Chapter {2}

Service-Oriented Architecture

As we explored in the previous chapter, Web services promote an environment for systems that is loosely coupled and interoperable. Many of the concepts for Web services come from a conceptual architecture called service-oriented architecture (SOA). SOA configures entities (services, registries, contracts, and proxies) to maximize loose coupling and reuse. This chapter describes these entities and their configuration in an abstract way. Although you will probably use Web services to implement your service-oriented architecture, this chapter explains SOA without much mention of a particular implementation technology. This is done so that in subsequent chapters, you can see the areas in which Web services achieve some aspects of a true SOA and other areas in which Web services fall short. Although Web services are a good start toward service-oriented architecture, this chapter will discuss what a fully implemented SOA entails. We will examine the following issues:

What is SOA? What are its entities?

What are the properties of SOA?

How do I design an interface for a service?

Before we analyze the details of SOA, it is important to first explore the concept of software architecture, which consists of the software's coarse-grained structures. Software architecture describes the system's components and the way they interact at a high level.

These components are not necessarily entity beans or distributed objects. They are abstract modules of software deployed as a unit onto a server with other components. The interactions between components are called *connectors*. The configuration of components and connectors describes the way a system is structured and behaves, as shown in Figure 2.1. Rather than creating a formal definition for software architecture in this chapter, we will adopt this classic definition: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally

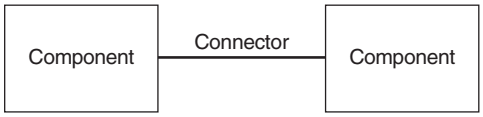

Figure 2.1 Software architecture describes a system's components and connectors.

visible properties of those components, and the relationships among them." (Bass, Clements, and Kazman 1997)

Service-oriented architecture is a special kind of software architecture that has several unique characteristics. It is important for service designers and developers to understand the concepts of SOA, so that they can make the most effective use of Web services in their environment.

SOA is a relatively new term, but the term "service" as it relates to a software service has been around since at least the early 1990s, when it was used in Tuxedo to describe "services" and "service processes" (Herzum 2002). Sun defined SOA more rigorously in the late 1990s to describe Jini, a lightweight environment for dynamically discovering and using services on a network. The technology is used mostly in reference to allowing "network plug and play" for devices. It allows devices such as printers to dynamically connect to and download drivers from the network and register their services as being available.

The goal in developing Jini was to create a dynamically networked environment for devices, services, and applications. In this environment, services and devices could be added to and removed from the network dynamically (Sun Microsystems, *Jini Network Technology, www.sun.com/jini*). There is more interest lately in the software development community about the concepts behind SOA because of the arrival of Web services.

Figure 2.2 shows that other technologies can be used to implement service-oriented architecture. Web services are simply one set of technologies that can be used to implement it successfully.

The most important aspect of service-oriented architecture is that it separates the service's implementation from its interface. In other words, it separates the "what" from the "how." Service consumers view a service simply as an endpoint that supports a particular request format or contract. Service consumers are not concerned with how the service goes about executing their requests; they expect only that it will.

Consumers also expect that their interaction with the service will follow a contract, an agreed-upon interaction between two parties. The way the service executes tasks given to it by service consumers is irrelevant. The service might fulfill the request by executing a servlet, a mainframe application, or a Visual Basic application. The only requirement is that the service send the response back to the consumer in the agreed-upon format.

SOA Entities

The "find, bind, and execute" paradigm as shown in Figure 2.3 (Talking Blocks 2001) allows the consumer of a service to ask a third-party registry for the service that matches its criteria. If the registry has such a service, it gives the consumer a contract and an endpoint address for the service. SOA consists of the following six entities configured together to support the find, bind, and execute paradigm.

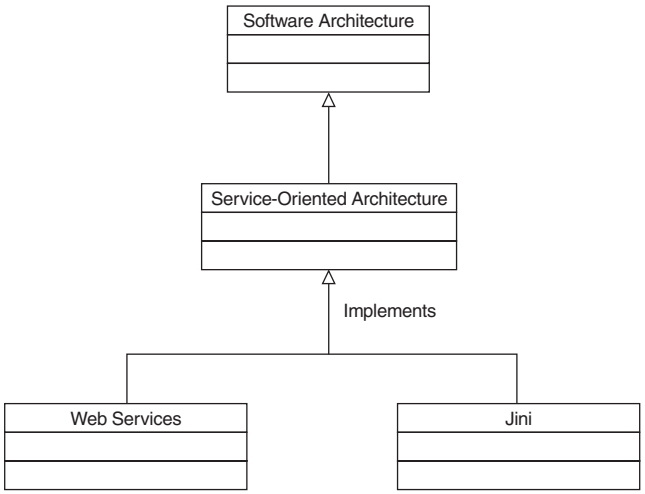

Figure 2.2 Web services are one set of technologies for implementing service-oriented architecture.

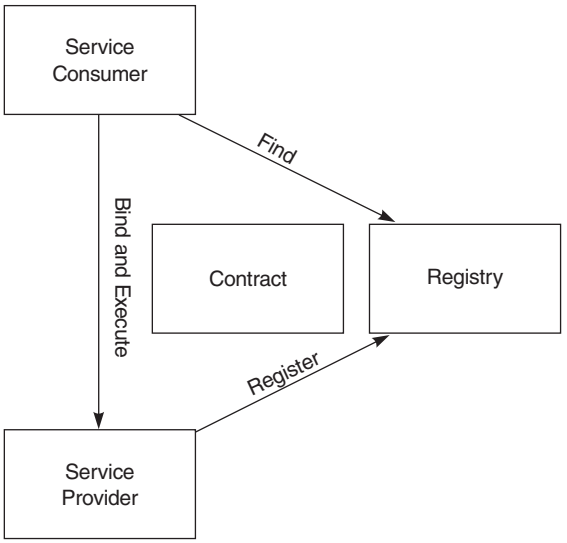

Figure 2.3 The "find-bind-execute" paradigm.

Service Consumer

The service consumer is an application, service, or some other type of software module that requires a service. It is the entity that initiates the locating of the service in the registry, binding to the service over a transport, and executing the service function. The service consumer executes the service by sending it a request formatted according to the contract.

Service Provider

The service provider is the service, the network-addressable entity that accepts and executes requests from consumers. It can be a mainframe system, a component, or some other type of software system that executes the service request. The service provider publishes its contract in the registry for access by service consumers. Chapter 3 describes the issues involved with creating a service provider by using component-based development techniques.

Service Registry

A service registry is a network-based directory that contains available services. It is an entity that accepts and stores contracts from service providers and provides those contracts to interested service consumers.

Service Contract

A contract is a specification of the way a consumer of a service will interact with the provider of the service. It specifies the format of the request and response from the service. A service contract may require a set of preconditions and postconditions. The preconditions and postconditions specify the state that the service must be in to execute a particular function. The contract may also specify quality of service (QoS) levels. QoS levels are specifications for the nonfunctional aspects of the service. For instance, a quality of service attribute is the amount of time it takes to execute a service method.

Service Proxy

The service provider supplies a service proxy to the service consumer. The service consumer executes the request by calling an API function on the proxy. The ser-

vice proxy, shown in Figure 2.4, finds a contract and a reference to the service provider in the registry. It then formats the request message and executes the request on behalf of the consumer. The service proxy is a convenience entity for the service consumer. It is not required; the service consumer developer could write the necessary software for accessing the service directly.

The service proxy can enhance performance by caching remote references and data. When a proxy caches a remote reference, subsequent service calls will not require additional registry calls. By storing service contracts locally, the consumer reduces the number of network hops required to execute the service.

