CHAPTER 1

Realizing the Promise of SOA

Those who do not remember the past are condemned to repeat it.
— George Santayana

Everyone has heard the many promises and benefits of Service-Oriented Architecture (SOA), and you've all probably heard a dozen different definitions of what SOA is or isn't. We're going to take a different approach. We want to paint a picture of what SOA can deliver and the promise of SOA, and then describe the challenges that organizations face in realizing that promise. Together, the vision and the challenges provide a set of requirements that the architecture must meet to make your implementation of SOA successful at delivering the promised benefits. Throughout the book, we'll describe the detailed architecture, design principles, and techniques that meet those architectural requirements, make the architecture actionable, and deliver results. In this chapter, you look at:

- What did and didn't work in the past
- The promise of SOA to the enterprise
- The challenges of delivering on that promise
- How to meet the challenge (the subject of this book)

But first, let's start with a little story. The scenario is true although the names have been changed.

Once Upon a Time . . .

Back in 1994, a major U.S. bank was trying to resolve a problem with customer service. Like pretty much every bank at that time, all of the different products (i.e., different types of accounts) were implemented on different mainframe systems. When you telephoned the customer service representative, you spoke to a beleaguered person with numerous green screen terminals on his or her desktop.

If you wanted information about your checking account, the customer service representative went to one terminal and entered your account number. If you wanted information about your savings account, the representative had to get a different account number from you and enter that in a different terminal. Each account system had a different interface. Together, they provided a confusing mix of commands and interaction that necessitated expensive training and was error prone. Customer satisfaction with problem resolution was low, employee satisfaction was low, and retention of both was problematic.

So what's a bank to do? First, they set about rationalizing the interface to all of the systems into a consistent interface, on a single terminal. Solutions such as 3270 emulators and PCs were tossed around but discarded because they only reduced the number of terminals, not the complexity of multiple interfaces. Instead, the bank took a gamble on a relatively new, distributed technology, Common Object Request Broker Architecture (CORBA).

The specific technology they chose is less important than the approach. The first thing they did was to create distributed objects to represent the different types of accounts. These objects provided an abstraction layer between the user interface and the mainframe systems that actually implemented the accounts. Next, they wrote a new user interface, using Visual Basic (VB), that provided account information to the customer service representatives by accessing the different systems via the CORBA objects.

It took about 6 months to get the basic functions in place — a new user interface, VB/CORBA bridging, and simple account objects — and then they were able to start replacing some of the green screen terminals. At this point, they began to understand the potential of the approach. They had essentially implemented the beginnings of a 3-tiered application architecture by separating the presentation, business logic, and operational systems. Figure 1-1 shows a simplified view of their solutions.

The next enhancement was to implement a customer relationship object in the logic tier. What this did was to take any account number or customer name, find all of the accounts that belonged to that customer, and provide that information to the customer service representative. Now, the customers didn't need to keep track of all their different account numbers in order to do business with the bank. The next incremental improvement was to automatically look up

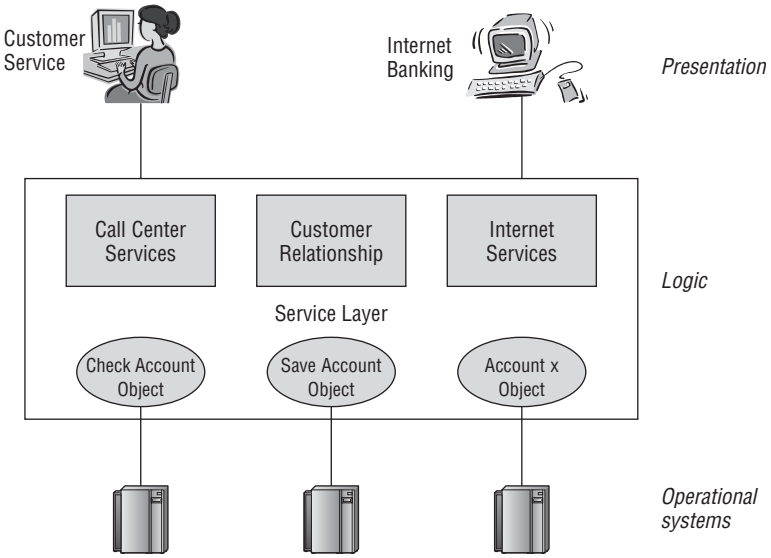

Figure 1-1 Bank customer service solution

information about each account and display a summary on the customer service representative's terminal. Now, without any additional effort on their part, the representatives had a broader view of the customers and a better understanding of their relationship to the bank. This allowed them to better serve the customers requests and at the same time offer additional value or services (i.e., turn a customer support scenario into a sales opportunity). Customer and employee satisfaction started to go up as the new approach started to pay off.

Over the next 2 years, the bank continued to provide more business objects in the logic tier and better features in the interface. The bank built up a library of about 250 objects (services) that served the needs of multiple channels, including the initial customer service representatives as well as ATMs and touch-tone dial-in systems. Things were going along smoothly in 1997 until a disruptive technology had a huge impact on banking, and everything else for that matter. All of a sudden, everybody wanted to do Internet banking.

