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.

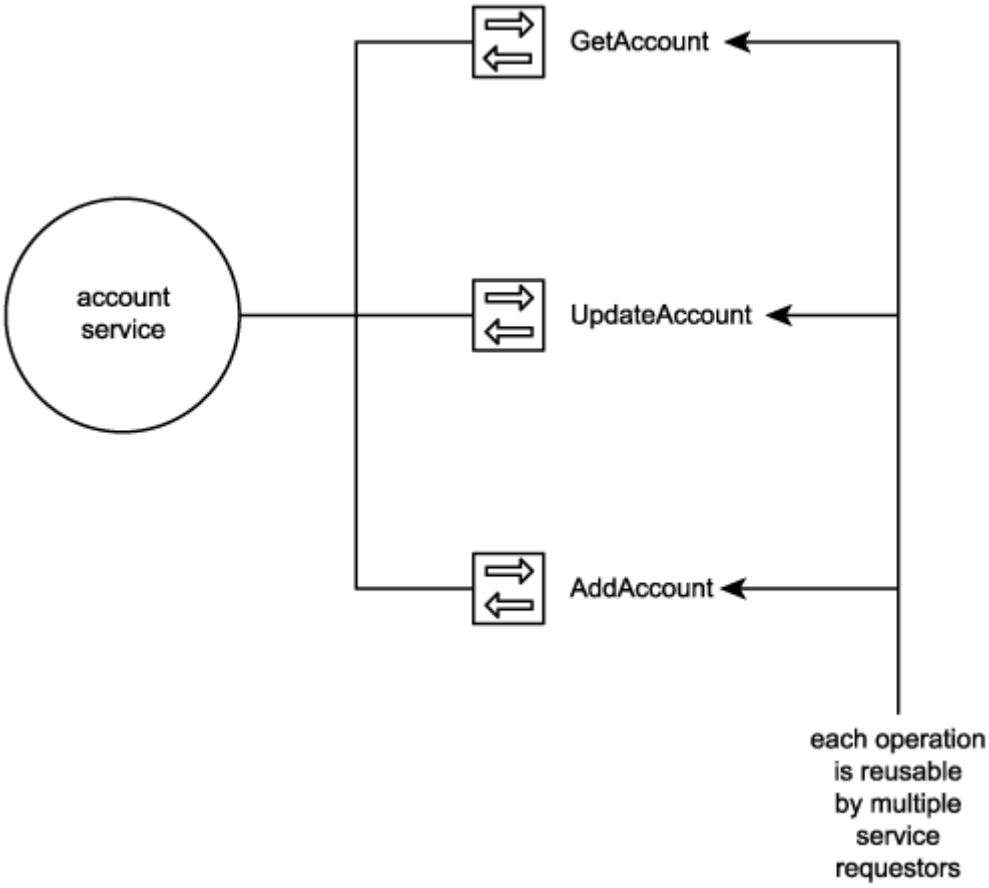
To fully understand how service-orientation principles shape service-oriented architecture, we need to explore the implications their application will have on all of the primary parts that comprise SOA. Let's take a closer look at each of the principles.

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.



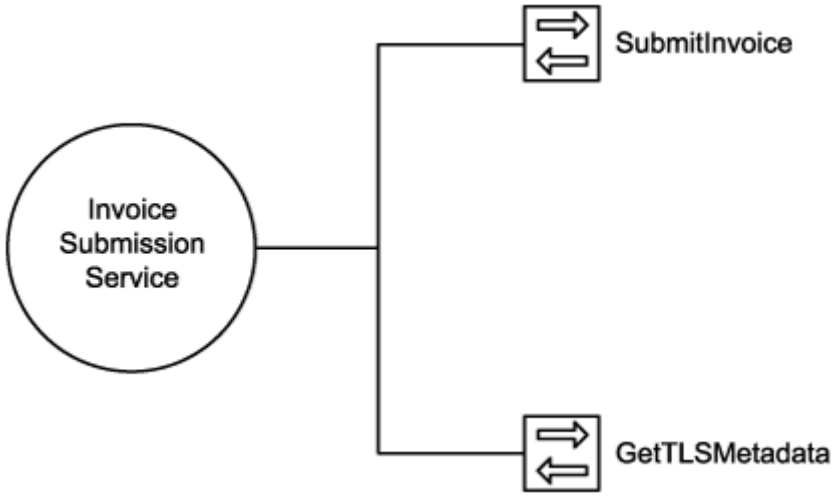
Messaging also indirectly supports service reusability through the use of SOAP headers. These allow for messages to become increasingly self-reliant by grouping metadata details with message content into a single package (the SOAP envelope). Messages can be equipped with processing instructions and business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

Figure 8.14. The RailCo Invoice Submission Service and its operations.