In addition, proxies can improve performance by eliminating network calls altogether by performing some functions locally. For service methods that do not require service data, the entire method can be implemented locally in the proxy. Methods such as currency conversion, tip calculators, and so on, can be implemented entirely in the proxy. If a method requires some small amount of service data, the proxy could download the small amount of data once and use it for subsequent method calls. The fact that the method is executed in the proxy rather than being sent to the service for execution is transparent to the service consumer. However, when using this technique it is important that the proxy support only methods the service itself provides. The proxy design pattern (Gamma et al. 2002) states that the proxy is simply a local reference to a remote object. If the proxy in any way changes the interface of the remote service, then technically, it is no longer a proxy.

A service provider will provide proxies for many different environments. A service proxy is written in the native language of the service consumer. For instance, a service provider may distribute proxies for Java, Visual Basic, and Delphi if those are the most likely platforms for service consumers. Although the service proxy is not required, it can greatly improve both convenience and performance for service consumers.

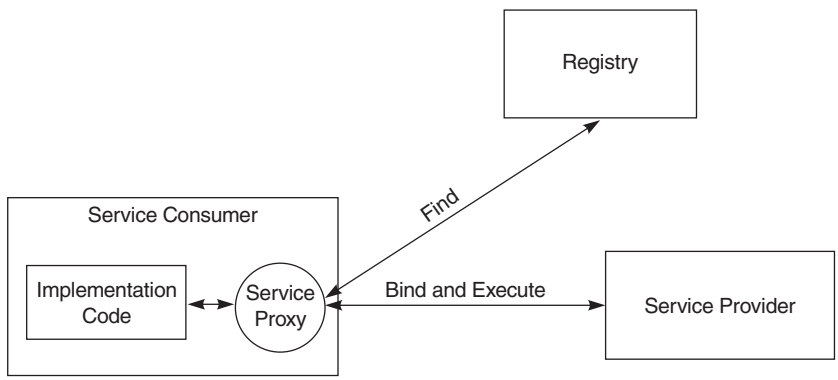

Figure 2.4 A service proxy.

Service Lease

The service lease, which the registry grants the service consumer, specifies the amount of time the contract is valid: only from the time the consumer requests it from the registry to the time specified by the lease (Sun Microsystems, *Jini Technology Core Specification*, 2001). When the lease runs out, the consumer must request a new lease from the registry.

The lease is necessary for services that need to maintain state information about the binding between the consumer and provider. The lease defines the time for which the state may be maintained. It also further reduces the coupling between the service consumer and the service provider, by limiting the amount of time consumers and providers may be bound. Without the notion of a lease, a consumer could bind to a service forever and never rebind to its contract again. This would have the effect of a much tighter coupling between the service consumer and the service provider.

With a service lease, if a producer needs to somehow change its implementation, it may do so when the leases held by the services consumers expire. The implementation can change without affecting the execution of the service consumers, because those consumers must request a new contract and lease. When the new contract and lease are obtained, they are not guaranteed to be identical to the previous ones. They might have changed, and it is the service consumer's responsibility to understand and handle this change. ▷

> SOA Characteristics

Each system's software architecture reflects the different principles and set of tradeoffs used by the designers. Service-oriented software architecture has these

▷ While Web services provide support for many of the concepts of SOA, they do not implement all of them. They do not currently support the notion of a contract lease. Also, no official specification provides QoS levels for a service. An organization cannot implement a complete service-oriented architecture given these limitations with Web services. In addition, service consumers can execute Web services directly if they know the service's address and contract. They do not have to go to the registry to obtain this information. Today, in fact, most organizations implement Web services without a registry. Consequently, the extent to which an organization implements an SOA with Web service varies greatly.

characteristics Bieber and Carpenter 2001, Stevens, *Service-Oriented*, 2002, Sun Microsystems, *Jini Technology Architectural Overview* 2001):

- Services are discoverable and dynamically bound.
- Services are self-contained and modular.
- Services stress interoperability.
- Services are loosely coupled.
- Services have a network-addressable interface.
- Services have coarse-grained interfaces.
- Services are location-transparent.
- Services are composable.
- Service-oriented architecture supports self-healing.

Discoverable and Dynamically Bound

SOA supports the concept of service discovery. A service consumer that needs a service discovers what service to use based on a set of criteria at runtime. The service consumer asks a registry for a service that fulfills its need. The best way to explain dynamic binding and discover is to use an example. For example, a banking application (consumer) asks a registry for all services that perform credit-card validation. The registry returns all entries that support this. The entries also contain information about the service, including transaction fees. The consumer selects the service (provider) from the list based on the lowest transaction fee.

Using a pointer from the registry entry, the consumer then binds to the provider of the credit card service. The description of the service consists of all the arguments necessary to execute the service. The consumer formats a request message with the data, based on the description provided by the directory pointer.

The consumer then binds the message to a transport type that the service expects and sends the service the request message over the transport. The service provider executes the credit-card validation and returns a message, whose format is also specified by the service description. The only dependency between producer and consumer is the contract, which the third-party registry provides. The dependency is a runtime dependency and not a compile-time dependency. All the information the consumer needs about the service is obtained and used at runtime.

This example shows how consumers execute services dynamically. Clients do not need any compile-time information about the service. The service interfaces

are discovered dynamically, and messages are constructed dynamically. The removal of compile-time dependencies improves maintainability, because consumers do not need a new interface binding every time the interface changes.

This method of service execution is powerful. The service consumer does not know the format of the request message or response message or the location of the service until the service is actually needed. If the transaction fees for the credit-card validation services changed from minute to minute, consumers could still ensure that they received the best price.

Self-Contained and Modular

Services are self-contained and modular. One of the most important aspects of SOA is the concept of modularity. A service supports a set of interfaces. These interfaces should be cohesive, meaning that they should all relate to each other in the context of a module. The principles of modularity should be adhered to in designing the services that support an application so that services can easily be aggregated into an application with a few well-known dependencies. Since this is such an important concept when creating services, we will explain some of the principles of modularity and, in particular, how they apply to the creation of services. Bertrand Meyer (Meyer 1997) outlined the following five criteria for determining whether a component is sufficiently modular. These criteria apply equally well when determining whether a service is sufficiently modular.

Modular Decomposability

The *modular decomposability* of a service refers to the breaking of an application into many smaller modules. Each module is responsible for a single, distinct function within an application. This is sometimes referred to as "top-down design," in which the bigger problems are iteratively decomposed into smaller problems. For instance, a banking application is broken down into a savings account service, checking account service, and customer service. The main goal of decomposability is reusability. The goal for service design is to identify the smallest unit of software that can be reused in different contexts. For instance, a customer call-center application may need only the customer's telephone number and thus need access only the customer service to retrieve it.

Modular Composability

The *modular composability* of a service refers to the production of software services that may be freely combined as a whole with other services to produce new sys-

tems. Service designers should create services sufficiently independent to reuse in entirely different applications from the ones for which they were originally intended. This is sometimes referred to as *bottom-up design*. Sometimes, the composability and decomposability approaches to service design can create two different designs. The bottom-up approach is more focused on the application functions. The top-down design tends to be more focused on the business problem. It is important to use both methods to find the right interface for a service.

The typical design process starts as a decomposition exercise. When the designers get to a point at which they have exhausted the top-down design, performing a bottom-up analysis should validate the design. The bottom-up analysis starts by defining the significant scenarios that the modules need to support. For instance, in a banking application, a scenario is "deposit money into checking account." The significant scenarios will cover the important functional aspects of the modular design.

Once designers define the scenarios, they create sequence diagrams to illustrate the messages that flow between modules to satisfy the scenarios. Once the scenarios are satisfied, the designer can perform additional iterations of bottom-up and top-down analysis to tune the design of the modules.