Again, what's a bank to do? Well, while most of their competitors pondered the problem and scrambled to look at solutions like screen scraping, this bank didn't have to. They had invested in building up an architectural approach to the problem, namely separation of presentation from logic and logic from operational systems, and they had invested in building up an effective library of services in the logic layer. Therefore, all they had to do was implement a new Internet presentation. Of course, some minor changes to services were required as well as some new services to support security and other Internet specifics, but the bank's challenges were comparatively simple and they were

up on the Internet in less than 6 months. This was a full 6–12 months faster than their competitors, who struggled to catch up. And it was a real implementation that built toward the future, not a quick-and-dirty hack that needed to be replaced later. Many of the bank's competitors have never caught up.

Two years later, the bank merged with another major bank. This time the problem was how to integrate the new bank's systems into the other bank's Internet operations. Imagine the challenges involved, and imagine the surprise when 100% of the combined customers were able to access their accounts via the Internet on the first official day of merged operations! Okay, in reality, a few months were spent making this possible before the official opening day, but again the architectural investment paid off. Instead of adding a new presentation, the bank added new systems to the operational layer and enhanced the logic layer so that it was possible to access the new types of accounts and systems. Only very minor changes were required in the presentation layer.

Since the initial introduction of their Internet banking capability, the implementation and infrastructure has been enhanced to support tens of millions of transactions per day. And, since the merger, hundreds of other banks have been acquired and merged into the architecture. They were the competitors that never caught up, that never invested in architecturally sound IT solutions.

But of course, all of this didn't just happen by accident. The bank was fortunate to have a perceptive, skilled, and forward-thinking architect involved in the project. The architect quickly realized both the potential and the challenges and set about making changes to address them. First and foremost was the adoption of an architecture that distributed responsibilities across layers and tiers.

Second, the bank understood the challenge of creating the right kind of services in the logic tier and of having developers reuse them. To accomplish this, the bank created a new position, a reuse manager, for fostering and managing reuse. This person was responsible for helping developers create the right services with the right interfaces, helping presentation applications find and reuse services, and setting out an overall vision and roadmap of what services would be needed over time.

Finally, the bank realized that the existing organizational structure was not conducive to creating or using services. Instead of having monolithic application groups, they divided IT into groups that built the business services, and into other groups that used the services in their presentations and applications. After some obvious learning curves and attitude adjustments, the bank was able to drop the time to enhance or develop new user applications from 6 months under the monolithic model to 4–6 weeks under the service model. And, the more services that were added to the service library, under the careful direction of the reuse manager, the shorter this timeframe became.

So, with a successful implementation of SOA, the bank was able to improve customer retention and satisfaction, reduce costs and time to market, take

advantage of disruptive technologies, quickly absorb acquisitions, and keep ahead of their competitors. No wonder businesses are interested in SOA. From a more technical point of view, the bank was able to integrate multiple systems, support multiple channels and devices, scale horizontally to support very large-scale and highly reliable requirements, incrementally add new functionality, manage reuse, and converge on a common service infrastructure.

The moral of the story is this: SOA isn't about technology, and SOA doesn't just happen. SOA is an architectural approach to building systems that requires an investment in architecture and IT, a strategic and business vision, engineering discipline and governance, and a supporting organizational structure. Ignore these things and you end up with another broken promise. Put them together well, and you can deliver the promise and potential of agility, flexibility, and competitive advantage.

Learning from History

As can be seen from this story, SOA is not new. It has been around for years, well before the term was coined, by most accounts, in 1996. Forward-thinking companies like the bank whose story was told earlier, and many other finance and telecom companies were able to implement service layers using a variety of distributed technologies, including CORBA and the Distributed Common Object Model (DCOM). Other technologies, like Tuxedo, were inherently service-oriented and stateless, and formed the basis of some of the largest, high-performance, distributed applications of their day.

So while it is not difficult to find companies that were successful in implementing SOA, it's much easier to find companies that failed in their SOA. IT graveyards are filled with failed projects and sometimes the vendors of the technologies that promised the elusive, and ultimately ineffective, silver bullet. Figure 1-2 shows a brief timeline of SOA activity.

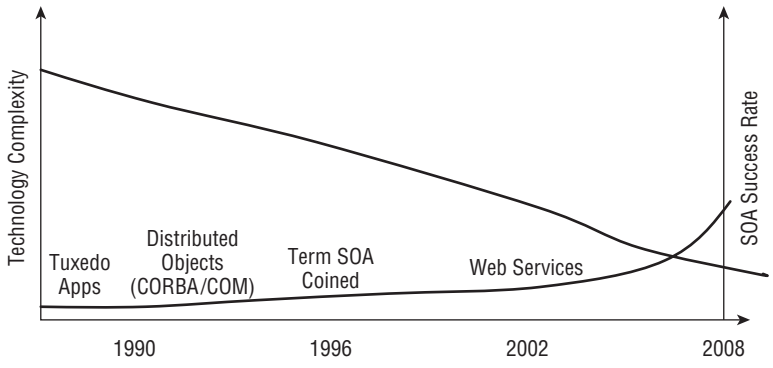

Figure 1-2 SOA timeline

What Went Wrong?

You might ask why some projects succeeded while others failed. Luckily, you have the opportunity to look back and examine both the successes and failures to discover patterns, and to then plan a path forward that avoids the failed behavior and embraces the successful activities.