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.

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.

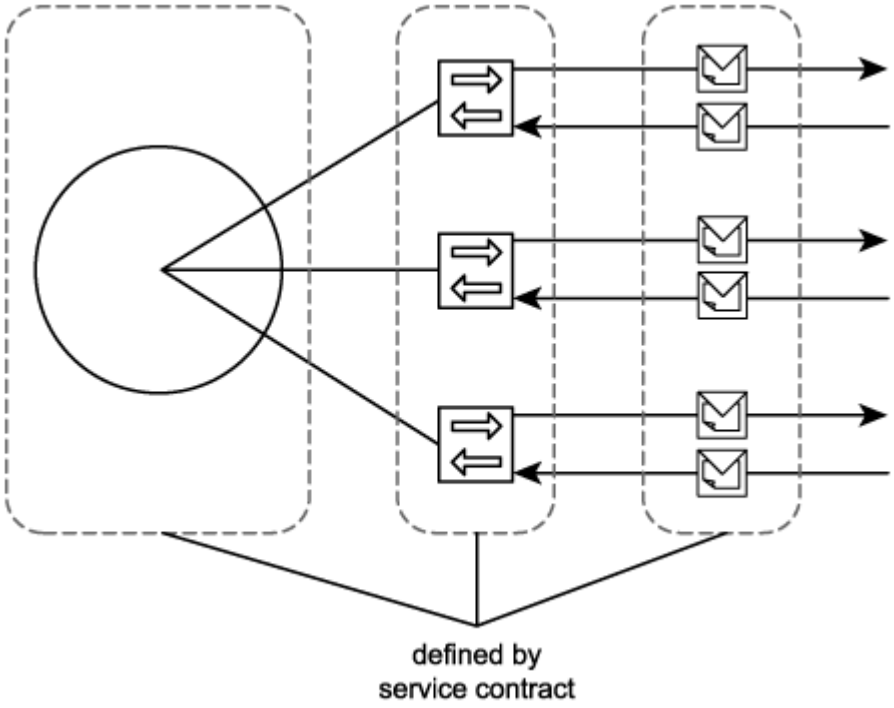
8.3.2. Services share a formal contract

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.

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



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.

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.

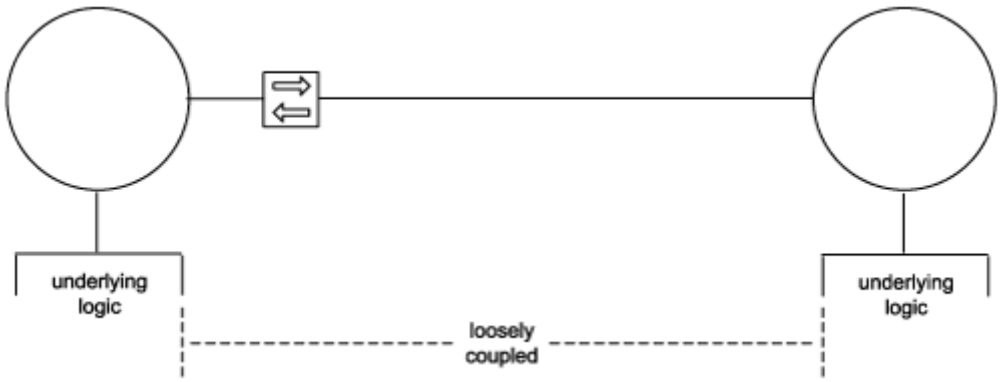
The completed application form is much like a contract between the service provider and the requestor of the service. Upon accepting the form, the service provider agrees to act on the request.

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).

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

[View full size image]



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.

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.

