Pro Scalable .NET 2.0 Application Designs

Joachim Rossberg
Rickard Redler

Apress®

Pro Scalable .NET 2.0 Application Designs

ISBN: 1-59059-541-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Jason Lefebvre
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beckie Stones
Copy Edit Manager: Nicole LeClerc
Copy Editor: Julie M. Smith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Lori Bring
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Linda Seifert
Indexer: Broccoli Information Management
Interior Designer: Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 7

Service Oriented Architecture (SOA)

Over the years, new technologies and architectures have popped up in the IT world. Some of them have been heavily promoted by only a few advocates, while others have become widely accepted. Quite a few of them have been hyped by developers and, to some extent, decision-makers alike, but these solutions still haven't managed to get a real breakthrough and become used in the way they were intended. Some argue that this has happened to Component-based Development (CBD), as you will see shortly, and that this has paved the way for Service Oriented Architecture (SOA).

Overview

We have seen an evolution: from procedural programming to functional development to object orientation to component-based development. These techniques have all had their benefits, and object orientation, for example, is probably something that won't ever be unnecessary, no matter which architecture you design your applications with.

It's been a long winding road to reach the latest stop on the architecture tour—Service Oriented Architecture. However, SOA seems to have been one of the architectures that have been widely accepted. How can this have happened so soon? Well, Component-based Development, or CBD, promised all kinds of benefits, such as reuse and an open market in components. These benefits would (purportedly) drastically reduce the time it takes to develop new applications and systems, according to Lawrence Wilkes and Richard Veryard, in their paper "Service-Oriented Architecture: Considerations for Agile Systems." Many companies used, and still use, techniques like CORBA or COM, but they never got the breakthrough and the results that were promised to them. You may wonder why this has happened and we wouldn't be surprised. But if we look back, we really can't say that the component market has blossomed as it was promised or that companies reuse components to a great extent.

According to Wilkes and Veryard, many companies saw other benefits instead. Some of these were improved scalability and the possibility to replace components as needed for instance.

But, we think that the reason that companies have failed to reuse components lies more with us: the developers and architects. How much effort did we really put into reusing components? Think about it for awhile, how many times have you reused components compared to the number of times you have reused code snippets? We can honestly say that for us, code reuse has been greater than components reuse. It's been easier to build a new component reusing code snippets than it has been to adapt old components to new requirements. We do make sure that we build systems using components and object orientation, but we don't build them after the principles of CBD.

In 2003, Don Box, speaking at the XML Web Services One conference, likened objects to software-based integrated circuits and said that programmers would do better to focus on services instead (Vincent Shek, *Microsoft Architect Touts SOAs*, `http://www.eweek.com/article2/0,1759,1655790,00.asp`). We agree with Don Box, and we tried to push for this in the first edition of our book.

When web services emerged a few years ago, new ideas for architecture came to the surface. Web services provided us with benefits in several areas:

- *Platform independence*. We can use web services built on any platform, as long as they use standard protocols for communication (that is SOAP, XML, and WSDL). We really don't care how they've been implemented, so long as we know how to contact them and what their interface is.
- *Loose coupling*. Components hold a connection: clients using web services simply make a call.
- *Discovery*. We can look up a specific service using a directory service like UDDI.

These features of web services fit nicely into the more and more connected world of today. But, if we're going to build systems in large enterprises in the future, we need to change the way we look at architecture. This is where the Service Orientation (SO) comes into Service Oriented Architecture (SOA).

What Is a Service?

What is a service? Many people have a hard time understanding what to think when they are discussing services. At one of my recent .NET courses, the students and I were involved in a long process until a revelation occurred among the participants. At first, they couldn't stop thinking in a traditional, component-based way, but after several discussions, the coin finally dropped down. After that first barrier was removed, all our architecture discussions went very smoothly, and suddenly there wasn't any problem at all. The students now had lots of ideas and thoughts about how to implement services in their own projects.

Let's start with a real life example. I recently applied for a renewal of my driving license from the Swedish Road Administration (SRA). The SRA's whole procedure for handling this can be said to be a service. The service, in this case, being to process my application and then, based on various (and for me, hidden) facts and procedures, they either issue me a new license or reject my application. I don't need to bother about how the SRA processed my information or which routines they followed internally to process my application. In this case, my application is a request to a service which SRA expose and their answer (whether a new license or a rejection) is the response to my request.

So, a service doesn't have to be a mystery or something hard to understand. Think of it as something that encapsulates a business process or an information area in your business. A common scenario could be the management of customer information, something that most companies have to deal with. The service CustomerManagment can expose extensive functionality, based on the input parameters. For instance we can find functions that let us add new customers, update customer information, delete customers, return customer information and lots of other things concerning customer management. One of the keys is that the user of the service, the consumer, does not have to know what goes on behind the hood. All the processing is hidden from the consumer, who should only need to know how to call the service.

One common problem is confusion as to where to draw the line between the depth of our services during the design phase. We need to make sure that the service we are about to develop is valid and does not cover too much or to little. We will get back to this a little further down.

What Is This Thing Called SOA?

If we have built some cool web service that our company uses does this mean we have implemented an SOA? Definitely not, we say. Web services are only parts of a service-oriented architecture. What web services really are, to be honest, is just a set of protocols by which services can be published, according to Sprott and Wilkes, at `http://www.cbdi.com`. Web services are programmatic interfaces to capabilities which conform to the WSnn specifications (WSnn is just an acronym for all the parts of web services available to us like SOAP, WS-Security, WS-Federation, and so on).

But who are Sprott and Wilkes? Why should you listen to them? These two guys are perhaps not as well known as Don Box, but they have been in this business a long time, and they are considered to be experts. They both work for CBDI Forum, which is an independent analyst firm and thinktank, and David Sprott is one of its founders. They both give plenty of lectures and write numerous articles on various aspects of business software creation, including, to a large extent, web services and SOA.

But web services aren't a must in SOA either. We can expose our services by other means as well (using message queues, among other things), although web services will probably be the way we implement SOA in the foreseeable future.

If you look around on the Internet, you can find different definitions for SOA. The World Wide Web Consortium (W3C) defines SOA as "a set of components which can be invoked, and whose interface descriptions can be published and discovered." This sounds like web services doesn't it? This view might perhaps be a tad too technical and probably not entirely correct either.

Sprott and Wilkes of CBDi Forum (`http://www.cbdiforum.com`) define SOA as "the policies, practices, and frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a simple, standards-based form of interface." As we can see, this definition is both wider and more correct, if you ask us.

Why Use SOA?

There are several implications to consider when implementing an SOA in an organization. SOA is much more than simply developing a few web services and thinking you're finished. If you're going to successfully make the move to service orientation, you'll need to have the whole company management on the wagon as well. So how do we do that?

Let's start with considering EAI or Enterprise Application Integration. This topic was discussed earlier, in Chapter 1. One major problem that many large organizations struggle with is the integration of their existing applications into their new infrastructure. The problem is that many times they only succeed in exposing functionality in a proprietary way, shutting out everybody who doesn't speak this language.

The solution, in many cases, has been developing interfaces sitting on top of these systems. This way, a lot of legacy applications have had their functionality exposed to new applications in a convenient and standards-based way. The problem with this is that we have to develop the interfaces that enable this integration, and this obviously costs a lot of money. Imagine if the architects implementing the legacy applications had a way of making sure that the applications could easily be accessible by the new systems without new development being necessary!

With SOA, this isn't just a dream anymore. Since the whole idea is to expose services through a standards-based interface, not proprietary to anyone, we can make sure that their functionality can be accessed by new systems easily. This way we don't have to build special interfaces to give access to their functionality.

We think this might be the strongest argument of all. By using TCP, XML, SOAP, web services, and any other standards-based technology, today we can build systems that are still accessible to all new applications fulfilling these standards in the future. And, since they are standards, we have a whole lot more certainty that they will be around in 10 or 20 years time.

This will save companies a lot of future development costs. And let's face it. It won't have to be 10 or 20 years before we can start calculating savings from this approach. Enterprises have new systems coming out all the time, and using SOA we make sure that we can reuse functionality in a far better way than we could with component-based development.

Remember though, it's no walk in the park to implement SOA in an organization. It will take a whole lot of consideration before it can be rolled out properly. SOA is not intended for single applications either, but for organizations as a whole. This means that if we're going to be successful we need to incorporate SOA in the enterprise IT plan from day one. If SOA isn't considered for all new development or all the future enhancements for the existing systems, we will fail in implementing service orientation.

But once we start letting SOA influence all IT issues in our organization, we make certain that the ROI (Return On Investment) will be great.

Services and SOA

Now we'll start looking at the characteristics of services and examine what some very influential people think about them. We'll then take a look at how architecture in SOA should be considered. Let's start with some more of Don Box's thoughts and take it from there.

