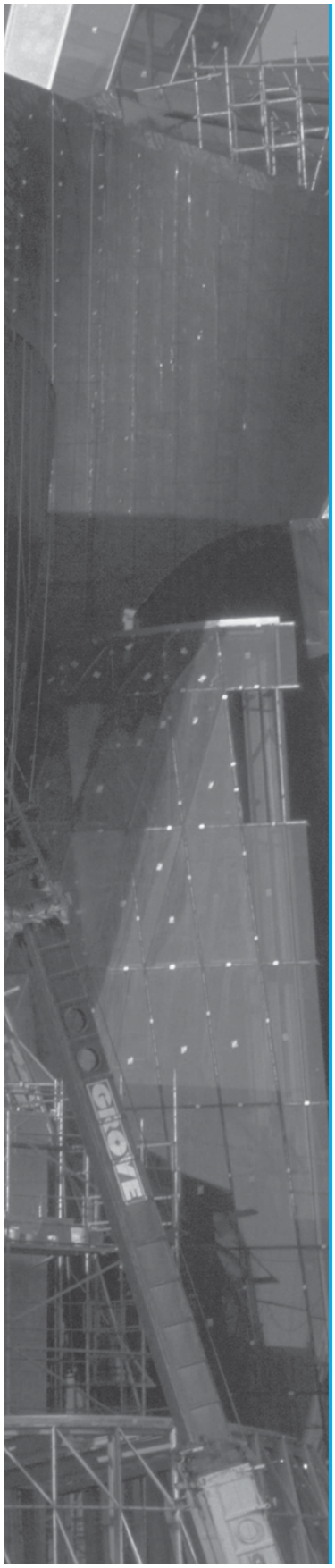

19

Service-oriented architecture

Objectives

The objective of this chapter is to introduce service-oriented software architecture as a way of building distributed applications using web services. When you have read this chapter, you will:

- understand the basic notions of a web service, web service standards, and service-oriented architecture;
- understand the service engineering process that is intended to produce reusable web services;
- have been introduced to the notion of service composition as a means of service-oriented application development;
- understand how business process models may be used as a basis for the design of service-oriented systems.

Contents

The development of the Web in the 1990s revolutionized organizational information exchange. Client computers could gain access to information on remote servers outside their own organizations. However, access was solely through a web browser and direct access to the information by other programs was not practical. This meant that opportunistic connections between servers where, for example, a program queried a number of catalogs from different suppliers, were not possible.

To get around this problem, the notion of a web service was proposed. Using a web service, organizations that wish to make their information accessible to other programs can do so by defining and publishing a web service interface. This interface defines the data available and how it can be accessed. More generally, a web service is a standard representation for some computational or information resource that can be used by other programs. These may be information resources, such as a parts catalog; computer resources, such as a specialized processor; or storage resources. For example, an archive service could be implemented that permanently and reliably stores organizational data that, by law, has to be maintained for many years.

A web service is an instance of a more general notion of a service, which is defined (Lovelock et al., 1996) as:

> *"an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production".*

The essence of a service, therefore, is that the provision of the service is independent of the application using the service (Turner et al., 2003). Service providers can develop specialized services and offer these to a range of service users from different organizations.

Service-oriented architectures (SOAs) are a way of developing distributed systems where the system components are stand-alone services, executing on geographically distributed computers. Standard XML-based protocols, such as SOAP and WSDL, have been designed to support service communication and information exchange. Consequently, services are platform and implementation-language independent. Software systems can be constructed by composing local services and external services from different providers, with seamless interaction between the services in the system.

Figure 19.1 encapsulates the idea of a SOA. Service providers design and implement services and specify the interface to these services. They also publish information about these services in an accessible registry. Service requestors (sometimes called service clients) who wish to make use of a service discover the specification of that service and locate the service provider. They can then bind their application to that specific service and communicate with it, using standard service protocols.

From the outset, there has been an active standardization process for SOA, working alongside technical developments. All of the major hardware and software companies are committed to these standards. As a result, SOA have not

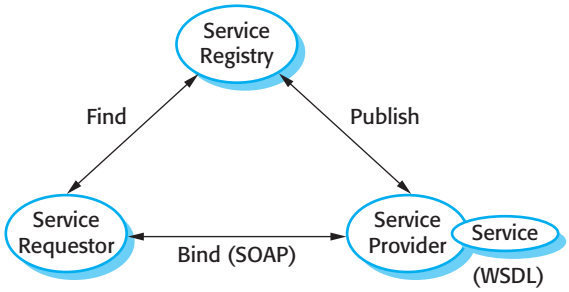

Figure 19.1 Service-oriented architecture

suffered from the incompatibilities that normally arise with technical innovations, where different suppliers maintain their proprietary version of the technology. Figure 19.2 shows the stack of key standards that have been established to support web services. Because of this early standardization, problems, such as the multiple incompatible component models in CBSE, discussed in Chapter 17, have not arisen in service-oriented system development.

Web service protocols cover all aspects of SOAs, from the basic mechanisms for service information exchange (SOAP) to programming language standards (WS-BPEL). These standards are all based on XML, a human and machine-readable notation that allows the definition of structured data where text is tagged with a meaningful identifier. XML has a range of supporting technologies, such as XSD for schema definition, which are used to extend and manipulate XML descriptions. Erl (2004) provides a good summary of XML technologies and their role in web services.