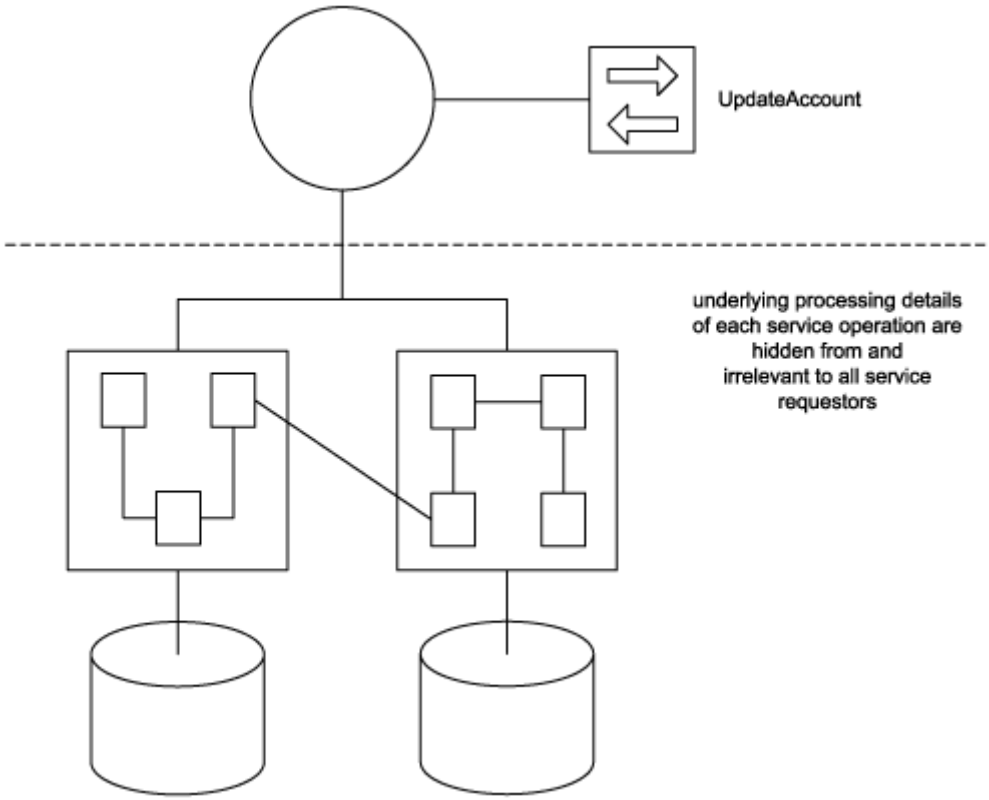
8.3.4. Services abstract underlying logic

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.

[View full size image]



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.

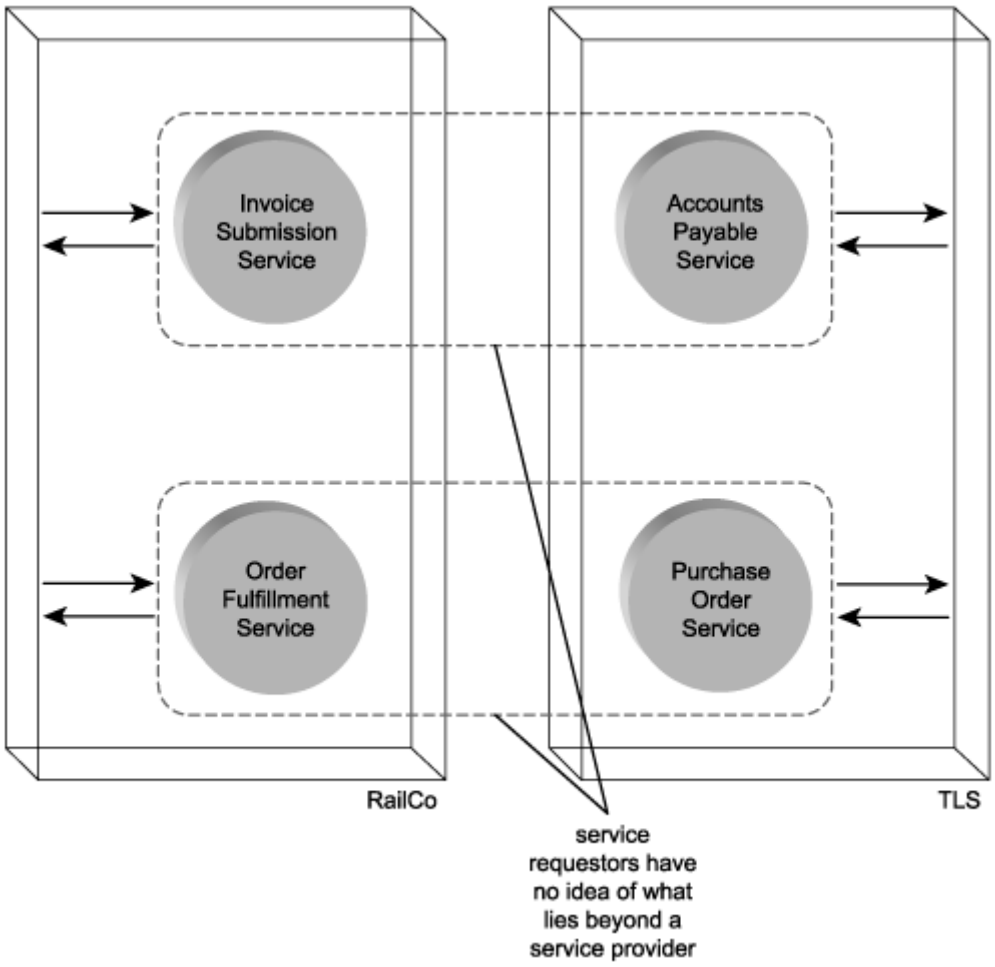
Service interface-level abstraction is one of the inherent qualities provided by Web services. The loosely coupled communications structure requires that the only piece of knowledge services need to interact is each others' service descriptions.

