Fuzzy

It's easy to transform objects into components and Web services, but how do we know which is right for the job?

If you are an object-oriented programmer, you will understand the code snippet in figure 1, even if you are not familiar with the language (C#, not that it matters). You will not be surprised to learn that this program will print out the following line to the console: woof

Now you may or may not be interested in making dogs say "woof," but the principles that make this dog bark are the same principles that run the most sophisticated banking systems in the world.

So why dogs? Dogs have two endearing qualities that make them worth studying over banking systems. First, they are much cuter than bankers. Second, their implementation is trivial and therefore less distracting to the point of this article. So back to myDog.

Boundaries

Objects, Components, and Web Services

ROGER SESSIONS, OBJECTWATCH

Fuzzy Boundaries

Objects, Components, and Web Services

With a few tweaks to our code and a toggle or two to a compiler switch, you can take this Dog class and turn it into a distributed Dog component. A few other tweaks and toggles will turn this same Dog class into an Internet-enabled Dog Web service. The compiler vendors have made it easy to take objects and transform them into either components or Web services.

Nice, yes?

Unfortunately, there is a problem: the compiler vendors have done too good a job. The ability to transform objects into either components or Web services is very powerful. Maybe too powerful.

Shakespeare said, "A power I have, but of what strength and nature I am not yet instructed" (*Measure for Measure*, Act 1, Scene 1). This, in a nutshell, captures the dilemma of objects, components, and Web services. The compiler vendors have given us the *power* of transformation, but no *instruction* as to when and how to use this power. The result is some very fuzzy boundaries between what should be an object, what should be a component, and what should be a Web service.

Before looking at why one should, or shouldn't, willy-nilly transform objects into components and Web services, we need to look more closely at exactly what it means to be an object, a component, or a Web service.

LOCATION AND ENVIRONMENT

Objects, components, and Web services have many features in common. Specifically:

- All are blobs of code that can do something.
- All have interfaces that describe what they can do.
- All live in a process somewhere.
- All live to do the bidding of a client.
- All support the concept of a client making requests by "invoking a method."
- All can be described by figure 2.

The differences among these three entities are primarily driven by two factors: location and environment. *Location* refers to the relative locations of the entity (e.g., object) and client, or, more specifically, the relative location of the processes in which the entity and the client live. *Environment* refers to the hosting runtime environment for the entity and the client (for example, IBM's WebSphere or Microsoft's .NET).

Objects, components, and Web services have many other differences (some quite unexpected), but they are mainly derivatives of location and environment. So let's start with these.

When both the entity and the client are located in the same process, the relationship is characterized as an *object* relationship. The

myDog Code Example

```
public class Dog                    // Define Dog class
 {
   public string SpeakToMe()        // Define SpeakToMe method
   {
     return "woof";                 // Dogs SpeakToMe by
   }                                // returning "woof"
 }
 public class TestDriver            // Test program
 {
   public static void Main ()
   {
     Dog myDog;                     // Declare myDog to be a Dog
     string whatHeSaid;             // Declare a local string variable
     myDog = new Dog();             // Instantiate myDog
     whatHeSaid = myDog.SpeakToMe();        // Invoke SpeakToMe on myDog
     Console.WriteLine(whatHeSaid);                 // Print out result
   }
 }
```

FIG 1

environments for the entity and the client must be the same, because a single process can't live in more than one environment. We could implement an object relationship in, for example, Java or C#, both of which are established object-oriented technologies.

When the entity and the clients are located in different processes, then the environment becomes the defining characteristic. When the environment is the same for the client and the entity, the relationship is characterized as a *component* relationship. You can implement a component relationship in, for example, WebSphere's EJB (Enterprise Java Beans) environment or Microsoft's .NET managed components environment, two popular component-supporting technologies.

When the environment is different for the client and the entity, the relationship is characterized as a W*eb service* relationship. Web service technologies include SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), UDDI (Universal Description, Discovery, and Integration), and others. Figure 3 contrasts objects, components, and Web services based on differences in location and environment.