The Four Tenets of Don Box

To better understand what a service is and why we must think of architecture in a slightly different way than we did with object-oriented programming and components, we'll now look at the four important tenets that Don Box, of Microsoft's Windows Communication Foundation team, identifies as crucial. For the complete coverage of this, please see "A Guide to Developing and Running Connected Systems with Windows Communication Foundation" by Don Box. This document is available on the Microsoft web site and is well worth its time.

Don Box's first tenet is that *boundaries are explicit*. When services that build up an application or system are spread across large geographical areas, things get more complex. We might run into large geographical distances, multiple trust authorities, and perhaps, different execution environments. Communication in such an environment, whether between applications or even between developers, gets potentially costly, as complexity is great and performance may suffer. An SO solution to this is to keep the surface area as small as possible, by reducing the number and complexity of abstractions that must be shared across service boundaries. Service Orientation, according to Box, is based on a model of explicit message-passing, rather than implicit method invocation, as it is in component-based development.

Don Box then states that *services are autonomous*. Services must always assume that nobody is in control of the entire system. Having total control of a system would be impossible in a service-oriented architecture. Components or object-oriented programs are often deployed as a unit, even though many efforts have been made to change this over the last decade. In SO this isn't always the case, if it ever can be. Service-oriented systems are often deployed across a large area instead. The individual services, are often deployed as a unit, but the whole system is dispersed. A SO system never actually stands still, because new services may be added all the time and old services given new interfaces. This increased ubiquity of service interaction makes it necessary for us to reduce the complexity of this interaction.

When a service calls another service it can't expect a return value because messages are only passed one way. We can get a return message of course but this can disappear on the way and we can never be certain it arrives in the end. This means that the service and the consumer are independent of each other. They are autonomous, or sovereign as Bruce Johnson puts it on Bruce Johnson's SOA(P) Box (`http://objectsharp.com/Blogs/bruce/`). So in a scenario where a service calls another service and gets no expected return message, the service must be self healing so that there isn't a noticeable impact. The caller must not have to depend on anybody else to complete the invoked method.

In a connected world, many things can go wrong with communication, and our services must take this into account. When a service accepts an incoming message that is malformed, or even sent for a malicious purpose, it must be able to deal with it. The way to do this is to require the caller to prove that he has all the rights and privileges necessary to get the job done. This burden is never placed on the service itself. One way to solve this is to set up trust relationships instead of per-service authentication mechanisms.

The third tenet is one of the most important ones: *services share schema and contract, not class.* In OO programming, abstractions are developed in the form of classes. They are very convenient, as they share both structure and behavior in a single named unit. This is not the case in SO, as services interact only based on schemas (for structure) and contract (for behavior). Every service we build must advertise a contract that describes the messages it can send or receive. Even though this simplifies things we must still be aware that these schemas and contracts remain stable over time. We can't change these when they are implemented. To increase future flexibility, we can use XML element wild cards and/or optional SOAP header blocks, so that we don't break already deployed code. We might even have to build a new service function instead, and make that available.

The last tenet that Don Box mentions is that *service compatibility is determined based on policy*. Every service must advertise its capabilities and requirements in the form of machine-readable policies, or operational requirements, in other words. One example is a service that requires the caller to have a valid account with the service provider. Another could be a situation where all communications with the service should be encrypted.

These four tenets seem pretty straightforward to us and we strongly agree with them. But, let's look at how others characterize SOA.

Sprott and Wilkes SOA Characteristics

Sprott and Wilkes also show some principles of good service design enabled by service-oriented architecture characteristics.

- *Abstracted*. A service is abstracted from its implementation. This means a consumer of a service shouldn't have to worry about how the service is implemented, only how to invoke it.
- *Published*. The implementation should be hidden by a precise, published specification of the service functionality. This is something we recognize from Don Box's tenets, when he talks about policies.
- *Formal*. There must be a formal contract between the provider and the consumer, which is also something Box stresses.
- *Relevant*. Functionality should be presented at a level of granularity that the recognized by the user as a meaningful service.
- *Reusable*. We should reuse services, not copy and paste code.

Web services enable good service design because they are

- *Technology neutral*. We do not care about their platform as long as they are:
- *Standardized*. This means that they should use standards-based protocols (SOAP, XML, and WSDL).
- *Consumable*. Web services enable automated discovery and usage.

There is nothing here that contradicts what Don Box says and we agree with what Sprott and Wilkes have come up with.

Sundblad and Sundblad

Let's now take a look at what Sundblad and Sundblad say in Sten Sundblad's Swedish document "Service Oriented Architecture—An Overview," or as its called in Swedish "Serviceorienterad arkitektur—En översikt."

Sundblad starts by pointing out that the idea of SOA is to organize IT systems as a set of services that can be reached through a message-based interface. It builds on the same foundation as SOAP and XML, but is not dependent on either. The idea itself is nothing new—it's been around for over ten years and was started in the CORBA community in the 1990s. But now, thanks to XML, SOAP, and web services, the idea has been getting the attention it deserves. Sunblad continues that the reason for this is obviously because these building blocks have made it possible for a service to be both language- and platform-independent.

Sundblad also states five characteristics that are usually mentioned when talking about services. We'll now take a closer look at each one of these.

- A service should be autonomous.
- A service should have a message-based interface.
- A service should have its own logic that encapsulates and protects its own data.
- A service should be clearly separated from and loosely coupled to the surrounding world.
- A service should share schema—not class.

Autonomy

You'll recognize this characteristic from Don Box's four tenets. Autonomy means that the service should be possible to develop, maintain, and evolve independently of its consumers. The service must take full responsibility for its own security and be suspicious of anything and anybody who tries to use it. Don Box also stresses that these features are important for a service to possess.

Message-Based Interface

No consumer should ever be in contact with the inner logic of a service. You must develop the services so that they are only accessible when you send a message to them with a wish to get the service performed. It's the logic of the service that handles the message and determines whether to process the request or not.

Let's take a little example from real life. When I recently applied for a renewal of my driving license I got a form sent to me via snail mail. This form was preprinted with text, leaving spaces for me to insert information about myself and attach a photo showing my best side. I also had to sign it, so that my signature could be printed on the license. The form's instructions explained how I should fill it out and how to return it to the authorities. After a few weeks, I would either get a new license or an answer as to why my application would not be approved. (And yes, I was approved.) Nowhere in this process did I have to know how the authorities handled my application or how they would process my information. The only thing I needed to care about was filling my form out correctly and making sure I sent it back to the correct address.

If we translate this experience into our examination of services, we can say that the form I filled out was a message that was being sent to the service. I also got a message in return, in this case either a new license or a rejection. But these messages are totally asynchronous. They don't depend on each other in any way, except for the fact that my application must come in before the agency's reply. But otherwise, there is no connection between me and the authorities, during the waiting period. So, this example shows not only a message-based interface, but also the autonomy of this service. Keep this example in mind, because we will return to it in a little while.

Even though we often talk about web services when SOA is discussed, Sundblad firmly points out that this does not have to be the case. There are other ways to display a message-based service, including by using a message queue. But, to be realistic, we could probably assume that most service-oriented solutions will use web services.

One thing worth noticing here is that using a message-based interface means that we should send all parameters encapsulated in the message. This goes for both the consumer and the service. We don't build service, for instance, that take integers as parameters—instead we embed the integers in an XML document and send this to the service. In return, we'll get an

XML document with the result of our request. A benefit of this procedure is that it's easy to add new parameters for future needs without changing the signature. The only thing that has changed is that the content of the message and both old and new messages will now work.

Encapsulation of Data

One of the important aspects of a service is that it should be autonomous, as we have learned from all of the previously mentioned authors. According to Sundblad, this also means that a service should encapsulate its data. No other service or application should be allowed to touch the data (see Figure 7-1).

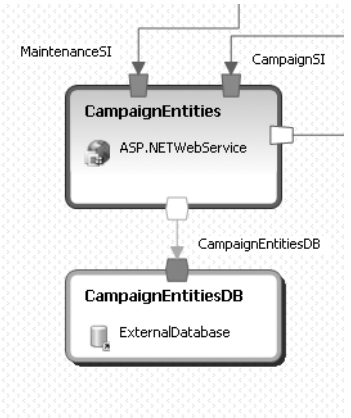

Figure 7-1. *A service should encapsulate its own data*

The only way that your service's data should be reached is when a user sends a message requesting that some modification of the data is to be performed. If the consumer is validated and has the correct credentials, and has sent a well-formed message according to the service specifications, its request will be performed. It's the logic of the service that it touches the data, never the message itself.

■ **Note** A service's data can be both persistent and non-persistent. The objects in a service often have non-persistent data in their RAM, just like any other object, but instead of relying on another component to save persistent data, in this case the service itself saves this to the database.

