

March 4, 2008

## Java Object Persistence: State of the Union

*In this virtual panel, the editors of **InfoQ.com** (Floyd Marinescu) and **ODBMS.org** (Roberto V. Zicari) asked a group of leading persistence solution architects their views on the current state of the union in persistence in the Java community.*

### *Panelists:*

- **Mike Keith:** EJB co-spec lead, main architect of Oracle Toplink ORM
- **Ted Neward:** Independent consultant, often blogging on ORM and persistence topics
- **Carl Rosenberger:** lead architect of db4objects, open source embeddable object database
- **Craig Russell:** Formerly the spec lead of Java Data Objects (JDO) JSR, architect of entity bean engine in Sun's appservers prior to Glassfish

### *Q1: Do we still have an "impedance mismatch problem"?*

**Mike Keith:** As long as an object model is being used to read from and write to a relational data model then the impedance mismatch will exist. It becomes a problem when there are no strategies, technologies or tools to help overcome the mismatch, or when the mismatch results in a convoluted design or prohibitively bad performance. There will always be some scenarios that lead to this situation simply because of the nature of the mismatch. Technologies and tools can help a great deal, and do meet the needs of the vast majority of applications, but they can't turn water into wine.

**Ted Neward:** Absolutely. So long as we have different 'shapes' to data in their transient and durable forms—such as the current dilemma where we try to take transient data in the shape of objects and store it to a relational durable data store—we will have an impedance mismatch problem. Object to relational, object to hierarchical/XML, hierarchical to relational, they're all impedance mismatch problems and they all end up facing the same basic problem: pounding round things into square or triangular holes. One solution

is to incorporate those different 'shapes' into our tools, such as adding sets as a first-class concept to our programming languages to make it easier to work with sets in the data store, but even that's only going to take us so far.

**Carl Rosenberger:** No, unless you are stuck with a relational database. Let's take a look from the viewpoint of using two specific languages, Java and SQL:

In Java you can be very productive by writing small, simple and concise methods that are easily understandable. In a relational database there is no concept of navigation and you have to know the table schema by heart to get anything out of the database. Imagine you want to find out the salary of an employee's boss. Using Java and an IDE with autocompletion, you can write code like the following within seconds:

```
employee.department().manager().salary();
```

Using SQL, it could take you minutes to figure out the SQL statement for the same task ...

```
SELECT e2.salary FROM  
  employee e1, employee e2, department,  
WHERE  
  e1.id = current_employee_id  
AND  
  e1.department = department.id  
AND  
  department.manager = e2.id
```

... and would you be sure that the syntax is correct and that you did not mix up the two references to the employee table?

When you have to use both Java and SQL to write an application, you lose development time, quality, efficiency and performance for the interface between two very different languages.

The overhead for having to maintain two very different codebases more than doubles the required work and it inhibits refactoring and agile processes.

If you embrace object technology all the way by using an object database there is no impedance mismatch.

**Craig Russell:** It depends on your business domain and your language. If you're working in C++ and exploit multiple inheritance to model your domain, there's still a lot of work to store domain objects in relational databases.

The most significant factor is whether you're trying to use Java domain objects to represent what is already in your database, or whether you're trying to use the database to store Java domain objects. In the former situation, most relational schema can be represented quite well with Java domain objects. There is more of a chance of a mismatch if you're trying to store Java domain objects in a relational database.

In this case, there remain a few areas of impedance mismatch. For example, modeling a Collection<Coin> is difficult in a relational schema. In ODMG, we modeled a Bag, which is the logical equivalent of a Java Collection. In the Java Collection Framework, there is no concrete class that has the right Bag semantics. The closest is an ArrayList but when persisting this type, extra information has to be artificially created (handling duplicates means introducing an artificial column in the database or using a table that has no primary key).

Another area of impedance mismatch is inheritance. Relational schema don't exactly match the Java concept, although many relational schema in practice include a discriminator column that is used to distinguish between rows of one subtype and another. This technique predates many of the object-relational mapping solutions that now exist.