Looking at the failures uncovers two main patterns. First, the technologies that we mentioned were too difficult for the average programmer to master. Distributed computing with CORBA or DCOM was just too difficult for the masses. Sophisticated IT departments had the system programmers and architects to manage these technologies, but most organizations did not. Visual Basic (VB) programmers and other client/server Rapid Application Development (RAD) application programmers didn't cut it, and the underlying platforms did not have enough of the complexities of distributed applications built into them.

The other problem was that, as an industry, we had not yet figured out what a good service was. No one knew what the right characteristics of a service or its interface or interaction style were. And if you could figure these things out, you then had to describe them in a service abstraction, and finally implement the service abstraction on top of the object abstraction naturally provided by the distributed technology. Again, some sophisticated people figured all this out, but most didn't. The hurdles to create any service were so great that most attempts failed well before the developers had to worry about whether they were building good services or what SOA meant, how to build it, or how to use it.

The situation today is much better. Web Services are much easier to use than previous technologies. This is not because the technologies are really any simpler (see the sidebar ''It's Not So Simple''), but mostly because the tools and environments have advanced greatly. It is now possible to develop services without really knowing what a service is or anything much about distributed technologies (we can debate whether this is good or bad later . . .).

Instead, the implicit knowledge of distribution and services is built into the platform, whether it is based on Java, .NET, or something else. And the service abstraction layer is built into the Web Service technologies.

IT'S NOT SO SIMPLE

Distributed technologies have had a long history, a history that tends to repeat itself. In the early days, we came up with the Distributed Computing Environment (DCE). Originally, this was a Remote Procedure Call (RPC) mechanism aimed at allowing different UNIX systems to communicate. Once the basics were worked out, people tried to use it for real enterprise applications and realized that it needed more capabilities such as security, transactions, reliability, and so on.

Next was CORBA, a mechanism for distributing objects. Initially, it was pretty simple, until people tried to use it to create real enterprise applications. Soon they realized that it needed security, transactions, reliable delivery, and so on, and it became complicated.

So a simpler technology was invented, Java. And all was well and good until people tried to use it to build real enterprise applications. All of a sudden it needed to have security, transactions, messaging, and so on.

Finally, Web Services came along, invented by developers so ignorant of history that they actually had the audacity to call the protocol SOAP, Simple Object Access Protocol. And all was fine until people tried to build real applications with it and discovered that they needed security, transactions, reliable messaging, and so on. You know the rest.

Hmm. What will be next?

What Went Right?

If you look at what worked, you get a broader picture. Not every company that mastered the technology managed to succeed with SOA. As has always been true with IT, technology alone is not enough to solve business problems.

The first thing that successful companies had was an understanding of not only how to use the technology but also what to do with it. These companies had an architectural vision that described the construction of applications in terms of a logical distribution of responsibility across tiers. The architecture went on to describe how services fit into that mix, what services were, how to build them, and how to use them.

The next, and equally important, aspect shared by successful companies was a business vision that described what business the company was in, what information and processes were necessary to run the business, what capabilities were needed to support those processes, and what services were needed to provide those capabilities. In addition, the vision included a roadmap that allowed for a prioritization and ordering of service implementations.

The vision and roadmap were combined with processes that helped the organization implement them. Two major aspects of this were: first, to help applications use existing services and, second, to help service providers create the right services, ones that didn't overlap with existing services or leave gaps in the roadmap.

Another aspect of successful SOA implementations was a structure that supported the consumer-and-provider nature of services. In addition to an organizational structure that separated these roles, the underlying architecture and infrastructure supported the discovery and publishing functions of consumers and providers.

Finally, the architecture and process were tied into an implementation methodology that supported the use and creation of services within applications and was informed by the overall enterprise context, business vision, and roadmap.

What Can You Learn?

So, what can you learn from this? First, success is not based on the technology. Technology can cause you to fail, but it doesn't make you succeed. Although previous technologies were too hard for most organizations, and the current technologies and tools are much better, there is more to it. You need to know how to use the technologies to build enterprise applications, not just isolated services. This requires architecture, vision, reuse, process, and organization, as illustrated in Figure 1-3.

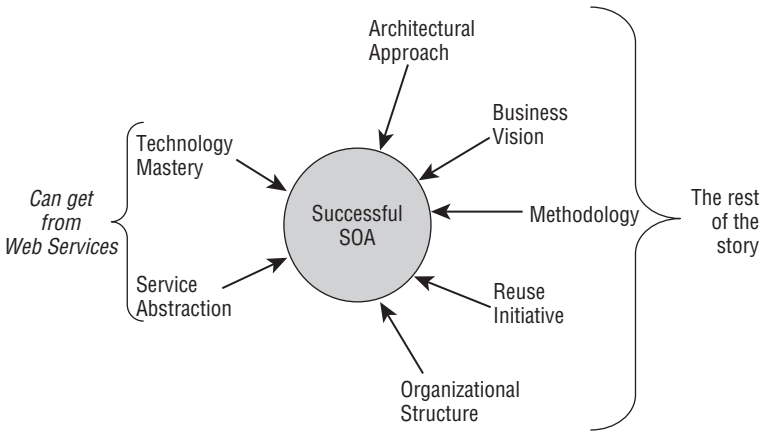

Figure 1-3 Ingredients of historically successful SOA

The Promise of SOA

Another way to assess the promise of SOA is to look at the motivations and expectations of the people who are engaged in SOA activities. In a 2006 survey conducted by the Cutter Consortium, the motivations for SOA included a range of technical and business reasons. The most common motivations were: agility, flexibility, reuse, data rationalization, integration, and reduced costs. Some of the more telling specific responses included:

- ''Strategic reuse of assets across multiple department's applications''
- ''Need to provide more agile support to business processes, and to handle change management impacts more efficiently and effectively''

- ''Master Data Management''
- ''Speed and ease of project deployment, concerns with duplication of work between projects''
- ''Support external collaborators''
- ''Efficiency in terms of time to market and development cost''
- ''Bring together diverse lines of business across many geographies with faster speed to market''
- ''Integrate legacy systems''

Not surprisingly, the motivations for adopting SOA echo the concerns that most enterprise IT organizations are struggling with.

The Challenges of SOA

If we examine the history and look at the goals or motivations for SOA, we can determine the challenges that organizations face in delivering on its promise. Let's restate the expectations, history, and goals in terms of four questions and then look at the issues they raise and the corresponding architectural requirements.

- What is required to provide agility, flexibility, and the strategic reuse of assets across multiple departments?
- What is required to bring more efficiency in terms of time to market and development costs, while delivering new capabilities to the organization?
- How will the integration of existing applications or enterprise data help to bring together diverse lines of business across geographies with faster time to market?
- How will SOA's agility and flexibility improve relationships and provide better alignment of business and IT?

Figure 1-4 illustrates the four major challenges facing SOA adoption today.

Reuse

Reuse seems to have been the holy grail of software for decades. But the objects and components failed to live up to the promise of the marketeers. Now, services are the next great hope for reuse. If we're smart enough to learn from the past, we can be more successful with services. SOA will march on either way (see the sidebar ''Does SOA Need Reuse?'').

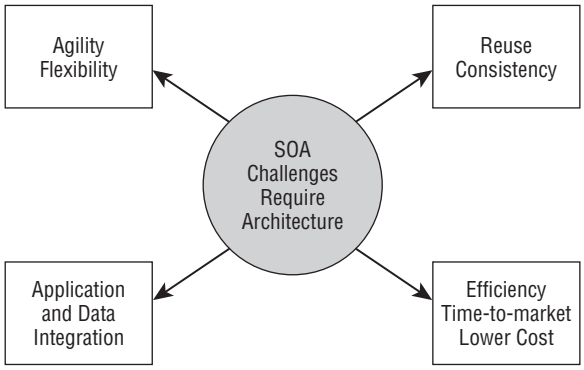

Figure 1-4 SOA challenges

DOES SOA NEED REUSE?

The object revolution of the late 1980s promised great increases in productivity and reductions in cost based on reuse. However, the reuse didn't really happen, except in some limited situations. But, it turns out that object orientation provides a better paradigm for development of complex software systems and that it is the prominent model supported by development tools. Every time you use a web page, you see object reuse. Thus, it has been widely adopted in spite of not attaining the promise of custom object reuse.

Components came along in the 1990s, promising to solve the reuse problem that objects hadn't. The advantage of components was that they provided a way to package functionality that matched the distributed, web-based systems that were being built. Once again, reuse was not achieved on a large scale. Yet, components are entrenched in modern systems because they bring with them all of the advantages of application servers such as distribution, scalability, and redundancy.

Now, the 2000s bring back the promise of reuse with services. Services provide a larger-granularity, run-time unit of functionality and reuse. Will enterprises be any more successful in achieving reuse with services than with previous technologies? At one level, services may not make that much difference. The march toward service orientation is well underway. Product vendors are structuring everything from infrastructure to software applications to development tools to support a service-oriented approach. Similar to objects, the advantages of services as a construction paradigm for enterprise applications will make SOA a reality regardless of how much the independently developed services actually get reused. So, services will probably be the future architectural and development paradigm, if for no other reason than because they are better for the software providers that provide infrastructure, tools, applications, Independent Software Vendors (ISVs), and so on.

However, many of the benefits that organizations hope to achieve with SOA require that services be reused within their environment. Those enterprises that achieve reuse will reap more of the benefits, be more agile, and be more competitive. Therefore, it behooves us to look at what did and didn't work in terms of reuse, and apply those lessons to services. Guess what? In every instance, technology was not the issue when it came to reuse. It's true that services have some technical features that make them better for reuse than components, just as components had technical features that were superior to those of objects. But the main roadblocks to reuse have, and will continue to be, organizational, methodological, and political.

Let's look at these issues from the perspective of the service consumer. When an application or process wants to use a service, it first needs a way to find and evaluate candidate services. Then, once it decides to use the service, it has dependencies on that service. Therefore, the service consumer needs to be guaranteed that the service will operate reliably, that bugs will be fixed in a timely manner, that requests for enhancements will be considered, that it will continue to operate and be supported for a reasonable amount of time, and, most importantly, that new versions of a service won't cause existing consumer applications to stop working. To make things more complicated, in an enterprise, the service consumer often needs to rely on another organization for that guarantee.

