Service-Oriented Architecture

Concepts, Technology, and Design

Thomas Erl

PRENTICE HALL PROFESSIONAL TECHNICAL REFERENCE
UPPER SADDLE RIVER, NJ • BOSTON • INDIANAPOLIS • SAN FRANCISCO
NEW YORK • TORONTO • MONTREAL • LONDON • MUNICH • PARIS • MADRID
CAPETOWN • SYDNEY • TOKYO • SINGAPORE • MEXICO CITY

Library of Congress Number: 2005925019

ISBN 0-13-185858-0
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, July 2005

Contents

Chapter 7

Part III

SOA and Service-Orientation

So far the focus has been on the aspects of SOA as a whole. We have discussed the numerous extensions available to SOA as well as its fundamental concepts. We now turn our attention to the underlying paradigm primarily responsible for defining SOA and distinguishing it as an architectural model.

The principles and concepts covered in the next two chapters discuss the spectrum of service-orientation in detail. This establishes necessary theory that applies to the rudimentary components of primitive SOA, but also provides concepts that can be propagated and leveraged throughout service-oriented environments. For example, topics covered in these chapters form the basis for the service modeling and design processes provided in the subsequent *Building SOA* parts of this book.

Chapter 8

Principles of Service-Orientation

Before we can begin building a service-oriented solution, we need to understand what makes a service suitable for SOA. In other words, how can we build Web services that are truly service-oriented?

The answer lies in service-orientation. This approach to modeling business automation logic has resulted in a set of commonly accepted principles applied to each unit of logic that constitutes a service within an SOA. It is through the application of these principles that the primitive components of an SOA (services, descriptions, messages) are shaped in support of service-orientation.

This chapter begins with a look at how service-orientation applies to the enterprise as a whole and then discusses individual principles in-depth.

In Plain English sections

A knowledge of the principles of service-orientation is perhaps even more important than concepts covered in past chapters. They are core to the design of services regardless of what underlying technology is used to implement them. Therefore, our *In Plain English* sections return to supplement the descriptions of individual principles.

How case studies are used: As you might recall from the case study background information provided in Chapter 2, one of RailCo's business goals was to improve their existing automation processes by moving toward SOA.

In this chapter we examine the services built so far as part of RailCo's technical environment and discuss how they comply to or diverge from individual principles of service-orientation. Existing TLS services that already possess service-orientation characteristics are used for comparison purposes.

8.1 Service-orientation and the enterprise

The collective logic that defines and drives an enterprise is an ever-evolving entity constantly changing in response to external and internal influences. From an IT perspective, this *enterprise logic* can be divided into two important halves: business logic and application logic (Figure 8.1).

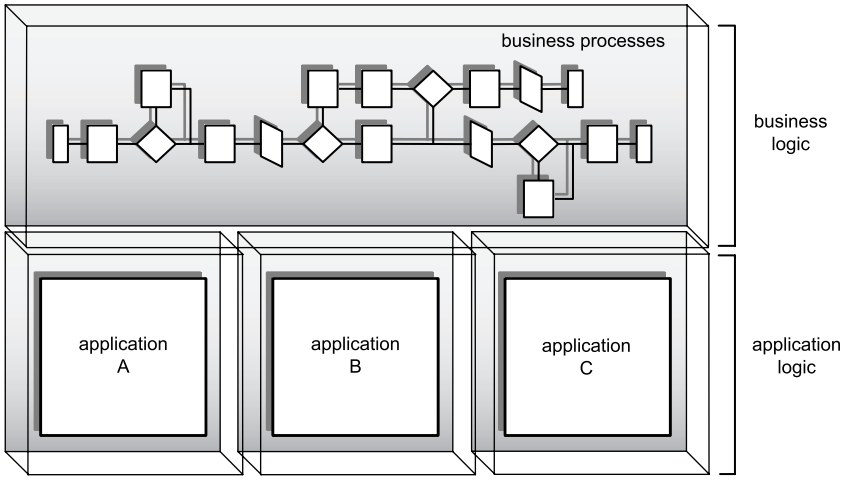

Figure 8.1
The business and application logic domains.

Each exists in a world of its own, and each represents a necessary part of contemporary organization structure. Business logic is a documented implementation of the business requirements that originate from an enterprise's business areas. Business logic is generally structured into processes that express these requirements, along with any associated constraints, dependencies, and outside influences.

Application logic is an automated implementation of business logic organized into various technology solutions. Application logic expresses business process workflows through purchased or custom-developed systems within the confines of an organization's IT infrastructure, security constraints, technical capabilities, and vendor dependencies.

Service-orientation applies to enterprise logic. It introduces new concepts that augment the manner in which this logic is represented, viewed, modeled, and shared. While the principles behind service-orientation exist in a vacuous realm of abstraction and theory, they are a necessary counterpart to the real world environments that require their guidance and structure.

The concepts introduced by service-orientation are realized through the introduction of services. Let's have a look at where services are located within the overall structure of an automated organization. As Figure 8.2 illustrates, services establish a high form of abstraction wedged between traditional business and application layers. When positioned here, services can encapsulate physical application logic as well as business process logic.

Services modularize the enterprise, forming standalone units of logic that exist within a common connectivity layer. Services can be layered so that parent services can encapsulate child services. This allows for the service layer to consist of multiple layers of abstraction (as explained later in Chapter 9).

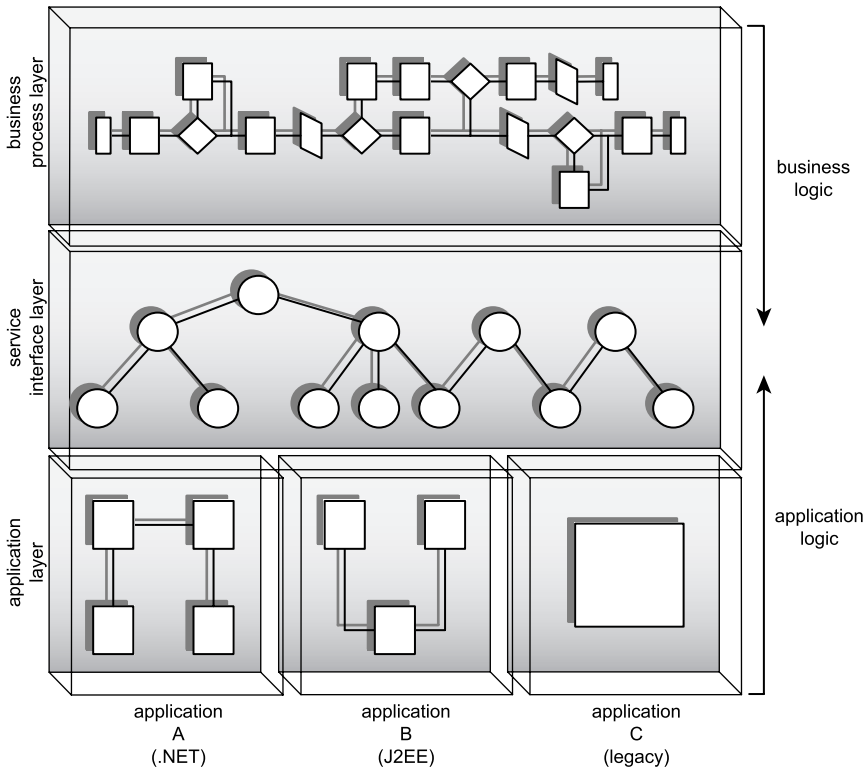

Figure 8.2
The service interface layer positioned between enterprise layers that promote application and business logic.

In Figure 8.2 we display a fragmented application layer, where individual applications are confined to the boundaries that represent their respective proprietary platform environments. Though we show services as existing in a single, continuous layer, this only illustrates the open connectivity that exists among service interfaces. Freed from proprietary ties, services can communicate via open protocols.

On a physical level, services are developed and deployed in proprietary environments, wherein they are individually responsible for the encapsulation of specific application

logic. Figure 8.3 shows how individual services, represented as service interfaces within the service interface layer, represent application logic originating from different platforms.

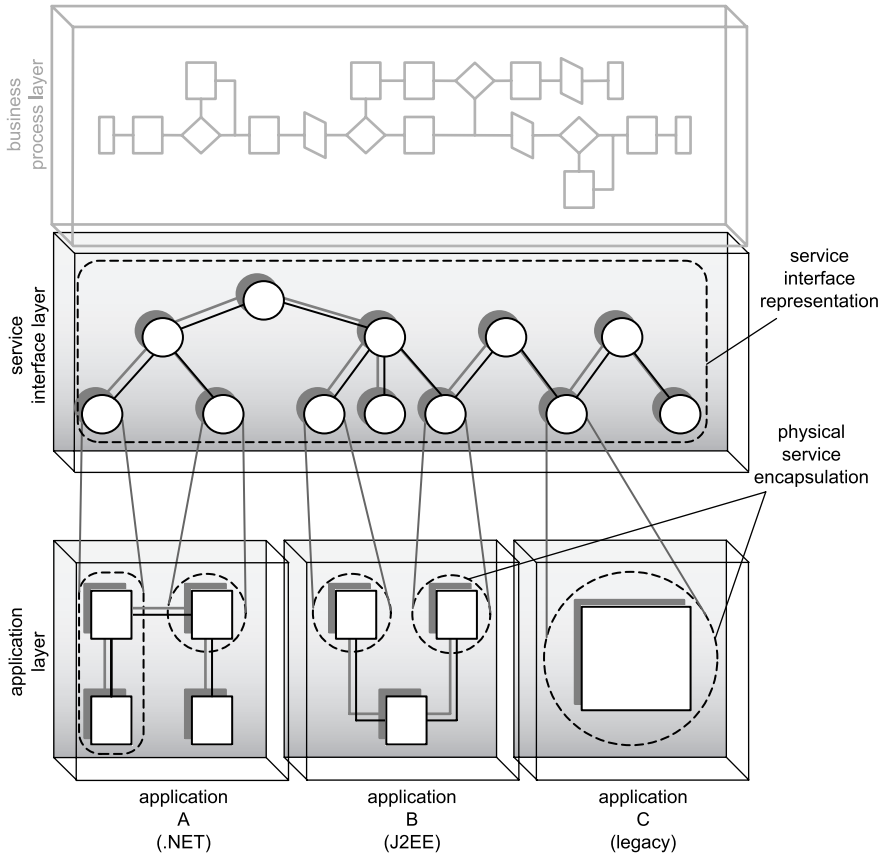

Figure 8.3
The service interface layer abstracts connectivity from service deployment environments.

SUMMARY OF KEY POINTS

- Enterprise logic can be divided into two domains: business logic and application logic. Service-oriented principles can be applied to both.
- The service interface layer positions services to represent business logic and abstract application logic.

8.2 Anatomy of a service-oriented architecture

Chapter 5 established the components of the basic (first-generation) Web services framework. This framework can be applied to implement services in just about any environment. For example, services can be appended to traditional distributed applications or used as wrappers to expose legacy system logic. However, neither of these environments resembles a "real" service-oriented architecture.

To best understand what constitutes a true SOA, we need to abstract the key components of the Web services framework and study their relationships more closely. To accomplish this, we begin by revisiting these familiar components and altering our perspective of them. First, we re-label them to reflect terminology more associated with service-orientation. Then we position them into a logical view wherein we subsequently re-examine our components within the context of SOA.

