Service descriptions (with WSDL)

When we covered loose coupling in Chapter 3 (See 8.1) as part of our primitive SOA discussion, we introduced the essential need for service descriptions. This part of SOA provides the key ingredient to establishing a consistently loosely coupled form of communication between services implemented as Web services.

For this purpose, description documents are required to accompany any service wanting to act as an ultimate receiver. The primary service description document is the WSDL definition (Figure 5.14).

Figure 5.14. WSDL definitions enable loose coupling between services.

- Service endpoints and service descriptions

A WSDL describes the point of contact for a service provider, also known as the service endpoint or just endpoint. It provides a formal definition of the endpoint interface (so that requestors wishing to communicate with the service provider know exactly how to structure request messages) and also establishes the physical location (address) of the service.

Let's dig a bit deeper into how the service description document itself is organized. A WSDL service description (also known as WSDL service definition or just WSDL definition) can be separated into two categories:

- abstract description
- concrete description

Figure 5.16 shows how these individual descriptions comprise a WSDL definition. Note the logical hierarchy established within the parts of the abstract definition. We will explain each of these parts shortly.
Figure 5.16. WSDL document consisting of abstract and concrete parts that collectively describe a service endpoint.

Abstract description

An abstract description establishes the interface characteristics of the Web service without any reference to the technology used to host or enable a Web service to transmit messages. By separating this information, the integrity of the service description can be preserved regardless of what changes might occur to the underlying technology platform.

Below is a description of the three main parts that comprise an abstract description.

portType, operation, and message

The parent portType section of an abstract description provides a high-level view of the service interface by sorting the messages a service can process into groups of functions known as operations.

Each operation represents a specific action performed by the service. A service operation is comparable to a public method used by components in traditional distributed applications. Much like component methods, operations also have input and output parameters. Because Web services rely exclusively on messaging-based communication, parameters are represented as messages. Therefore, an operation consists of a set of input and output messages.
Note that the transmission sequence of these messages can be governed by a predetermined message exchange pattern that also is associated with the operation. (Message exchange patterns are discussed in Chapter 6.)

**Note**

The term "portType" is being renamed to "interface" in version 2.0 of the WSDL specification.

### 5.3.3. Concrete description

For a Web service to be able to execute any of its logic, it needs for its abstract interface definition to be connected to some real, implemented technology. Because the execution of service application logic always involves communication, the abstract Web service interface needs to be connected to a physical transport protocol. This connection is defined in the *concrete description* portion of the WSDL file, which consists of three related parts: **binding, port, and service**

A WSDL description's *binding* describes the requirements for a service to establish physical connections or for connections to be established with the service. In other words, a binding represents one possible transport technology the service can use to communicate. SOAP is the most common form of binding, but others also are supported. A binding can apply to an entire interface or just a specific operation.

Related to the binding is the *port*, which represents the physical address at which a service can be accessed with a specific protocol. This piece of physical implementation data exists separately to allow location information to be maintained independently from other aspects of the concrete description. Within the WSDL language, the term *service* is used to refer to a group of related endpoints.

**Note**

. **Metadata and service contracts**

So far we've established that the abstract and concrete descriptions provided by a WSDL definition express technical information as to how a service can be interfaced with and what type of data exchange it supports.

WSDL definitions frequently rely on XSD schemas to formalize the structure of incoming and outgoing messages. Another common supplemental service description document is a
policy. Policies can provide rules, preferences, and processing details above and beyond what is expressed through the WSDL and XSD schema documents. (Policies are explained in Chapter 7 (See 9.2).)

So now we have up to three separate documents that each describe an aspect of a service:

- WSDL definition
- XSD schema
- policy

Each of these three service description documents can be classified as service metadata, as each provides information about the service. Service description documents can be collectively viewed as establishing a service contract—a set of conditions that must be met and accepted by a potential service requestor to enable successful communication.

Note that a service contract can refer to additional documents or agreements not expressed by service descriptions. For example, a Service Level Agreement (SLA) agreed upon by the respective owners of a service provider and its requestor can be considered part of an overall service contract (Figure 5.17).

Figure 5.17. A service contract comprised of a collection of service descriptions and possibly additional documents.

Semantic descriptions

Most of the metadata currently provided by services focuses on expressing technical information related to data representation and processing requirements. However, these
service description documents generally do not prove useful in explaining details about a service's behavioral characteristics. In fact, the most challenging part of providing a complete description of a Web service is in communicating its semantic qualities.

Examples of service semantics include:

- how a service behaves under certain conditions
- how a service will respond to a specific condition
- what specific tasks the service is most suited for

Most of the time service semantics are assessed by humans, either verbally by discussing the qualities of a service with its owner, or by reading supplementary documentation published alongside service descriptions. The ultimate goal is to provide sufficient semantic information in a structured manner so that, in some cases, service requestors can go as far as to evaluate and choose suitable service providers independently.