The following list discusses the architectural requirements for effective reuse:

- The ability to publish, search for, evaluate, and register as a consumer of a service
- Sufficient variability in service function to meet consumers' needs
- Capabilities for managing and maintaining a service life cycle across organizational boundaries
- The ability to guarantee the availability and lifetime of a service version
- Mechanisms for decoupling the consumer's life cycle from the provider's

CONSISTENCY, CONSISTENCY, CONSISTENCY

We often promote reuse as a way to reduce development costs or time to market. Although you can achieve improvements in both these areas, often it is consistency that is the most important value of reuse. SOA allows you to separate access to functions or data such that every application that needs to make use of the function or data can use the same service to get it.

(continued)

CONSISTENCY, CONSISTENCY, CONSISTENCY *(continued)*

How many enterprises suffer from redundant data or applications? (All of them, probably.) What is the result? Users get different results depending on how they go about doing something. When the users are customers, this results in dissatisfaction and lost customers. You've all heard of problems such as a customer having to call multiple different departments to correctly change his or her address, or an item being available through one system, but not another.

Imagine an enterprise-wide customer service that manages the shared customer information (such as addresses) for all systems and only needs to be changed once. Or, a single inventory service used by all order-management processes where they get consistent results about availability. SOA provides an approach for consistency of processes and data for both internal and external customers. This is something that the business sponsors understand and are often more willing to pay for than the promise of reduced costs and reuse.

Efficiency in Development

Making development more efficient means building more functionality, in less time, at less cost. Doing so depends on a variety of factors, including the reuse of services and the ability to quickly compose applications from those services. This in turn requires a different approach to service and solution development than the approach that was used in the past.

Developers of services can no longer create services in isolation, but rather, the services must fit into the overall architecture and conform to the enterprise business and information models. However, the initial version of a service cannot be expected to meet the requirements of all possible, future users. There has to be a managed process for deciding on, funding, and implementing enhancements to accommodate those additional users. But at the same time, enhancements to services need to be done in a controlled fashion that maintains the integrity of the service architecture and design, and conforms to versioning and compatibility requirements.

Developers of solutions that will consume services need to be able to easily find existing services and to evaluate them, determine what they do, and request enhancements. Furthermore, methods and tools for modeling and composing business processes from existing services need to be established. When projects are implementing business processes, a system design methodology is needed that focuses on composing business processes from

the existing services. And, there has to be a variety of different kinds of services available, at different levels of organizational scope and granularity, to fully support the composition of business processes.

There also has to be an analysis and design methodology for the services themselves that describes the characteristics of the different types of services and explains the interaction, interface, and implementation design decisions.

Finally, there have to be organizational changes to support service development and use across the enterprise that match the consumer and provider nature of services.

The following architectural requirements are necessary for effective development productivity:

- Have a reference architecture that guides the development of services.
- Use Business Process Management (BPM) to define business processes, based on service composition and a layered set of services. Use BPM to drive the discovery and design of required services.
- Have efficient processes that manage the integrity of the total set of services for both providers and consumers in accordance with the overall vision and the business and information models.

Integration of Applications and Data

The integration of existing applications and data is perhaps the most perplexing challenge facing enterprise IT organizations. Billions have been spent over the past decades on enterprise application integration (EAI) to implement application integration, but results are mixed. Too often, fragile and unmaintainable solutions have been put in place that created a rat's nest of point-to-point connections over a variety of different technologies and protocols.

SOA, based on Web Services, promises to simplify integration by providing universal connectivity to existing systems and data. But, as with everything else, technology is only a small part of the solution. Again, you can look at what did and didn't work with EAI to craft a strategy for moving forward. And when you do, you see that an overall, enterprise-wide, architectural solution is required. You should no longer be connecting individual applications directly with point-to-point connections, but rather, providing services that connect individual applications into the overall enterprise.

The really hard part, however, is getting the new interfaces to the existing system right. Here, the tools are often our own worst enemy. The vendors

trumpet their wiz-bang Web Services Description Language (WSDL) generators that can take an existing schema or application programming interface (API) and generate a service interface. Although this is seductive, it is wrong. You should not be exposing the data models or APIs of 20-year-old applications directly as services. The chances that these old APIs represent what your enterprise needs today are slim at best. Instead, you should transform them into new interfaces that meet the strategy, goals, and requirements of the enterprise today and in the future.

A similar situation exists for data integration. How many millions were spent on failed projects to implement a global enterprise data model? Too often, applications could not be retrofitted to the model, the cost of change was too high, or business units wouldn't go along with the changes. Yet, for services to fit together into a business process or to be composed together in a meaningful way, they have to share a common data model and semantics. Here's the difference, however: They do not have to agree on every single item and field of data. They have to agree only on what the shared, enterprise-wide data should be. Then, each application can translate between its own, internal version of the data and the shared, enterprise (external) representation of the data.

The following architectural requirements are necessary for integration:

- Have an enterprise, common semantic model for the shared information.
- Have a reference architecture that differentiates between business services and integration services.
- Have a reference architecture that describes common patterns for integration.
- Have infrastructure capabilities that enable semantic transformation between existing systems and the enterprise model.