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.

[View full size image]



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.

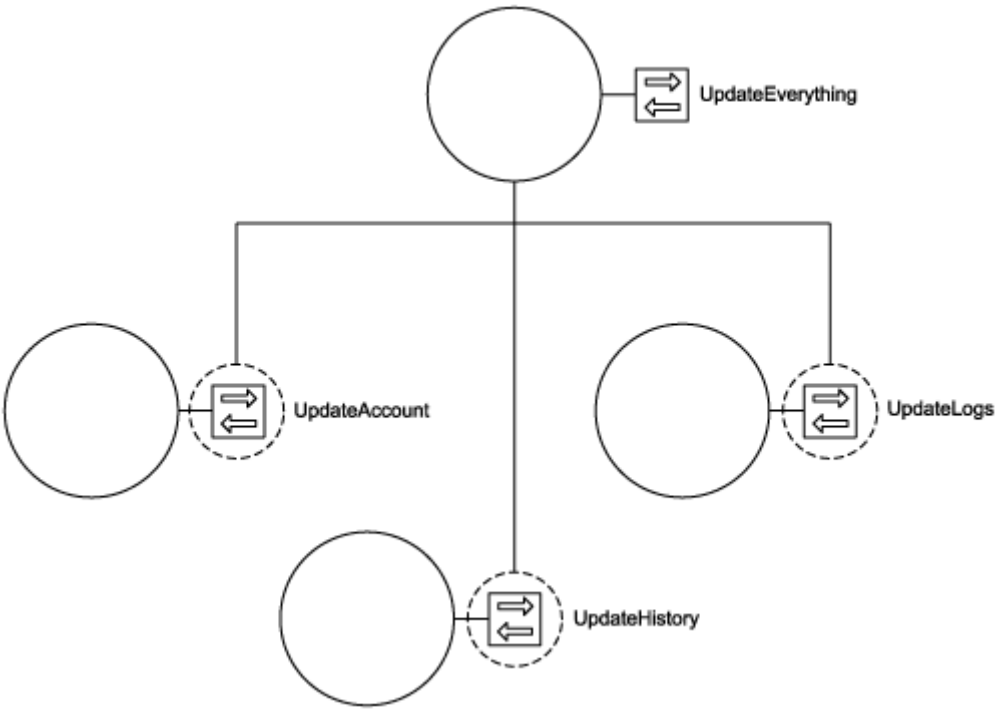
These and other tasks are performed completely unbeknownst to us. We don't know or necessarily care what the Business License Office needs to do to process our application. We are just interested in the expected outcome: the issuance of our license.

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.

[View full size image]



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.