Given this discussion, you might wonder why we have objects or components at all. Both object and component relationships could be implemented using Web service technologies.

The reason we still need objects and components is efficiency. We should do things as efficiently as possible given the constraints within which we must work. When we are not constrained by needing multiple environments, components are more efficient than Web services. When we are not constrained by needing multiple processes, objects are more efficient than either components or Web services. Web services are the least efficient of all of these systems, unless we are working under constraints that prohibit the use of either objects or components.

EFFICIENCY

Let's consider why objects are more efficient than components, and components are more efficient than Web services.

Since objects live in the same process as their clients, their method resolution (mapping of method invocation to actual code) does not span process boundaries. The method resolution can therefore be done using some type of optimized table lookup system, as shown in figure 4.

Components live in different processes from their clients. They typically use a transparent (i.e., invisible to the client) inter-process transport mechanism layered on top of the native object method resolution. Typically this mechanism includes a surrogate in the client process that accepts the method request, packages it into an inter-process communications package, and sends that package to a proxy living in the component process. That proxy receives the package, determines which method has been requested, and then makes that request through native object method resolution.

Components are slower than objects because inter-

Objects, Components, and Web Services

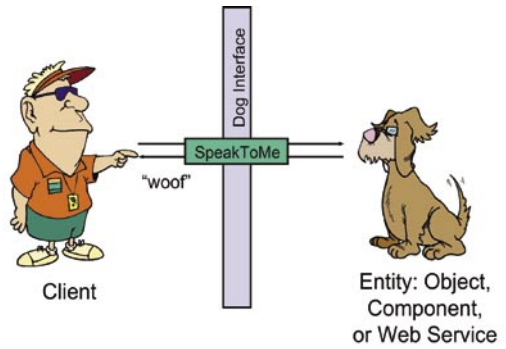

FIG 2

Location and Environment

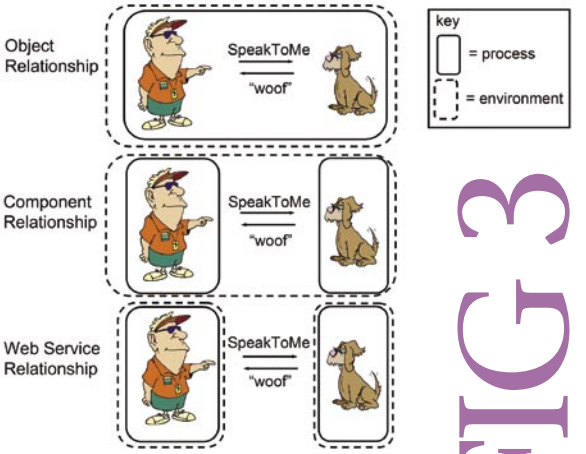

FIG 3

Fuzzy Boundaries

Objects, Components, and Web Services

process communications are slower than intra-process communications. Components do have one opportunity for optimization, however: leveraging native (e.g., .NET) communications protocols to move the request from the client to the component process. This is shown in figure 5.

Web services are separated from their clients not only by process boundaries, but also by environment boundaries. This means that they must use standardized Web services transport protocols (such as SOAP over HTTP) rather than the optimized native inter-process transport protocols. Web services transport protocols represent agreements among all of the environment vendors and are effectively least-common-denominator protocols. These protocols achieve the often important goal of environmental interoperability at the significant expense of performance. Web service method resolution is shown in Figure 6.

To summarize, object method resolution is the fastest because all communication occurs within a single process. Component method resolution is slower because it requires inter-process communications, albeit optimized inter-process communications. Web service method resolution is the slowest, because not only does it require inter-process communications, but it also uses least-common-denominator protocols for achieving those communications.

HOW DO YOU CHOOSE?

Clearly you want the fastest communications that will meet your constraints. Let's consider building a point-of-sale (POS) system. Your POS system may want to interact with an inventory system, say, to let it know that someone has just purchased an espresso machine and that now would therefore be a good time to decrement the espresso machine inventory.

