code, but on the wish of the programmer the strength of the typing can be relaxed.
9. Rich library of access to multimedia of various kinds.
10. Dynamic security and safety system that will allow the database administrator for quick changes of security rules after discovering new threats.

##