***Q2: In terms of what you're seeing used in the industry, how would you position the various options available for persistence for new projects?***

**Mike Keith:** If you start with the assumption that the vast majority of projects need the data store to be a relational database then people's persistence demands mostly ending up falling in 1 of 3 categories. There are always the few edge cases and some slight variants, but these 3 categories pretty much subsume mainstream applications that persist Java objects:

1. *JDBC* – A large number of developers still rely on JDBC for various reasons. Maybe they don't need or want a mapping layer, or maybe their employer simply won't allow it. Whatever the reason, JDBC is still a viable option and when used in appropriate ways, and for the right reasons, can be a good fit for some applications.
2. *Lightweight and custom frameworks* – Sometimes people that require hand-coded SQL find that if they put a thin layer on top of it then they can increase the readability and maintainability of their code. For these applications a simple JDBC framework, or a home-grown solution that is suited to the development and coding practices of the application, can fit the bill.
3. *ORM* – An object-relational mapping tool includes a mapping metadata model to dictate how the object state gets stored into and queried from the database. There are a great many folks that are happy with ORM technology and use it to meet their needs. The Java Persistence API (JPA) is the standard ORM API in Java EE as well as non-enterprise Java environments.

With the rise in popularity of SOA comes the class of application that simply needs to send Java objects to an XML sink. The sink may end up taking the form of a web service port, a BPEL process, or a queue for later processing. The ability to map objects to XML is solved by standards like JAXB and SDO, and there are projects like Eclipse Persistence Services that implement all of these standards, plus add more functionality besides.

**Ted Neward:** There's obviously a variety of options, and each has its own advantages and disadvantages. Mike does a good job of summarizing three of the core options, to which I would add a few more:

1. *Stored procedures*, in which we put all storage and retrieval logic (and sometimes data-centric business logic) into executable code inside the database, where any application--be it written in Java, C++, C# or Ruby--will be subject to the same centralized logic. This is often combined with the straight JDBC/call-level-interface approach or an O/R-M to simplify calling the sproc and harvesting the results.
2. *The Object Database (OODBMS)*, storing actual objects themselves into the database, and which Carl goes into in more detail.

Naturally, there's various hybrid approaches to the above, but this is how it generally splits out.

**Carl Rosenberger:** When people speak of databases, they usually mean enterprise databases that service many different users, applications and programming languages. For these legacy databases SQL will be around for a long time to come.

In the near future we will have 5 billion mobile phones and only 2 billion PCs. The market for applications on devices (and databases on devices ! ) will outgrow the market on PCs.

For databases on devices it is common that:

- they only serve one programming language,
- there is no DBA to tune access,
- the user does not type ad-hoc queries into his device by hand.

If applications on devices are written in an object-oriented language, is there any single reason to use a relational database or SQL?

I don't think so.

Microsoft has made a very strong statement with LINQ: They go away from SQL and towards making database access an integral concept of the programming language.

This is the future for new projects, simply because it is the most efficient way to develop and to maintain code.

With the adoption of LINQ, a functional language concept will become an accepted mainstream standard to query databases.

Being involved in the development of an object database myself, I am very delighted. The fun part for us: LINQ permits the use of normal C# method calls in queries: You can do navigation. Since object databases physically store objects very similar to their representation in RAM, object databases can optimize for LINQ much better than relational databases. Where an object database can use a direct pointer for navigation, a relational database needs to maintain and look up two indexes to be able to query over a join.

LINQ is a dream for object database vendors. All of a sudden there is a standard for querying object databases and it's a piece of cake to outperform any relational system by far.

We would love to see a standard like LINQ for Java. With Native Queries we provide a very similar approach: We make it possible to use 100% plain Java to formulate database queries.

I can judge the demands of the industry best in areas where we see a strong pull for our product:

- applications for devices with constrained resources, where high performance has to come with low memory consumption,
- applications with complex classes with many attributes and/or deep inheritance hierarchies,
- applications where classes are frequently refactored and maintenance of a relational database and mappings would be a nightmare,
- applications that need to go to market very quickly,
- Rich-Client applications, written with technology like Eclipse RCP, JavaFX or Silverlight.

Since we seem to have the happiest users in these areas, I conclude that this is where today's object databases play out their advantages most.