8.2.1 Logical components of the Web services framework

The communications framework established by Web services brings us the foundation technology for what we've classified as contemporary SOA. Because we covered this framework in Chapter 5, we will use it as a reference point for our discussion of service-orientation.

Let's first recap some Web services fundamentals within a logical modeling context. As shown in Figure 8.4, each Web service contains one or more operations. Note that this diagram introduces a new symbol to represent operations separately from the service.

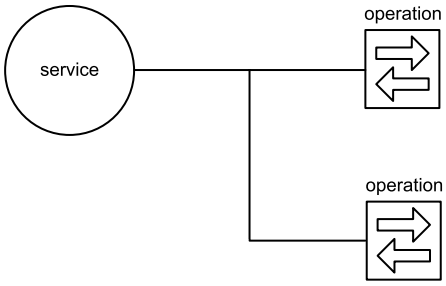

Figure 8.4
A Web service sporting two operations.

Each operation governs the processing of a specific function the Web service is capable of performing. The processing consists of sending and receiving SOAP messages, as shown in Figure 8.5.

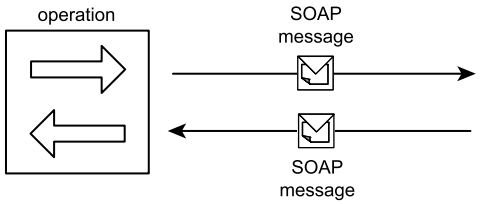

Figure 8.5
An operation processing outgoing and incoming SOAP messages.

By composing these parts, Web services form an activity through which they can collectively automate a task (Figure 8.6).

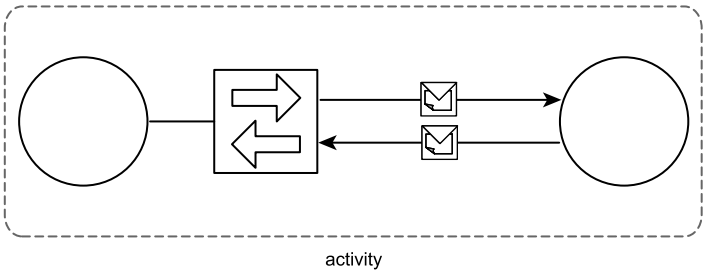

Figure 8.6
A basic communications scenario between Web services.

8.2.2 Logical components of automation logic

The Web services framework provides us not only with a technology base for enabling connectivity, it also establishes a modularized perspective of how automation logic, as a whole, can be comprised of independent units. To illustrate the inherent modularity of Web services, let's abstract the following fundamental parts of the framework:

- SOAP messages
- Web service operations
- Web services
- activities

The latter three items represent units of logic that perform work and communicate using SOAP messages. To better illustrate this in a service-oriented perspective, let's replace these terms with new ones, as follows:

- messages
- operations
- services
- processes (and process instances)

You'll notice that these are quite similar to the terms we used before. The one exception is the use of "process" instead of "activity." In later chapters we actually use the word "activity" in different contexts when modeling service-oriented business processes.

For now, the one discrepancy to be aware of is that while a Web service activity is typically used to represent the temporary interaction of a group of Web services, a process is a static definition of interaction logic. An activity is best compared to an instance of a process wherein a group of services follow a particular path through the process logic to complete a task.

Regardless, for the purposes of our discussion of service-orientation, we'll continue with our look at how automation logic is comprised of the four identified parts. We can further qualify these parts by relating each to different sized units of logic, as follows:

- messages = units of communication
- operations = units of work
- services = units of processing logic (collections of units of work)
- processes = units of automation logic (coordinated aggregation of units of work)

Figure 8.7 provides us with a primitive view of how operations and services represent units of logic that can be assembled to comprise a unit of automation logic.

Next, in Figure 8.8, we establish that messages are a suitable means by which all units of processing logic (services) communicate. This illustrates that regardless of the scope of logic a service represents, no actual processing of that logic can be performed without issuing units of communication (in this case, messages).

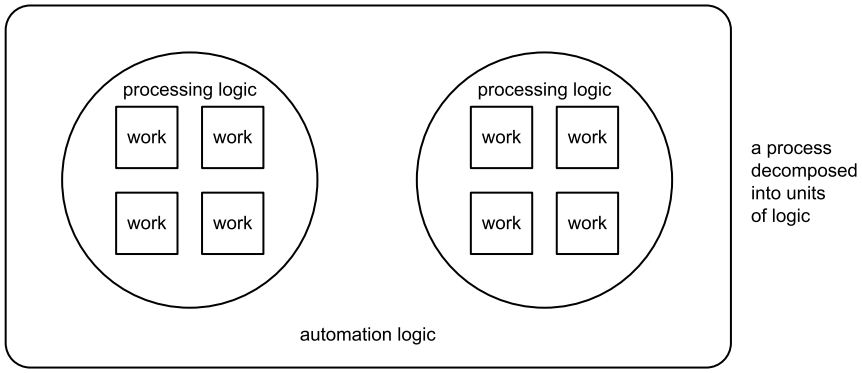

Figure 8.7
A primitive view of how SOA modularizes automation logic into units.

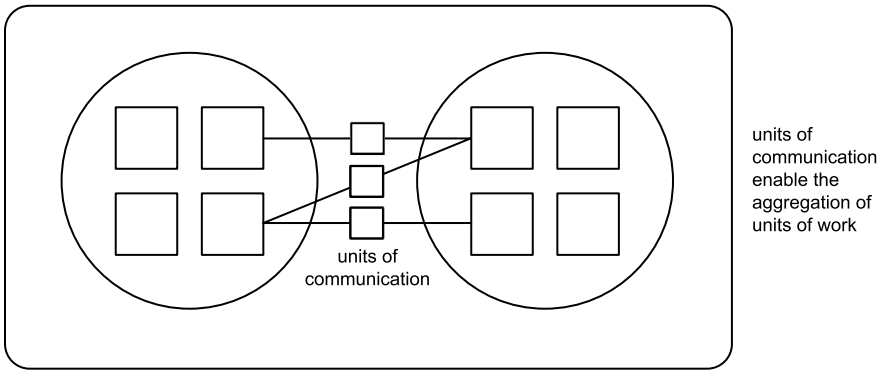

Figure 8.8
A primitive view of how units of communication enable interaction between units of logic.

The purpose of these views is simply to express that processes, services, and operations, on the most fundamental level, provide a flexible means of partitioning and modularizing logic. Regardless of the technology platform used, this remains the most basic concept that underlies service-orientation. In being able to derive this view from the Web services framework, we also have demonstrated the suitability of the Web services platform as a means of implementation for SOA.

8.2.3 Components of an SOA

We'll continue to work with our components of automation logic, but we now broaden our discussion to how the characteristics and behaviors of these components are formed within service-oriented architecture.

Each of the previously defined components establishes a level of enterprise logic abstraction, as follows:

- A message represents the data required to complete some or all parts of a unit of work.
- An operation represents the logic required to process messages in order to complete a unit of work (Figure 8.9).

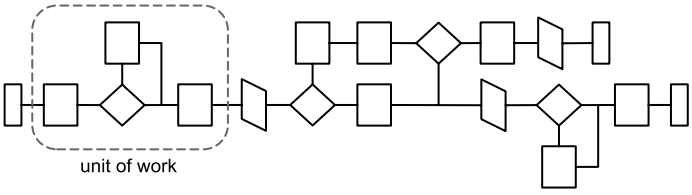

Figure 8.9
The scope of an operation within a process.

- A service represents a logically grouped set of operations capable of performing related units of work.
- A process contains the business rules that determine which service operations are used to complete a unit of automation. In other words, a process represents a large piece of work that requires the completion of smaller units of work (Figure 8.10).

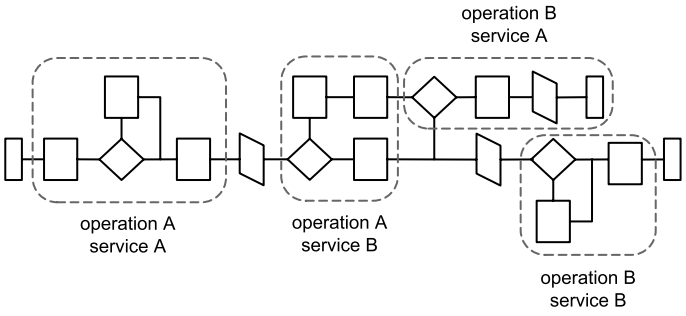

Figure 8.10
Operations belonging to different services representing various parts of process logic.

8.2.4 How components in an SOA inter-relate

Having established the core characteristics of our SOA components, let's now look at how these components are required to relate to each other:

- An operation sends and receives messages to perform work.
- An operation is therefore mostly defined by the messages it processes.
- A service groups a collection of related operations.
- A service is therefore mostly defined by the operations that comprise it.
- A process instance can compose services.
- A process instance is not necessarily defined by its services because it may only require a subset of the functionality offered by the services.
- A process instance invokes a unique series of operations to complete its automation.
- Every process instance is therefore partially defined by the service operations it uses.

Figures 8.11 and 8.12 further illustrate these relationships.

A service-oriented architecture is an environment standardized according to the principles of service-orientation in which a process that uses services (a service-oriented process) can execute. Next, we'll take a closer look at what exactly the principles of service-orientation consist of.

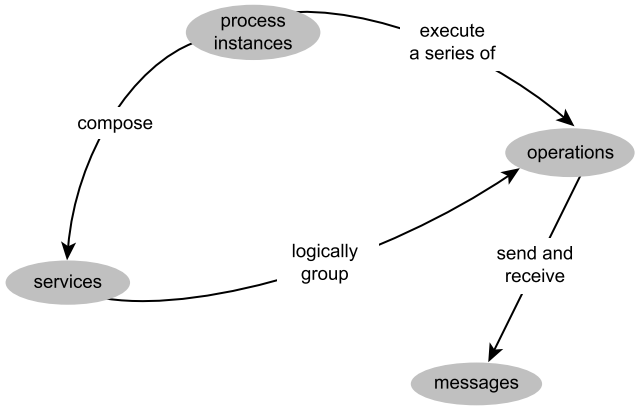

Figure 8.11
How the components of a service-oriented architecture relate.

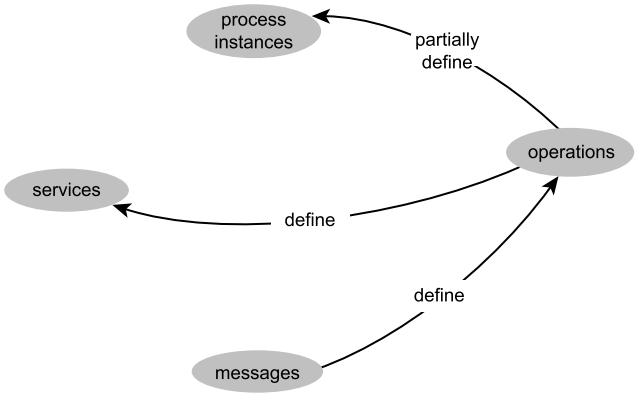

Figure 8.12
How the components of a service-oriented architecture define each other.