Semantic information is usually of greater importance when dealing with external service providers, where your knowledge of another party's service is limited to the information the service owner decides to publish. But even within organizational boundaries, semantic characteristics tend to take on greater relevance as the amount of internal Web services grows.

Although service policies can be designed to express preferences and assertions that communicate aspects of service behavior, efforts are currently underway (primarily by the W3C) to continually extend the semantic information provided by service description documents. For the time being, we must focus on the service description capabilities offered to us through WSDL definitions, XSD schemas, and policies.

### 5.3.6. Service description advertisement and discovery

As we've established, the sole requirement for one service to contact another is access to the other service's description. As the amount of services increases within and outside of organizations, mechanisms for advertising and discovering service descriptions may become necessary. For example, central directories and registries become an option to keep track of the many service descriptions that become available. These repositories allow humans (and even service requestors) to:

- locate the latest versions of known service descriptions
- discover new Web services that meet certain criteria

When the initial set of Web services standards emerged, this eventuality was taken into account. This is why UDDI formed part of the first generation of Web services standards. Though not yet commonly implemented, UDDI provides us with a registry model worth describing.

#### Private and public registries

UDDI specifies a relatively accepted standard for structuring registries that keep track of service descriptions ([Figure 5.18](#)). These registries can be searched manually and accessed programmatically via a standardized API.
Public registries accept registrations from any organizations, regardless of whether they have Web services to offer. Once signed up, organizations acting as service provider entities can register their services.

Private registries can be implemented within organization boundaries to provide a central repository for descriptions of all services the organization develops, leases, or purchases.

Following are descriptions of the primary parts that comprise UDDI registry records.

Business entities and business services

Each public registry record consists of a business entity containing basic profile information about the organization (or service provider entity). Included in this record are one or more business service areas, each of which provides a description of the services offered by the business entity. Business services may or may not be related to the use of Web services.

Binding templates and tModels

You might recall that WSDL definitions stored implementation information separately from the actual interface design. This resulted in an interface definition that existed independently from the transport protocols to which it was eventually bound. Registry records follow the same logic in that they store binding information in a separate area, called the binding template.

Each business service can reference one or more binding templates. The information contained in a binding template may or may not relate to an actual service. For example, a binding template may simply point to the address of a Web site. However, if a Web service is being represented, then the binding template references a tModel. The tModel section of a UDDI record provides pointers to actual service descriptions (Figure 5.19).

Figure 5.19. The basic structure of a UDDI business entity record.
Messaging (with SOAP)

Because all communication between services is message-based, the messaging framework chosen must be standardized so that all services, regardless of origin, use the same format and transport protocol. Additionally, within SOAs, so much emphasis is placed on a message-centric application design that an increasing amount of business and application logic is embedded into messages. In fact, the receipt of a message by a service is the most fundamental action within SOA and the sole action that initiates service-oriented automation. This further demands that the messaging framework be extremely flexible and highly extensible.

The SOAP specification was chosen to meet all of these requirements and has since been universally accepted as the standard transport protocol for messages processed by Web services. Since its initial release, SOAP has been further revised to accommodate more sophisticated message structures in support of enterprise distributed applications and enterprise SOAs.
Even though it was originally named the Simple Object Access Protocol, the SOAP specification's main purpose is to define a standard message format. The structure of this format is quite simple, but its ability to be extended and customized has positioned SOAP messaging as the driving force behind many of the most significant features of contemporary SOAs. This section takes a closer look at the details of the SOAP message format.

**Note**

As of version 1.2 of the SOAP specification, the word "SOAP" is no longer an acronym that stands for "Simple Object Access Protocol." It is now considered a standalone term.

**Envelope, header, and body**

Every SOAP message is packaged into a container known as an *envelope*. Much like the metaphor this conjures up, the envelope is responsible for housing all parts of the message (Figure 5.21).

**Figure 5.21. The basic structure of a SOAP message.**

Each message can contain a *header*, an area dedicated to hosting meta information. In most service-oriented solutions, this header section is a vital part of the overall architecture,
and though optional, it is rarely omitted. Its importance relates to the use of header blocks through which numerous extensions can be implemented (as described next).

The actual message contents are hosted by the message body, which typically consists of XML formatted data. The contents of a message body are often referred to as the message payload.

Header blocks

A primary characteristic of the SOAP communications framework used by SOAs is an emphasis on creating messages that are as intelligence-heavy and self-sufficient as possible. This results in SOAP messages achieving a level of independence that increases the robustness and extensibility of this messaging framework qualities that are extremely important when relying on communication within the loosely coupled environment that Web services require.

Message independence is implemented through the use of header blocks, packets of supplementary meta information stored in the envelope’s header area. Header blocks outfit a message with all of the information required for any services with which the message comes in contact to process and route the message in accordance with its accompanying rules, instructions, and properties. What this means is that through the use of header blocks, SOAP messages are capable of containing a large variety of supplemental information related to the delivery and processing of message contents.