**Craig Russell:** The lowest level of abstraction with Java and SQL is the ubiquitous JDBC programming model. It gives the best level of control and allows you to exploit every feature of the driver and database. But it is also the hardest to get right and doesn't lend itself to abstraction.

Many programmers find that using libraries to implement pooled resources improves performance without having to write more code, but the tough part of getting the SQL right; and handling prepared statements and result sets is still a challenge.

A higher level is found in tools like iBATIS, which take away a lot of the drudgery of manipulating prepared statements and result sets, but still require writing SQL, which is highly database dependent and still requires work to use the mapped domain objects.

The highest level of tooling that I'm aware of in Java is found in object-relational mapping solutions such as Enterprise Java Beans Container Managed Persistence, JPOX, OpenJPA, Hibernate, TopLink, and Cayenne. Java Persistence API and JDO are JCP standards which are implemented by many of these solutions. These tools have a pretty high level of automation and allow both creation of Java domain classes from a database schema as well as creation of database schema from Java domain objects.

If you go outside the Java language, tools such as Grails and Ruby on Rails offer very high functionality and additionally provide web framework implementations that get programmers out of the tedium of hand crafting every call to and from the web server.

***Q3: What are in your opinion the pros and cons of these existing solutions?***

**Mike Keith:** This is a question that is the stuff of large books, and one could go on ad infinitum about when and where you should use one versus the other. The short version is that I believe they all have their place, and that no one solution is going to be the best choice for all applications. My only hope is that people will do a little homework before they choose the solution that they plan to spend as much as 30% of their application resources on.

**Ted Neward:** That is a very long answer, easily more complicated than I can put down here.

**Carl Rosenberger:** One of the key advantages of object-oriented practices is that they reduce the cost for long term maintenance. If you use an object database, you extend this advantage to your data model (your classes, what else) and to all the code that interacts with it.

Using an object database, all your code can be written in one programming language. Refactorings can be fully automated, using modern IDEs.

Object databases allow 100% of your code to be compile-time checked, and this includes all code that interacts with the database.

Since object databases can handle object references in exactly the same way as a virtual machine does in RAM (using one-way pointers, not indexed tuples) object databases can be a lot faster than relational databases.

Native embedded object databases, that run in the same VM as the application, allow profiling and tuning bottleneck use cases deep down into the database. That makes them the perfect choice for embedded use on mobile phones and on handhelds, where machine performance and memory are limited.

SQL is well understood. There is a choice of mature SQL databases with a long history and there are many mature tools to help to write, tune and query SQL systems.

SQL is an acceptable standard for ad-hoc querying and ad-hoc aggregation of data.

**Craig Russell:** Generally, the higher up the abstraction level you go, the faster you can write and deploy applications.

But to get the last penny of performance out of your database investment, some applications will require direct use of SQL.

So you can't get away from a deep understanding of the specific SQL dialect that your database provider accepts. But most of the tools allow you to do most of your work at a high level of abstraction, only requiring you to dive into the SQL for corner cases.

***Q4: Do you believe that Object Relational Mappers are a suitable solution to the "object persistence" problem? If yes why? If not, why?***

**Mike Keith:** Object relational mapping technology is simply both a reality and a necessity in a world where the huge majority of corporate data lives in relational databases and object-oriented programming paradigms have become the norm for application development. Because of this reality the question is not really how to solve the object persistence problem, but how to solve the problem that exists in roughly 97% of application development scenarios where the application is written in Java and must retrieve from and store its data in a relational database. So far, object-relational mapping is the most flexible and practical solution to have emerged. Is it suitable? It

certainly is for the scores of folks that successfully use it. Is it universally the best solution? I am pretty sure that if you asked enough people you would get some that disagreed with it.

**Ted Neward:** I think they work for those developers who are willing to put Productivity and ease of development above performance and relational accessibility.

Unfortunately, too often a project discovers that surrendering Performance or a solid relational model and accessibility is not as acceptable as the developers think, and by that time, it's usually too late to easily replace the O/R-M with something more "relational friendly" without major refactoring-and-or-rewrites.