SUMMARY OF KEY POINTS

- The logical parts of an SOA can be mapped to corresponding components in the basic Web services framework.
- By viewing a service-oriented solution as a unit of automation logic, we establish that SOA consists of a sophisticated environment that supports a highly modularized separation of logic into differently scoped units.
- SOA further establishes specific characteristics, behaviors, and relationships among these components that provide a predictable environment in support of service-orientation.

8.3 Common principles of service-orientation

In Chapter 3 we established that there is no single definition of SOA. There is also no single governing standards body that defines the principles behind service-orientation. Instead, there are many opinions, originating from public IT organizations to vendors and consulting firms, about what constitutes service-orientation.

Service-orientation is said to have its roots in a software engineering theory known as "separation of concerns." This theory is based on the notion that it is beneficial to break down a large problem into a series of individual concerns. This allows the logic required to solve the problem to be decomposed into a collection of smaller, related pieces. Each piece of logic addresses a specific concern.

This theory has been implemented in different ways with different development platforms. Object-oriented programming and component-based programming approaches, for example, achieve a separation of concerns through the use of objects, classes, and components.

Service-orientation can be viewed as a distinct manner in which to realize a separation of concerns. The principles of service-orientation provide a means of supporting this theory while achieving a foundation paradigm upon which many contemporary SOA characteristics can be built. In fact, if you study these characteristics again, you will notice that several are (directly or indirectly) linked to the separation of concerns theory.

As previously mentioned, there is no official set of service-orientation principles. There are, however, a common set of principles most associated with service-orientation. These are listed below and described further in this section.

- **Services are reusable**—Regardless of whether immediate reuse opportunities exist, services are designed to support potential reuse.
- **Services share a formal contract**—For services to interact, they need not share anything but a formal contract that describes each service and defines the terms of information exchange.
- **Services are loosely coupled**—Services must be designed to interact without the need for tight, cross-service dependencies.
- **Services abstract underlying logic**—The only part of a service that is visible to the outside world is what is exposed via the service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestors.
- **Services are composable**—Services may compose other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.
- **Services are autonomous**—The logic governed by a service resides within an explicit boundary. The service has control within this boundary and is not dependent on other services for it to execute its governance.
- **Services are stateless**—Services should not be required to manage state information, as that can impede their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.

- **Services are discoverable**—Services should allow their descriptions to be discovered and understood by humans and service requestors that may be able to make use of their logic.

Of these eight, autonomy, loose coupling, abstraction, and the need for a formal contract can be considered the core principles that form the baseline foundation for SOA. As explained in the *How service-orientation principles inter-relate* section later in this chapter, these four principles directly support the realization of other principles (as well as each other).

There are other qualities commonly associated with services and service-orientation. Examples include self-descriptive and coarse-grained interface design characteristics. We classify these more as service design guidelines, and they are therefore discussed as part of the design guidelines provided in Chapter 15.

NOTE

You may have noticed that the reusability and autonomy principles also were mentioned as part of the contemporary SOA characteristics described in Chapter 3. This overlap is intentional, as we simply are identifying qualities commonly associated with SOA as a whole as well as services designed for use in SOA. We further clarify the relationship between contemporary SOA characteristics and service-orientation principles in Chapter 9.

To fully understand how service-orientation principles shape service-oriented architecture, we need to explore the implications their application will have on all of the primary parts that comprise SOA. Let's take a closer look at each of the principles.

8.3.1 Services are reusable

Service-orientation encourages reuse in all services, regardless if immediate requirements for reuse exist. By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased. Inherently reusable services also reduce the need for creating wrapper services that expose a generic interface over top of less reusable services.

This principle facilitates all forms of reuse, including inter-application interoperability, composition, and the creation of cross-cutting or utility services. As we established earlier in this chapter, a service is simply a collection of related operations. It is therefore the logic encapsulated by the individual operations that must be deemed reusable to warrant representation as a reusable service (Figure 8.13).

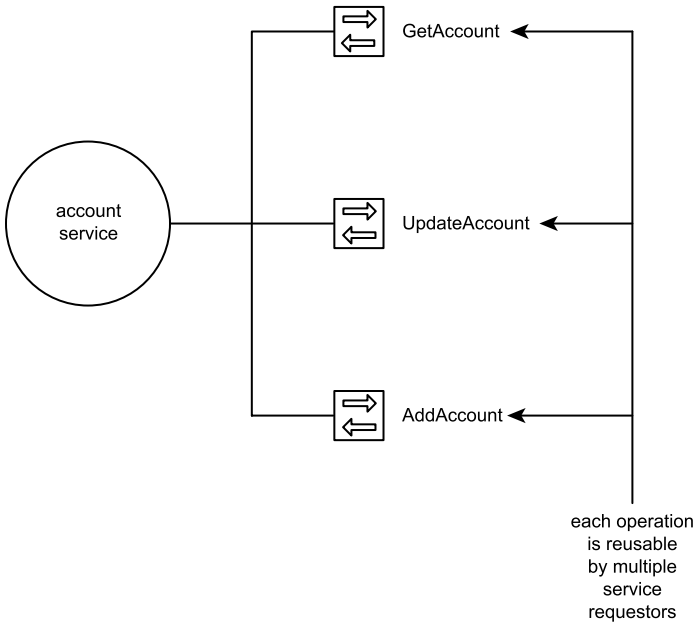

Figure 8.13
A reusable service exposes reusable operations.

Messaging also indirectly supports service reusability through the use of SOAP headers. These allow for messages to become increasingly self-reliant by grouping metadata details with message content into a single package (the SOAP envelope). Messages can be equipped with processing instructions and business rules that allow them to dictate to recipient services how they should be processed.

The processing-specific logic embedded in a message alleviates the need for a service to contain this logic. More importantly, it imposes a requirement that service operations become less activity-specific—in other words, more generic. The more generic a service's operations are, the more reusable the service.

CASE STUDY

RailCo delivered the Invoice Submission Service for the sole purpose of being able to connect to TLS's new B2B system. This Web service's primary function therefore is to send electronic invoice documents to the TLS Accounts Payable Service. The service contains the following two operations: SubmitInvoice and GetTLSMetadata (Figure 8.14).

The SubmitInvoice operation simply initiates the transmission of the invoice document. You might recall in the *Metadata exchange* section of Chapter 7 that an operation was added to periodically check the TLS Accounts Payable Service for changes to its service description. This new operation is GetTLSMetadata.

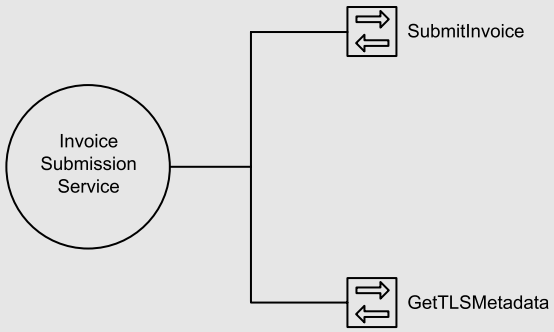

Figure 8.14
The RailCo Invoice Submission Service and its operations.

Because they were built to meet immediate and specific business requirements, these operations have no real reuse potential. The SubmitInvoice operation is designed to forward SOAP messages containing specific headers required by TLS and containing an invoice XML document structured according to a schema also defined by TLS. By its very name, the GetTLSMetadata operation identifies itself as existing for one reason: to query a specific endpoint for new metadata information.

The TLS Accounts Payable Service, on the other hand, provides a series of generic operations related to the processing of accounts payable transactions. This service is therefore used by different TLS systems, one of which is the aforementioned B2B solution.

In Chapters 11 and 12 we will submit the RailCo Invoice Submission Service to a modeling exercise in an attempt to reshape it into a service that implements actual service-orientation principles, including reusability.

IN PLAIN ENGLISH

One day, a government inspector stops by our car washing operation. Not knowing who he is, I ask if he would like his car washed and waxed or just washed. He responds by asking a question of his own. "Do you have a business license for this operation?"

A subsequent conversation between the inspector and our team results in the revelation that we have indeed been operating without a business license. We are therefore ordered to cease all work until we obtain one. We scramble to find out what needs to be done. This leads us to visit the local Business License Office to start the process of acquiring a license.

The Business License Office provides a distinct service: issuing and renewing business licenses. It is not there to service just our car washing company; it is there to provide this service to anyone requesting it. Because its service is designed to facilitate multiple service requestors, the logic that enables the service can be classified as being reusable.

8.3.2 Services share a formal contract

Service contracts provide a formal definition of:

- the service endpoint
- each service operation
- every input and output message supported by each operation
- rules and characteristics of the service and its operations

Service contracts therefore define almost all of the primary parts of an SOA (Figure 8.15). Good service contracts also may provide semantic information that explains how a service may go about accomplishing a particular task. Either way, this information establishes the agreement made by a service provider and its service requestors.

Because this contract is shared among services, its design is extremely important. Service requestors that agree to this contract can become dependent on its definition. Therefore, contracts need to be carefully maintained and versioned after their initial release.

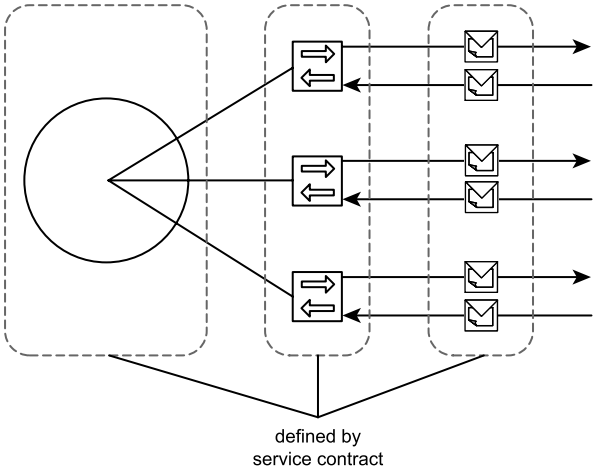

Figure 8.15
Service contracts formally define the service, operation, and message components of a service-oriented architecture.

As explained in Chapter 5, service description documents, such as the WSDL definition, XSD schemas, and policies, can be viewed collectively as a communications contract that expresses exactly how a service can be programmatically accessed.

CASE STUDY

From the onset, RailCo and TLS agreed to each other's service contracts, which enabled these two companies to interact via the TLS B2B system. The rules of the contract and the definition of associated service description documents all are provided by TLS to ensure a standardized level of conformance that applies to each of its online vendors.

One day, RailCo is informed that TLS has revised the policy published with the Accounts Payable Service. A new rule has been added where TLS is offering better payment terms to vendors in exchange for larger discounts. RailCo has the choice to continue pricing their products at the regular amounts and face a payment term of 60 days for their invoices or reduce their prices to get a payment term of 30 days.

Both of these options are acceptable contract conditions published by TLS. After some evaluation, RailCo decides not to take advantage of the reduced payment terms and therefore does not adjust its product prices.

IN PLAIN ENGLISH

For us to get a business license, we must fill out an application form. This process essentially formalizes our request in a format required and expected by the Business License Office.

The completed application form is much like a contract between the service provider and the requestor of the service. Upon accepting the form, the service provider agrees to act on the request.

8.3.3 Services are loosely coupled

No one can predict how an IT environment will evolve. How automation solutions grow, integrate, or are replaced over time can never be accurately planned out because the requirements that drive these changes are almost always external to the IT environment. Being able to ultimately respond to unforeseen changes in an efficient manner is a key goal of applying service-orientation. Realizing this form of agility is directly supported by establishing a loosely coupled relationship between services (Figure 8.16).