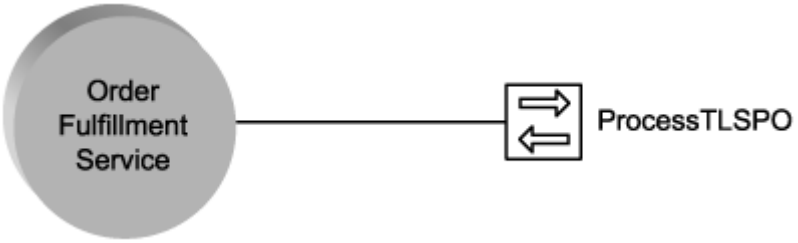
The requirement for any service to be composable also places an emphasis on the design of service operations. Composability is simply another form of reuse, and therefore operations need to be designed in a standardized manner and with an appropriate level of granularity to maximize composition opportunities.

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.

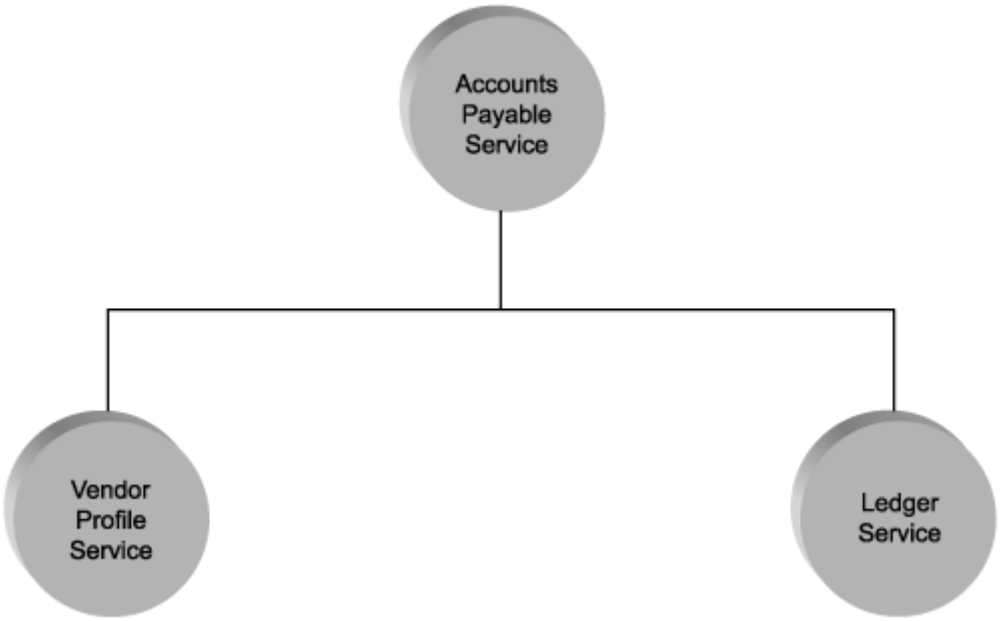
Figure 8.20. The RailCo Order Fulfillment Service with its one operation.



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.

