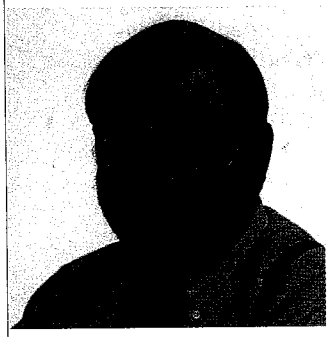

Ten Rules for Distributed Object Systems

BY ROGER SESSIONS

It's sad watching a client fail. Recently one of my clients failed. The company had attempted to build a relatively simple distributed object system. About 10 person-years had been invested in the project. The company had promised that the next major release of this product would be based on distributed objects. The project had high visibility. And the whole thing went down the drain.

Like most failures, this one was predictable. The developers had little experience in object-oriented programming, no expertise in distributed object systems, and almost no time allocated for training. They immediately started designing an overly complex system with no provision for testing or debugging.

As the second extension of the project due date approached, the project managers started getting desperate. The system was slow and lacked much of the committed functionality. What functionality it did have would occasionally work, but more often would hang or crash. No one had any idea why it worked when it worked, and why it didn't when it didn't.

So the managers made a desperate move. They called in a consultant: me. They told me they had to ship within two weeks. They wanted me to tell them how to fix their problems—in less than two days.

I spent a day with them reviewing the design. I had no choice but to deliver the bad news. The design was a mess. The implementation was hopelessly flawed. The best option was to flush it all and start again.

I tried to sound as positive as I could, given the depressing circumstances. I used phrases like, "This is an excellent prototype that should help clarify your goals," and "This is a really nice demonstration of a system bottleneck—see how all these little arrows are all pointing at the same object?" But the message was clear. The team had no hope of meeting its deadline. The system had no more chance of working than a dog has of flying.

This situation started me thinking. Obviously, this client had violated many of the basic rules of developing distributed object systems. But just what are these rules?

Over the many years that I've been in this field, I've designed, implemented, and consulted on many distributed object systems. I've given lectures at more conferences than I can count and spoken to more people than I can even guess at about using distributed objects. I've heard and witnessed many stories of success and many of failure. So what determines success or failure when using this technology?

When a distributed object system works well, it is beautiful and harmonious. When it doesn't, it's repugnant and irritating. So what makes one distributed system a symphony, and what makes another a cacophony?

I believe in 10 basic rules for implementing distributed object systems; rules so basic that anyone observing all 10 is almost guaranteed success, and anyone who ignores even one is headed down a dark path indeed.

Rule one

Understand what a distributed object is (and isn't). This is a funny rule, isn't it? Why would people use distributed objects without understanding what they are? However, this rule is commonly violated.

Many people begin with an incomplete understanding of object-oriented programming and then try to extrapolate this pseudo-understanding to distributed objects. Others who do understand object-oriented programming then assume distributed objects are the same thing. They are not.

Distributed objects are more like components than objects. They are large things that know how to perform specific functions for you. They generally provide some kind of business capability.

Let's consider a few of the traits that differentiate nondistributed objects (such as C++ objects) from distributed objects.

- **Purpose.** The purpose of a nondistributed object is to manage the complexity of the data and algorithms required to solve some programming problem. A good example of a nondistributed object is a collection class. The purpose of a distributed object is to perform a related set of business functions for multiple remote clients. A good example of a distributed object is an inventory object.
- **Client view.** The client view of a nondistributed object is a class, which includes the definitions of methods, the algorithms for those methods, and the internal data of a nondistributed object. A given class can have only one implementation. The client view of a distributed object is an interface, which only defines the behaviors that clients can expect the distributed object to perform. A given interface can (and often does) have many implementations.

- **Performance.** The cost of a method invocation on a nondistributed object is measured in tenths of a microsecond. For a nondistributed object, you can ignore the invocation costs of methods when determining its performance. The cost of an operation invocation on a distributed object is measured in milliseconds. You must consider this cost very carefully when analyzing the overall system performance.
- **Complexity.** A nondistributed object is typically low in complexity and is often composed of only a few hundred lines of code. The design goal of nondistributed objects is to be simple. A distributed object is usually quite complex and composed of perhaps hundreds of thousands of lines of code—a single distributed object may actually consist of dozens, or even hundreds, of local, nondistributed objects, but if so, their existance is invisible to the client.
- **Number.** Nondistributed-object–based systems are typically composed of a large number of classes (perhaps thousands) and an even larger number of nondistributed objects. Distributed-object–based systems are typically composed of a small number of interfaces (perhaps 10) and not many more distributed objects (perhaps dozens at most).
- **Location.** Nondistributed objects are always located in the address space of their clients. Distributed objects are never located in the address space of their clients.
- **Concurrency.** Nondistributed objects are only used by a single client and don't have to worry about concurrency. Distributed objects are used by a large number of clients and have to deal with complex concurrency issues.