Clearly Separated From and Loosely Coupled to the World

As you may already understand, a service must be clearly separated from its surrounding world—just like the driving license renewal process, where the service (renewal) was separated from the message sender. The driving license example also showed that the process of determining whether or not the application for a new license was approved (or not) was loosely coupled to

the renewer. The same goes for services in service orientation. The renewer, in the case of the driving license, sent a message to the authorities, waited, and finally got the reply. No persistent connection between the two existed at any time. When a consumer sends a message to a service, no persistent connection should exist either. If a new message needs to be sent, a new non-persistent connection has to be made. The only thing that consumer and the service share is the structure of the message and the contract that controls the communication between the two.

Share Schema—Not Class

For SOA to work, the consumer and the service have to agree on the schema that determines the structure of the message.

Since the object model of consumer and service might be completely different, the service doesn't need to, (and shouldn't) share its object model. It should only share the schema of the message structure. In our driving license example, the schema would be the form and its instructions. When we are talking about web services, the schema would be the WSDL file.

Sundblad continues that when you first start designing your services you should always start by defining the message-based interface that your service will expose. After you do this, it's time to think about how to design the logic. This is so that you can create all of the XML schemas and contracts before you even think about the objects that implement the interfaces.

Four Types of Services

What most people agree upon is that the concept of services encapsulates a whole business process or business information area. All businesses have somewhere between 10 and 20 higher level services according to Sten Sundblad in the Swedish paper "Service Oriented Architecture and processes" (unfortunately only available in Swedish). Higher level services could include the Purchasing process or the Sales process, for example.

Sundblad pushes for doing a process-oriented analysis of the business in question, when you're implementing your SOA. As you do this, you'll find that there are at least four kinds of services that should implement the functional requirements as follows:

- Entity Services
- Activity Services
- Process Services
- Presentation Services

The *entity services* are services that are rather persistent or stable over time. A business is not likely to change these processes, because they supply and protect the information that the business is depending on. If you think of these different kinds of services as four layers, they would be the foundation at the bottom level of Figure 7-2.

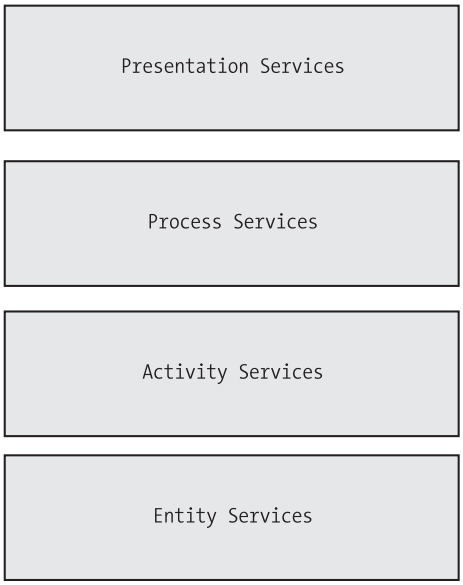

Figure 7-2. *Four kinds of services in a business*

Activity services are slightly less stable than entity services. In other words, they may be changed over time. Activity services are placed one layer up from entity services, and are the services that perform the actual work in one or more processes.

High-level *process services* are in the next layer. It's very probable that these kinds of services will change over time, since they often reflect the changes in a company's way of doing business and are often trimmed away to give competitive benefits when the company needs them. When we change these no lower level services should have to change.

Presentation services comprise the top layer. These are the user interface services and they provide users with a way or of accessing other services. As you probably recognize from your own company, these kinds of services are very likely to be changed as times goes by.

Architecture of a Service

Now you know what services are and what they do. You also have a pretty good idea about how they interact with the outside world. As we mentioned, though SOA is a way of structuring a whole business IT support, it's not intended for single applications. So, if a company wants to implement SOA, they'll first have to get a grip of the total IT structure within the company.

How should you structure your services on the inside? Well, when we design services we often use Sundblad & Sundblad's reference architectures.

■ **Note** We can also choose to design individual services using standard Object Orientation techniques. There is nothing contradictory in this and this will probably be a common way of developing services.

We will give the following two examples of these reference architectures here:

- Entity Services Architecture
- Presentation Services Architecture

Entity Services

Figure 7-3 shows an overview of the architecture of an entity service, according to Sundblad & Sundblad.

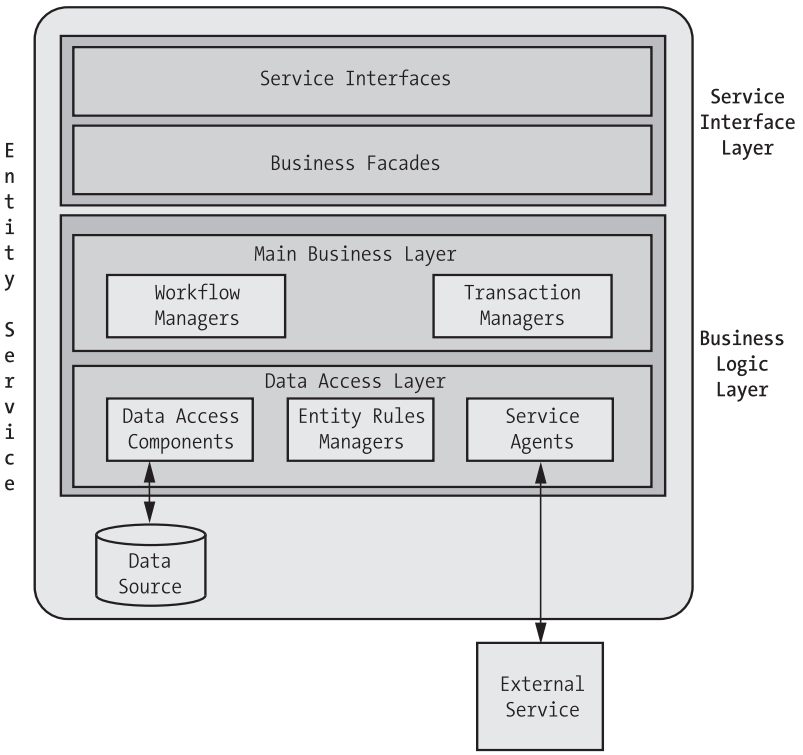

Figure 7-3. *Sundblad & Sundblad's reference architecture of entity services*

Compare this to the architecture of a component-based application, as shown in Figure 7-4, and you see several similarities.

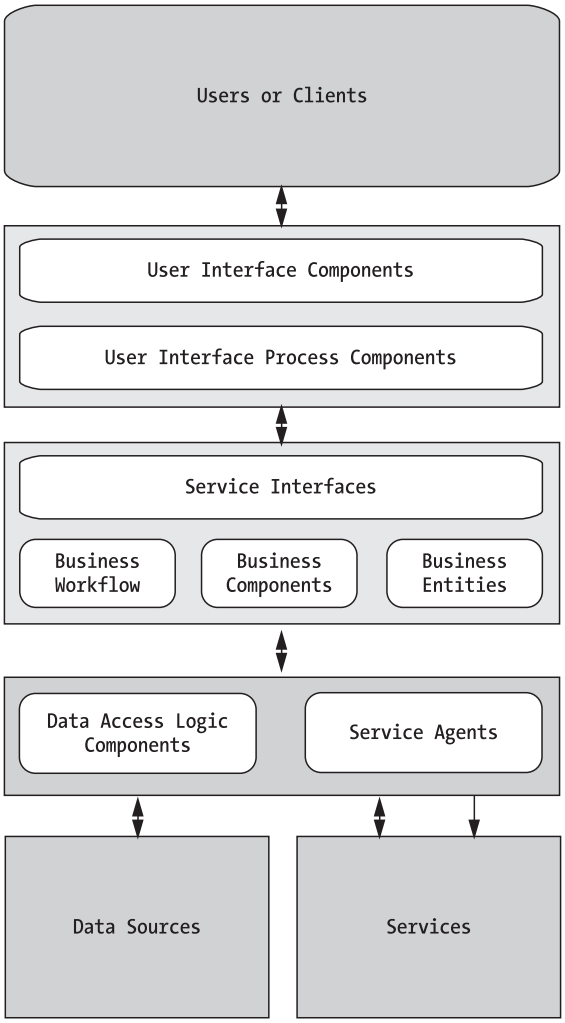

Figure 7-4. *The standard architecture of a component-based application*

We find that this architecture is layered and can be executed in one process, and, since it's layered, that we have the opportunity to scale the service on several servers or server farms. In the business interface layer we find the Service Interfaces (SI) that our service will expose. They will be implemented as web services, which will communicate with the business facades, which in their turn will handle the communication with the business logic.

Figure 7-3 also clearly shows that the data source is kept within the boundaries of the service, so that no one other than the developer can access the data except by sending a message request to the service. It's only the data access layer that handles communication with the database. Service agents manage communication with external services, if this is needed.

Presentation Services

Figure 7-5 shows the reference architecture for presentation services. These are interfaces for users, which are built as Windows or web forms. But, in a true SOA, they are also services with clear boundaries and autonomy.