This alleviates services from having to store and maintain message-specific logic. It further reinforces the characteristics of contemporary SOA related to fostering reuse, interoperability, and composability. Web services can be designed with generic processing functionality driven by various types of meta information the service locates in the header blocks of the messages it receives.

The use of header blocks has elevated the Web services framework to an extensible and composable enterprise-level computing platform. Practically all WS-* extensions are implemented using header blocks. (Chapter 17 (See 12.5) provides various examples of what SOAP headers look like.)

Examples of the types of features a message can be outfitted with using header blocks include:

- processing instructions that may be executed by service intermediaries or the ultimate receiver
- routing or workflow information associated with the message
- security measures implemented in the message
- reliability rules related to the delivery of the message
- context and transaction management information
- correlation information (typically an identifier used to associate a request message with a response message)
- **Message styles**
  - The SOAP specification was originally designed to replace proprietary RPC protocols by allowing calls between distributed components to be serialized into XML documents, transported, and then deserialized into the native component format
upon arrival. As a result, much in the original version of this specification centered around the structuring of messages to accommodate RPC data.

- This *RPC-style* message runs contrary to the emphasis SOA places on independent, intelligence-heavy messages. SOA relies on *document-style* messages to enable larger payloads, coarser interface operations, and reduced message transmission volumes between services.

**Attachments**

- To facilitate requirements for the delivery of data not so easily formatted into an XML document, the use of *SOAP attachment* technologies exist. Each provides a different encoding mechanism used to bundle data in its native format with a SOAP message. SOAP attachments are commonly employed to transport binary files, such as images.

- **Faults**

- Finally, SOAP messages offer the ability to add exception handling logic by providing an optional *fault* section that can reside within the body area. The typical use for this section is to store a simple message used to deliver error condition information when an exception occurs.

- **Nodes**

- Although Web services exist as self-contained units of processing logic, they are reliant upon a physical communications infrastructure to process and manage the exchange of SOAP messages. Every major platform has its own implementation of a SOAP communications server, and as a result each vendor has labeled its own variation of this piece of software differently. In abstract, the programs that services use to transmit and receive SOAP messages are referred to as *SOAP nodes* (*Figure 5.23*).

- **Figure 5.23.** A SOAP node transmitting a SOAP message received by the service logic.
As with the services that use them, the underlying SOAP nodes are given labels that identify their type, depending on what form of processing they are involved with in a given message processing scenario.

Below is a list of type labels associated with SOAP nodes (in accordance with the standard SOAP Processing Model). You’ll notice that these names are very similar to the Web service roles we discussed at the beginning of this chapter. The SOAP specification has a different use for the term "role" and instead refers to these SOAP types or labels as concepts.

- **SOAP sender** a SOAP node that transmits a message
- **SOAP receiver** a SOAP node that receives a message
- **SOAP intermediary** a SOAP node that receives and transmits a message, and optionally processes the message prior to transmission
- **initial SOAP sender** the first SOAP node to transmit a message
- **ultimate SOAP receiver** the last SOAP node to receive a message

**SOAP intermediaries**

The same way service intermediaries transition through service provider and service requestor roles, SOAP intermediary nodes move through SOAP receiver and SOAP sender types when processing a message (Figure 5.25).

**Figure 5.25. Different types of SOAP nodes involved with processing a message**

SOAP nodes acting as intermediaries can be classified as forwarding or active. When a SOAP node acts as a forwarding intermediary, it is responsible for relaying the contents of a message to a subsequent SOAP node. In doing so, the intermediary will often process and alter header block information relating to the forwarding logic it is executing. For example, it
will remove a header block it has processed, as well as any header blocks that cannot be relayed any further.

*Active intermediary* nodes are distinguished by the type of processing they perform above and beyond forwarding-related functions. An active intermediary is not required to limit its processing logic to the rules and instructions provided in the header blocks of a message it receives. It can alter existing header blocks, insert new ones, and execute a variety of supporting actions.

### 5.4.3. Message paths

A *message path* refers to the route taken by a message from when it is first sent until it arrives at its ultimate destination. Therefore, a message path consists of at least one initial sender, one ultimate receiver, and zero or more intermediaries (Figure 5.26). Mapping and modeling message paths becomes an increasingly important exercise in SOAs, as the amount of intermediary services tends to grow along with the expansion of a service-oriented solution. Design considerations relating to the path a message is required to travel often center around performance, security, context management, and reliable messaging concerns.

**Figure 5.26. A message path consisting of three Web services.**

---

**Message exchange patterns**

Every task automated by a Web service can differ in both the nature of the application logic being executed and the role played by the service in the overall execution of the business task. Regardless of how complex a task is, almost all require the transmission of multiple messages. The challenge lies in coordinating these messages in a particular sequence so

---
that the individual actions performed by the message are executed properly and in alignment with the overall business task (Figure 6.2).

**Figure 6.2. Not all message exchanges require both requests and responses.**

Message exchange patterns (MEPs) represent a set of templates that provide a group of already mapped out sequences for the exchange of messages. The most common example is a request and response pattern. Here the MEP states that upon successful delivery of a message from one service to another, the receiving service responds with a message back to the initial requestor.

