A Conversation with Erik Meijer and José Blakeley

Photography by Craig Harrold

In any discussion about ORM (object-relational mapping), Microsoft's approach is inevitably a part of the conversation. With LINQ (language-integrated query) and the Entity Framework, Microsoft divided its traditional ORM technology into two parts: one part that handles querying (LINQ) and one part that handles mapping (Entity Framework). To understand more about these technologies and why Microsoft took this approach, we invited two Microsoft engineers closely involved with their development, Erik Meijer and José Blakeley, to speak with *Queue* editorial board member Terry Coatta.

Meijer is an accomplished programming-language

The Microsoft

PERSPECTIVE

ON ORM

designer who has worked on a wide range of languages, such as Haskell, Mondrian, X#, C-Omega, C#, and Visual Basic (his

personal favorite). He runs the Data Programmability Languages Team at Microsoft, where his primary focus has been on removing the impedance mismatch between databases and programming languages. One of the fruits of these efforts is LINQ, which not only adds a native querying syntax to .NET languages such as C# and Visual Basic, but also allows developers to query data sources

other than tables, such as objects or XML. Some readers might recognize him from his brief stint as the “Head in the Box” on Microsoft’s VBTV.

Blakeley is lead architect in the SQL server engine working on server-side programmability, scale-out query processing, and object-relational technologies. He joined Microsoft in 1994 and since then has been an architect of several of Microsoft’s data-access technologies. Like Meijer, Blakeley’s main focus is on the impedance-mismatch problem. He served as the lead architect for the ADO .NET Entity Framework, which works with LINQ to raise the level of abstraction and simplify data programming. Before joining Microsoft, Blakeley was a member of the technical staff at Texas Instruments, where he developed the Open OODB (object-oriented database) management system for DARPA.

TERRY COATTA I think that a number of our *Queue* readers really don’t understand how LINQ differs from what developers have been doing up until now. A lot of people are embedding SQL queries directly into their application code. Can you tell us what’s different about LINQ and why developers should care about it?

ERIK MEIJER While, superficially, LINQ might look like you’re embedding SQL queries in your code, actually it’s radically different. What LINQ really does is allow you to query arbitrary collections, which could be tables, objects, in-memory objects, or XML.

The secret sauce behind LINQ is what we call the standard query operators. If you have a data source on which you can define these standard query operators, then you can query it using LINQ. Think about it like this: you have SQL, which is based on relational algebra. Now imagine that we abstract from the relational part and have a query algebra that is represented by these standard query operators.

So, now the languages have this query syntax that they translate into calls to these query operators, but each data source can give a different implementation of these operators, allowing you to query over a wide range of data sources. One example is querying over tables, but it’s very different from using SQL.

The other problem with embedding SQL inside the language is that it’s not really well integrated, so SQL is embedded but it has a different syntax and type system. You’ll have variables in your program you want to use inside the embedded SQL, and maybe the other way around, because you’ll want to use the results that you get back from the query in your program. There’s usually quite a nasty boundary between the two worlds, but with

LINQ it’s completely integrated into the language, so the type system—everything—is that of the host language. It’s a seamless experience.

Those are the two big differences. The fact that it’s integrated in the language means that you get IntelliSense, compile-time type checking—all those things that are much harder when you are embedding SQL inside the language.

TC You talked about an abstract query algebra. Is it a stretch to apply that algebra to different data sources such as relational tables and XML, or do you think that it’s a pretty natural match?

EM I think it’s a pretty natural match. Let’s take the two concrete ones—SQL and XQuery. The syntax is slightly different. In SQL you start with **SELECT** and then you do a **FROM** and a **WHERE**, and in XQuery you start with a **FROM** and end with **RETURN** instead of **SELECT**. But conceptually you’re doing exactly the same thing.

JOSÉ BLAKELEY If I may add a little, at the outside level, it’s the same type of collections-based operation. If there is a difference, it would be that in the relational world you would be going over collections of records. In the XML world you would be going over collections of objects that model the infosets for XML. It’s a collection of elements, a collection of attributes, and those kinds of things. In essence, you are performing operations over collections of things.

TC I would like to step sideways for a moment and talk about the Entity Framework. A lot of systems are built where the developer issues an SQL query, brings some results into a data table or dataset, and binds the result of that into UI components. What’s different about the Entity Framework? Why should developers care about it?