Loose coupling is a condition wherein a service acquires knowledge of another service while still remaining independent of that service. Loose coupling is achieved through the use of service contracts that allow services to interact within predefined parameters.

It is interesting to note that within a loosely coupled architecture, service contracts actually tightly couple operations to services. When a service is formally described as being the location of an operation, other services will depend on that operation-to-service association.

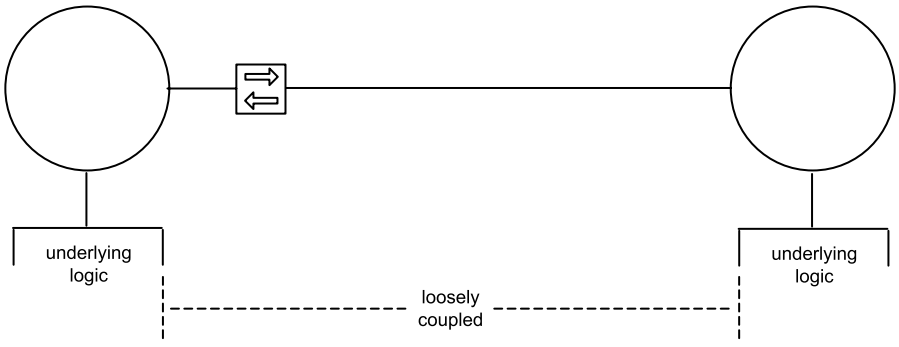

Figure 8.16
Services limit dependencies to the service contract, allowing underlying provider and requestor logic to remain loosely coupled.

CASE STUDY

Through the use of service contracts, RailCo and TLS services are naturally loosely coupled. However, one could say that the extent of loose coupling between the two service provider entities is significantly different.

TLS services are designed to facilitate multiple B2B partners, as well as internal reuse and composition requirements. This makes TLS services *very* loosely coupled from any of its service requestors.

RailCo's services, on the other hand, are designed specifically to interact with designated TLS services that are part of the overall TLS B2B solution. No attempt was made to make these services useful for any other service requestors. RailCo services are therefore considered *less* loosely coupled than TLS services.

IN PLAIN ENGLISH

After we have submitted our form, we are not required to remain at the Business License Office, nor do we need to stay in touch with them. We only need to wait until the application is processed and a license is (hopefully) issued.

This is much like an asynchronous message exchange, but it is also a demonstration of a loosely coupled relationship between services or between service provider and requestor. All we need to interact with the Business License Office is an application form that defines the information the office requires to process our request. Prior to and subsequent to the submission of that request, our car washing team (service requestor) and the Business License Office (service provider) remain independent of each other.

8.3.4 Services abstract underlying logic

Also referred to as *service interface-level abstraction*, it is this principle that allows services to act as black boxes, hiding their details from the outside world. The scope of logic represented by a service significantly influences the design of its operations and its position within a process.

There is no limit to the amount of logic a service can represent. A service may be designed to perform a simple task, or it may be positioned as a gateway to an entire automation solution. There is also no restriction as to the source of application logic a service can draw upon. For example, a single service can, technically, expose application logic from two different systems (Figure 8.17).

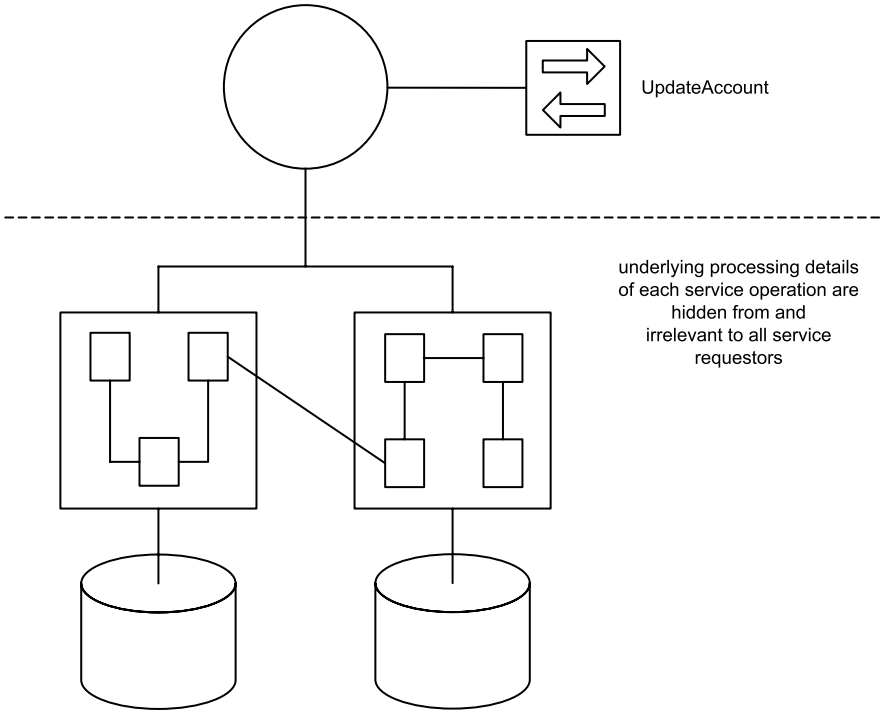

Figure 8.17
Service operations abstract the underlying details of the functionality they expose.

Operation granularity is therefore a primary design consideration that is directly related to the range and nature of functionality being exposed by the service. Again, it is the individual operations that collectively abstract the underlying logic. Services simply act as containers for these operations.

Service interface-level abstraction is one of the inherent qualities provided by Web services. The loosely coupled communications structure requires that the only piece of knowledge services need to interact is each others' service descriptions.

CASE STUDY

Because both RailCo and TLS employ Web services to communicate, each environment successfully implements service interface-level abstraction. On RailCo's end, this abstraction hides the legacy systems involved with generating electronic invoice documents and processing incoming purchase orders. On the TLS side, services hide service compositions wherein processing duties are delegated to specialized services as part of single activities (Figure 8.18).

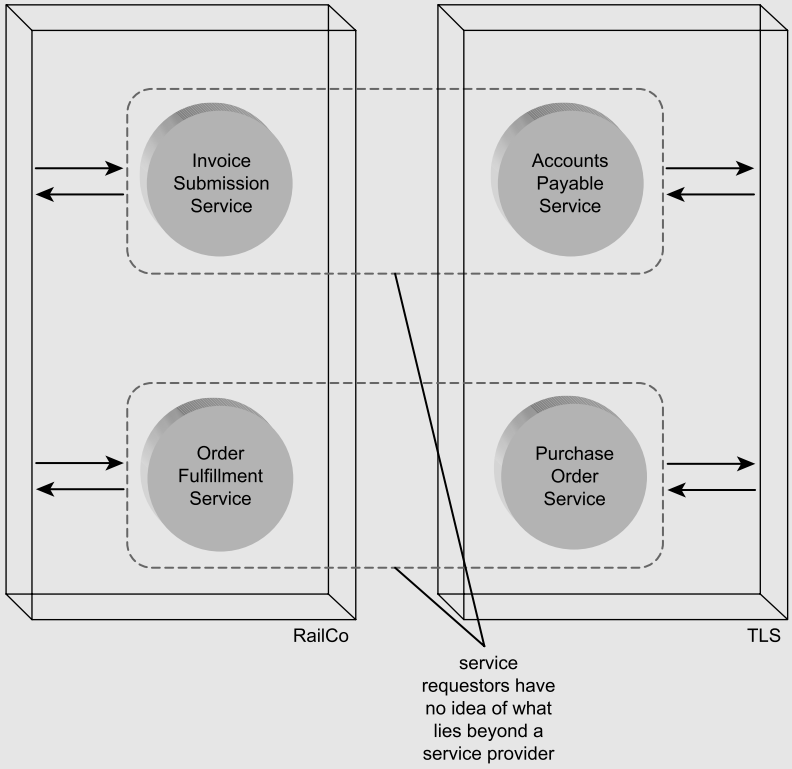

Figure 8.18
Neither of RailCo's or TLS's service requestors require any knowledge of what lies behind the other's service providers.

IN PLAIN ENGLISH

The tasks required for the Business License Office to process our request include:

- A name check to ensure that the name of our company "Oasis Car Wash" isn't already taken.
- A background check of the company principals to ensure that none of us have had past bankruptcies.
- A verification of our sub-lease agreement to ensure that we are, in fact, allowed to operate at the gas station we have been using.

> These and other tasks are performed completely unbeknownst to us. We don't know or necessarily care what the Business License Office needs to do to process our application. We are just interested in the expected outcome: the issuance of our license.

8.3.5 Services are composable

A service can represent any range of logic from any types of sources, including other services. The main reason to implement this principle is to ensure that services are designed so that they can participate as effective members of other service compositions if ever required. This requirement is irrespective of whether the service itself composes others to accomplish its work (Figure 8.19).

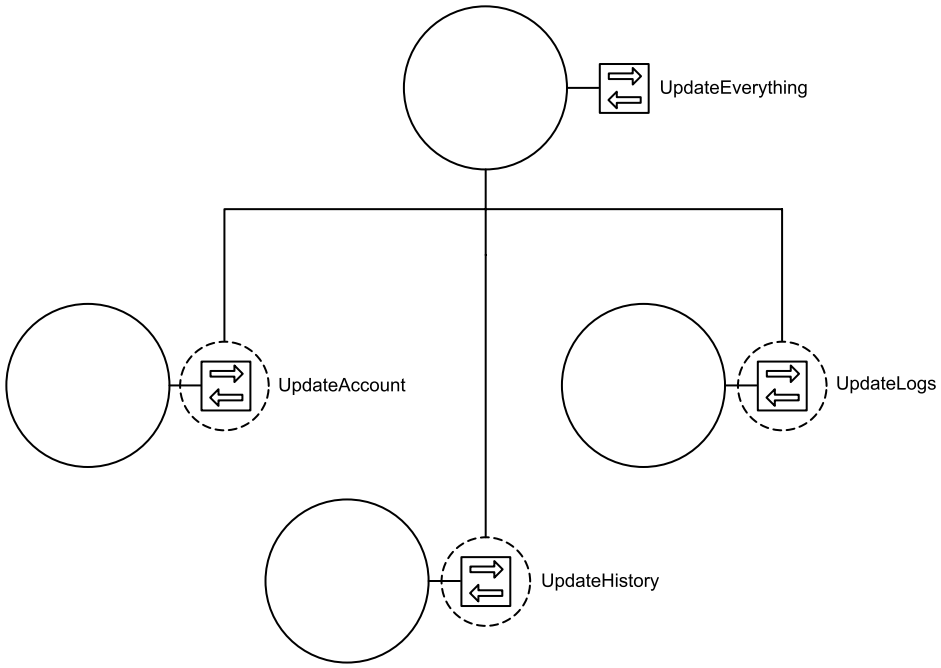

Figure 8.19
The UpdateEverything operation encapsulating a service composition.

A common SOA extension that underlines composability is the concept of orchestration. Here, a service-oriented process (which essentially can be classified as a service composition) is controlled by a parent process service that composes process participants.