Many MEPs have been developed, each addressing a common message exchange requirement. It is useful to have a basic understanding of some of the more important MEPs, as you will no doubt be finding yourself applying MEPs to specific communication requirements when designing service-oriented solutions.

**Primitive MEPs**

Before the arrival of contemporary SOA, messaging frameworks were already well used by various messaging-oriented middleware products. As a result, a common set of **primitive MEPs** has been in existence for some time.

**Request-response**

This is the most popular MEP in use among distributed application environments and the one pattern that defines synchronous communication (although this pattern also can be applied asynchronously).

The **request-response MEP** (Figure 6.3) establishes a simple exchange in which a message is first transmitted from a source (service requestor) to a destination (service provider). Upon receiving the message, the destination (service provider) then responds with a message back to the source (service requestor).

**Figure 6.3. The request-response MEP.**
Fire-and-forget

This simple asynchronous pattern is based on the unidirectional transmission of messages from a source to one or more destinations (Figure 6.5).

Figure 6.5. The fire-and-forget MEP.

variations of the fire-and-forget MEP exist, including:

- The single-destination pattern, where a source sends a message to one destination only.
- The multi-cast pattern, where a source sends messages to a predefined set of destinations.
- The broadcast pattern, which is similar to the multi-cast pattern, except that the message is sent out to a broader range of recipient destinations.

The fundamental characteristic of the fire-and-forget pattern is that a response to a transmitted message is not expected.

Complex MEPs

Even though a message exchange pattern can facilitate the execution of a simple task, it is really more of a building block intended for composition into larger patterns. Primitive MEPs can be assembled in various configurations to create different types of messaging models, sometimes called complex MEPs.

A classic example is the publish-and-subscribe model. Although we explain publish-and-subscribe approaches in more detail in Chapter 7 (See 9.2), let's briefly discuss it here as an example of a complex MEP.

The publish-and-subscribe pattern introduces new roles for the services involved with the message exchange. They now become publishers and subscribers, and each may be involved in the transmission and receipt of messages. This asynchronous MEP accommodates a requirement for a publisher to make its messages available to a number of subscribers interested in receiving them.

The steps involved are generally similar to the following:

Step 1. The subscriber sends a message to notify the publisher that it wants to receive
messages on a particular topic.

Step 2. Upon the availability of the requested information, the publisher broadcasts messages on the particular topic to all of that topic's subscribers.

This pattern is a great example of how to aggregate primitive MEPs, as shown in Figure 6.7 and explained here:

- Step 1 in the publish-and-subscribe MEP could be implemented by a request-response MEP, where the subscriber’s request message, indicating that it wants to subscribe to a topic, is responded to by a message from the publisher, confirming that the subscription succeeded or failed.
- Step 2 then could be supported by one of the fire-and-forget patterns, allowing the publisher to broadcast a series of unidirectional messages to subscribers.

Figure 6.7. The publish-and-subscribe messaging model is a composite of two primitive MEPs.

On its own, the SOAP standard provides a messaging framework designed to support single-direction message transfer. The extensible nature of SOAP allows countless messaging characteristics and behaviors (MEP-related and otherwise) to be implemented via SOAP header blocks. The SOAP language also provides an optional parameter that can be set to identify the MEP associated with a message. (Note that SOAP MEPs also take SOAP message compliance into account.)

6.1.3. MEPs and WSDL
Operations defined within service descriptions are comprised, in part, of message definitions. The exchange of these messages constitutes the execution of a task represented by an operation. MEPs play a larger role in WSDL service descriptions as they can coordinate the input and output messages associated with an operation. The association of MEPs to WSDL operations thereby embeds expected conversational behavior into the interface definition.

WSDL operations support different configurations of incoming, outgoing, and fault messages. These configurations are equivalent to message exchange patterns, but within the WSDL specification, they often are referred to simply as patterns. It is important to note that WSDL definitions do not restrict an interface to these patterns; they are considered minimal conversational characteristics that can be extended.

Release 1.1 of the WSDL specification provides support for four message exchange patterns that roughly correspond to the MEPs we described in the previous section. These patterns are applied to service operations from the perspective of a service provider or endpoint. In WSDL 1.1 terms, they are represented as follows:

- **Request-response operation** Upon receiving a message, the service must respond with a standard message or a fault message.
- **Solicit-response operation** Upon submitting a message to a service requestor, the service expects a standard response message or a fault message.
- **One-way operation** The service expects a single message and is not obligated to respond.
- **Notification operation** The service sends a message and expects no response.

Of these four patterns (also illustrated in Figure 6.9), only the request-response operation and one-way operation MEPs are recommended by the WS-I Basic Profile.