JB A very common problem that application developers face is what has been called impedance mismatch. Usually the data model used by the application side of the problem is richer and more directly associated with the business problem at hand. The vocabulary is very closely related to the needs of the business—your customers, your line items, your products, and your inventory—whereas in the database you’re dealing with similar concepts, but the artifacts of database normalization and schema design get in the way of what the applications really need.

All of a sudden, database concerns start being inserted. These are foreign to the application, so every single application has to have a layer that bridges the gap between the world of data and the world of the program. Tons of application developers see that mapping, if you will, as part of the application.

In the past there have been many attempts to remove the impedance mismatch. In the late '80s and early '90s, object databases and persistent programming languages tried to bridge that gap and were successful—maybe not from the business point of view, but from a technical point of view. They were able to provide a seamless experience between the world of the program, the type system of the programming language, and the world of persistence.

Object database systems failed because they didn't have strong support for queries, query optimization, and execution, and they didn't have strong, well-engineered support for transactions. At the same time relational databases grew their capabilities and the world continued to gravitate around relational database systems.

Later on, various kinds of technology tried to bridge that gap. ORM systems try to generalize that mapping problem by providing tools that do it automatically on behalf of the user. The Entity Framework is an object-relational mapping technology that provides a general solution to the complex mapping problem to enable the user to focus on the core business problem and not on the system and infrastructure aspects of the problem.

TC You mentioned object-relational mappers. A certain portion of our audience has worked with products such as Hibernate or NHibernate and other commercial ORM systems. One of the interesting characteristics of LINQ and the Entity Framework is that they divide traditional ORM into two pieces: one part handling mapping and one part handling querying. Is that a correct view, and why is this separation reasonable?

JB Several OR mappers bundle these two concerns together, and that actually makes sense when the only problem you're trying to solve is how to bridge the gap between the application and the database.

But we should also look at another very broad class of mapping scenarios. We are building database management systems and data services around SQL Server—data services such as replication, reporting services, and OLAP

(online analytical processing). These all provide services at higher semantic levels of abstraction than does the relational model.

For example, many years ago SQL Server Replication introduced the concept of *logical records*. When you are setting up a configuration to replicate customers, their orders, and line items, you want to be able to replicate not just orders and not just line items, but the relationship that exists between the two. We invented logical

records to address that issue and provide a more convenient solution to the users of the replication component.

In a similar manner, Microsoft SQL Server Reporting Services enables users to define reports in terms that are at higher levels of abstraction than the tables and relations. They actually have a model in terms of entities and relationships, because if you are trying to build a report that has many parts to it, you want to be able to model those relationships before producing the report. In the guts of the reporting services system is a data model and mapping technology that allows us to solve that mapping problem.

The tools to manage SQL Server are another example. SQL Server Management Studio, for example, allows you to build a user interface over the objects in the database—your servers, your tables, your views—and to browse through different database schemas.

That particular set of tools contains an object-relational mapping component. If you look at each one of these data services as an instance of an application, yet again we find that we have to build special object-relational mappings to meet the needs of those data services.

Thus, when we looked at the impedance mismatch both of applications and of data services, we realized that for the data-services case, you don't want objects with methods and behaviors. What you want is a value-based, richer structural data model.

By value-based, I mean the ability to have high-level constructs such as entities and relationships but without the behaviors. Just as the relational model is a value-based model, we felt that we needed to provide a layer of

abstraction that is richer in terms of entities and relationships. Therefore, the Entity Data Model and the Entity Framework became a natural layer of abstraction that we felt had to be built, and it's at that level of abstraction where the mapping between richer-level entities and semantic concepts such as inheritance is abstracted.

Now the Entity Data Model, which is the formalism that defines the Entity Framework value-based layer, is very close to the object data model of .NET, modulo the behaviors. We decided to let the Entity Framework take care of all the mapping concerns and then just build thin programming-language veneers, or wrappers, over entities to expose a variety of programming-language bindings over this infrastructure. You can have a binding over C#, Visual Basic, or XML—you pick the programming model that you want.

EM I would like to point out that there's a deep analogy with how I explained LINQ in the beginning. We're trying to abstract not one particular case where you go from tables to objects, but rather a wide variety of different things for different uses. So instead of having a one-off thing, we're trying to generalize this concept so that there are many other situations in which it's applicable.