There are many inventory systems in the world. When

Object Method Resolution

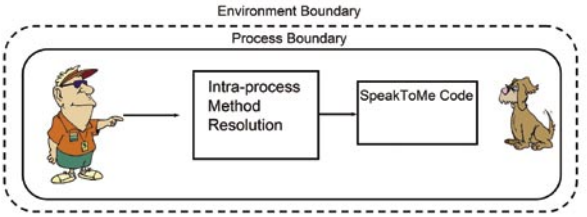

FIG 4

Component Method Resolution

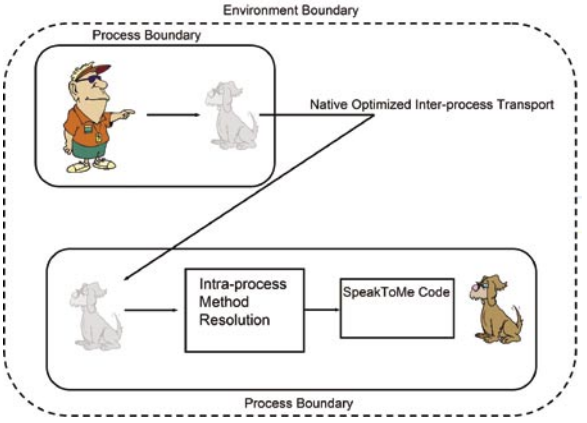

FIG 5

Web Service Method Resolution

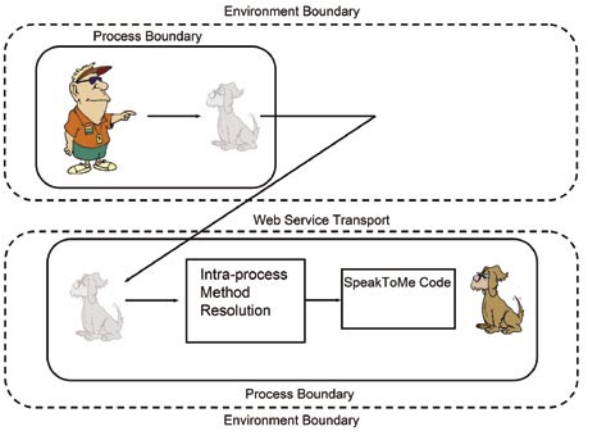

FIG 6

building the POS system, you may not know with which of these inventory systems you will eventually be interacting. Similarly, the inventory system, when it was built, probably had no idea which POS system would be the one letting it know about espresso machine sales.

Since the POS and inventory systems were developed independently, they may well have been written for different environments. The POS might be, for example, a WebSphere system and the POS a .NET system. Although it is possible the POS and the inventory systems were developed for the same environment (say, both for WebSphere), you don't want to take any chances. You should use environment-agnostic Web service protocols to connect the two.

Now let's look more closely at the POS itself rather than how it connects with other systems. The POS is built by a single group of coordinated developers. They are not likely to build part of the POS in WebSphere and part of it in .NET. They are going to choose one environment and stick with it.

This does not mean, however, that they are not using distributed programming. They probably do have different parts of the POS running on different machines and processes, perhaps because they are using a three-tier architecture. Each sales station is probably running its own machine/process; there may be another part of the POS that runs on a machine that consolidates sales across the store, another part that consolidates sales across the region, and yet another part that stores data in a central data repository. All of these require distribution; they just don't require different environments. Therefore, the most efficient technology available is the component technology.

Within a given process of the POS, perhaps hundreds of thousands of lines of code are bouncing back and forth. Objects are a great way to organize this code. Everything is happening within a single process, so you don't need the overhead of components and certainly don't need the overhead of Web services.

For the POS architect, it is not a choice among objects, components, and Web services. It is a matter of choosing which to use for what kinds of communications.

Good system architectures therefore don't look at objects, components, and Web services as mutually exclusive choices. Instead, they look at them as building blocks—all useful, but for different purposes. Web services are useful for tying together autonomous systems; components for coordinating the process distribution within a system; objects for organizing the code within a process.