Briefly, the key standards for web SOAs are as follows:

1. *SOAP* This is a message interchange standard that supports the communication between services. It defines the essential and optional components of messages passed between services.
2. *WSDL* The Web Service Definition Language (WSDL) is a standard for service interface definition. It sets out how the service operations (operation names, parameters, and their types) and service bindings should be defined.
3. *WS-BPEL* This is a standard for a workflow language that is used to define process programs involving several different services. I discuss the notion of process programs in Section 19.3.

A service discovery standard, UDDI, was also proposed but this has not been widely adopted. The UDDI (Universal Description, Discovery and Integration) standard defines the components of a service specification, which may be used to discover the existence of a service. These include information about the service provider, the services provided, the location of the WSDL description of the service interface, and information about business relationships. The intention was that this standard would allow companies to set up registries with UDDI descriptions defining the services that they offered.

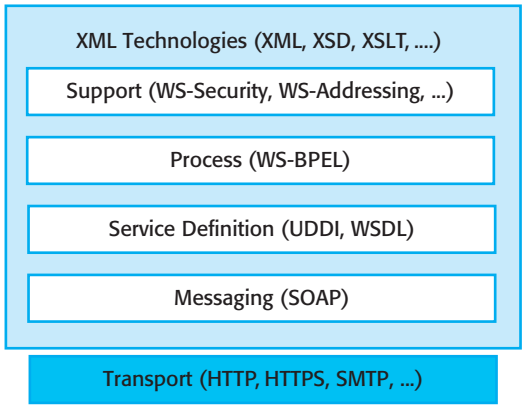

Figure 19.2 Web service standards

A number of companies, such as Microsoft, set up UDDI registries in the early years of the 21st century but these have now all closed. Improvements in search engine technology have made them redundant. Service discovery using a standard search engine to search for appropriately commented WSDL descriptions is now the preferred approach for discovering external services.

The principal SOA standards are supported by a range of supporting standards that focus on more specialized aspects of SOA. There are a very large number of supporting standards because they are intended to support SOA in different types of enterprise application. Some examples of these standards include the following:

1. WS-Reliable Messaging, a standard for message exchange that ensures messages will be delivered once and once only.
2. WS-Security, a set of standards supporting web service security including standards that specify the definition of security policies and standards that cover the use of digital signatures.
3. WS-Addressing, which defines how address information should be represented in a SOAP message.
4. WS-Transactions, which defines how transactions across distributed services should be coordinated.

Web service standards are a huge topic and I don't have space to discuss them in detail here. I recommend Erl's books (2004; 2005) for an overview of these standards. Their detailed descriptions are also available as public documents on the Web.

Current web services standards have been criticized as being 'heavyweight' standards that are over-general and inefficient. Implementing these standards requires a considerable amount of processing to create, transmit, and interpret the associated XML messages. For this reason, some organizations, such as Amazon, use a simpler, more efficient approach to service communication using so-called RESTful services (Richardson and Ruby, 2007). The RESTful approach supports efficient service

RESTful web services

REST (REpresentational State Transfer) is an architectural style based on transferring representations of resources from a server to a client. It is the style that underlies the web as a whole and has been used as a much simpler method than SOAP/WSDL for implementing web services.

A RESTful web service is identified by its URI (Universal Resource identifier) and communicates using the HTML protocol. It responds to HTML methods GET, PUT, POST, and DELETE and returns a resource representation to the client. Simplistically, POST means create, GET means read, PUT means update, and DELETE means delete.

RESTful services involve a lower overhead than so-called 'big web services' and are used by many organizations implementing service-based systems that do not rely on externally provided services.

http://www.SoftwareEngineering-9.com/Web/Services/REST/

interaction but it does not support enterprise-level features such as WS-Reliability and WS-Transactions. Pautasso et al. (2008) compare the RESTful approach with standardized web services.

Building applications based on services allows companies and other organizations to cooperate and make use of each other's business functions. Thus, systems that involve extensive information exchange across company boundaries, such as supply chain systems where one company orders goods from another, can easily be automated. Service-based applications may be constructed by linking services from various providers using either a standard programming language or a specialized workflow language, as discussed in Section 19.3.

SOAs are loosely coupled architectures where service bindings can change during execution. This means that a different, but equivalent version of the service may be executed at different times. Some systems will be solely built using web services and others will mix web services with locally developed components. To illustrate how applications that use a mixture of services and components may be organized, consider the following scenario:

> *An in-car information system provides drivers with information on weather, road traffic conditions, local information, and so forth. This is linked to the car radio so that information is delivered as a signal on a specific radio channel. The car is equipped with GPS receiver to discover its position and, based on that position, the system accesses a range of information services. Information may then be delivered in the driver's specified language.*

Figure 19.3 illustrates a possible organization for such a system. The in-car software includes five modules. These handle communications with the driver, with a GPS receiver that reports the car's position and with the car radio. The Transmitter and Receiver modules handle all communications with external services.