Rule two

Use a standard. Distributed object systems by their very nature need to span languages, computers, operating systems, and network protocols. Your only hope of managing this complexity is to base your distributed object systems on well-understood standards with many available implementations.

The most widely recognized distributed object standard is the CORBA standard, based on work done by the Object Management Group (OMG). Fortunately, one of the industry's best implementations of this standard is available to OS/2: SOMobjects. SOMobjects will soon be available on all IBM platforms and many non-IBM platforms as well. OS/2 is therefore an ideal platform for developing distributed object systems.

Basing your distributed object system on the CORBA standard will give you the following important benefits:

- You will have the widest possible choice of machines on which to place objects. Objects that need close integration to databases can be placed on the database host. Objects that need high reliability and scalability can be placed on a Tandem machine, which has ported a version of SOM designed to support such features. Objects that make use of special AS/400 features can live on that host.
- You won't have to worry about underlying communications protocols. The CORBA architecture hides communications protocols under operation invocations. You will never know what communications protocol you are using.
- You won't have to worry about the language that is being used to implement your objects. Because CORBA is a well-accepted standard, it is or will be supported by most popular programming languages.
- You will have access to objects being developed by independent software houses. The CORBA standard is expected to enable a whole software components industry.
- You will have access to a host of well-defined object services, such as persistence, security, naming, events management, and many others. These services will simplify your systems development and increase the portability of your products.
- You will be independent of any one vendor. Although IBM's SOMobjects is one of the leaders in the field, there are at least five other significant competitors that offer CORBA implementations.

Rule three

Design distribution into the system from the beginning. If you have followed rule one, then you understand the difference between regular objects and distributed objects. Rule three says you should deal with the various distribution issues from the beginning of your design. You don't need to implement every design feature immediately. In fact, it's better to get the basics working before you worry about the details. However, you should understand and plan for the problems that may arise as you move to a highly distributed system.

You should consider the following issues, most of which are irrelevant for nondistributed objects:

- Where will your distributed objects live, and why?
- How will information pass between the distributed objects?
- What is the maximum throughput for each distributed object, and where in your overall system will bottlenecks occur?
- How many users must the system be prepared to accomodate?
- How will your system scale up when the number of users increases, anticipating, at worst, that your system actually works, and the whole world wants to use it?
- How will distributed objects be instantiated and de-instantiated?
- How will users find the distributed objects they need? Will they use a naming service, a trading service, or some other mechanism? How will the objects become registered with the appropriate services?
- How reliable do you need the object references to be? If a process has a reference to a remote object, and the server containing the remote object goes down, what are your expectations of the system?
- How will your distributed objects

keep their stateas synchronized with external databases?
- How will your distributed objects participate in transactions?

Rule four

Figure out how your distributed objects will interact. Typically, the interactions between nondistributed objects are fairly simple. A client object knows about a target object either by maintaining a reference to the target object as part of its state or by having the target object passed in as a parameter to one of the client object methods. The client object then invokes methods directly on the target object.

The interactions between distributed objects include many more possibilities, in terms of both how client objects find target objects and how the client objects invoke methods on the target objects.

Strictly speaking, a distributed client object never has a direct reference to a distributed target object. Instead, the distributed client object has an indirect reference that is interpreted by the server on which the target object lives. Thus, the server always has an opportunity to redirect the method to another object, or another server, should it choose to do so.

Even indirect references are not commonly used by client objects. A client object will more commonly use a naming service, trading service, or property service to find an appropriate target object with which to interact.

Sometimes, no direct interaction occurs between a client and its target distributed object, as when the interactions go through the event service. When using an event service, one object may simply notify the world that a particular event has occurred, and any object that wants to deal with that event is free to do so.

These examples are just a few of the possibilities for distributed object interaction. You need to understand what possibilities exist and which will best meet your needs.

Rule five

Avoid writing code whenever possible. The CORBA standard is particularly suited to code reuse. The opportunities for code reuse fall into two general areas: buying and wrapping.

Because you have been so clever as to build your system on a well-accepted standard (see rule two), you will have more opportunities to purchase prebuilt distributed objects. Because you have been so clever as to choose a standard that is object-based, rather than component-based, you can easily specialize these objects to do exactly what you want. This discussion is a bit futuristic, since the market for object-based components is still in its fledgling stages, but the future is bright.

When you cannot purchase, you can often wrap. Because CORBA has a clean separation between interface and implementation, a client has no dependencies on how a method is actually implemented. A method can be implemented by wrapping an executable program, an existing object method, a subroutine, a shell script, a stored function in your database, or almost any code package you can imagine.

Do not fall into the trap of thinking you must implement every distributed object operation from scratch. Search every nook and cranny for reuse opportunities.

Rule six

Prototype. Okay. You think you're smart. You've followed rule one and think you understand what distributed objects are. You've followed rule three and have done a careful distributed system design. You've followed rule four, and understand all the interactions between your distributed objects. But you aren't smart. You're stupid. If you know you're stupid, then there's still hope for you. If you think you know what you're doing at this point, then you're in serious trouble.