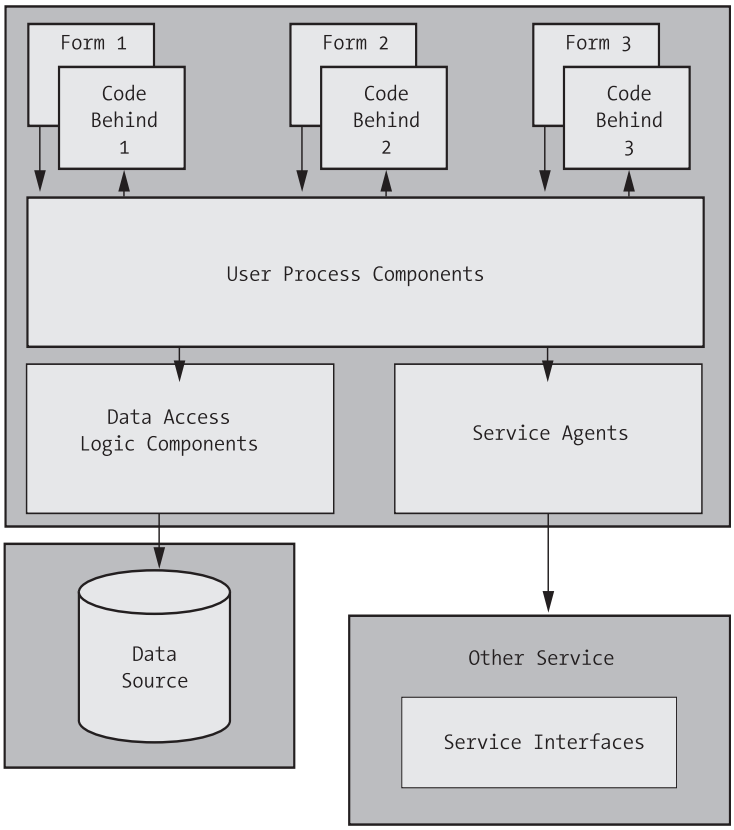

Figure 7-5. *Sundblad & Sundblad's reference architecture for presentation services*

The user process components do the same as user interface components which, as you may recall, is to avoid redundancy in our forms.

You can also see that you should have service agents handling your communication with external services, such as process services or entity services. This communication is based on SOAP, XML, and WSDL just like any other service would be. In the Figure 7-5 you can also see that a presentation service has a data source of its own, where it stores session data or cached data, such as product catalogs, or any other data that isn't likely to change often. The contents of a customer's shopping cart before checkout would be excellent information to store here.

Transactions in SOA

You learned earlier that services should be self-healing. Nothing but the service itself should be responsible for making sure its requests are carried out. In order for a service to keep an atomic transaction going that spans over several services, we need to make sure it's certain that the whole transaction is performed. In Figure 7-6 you can see an example of a service sending update requests to two other services in one transaction.

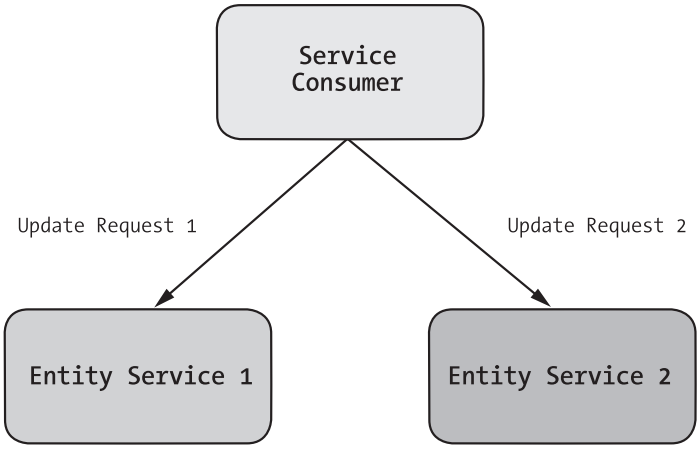

Figure 7-6. *An atomic transaction spanning two services*

If both of these service calls should fail (see Figure 7-7), nothing really bad will happen. This is because nothing has been changed in any of the participating service's data sources during the transaction, so no harm is done.

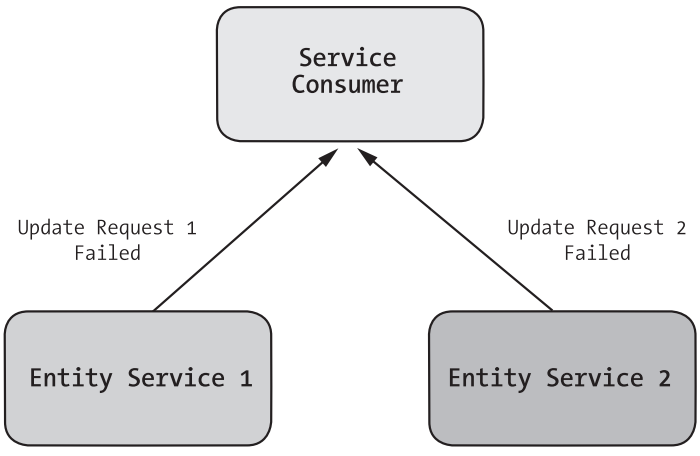

Figure 7-7. *If both requests fail no harm is done*

But if one request fails and the other succeeds, things can be much worse. In Figure 7-8, you see that the update request to Entity Service 1 fails while the other request is OK. Now you have a problem, because the state of the services is out of balance.

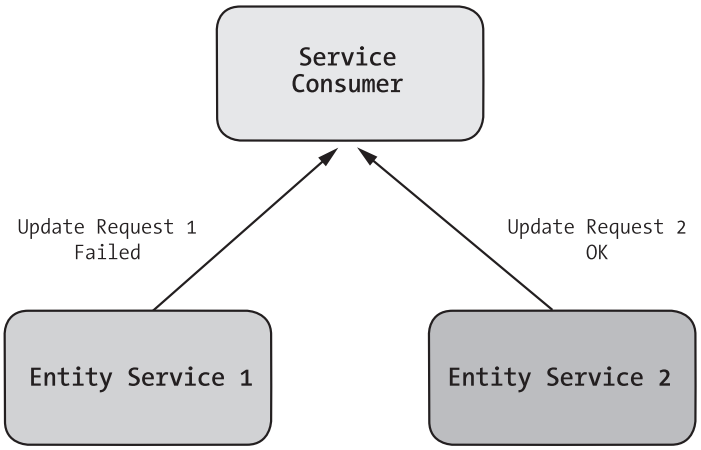

Figure 7-8. *One request fails*

How can you solve this? You can do this by building logic into your services that send a compensating request (as in Figure 7-9) to the service, whenever the update was OK. In other words, you must write logic of your own that will perform a rollback when you have services that need to use transactions. Then your services must try the transaction again (since it needs to get it done) until it is successful.

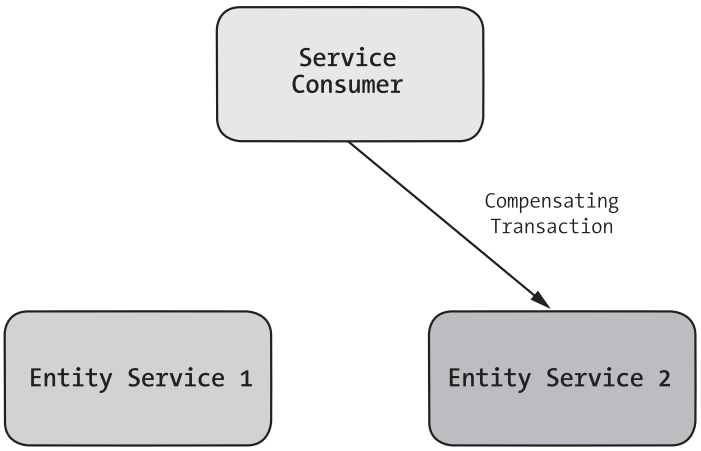

Figure 7-9. *Sending a compensating request after the failure of an atomic transaction*

So, we've just given you an example of how a service must always be prepared for the unexpected and be able to handle situations like this. In other words, a service needs to be self-healing, as we mentioned earlier.

Note that the responsibility always is placed on the service that is keeping the transaction together and not on the services that are part of the actual transaction.

Summary So Far

So, what have you learned so far? It seems like there is an agreement as to what the most important parts of services and SOA are among various industry people. You now know that there are several characteristics a service must have:

- Be autonomous.
- Share schema and contract—not class.
- A message-based interface.
- Provide its own security.
- Clear boundaries.
- Be loosely coupled to the world.
- Encapsulate its own data.

There are also at least four types of services that we can use:

- Entity Services
- Activity Services
- Process Services
- Presentation Services

You have also learned that while web services are great for accomplishing an SOA, they are *not* mandatory. You're also aware that SOA is not intended for single applications either, but rather for a whole business IT infrastructure.

Component-Based Application Design vs. Service Design

Now we'll give you an example of how to implement Service Orientation. In Chapter 1 we showed you an example of using UML, and we'll use that example as a platform for this one. First, you'll see how to implement this as an ordinary component-based application and then you'll see how it could look as services instead. Please keep in mind that all aspects of the application have been overly simplified, just for the sake of more easily illustrating our points. As they say about stunts in the movies; don't try this at home.