Modular Understandability

The *modular understandability* of a service is the ability of a person to understand the function of the service without having any knowledge of other services. For instance, if a banking application implements a checking account service that does not implement a deposit function but instead relies on the client to use a separate deposit service, this would detract from the service's modular understandability. The modular understandability of a service can also be limited if the service supports more than one distinct business concept. For example, a service called *CustomerCheckingAccount* that mixes the semantics of both a customer service and a checking account service also limits modular understandability. The modular understandability is especially important for services, because any unknown consumer can find and use a service at any time. If the service is not understandable from a functional perspective, the person deciding whether to use the service will have a difficult time making a decision.

Modular Continuity

The *modular continuity* of a service refers to the impact of a change in one service requiring a change in other services or in the consumers of the service. An interface that does not sufficiently hide the implementation details of the service creates a domino effect when changes are needed. It will require changes to other

services and applications that use the service when the internal implementation of the service changes. Every service must hide information about its internal design. A service that exposes this information will limit its modular continuity, because an internal design decision is exposed through the interface.

Modular Protection

The *modular protection* of a service is sufficient if an abnormal condition in the service does not cascade to other services or consumers. For instance, if an error in the checking account service causes invalid data to be stored on a database, this could impact the operation of other services using the same tables for their data. Faults in the operation of a service must not impact the operation of a client or other service or the state of their internal data or otherwise break the contract with service consumers. Therefore, we must ensure that faults do not cascade from the service to other services or consumers.

In addition to the above criteria for modularity, two rules ensure that a service's modularity and independence are not compromised: Direct mapping and contracts and information hiding.

Direct Mapping

A service should map to a distinct problem domain function. During the process of understanding the problem domain and creating a solution, the designer should create boundaries around service interfaces that map to a distinct area of the problem domain. This is important so that the designer creates a self-contained and independent module. For instance, interfaces that deposit, withdraw, and transfer from a checking account should map to the checking account service. This sounds simplistic, but it is easy to accidentally pollute a service's interface with functions that

- Logically belong in another existing service
- Belong in a new service
- Span multiple services and require a new composite service
- Are really internal knowledge that should not be exposed through an interface

To directly map a service's interfaces to a distinct business concept in the problem domain, the service designer needs a good understanding of the problem domain. Creating a conceptual service model provides this understanding.

Conceptual Service Model The conceptual service model consists of a model of the problem domain. Techniques for defining module interfaces assume that the problem domain is known a priori. In other words, the application's problem domain is known when the designers and developers create or enhance an application. With service-based development, this is not always the case. Services may be assembled into applications in the future that the service designer had no knowledge of when the service was designed. Therefore, designers must estimate service interfaces based on the service's expected use.

The conceptual model of the business, sometimes referred to as the *business architecture* (Fowler 1997), helps drive the expected use of the services. A conceptual model is one created without regard for any application or technology. It typically consists of a structural model derived from a set of use cases that illustrate how the business works. For instance, a bank manages checking accounts, savings accounts, and customer information. A conceptual service model for this domain might look similar to Figure 2.5, although it would be illustrated in much more detail.

This logical model of the business provides the basis for creating and managing service interfaces. Each entity in the logical model is either a stateful entity or

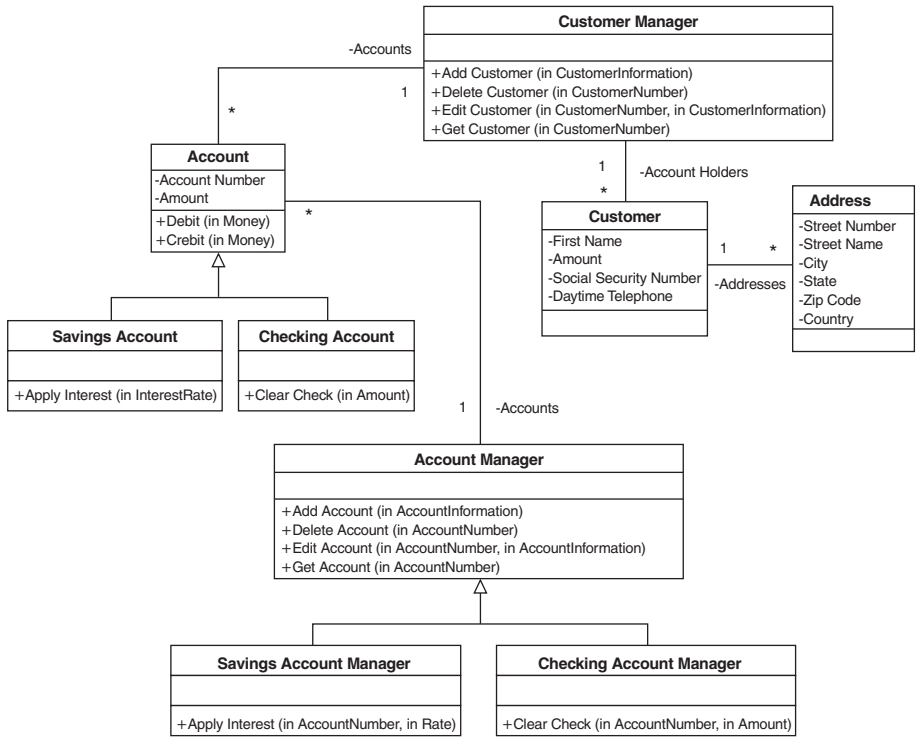

Figure 2.5 A conceptual service model.

a stateless entity. The manager classes are stateless entities, and the other classes are stateful entities. For example, the account entity contains attributes that contain the state of a single account; namely, account number and amount. The savings account, checking account, customer, and address entities also maintain state and are also stateful entities. These entities can be translated into software as entity beans and/or rows in databases. Each stateful entity also has a key that uniquely identifies it within the system.

The entity classes are not directly accessible to the service consumer in SOA. However, in component-based systems, a component consumer accesses an entity component by obtaining a handle to the component. The handle maintains a stateful connection to an entity that has a unique key to identify it. In service-oriented architecture, service consumers cannot access these entities. The service consumer accesses them indirectly by going through the manager interfaces. In SOA, these manager interfaces are implemented as service interfaces.

The manager classes in Figure 2.5 are stateless classes that manage entities of a particular class. They are the classes that perform create, read, update, and delete (CRUD) operations on the entities they manage. Because the manager classes do not represent a single entity but manage multiple entities, the interfaces for the manager classes require that a unique key be passed in. The unique key identifies the entity for which the action is to be performed. For instance, the *ApplyInterest* method of *SavingsAccountManager* requires the rate as well as the account number to identify the entity for executing *ApplyInterest* behavior. The *ApplyInterest* method on the *SavingsAccount* entity does not need a unique key, because it represents a single savings account instance.

The manager classes comprise the basis of the design of the service layer interfaces. The manager interfaces may be converted directly into service interfaces, and the stateful entities may be converted directly into persistent state. The persistent state may be an Enterprise JavaBean, a database row, or both. Stateful entities are not exposed outside the service. The service interface is a stateless interface. Services manipulate stateful entities on behalf of consumers, based on the method consumers call when requesting an operation to perform. Consumers pass in the unique key of the entity they are manipulating and the data for the operation. The service locates the entity that matches the unique key and performs the operation on it with the data.

The integrity of the service layer interfaces will be maintained only if the interfaces map directly to the logical model for the business. Because different applications will use the same services, the logical model must cross application boundaries. Developers should add functionality to services as new applications need those functions. The logical model provides the city plan for developing the service layer. As developers build applications, the software will support more of the logical model's functionality. It is difficult to maintain services' in-

tegrity over time, because new applications need to interact with services in different ways. The more closely this conceptual model maps to the overall structure of the business it supports, the longer-lived the service layer will be.