The car communicates with an external mobile information service that aggregates information from a range of other services, providing information on weather,

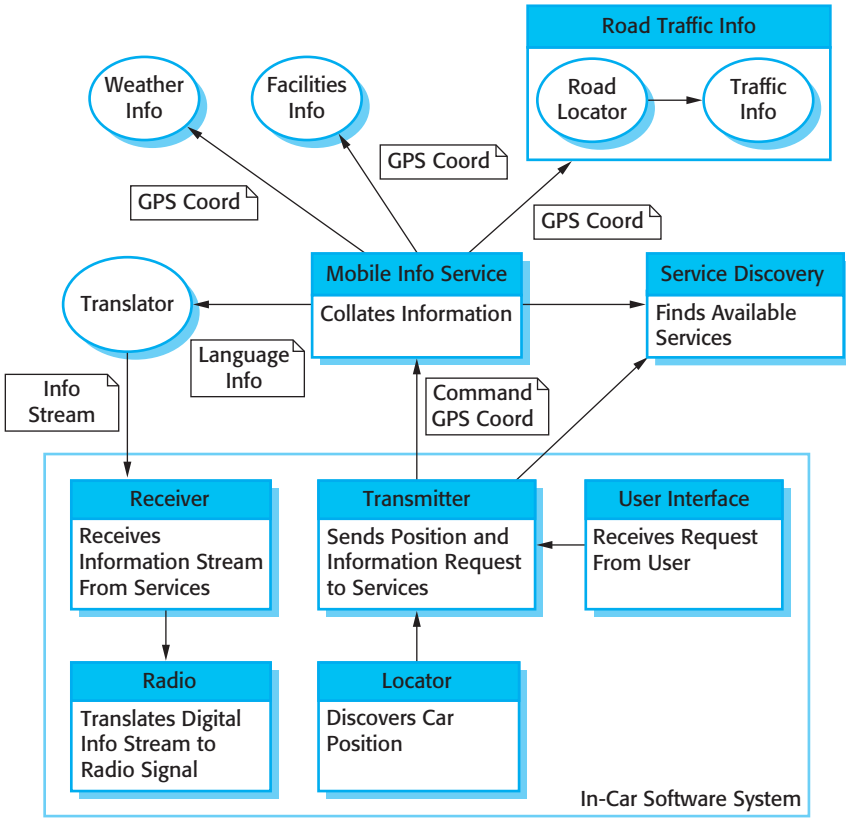

Figure 19.3 A service-based, in-car information system

traffic information, and local facilities. Different providers in different places offer these services, and the in-car system uses a discovery service to locate appropriate information services and bind to them. The discovery service is also used by the mobile information service to bind to the appropriate weather, traffic, and facilities services. Services exchange SOAP messages that include GPS position information used by the services to select the appropriate information. The aggregated information is then sent to the car through a service that translates that information into the driver's preferred language.

This example illustrates one of the key advantages of the service-oriented approach. It is not necessary to decide when the system is programmed or deployed what service provider should be used or what specific services should be accessed. As the car moves around, the in-car software uses the service discovery service to find the most appropriate information service and binds to that. Because of the use of a translation service, it can move across borders and therefore make local information available to people who don't speak the local language.

A service-oriented approach to software engineering is a new software engineering paradigm that is, in my view, as important a development as object-oriented software

Service-oriented and component-oriented software engineering

Services and components obviously have much in common. They are both reusable elements and, as I discussed in Chapter 17, it is possible to think of a component as a provider of services. However, there are important differences between services and components, and between a service-oriented and a component-oriented approach to software engineering.

http://www.SoftwareEngineering-9.com/Web/Services/Comps.html

engineering. This paradigm shift will be accelerated by the development of 'cloud computing' (Carr, 2009), where services are offered on a utility computing infrastructure hosted by major providers, such as Google and Amazon. This has had and will continue to have profound effects on systems products and business processes. Newcomer and Lomow (2005), in their book on SOA, summarize the potential of service-oriented approaches:

> *"Driven by the convergence of key technologies and the universal adoption of Web services, the service-oriented enterprise promises to significantly improve corporate agility, speed time-to-market for new products and services, reduce IT costs and improve operational efficiency."*

We are still at a relatively early stage in the development of service-oriented applications that are accessed over the Web. However, we are already seeing major changes in the ways that software is implemented and deployed, with the emergence of systems such as Google Apps and Salesforce.com. Service-oriented approaches at both the application and the implementation level means that the Web is evolving from an information store to a systems implementation platform.

19.1 Services as reusable components

In Chapter 17, I introduced component-based software engineering (CBSE), in which software systems are constructed by composing software components that are based on a standard component model. Services are a natural development of software components where the component model is, in essence, a set of standards associated with web services. A service can therefore be defined as the following:

> *A loosely-coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols.*

A critical distinction between a service and a software component, as defined in CBSE, is that services should be independent and loosely coupled; that is, they should always operate in the same way, irrespective of their execution environment. Their interface is a 'provides' interface that allows access to the service functionality. Services are intended to be independent and usable in different contexts. Therefore, they do not have a 'requires' interface that, in CBSE, defines the other system components that must be present.

Services communicate by exchanging messages, expressed in XML, and these messages are distributed using standard Internet transport protocols such as HTTP and TCP/IP. I have discussed this message-based approach to component communication in Section 18.1.1. A service defines what it needs from another service by setting out its requirements in a message and sending it to that service. The receiving service parses the message, carries out the computation and, on completion, sends a reply, as a message, to the requesting service. This service then parses the reply to extract the required information. Unlike software components, services do not use remote procedure or method calls to access functionality associated with other services.