**Figure 6.9. The four basic patterns supported by WSDL 1.1.**

Not only does WSDL support most traditional MEPs, recent revisions of the specification have extended this support to include additional variations. Specifically, release 2.0 of the WSDL specification extends MEP support to eight patterns (and also changes the terminology) as follows.
• The *in-out pattern*, comparable to the request-response MEP (and equivalent to the WSDL 1.1 request-response operation).
• The *out-in pattern*, which is the reverse of the previous pattern where the service provider initiates the exchange by transmitting the request. (Equivalent to the WSDL 1.1 solicit-response operation.)
• The *in-only pattern*, which essentially supports the standard fire-and-forget MEP. (Equivalent to the WSDL 1.1 one-way operation.)
• The *out-only pattern*, which is the reverse of the in-only pattern. It is used primarily in support of event notification. (Equivalent to the WSDL 1.1 notification operation.)
• The *robust in-only pattern*, a variation of the in-only pattern that provides the option of launching a fault response message as a result of a transmission or processing error.
• The *robust out-only pattern*, which, like the out-only pattern, has an outbound message initiating the transmission. The difference here is that a fault message can be issued in response to the receipt of this message.
• The *in-optional-out pattern*, which is similar to the in-out pattern with one exception. This variation introduces a rule stating that the delivery of a response message is optional and should therefore not be expected by the service requestor that originated the communication. This pattern also supports the generation of a fault message.
• The *out-optional-in pattern* is the reverse of the in-optional-out pattern, where the incoming message is optional. Fault message generation is again supported.

Until version 2.0 of WSDL becomes commonplace, these new patterns will be of limited importance to SOA. Still, it is useful to know in what direction this core standard is heading.

**Atomic transactions**

Transactions have been around for almost as long as automated computer solutions have existed. When managing certain types of corporate data, the need to wrap a series of changes into a single action is fundamental to many business process requirements. *Atomic transactions* implement the familiar commit and rollback features to enable cross-service transaction support (Figure 6.20).

**Figure 6.20.** Atomic transactions apply an all-or-nothing requirement to work performed as part of an activity.
The protocols provided by the WS-AtomicTransaction specification enable cross-service transaction functionality comparable to the ACID-compliant transaction features found in most distributed application platforms.

For those of you who haven't yet worked with ACID transactions, let's quickly recap this important standard. The term "ACID" is an acronym representing the following four required characteristics of a traditional transaction:

- **Atomic** Either all of the changes within the scope of the transaction succeed, or none of them succeed. This characteristic introduces the need for the rollback feature that is responsible for restoring any changes completed as part of a failed transaction to their original state.
- **Consistent** None of the data changes made as a result of the transaction can violate the validity of any associated data models. Any violations result in a rollback of the transaction.
- **Isolated** If multiple transactions occur concurrently, they may not interfere with each other. Each transaction must be guaranteed an isolated execution environment.
- **Durable** Upon the completion of a successful transaction, changes made as a result of the transaction can survive subsequent failures.

### 6.4.2. Atomic transaction protocols

WS-AtomicTransaction is a coordination type, meaning that it is an extension created for use with the WS-Coordination context management framework we covered in the previous section. To participate in an atomic transaction, a service first receives a coordination context from the activation service. It can subsequently register for available atomic transaction protocols.

The following primary transaction protocols are provided:

- A **Completion** protocol, which is typically used to initiate the commit or abort states of the transaction.
- The **Durable 2PC** protocol for which services representing permanent data repositories should register.
- The Volatile 2PC protocol to be used by services managing non-persistent (temporary) data.

Most often these protocols are used to enable a two-phase commit (2PC) that manages an atomic transaction across multiple service participants.

### 6.4.3. The atomic transaction coordinator

When WS-AtomicTransaction protocols are used, the coordinator controller service can be referred to as an **atomic transaction coordinator**. This particular implementation of the WS-Coordination coordinator service represents a specific service model. The atomic transaction coordinator (Figure 6.21) plays a key role in managing the participants of the transaction process and in deciding the transaction's ultimate outcome.

**Figure 6.21. The atomic transaction coordinator service model.**

As previously mentioned, the atomic transaction coordinator is tasked with the responsibility of deciding the outcome of a transaction. It bases this decision on feedback it receives from all of the transaction participants.

The collection of this feedback is separated into two phases. During the **prepare** phase (Figure 6.22), all participants are notified by the coordinator, and each is asked to prepare and then issue a vote. Each participant's vote consists of either a "commit" or "abort" request (Figure 6.23).
After the votes are collected, the atomic transaction coordinator enters the *commit* phase. It now reviews all votes and decides whether to commit or rollback the transaction. The conditions of a commit decision are simple: if all votes are received and if all participants voted to commit, the coordinator declares the transaction successful, and the changes are committed. However, if any one vote requests an abort, or if any of the participants fail to respond, then the transaction is aborted, and all changes are rolled back (Figure 6.24).