Direct mapping is only the first rule we need to implement for modularity. Contracts and information hiding is the second.

Contracts and Information Hiding

An interface contract is a published agreement between a service provider and a service consumer. The contract specifies not only the arguments and return values a service supplies but also the service's preconditions and postconditions. The preconditions are those that must be satisfied before calling the service, to allow the service to function properly. For instance, consider a credit-card validation service that is a two-step process. In the first step, the application sends the account number and amount information to the service. The service responds with an OK. However, for the transaction to go through, the consumer must send a confirmation message to the service. The precondition of the confirmation function is that the information for the confirmation has previously been sent.

The postcondition is the system's state after a function has been executed. The postcondition of the initial submission of information is that the information has been stored for a subsequent commit request.

Parnas and Clements best describe the principles of information hiding:

> *Our module structure is based on the decomposition criterion known as information hiding [IH]. According to this principle, system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. Each data structure is used in only one module; one or more programs within the module may directly access it. Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.*
>
> (Parnas and Clements 1984)

This statement assumes that the software executes in a single machine. With service-oriented architecture, we take this principle a little further. The service should never expose its internal data structures. Even the smallest amount of internal information known outside the service will cause unnecessary dependencies between the service and its consumers. Although the information stored in the data structures is necessarily exposed, that information must be trans-

formed from the internal storage structure into an external structure. In other words, the internal data semantics must be mapped into the external semantics of an independent contract. The contract depends only on the interface's problem domain, not on any implementation details.

Exposing internal implementation details is easy to do by creating an interface design with arguments that map to the service's implementation aspects rather than to its functional aspects. For instance, consider a credit-card validation service. The service requires that a credit-card validation request contain the account number, amount, and a special system code. The service uses the system code to determine in which internal database to find the account. The special system code is exposed through the interface, and it exposes information about the internal structure of the service.

There is no functional reason to expose the system code outside the service, because the service should identify the database itself based on the functional data passed in to it. This information is necessary strictly for implementation. Service maintainability is severely affected when designers implement designs such as this. If the internal structure of the service changes, clients of this service are likely to require changes also. If a third internal system is added, for example, clients will have to be updated, even though the interface contract has not changed. This design is generally not consistent with the principles of information hiding and modular design.

The principle of separating the service's interface from its implementation is relevant to the topic of modular software design. It is often thought that service-oriented architecture enforcesthis principle, which is not strictly true. Service-oriented architecture promotes the idea of separation, but as the previous example illustrates, implementation details can pollute a service's interface.

These techniques and concepts help create modular services. Services also stress interoperability, or the ability of different types of systems to use a service.

Interoperability

Service-oriented architecture stresses interoperability, the ability of systems using different platforms and languages to communicate with each other. Each service provides an interface that can be invoked through a connector type. An interoperable connector consists of a protocol and a data format that each of the *potential* clients of the service understands. Interoperability is achieved by supporting the protocol and data formats of the service's current and potential clients.

Techniques for supporting standard protocol and data formats consist of mapping each platform's characteristics and language to a mediating specification. The mediating specification maps between the formats of the interoperable

data format to the platform-specific data formats. Sometimes this requires mapping character sets such as ASCII to EBCDIC as well as mapping data types. For instance, Web services is a mediating specification for communicating between systems. JAX-RPC and JAXM map Java data types to SOAP. Other platforms that support Web services mediate between Web service specifications and their own internal specifications for character sets and data types.

Loose Coupling

Coupling refers to the number of dependencies between modules. There are two types of coupling: loose and tight. Loosely coupled modules have a few well-known dependencies. Tightly coupled modules have many unknown dependencies. Every software architecture strives to achieve loose coupling between modules. Service-oriented architecture promotes loose coupling between service consumers and service providers and the idea of a few well-known dependencies between consumers and providers.

A system's degree of coupling directly affects its modifiability. The more tightly coupled a system is, the more a change in a service will require changes in service consumers. Coupling is increased when service consumers require a large amount of information about the service provider to use the service. In other words, if a service consumer knows the location and detailed data format for a service provider, the consumer and provider are more tightly coupled. If the consumer of the service does not need detailed knowledge of the service before invoking it, the consumer and provider are more loosely coupled.

SOA accomplishes loose coupling through the use of contracts and bindings. A consumer asks a third-party registry for information about the type of service it wishes to use. The registry returns all the services it has available that match the consumer's criteria. The consumer chooses which service to use, binds to it over a transport, and executes the method on it, based on the description of the service provided by the registry. The consumer does not depend directly on the service's implementation but only on the contract the service supports. Since a service may be both a consumer and a provider of some services, the dependency on only the contract enforces the notion of loose coupling in service-oriented architecture.

Although coupling between service consumers and service producers is loose, implementation of the service can be tightly coupled with implementation of other services. For instance, if a set of services shares a framework, a database, or otherwise has information about each other's implementation, they may be tightly coupled. In many instances, coupling cannot be avoided, and it sometimes contradicts the goal of code reusability.

Network-Addressable Interface

The role of the network is central to the concept of SOA. A service must have a network-addressable interface. A consumer on a network must be able to invoke a service across the network. The network allows services to be reused by any consumer at any time. The ability for an application to assemble a set of reusable services on different machines is possible only if the services support a network interface. The network also allows the service to be location–independent, meaning that its physical location is irrelevant.

It is possible to access a service through a local interface and not through the network, but only if both the consumer and service provider are on the same machine. This is done mainly to enhance performance. Although a service may be configured for access from a consumer on the same machine, the service must also simultaneously support a request from across the network.

Because of this requirement, service interface design is focused to a large extent on performance. In a pure object-based system design, data and behavior are encapsulated into objects. This design works well for objects in the same machine. However, when those objects are distributed across a network, performance degrades quickly because of the "chatter" that occurs between fine-grained objects. Because we can assume that services will be distributed, it is possible to design service interfaces to be more coarse-grained and, as a result, enhance network performance.

Coarse-Grained Interfaces

The concept of granularity applies to services in two ways. First, it is applied to the scope of the domain the entire service implements. Second, it is applied to the scope of the domain that each method within the interface implements.

The levels of granularity are relative to each other. For instance, if a service implements all the functions of a banking system, then we consider it coarse-grained. If it supports just credit-card validation, we consider it fine-grained. In addition, if a method for inquiring about a customer returns all customer information, including address, this method would be coarser-grained than a method that does not return the customer's address.

The appropriate level of granularity for a service and its methods is relatively coarse. A service generally supports a single distinct business concept or process. It contains software that implements the business concept so that it can be reused in multiple large, distributed systems.

Before components and services, distributed systems were centered on the idea of distributed objects (Object Management Group 2002). Distributed object-

based systems consist of many fine-grained networked objects communicating with each other across a network. Each object has dependencies with many other objects in the system. Since accessing an object requires a network hop and thus does not perform well, the design principles for distributed object-based systems quickly moved toward coarser-grained interfaces.

Figure 2.6 illustrates a distributed object-based system. The number of connections between objects is great. As system size and complexity grows, these dependencies become difficult to manage. Performance suffers because of the large number of network hops. Maintainability also suffers because of the large number of dependencies between objects. Since any object can connect to and use any other object, it becomes difficult to know what dependencies exist. When the developer makes a necessary change to an interface, it might affect a large number of other distributed objects. The developer must then compile and deploy together all the changed objects and the objects that depend on them.