**Carl Rosenberger:** O-R mappers are a workaround. To some degree they help to reduce the work to write SQL by hand.

No workaround comes without a cost. Here are three areas where O-R mapping tools are worse than ideal:

- Performance  
O-R mapping tools make applications slow. Objects need to be fully loaded into memory to run object-oriented business logic.
- Complexity  
O-R mappers introduce own side-effects through the way they cache and transfer objects. Developers lose time for understanding how their O-R mapping tool behaves in concurrency scenarios.
- Design Constraints  
No O-R mapping tool can deal with all constructs that OO programming languages offer, simply because not all constructs have an equivalent in the relational world. There will always have to be tradeoffs for designing the application class model to work well with an O-R mapping tool.

Workarounds never are a cure for a problem. They are only an intermediate step until a true solution is found.

**Craig Russell:** For many applications, ORM are a perfectly fine tool that balances the needs of the developer with the requirements of managing a database data center resource. This is most evident where the database schema are carefully controlled and managed. And at the other extreme,

when the application programmer controls the database schema by writing Java classes and generating schema from them, one big danger is that the application will not perform well and it may be tough to redesign the domain classes to generate the right schema, and then data migration becomes an issue.

***Q5: Do you believe that Relational Database systems are a suitable solution to the "object persistence" problem? If yes why? If not, why?***

**Mike Keith:** Again, relational databases are not normally the answer, they are the question. Unless you are a brand new company developing a green field application then chances are that the app is going to have to get its data from a relational database. In fact, most brand new applications even turn to relational databases for storage because of the maturity of the technology and track record for scalability and reliability. The question is how to access the data in that relational database yet write the business logic in an object-oriented language like Java?

**Ted Neward:** I think whether they are suitable or not is missing the point—the modern IT infrastructure expects and needs enterprise-wide data to be stored in a relational database for a variety of reasons, and developers need to find ways to get their objects into that data. However, doing so directly from the user interface is often not necessary, and an intermediate object database which then updates a relational store in a batched or periodic basis can be a powerful productivity enhancement during development.

**Carl Rosenberger:** No. The impedance mismatch is the problem, not the solution.

**Craig Russell:** In conjunction with any of the Java to DB tools, yes.

***Q6: Do you believe that Object Database systems are a suitable solution to the "object persistence" problem? If yes why? If not, why?***

**Mike Keith:** Object databases are like the betamax of the 20<sup>th</sup> century. They did one thing quite well, but just didn't catch on. They became marginalized, not because they were terrible, or hard to use, or slow, although some were indeed all of those, but because they did not serve the heterogeneous

application needs of corporations. While IT applications are continually changing, both in function and the technology used to build them, the data they access stays pretty much the same. The data must be easily accessible from all technologies and applications, a requirement that relational databases met but object databases did not do so well at.

**Ted Neward:** Sure, so long as the data stored in the OODBMS doesn't need to be Directly accessed by non-object technologies. This remains the biggest drawback to OODBMSes.

**Carl Rosenberger:** Of course. Which problem? With an object database you can store any object with just one line of code. There is no additional work for maintenance of tables and mappings. Querying is an integral part of the programming language. Code that interacts with the database is typesafe, compile-time-checked and automatically refactorable, just like all other code.

**Craig Russell:** The market for Object Databases has more to do with the application domain than with the programming model used to access it. Much of the rationale for Object Databases is the performance of the system with specific domain models and workloads. For extremely demanding applications, object databases can outperform relational systems by orders of magnitude. The challenge in these cases tends to be the tooling around the object databases such as query and report generation tools, event propagation, synchronization with other databases, and fail-safe operation.

***Q7: What would you wish as a new research/development in the area of Object Persistence in the next 12 months?***

**Mike Keith:** It's amazing how long data can survive. One of the problems with data survival is that the proprietary storage schemes that were invented, many of which were created before relational databases even became popular, must still be used to access the data. Ask system integrators what they have to deal with and you'll likely get a longer discussion than you bargained for.

Furthermore, XML has become ubiquitous as a global data language so it turns out that the service-oriented objects of this era need to regularly change shape from objects to XML to relational records, or even records in

funky custom data stores. The research of the next year or more is going to have to solve that problem in a way that is satisfactory to developers at each step along the way. It won't quite be one persistence to rule them all, but hopefully a set of unified persistence solutions that are consistent, integrated and provide what is needed at every level.