When you intend to use a web service, you need to know where the service is located (its URI) and the details of its interface. These are described in a service description expressed in an XML-based language called WSDL. The WSDL specification defines three things about a web service: what the service does, how it communicates, and where to find it:

1. The 'what' part of a WSDL document, called an interface, specifies what operations the service supports, and defines the format of the messages that are sent and received by the service.
2. The 'how' part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a web service.
3. The 'where' part of a WSDL document describes the location of a specific web service implementation (its endpoint).

The WSDL conceptual model (Figure 19.4) shows the elements of a service description. Each of these is expressed in XML and may be provided in separate files. These parts are:

1. An introductory part that usually defines the XML namespaces used and which may include a documentation section providing additional information about the service.
2. An optional description of the types used in the messages exchanged by the service.
3. A description of the service interface; that is, the operations that the service provides for other services or users.
4. A description of the input and output messages processed by the service.

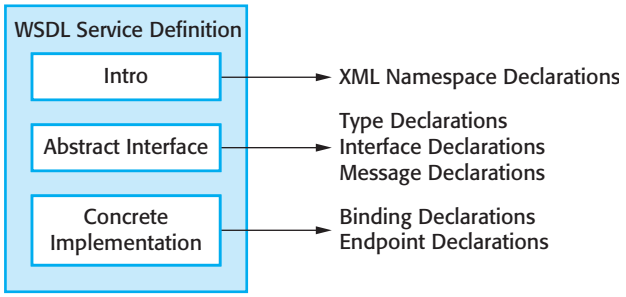

Figure 19.4 Organization of a WSDL specification

5. A description of the binding used by the service (i.e., the messaging protocol that will be used to send and receive messages). The default is SOAP but other bindings may also be specified. The binding sets out how the input and output messages associated with the service should be packaged into a message, and specifies the communication protocols used. The binding may also specify how supporting information, such as security credentials or transaction identifiers, is included.
6. An endpoint specification which is the physical location of the service, expressed as a Uniform Resource Identifier (URI)—the address of a resource that can be accessed over the Internet.

Complete service descriptions, written in XML, are long, detailed, and tedious to read. They usually include definitions of XML namespaces, which are qualifiers for names. A namespace identifier may precede any identifier used in the XML description, making it possible to distinguish between identifiers with the same name that have been defined in different parts of an XML description. You don't have to understand the details of namespaces to understand the examples here. You only need to know that names may be prefixed with a namespace identifier and that the namespace:name pair should be unique.

WSDL specifications are now rarely written by hand and most of the information in a specification can be automatically generated. You don't need to know the details of a specification to understand the principles of WSDL so I focus here on the description of the abstract interface. This is the part of a WSDL specification that equates to the 'provides' interface of a software component. Figure 19.5 shows part of the interface for a simple service that, given a date and a place, specified as a town within a country, returns the maximum and minimum temperature recorded in that place on that date. The input message also specifies whether these temperatures are to be returned in degrees Celsius or degrees Fahrenheit.

In Figure 19.5, the first part of the description shows part of the element and type definition that is used in the service specification. This defines the elements PlaceAndDate, MaxMinTemp, and InDataFault. I have only included the specification of PlaceAndDate, which you can think of as a record with three fields—town, country, and date. A similar approach would be used to define MaxMinTemp and InDataFault.

```
Define some of the types used. Assume that the namespace prefix 'ws' refers
to the namespace URI for XML schemas and the namespace prefix associated
with this definition is weathns.

<types>
   <xs: schema targetNameSpace = "http://.../weathns"
        xmlns: weathns = "http://.../weathns" >
        <xs:element name = "PlaceAndDate" type = "pdrec" />
        <xs:element name = "MaxMinTemp" type = "mmtrec" />
        <xs:element name = "InDataFault" type = "errmess" />

        <xs:complexType name = "pdrec"
        <xs:sequence>
             <xs:element name = "town" type = "xs:string"/>
             <xs:element name = "country" type = "xs:string"/>
             <xs:element name = "day" type = "xs:date" />
        </xs:complexType>

        Definitions of MaxMinType and InDataFault here

   </schema>

</types>

Now define the interface and its operations. In this case, there is only a
single operation to return maximum and minimum temperatures.

<interface name = "weatherInfo" >
   <operation name = "getMaxMinTemps" pattern = "wsdlns: in-out">
   <input messageLabel = "In" element = "weathns: PlaceAndDate" />
   <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
   <outfault messageLabel = "Out" element = "weathns:InDataFault" />
</operation>
</interface>
```

Figure 19.5 Part of a WSDL description for a web service

The second part of the description shows how the service interface is defined. In this example, the service weatherInfo has a single operation, although there are no restrictions on the number of operations that may be defined. The weatherInfo operation has an associated in-out pattern meaning that it takes one input message and generates one output message. The WSDL 2.0 specification allows for a number of different message exchange patterns such as in-only, in-out, out-only, in-optional-out, out-in, etc. The input and output messages, which refer to the definitions made earlier in the types section, are then defined.

The major problem with WSDL is that the definition of the service interface does not include any information about the semantics of the service or its non-functional characteristics, such as performance and dependability. It is simply a description of the service signature (i.e., the operations and their parameters). The programmer

who plans to use the service has to work out what the service actually does and what the different fields in the input and output messages mean. The performance and dependability have to be discovered by experimenting with the service. Meaningful names and documentation help with understanding the functionality that is offered but it is still possible for readers to misunderstand the service.