Now is the time to roll up your sleeves and prototype. Make sure the logic of your operation implementations actually works. Make sure that the object interactions work the way you thought. Make sure the servers' robustness matches your expectations. Make sure the system scales up as predicted. Make sure the performance of operations is as you expected. I assure you, you will find more design problems in the first two weeks of prototyping than you would in six months of arguing on whiteboards.

In reality, there is no such thing as skipping the prototyping step. People merely delude themselves into believing that they can do so. They will learn the hard way.

Don't be ashamed of being stupid. I am stupid, and I've probably been in this field a lot longer than you have.

Rule seven

Distribute incrementally. Testing and debugging are more difficult when objects are distributed than when they are nondistributed, and more difficult still when they are distributed remotely than when they are distributed on the same machine.

You should, therefore, locally test as much of your system as you possibly can. The technology for debugging and validating nondistributed objects is far more advanced than the technology for debugging and validating remote objects. Once you have everything working locally, move the objects onto other processes on the same machine. Once that works, move the objects onto other machines.

You may be tempted to skip this step, but you'll pay dearly if you do. You'll spend weeks trying to find problems that would've been obvious in local mode with a good debugger.

Rule eight

Spend your time designing, not choosing design tools. Keep in mind that distributed systems are composed of a relatively small number of complex objects. You really don't need complex design tools to design distributed object systems. I've seen many projects where protracted religious wars were fought over design methodologies. I've yet to see one project where the choice of a particular design methodology or tool made a bit of difference to the eventual success or failure of the project.

Some design tools claim to directly produce IDL (the language that describes the interfaces of CORBA objects). I say, "Who cares." IDL isn't that difficult to write. Use whatever tools you want. Just keep them simple. And don't waste a lot of time arguing about them.

Rule nine

Allow time. Many projects get into the distributed object arena thinking it

will allow them to develop their first system in a fraction of the time it would take using conventional technology. Please. Get real.

If this project is your first, you have a lot of learning to do. Your team's first learning task is to make sure they really understand object-oriented programming. Once they've gotten past that hurdle, they need to understand distributed objects. The leap from nondistributed object-oriented programming to distributed object-oriented programming is at least as great as the leap from procedural to object-oriented programming.

It will probably take people a long time to master these new ideas. They are going to make many mistakes (see rule six). Plan adequate time for learning the theory of distributed objects and the many parts of the CORBA architecture. Plan for lots of hands-on training time. Eventually, you'll get to the point where your system development time is dramatically reduced, but not on your first system.

Rule ten

Focus on expertise early. You cannot design and implement a decent distributed object system without some expertise on board—not only expertise in object-oriented programming, but expertise in distributed object systems.

If you have in-house experts, bring them in early. If you don't, hire a consultant and do so before you have your first design meeting. You may only need a few weeks or a few months of a consultant's time. Whatever you need, the cost of bringing in a consultant is minuscule compared to the cost of spending person-years going down rat holes and blind alleys that could have been avoided, or of building systems that fail the first time your customers try to use them.

Speaking as one who has been there, I can tell you: It's a lot more gratifying to help design and build a system that works well than it is to have to tell somebody, after the fact, that a brand new, multimillion dollar creation isn't worth the disk space on which it's written.

Epilogue

That's it. My 10 rules for developing and implementing distributed object systems. Perhaps I should have added an eleventh rule: Have fun. This is a great technology. It's fantastic to conduct an orchestra of distributed objects, each doing its own unique thing, yet cooperating in a wonderful, harmonious, symphonic movement.

OS/2 is an ideal platform for trying out this technology. Now go to it. Create a masterpiece. OS/2

Roger Sessions is president of Object-Watch Inc. His latest book is Object Persistence: Beyond Object-Oriented Databases. *Roger also publishes the SOMobjects Home Page (http://www.fc.net/~roger/owatch.htm) and an Internet newsletter called* ObjectWatch on SOM. *He can be contacted via e-mail at roger@fc.net.*

Why does this publication and 1600 others open their books?

Every year?

Believe it or not, some publications actually keep their subscribers undercover. They steadfastly refuse to let BPA International or any other independent, not-for-profit organization audit their circulation records. Who's to say their circulation is what they claim?

On the other hand, 1600 publications -- including this one -- are members of BPA International. Every year BPA International auditors scrutinize our circulation records and verify the number of our subscribers, their geographic distribution and other important information such as business and occupational data.

Our annual BPA audit helps advertisers determine if they are reaching the right people with their products' messages.

But more important, a BPA audit helps you, the subscriber. Because the more advertisers know about our circulation, the better they can provide you with information that meets your needs. Similarly, the more *we* know, the better able we are to give you targeted news and information.

BPA International: the proven leader in circulation marketing intelligence for business and consumer media.

270 Madison Avenue, New York, NY 10016-0699.
(212) 779-3200; fax (212) 779-3615.

BPA INTERNATIONAL®

Reader Service No. 37