In the Chapter 1 example, you might recall that we used several actors, including Sales_Clerk, Supply_System, Sales_Manager, and Customer. We'll only consider Sales_Clerk and Customer for this example.

We found that a Sales_Clerk had a use case called Maintain Campaign and that Customer had one called Request Campaign Email, as shown in Figure 7-10.

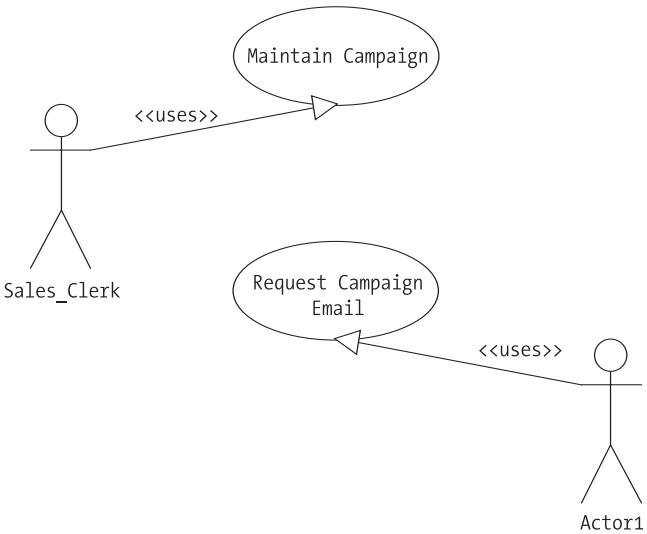

Figure 7-10. *The use cases from Chapter 1*

For this example, we will develop one User Interface (UI) for each of the two actors that we're working with, as you can see in Figure 7-11.

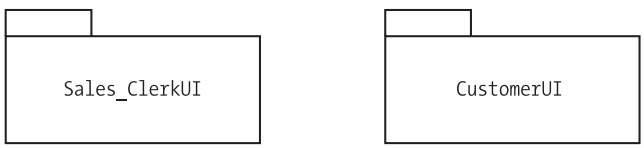

Figure 7-11. *The User Interfaces for our actors*

These UIs will be implemented as one web application that encompasses both the customer and the sales clerk. In almost all applications, you'll find that several functions are similar or even the same for many user interfaces. Therefore we'll avoid such redundancy in our example by implementing these functions in a User Interface Process (UIP), as we see in Figure 7-12.

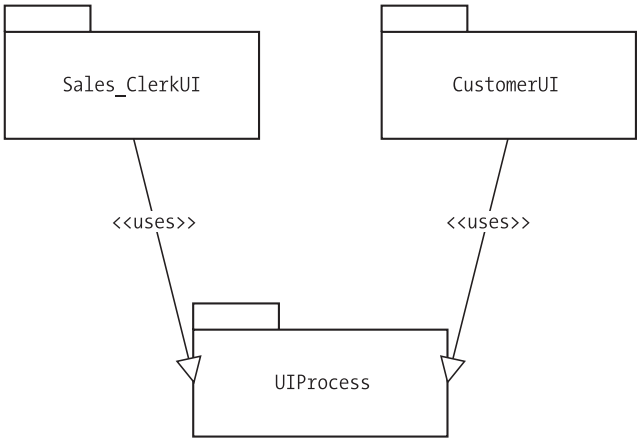

Figure 7-12. *User Interfaces and User Interface Process*

We'll connect all the user interface objects with the business logic. This way we isolate the UI objects from all contact with business objects (see Figure 7-13).

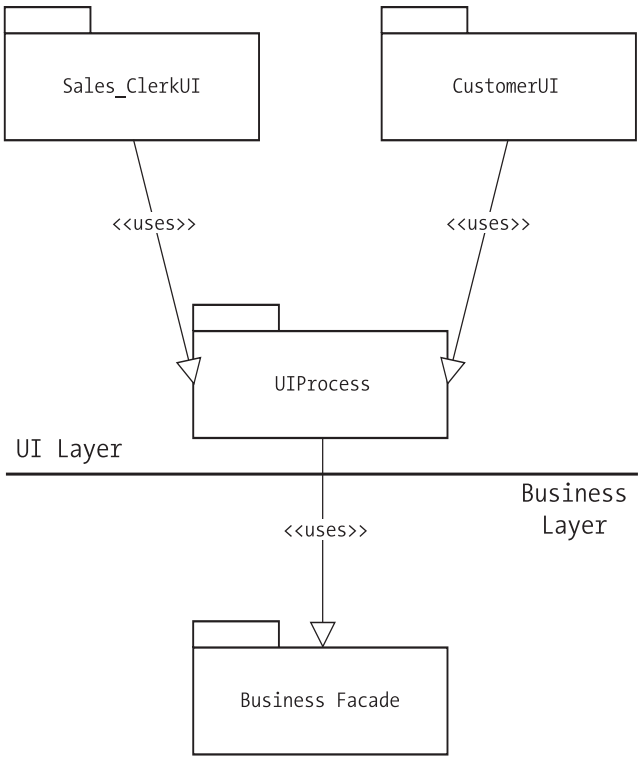

Figure 7-13. *The UI processes are isolated from the business objects.*

As you know already, the reason for having facade objects is that they delegate work to the real logic, implemented as business objects. Inside the business logic, there are three kinds of business objects that you can delegate to (see Figure 7-14).

- *Business Rules Managers*. Evaluates new and changed information against a predefined set of rules.
- *Workflow Managers*. Handles workflows.
- *Transaction Managers*. Responsible for keeping transactions and transaction integrity together.

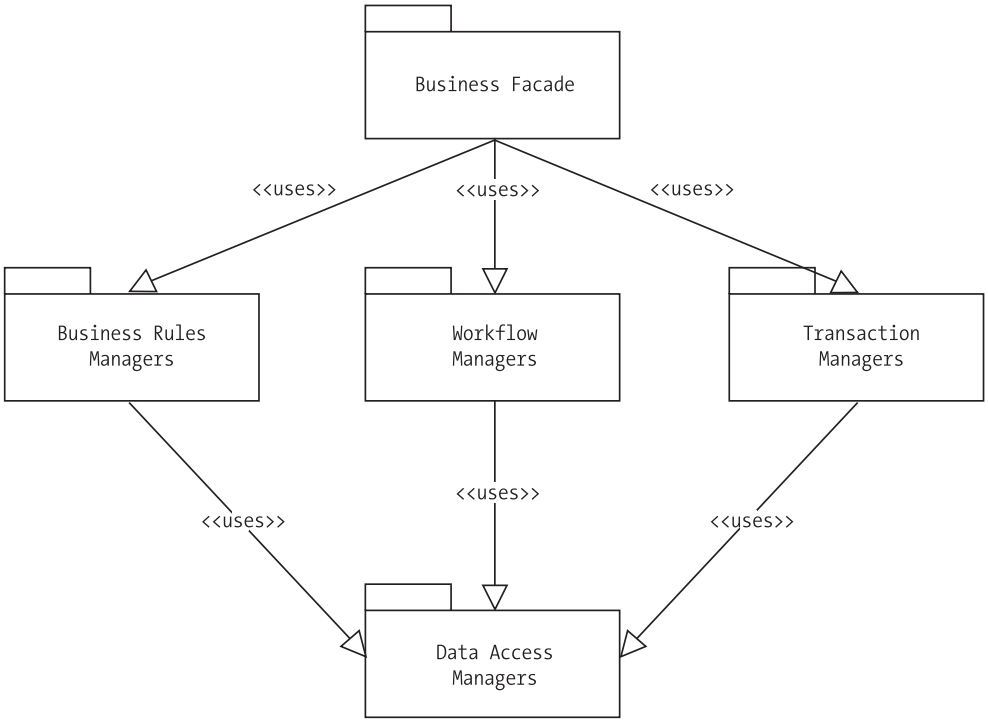

Figure 7-14. *The business objects*

The data access managers will then help our application to access its data.