**Ted Neward:** Integration of relational and set theory concepts into programming languages.

**Carl Rosenberger:**

(1) Merging object-orientation and functional languages

I like functional programming languages for their powerful concepts to work against sets, with list comprehensions. Still I do believe that objects are the very best way to represent state.

I see a bright future for programming languages that merge functional concepts and object-orientation. Both Scala and LINQ are great steps in the right direction.

It is no rocket science to analyse list comprehensions and to run them against database indexes. Conceptually this is quite similar to db4o Native Queries. I hope we get to try this out with Scala list comprehensions....if not I am sure someone in our community will.

(2) New Concurrency Concepts

The trend in hardware development to go multicore will require new programming language concepts to handle concurrency.

Software Transactional Memory and Actors are two great approaches that we may see adopted in mainstream programming languages. Both approaches deserve thought on how they can be integrated best with database processing.

(3) Managing State

In an object-oriented world we have state on clients and in distributed systems: Objects.

I have seen surprisingly little database research about the problem how objects on clients can be kept in sync with the committed state on the server.

Relational databases provide concepts for isolated reads and for locking but they don't tackle the real problem. They work as if there would be no state on clients. This is another aspect of the impedance mismatch.

In a concurrent world, I don't believe in locking.

I think the root of the problem can be tackled better by either pushing changes to clients or by working with object versions that are consistent for a certain point in time.

It would be very interesting to test both approaches empirically to find criteria why and where one or the other approach works better.

**Craig Russell:** Tooling and application generation, especially integration with front end scripting code generation, remain weak points of object persistence.

**Q8: If you were all powerful and could have influenced technology adoption in the last 10 years, what would today's typical project use as a persistence mechanism and why?**

**Mike Keith:** I would have made EJB a POJO model to begin with and saved the world from 8 years of pain. I also would have made it work with Smalltalk. ;-)

**Ted Neward:** I would never dictate a solution to a project. "From each technology, according to its abilities, to each project, according to its needs."

**Carl Rosenberger:** Let's look at the future and not at the past: Working for db4objects I hope to be in the happy position to have some influence on the next 10 years. Today's new mobile phone and device projects (Android ! ) should use object database technology, if they don't do that already.

I am not an expert for largescale enterprise databases but I doubt that grid databases are nearly as efficient as they could be. Since most of today's mature largescale database engines are written in C or C++ and since these

languages do not provide high level concepts for distribution and concurrency, I believe that maintenance and improvement of these systems must be very tedious and inefficient.

Probably it is still very well possible for a few smart people to write the worlds fastest huge distributed database with a reasonable effort that would pay off very quickly. If I was all powerful, I would like to give this project a try at some stage. Maybe it makes sense to wait until the latest emerging concurrency concepts have matured a little more.

Relations do not define:

- where consistency begins and ends
- who owns the master copy of a tuple
- how data is to be distributed in a cluster
- how prefetching makes sense
- how data is physically clustered

Principally I see advantages for concepts that take all the knowledge that is present in the programming language and transfers it to the database.

From a theoretical standpoint the eventual adoption of object databases may make sense even for large enterprise databases if all enterprise applications accessing the database were written in compatible object-oriented languages.

Ideally there would be a common object-oriented domain model that could be shared between all enterprise applications worldwide.

Actually I thought about this more than 10 years ago. If I was all powerful everyone would be using such a model today.

...and then there would be no question at all that an object database is the most natural persistence solution for all applications.

**Craig Russell:** Today's typical project would use a world class IDE in conjunction with plugins for an object persistence layer that could use either relational or object databases for object storage.

The IDE could generate complete web framework code including javascript artifacts for highly interactive rendering and user interface.

The Java VM would allow generic instrumentation of classes specifically for the purpose of change tracking and to avoid vendor-specific byte-code transformation. Runtime performance tools would dynamically tune applications with no direct human intervention.