**Figure 6.24.** The coordinator aborting the transaction and notifying participants to rollback all changes.

[View full size image]
Atomic transactions and SOA

Much of the transactional functionality implemented in service-oriented solutions is done so among the components that execute an activity on behalf of a single service. However, as more services emerge within an organization and as service compositions become more commonplace, the need to move transaction boundaries into cross-service interaction scenarios increases. Being able to guarantee an outcome of an activity is a key part of enterprise-level computing, and atomic transactions therefore play an important role in ensuring quality of service.

Not only do atomic transactional capabilities lead to a robust execution environment for SOA activities, they promote interoperability when extended into integrated environments. This allows the scope of an activity to span different solutions built with different vendor platforms, while still being assured a guaranteed all-or-nothing outcome. Assuming, of course, that WS-AtomicTransaction is supported by the affected applications, this option broadens the application of the two-phase commit protocol beyond traditional application boundaries (thus, supporting service interoperability). Figure 6.25 illustrates how atomic transactions support these aspects of SOA.

Business activities
Business activities govern long-running, complex service activities. Hours, days, or even weeks can pass before a business activity is able to complete. During this period, the activity can perform numerous tasks that involve many participants.

What distinguishes a business activity from a regular complex activity is that its participants are required to follow specific rules defined by protocols. Business activities primarily differ from the also protocol-based atomic transactions in how they deal with exceptions and in the nature of the constraints introduced by the protocol rules.

For instance, business activity protocols do not offer rollback capabilities. Given the potential for business activities to be long-running, it would not be realistic to expect ACID-type transaction functionality. Instead, business activities provide an optional compensation process that, much like a "plan B," can be invoked when exception conditions are encountered (Figure 6.27).

Figure 6.27. A business activity controls the integrity of a service activity by providing participants with a "plan B" (a compensation).

**Business activity protocols**

As with WS-AtomicTransaction, WS-BusinessActivity is a coordination type designed to leverage the WS-Coordination context management framework. It provides two very similar protocols, each of which dictates how a participant may behave within the overall business activity.

- The BusinessAgreementWithParticipantCompletion protocol, which allows a participant to determine when it has completed its part in the business activity.
- The BusinessAgreementWithCoordinatorCompletion protocol, which requires that a participant rely on the business activity coordinator to notify it that it has no further processing responsibilities.

Business activity participants interact with the standard WS-Coordination coordinator composition to register for a protocol, as was explained in the previous Coordination (See 9.1.3) section.

6.5.2. The business activity coordinator
When its protocols are used, the WS- Coordination controller service assumes a role specific to the coordination type: in this case, it becomes a *business activity coordinator* (Figure 6.28). As explained in the previous section, this coordinator has varying degrees of control in the overall activity, based on the coordination protocols used by the participants.

**Figure 6.28. The business activity coordinator service model**

During the lifecycle of a business activity, the business activity coordinator and the activity participants transition through a series of states. The actual point of transition occurs when special notification messages are passed between these services.

For example, a participant can indicate that it has completed the processing it was required to perform as part of the activity by issuing a *completed notification*. This moves the participant from an *active state* to a *completed state*. The coordinator may respond with a *close* message to let the participant know that the business activity is being successfully completed.

However, if things don't go as planned during the course of a business activity, one of a number of options are available. Participants can enter a *compensation state* during which they attempt to perform some measure of exception handling. This generally invokes a separate compensation process that could involve a series of additional processing steps. A compensation is different from an atomic transaction in that it is not expected to rollback any changes performed by the participating services; its purpose is generally to execute plan B when plan A fails.

Alternatively, a *cancelled state* can be entered. This typically results in the termination of any further processing outside of the *cancellation notifications* that need to be distributed.

What also distinguishes business activities from atomic transactions is the fact that participating services are not required to remain participants for the duration of the activity.
Because there is no tight control over the changes performed by services, they may leave the business activity after their individual contributions have been performed. When doing so, participants enter an *exit* state by issuing an *exit notification* message to the business activity coordinator.

These and other states are defined in a series of state tables documented as part of the WS-BusinessActivity specification. These tables establish the fundamental rules of the business activity protocols by determining the sequence and conditions of allowable states.

### 6.5.4. Business activities and atomic transactions

It is important to note that the use of a business activity does not exclude the use of atomic transactions. In fact, it is likely that a long-running business activity will encompass the execution of several atomic transactions during its lifetime ([Figure 6.29](#)).

**Figure 6.29.** Two atomic transactions residing within the scope of a business activity.
Business activities fully complement the composable nature of SOA (Figure 6.30) by tracking and regulating complex activities while also allowing them to carry on for long periods of time. Service autonomy and statelessness are preserved by permitting services to participate within an activity for only the duration they are absolutely required to. This also allows for the design of highly adaptive business activities wherein the participants can augment activity or process logic to accommodate changes in the business tasks being automated. Through the use of the compensation process, business activities increase SOA's quality of service by providing built-in fault handling logic.

Figure 6.30. A business activity relating to other parts of SOA.