A typical system has relatively few cross-system connection points, such as the one connecting the POS and the inventory system. There may be many more distri-

Usage Hierarchy

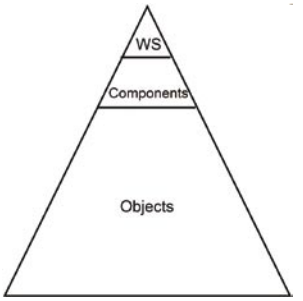

FIG 7

TABLE 1

Comparison of Objects, Components, and Web Service Attributes

	Objects	Components	Web services
Locality	In the same process as its client	In a different process than its client	In a different organization than its client
Environment	In the same environment as its client	In the same environment as its client	In a different environment than its client
Speed	Very fast communications with its client	Slow communications with its client	Very slow communications with its client
Builder relationship	Probably built by same person that built its client	Probably built by the same group that built its client	Probably built by a different company than built its client
Quantity	Tons—every place you look!	Quite a few—at least one for each process	Hardly any—perhaps one per major software system

Fuzzy Boundaries

Objects, Components, and Web Services

bution points within the system as a whole, and there will probably be a great deal of code in any one of those processes.

Thus, you would expect to see a hierarchy of usage that looks like a pyramid, as shown in figure 7—tons of objects, relatively few components, hardly any Web services. This is not a judgment of Web services, just an observation on the number of connection points typically found in autonomous systems.

You can also interpret the pyramid of figure 7 as code layers. At the highest level, the autonomous system (e.g., POS system) is packaged as a Web service. Within the Web service, components are used to implement distribution. Within our components, objects organize the code.

Table 1 compares objects, components, and Web services based on the discussion up to now.

Given the hierarchy shown in figure 7 and the various attributes summarized in table 1, you can start to get a better picture of how objects, components, and Web services might be combined in actual systems. Figure 8 shows the point-of-sale and inventory systems, and one possibility for how the three entities might be related to each other.

Figure 8 shows different functions for objects, components, and Web services that leverage their specific abilities. The Web service receives a work request from an outside organization. The component does some major piece of business functionality. The objects organize the code.

You can further categorize the components inside a Web service by the function they provide. Notice that in figure 8, different components have different colored hats. I have used hat color to indicate one of three major types of functionality the component provides:

- **Point-of-entry components.** These red-hat components are the first inside the Web service to receive the request.
- **Application components.** These silver-hat components actually perform the business functionality of the Web service, such as processing the sale in the POS system.
- **Point-of-exit components.** These green-hat components are responsible for preparing work requests for other Web services.

One nice thing about this categorization of objects, three types of components, and Web services is that it can be codified into a more rigorous set of architectural principles. The principles that I use are those codified into what I call the Software Fortress Model (SFM). (For more information: Sessions, R. 2003. *Software Fortresses: Modeling Enterprise Architectures*. Addison Wesley.)

In the SFM, red-hat (point-of-entry) components are called *guards*. Silver-hat (application) components are called *workers*. Green-hat (point-of-exit) components are called *envoys*. Using the SFM, I would redraw figure 8 as figure 9, which emphasizes the self-contained nature of the Web service as a software fortress and the specialized natures of the various components. Figure 9 is simplified by leaving out the objects, which are seen as implementation details of the components, and ignoring process boundaries, which are assumed to be equivalent to component boundaries.

An advantage of a model such as the SFM is that it defines principles that can be used as the basis for inspecting systems for "correctness." When it comes to software systems, there are many facets of *correctness*, including security, error handling, performance, scalability, and reliability of a system, in addition to the obvious "does it work?"