19.2 Service engineering

Service engineering is the process of developing services for reuse in service-oriented applications. It has much in common with component engineering. Service engineers have to ensure that the service represents a reusable abstraction that could be useful in different systems. They must design and develop generally useful functionality associated with that abstraction and ensure that the service is robust and reliable. They have to document the service so that it can be discovered and understood by potential users.

There are three logical stages in the service engineering process, as shown in Figure 19.6. These are as follows:

1. Service candidate identification, where you identify possible services that might be implemented and define the service requirements.
2. Service design, where you design the logical and WSDL service interfaces.
3. Service implementation and deployment, where you implement and test the service and make it available for use.

As I discussed in Chapter 16, the development of a reusable component may start with an existing component that has already been implemented and used in an application. The same is true for services—the starting point for this process will often be an existing service or a component that is to be converted to a service. In this situation, the design process involves generalizing the existing component so that application-specific features are removed. Implementation means adapting the component by adding service interfaces and implementing the required generalizations.

19.2.1 Service candidate identification

The basic notion of service-oriented computing is that services should support business processes. As every organization has a wide range of processes, there are therefore many possible services that may be implemented. Service candidate identification therefore involves understanding and analyzing the organization's business processes to decide which reusable services could be implemented to support these processes.

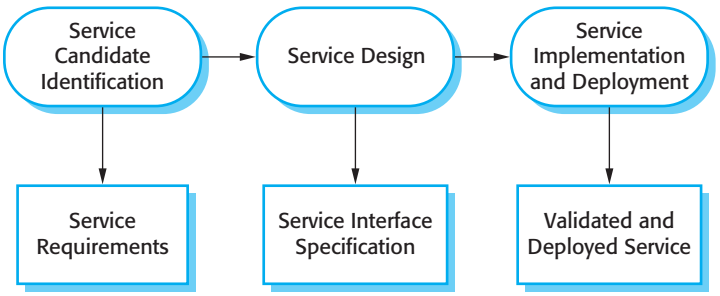

Figure 19.6 The service engineering process

Erl suggests that there are three fundamental types of service that may be identified:

1. *Utility services* These are services that implement some general functionality that may be used by different business processes. An example of a utility service is a currency conversion service that can be accessed to compute the conversion of one currency (e.g., dollars) to another (e.g., euros).
2. *Business services* These are services that are associated with a specific business function. An example of a business function in a university would be the registration of students for a course.
3. *Coordination or process services* These are services that support a more general business process which usually involves different actors and activities. An example of a coordination service in a company is an ordering service that allows orders to be placed with suppliers, goods accepted, and payments made.

Erl also suggests that services can be thought of as task-oriented or entity-oriented. Task-oriented services are those associated with some activity, whereas entity-oriented services are like objects. They are associated with a business entity such as, for example, a job application form. Figure 19.7 shows some examples of services that are task- or entity-oriented. Utility or business services may be entity-or task-oriented but coordination services are always task-oriented.

Your goal in service candidate identification should be to identify services that are logically coherent, independent, and reusable. Erl's classification is helpful in this respect as it suggests how to discover reusable services by looking at business entities and business activities. However, identifying service candidates is sometimes difficult because you have to envisage how the services will be used. You have to think of possible candidates then ask a series of questions about them to see if they are likely to be useful services. Possible questions that you might ask to identify potentially reusable services are:

1. For an entity-oriented service, is the service associated with a single logical entity that is used in different business processes? What operations are normally performed on that entity that must be supported?

	Utility	Business	Coordination
Task	Currency converter Employee locator	Validate claim form Check credit rating	Process expense claim Pay external supplier
Entity	Document style checker Web form to XML converter	Expenses form Student application form	

Figure 19.7 Service classification

2. For a task-oriented service, is the task one that is carried out by different people in the organization? Will they be willing to accept the inevitable standardization that occurs when a single support service is provided?
3. Is the service independent (i.e., to what extent does it rely on the availability of other services)?
4. For its operation, does the service have to maintain state? Services are stateless, which means that they do not maintain internal state. If state information is required, a database has to be used and this can limit service reusability. In general, services where the state is passed to the service are easier to reuse, as no database binding is required.
5. Could the service be used by clients outside of the organization? For example, an entity-oriented service associated with a catalog could be accessed by both internal and external users.
6. Are different users of the service likely to have different nonfunctional requirements? If they do, then this suggests that more than one version of a service should perhaps be implemented.

The answers to these questions help you select and refine abstractions that can be implemented as services. However, there is no formulaic way of deciding which are the best services and so service identification is a skill- and experience-based process.

The output of the service selection process is a set of identified services and associated requirements for these services. The functional service requirements should define what the service should do. The non-functional requirements should define the security, performance, and availability requirements of the service.

To help you understand the process of service candidate identification and implementation, consider the following example:

A large company, which sells computer equipment, has arranged special prices for approved configurations for some customers. To facilitate automated ordering, the company wishes to produce a catalog service that will allow customers to select the equipment that they need. Unlike a consumer catalog, orders are not placed directly through a catalog interface. Instead, goods are ordered through the web-based procurement system of each company that accesses the catalog as a web service. Most companies have their own budgeting and

approval procedures for orders and their own ordering process must be followed when an order is placed.