Orchestration

Organizations that already have employed enterprise application integration (EAI) middleware products to automate business processes or to integrate various legacy environments will likely already be familiar with the concept of orchestration. In these systems, a centrally controlled set of workflow logic facilitates interoperability between two or more different applications. A common implementation of orchestration is the hub-and-spoke model that allows multiple external participants to interface with a central orchestration engine.

One of the driving requirements behind the creation of these solutions was to accommodate the merging of large business processes. With orchestration, different processes can be connected without having to redevelop the solutions that originally automated the processes individually. Orchestration bridges this gap by introducing new workflow logic. Further, the use of orchestration can significantly reduce the complexity of solution environments. Workflow logic is abstracted and more easily maintained than when embedded within individual solution components.

The role of orchestration broadens in service-oriented environments. Through the use of extensions that allow for business process logic to be expressed via services, orchestration can represent and express business logic in a standardized, services-based venue. When building service-oriented solutions, this provides an extremely attractive means of housing and controlling the logic representing the process being automated.
Orchestration further leverages the intrinsic interoperability sought by service designs by providing potential integration endpoints into processes. A key aspect to how orchestration is positioned within SOA is the fact that orchestrations themselves exist as services. Therefore, building upon orchestration logic standardizes process representation across an organization, while addressing the goal of enterprise federation and promoting service-orientation.

Figure 6.32. An orchestration controls almost every facet of a complex activity.

A primary industry specification that standardizes orchestration is the Web services Business Process Execution Language (WS-BPEL). This book recognizes WS-BPEL as a key second-generation extension and therefore uses its concepts and terminology as the basis for a number of discussions relating to business process modeling.

Business protocols and process definition

The workflow logic that comprises an orchestration can consist of numerous business rules, conditions, and events. Collectively, these parts of an orchestration establish a business protocol that defines how participants can interoperate to achieve the completion of a business task. The details of the workflow logic encapsulated and expressed by an orchestration are contained within a process definition.

6.6.2. Process services and partner services

Identified and described within a process definition are the allowable process participants. First, the process itself is represented as a service, resulting in a process service (which happens to be another one of our service models, as shown in Figure 6.33).

Figure 6.33. A process service coordinating and exposing functionality from three partner services.
Other services allowed to interact with the process service are identified as partner services or partner links. Depending on the workflow logic, the process service can be invoked by an external partner service, or it can invoke other partner services (Figure 6.34).

Figure 6.34. The process service, after first being invoked by a partner service, then invokes another partner service.

Basic activities and structured activities

WS-BPEL breaks down workflow logic into a series of predefined primitive activities. Basic activities (receive, invoke, reply, throw, wait) represent fundamental workflow actions which can be assembled using the logic supplied by structured activities (sequence, switch, while, flow, pick). How these activities can be used to express actual business process logic is explored in Chapter 16 (See 12.4).

6.6.4. Sequences, flows, and links
Basic and structured activities can be organized so that the order in which they execute is predefined. A sequence aligns groups of related activities into a list that determines a sequential execution order. Sequences are especially useful when one piece of application logic is dependent on the outcome of another.

Flows also contain groups of related activities, but they introduce different execution requirements. Pieces of application logic can execute concurrently within a flow, meaning that there is not necessarily a requirement for one set of activities to wait before another finishes. However, the flow itself does not finish until all encapsulated activities have completed processing. This ensures a form of synchronization among application logic residing in individual flows.

Links are used to establish formal dependencies between activities that are part of flows. Before an activity fully can complete, it must ensure that any requirements established in outgoing links first are met. Similarly, before any linked activity can begin, requirements contained within any incoming links first must be satisfied. Rules provided by links are also referred to as synchronization dependencies.

### 6.6.5. Orchestrations and activities

As we defined earlier, an activity is a generic term that can be applied to any logical unit of work completed by a service-oriented solution. The scope of a single orchestration, therefore, can be classified as a complex, and most likely, long-running activity.

### 6.6.6. Orchestration and coordination

Orchestration, as represented by WS-BPEL, can fully utilize the WS-Coordination context management framework by incorporating the WS-BusinessActivity coordination type. This specification defines coordination protocols designed to support complex, long-running activities.

### 6.6.7. Orchestration and SOA

Business process logic is at the root of automation solutions. Orchestration provides an automation model where process logic is centralized yet still extensible and composable (Figure 6.35). Through the use of orchestrations, service-oriented solution environments become inherently extensible and adaptive. Orchestrations themselves typically establish a common point of integration for other applications, which makes an implemented orchestration a key integration enabler.
These qualities lead to increased organizational agility because:

- The workflow logic encapsulated by an orchestration can be modified or extended in a central location.
- Positioning an orchestration centrally can significantly ease the merging of business processes by abstracting the glue that ties the corresponding automation solutions together.
- By establishing potentially large-scale service-oriented integration architectures, orchestration, on a fundamental level, can support the evolution of a diversely federated enterprise.

**Choreography**

In a perfect world, all organizations would agree on how internal processes should be structured, so that should they ever have to interoperate, they would already have their automation solutions in perfect alignment.