The requirement for any service to be composable also places an emphasis on the design of service operations. Composability is simply another form of reuse, and therefore operations need to be designed in a standardized manner and with an appropriate level of granularity to maximize composition opportunities.

CASE STUDY

As with RailCo's Invoice Submission Service, its Order Fulfillment Service was created to meet a specific requirement in support of communication with TLS's B2B solution.

The Order Fulfillment Service contains just one public operation called ProcessTLSPO (Figure 8.20). This operation is designed in compliance with TLS vendor service specifications so that it is fully capable of receiving POs submitted by the TLS Purchase Order Service. Part of this compliance requires the operation to be able to process custom SOAP headers containing proprietary security tokens.

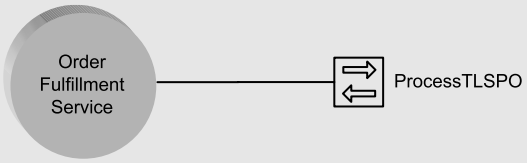

Figure 8.20
The RailCo Order Fulfillment Service with its one operation.

Though the Order Fulfillment Service is capable of acting as a composition member, its potential for being useful to any future compositions is limited. Composition support is similar to reusability in that generic functionality exposed by operations make a service more composable. This RailCo service provides one operation that performs a very specialized function, customized to processing a specific document from a specific source. It will likely not be a suitable composition member, but it can act as a controller service, composing other services to complete its PO processing tasks.

The TLS Accounts Payable Service already establishes a well-defined composition, wherein it acts as a controller service that composes the Vendor Profile and Ledger Services (Figure 8.21). Because they each expose a complete set of generic

operations, all three of these services are capable of participating in other composition configurations.

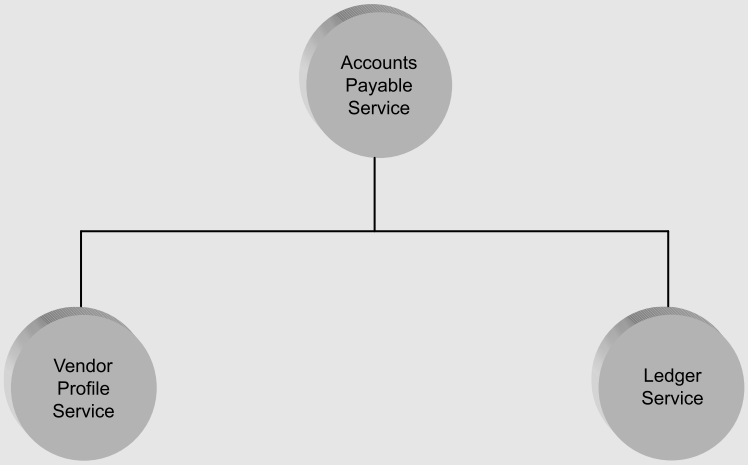

Figure 8.21
The TLS Accounts Payable Service composition.

IN PLAIN ENGLISH

Given that the services provided by the Business License Office are distinct and reusable, it can be asked to assist other government offices to participate in the completion of other services. For example, the Business Relocation Office manages all administrative paperwork for businesses that need to be moved when their location is scheduled for demolition.

As part of its many tasks, this office takes care of revising the business license information for the affected company. It does so by enlisting the Business License Office and requesting that they issue a new business license for a particular organization.

By reusing the services offered by the Business License Office, the Business Relocation Office has effectively composed services, much like a controller service reuses and composes other service providers.

8.3.6 Services are autonomous

Autonomy requires that the range of logic exposed by a service exist within an explicit boundary. This allows the service to execute self-governance of all its processing. It also eliminates dependencies on other services, which frees a service from ties that could

inhibit its deployment and evolution (Figure 8.22). Service autonomy is a primary consideration when deciding how application logic should be divided up into services and which operations should be grouped together within a service context.

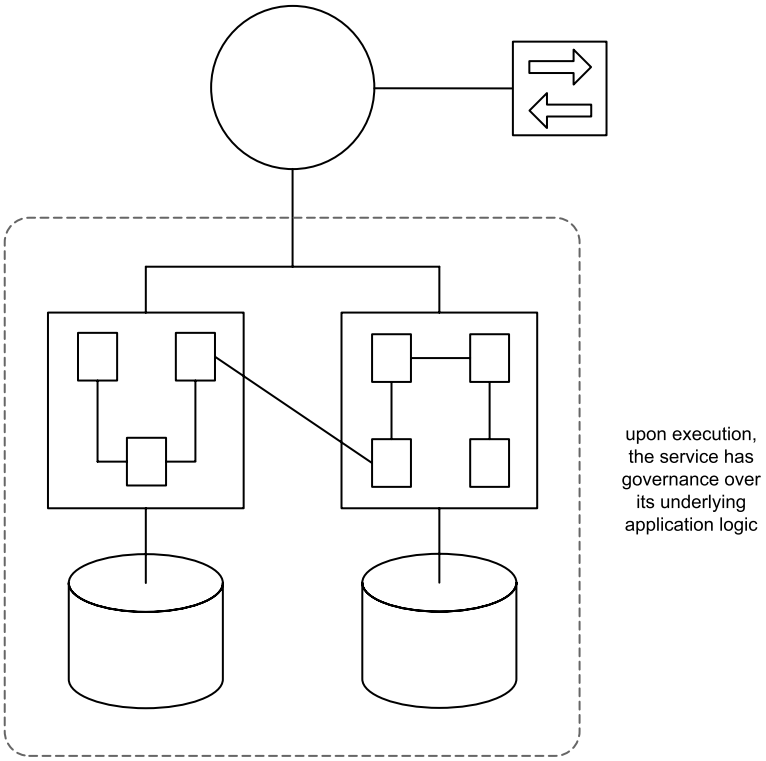

Figure 8.22
Autonomous services have control over underlying resources.

Deferring the location of business rules is one way to strengthen autonomy and keep services more generic. Processes generally assume this role by owning the business rules that determine how the process is structured and, subsequently, how services are composed to automate the process logic. This is another aspect of orchestration explored in the *Orchestration service layer* section in Chapter 9.

Note that autonomy does not necessarily grant a service exclusive ownership of the logic it encapsulates. It only guarantees that at the time of execution, the service has control over whatever logic it represents. We therefore can make a distinction between two types of autonomy.

- *Service-level autonomy*—Service boundaries are distinct from each other, but the service may share underlying resources. For example, a wrapper service that encapsulates a legacy environment that also is used independently from the service has service-level autonomy. It governs the legacy system but also shares resources with other legacy clients.
- *Pure autonomy*—The underlying logic is under complete control and ownership of the service. This is typically the case when the underlying logic is built from the ground up in support of the service.

CASE STUDY

Given the distinct tasks they perform, the following three RailCo services all are autonomous:

- Invoice Submission Service
- Order Fulfillment Service
- TLS Subscription Service

Each represents a specific boundary of application logic that does not overlap with the boundary of any other services.

Autonomy in RailCo's services was achieved inadvertently. No conscious effort was made to avoid application overlap, as the services were delivered to simply meet specific connectivity requirements.

As shown in Figure 8.23, the Invoice Processing and Order Fulfillment Services encapsulate legacy logic. The legacy accounting system also is used by clients independently from the services, which makes this service-level autonomy. The TLS Notification Service achieves pure autonomy, as it represents a set of custom components created only in support of this service.

In environments where a larger number of services exist and new services are built on a regular basis, it is more common to introduce dedicated modeling processes so pure service autonomy is preserved among individual services. At TLS, for example, services undergo a service-oriented analysis to guarantee autonomy and avoid encapsulation overlap. (Service-oriented analysis is explained in Chapters 11 and 12.)

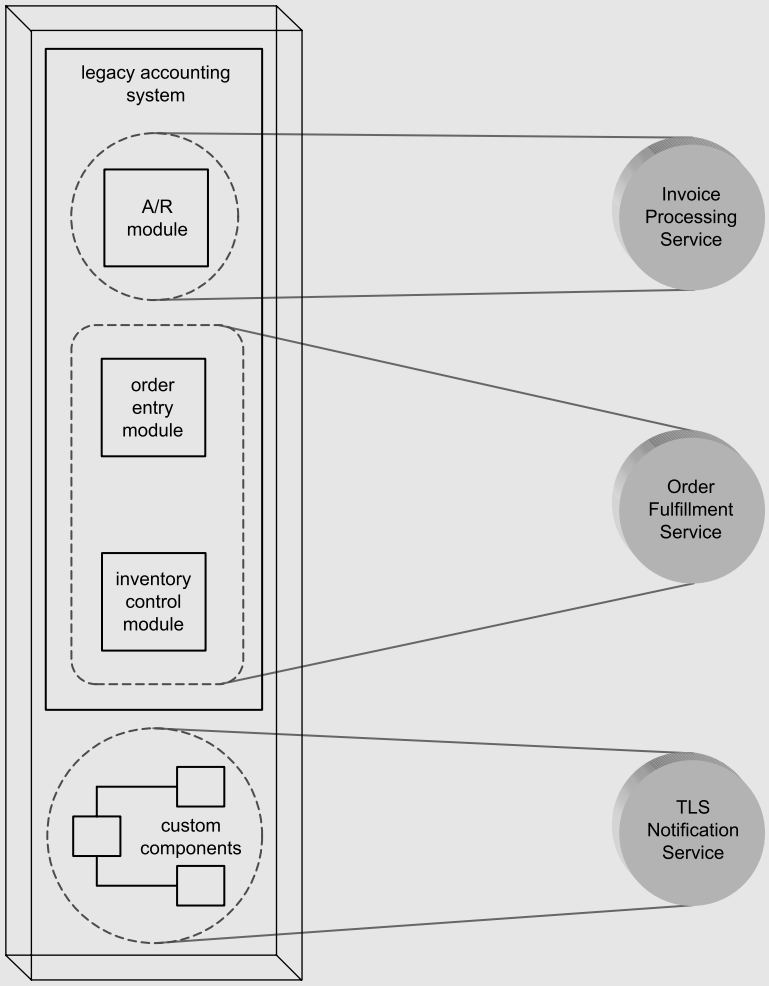

Figure 8.23
RailCo's services luckily encapsulate explicit portions of legacy and newly added application logic.

IN PLAIN ENGLISH

Let's revisit the three tasks performed by the Business License Office when processing an application for a new business license:

- name check
- background check
- location verification

The Business License Office owns the corporate name database required to perform a name check. Also the office has personnel dedicated to visiting and verifying business site locations. When completing these two tasks, the Business License Office therefore has complete self-governance. However, when having to perform a background check, the office must share a database system with the Revenue Office. When it gets access, it can retrieve an abbreviated credit history for each of the company principals listed on the application.

The Business License Office's reliance on the shared database reduces its independence somewhat. However, its overall ability to perform the tasks within its own boundary give it a degree of autonomy.

8.3.7 Services are stateless

Services should minimize the amount of state information they manage and the duration for which they hold it. State information is data-specific to a current activity. While a service is processing a message, for example, it is temporarily stateful (Figure 8.24). If a service is responsible for retaining state for longer periods of time, its ability to remain available to other requestors will be impeded.

Statelessness is a preferred condition for services and one that promotes reusability and scalability. For a service to retain as little state as possible, its individual operations need to be designed with stateless processing considerations.