TC There has been a lot of press in the developer com-

munity about functional languages—how they are up and coming and how they are going to help us do multicore. My perspective is that something like LINQ is almost a hybrid system because it's bringing elements of functional programming directly into the imperative programming environment that most of us are operating in. I'm curious, do you see it that way, and how do you see this evolving?

EM Yes, definitely. There is a very clear link to functional programming. For example, if you really want to get geeky, LINQ is just another way of doing what is known in Haskell as *monads*. So it's really taking concepts from functional programming and bringing them into object-oriented languages.

The functional programming aspects are also visible if you look at the way these LINQ operators—the standard query operators—are defined. They all take functions as their argument. If you have a **WHERE** operator, for example, it takes a predicate to check whether to filter out something. What is a predicate? Well, that's a function from some value to a Boolean. The way you parse that predicate is by a lambda expression, and, again, that comes directly from functional programming.

Another aspect to consider is tuples, or anonymous types. Of course, they are used in SQL for rows, but a lot of functional programming languages also have tuples, and again we use them when we're doing projections.

Type inference is another example where functional programming concepts come in. It has been in languages such as ML and Haskell for at least 30 years, and we have now moved it into C# and Visual Basic.

Speaking of many-core programming, there's a project called P-LINQ, which is an implementation of LINQ to run over many-core. Again, you describe all these things in a more declarative way using functions that you pass around, which allows the compiler to take this query and manipulate it, optimize it, and run it in parallel.

In a sense it's similar to what SQL Server does when it takes the query and runs it in parallel. But now we've moved that to the realm of the programming language. So, yes, there is definitely a mixture or infusion of functional programming and object-oriented programming.

I would like to stress that functional programming and object-oriented programming are more similar than most people realize. A lambda expression or a delegate is really a special class that has one method called **invoke** or **apply**, or something similar. Because it's a class that has only one method, you can add syntactic sugar such that you don't have to mention the **apply** method but you can supply the arguments directly to the delegate,

and then the compiler inserts the applier—in this case, invoke.

If you look under the hood, the difference between object-oriented programming and functional programming is not that big. It's not some weird Frankenstein monster. It really blends very nicely, and that is because under the hood it just fits.

TC You've both been talking about layers of abstraction and the fact that there are these new layers in the software. This is often challenging for programmers when they try to performance-tune or debug systems, because now there are these perhaps quite complex layers of software sitting underneath their code, and what those layers of software are doing can matter to them from a debugging or performance-tuning perspective.

We use NHibernate where I work, and we've found that even though we can see what the plans for the queries are, sometimes it's difficult for us to understand how what we did at the level of NHibernate turned into this series of queries that we now see coming out the bottom. From a tuning perspective, we can say, "OK, I can see that I want to change this, but what do I do in the programming language to actually make that happen, because there's a layer in between now and we're not sure what it's doing?" Is that a concern for you? Are you doing anything in particular in the context of LINQ and the Entity Framework to help make those layers more transparent or to enable the developer to inspect them?

JB One of the problems in NHibernate is that the mapping and the programming binding are all bundled together. The specific algorithm that is used, for example, to map a particular expression to inheritance is known only by somebody who deeply understands the way in which Hibernate has been implemented. That is not the case in the Entity Framework, where the mapping between tables and high-level concepts, entities, and relationships is defined declaratively through mapping view expressions. It's a view layer, so every single semantic concept—entities, relationships, inheritance hierarchies—is defined formally in terms of database views so there is always a declarative mapping layer that the developer has full control of. As you're developing your entity data model and the mapping to the underlying relational model, the actual mapping is described openly in terms of views. There is much more transparency in the way in which the mapping operation is happening. That helps the user in case there are problems.

For example, if your system supports inheritance, some application developers may define very deep inheritance hierarchies. Mapping very deep hierarchies becomes

very complex, so when developers see the kind of view expressions that are generated to map that deep hierarchy, they may realize that they should review the depth of the hierarchy and use a different kind of mapping.

TC I understand that having the mapping materialized as views certainly adds a layer of transparency to it, but even so, as a programmer you would have to have a reasonable knowledge of what is going on with that. For example, if you have a deep inheritance hierarchy, that's typically going to show up as a deep set of joins to pull together the data from the multiple tables in order to create that finalized entity. But unless you understand that, unless you have a sophisticated understanding of the fact that inheritance and joining are related to one another, it's going to be difficult for you to improve the performance of those things. Is it possible to create tools that help the programmer more directly understand the connections?