Though this vision has about a zero percent chance of ever becoming reality, the requirement for organizations to interoperate via services is becoming increasingly real and increasingly complex. This is especially true when interoperation requirements extend into the realm of collaboration, where multiple services from different organizations need to work together to achieve a common goal.

The Web Services Choreography Description Language (WS-CDL) is one of several specifications that attempts to organize information exchange between multiple organizations (or even multiple applications within organizations), with an emphasis on public collaboration (Figure 6.37). It is the specification we've chosen here to represent the concept of choreography and also the specification from which many of the terms discussed in this section have been derived.

**Figure 6.37. A choreography enables collaboration between its participants.**
An important characteristic of choreographies is that they are intended for public message exchanges. The goal is to establish a kind of organized collaboration between services representing different service entities, only no one entity (organization) necessarily controls the collaboration logic. Choreographies therefore provide the potential for establishing universal interoperability patterns for common inter-organization business tasks.

**Note**

While the emphasis on choreography is B2B interaction, it also can be applied to enable collaboration between applications belonging to a single organization. The use of orchestration, though, is far more common for this requirement.

### 6.7.2. Roles and participants

Within any given choreography, a Web service assumes one of a number of predefined roles. This establishes what the service does and what the service can do within the context of a particular business task. Roles can be bound to WSDL definitions, and those related are grouped accordingly, categorized as participants (services).

### 6.7.3. Relationships and channels

Every action that is mapped out within a choreography can be broken down into a series of message exchanges between two services. Each potential exchange between two roles in a choreography is therefore defined individually as a relationship. Every relationship consequently consists of exactly two roles.

Now that we've defined who can talk with each other, we require a means of establishing the nature of the conversation. *Channels* do exactly that by defining the characteristics of the message exchange between two specific roles.
Further, to facilitate more complex exchanges involving multiple participants, channel information can actually be passed around in a message. This allows one service to send another the information required for it to be communicated with by other services. This is a significant feature of the WS-CDL specification, as it fosters dynamic discovery and increases the number of potential participants within large-scale collaborative tasks.

6.7.4. Interactions and work units

Finally, the actual logic behind a message exchange is encapsulated within an *interaction*. Interactions are the fundamental building blocks of choreographies because the completion of an interaction represents actual progress within a choreography. Related to interactions are *work units*. These impose rules and constraints that must be adhered to for an interaction to successfully complete.

6.7.5. Reusability, composability, and modularity

Each choreography can be designed in a reusable manner, allowing it to be applied to different business tasks comprised of the same fundamental actions. Further, using an import facility, a choreography can be assembled from independent *modules*. These modules can represent distinct sub-tasks and can be reused by numerous different parent choreographies (*Figure 6.38*).

*Figure 6.38. A choreography composed of two smaller choreographies*
choreography in effect composes a set of non-specific services to accomplish a task, choreographies themselves can be assembled into larger compositions.

6.7.6. Orchestrations and choreographies

While both represent complex message interchange patterns, there is a common distinction that separates the terms "orchestration" and "choreography." An orchestration expresses organization-specific business workflow. This means that an organization owns and controls the logic behind an orchestration, even if that logic involves interaction with external business partners. A choreography, on the other hand, is not necessarily owned by a single entity. It acts as a community interchange pattern used for collaborative purposes by services from different provider entities (Figure 6.39).

Figure 6.39. A choreography enabling collaboration between two different orchestrations.

One can view an orchestration as a business-specific application of a choreography. This view is somewhat accurate, only it is muddled by the fact that some of the functionality provided by the corresponding specifications (WS-CDL and WS-BPEL) actually overlaps. This is a consequence of these specifications being developed in isolation and submitted to separate standards organizations (W3C and OASIS, respectively).

An orchestration is based on a model where the composition logic is executed and controlled in a centralized manner. A choreography typically assumes that there is no single owner of collaboration logic. However, one area of overlap between the current orchestration and choreography extensions is the fact that orchestrations can be designed to include multi-organization participants. An orchestration can therefore effectively establish cross-enterprise activities in a similar manner as a choreography. Again, though, a primary distinction is the fact that an orchestration is generally owned and operated by a single organization.

6.7.7. Choreography and SOA
The fundamental concept of exposing business logic through autonomous services can be applied to just about any implementation scope. Two services within a single organization, each exposing a simple function, can interact via a basic MEP to complete a simple task. Two services belonging to different organizations, each exposing functionality from entire enterprise business solutions, can interact via a basic choreography to complete a more complex task. Both scenarios involve two services, and both scenarios support SOA implementations.

Choreography therefore can assist in the realization of SOA across organization boundaries (Figure 6.40). While it natively supports composability, reusability, and extensibility, choreography also can increase organizational agility and discovery. Organizations are able to join into multiple online collaborations, which can dynamically extend or even alter related business processes that integrate with the choreographies. By being able to pass around channel information, participating services can make third-party organizations aware of other organizations with which they already have had contact.