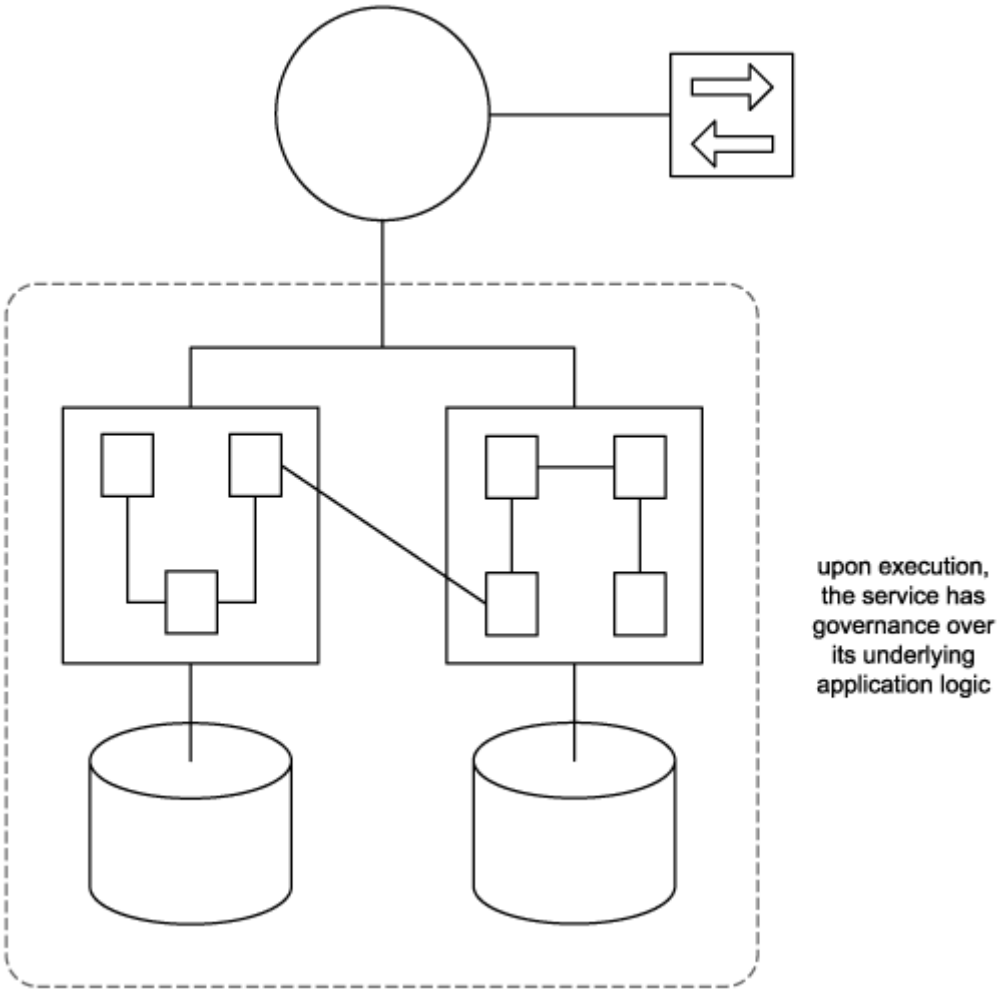
As part of its many tasks, this office takes care of revising the business license information for the affected company. It does so by enlisting the Business License Office and requesting that they issue a new business license for a particular organization.

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:

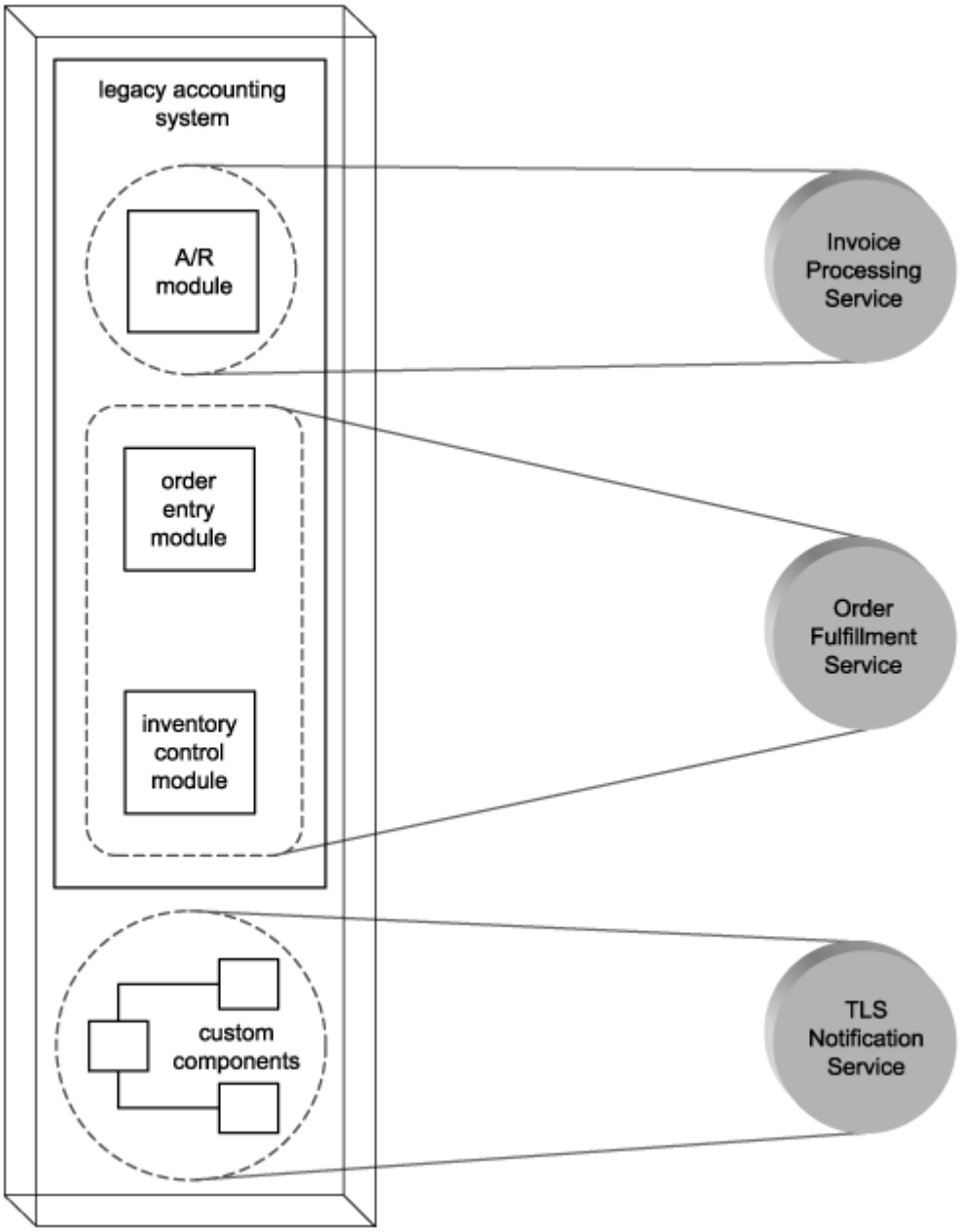
- Invoice Submission Service
- Order Fulfillment Service
- TLS Subscription Service