A service-based system controls the network access to the objects within the service through a set of coarse-grained interfaces, as shown in Figure 2.7. A service may still be implemented as a set of fine-grained objects, but the objects themselves are not accessible over a network connection. A service implemented as objects has one or more coarse-grained objects that act as distributed façades. These objects are accessible over the network and provide access to the internal object state from external consumers of the service. However, objects internal to the service communicate directly with each other within a single machine, not

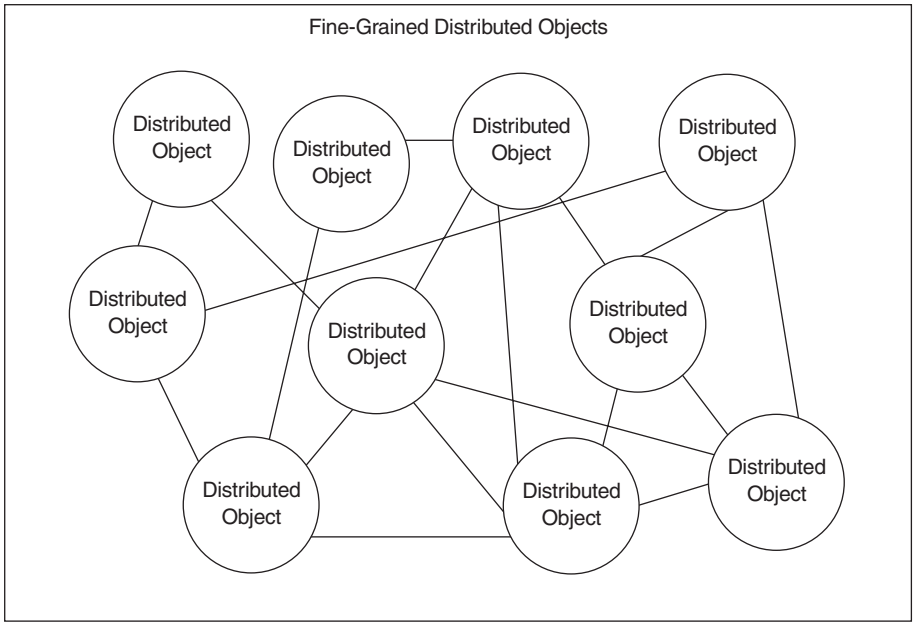

Figure 2.6 Fine-grained distributed objects.

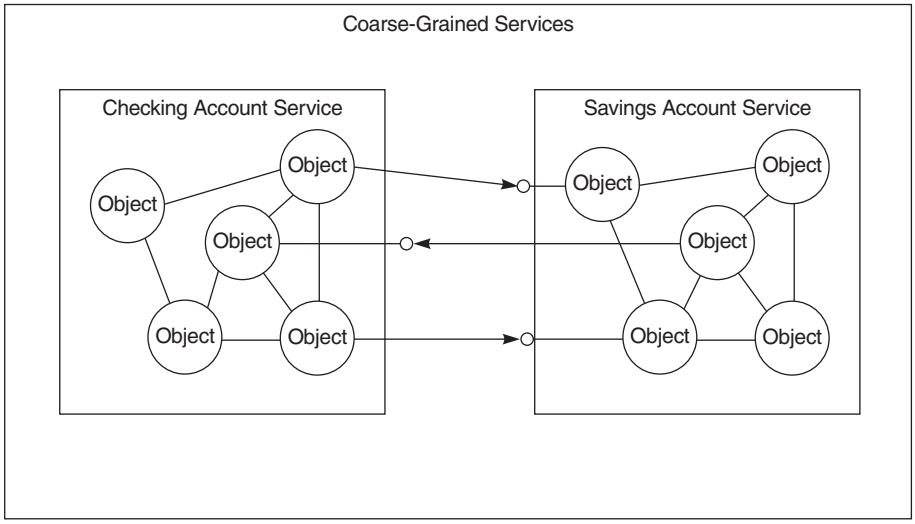

Figure 2.7 Coarse-grained services.

across a network connection. All service interfaces are relatively coarse–grained compared with distributed object interfaces. However, within the range of coarse, there are options. It is important to understand these options for interface design.

One of the benefits of service-oriented architecture is service composition. Developers compose services into applications. Unfortunately, one cannot always know how services will be used in these applications. It is especially difficult to predict how services will be used in future applications. This uncertainty is one of the greatest challenges for service designers, who typically attempt to anticipate future applications when determining the structure of an interface. Because services are executed across a network, it is especially important for interfaces to be correct. If they are not, service consumers will either receive more data than they need or will have to make multiple trips to the service to retrieve all the data they need.

While services in general support coarser-grained interfaces than distributed object-based systems and component-based systems do, the range of coarse still contains degrees of granularity, as Figure 2.8 shows. Within the range of granularity expected for services, designers still need to decide interface coarseness.

As explained previously, the service itself can be coarse-grained or fine-grained. This refers to how much functionality the service covers. Let's assume developers need to create an application for manipulating both a checking account and a savings account. Developers have two choices when creating a service to support this function. They could create a coarse-grained service called *BankAccountService* that manipulates both checking and savings accounts, or they could create two fine-grained services–a *SavingsAccountService* and a *CheckingAccountService*.

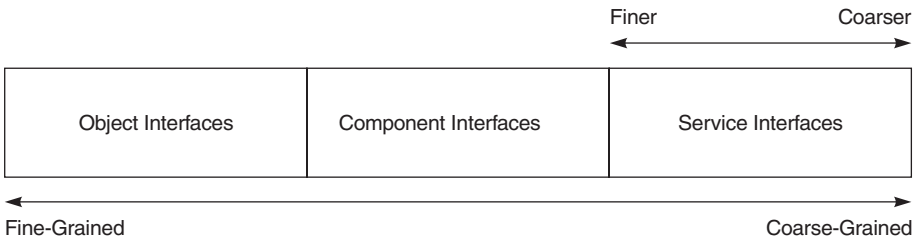

Figure 2.8 Degrees of granularity.

Because *BankAccountService* supports the functionality of both checking and savings, it is coarser-grained.

Granularity also applies to the way developers implement service methods. Suppose *BankAccountService* contains a method called *GetAccountHolder.* A coarse-grained implementation of this function would return the account holder's name and address. A fine-grained version would return just the name. A separate method, called *GetAccountHoldersAddress,* would return the address. A service method that returns more data is a coarse-grained method. A service method that returns less, more specific, data is a fine-grained method. Sometimes service consumers need both fine-grained and coarse-grained methods for a similar function. This is the concept of multi-grained services.

Multi-Grained Services

Because services will be used in ways the designers cannot fully anticipate when designing them, the decision about granularity does not have to be absolute. Services do not have to be coarse-grained or fine-grained; they can be coarse-grained and fine-grained, or *multi-grained* (Stevens 2002). In other words, *BankAccountService, SavingsAccountService,* and *CheckingAccountService* can exist simultaneously, as in Figure 2.9. If service consumers needs access only to a customer's savings account, they should use *SavingsAccountService.* There is no need for them to know anything about checking accounts. If, on the other hand, they need to know about both checking accounts and savings accounts, they should use *BankAccountService.*

Why is it necessary to create a composite *BankAccountService* if there is already a *SavingsAccountService* and *CheckingAccountService*? The reason is that services should be as easy as possible to use, and they should meet the expectations of the consumers that use them. It is logical that a consumer would more often than not want access to both checking and savings accounts. Implementing both interfaces is best, because it provides all service consumers with the interfaces that best suit their needs.