It would be nice to have something that would tell you that this portion of the LINQ query is associated with this portion of the SQL that's coming out the bottom. The programmers don't have to deduce it because the system is actually telling them what's going on there.

EM Let me give you a two-part answer. People probably will now say, "Oh, Erik is just an academic." But the thing is, if you look historically, the kind of concerns that you raise have always been there whenever a new level of abstraction is introduced. When we moved from assembler to higher-level programming, people said, "I don't have all the control that I have on the assembly language level and I don't trust the compiler to do it, so I want to look at the assembly." How often now when you're writing in high-level languages do you look at the assembler code that the compiler is generating?

Over time, as the mapping from the abstraction to the next layer down gets better, this concern will go away and the tools will get better and people will get a better feeling of what's happening. The points you raise are valid, but I think it shows that we are very early in this game. We're just introducing this level of abstraction and people are not used to it, and the tools maybe are not very mature. That's where these concerns come from.

TC Where are developers going to get tripped up in LINQ and the Entity Framework? We have some new technologies here, and they're introducing relatively sophisticated layers of abstraction. When developers start working with these things and maybe haven't had a lot of experience with persistence or with databases or other things like that, what should they be careful of? Where should they be looking to avoid falling into problems?

EM I can give you a concrete example. When you're

writing queries in LINQ, they are what we call *deferred*. A more popular way of saying that is that it uses *lazy evaluation*, so when you write your query, nothing happens. When you send the query to the database and you get back the results and iterate over them, *then* something happens.

If you have side effects in your query, however, that's a problem. Suppose in your **WHERE** clause you have a side effect or an exception is thrown. Now that side effect happens at a very different point and place and time than when you wrote your query. We all know that using side effects inside queries is not a good idea, but since we now have integrated these queries in a programming language, it becomes quite easy for people to write side-effecting code.

Suppose you want to open your file system, look at all your directories, and write a query over your directory structure, filtering out all PostScript files or Word files. In .NET you have the disposable patterns. People would write

USING, open my file system, do a query

but since that query is deferred, by the time you're iterating over the results, you have disposed of the object and certain bad things will happen.

This combination of deferred execution and side effects can definitely trip people up. But, again, SQL people have known forever that you should not do side effects in queries.

JB LINQ is a brand-new development in programming languages. The idea that you can express your intent in terms of higher-level query expressions is really new. I believe that in bridging the gap between procedural programming and set-oriented programming it's going to take a while for developers to move away from the procedural aspects.

If we could actually try to minimize the number of "for eachees" that are part of programs and think hard about writing expressions that can be pushed as far as they can to the database, that would be a way to write better programs. Then, maybe even introducing compile checks in the compiler to prevent you from using side-effecting expressions or functions in queries would also be a step toward helping developers avoid tripping themselves up.

Beyond LINQ and the primary language surface, mapping is complex. We need to figure out the best way to explain the mapping without overwhelming the developer with massive query expressions that represent those mappings, and we need to come up with a more graphi-

cal way of describing what the mapping is doing. This may actually go a long way toward helping. Complexity is one of the challenges we need to overcome.

TC With respect to the Entity Framework, what common mistakes do you see people making? Erik mentioned the side effects, and that's really easy to see. You wag your finger at people and you say, "Don't do that!" Does the Entity Framework have any of those kinds of common traps or errors that people fall into?

JB When people see that they can start modeling their concerns at a high level of abstraction, they sometimes overuse concepts. Having deep inheritance becomes a common mistake. It's very hard for people to balance the value of inheritance with its complexities. Coming up with best practices on that will be a very good thing for us to do.

TC As I mentioned, I've been using NHibernate in a .NET environment for about four years, and I'm curious about your responses to some interesting things I've observed with regard to LINQ and the Entity Framework.

There's a particular style of using NHibernate that I think is a radical shift for programmers. The style that we use is to isolate all changes to persistent objects within an NHibernate session. We pull objects into that session, do some operations on them, and then commit them back out to the database.

