Editors' Corner

John Pugh

Paul White

PERSISTENCE. THIS ONE word causes more grief and costs more time in terms of system development with objects than any other single issue. In theory, the problem is a very simple one. Is there a way for an application to store and retrieve the data contained within its objects?

Will the persistence mechanism also maintain the relationships between the objects in addition to the raw data? Will it support object identity properly? And how fast (slow) will it execute?

Providing a suitable answer for all of these questions is extremely difficult. Persistence has always been an issue with object-oriented languages, Smalltalk being no exception. The first persistence mechanism designed for Smalltalk used the storeOn: method to ask an object to generate a string that, when evaluated, would return the object itself. Clever idea, but not very practical. The next idea was to use a Loader/Dumper style implementation, where an entire object (and all its parts) can be stored in a file, and later retrieved. This worked well as far as it went. The shortcoming was that an object being retrieved had no notion of the relationships it used to have with other objects, and vice versa. And it certainly did not scale in terms of size or maintenance.

It is our job to design systems that are easy to maintain.

The correct solution, of course, is to use a true database of some flavor. Be it IMS, Oracle, Sybase, Gemstone, Versant, etc., the idea is to store and retrieve objects using a facility designed to do just that.

So, what's so hard about this? As most of you know, the difficult part is not the "data" stored with the object. It is simple to store and retrieve data such as a name, phone number and date of birth of a customer object. The difficulty comes in maintaining the relationships between the objects as designed in our systems. For example, our customer object may have a reference to an account object (or many) as well as relationships with other customers (a bank would keep track of our relationship with our spouses). The challenge is how keep track of these types of relationships? And how can we support the dynamic nature of these relationships?

The quick answer, of course, is to use an object persistence storage mechanism. Tools such as GemStone, Versant, and others handle these complex and dynamic relationships with next to no effort. There are other issues to consider when deciding whether or not to employ these new persistence mechanisms, but ease of managing these relationships isn't one of them. However, most of us aren't in the position to entertain such a choice (and maybe we shouldn't anyway). Our organizations have invested significantly in other technologies that serve the overall organization quite well. Furthermore, since the data used by our Smalltalk applications is often shared with others, a more traditional route may be the only practical choice.

Using a relational mechanism poses some significant challenges. For example, determining how to map the relationships found in our object model to tables is nontrivial. The greater the difference between the object model to the data model, the more difficult the mappings can become. For example, many-to-many relationships (customers to accounts) require an intermediate table in the relational world, but in the object world, they're really a nonissue. Storing and retrieving objects from non-relational mechanisms is proving to be even more difficult. Many relationships that exist in objects are virtually impossible to duplicate in, for example, a CICS transaction.

Having stated the difficulties, many have been able to successfully bridge the worlds. Most of these solutions are "home-grown." Our concern with home-grown solutions is that even once we describe how to overcome the hurdles and discover the mappings required, we're still faced with two significant challenges. First, the execution speed can often be painful (especially with writes) if not very careful about the database commands generated (e.g., SQL statements or stored procedures). Second, and more important, is the effort that is required to maintain the mappings.

This point concerning maintenance is being overlooked by far too many shops who are building their own interfaces. Many are saying "I can build it myself" and they can. But the impact on the elegance of the implementation is going to lead to systems that are tougher to extend and difficult to understand. What many seem to fail to grasp is, even if they can understand it while they're writing the code, will the person coming behind them to maintain it be able to understand? And what about the person behind them? The real costs of our solutions will be seen down the road. We strongly believe it is our job as software engineers to design systems that are easy to maintain—if we fail, we haven't done our organizations any favors.

Enjoy the issue.