Service designers create multi-grained service interfaces by first creating fine-grained services and then wrapping them in coarse-grained façades. It is also pos-

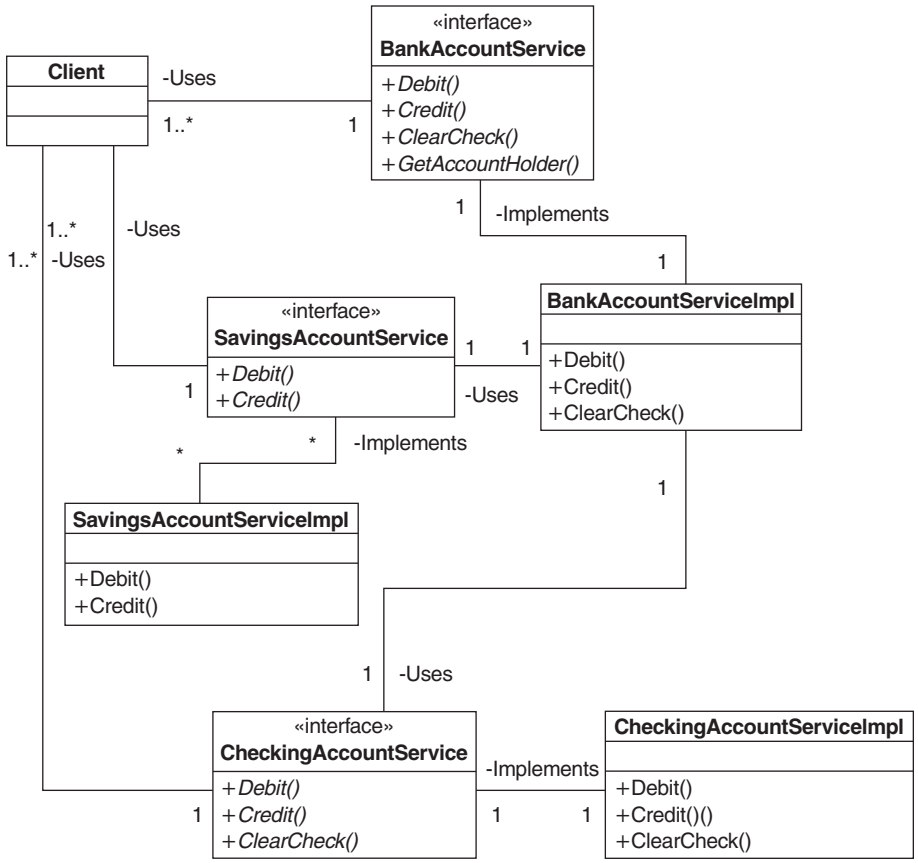

Figure 2.9 Multi-grained services.

sible to create fine-grained façades that access coarse-grained services. However, it is better to create finer-grained base services, because developers will have more flexibility when deploying them. It is difficult to break up a larger service and deploy it onto multiple machines. However, it is easy to deploy a large number of small-grained services to multiple machines.

The granularity of the service is a crucial design decision. If it is incorrectly predicted, consumers will have access to more functionality than they need. This can be a problem for security at the service level. It might not be possible to restrict a consumer from some methods and not others, only to the entire service. If this is the case, the entire service might have to be opened up to consumers. Developers can do this if they design services at the appropriate level of granularity.

The service interfaces constitute an established contract between the services and the clients. One of the tradeoffs of creating multiple interfaces is that each interface is essentially a published contract. Additional interfaces make managing these contracts between clients and services more difficult, because a change

to a functional requirement will affect multiple interfaces. Although it is important to provide the best possible interfaces to consumers, it is also important not to substantially compromise the service's maintainability.

Multi-Grained Methods

The granularity of the methods within a service is of equal or greater importance than the granularity of the service itself. Using the previous bank account example, consider the retrieval of account holder information from the bank account service. There are several ways to implement this interface:

- A method in *BankAccountService* called *GetAccountHolder* that returns only account-holder information and not the address
- Two methods in *BankAccountService,* called *GetAccountHolder* and *GetAccountHolderAddress; GetAccountHolder* would not return address information
- A method in *BankAccountService* called *GetAccountHolder* that could return both the name and address of the account holder
- A method in *BankAccountService* called *GetAccountHolder* that could have a switch that tells the service whether to return address information as well as account-holder information
- A method in *BankAccountService* called *GetAccountHolder* that could accept a list of attributes it wants the service to return; the consumer can choose to get the address by adding the address attributes to the attribute list it passes in to the service

Let's examine these options and their consequences. As Figure 2.10 shows, *BankAccountService* returns just account-holder information.

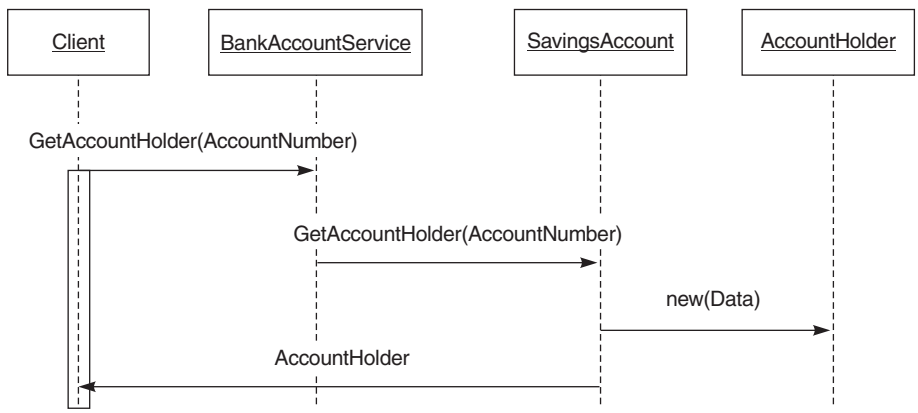

Figure 2.10 A method that returns only account-holder information.

This scenario works well if the consumer needs only account-holder information. But a consumer who needs address information as well is out of luck. The address information could be retrieved from the service by adding a *GetAccountHolderAddress* method, as illustrated in Figure 2.11.

This solves the problem of retrieving address information, but if most of the consumers need address information, more trips are necessary. Having the *GetAccountHolder* method return both account-holder information and address information in one call would improve performance and reduce the work necessary for the consumer to assemble the two results.

Figure 2.12 illustrates this scenario.

This solution works well for consumers who always retrieve address information, but if they almost never need this information, more data than necessary will travel across the network. It will also take longer for service consumers to extract the account-holder data they need from the larger message.

Another solution is to pass in an argument that directs the service whether to return address information. A *BankAccountService* would have only one *GetAccountHolder* method. The developer would add an additional argument to the method, to instruct the service whether to return address information as well.

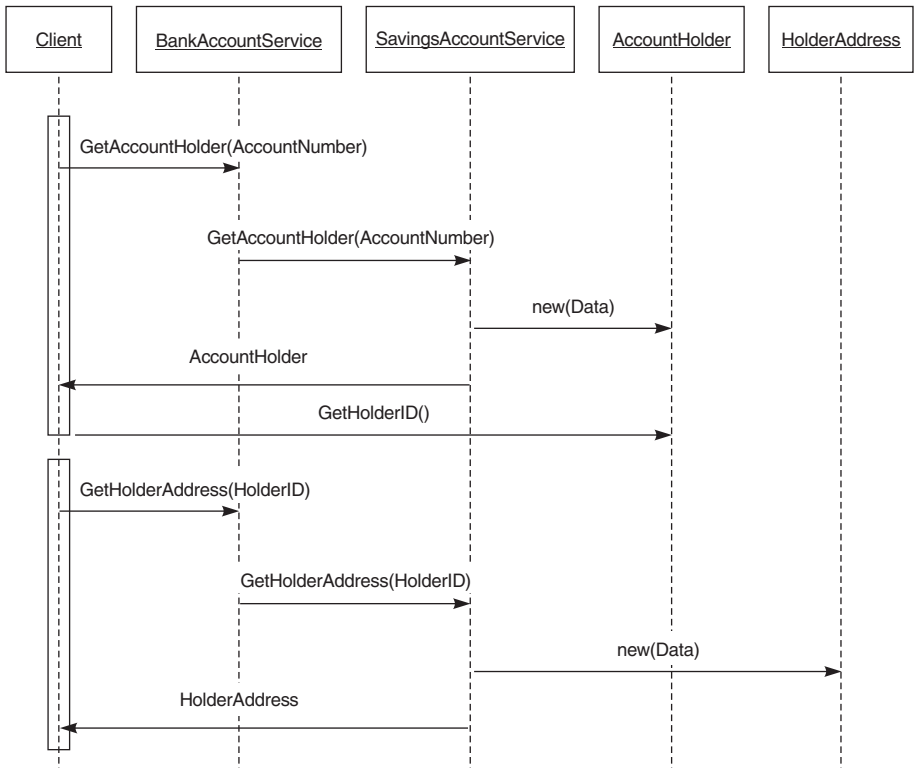

Figure 2.11 A method that returns both the account-holder's information and address.