So now we have the structure of our application. When we deploy it, it will be very much like in Figure 7-15, and we can see that it probably will be implemented on three servers or server farms. These include the following:

- Web farm
- Application server (farm)
- Database server (not shown in Figure 7-15)

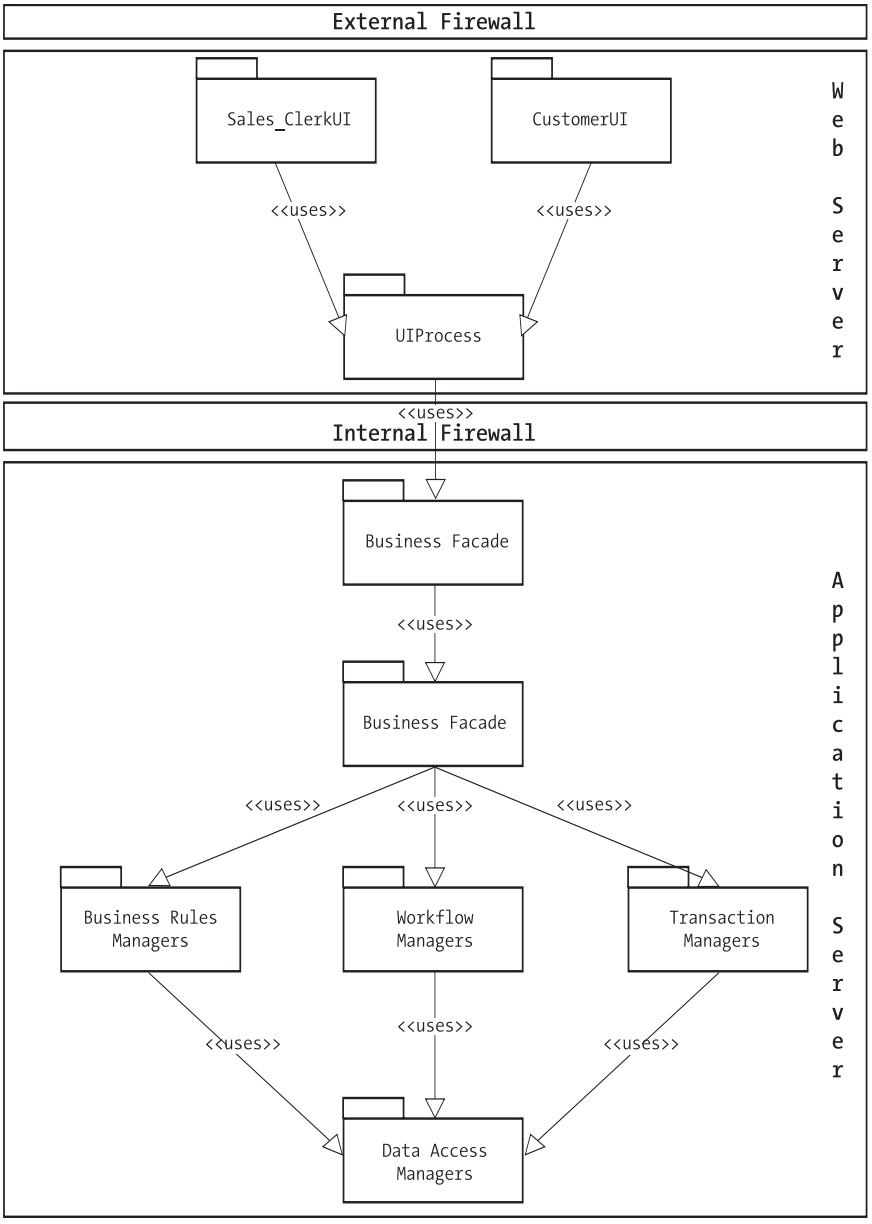

Figure 7-15. *The physical implementation*

This is pretty much a standard way of developing applications and you'll notice that we also employed our recommended way of doing things, as outlined in previous chapters.

So, let's think about this for a while. What are the potential problems with this architecture? Well, a major one you should consider is that of future development. Let's say we need a new application in our enterprise, which also needs to access the same data as the one just described (see Figure 7-16). What happens then?

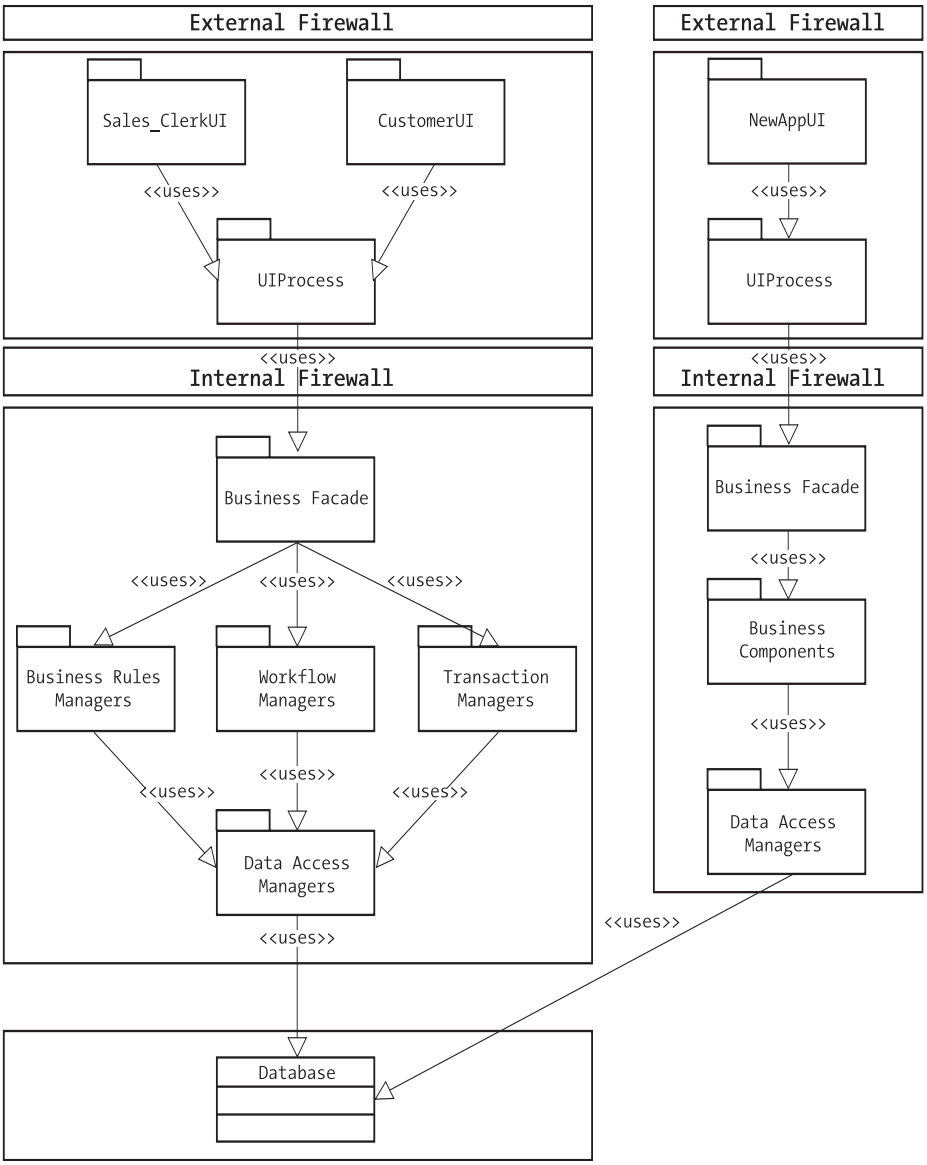

Figure 7-16. *A new application needs to use data from an existing application*

One thing that certainly will happen is that the database will have to change to reflect the needs of the new application. And as Sundblad writes, the more you let additional applications extend the original requirements, the more complex your maintenance and new development will be.

Sundblad also stresses the fact that the original rules will become dispersed. If both applications need to access the same data, both applications will also have to implement the rules for access control and for deciding which data is valid. This will complicate things even more.

One possible solution to this problem is to reuse the business components of the original application by letting the business layer of the new application connect to the business facade of the old one. We could also create a new, separate, database for the new application (see Figure 7-17).