A primary quality of SOA that supports statelessness is the use of document-style messages. The more intelligence added to a message, the more independent and self-sufficient it remains. Chapters 6 and 7 explore various WS-* extensions that rely on the use of SOAP headers to carry different types of state data.

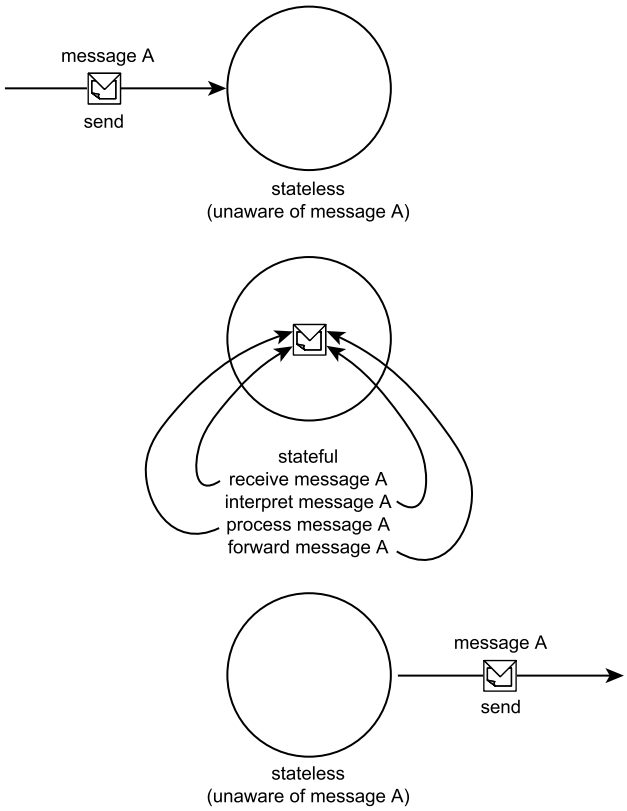

Figure 8.24
Stateless and stateful stages a service passes through while processing a message.

CASE STUDY

As with loose coupling, statelessness is a quality that can be measured in degrees. The RailCo Order Fulfillment Service is required to perform extra run-time parsing and processing of various standard SOAP header blocks to successfully receive a purchase order document submitted by the TLS Purchase Order Service. This processing ties up the Order Fulfillment Service longer than, say, the Invoice Submission Service, which simply forwards a predefined SOAP message to the TLS Accounting Service.

IN PLAIN ENGLISH

During the initial review of the application, our company was briefly discussed by personnel at the Business License Office. But after the application was fully processed, no one really retained any memory of our request.

Though the details of our application have been logged and recorded in various repositories, there is no further need for anyone involved in the processing of our request to remember further information about it once the application processing task was completed. To this extent, the Business License Office simulates a degree of statelessness. It processes many requests every day, and there is no benefit to retaining information about a completed request.

8.3.8 Services are discoverable

Discovery helps avoid the accidental creation of redundant services or services that implement redundant logic. Because each operation provides a potentially reusable piece of processing logic, metadata attached to a service needs to sufficiently describe not only the service's overall purpose, but also the functionality offered by its operations.

Note that this service-orientation principle is related to but distinct from the contemporary SOA characteristic of discoverability. On an SOA level, discoverability refers to the architecture's ability to provide a discovery mechanism, such as a service registry or directory. This effectively becomes part of the IT infrastructure and can support numerous implementations of SOA. On a service level, the principle of discoverability refers to the design of an individual service so that it can be as discoverable as possible.

CASE STUDY

RailCo provides no means of discovery for its services, either internally or to the outside world. Though outfitted with its own WSDL definition and fully capable of acting as a service provider, the Invoice Submission Service is primarily utilized as a service requestor and currently expects no communication outside of the TLS Accounts Payable Service

Similarly, the RailCo Order Fulfillment Service was registered manually with the TLS B2B solution so that it would be placed on the list of vendors that receive purchase orders. This service provides no reusable functionality and is therefore considered to have no immediate requirement for discovery.

Due to the reusable nature of TLS services and because of the volume of services that are expected to exist in TLS technical environments, an internal service registry was established (as shown in Figure 8.25 and originally explained in Chapter 5). This piece of TLS infrastructure promotes discoverability and prevents accidental redundancy. It further leverages the existing design standards used by TLS that promote the creation of descriptive metadata documents in support of service discoverability.

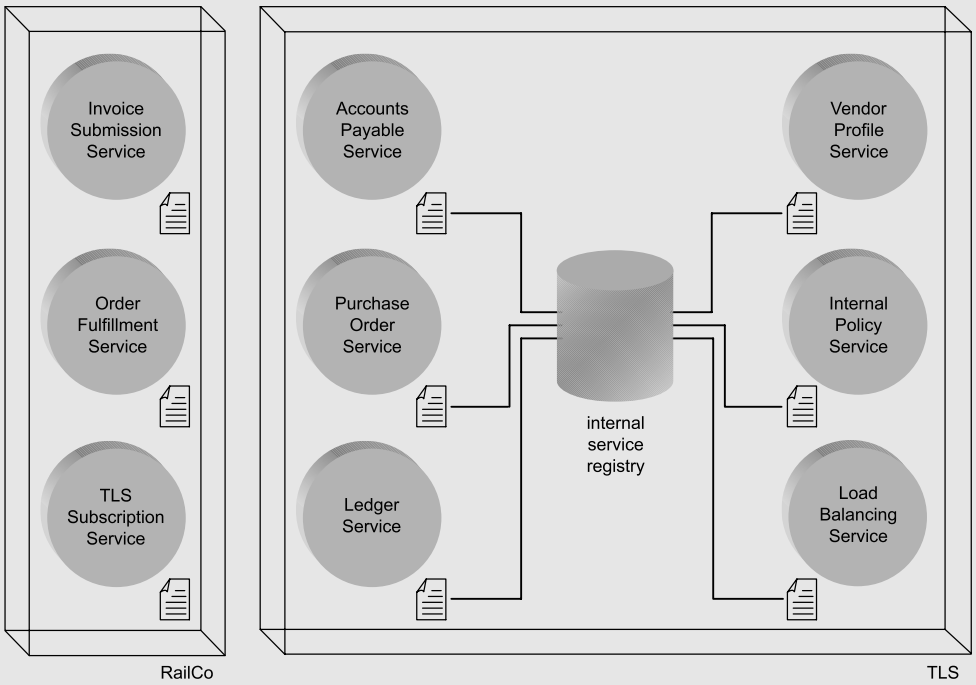

Figure 8.25
RailCo's services are not discoverable, but TLS's inventory of services are stored in an internal registry.

TLS is not interested in making its services publicly discoverable, which is why it does not register them with a public service registry. Vendors that participate in the TLS B2B system only are allowed to do so after a separate negotiation, review, and registration process.

IN PLAIN ENGLISH

After some time, our business license is finally issued. Upon receiving the certificate in the mail, we are back in business. Looking back at how this whole process began, though, there is one step we did not discuss. When we first learned that we were required to get a business license, we had to find out where the Business License Office was located. This required us to search through the phone book and locate a listing with contact information.

A service registry provides a discovery mechanism very much like a phone book, allowing potential requestors to query and check candidate service providers. In the same manner in which a registry points to service descriptions, the phone book listing led us to the location at which we were able to obtain the original business license application form.

More relevant to the principle of service discoverability is the fact that steps were taken to make the Business License Office itself discoverable. Examples include signs in the lobby of the high-rise in which the office is located, a sign on the office entrance door, brochures located at other offices, and so on.

SUMMARY OF KEY POINTS

- Different organizations have published their own versions of service-oriented principles. As a result, many variations exist.
- The most common principles relate to loose coupling, autonomy, discoverability, composability, reuse, service contracts, abstraction, and statelessness.

8.4 How service-orientation principles inter-relate

When reading through the previous descriptions, a number of questions might come to mind, such as:

- What's the difference between reusability and composability? (Aren't you reusing a service when you compose it?)
- What's the difference between autonomy and statelessness? (Aren't both a representation of service independence?)
- What's the difference between loose coupling and the use of a service contract? (Doesn't a service contract automatically implement loose coupling?)

To answer these and other questions, this section revisits our service-orientation principles to explore how each relates to, supports, or is affected by others. To accomplish this, we abbreviate the original names we assigned each principle, as follows:

- Services are reusable = service reusability
- Services share a formal contract = service contract
- Services are loosely coupled = service loose coupling
- Services abstract underlying logic = service abstraction
- Services are composable = service composability
- Services are autonomous = service autonomy
- Services are stateless = service statelessness
- Services are discoverable = service discoverability

We intentionally prefix each principle with the word "service" to emphasize that the principle applies to the design of a service only, as opposed to our SOA characteristics, which apply to the design of SOA as a whole.

NOTE

Each relationship is essentially described twice within these sections. This repetitiveness is intentional, as this part of the chapter is provided more for reference purposes. Feel free to skip ahead if you are not interested in learning about each individual principle-to-principle relationship at this point.

8.4.1 Service reusability

When a service encapsulates logic that is useful to more than one service requestor, it can be considered reusable. The concept of reuse is supported by a number of complementary service principles, as follows.

- Service autonomy establishes an execution environment that facilitates reuse because the service has independence and self-governance. The less dependencies a service has, the broader the applicability of its reusable functionality.
- Service statelessness supports reuse because it maximizes the availability of a service and typically promotes a generic service design that defers activity-specific processing outside of service logic boundaries.

- Service abstraction fosters reuse because it establishes the black box concept, where processing details are completely hidden from requestors. This allows a service to simply express a generic public interface.
- Service discoverability promotes reuse, as it allows requestors (and those that build requestors) to search for and discover reusable services.
- Service loose coupling establishes an inherent independence that frees a service from immediate ties to others. This makes it a great deal easier to realize reuse.

Additionally, the principle of service reuse itself enables the following related principle:

- Service composability is primarily possible because of reuse. The ability for one service to compose an activity around the utilization of a collection of services is feasible when those services being composed are built for reuse. (It is technically possible to build a service so that its sole purpose is to be composed by another, but reuse is generally emphasized.)

Figure 8.26 provides a diagram showing how the principle of service reusability relates to others.

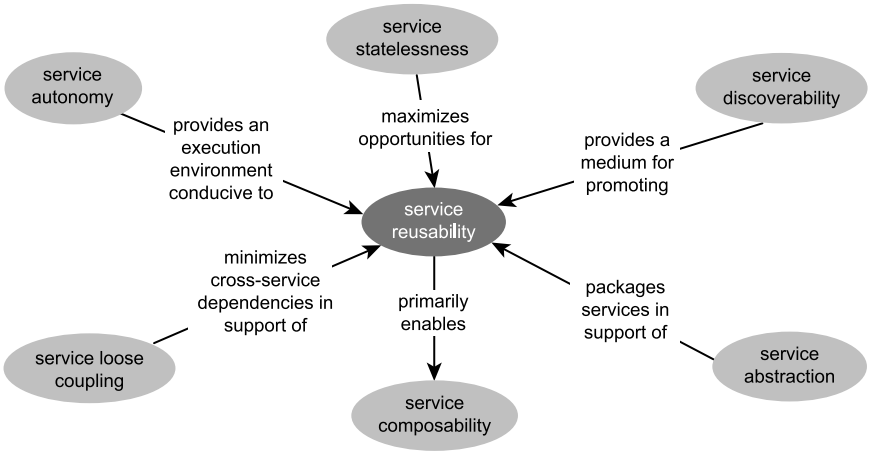

Figure 8.26
Service reusability and its relationship with other service-orientation principles.