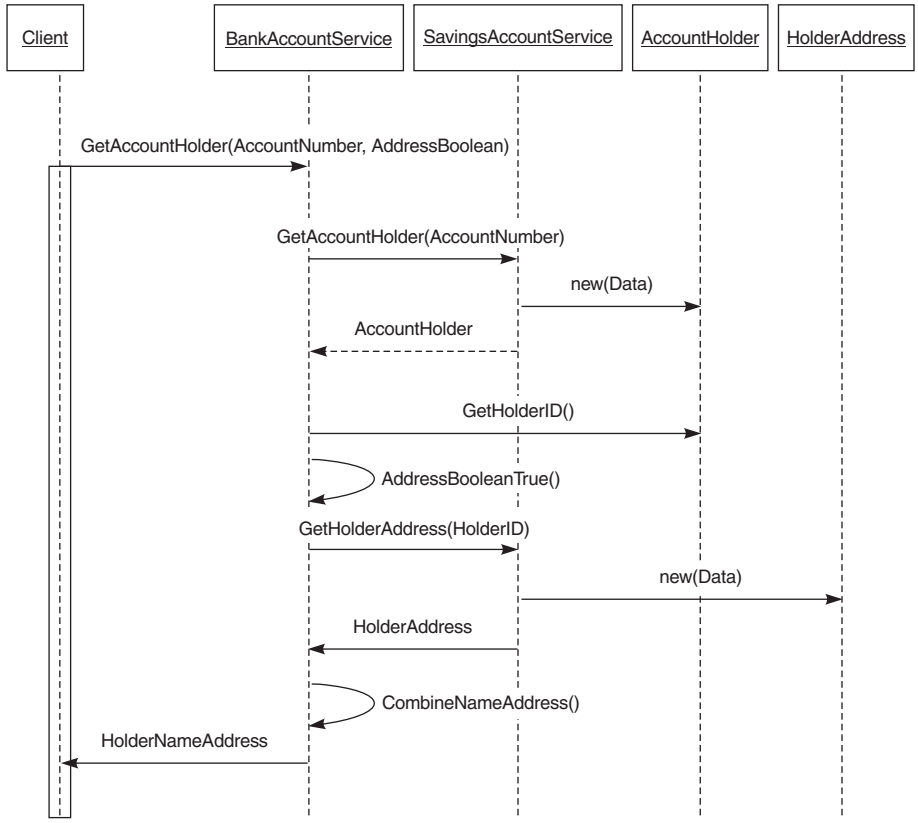

Figure 2.12 A method that returns either the account-holder's information or address.

Consumers who need only account-holder information could pass in the proper switch to retrieve it. Users who need address information as well could pass in the proper switch to retrieve both.

But what if consumers need only zip codes for all account holders? They would have to retrieve both account-holder information and address information and extract zip codes from a very large message. What if consumers pass in the list of attributes in which they're interested?

This sophisticated alternative implements an interface that accepts a list of attributes to return to the consumer. Instead of sending the account number and an address indicator, consumers submit a list of all of the attributes to return. The list may contain just first and last names or may include all or portions of the address data, such as city and street address. The service would interpret this list and construct the response to consumers to include only the data requested. This solution minimizes both the number of trips consumers make to the service and the amount of data that must travel the network for each request. Figure 2.13 illustrates this option.

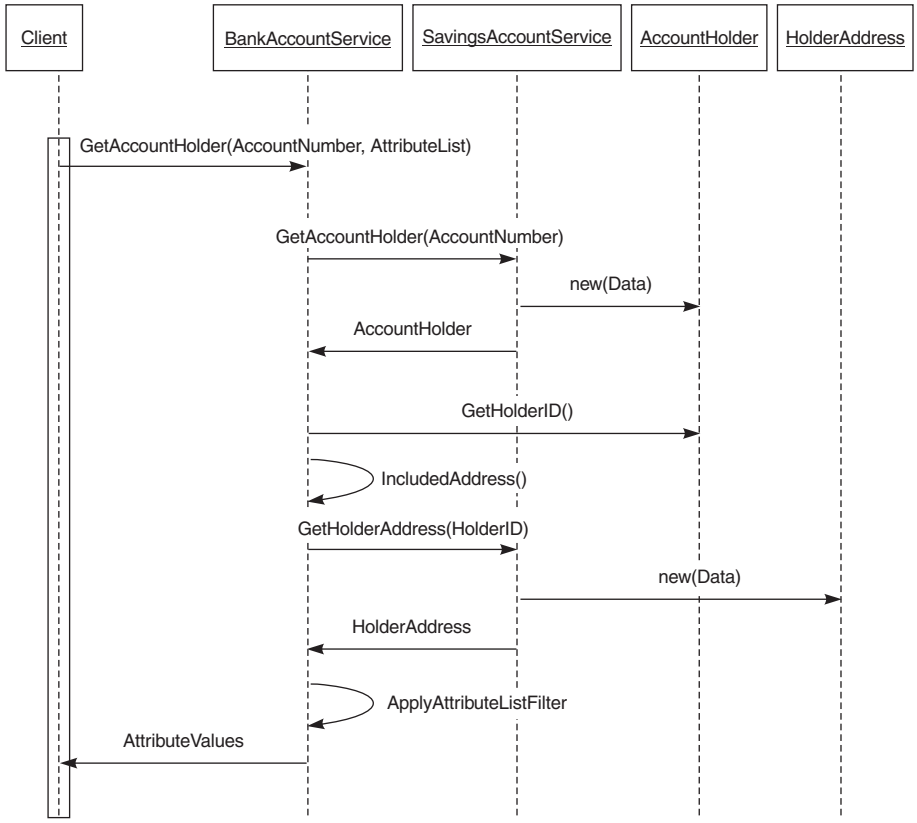

Figure 2.13 A method that returns just the attributes requested.

This approach has two downsides. The first is that the request message will be larger than any of the previous solutions, because the consumer must send the request data as well as the data map on each request. If all service consumers need the exact same data from the service, this solution would perform worse than the previously discussed alternatives.

Second, this solution is also more complex to implement for service developers, and service consumers might find the interface more difficult to understand and use. To alleviate this problem, a service proxy could wrap the complexities of the service interface and provide a simple interface for consumers. A consumer would use multiple distinct and simple service methods on the proxy. The methods map to the way the consumer wants to use the service. The proxy would internally map these multiple methods into a single service-request interface format that accepts a map of data to return. The advantage of this technique is that it allows the service to support any granularity, while providing specific granularities to consumers based on their domain understanding.