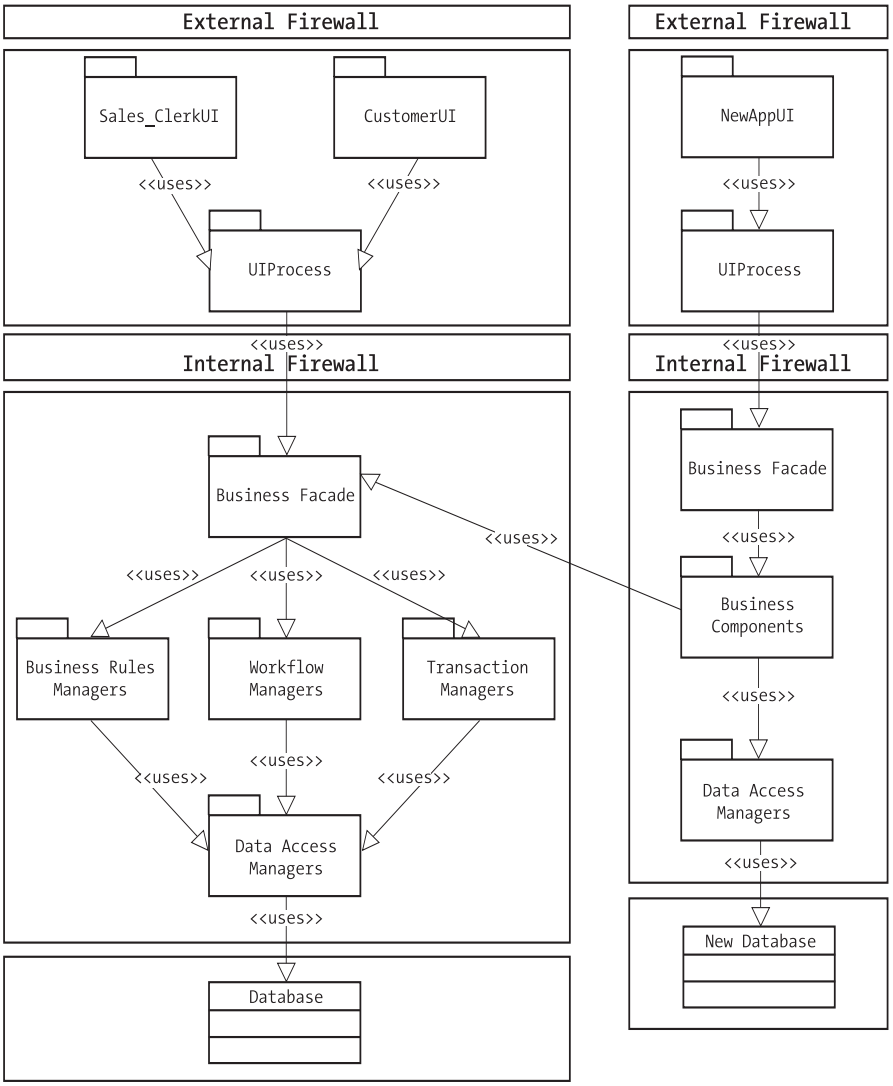

Figure 7-17. *Letting a new application use the business objects of an existing application*

This would work fine for just one additional application, but what if several others are added? Then the situation would soon be unbearable. That's why services are such a great idea. If we had built our application with SO in mind from the start, we'd have autonomous services ready to be used by anybody we authorized. Figure 7-18 shows what such a scenario could look like.

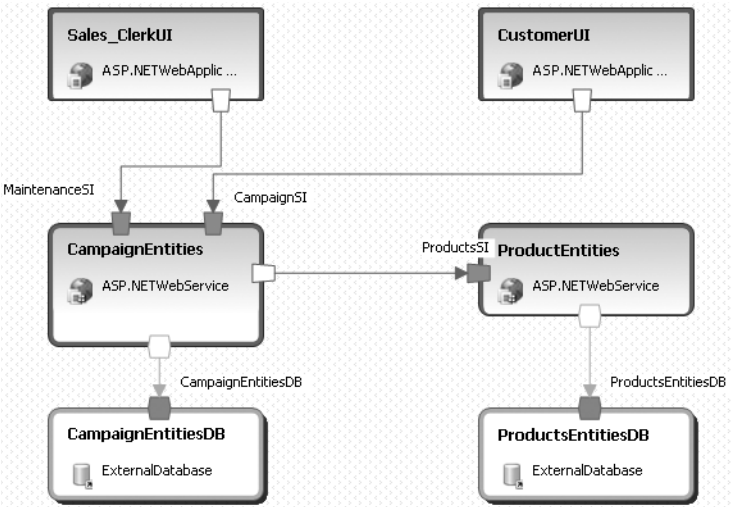

Figure 7-18. *An SO design*

In this scenario, we've built our UI as presentation services and the business logic as entity services, using Sundblad & Sundblad's recommended architectures. We can also see that each entity service is responsible for its own data. Nobody else touches it without sending a message to the service (see Figure 7-19).

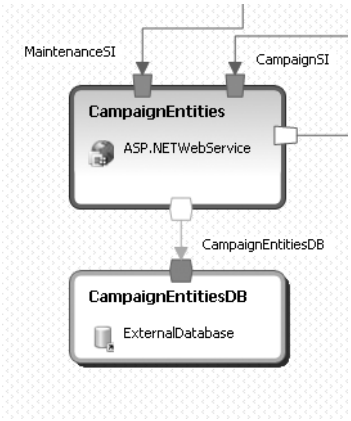

Figure 7-19. *The entity services protect their own data*

If we wanted to add a new application (or service as it would be called here), there's an easier and more flexible way we can do this. In this technique, a new service quite simply connects to the Service Interfaces (SI) of the existing service that it needs to use functions from, as shown in Figure 7-20. We've have created a new service that uses the CampaignEntities service.

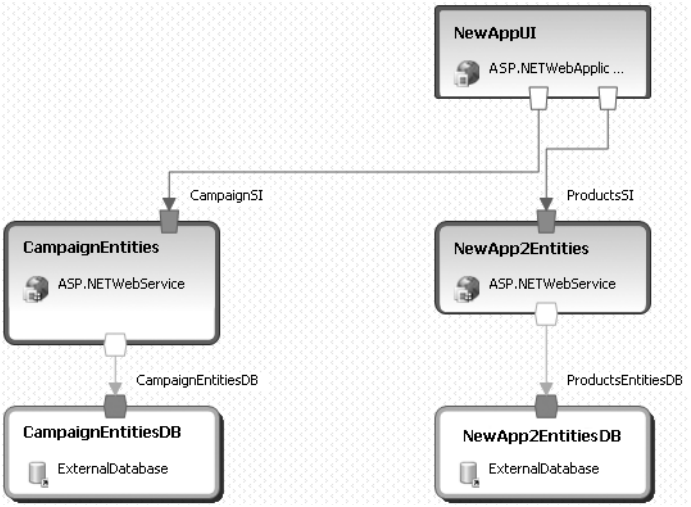

Figure 7-20. *Adding new services to our SOA*

There is no need for splitting the rules across several applications here, because all the services are responsible for their own data and security. This gives us the benefit of being able to add any new services we want in the future, and in a much easier way.

You can see our deployment of this scenario in Figure 7-21. Here, the UI services would be in the DMZ, while the rest of the services are implemented behind our Intranet firewall.

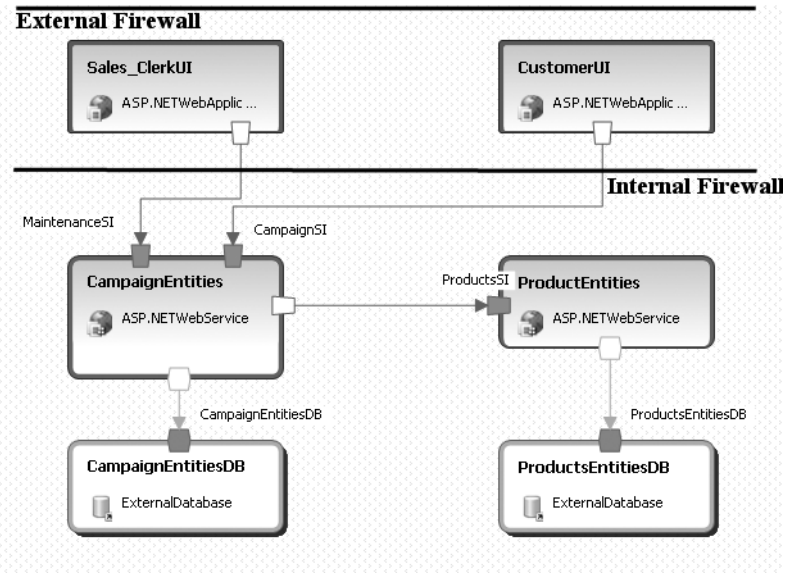

Figure 7-21. *Deploying services*

Scalability Issues in an SOA

When systems are built up of independent and autonomous services, you no longer have control over all parts of the system. This also means that you no longer have control of performance and scalability. This is most likely OK, since maximum performance is not always the most important thing.

What you can do is to make sure that all of the parts of the service or services you're responsible for are optimized as much as possible. You can use the same techniques we covered in earlier chapters to accomplish this.

If you have a layered service, like the reference architectures we saw previously, you can scale the presentation services (as long as they are web forms) on a web server farm. Web Services Interfaces in our entity services can also be scaled this way. You can optimize performance internally as well, since the service is a single unit. This does mean that it might be great to use .NET Remoting for communication between components.

There are a lot of tricks you can use. Consider your services carefully and choose performance enhancers where they fit. Try to find and adjust the bottlenecks you can and have control over.

Other Issues to Consider

Making the choice of an SOA for a given domain is a delicate task. It's not uncommon for an architect to focus on the online transactions between one or more services, but then forget (or at least postpone) to think about the architecture of offline transactions (also known as *batches*).

Once the architects realize that they've made this kind of mistake they, at best, *try* to run batches in the same manner as the online transactions (for example, one transaction at a time). But in medium to large systems though, this is devastating for the performance.

Consider a 7x24 system that is very transaction-oriented and that has a lot of online registrations as well as batch registrations (for example, large files whose content should be imported into the system on a regular basis).

In this case, the system must be available to the online users, with an acceptable level of performance 24 hours a day and the system must also be able to run the batches in a reasonable amount of time. These two requirements can be difficult to fulfil and most likely impossible to satisfy, if these things aren't carefully considered in the architecture.