Agility, Flexibility, and Alignment

Agility and flexibility occur when new processes can quickly and efficiently be created from the existing set of services. Achieving agility and flexibility requires an easily searchable catalog that lists the functions and data provided by the available services. In addition, an efficient way to assemble the business processes from the services needs to be available.

The services that compose the catalog must support a variety of different processes, at a variety of different levels, and have minimal gaps or overlaps in functionality. At the same time, the services must share and conform to a

common enterprise semantic model. This doesn't just happen by itself, or by accident. The SOA architectural approach needs:

- A business architecture that lays out a roadmap for the processes and services of the enterprise now and over time, and identifies the functional and application capabilities to support those services. In addition, the business architecture needs to specify the desired outcomes so that business processes can be measured against achieving them.
- An information architecture that lays out a roadmap for the shared enterprise semantics and data model.
- An application architecture that defines a hierarchy of service types, how to compose processes from services, how to produce and consume services, and how to measure services contributions toward business outcomes.
- A technology architecture that defines what the technologies are and how they are used to support processes, services, integration, data access and transformations, and so on.

Obviously, business needs to be involved in the development of the enterprise business and information architecture and roadmaps. But, that alone does not achieve alignment of business intentions with implemented IT systems. There has to be a process that directly integrates the enterprise architecture (business, information, application, and technology) into the development process. In addition, there needs to be an organizational and governance structure in place to support and enforce it.

The following list defines requirements of SOA for alignment:

- Have a reference architecture that defines the business and information aspects of SOA and their relationship to the enterprise.
- Have an enterprise, common semantic model that is used to inform the service interface design.
- Use model-based development techniques to ensure traceability between the business models and the implemented systems.
- Have processes that enable and validate conformance.

Table 1-1 summarizes the overall architectural requirements needed. Obviously, there is some overlap between the architectural requirements for the different challenges. This is a good thing. It indicates that a holistic architectural approach can not only meet the different challenges but also integrate the solutions.

Table 1-1 Summary of architectural requirements

CHALLENGE	ARCHITECTURAL REQUIREMENT
Reuse	Ability to publish, search for, evaluate, and register as a consumer of a service.
	Capabilities for managing and maintaining a service life cycle across organizational boundaries.
	Ability to guarantee availability and lifetime of a service version.
	Mechanisms for decoupling the consumer's life cycle from the provider's.
Efficient Development	Have a reference architecture that guides the development of services.
	Use BPM to define business processes, based on service composition and a layered set of services.
	Have efficient processes that manage the integrity of the total set of services for both providers and consumers in accordance with the overall vision and business and information models.
Integration of Applications and Data	Have an enterprise, common semantic model for the shared information.
	Have a reference architecture that differentiates between business services and integration services.
	Have a reference architecture that describes common patterns for integration.
	Have infrastructure capabilities that enable semantic transformation between existing systems and the enterprise model.
Agility, Flexibility, and Alignment	Have a reference architecture that defines the business and information aspects of SOA and their relationship to the enterprise.
	Have an enterprise, common semantic model that is used to inform the service interface design.
	Use model-based development techniques to ensure the traceability between the business models and the implemented systems.
	Have processes that enable and validate conformance.

Meeting the Challenge

Examining the promise of SOA and the goals of the organizations that adopt it leads to a set of requirements for meeting the challenges laid out in this chapter. Let's summarize the requirements for SOA.

Reference Architecture

Creating and maintaining a reference architecture is one of the more important but difficult best practices for SOA and is an important critical success factor in achieving SOA goals. Yet, often, organizations will have only an informal architecture, or none at all. Figure 1-5 shows the major components of an SOA reference architecture. The reference architecture represents a more formal architectural definition, one that can be used for objective validation of services and applications. For SOA, the reference architecture should:

- Support enterprise concepts, particularly the subarchitectures of business, information, application, and technology
- Specify a hierarchy and taxonomy of services and service types
- Define how services fit into an overall enterprise application, such as a portal
- Provide a separation between business, application, and technology concepts
- Be integrated into the development process

Chapter 2 covers the reference architecture in detail.

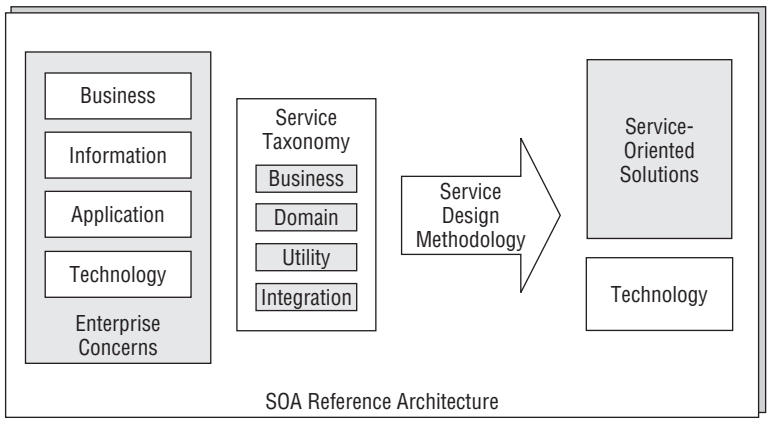

Figure 1-5 Aspects of an enterprise SOA reference architecture

Common Semantics

Defining a common, enterprise semantic and information model is key to achieving agility and flexibility. Without it services cannot be easily combined to form meaningful business processes. For example, imagine a process that combines different travel activities, such as air, hotel, and rental car into a trip based on a customer and their companions. The customer wants to see all of the related activities and only wants to provide the information once. The airlines