**Q9:** *Any parting words about this topic?*

**Mike Keith:** The interesting thing about object persistence is how passionate people are about it. People seem to perk up their ears and draw their swords at the mere mention of persistence, and community forums need only publish some critical view in order to get an endless stream of comments during an otherwise quiescent time period.

It may be because persistence has such a dramatic effect on the performance of an application, or it may just be that persistence is so darn interesting, but after 18 years of studying and working in the persistence domain I still feel like it is worth spending time on. One thing that everybody seems to agree on is that developers that do not pay attention to the persistence aspects of their applications do so at their peril.  
Thanks for the invitation to participate.

**Ted Neward:** Thanks for having me.

**Carl Rosenberger:** I believe in bionics: We can learn a lot from how nature solves problems. Let's take a look at how nature has organized knowledge: It uses lots of slow redundant systems (humans) with constantly improving highly efficient communication techniques. Any node can communicate with any node and the direction and the strength of connections can be adjusted to match current needs.

If we take this as the model, smart grid computing that adopts and learns from mistakes is the future, also for databases.

Now how does this fit in with the common domain model I described above? Very well, I think. Humans have a common object-oriented domain model that is comparatively stable over time: Natural language.

Because natural language was used we can still read and understand texts that were written millenniums ago.

We are very lucky that old Greek texts were not written with OR mappers. Evolution is a slow continuous process. Eventually the best concept wins.

Thanks for the invitation to participate.

**Craig Russell:** Thanks for the invitation to participate.

## About the authors

### Mike Keith

Mike Keith has over 15 years of teaching, research and practical experience in object-oriented and distributed systems, specializing in object persistence. Mike Keith was the co-specification lead for EJB 3.0 (JSR 220) and a member of the Java EE 5 expert group (JSR 244). He co-authored the premier JPA reference book called Pro EJB 3: Java Persistence API and has over 15 years of teaching, research and development experience in object-oriented and distributed systems, specializing in object persistence. He is currently an architect for Oracle TopLink and the Oracle OC4J Container and is a popular speaker at numerous conferences and events around the world.

### Ted Neward

Ted Neward is an independent software development architect and mentor in the Sacramento, California area. He is the author of a number of books, including "Server-Based Java Programming" (Manning), and the forthcoming "Effective Enterprise Java" (Addison-Wesley) and co-author of "SSCLI Essentials" (OReilly) with David Stutz and Geoff Shilling, as well as "C# In a Nutshell" (OReilly) with Peter Drayton and Ben Albahari.

Ted has a number of [technical white papers available](#) for free download , and his weblog lives at [neward.net/ted/weblog](http://neward.net/ted/weblog), where he pontificates on technical issues as the mood strikes. He serves on JSR 175, the Java Community Process specification for custom metadata in J2SE 1.5, and is a Microsoft MVP (Most Valuable Professional) for the C# team. He is also an instructor with [DevelopMentor](#), where he teaches and authors in both the Java and .NET curriculum, and currently he works as the Editor-in-Chief of [TheServerSide.NET](#), a community portal dedicated to enterprise .NET architecture and issues.

### Carl Rosenberger

Carl Rosenberger is chief software architect at db4objects, Inc. and the lead programmer of the db4o database engine. Rosenberger started db4o in 2000 when he felt the need for a simple and easy-to-use object

database to meet the needs of object-oriented software developers. He first offered db4o commercially in 2002 and thus laid the foundation for the incorporation of db4objects in 2004, when more than 100,000 downloads and some 100 customers had validated and embraced the product. Prior to founding db4objects, Rosenberger was chief software architect for APSIS Software and Lawconsult. He is member of the EJB 3 expert group (JSR 220) and focuses on object querying APIs.

## **Craig Russell**

Craig Russell is a senior staff engineer at Sun Microsystems. He is specification lead for Java Data Objects (JSRs 12 and 243) and leads the implementation team for its reference implementation and technology compatibility kit. He was the architect of the Container Managed Persistence component of the J2EE Reference Implementation and Sun Java System Application Server. Craig Russell is a member of the Apache Software Foundation, the chair of the Apache OpenJPA Project Management Committee, and a member of the Apache Incubator project responsible for bringing projects into Apache.

##