8.4.2 Service contract

A service contract is a representation of a service's collective metadata. It standardizes the expression of rules and conditions that need to be fulfilled by any requestor wanting to interact with the service.

Service contracts represent a cornerstone principle in service-orientation and therefore support other principles in various ways, as follows:

- Service abstraction is realized through a service contract, as it is the metadata expressed in the contract that defines the only information made available to service requestors. All additional design, processing, and implementation details are hidden behind this contract.
- Service loose coupling is made possible through the use of service contracts. Processing logic from different services do not need to form tight dependencies; they simply need an awareness of each other's communication requirements, as expressed by the service description documents that comprise the service contract.
- Service composability is indirectly enabled through the use of service contracts. It is via the contract that a controller service enlists and uses services that act as composition members.
- Service discoverability is based on the use of service contracts. While some registries provide information supplemental to that expressed through the contract, it is the service description documents that are primarily searched for in the service discovery process.

The diagram in Figure 8.27 illustrates how the principle of service contract usage relates to others.

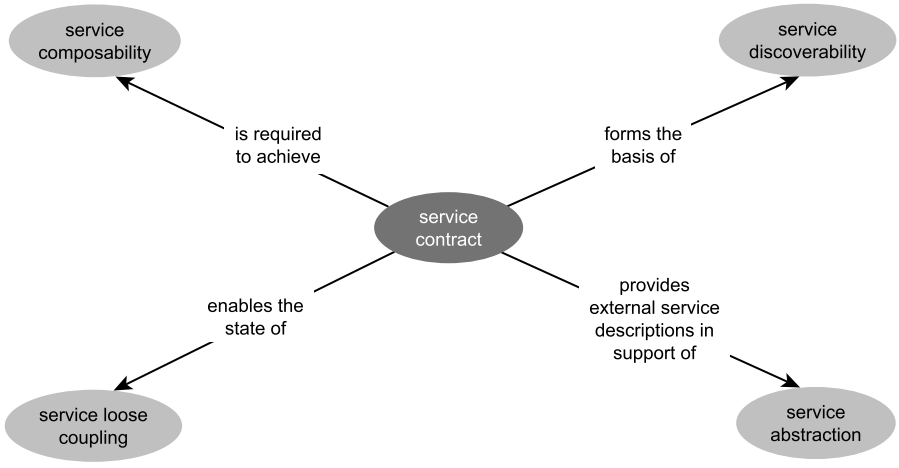

Figure 8.27
The service contract and its relationship with other service-orientation principles.

8.4.3 Service loose coupling

Loose coupling is a state that supports a level of independence between services (or between service providers and requestors). As you may have already noticed, independence or non-dependency is a fundamental aspect of services and SOA as a whole. Therefore, the principle of persisting loose coupling across services supports the following other service-orientation principles:

- Service reusability is supported through loose coupling because services are freed from tight dependencies on others. This increases their availability for reuse opportunities.
- Service composability is fostered by the loose coupling of services, especially when services are dynamically composed.
- Service statelessness is directly supported through the loosely coupled communications framework established by this principle.
- Service autonomy is made possible through this principle, as it is the nature of loose coupling that minimizes cross-service dependencies.

Additionally, service loose coupling is directly implemented through the application of a related service-orientation principle:

- Service contracts are what enable loose coupling between services, as the contract is the only piece of information required for services to interact.

Figure 8.28 demonstrates these relationships.

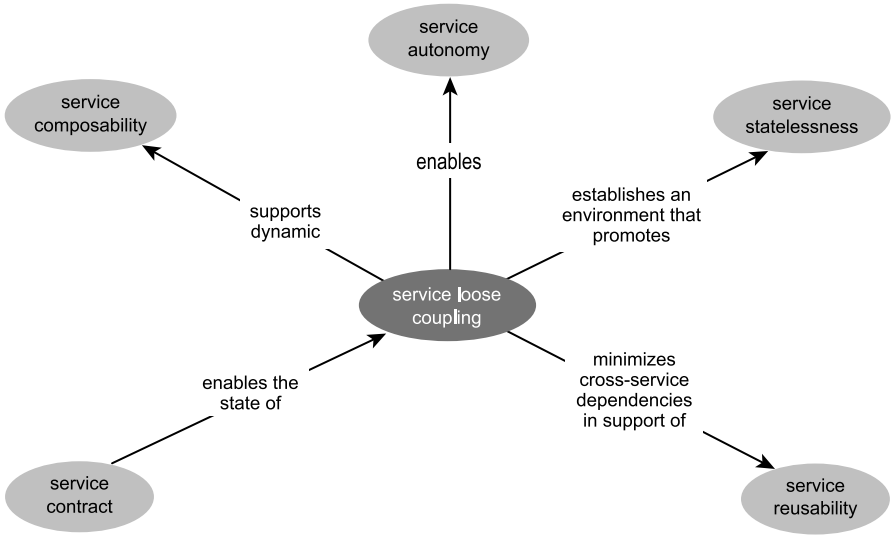

Figure 8.28
Service loose coupling and its relationship with other service-orientation principles.

8.4.4 Service abstraction

Part of building solutions with independent services is allowing those services to encapsulate potentially complex processing logic and exposing that logic through a generic and descriptive interface. This is the primary benefit of service abstraction, a principle that is related to others, as explained here:

- Service contracts, in a manner, implement service abstraction by providing the official description information that is made public to external service requestors.
- Service reusability is supported by abstraction, as long as what is being abstracted is actually reusable.

These relationships are shown in Figure 8.29.

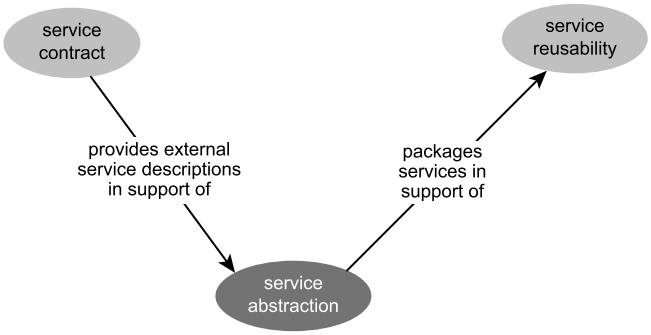

Figure 8.29
Service abstraction and its relationship with other service-orientation principles.

8.4.5 Service composability

Designing services so that they support composition by others is fundamental to building service-oriented solutions. Service composability therefore is tied to service-orientation principles that support the concept of service composition, as follows:

- Service reusability is what enables one service to be composed by numerous others. It is expected that reusable services can be incorporated within different compositions or reused independently by other service requestors.
- Service loose coupling establishes a communications framework that supports the concept of dynamic service composition. Because services are freed from many dependencies, they are more available to be reused via composition.
- Service statelessness supports service composability, especially in larger compositions. A service composition is reliant on the design quality and commonality of its collective parts. If all services are stateless (by, for example, deferring activity-specific logic to messages), the overall composition executes more harmoniously.
- Service autonomy held by composition members strengthens the overall composition, but the autonomy of the controller service itself actually is decreased due to the dependencies on its composition members.
- Service contracts enable service composition by formalizing the runtime agreement between composition members.

Figure 8.30 further illustrates these relationships.

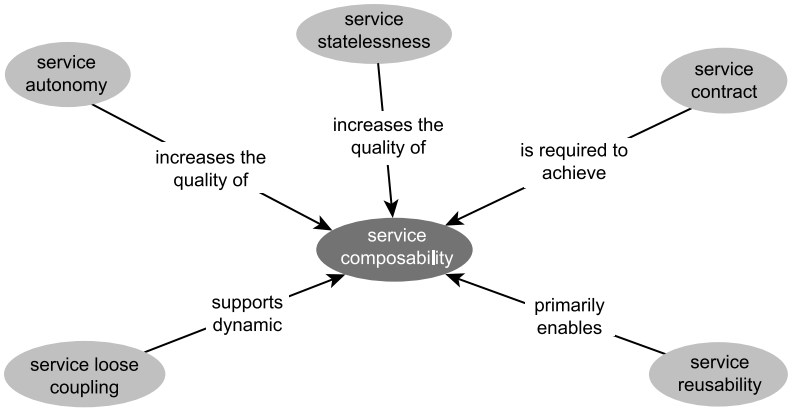

Figure 8.30
Service composability and its relationship with other service-orientation principles.

8.4.6 Service autonomy

This principle applies to a service's underlying logic. By providing an execution environment over which a service has complete control, service autonomy relates to several other principles, as explained here:

- Service reusability is more easily achieved when the service offering reusable logic has self-governance over its own logic. Service Level Agreement (SLA) type requirements that come to the forefront for utility services with multiple requestors, such as availability and scalability, are fulfilled more easily by an autonomous service.
- Service composability is also supported by service autonomy—for much of the same reasons autonomy supports service reusability. A service composition consisting of autonomous services is much more robust and collectively independent.
- Service statelessness is best implemented by a service that can execute independently. Autonomy indirectly supports service statelessness. (However, it is very easy to create a stateful service that is also fully autonomous.)
- Service autonomy is a quality that is realized by leveraging the loosely coupled relationship between services. Therefore service loose coupling is a primary enabler of this principle.

The diagram in Figure 8.31 shows how service autonomy relates to these other principles.

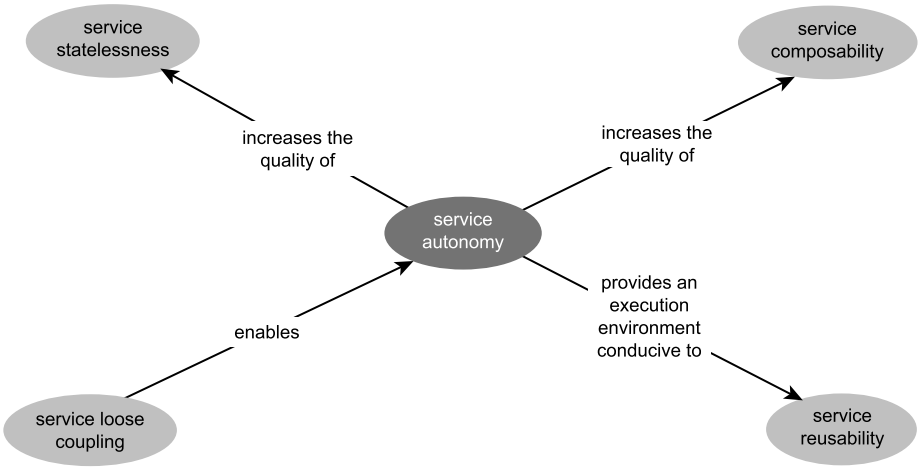

Figure 8.31
Service autonomy and its relationship with other service-orientation principles.

8.4.7 Service statelessness

To successfully design services not to manage state requires the availability of resources surrounding the service to which state management responsibilities can be delegated. However, the principle of statelessness is also indirectly supported by the following service-orientation principles:

- Service autonomy provides the ability for a service to control its own execution environment. By removing or reducing dependencies it becomes easier to build statelessness into services, primarily because the service logic can be fully customized to defer state management outside of the service logic boundary.
- Service loose coupling and the overall concept of loose coupling establishes a communication paradigm that is fully realized through messaging. This, in turn, supports service statelessness, as state information can be carried and persisted by the messages that pass through the services.