require the names of all passengers; the rental car agency needs to know if additional travelers are over age 25 and their relationship to the primary traveler; and the hotel needs a different set of information. If the services don't have some common understanding of what a customer is, and what a travel companion is, it won't be very easy to automate the combined processes or provide a single view or interface to the customer. Without common understanding, rather than agility and flexibility, each process requires special case code to combine the data. The common semantics should:

- Identify information that must be shared across the enterprise and between services
- Define the meaning and context of that information
- Identify techniques for mapping enterprise semantics to existing application data models

Chapter 5 describes the development of the common semantic model, and Chapter 6 shows how it is used in the design of service interfaces.

Governance

Governance has been defined as the art and discipline of managing outcomes through structured relationships, procedures, and policies. Governance enforces compliance with the architecture and common semantics and facilitates managing the enterprise-wide development, use, and evolution of services. Governance consists of a set of policies that service providers and consumers (and their developers) must conform to, a set of practices for implementing those policies, and a set of processes for ensuring that the policies are implemented correctly. There is typically an organizational structure in place to define and implement governance policies and often a repository to automate and enforce them. Governance of SOA should include:

- Policies regulating service definition and enhancements, including ownership, roles, criteria, review guidelines, and so on.
- Identification of roles, responsibilities, and owners.
- Policy enforcement that is integrated directly into the service repository (where appropriate).
- Guidelines, templates, checklists, and examples that make it easy to conform to governance requirements.
- Review of service interface definitions for new services and enhancements to existing services. The review ensures that the service definition conforms to standards and aligns with the business and information

models. The review is typically done by a service review board or the unit responsible for the service.

- Architectural review of solutions and services to ensure that they conform to the SOA and enterprise architecture. This review is typically done by an architecture review board.

Warning! Governance should not be primarily a review activity. If architecture is nothing more than extra steps in the process or a burden to developers, they will just ignore it. Effective governance follows a carrot-and-stick approach with an emphasis on enabling developers to build conforming applications (the carrot) and automating governance activities and policies. Reviews (the stick) should be a final check where process is minimal and exceptions are actually the exception.

We've seen countless articles and presentations (surprisingly by vendors) that talk about governance as a required activity from day 1. But we don't agree. There are enough challenges and barriers to get over for SOA to work, that you don't need another one to start with. When you have only a few services, you don't need a lot of processes to govern them. Figure out how to build and use services first, and then add governance. If you have to go back and correct things, fine. Certainly make sure that you have governance before you have 100 services, but don't put it in place when you have only one service. Chapter 12 discusses governance.

TYPES OF GOVERNANCE

We often discuss governance in terms of four different aspects of a service's life cycle:

- **Design-time governance — Policies and procedures to ensure that the right services are built and used**
- **Deploy-time governance — Policies that affect the deployment of services into production**
- **Run-time governance — Policies that affect the binding of consumers and providers**
- **Change-time governance — Policies and procedures that affect the design, versioning, and provisioning of service enhancements**

We have primarily discussed design-time and change-time governance as architectural requirements. Obviously, deploy-time governance is important for operational quality. Although automated run-time governance functions can provide benefits and sophistication to a SOA implementation, we don't think that it is a critical factor in achieving overall SOA success and value. Of course, it is important to specify policies regarding security and the authorization of

(continued)

TYPES OF GOVERNANCE *(continued)*

service consumers and providers. However, many successful SOA implementations today use very simple mechanisms to implement this rather than a sophisticated registry to automatically apply the policies during binding. On the other hand, governance of service interface design is necessary to achieve a consistent overall set of services, which is critical to achieving SOA success.

Business Process Modeling

Business processes need to change relatively frequently yet be based on stable underlying capabilities. The flexibility to do this comes from being able to quickly construct new business processes from business services, which are relatively stable. Business processes should:

- Be specified using Business Process Models and executed in a business process management system
- Be composed of activities that are implemented by business services (provided by the SOA)
- Pass information into, out of, and within the processes in the form of documents, which are built on top of the common information model

Chapter 4 describes the use of BPM in addressing business requirements and influencing service design.

Design-Time Service Discovery

To reuse services, you have to be able to find the services that exist, and you have to be able to examine them to see if they perform the functions required, provide the appropriate qualities of service, are reliable, and so on. It is important to understand the distinction between a run-time registry and a design-time repository, even though both functions may be implemented by the same software. A registry is used at run time to identify a service endpoint for a requested service interface. This is where run-time governance policies may be enforced. A repository is used at design time to find existing services for inclusion in processes during the design of that process. This is critical to enabling service reuse. Service discovery does not necessarily have to be based on a repository (although repositories do a good job of it) but should provide the following functions:

- A catalog of available services.
- Sophisticated search capabilities for identifying potential services.
- Capabilities for examining a service, its interface and implementation, and design and testing to determine if it is appropriate for the desired usage. This will often be through links to documents, models, code, reports, and the like that are stored in other systems.
- Metrics on service usage.
- Notification to interested parties about upgrades to services or other events.
- Automation of certain governance policies.
- Direct integration into the development environment.