The most common mistake is for an architect to handle all objects one at a time, which works fine for online transactions, but can cause huge problems with offline transactions. This can result in a thousand calls or more for single objects, when you're running a large batch for verifying a single object's correctness, for example. This has a devastating impact on performance for the batches—in particular if the objects are returned by another service master over the network—and it means that the batches are at risk of not ending in time.

Instead, the architect should design the architecture considering the offline transactions first, and then use the same routine for single, online transactions—*not* the other way around.

Also, the architect must carefully choose the scope of each offline transaction. In this case the architect has to decide how many objects that are affected by a single transaction before committing it. This has to be done in order to prevent poor performance for online transactions when they run simultaneously (i.e. increasing serializability).

It is also important for the architect to keep the traffic between service masters to a minimum; hence he or she should consider using cached data. Cached data can be stored very well in a database controlled by the consuming service master: and it doesn't have to be stored in-memory.

So what are we really saying here? Well, sometimes you need to think batch first and online processing second. This conclusion is not unique for the SOA architect though— it's applicable for any architecture that must consider both online and offline transactions, in medium to large 7x24 systems. Since many services will communicate over the internet this issue is even more important in such cases than when the services communicate solely inside the company network boundaries. Network performance will probably be better inside the company than across the internet which obviously increases performance.

Windows Communication Foundation

Finally, we'll mention some quick things about Windows Communication Foundation, formerly known as Indigo. According to Don Box, Windows Communication Foundation is "a unified programming model and infrastructure for developing connected systems." Windows Communication Foundation will ship with the next version of Windows (so far code-named Longhorn). The goal of Windows Communication Foundation is to simplify service-oriented development where autonomous applications are built using asynchronous message-passing. Windows Communication Foundation implements SOAP and other web services technologies (like the WS-* Specifications) that allow us to create better services. This new programming model will make it possible for us to add security, reliability, and transactions to SOAP-based applications.

Windows Communication Foundation will ship with Longhorn, but we'll also be able to download it for Windows XP and Windows Server 2003 as well. There will be small differences between these versions, as Windows Communication Foundation will be optimized to use all aspects of Longhorn. But Don Box contends that Windows Communication Foundation will provide the developer with a consistent service-oriented programming model, no matter which operating system you develop on.

Since Windows Communication Foundation uses SOAP as a communications mechanism, we can let our Windows Communication Foundation applications interact with other services on any platform as long as they also use SOAP (see Figure 7-22).

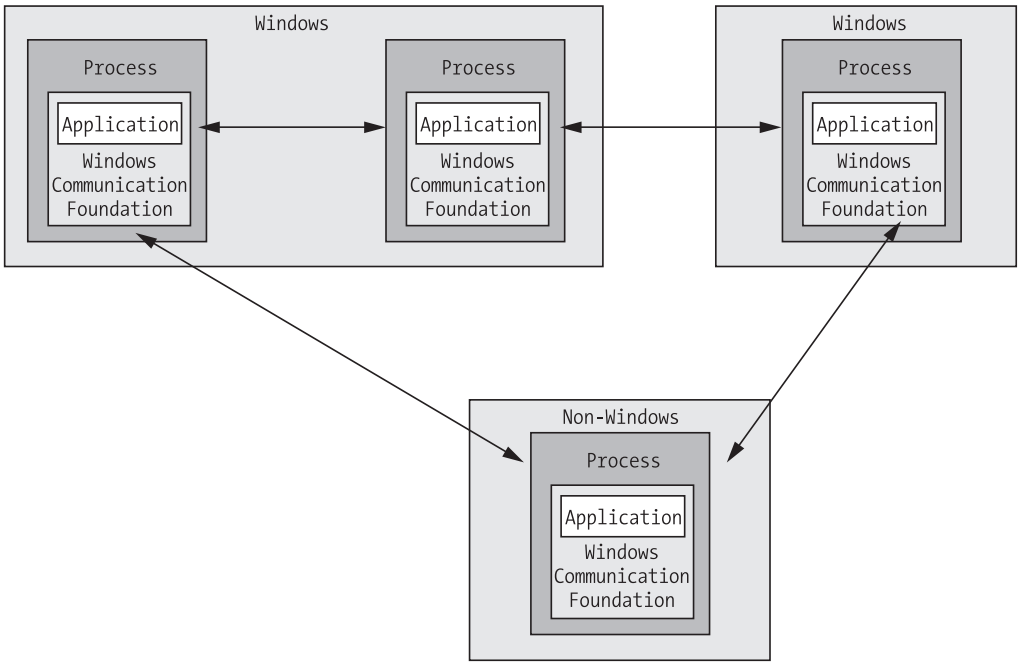

Figure 7-22. *Windows Communication Foundation will interact with any platform as long as SOAP is used*

What Is Windows Communication Foundation?

Windows Communication Foundation takes a bunch of existing technologies, including COM, COM+, MSMQ, .NET Remoting, ASMX, System.Messaging, and .NET Enterprise Services and offers their features as a single unified programming model for any CLR-compliant language (see Figure 7-23 for more information).

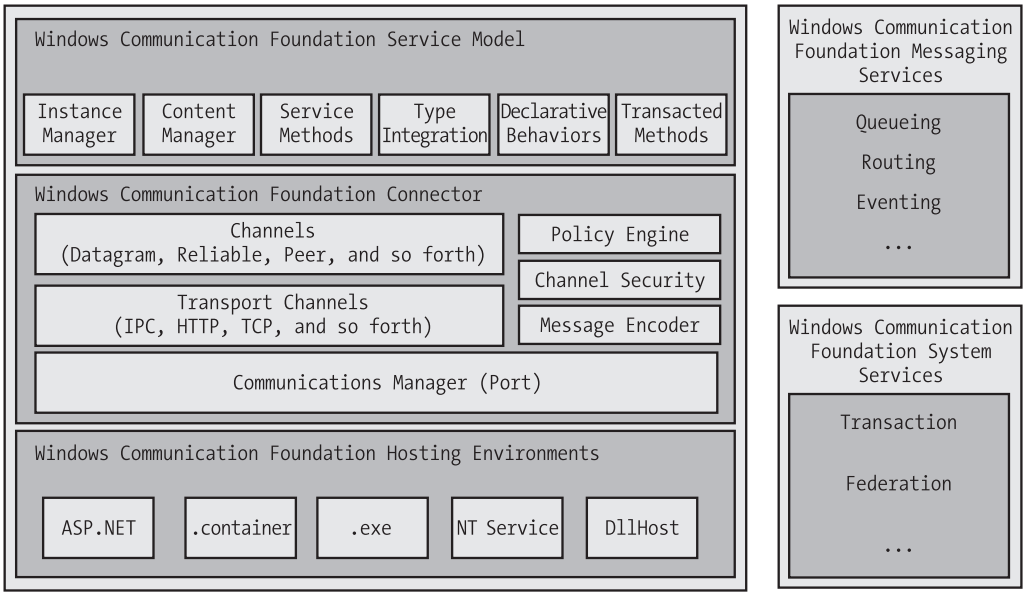

Figure 7-23. *The internals of Windows Communication Foundation, formerly code-named Indigo*

There are five major subsystems in Windows Communication Foundation.

- *The Windows Communication Foundation Service Model* has one major function and that is to associate incoming messages with user-defined code. When incoming messages are detected, Windows Communication Foundation routes them to an instance of user-defined service types. Windows Communication Foundation provides instance and context management for a service and allows developers to use declarative attributes to control the way instances are associated with incoming messages. Windows Communication Foundation's session management routes multiple messages to a common session object. By using declarative behavior, the service model can automate security and reliability as well.
- *The Windows Communication Foundation Connector* is a managed framework that provides both secure and reliable message-based connectivity. By using the SOAP data and the processing model, the connector allows us to let our services be independent of the target platform or transport.
- *The Windows Communication Foundation Hosting Environment* hosts systems like dllhost.exe, svchost.exe, ASP.NET and IIS, to mention a few.
- *System and Messaging Services* provide support for transactions, among other things. A service-based transaction manager, which developers can access via System.Transactions or the WS-AtomicTransactions protocol, is available. You can also find the WS-Federation implementation here, which allows you to securely broker authentication between the service and its corresponding trust authorities. You'll also find queuing and other stuff here.

At the time that we wrote this chapter, Windows Communication Foundation was available for download in beta versions. So, you have a chance of trying it out for yourself before Longhorn arrives, at least if you have a MSDN subscription. We as authors will be sure to do this as it looks promising and seems like it could make life easier for .NET developers.

Summary

This chapter has focused on Service Oriented Architecture and how this might differ from component-based applications. You have seen examples of services and how to use them. Implementing Service Orientation affects the whole company, not just the development of single applications. This means that it's quite an effort to implement SOA in your business. But, we believe that the future benefits are so great that you should definitely take a closer look at this area.