The catalog service is an example of an entity-oriented service that supports business operations. The functional catalog service requirements are as follows:

1. A specific version of the catalog shall be provided for each user company. This shall include the configurations and equipment that may be ordered by employees of the customer company and the agreed prices for catalog items.
2. The catalog shall allow a customer employee to download a version of the catalog for offline browsing.
3. The catalog shall allow users to compare the specifications and prices of up to six catalog items.
4. The catalog shall provide browsing and search facilities for users.
5. Users of the catalog shall be able to discover the predicted delivery date for a given number of specific catalog items.
6. Users of the catalog shall be able to place 'virtual orders' where the items required will be reserved for them for 48 hours. Virtual orders must be confirmed by a real order placed by a procurement system. This must be received within 48 hours of the virtual order.

In addition to these functional requirements, the catalog has a number of non-functional requirements:

1. Access to the catalog service shall be restricted to employees of accredited organizations.
2. The prices and configurations offered to one customer shall be confidential and shall not be available to employees of any other customer.
3. The catalog shall be available without disruption of service from 0700 GMT to 1100 GMT.
4. The catalog service shall be able to process up to 10 requests per second peak load.

Notice that there is no non-functional requirement related to the response time of the catalog service. This depends on the size of the catalog and the expected number of simultaneous users. As this is not a time-critical service, there is no need to specify it at this stage.

19.2.2 Service interface design

Once you have selected candidate services, the next stage in the service engineering process is to design the service interfaces. This involves defining the operations associated with the service and their parameters. You also have to think carefully

Operation	Description
MakeCatalog	Creates a version of the catalog tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalog.
Compare	Provides a comparison of up to six characteristics (e.g., price, dimensions, processor speed, etc.) of up to four catalog items.
Lookup	Displays all of the data associated with a specified catalog item.
Search	This operation takes a logical expression and searches the catalog according to that expression. It displays a list of all items that match the search expression.
CheckDelivery	Returns the predicted delivery date for an item if ordered that day.
MakeVirtualOrder	Reserves the number of items to be ordered by a customer and provides item information for the customer's own procurement system.

Figure 19.8 Functional descriptions of catalog service operations

about the design of the service operations and messages. Your aim should be to minimize the number of message exchanges that must take place to complete the service request. You have to ensure that as much information as possible is passed to the service in a message rather than using synchronous service interactions.

You should also remember that services are stateless and managing service-specific application state is the responsibility of the service user rather than the service itself. You may, therefore, have to pass this state information to and from services in input and output messages.

There are three stages to service interface design:

1. Logical interface design, where you identify the operations associated with the service, their inputs and outputs and the exceptions associated with these operations.
2. Message design, where you design the structure of the messages that are sent and received by the service.
3. WSDL development, where you translate your logical and message design to an abstract interface description written in WSDL.

The first stage, logical interface design, starts with the service requirements and defines the operation names and parameters. At this stage, you should also define the exceptions that may arise when a service operation is invoked. Figure 19.8 and Figure 19.9 show the operations that implement the requirements and the inputs, outputs, and exceptions for each of the catalog operations. At this stage, there is no need for these to be specified in detail—you add detail at the next stage of the design process.

Defining exceptions and how these can be communicated to service users is particularly important. Service engineers do not know how their services will be used.

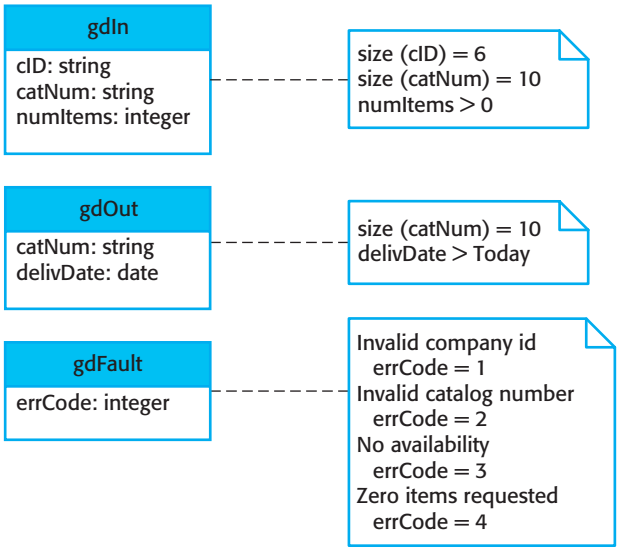

Figure 19.9 Catalog interface design

It is usually unwise to make assumptions that service users will have completely understood the service specification. Input messages may be incorrect so you should define exceptions that report incorrect inputs to the service client. It is generally good practice in reusable component development to leave all exception handling to the user of the component. The service developer should not impose their views on how exceptions should be handled.

Once you have established an informal logical description of what the service should do, the next stage is to define the structure of the input and output messages and the types used in these messages. XML is an awkward notation to use at this stage. I think it is better to represent the messages as objects and either define them using the UML or in a programming language, such as Java. They can then be manually or automatically converted to XML. Figure 19.10 shows the structure of the input and output messages for the getDelivery operation in the catalog service.

Notice how I have added detail to the description by annotating the UML diagram with constraints. These define the length of the strings representing the company and the catalog item, and specify that the number of items must be greater than zero and that delivery must be after the current date. The annotations also show which error codes are associated with each possible fault.

The final stage of the service design process is to translate the service interface design into WSDL. As I discussed in the previous section, a WSDL representation is long and detailed and hence it is easy to make mistakes at this stage if you do this manually. However, most programming environments that support service-oriented development (e.g., the ECLIPSE environment) include tools that can translate a logical interface description into its corresponding WSDL representation.