Service statelessness further supports the following principles:

- Service composability benefits from stateless composition members, as they reduce dependencies and minimize the overhead of the composition as a whole.

- Service reuse becomes more of a reality for stateless services, as availability of the service to multiple requestors is increased and the absence of activity-specific logic promotes a generic service design.

Figure 8.32 illustrates how service statelessness relates to the other service-orientation principles.

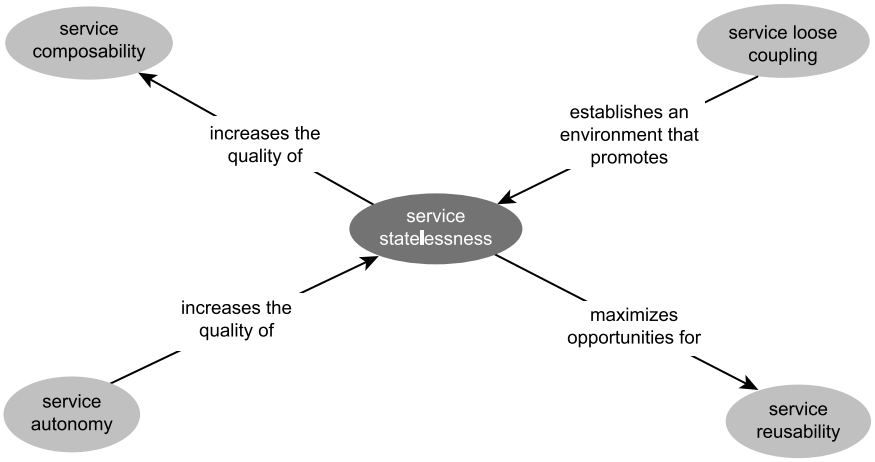

Figure 8.32
Service statelessness and its relationship with other service-orientation principles.

8.4.8 Service discoverability

Designing services so that they are naturally discoverable enables an environment whereby service logic becomes accessible to new potential service requestors. This is why service discoverability is tied closely to the following service-orientation principles:

- Service contracts are what service requestors (or those that create them) actually discover and subsequently assess for suitability. Therefore, the extent of a service's discoverability can typically be associated with the quality or descriptiveness of its service contract.
- Service reusability is what requestors are looking for when searching for services and it is what makes a service potentially useful once it has been discovered. A service that isn't reusable would likely never need to be discovered because it would probably have been built for a specific service requestor in the first place.

The diagram in Figure 8.33 shows how service discoverability fits in with the other service-orientation principles.

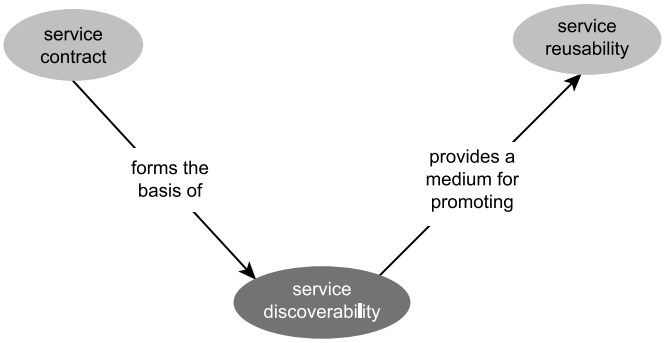

Figure 8.33
Service discoverability and its relationship with other service-orientation principles.

SUMMARY OF KEY POINTS

- Service-orientation principles are not realized in isolation; principles relate to and support other principles in different ways.
- Principles, such as service reusability and service composability, benefit from the support of other implemented principles.
- Principles, such as service loose coupling, service contract, and service autonomy, provide significant support for the realization of other principles.

8.5 Service-orientation and object-orientation (Part II)

Having now covered the fundamentals behind service-orientation principles, we can continue the discussion we began in the *Service-orientation and object-orientation (Part I)* section from Chapter 4.

Those of you familiar with object-oriented analysis and design probably will have recognized a similarity between a number of the service-orientation principles discussed and the more established principles of object-orientation.

Indeed, service-orientation owes much of its existence to object-oriented concepts and theory. Table 8.1 provides a look at which common object-orientation principles are related to the service-orientation principles we've been discussing.

Table 8.1 An overview of how service-orientation principles relate to object-orientation principles.

Service-Orientation Principle	Related Object-Orientation Principles
service reusability	Much of object-orientation is geared toward the creation of reusable classes. The object-orientation principle of modularity standardized decomposition as a means of application design. Related principles, such as abstraction and encapsulation, further support reuse by requiring a distinct separation of interface and implementation logic. Service reusability is therefore a continuation of this goal.
service contract	The requirement for a service contract is very comparable to the use of interfaces when building object-oriented applications. Much like WSDL definitions, interfaces provide a means of abstracting the description of a class. And, much like the "WSDL first" approach encouraged within SOA, the "interface first" approach also is considered an object-orientation best practice.
service loose coupling	Although the creation of interfaces somewhat decouples a class from its consumers, coupling in general is one of the primary qualities of service-orientation that deviates from object-orientation. The use of inheritance and other object-orientation principles encourages a much more tightly coupled relationship between units of processing logic when compared to the service-oriented design approach.
service abstraction	The object-orientation principle of abstraction requires that a class provide an interface to the external world and that it be accessible via this interface. Encapsulation supports this by establishing the concept of information hiding, where any logic within the class outside of what is exposed via the interface is not accessible to the external world. Service abstraction accomplishes much of the same as object abstraction and encapsulation. Its purpose is to hide the underlying details of the service so that only the service contract is available and of concern to service requestors.

Service-Orientation Principle	Related Object-Orientation Principles
service composability	Object-orientation supports association concepts, such as aggregation and composition. These, within a loosely coupled context, also are supported by service-orientation. For example, the same way a hierarchy of objects can be composed, a hierarchy of services can be assembled through service composability.
service autonomy	The quality of autonomy is more emphasized in service-oriented design than it has been with object-oriented approaches. Achieving a level of independence between units of processing logic is possible through service-orientation, by leveraging the loosely coupled relationship between services. Cross-object references and inheritance-related dependencies within object-oriented designs support a lower degree of object-level autonomy.
service statelessness	Objects consist of a combination of class and data and are naturally stateful. Promoting statelessness within services therefore tends to deviate from typical object-oriented design. Although it is possible to create stateful services and stateless objects, the principle of statelessness is generally more emphasized with service-orientation.
service discoverability	Designing class interfaces to be consistent and self-descriptive is another object-orientation best practice that improves a means of identifying and distinguishing units of processing logic. These qualities also support reuse by allowing classes to be more easily discovered. Discoverability is another principle more emphasized by the service-orientation paradigm. It is encouraged that service contracts be as communicative as possible to support discoverability at design time and runtime.

As it stands today, object-orientation and service-orientation are not necessarily competitive paradigms. Service-orientation clearly has several roots in object-orientation, and typical contemporary service-oriented solutions will consist of a mixture of services

(that adhere to service-orientation principles) and object-oriented components. With a balanced design, each set of principles can be properly positioned and leveraged to complement and support each other.

SUMMARY OF KEY POINTS

- Several principles of service-orientation are related to and derived from object-orientation principles.
- Some object-orientation principles, such as inheritance, do not fit into the service-oriented world.
- Some service-orientation principles, such as loose coupling and autonomy, are not directly promoted by object-orientation.

8.6 Native Web service support for service-orientation principles

Having now worked through the individual principles of service-orientation in some detail, it becomes evident that Web services provide inherent support for some of these principles. It is important to recognize specifically which principles are built into the structure of common Web services because this allows us to place an emphasis on those that require a conscious effort to realize.

Table 8.2 recaps the principles of service-orientation and explains to what extent they are natively supported by Web services.

Table 8.2 A look at which service-orientation principles are automatically supported by Web services.

Service-Orientation Principle	Web Service Support
service reusability	Web services are not automatically reusable. This quality is related to the nature of the logic encapsulated and exposed via the Web service.
service contract	Web services require the use of service descriptions, making service contracts a fundamental part of Web services communication.
service loose coupling	Web services are naturally loosely coupled through the use of service descriptions.

Service-Orientation Principle	Web Service Support
service abstraction	Web services automatically emulate the black box model within the Web services communications framework, hiding all of the details of their underlying logic.
service composability	Web services are naturally composable. The extent to which a service can be composed, though, generally is determined by the service design and the reusability of represented logic.
service autonomy	To ensure an autonomous processing environment requires design effort. Autonomy is therefore not automatically provided by a Web service.
service statelessness	Statelessness is a preferred condition for Web services, strongly supported by many WS-* specifications and the document-style SOAP messaging model.
service discoverability	Discoverability must be implemented by the architecture and even can be considered an extension to IT infrastructure. It is therefore not natively supported by Web services.

It turns out that half of the principles of service-orientation are natural characteristics of common Web services. This underlines the synergy of the marriage between SOA and the Web services technology platform and gives us a good indication as to why Web services have been so successful in realizing SOA.

It also highlights the principles that require special attention when building service-oriented solutions. The four principles identified as *not* being automatically provided by Web services are:

- service reusability
- service autonomy
- service statelessness
- service discoverability

Chapters 11 through 15 discuss service modeling and design in detail and provide guidelines to ensure that these important principles are taken into consideration when building services for use within SOA.

These processes further emphasize the other four principles as well—though they may be automatically implemented through the use of Web services, that does not mean they will necessarily be properly realized. For example, the fact that Web services require the use of service contracts has no bearing on how well individual service descriptions are designed.

SUMMARY OF KEY POINTS

- Service abstraction, composability, loose coupling, and the need for service contracts are native characteristics of Web services that are in full alignment with the corresponding principles of service-orientation.
- Service reusability, autonomy, statelessness, and discoverability are not automatically provided by Web services. Realizing these qualities requires a conscious modeling and design effort.

About the Author

Thomas Erl is an independent consultant with XMLTC Consulting in Vancouver, Canada. His previous book, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, became the top-selling book of 2004 in both Web Services and SOA categories. This guide addresses numerous integration issues and provides strategies and best practices for transitioning toward SOA.

Thomas is a member of OASIS and is active in related research efforts, such as the XML & Web Services Integration Framework (XWIF). He is a speaker and instructor for private and public events and conferences, and has published numerous papers, including articles for the *Web Services Journal*, *WLDJ*, and *Application Development Trends*.

For more information, visit `http://www.thomaserl.com/technology/`.

About SOA Systems

SOA Systems Inc. is a consulting firm actively involved in the research and development of service-oriented architecture, service-orientation, XML, and Web services standards and technology. Through its research and enterprise solution projects SOA Systems has developed a recognized methodology for integrating and realizing service-oriented concepts, technology, and architecture.

For more information, visit `www.soasystems.com`.

One of the consulting services provided by SOA Systems is comprehensive SOA transition planning and the objective assessment of vendor technology products.

For more information, visit `www.soaplanning.com`.

The content in this book is the basis for a series of SOA seminars and workshops developed and offered by SOA Systems.

For more information, visit `www.soatraining.com`.