The result is that most of the programmers who work for me never see a lock, which is totally different from the way that we used to program. Previously in a multi-threaded, concurrent system, you would build your objects and explicitly encode the locking in them. You would decide when you were going to acquire the locks and when you were going to release them, and there were often complicated protocols to make sure that locks were acquired in the same order. Now with NHibernate, there aren't any locks at all. There literally are no locks around our objects because all of the locking and concurrency control is essentially deferred down into the database. I'm interested in whether you see this as a radical shift for programmers.

EM Maybe what you're observing is really the power of optimistic concurrency and transactions, and the database world has known transactions for a long time. In some sense, transactions are a much easier way of dealing with concurrency than locks because transactions still allow you to think in a serial way. That's the whole ACID (atomicity, consistency, isolation, durability) thing. You can think of your world as applying some big update and it will either work or not, and if it works it will be isolated, and so on.

One thing that people are working on is taking those concepts of transactions and moving them more into the programming language. You want to do transactions in memory. This allows you to get rid of a lot of the locks you see in pure, kind of normal concurrent programming. I think there's a nice technology transfer there where we're moving not only queries into the programming languages, but also optimistic concurrency and transactions as well.

TC I think the key is essentially that notion of moving the transaction into the programmer's world, and that's really what has happened for us. We say to programmers: start a transaction, go touch whatever objects you need to touch, and commit it when you're finished.

What's interesting for us is that we're still working with SQL Server in a mode where we're not using optimistic concurrency control. One result is that although we have made it so programmers are no longer concerned about concurrency, we have created this situation where we very frequently get deadlocks in the database. It has been challenging for us, particularly in the context of NHibernate, because we end up wanting to be very careful about the locking that's happening in the database. You really think that the future of this is more in the optimistic concurrency control end of things?

JB Definitely. In fact, this trouble that you see in the NHibernate way of marking things perhaps is because the actual mapping is not entirely abstracted. In the Entity Framework, we know how the entities and the objects are assembled together in a set-oriented way from the underlying tables. Therefore, when the time comes to push an update down through mapping to the database, we know precisely how to sequence the ordering in which these updates need to be applied to avoid deadlocks.

TC I guess it's really more of a question of whether those kinds of features need to bubble up into the query language. What we're finding is that when we're making queries at the level of NHibernate, we really want to be able to provide locking hints on those queries—to say you're going to need to do this as an update lock, not as a shared lock, when you read this object in initially because we know that we're going to write it back later on. If you don't do it as an update lock, we know that we're going to end up getting a lot of deadlocks from multiple people coming in at the same time.

The real question is, does the notion of concurrency control, the locking modes, have to float up to the highest levels of the system to allow people the degree of control needed to do these sorts of things?

JB I don't think so. If you have a system that finds it

necessary to expose those physical concepts up to the program, then you've defeated the purpose of the abstraction. If you think about programming at the level of the database, say, in the form of a stored procedure using PL/SQL (Procedural Language SQL) or Transact SQL, you don't need to expose those kinds of things. They are available but their use is discouraged because the system should be able to handle that abstraction on behalf of the user. So my approach to this would be to resist the need to expose those locking constructs to the programmer as much as I can, and really work on cleaning up and fixing the mapping abstraction.

TC Imagine you've got the same kind of system that we've been talking about. It's basically objects and transactions in nature with that notion of "serializability" and stuff like that. You've got a workload coming in and there are many independent threads of control. What happens is that those independent requests are all sharing a common set of objects. What we found when we built this system in the most straightforward way was lots of aborted transactions because of conflicts between the requests coming in.

Do programmers have to worry about this, or do LINQ and the Entity Framework again have a way of helping us deal with this kind of stuff?

JB This is another scenario that is somewhat related to the need for tools, not only for mapping but also to be able to handle these potential concurrency issues. If we were able to build a workload-oriented tuning wizard that could take the workload of the application and its concurrency characteristics and predict where you're going to have lock contentions, where you're going to have deadlocks, that would be the way to solve that issue or to help mitigate it. Lacking those tools, the developer has to figure out a way to overcome those real situations.

EM Generally, you can raise the level of abstraction. To me that means taking away irrelevant details, but that means there are still details that are relevant. For example, what you describe is a relevant detail—the underlying system gets in trouble because you get a deadlock. By definition that has now become a relevant detail, so you have to take care of it. There's no magic. There's no such thing as a free lunch.

TC I guess we've all been involved in developing long enough to know that. ☺

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

© 2008 ACM 1542-7730/08/0500 \$5.00