Operation	Inputs	Outputs	Exceptions
MakeCatalog	*mcIn* Company id PDF-flag	*mcOut* URL of the catalog for that company	*mcFault* Invalid company id
Compare	*compIn* Company id Entry attribute (up to 6) Catalog number (up to 4)	*compOut* URL of page showing comparison table	*compFault* Invalid company id Invalid catalog number Unknown attribute
Lookup	*lookIn* Company id Catalog number	*lookOut* URL of page with the item information	*lookFault* Invalid company id Invalid catalog number
Search	*searchIn* Company id Search string	*searchOut* URL of web page with search results	*searchFault* Invalid company id Badly formed search string
CheckDelivery	*gdIn* Company id Catalog number Number of items required	*gdOut* Catalog number Expected delivery date	*gdFault* Invalid company id Invalid catalog number No availability Zero items requested
PlaceOrder	*poIn* Company id Number of items required Catalog number	*poOut* Catalog number Number of items required Predicted delivery date Unit price estimate Total price estimate	*poFault* Invalid company id Invalid catalog number Zero items requested

Figure 19.10 UML definition of input and output messages

19.2.3 Service implementation and deployment

Once you have identified candidate services and designed their interfaces, the final stage of the service engineering process is service implementation. This implementation may involve programming the service using a standard programming language such as Java or C#. Both of these languages include libraries with extensive support for service development.

Alternatively, services may be developed by implementing service interfaces to existing components or, as I discuss below, to legacy systems. This means that software assets that have already proved to be useful can be made more widely available. In the case of legacy systems, it may mean that the system functionality can be accessed by new applications. You can also develop new services by defining compositions of existing services. I cover this approach to service development in Section 19.3.

Once a service has been implemented, it then has to be tested before it is deployed. This involves examining and partitioning the service inputs (as explained

in Chapter 8), creating input messages that reflect these input combinations, then checking that the outputs are expected. You should always try to generate exceptions during the test to check that the service can cope with invalid inputs. Testing tools are available that allow services to be examined and tested, and that generate tests from a WSDL specification. However, these can only test the conformity of the service interface to the WSDL. They cannot test that the service's functional behavior.

Service deployment, the final stage of the process, involves making the service available for use on a web server. Most server software makes this very simple. You only have to install the file containing the executable service in a specific directory. It then automatically becomes available for use. If the service is intended to be publicly available, you then have to provide information for external users of the service. This information helps potential external users to decide if the service is likely to meet their needs and if they can trust you, as a service provider, to deliver the service reliably and securely. Information that you may include in a service description might be the following:

1. Information about your business, contact details, etc. This is important for trust reasons. Users of a service have to be confident that it will not behave maliciously. Information about the service provider allows them to check their credentials with business information agencies.
2. An informal description of the functionality provided by the service. This helps potential users to decide if the service is what they want. However, the functional description is in natural language, so it is not an unambiguous semantic description of what the service does.
3. A detailed description of the interface types and semantics.
4. Subscription information that allows users to register for information about updates to the service.

As I have discussed, a general problem with service specifications is that the functional behavior of the service is usually specified informally, as a natural language description. Natural language descriptions are easy to read, but they are subject to misinterpretation. To address this problem, there is an active research community concerned with investigating how the semantics of services may be specified. The most promising approach to semantic specification is based on an ontology-based description, where the specific meaning of terms in a description is defined in an ontology. Ontologies are a way of standardizing the ways that terminology is used and they define the relationships between different terms. They are becoming increasingly used to help assign semantics to natural language descriptions. A language called OWL-S has been developed for describing web service ontologies (OWL_Services_Coalition, 2003).

19.2.4 Legacy system services

Legacy systems are old software systems that are used by an organization. Usually, they rely on obsolete technology but are still essential to the business. It may not be cost effective to rewrite or replace these systems and many organizations would like

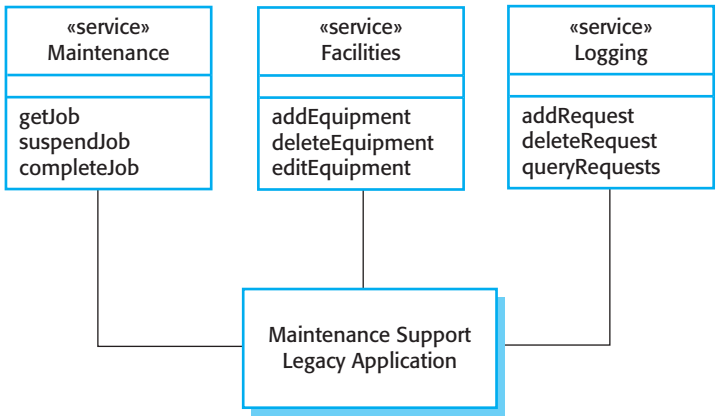

Figure 19.11 Services providing access to a legacy system

to use them in conjunction with more modern systems. One of the most important uses of services is to implement 'wrappers' for legacy systems that provide access to a system's functions and data. These systems can then be accessed over the Web and integrated with other applications.

To illustrate this, imagine that a large company maintains an inventory of its equipment and an associated database that keeps track of equipment maintenance and repairs. This keeps track of what maintenance requests have been made for different pieces of equipment, what regular maintenance is scheduled, when maintenance was carried out, how much time was spent on maintenance, etc. This legacy system was originally used to generate daily job lists for maintenance staff but, over time, new facilities have been added. These provide data about how much has been spent on maintenance for each piece of equipment and information to help to cost maintenance work to be carried out by external contractors. The system runs as a client–server system with special-purpose client software running on a PC.