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.

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



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.)

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

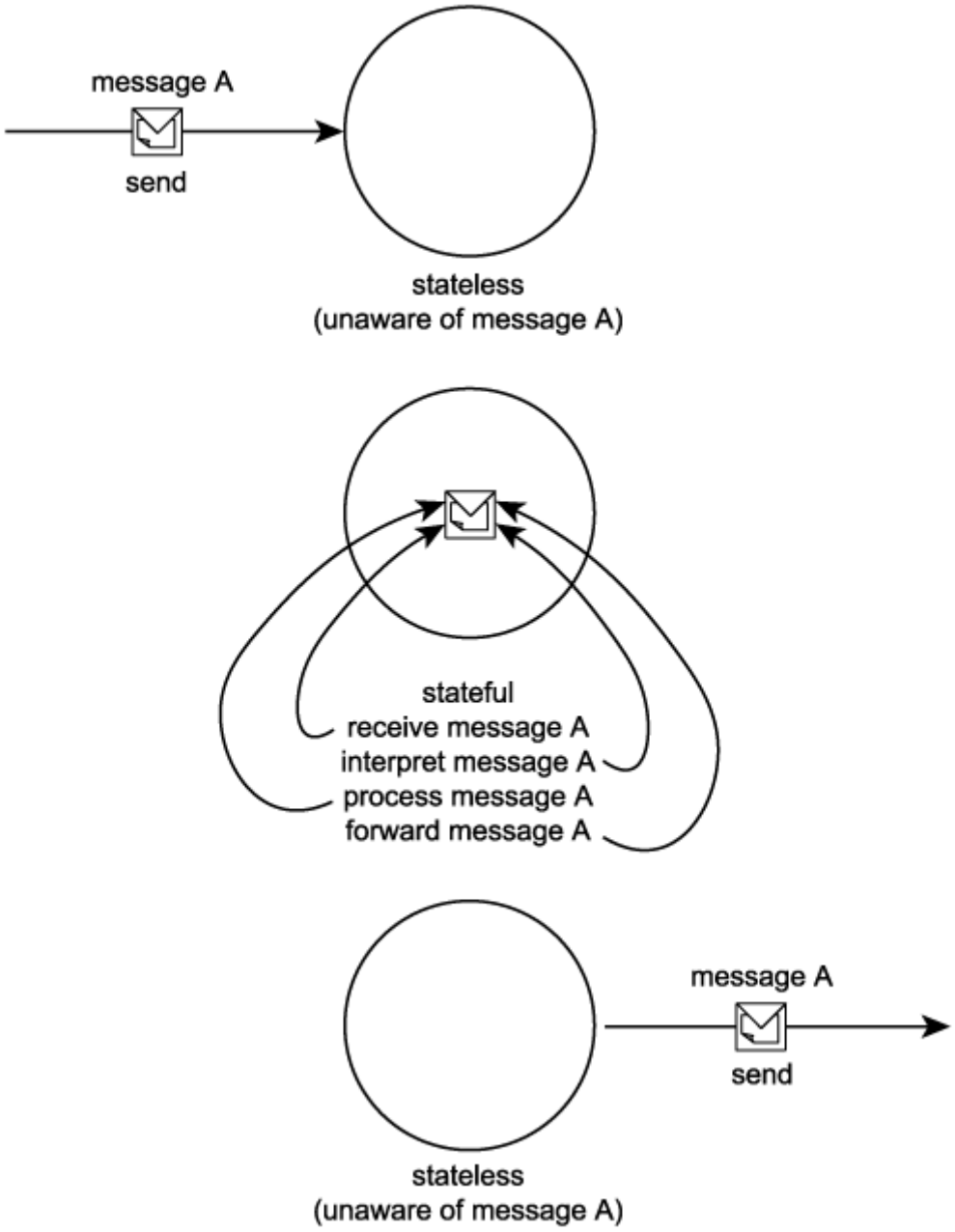
The Business License Office owns the corporate name database required to perform a name check. Also the office has personnel dedicated to visiting and verifying business site locations. When completing these two tasks, the Business License Office therefore has complete self-governance. However, when 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.