If these implementations are not possible, it is always better to return more data, to minimize network round trips, because future clients are likely to need

the data. It is also possible to implement several of these options, to solve the needs of multiple consumers. However, this increases the effort to maintain the service and also detracts somewhat from the service's modular understandability.

A service's ability to have multi-grained methods that return the appropriate amount of data is important to reduce network traffic. Extra network traffic is due either to excessive unnecessary data or to a large number of requests to get data.

Granularity is a difficult problem to reconcile when designing service interfaces. It is important to understand the options and implement the most appropriate interface. In the past, arguments surrounding service interfaces have focused mainly on determining the right granularity. Services actually require the designer to find the right granularities for service consumers.

Location Transparency

Location transparency is a key characteristic of service-oriented architecture. Consumers of a service do not know a service's location until they locate it in the registry. The lookup and dynamic binding to a service at runtime allows the service implementation to move from location to location without the client's knowledge. The ability to move services improves service availability and performance. By employing a load balancer that forwards requests to multiple service instances without the service client's knowledge, we can achieve greater availability and performance.

As mentioned earlier, a central design principle in object-oriented systems is separation of implementation from interface. This means that an object's interface and its implementation may vary independently. The primary motivation for this principle is to control dependencies between objects by enforcing the interface contract as their only means of interaction.

Service-oriented architecture takes this principle one step further, by reducing the consumer's dependency on the contract itself. This reduced dependency through the use of dynamic binding also has the effect of making the service's location irrelevant. Because the service consumer has no direct dependency on the service contract, the contract's implementation can move from location to location.

Composability

A service's composability is related to its modular structure. Modular structure enables services to be assembled into applications the developer had no notion of when designing the service. Using preexisting, tested services greatly enhances a system's quality and improves its return on investment because of the ease of reuse.

A service may be composed in three ways: application composition, service federations, and service orchestration.

An *application* is typically an assembly of services, components, and application logic that binds these functions together for a specific purpose. *Service federations* are collections of services managed together in a larger service domain. For example, a checking account service, savings account service, and customer service may be composed into a larger banking-account service. *Service orchestration* is the execution of a single transaction that impacts one or more services in an organization. It is sometimes called a *business process*. It consists of multiple steps, each of which is a service invocation. If any of the service invocations fails, the entire transaction should be rolled back to the state that existed before execution of the transaction.

For a service to be composed into a transactional application, federation, or orchestration, the service methods themselves should be *subtransactional*. That is, they must not perform data commits themselves. The orchestration of the transaction is performed by a third-party entity that manages all the steps. It detects when a service method fails and asks all the services that have already executed to roll back to the state that existed before the request. If the services have already committed the state of their data, it is more difficult for the method to be composed into a larger transactional context.

If the service cannot be subtransactional, it should be *undoable*. Especially when dealing with legacy systems, it is sometimes impossible to execute a function within the context of a transaction. For instance, consider an older system that manages checking accounts. The service is a façade for the legacy application. When the service receives a request to deposit money into a checking account, it puts the request into a queue. The legacy system reads the request from the queue and executes it. It is difficult to make this request subtransactional, but it can be undoable. If the deposit transaction is composed into a larger transaction and another step in the larger transaction fails, the checking account deposit transaction can be undone by withdrawing the same amount from the checking account. While to a developer this makes perfect sense, a customer would probably see the deposit and withdrawal transactions on his or her statement at the end of the month, so it should be used with care.

Self-Healing

With the size and complexity of modern distributed applications, a system's ability to recover from error is becoming more important. A *self-healing* system is one that has the ability to recover from errors without human intervention during execution.

Reliability measures how well a system performs in the presence of disturbances. In service-oriented architecture, services will be up and down from time to time. This is especially true for applications assembled from services from multiple organizations across the Internet. The extent to which a system is self-healing depends on several factors.

Reliability depends on the hardware's ability to recover from failure. The network must also allow for the dynamic connection to different systems at runtime. Modern Internet networking protocols inherently provide this capability.

Another aspect of self-healing is the architecture from which the application is built. Architecture that supports dynamic binding and execution of components at runtime will be more self-healing than one that does not. For instance, service-based systems are self-healing to a greater degree than previous architectures, because services are bound to and executed dynamically at runtime. If a service fails, the client may find, bind, and execute a different service, as long as the other service provides the same or a similar interface contract.

In addition, because service-based systems require that the interface be separate from the implementation, implementations may vary. For instance, a service implementation may run in a clustered environment. If a single service implementation fails, another instance can complete the transaction for the client without the client's knowledge. This capability is possible only if the client interacts with the services interface and not its implementation. This property is fundamental to all service-oriented architectures.

> Summary

Software architecture has been emerging as a discipline over the last decade (Garlan 2000). A system's software architecture describes its coarse-grained structures and its properties at a high level. As long as the technology supports those structures and properties, the technology can be considered to implement the architecture. For instance, Jini is a technology that supports service-oriented architecture, because it supports the properties of SOA.

It is important to apply the concepts of software architecture to any new technology to take full advantage of it. Service-oriented architecture is implemented by technologies other than Web services, but the term and concepts have gained popularity recently because of Web services. For instance, the computer industry has used the term *service* for about two decades to describe various platforms.

Some of the characteristics of service-oriented architecture are supported better by certain technologies than by others. For instance, CORBA and Jini are

less interoperable than Web services, but Jini excels in other properties (though this is arguable), such as discovery.

Interface design is perhaps the most difficult part of designing services in service-oriented architecture. The modularization techniques practiced for decades still apply to services. Service design is even more difficult, because the domain a service supports is not limited to a single application. Therefore, it is best to perform modularization starting with a conceptual model of the business rather than of a single application. If the interface design is done well, the services are more likely to be reusable in other applications, and organizations will realize a higher return on their investment.

Web services are refocusing organizations on the concepts of service-oriented architecture. Although highly reusable, loosely coupled architectures have been a goal for many organizations. Web services are fostering interest in and providing the technology to implement service-oriented architectures that enable them to realize their vision.

> References

Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice.* Addison-Wesley, 1997.

Bieber, G., and Carpenter, J. *Introduction to Service-Oriented Programming (Rev 2.1). www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf,* accessed October 2002.

Fowler, M. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

Garlan, D. *Software Architecture: A Roadmap.* ACM Press, 2000.

Herzum, P. *Web Services and Service-Oriented Architectures.* Executive Report, vol. 4, no. 10. Cutter Distributed Enterprise Architecture Advisory Service, 2002.

Meyer, B. *Object Oriented Software Construction.* Prentice Hall, 1997, pp. 39–48.

Object Management Group (OMG). *CORBA Basics. www.omg.org/gettingstarted/corbafaq.htm,* accessed October 2002.

Parnas, D., and Clements, P. *The Modular Structure of Complex Systems.* IEEE, 1984.

Potts, M. *Find Bind and Execute: Requirements for Web Service Lookup and Discovery. www.talkingblocks.com/resources.htm#,* accessed January 2003.

Stevens, M. *Service-Oriented Architecture Introduction, Part 2.* Developer.com, *http://softwaredev.earthweb.com/msnet/article/0,,10527_1014371,00.html,* accessed October 2002.

——. *Multi-grained Services.* Developer.com, *http://softwaredev.earthweb.com/java/sdjjavaee/article/0,,12396_1142661,00.html,* accessed October 2002.

Sun Microsystems. *Jini Network Technology, www.sun.com/jini.*

——. *Jini Technology Architectural Overview, http://wwws.sun.com/software/jini/whitepapers/architecture.html,* accessed October 2002.

——. *Jini Technology Core Specification: LE-Distributed Leasing. http://wwws.sun.com/software/jini/specs/jini1.2html/lease-spec.html,* accessed October 2002.