POS and Inventory System Usages of Objects, Components, and Web Services

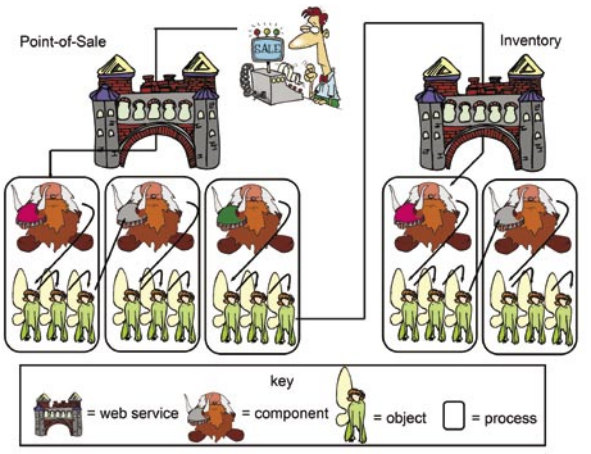

FIG 8

Most people falsely assume that the correctness of a large software system is largely related to the implementation details of the objects, components, and Web services that make up that system. In fact, the correctness is much more related to understanding the differences among these three types of entities.

Let's take security as an example of correctness, focusing on the specific security issue of trust. In other words, do I trust the client that is making this request? Where, in a large software system, should I ask the question, "Do I trust my client?" Do I ask it at the object, the component, or the Web service boundary?

This question is best answered by returning to table 1 and looking at the row labeled "Builder relationship." Here, relationship refers to that between the builder of the entity and the builder of the client.

The intersection of the objects column and the relationship row says, "Probably built by the same person who built its (the object's) client." Do you trust yourself? Of course! So this is not the right boundary at which to deal with trust issues.

The intersection of the components column and the relationship row says, "Probably built by the same group that built its client." Do you trust others in your group? Hopefully, the answer is yes.

The intersection of the Web services column and the relationship row says, "Probably built by a different company than built its client." Do you trust people in another company? Probably not. You have no control over them. You may not even believe that they are in the company they claim they are in. So it is at the Web services boundary that you should focus your trust concerns. Once we have crossed the code boundary that separates a Web service from its client, trust is no longer a relevant issue, not because you no longer care about it, but because you have already dealt with it at the appropriate boundary.

I don't have space here to give a similar treatment to the other correctness issues, but the trust analysis should give you a feeling for the importance of understanding the differences in the boundaries that define objects, components, and Web services and how to go about asking the right questions. When in doubt, the starting point for the discussion is table 1. Table 1, in turn, is driven by understanding how environment and location are pivotal concepts in "de-fuzzifying" the boundaries.

DEFINING BOUNDARIES

The language and tools vendors have done us a great service by making it easy to create components and/or Web services from objects. Unfortunately, they have left us with some very fuzzy boundaries dividing these three very different types of entities.

A starting point for using objects, components, and Web services correctly is understanding the fundamental differences among these three entities and how those differences define correct boundaries. Robert Frost quotes his crotchety neighbor in "Mending Wall" as saying, "Good fences make good neighbors." When it comes to designing enterprise software systems, his neighbor has a point. Q

Representative Software Fortress Model

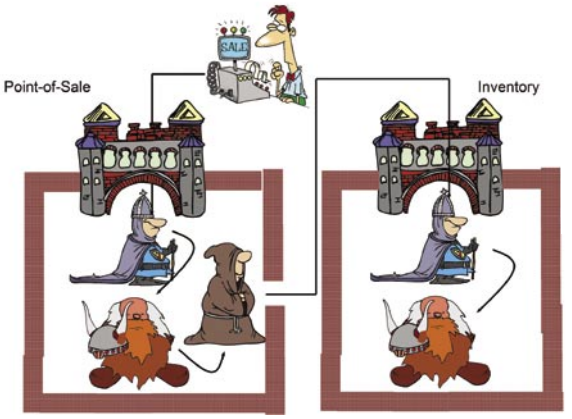

FIG 9

LOVE IT, HATE IT? LET US KNOW

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

ROGER SESSIONS is founder and CEO of ObjectWatch and is the author of six books, including *Software Fortresses: Modeling Enterprise Architectures* (Addison Wesley, 2003) and dozens of articles. He is on the board of directors of the International Association of Software Architects. His monthly *Architect Technology Advisory* helps enterprise architects stay abreast of new technology trends and how those trends will impact their software systems. His quarterly *ObjectWatch Newsletter* regularly prods and examines the software industry. Both publications are available at www.objectwatch.com. His writing has earned him a "Most Valuable Professional" designation from Microsoft.