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



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.

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 runtime 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

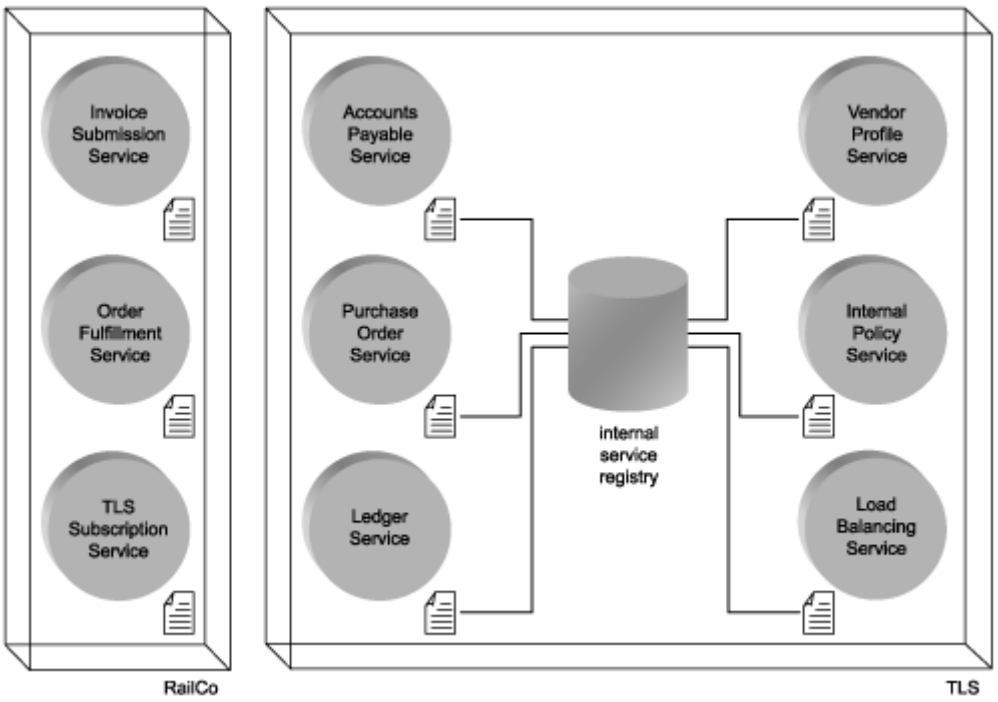
RailCo provides no means of discovery for its services, either internally or to the outside world. Though outfitted with its own WSDL definition and fully capable of acting as a service provider, the Invoice Submission Service is primarily utilized as a service requestor and currently expects no communication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

[View full size image]



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.