The Object Database Language "Galileo" - 25 years later.

By Renzo Orsini

Associate Professor
of Databases and Information Systems
Computer Science Department
Ca' Foscari University of Venice, Italy

Published in ODBMS.ORG, February 4, 2010.
www.odbms.org

Sometimes a look at the past can be useful to put in the right perspective a hot new discussion. So I thought when Roberto Zicari, editor of ODBMS.ORG, asked me a brief note about our work on Galileo object database language in light of the discussion on the new wave of "data stores", such as "document stores" and "NoSql databases". So, to follow Roberto's suggestions, I will present a brief history of the project, and then I will try to draw some considerations on this discussion based on my experience.

The early days

The Galileo project was born at the Department of Computer Science of the University of Pisa, Italy, in the first half of the 80's, by a group led by Antonio Albano and which comprised, in a period of time of almost twenty years, besides myself, a long list of researchers and students, so long in fact that I address the interested reader to this page where they are fully listed together with the whole project's bibliography: A History of the Galileo Language : http://www.dsi.unive.it/~orsini/Universita/GalileoHistory.html

Galileo was born in a period where the research on different aspects of programming languages was flourishing: the first paper on Galileo was published in 1982, like, for instance, the persistent programming language PS-Algol by Malcom Atkinson et al., while C++ and Ada were made available in 1983. In those years there was an increasing awareness about the importance of concepts like modules and generic programming, static type checking and type inference, polymorphism, and persistent programming.

The persistent programming, in particular, was then intended in the sense of: "what features should be in a language that allow the persistence of data to be really transparent with respect to the computation?", and not: "which is the best way to make a computation on persistent data, given a certain language?". A premise to this approach was that persistent data should have an ad-hoc store manager, maybe without the ACID properties of the DBMS (the acronym ACID was coined in 1983!). And for this reason, various ad-hoc data managers were developed and tested, like the Chunk Management System, or the Persistent Object Store.

The motivations behind the project

While we were well aware of such projects, our approach was radically different, since we had another objective in mind: we wanted to design a modeling language, namely a *conceptual modeling* language, as it is stated in the title of the most important paper of the first "phase of life" of Galileo: "Galileo, a strongly-typed, interactive, conceptual language" (which appeared in ACM Transactions on Database Language in 1985, with authors Antonio Albano, Luca Cardelli and myself).

What can you do with a conceptual language? Of course, you can write conceptual models for databases, models which are, however, executable. In other words, prototypes. We were in effect supporting, as the title of another paper said: “A prototyping approach to database application development”, and so we developed a main memory implementation of the language which covered almost all the aspects of the so-called Galileo 85 language.

The conceptual modeling features were described in the following way in the paper:

- data defined both declaratively, with abstraction mechanisms (aggregation, classification, and specialization), and procedurally (derived data);
- semantic integrity constraints, both standard (such as keys and mandatory values) and those described by a general-purpose constraint specification language;
- operations to give the behavioral semantics of the data in the schema.
-

Such features were integrated in a language which was:

- expression-based,
- strongly and statically typed (and with a limited form of type inference),
- higher order (with functions as denotable values),
- with concrete and abstract types to model structural and behavioral aspects of database “elements”,
- with type hierarchies to support the specialization abstraction mechanism of semantic data models as well as a software development methodology by data specialization,
- with classes as modifiable sets of database elements,
- with modules, to organize a conceptual scheme in meaningful and manageable units, so as to deal with data persistence without resorting to specific data types such as files of programming languages, and to deal with application-oriented views of data in a similar way to the view mechanism of DBMSs.
-

The rationale for such choices was that we wanted a language in which to write both rich and complex data models, as well as sophisticated executable prototypes, and in which to write complex applications, so that the language was due to have all or most of the features considered, at that time, important in programming language design.

Objects coming in

After the publication of the paper, in a very short time, the world of programming languages was radically “invaded” even more by the object-oriented languages, and the term “object-oriented database” was starting to be used (although the Object-Oriented Database System Manifesto was still a few year in the future). Memorable was a workshop “on persistent objects” in 1985 in Appin, Scotland, as well as a year later another one on “Object Oriented Database Management Systems”, in Pacific Grove, Monterey, California.

And so we realized that what we designed was in effect, with very few modifications, a language for object oriented databases, or, as we called it in the Pacific Grove workshop: “A strongly typed, interactive object-oriented database programming language”.

Such “objectification” of Galileo, and the diffusion of the first prototypal object database systems posed us a new problem: if we would implement a persistent object store for such a language, how it should be? While we were not so actively working on this, some progress was made in the years after, with the persistent store for the successor of Galileo, Fibonacci.

Meanwhile, the work continued on modeling and linguistic aspects, in the hope of integrating new abstraction mechanisms useful for database design inside the framework of the Galileo type system, which was more and more object-oriented. At the beginning of the ’90, we were ready to include in the language two new abstraction mechanisms: objects with roles, to model real world entities which change their role during their lifetime (and which can possess more than a role at a time); and object views, to enrich the range of use of a database by different applications. It was technically difficult, but with it was done with contributions of a new member of the team, Giorgio Ghelli, and we felt that this new work, together with a few other important aspects, was so significant that we could change the name of the language, so that Fibonacci was born.

In 1995, exactly then years after the paper on Galileo on TODS, a paper on Fibonacci was published on the VLDB Journal (authors Antonio Albano, Giorgio Ghelli and myself): “Fibonacci: a Programming Language for Object Databases”. The other important aspects of Fibonacci where: a) a shift of paradigm about relationships among sets of data (which were modeled not with the aggregation abstraction mechanism, but through the association mechanism, where an association is a set of tuples of objects and other values); b) a general trigger mechanism, and c) a shiny new persistent object store, operated upon through a Persistent Hierarchical Abstract Machine, or PHAM. Due to several factors, including an unhappy implementation choice (we used Modula-3 as implementation language), the work on the PHAM was ended a couple of years after.

Life after life

And Galileo? Well, a second life was ready for it. Antonio started a project with students to develop a lightweight implementation (i.e. of a simplified version) for personal computers for didactic purposes: in fact we used Galileo in teaching data base design, and we would like to have an implementation at hand for the course projects. Such implementation was so successful for its goal that over the years it was extended by smart students and included not only all the previous language, but objects with roles and views (as well as a simple form of session persistence), and, under the name of Galileo 97, it is still maintained today (even if with some difficulty due to the use of Pascal as implementation language). Readers interested in the current Galileo can look at the short Guided Tour in the project home page The Galileo Object Database Language and its main memory implementation.: http://www.dsi.unive.it/~orsini/Universita/galileo.html

A bit of reflection

So, after many years and many changes in the research fields touched by our project, after looking at the it from a certain distance, my first consideration on the nowadays discussion on new data stores is that it is a discussion largely about architectural aspects, i.e. (I would say “again”), not on modeling aspects neither on language issues. This is not so different from the past! When we developed and then presented our languages, people would often reply that the important thing was the “system” and its efficiency. And now it seems to me that the discussion is again full centered about system topics.

Maybe it is true that architectural aspects are the most important things, and that our work was the classical academic research, with few, if any, consequences on the real world. However, I think that a little more attention on modeling issues would help in clarifying some of the discussions. For instance, the use of certain abstraction mechanisms, with related implementation techniques, would lead to more “natural” and “efficient” solutions for certain classes of problems.

And even if I think that the programming language issue is all the more a delicate issue, should we all get stuck to the point that designing a new language for databases is no more attempted, since it is considered a losing battle against the use of well established programming languages and techniques? Sometimes a new language gives us a new way of thought, and this alone is an invaluable result.