In subsequent chapters, we provide methods for describing and categorizing services to assist you in locating them during development. Chapter 5 describes the creation and use of the service inventory in the discovery and design of service interfaces.

Model-Based Development

Model-based development is a best practice in software engineering in general and in SOA as well. Models provide a way to conceptualize and describe a system without getting bogged down in details, and to describe the major parts of a system and their relationships. A model-based development approach for SOA should incorporate the following:

- A higher level of abstraction for software development and the ability to visualize software and service designs
- Support for a domain-specific language (DSL) for the implementation of SOA
- Automatic integration of SOA reference architecture into the design environment and DSL
- Separation of business, services, and technology concerns

The design methodologies throughout this book use a model-based approach to SOA design, based on a set of SOA domain concepts and abstractions that make up a domain-specific language for SOA. Although it is helpful to be able to generate development artifacts directly from design models, and in fact many tools do exactly that, it is not strictly required. The proper design of services is critical to achieving SOA goals, and models are the lingua franca of design. What is required is the design of service interfaces and

implementations, and a way to pass those design models to development as specifications for construction. Of course, the more you can generate, the easier and less error prone that hand-off will be. Chapter 7 focuses on the technology-independent design of service implementations that lead to a model-based approach.

Best Practices in SOA Analysis and Design

There's a clever saying that goes ''In theory, there's no difference between theory and practice, but in practice there is.'' This difference is most often seen in the clash between architecture and development.

The architecture team is responsible for understanding the big picture. They must answer questions such as: How will SOA support the overall enterprise goals? How will it fit with other initiatives such as Single Sign-On (SSO)? What standards and technologies are important? How do they fit in with the enterprise technology roadmap? What strategy and tactics should be employed to introduce and phase in SOA? How will it be sold to management and the business? All of these are important and difficult questions that must be answered, and the architecture team or steering committee is the right place for this. We often call this a top-down approach.

The development team is responsible for implementing and deploying individual services. They have a different set of questions to answer: How will an individual service be implemented? How will the master data definition be translated to the individual systems of record that contain the data? How will the service be deployed? How will the service be managed? How will new versions be implemented and deployed? How will services be registered and discovered at run time? How will services be discovered and reused at design time? How will dependencies be minimized and managed? Again, these are very important and difficult questions that must be answered. We might call this a bottom-up approach.

With these questions and concerns, the architecture team is trying to maximize the value that SOA can provide in the delivery of enterprise solutions. Value comes from enabling and creating an enterprise service layer that supports the flexible creation of business processes. Value comes from being able to quickly modify these business processes without having to make difficult and expensive modifications to existing operational systems. Value comes from having consistent behavior across the enterprise for the same business function (i.e., having the business function implemented in a single service). Value comes from having modular business capabilities that can be outsourced

or sold as a service. To support this, the SOA has to describe how the different organizations in the enterprise can contribute to the overall SOA, and at the same time, meet their immediate business requirements.

Meanwhile, the technical team is trying to provide value by implementing specific business functionality in the best, most efficient, and most cost-effective manner — not just in the short term, but with an eye toward the total cost of ownership of IT systems. The manager of a technical team we worked with put it best. He sees SOA as a way to minimize and manage the collateral damage caused by changes. We've all heard the horror stories, such as the case of adding two digits to a part number that required $25 million and 1 year to implement (but added no business value), because it touched on almost every system in the enterprise. By applying a separation of concerns, having a Master Data Schema, and a set of services to manage the fundamental business entities, the required changes could have been isolated and minimized.

The theory naturally leads toward a top-down approach in which processes and services are driven by an overall enterprise model. These projects are often started with a high-level business process model or an overall enterprise system analysis activity. The practice leads us to a bottom-up approach in which services are implemented to meet a specific, immediate business requirement or project. These projects often start by service-enabling legacy systems or incorporating simple external services. Yet neither of these approaches is very effective. In order to meet both the enterprise goals and the immediate project goals, these organizations and concerns have to meet in the middle. Chapter 3 describes the overall process of initiating SOA and designing services based on a middle-out approach.

Summary

Effective SOA (and architecture in general) is the careful balance and blending of the big picture and the immediate requirements. It is the practical application of theory to meet a set of goals, now and in the future. In this middle-out approach, the architecture team provides an overall SOA that offers the guidance and context necessary to support the implementation and reuse of services. This is provided as a set of guidelines, patterns, frameworks, examples, and reference implementations. The technical teams use these to incorporate the requirements (business and information context) into their designs so that the services they implement provide the necessary business functions that are needed immediately, but can easily be extended to support other processes

and services in the future. These are the roles of the reference architecture, the architecture-driven design process, and the domain-specific modeling approach.

Chapter 2 describes the SOA reference architecture and how it meets the challenges and requirements introduced in this chapter. It describes the overall enterprise context, the architectural layers and tiers, the domain-specific concepts and abstractions, and specifically what a service is and the important architectural characteristics of a service.