The company now wishes to provide real-time access to this system from portable terminals used by maintenance staff. They will update the system directly with the time and resources spent on maintenance and will query the system to find their next maintenance job. In addition, call center staff require access to the system to log maintenance requests and to check their status.

It is practically impossible to enhance the system to support these requirements so the company decides to provide new applications for maintenance and call center staff. These applications rely on the legacy system, which is to be used as a basis for implementing a number of services. This is illustrated in Figure 19.11, where I have used a UML stereotype to indicate a service. New applications exchange messages with these services to access the legacy system functionality.

Some of the services provided are the following:

1. *A maintenance service* This includes operations to retrieve a maintenance job according to its job number, priority, and geographical location, and to upload details of maintenance that has been carried out to the maintenance database.

The service also provides operations that allow a maintenance job that has started but is incomplete to be suspended and restarted.

2. *A facilities service* This includes operations to add and delete new equipment and to modify the information associated with equipment in the database.
3. *A logging service* This includes operations to add a new request for service, delete maintenance requests, and query the status of outstanding requests.

Notice that the existing legacy system is not simply represented as a single service. Rather, the services that are developed to access the legacy system are coherent and support a single area of functionality. This reduces their complexity and makes them easier to understand and reuse in other applications.

19.3 Software development with services

The development of software using services is based around the idea that you compose and configure services to create new, composite services. These may be integrated with a user interface implemented in a browser to create a web application, or may be used as components in some other service composition. The services involved in the composition may be specially developed for the application, may be business services developed within a company, or may be services from an external provider.

Many companies are now converting their enterprise applications into service-oriented systems, where the basic application building block is a service rather than a component. This opens up the possibility of more widespread reuse within the company. The next stage will be the development of interorganizational applications between trusted suppliers, who will use each other's services. The final realization of the long-term vision of SOAs will rely on the development of a 'services market', where services are bought from external suppliers.

Service composition may be used to integrate separate business processes to provide an integrated process offering more extensive functionality. Say an airline wishes to provide a complete vacation package for travelers. As well as booking their flights, travelers can also book hotels in their preferred location, arrange car rentals or book a taxi from the airport, browse a travel guide, and make reservations to visit local attractions. To create this application, the airline composes its own booking service with services offered by a hotel booking agency, car rental and taxi companies, and reservation services offered by owners of local attractions. The end result is a single service that integrates the services from different providers.

You can think of this process as a sequence of separate steps as shown in Figure 19.12. Information is passed from one step to the next—for example, the car rental company is informed of the time that the flight is scheduled to arrive. The sequence of steps is called a workflow—a set of activities ordered in time, with each activity carrying out some part of the work. A workflow is a model of a business

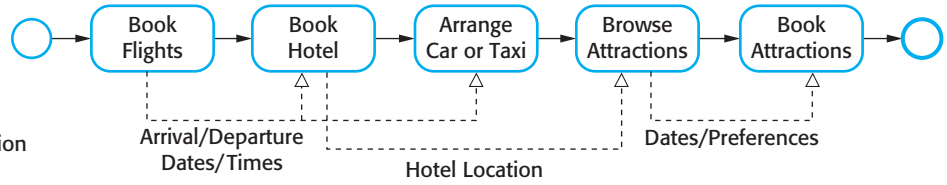

Figure 19.12 Vacation package workflow

process (i.e., sets out the steps involved in reaching a particular goal that is important for a business). In this case, the business process is the vacation booking service, offered by the airline.

Workflow is a simple idea and the above scenario of booking a vacation seems to be straightforward. In practice, service composition is much more complex than this simple model implies. For example, you have to consider the possibility of service failure and incorporate mechanisms to handle these failures. You also have to take into account exceptional demands made by users of the application. For example, say a traveler was disabled and required a wheelchair to be rented and delivered to the airport. This would require extra services to be implemented and composed, and additional steps to be added to the workflow.

You must be able to cope with situations where the workflow has to be changed because the normal execution of one of the services usually results in an incompatibility with some other service execution. For example, say a flight is booked to leave on June 1st and return on June 7th. The workflow then proceeds to the hotel booking stage. However, the resort is hosting a major convention until June 2nd, so no hotel rooms are available. The hotel booking service reports this lack of availability. This is not a failure; lack of availability is a common situation. You, therefore, then have to 'undo' the flight booking and pass the information about lack of availability back to the user. He or she then has to decide whether to change their dates or their resort. In workflow terminology, this is called a 'compensation action'. Compensation actions are used to undo actions that have already been completed but which must be changed as a result of later workflow activities.

The process of designing new services by reusing existing services is essentially a process of software design with reuse (Figure 19.13). Design with reuse inevitably involves requirements compromises. The 'ideal' requirements for the system have to be modified to reflect the services that are actually available, whose costs fall within budget and whose quality of service is acceptable.

In Figure 19.13, I have shown six key stages in the process of service construction by composition:

1. *Formulate outline workflow* In this initial stage of service design, you use the requirements for the composite service as a basis for creating an 'ideal' service design. You should create a fairly abstract design at this stage with the intention of adding details once you know more about available services.

2. *Discover services* During this stage of the process, you search service registries or catalogs to discover what services exist, who provides these services, and the details of the service provision.

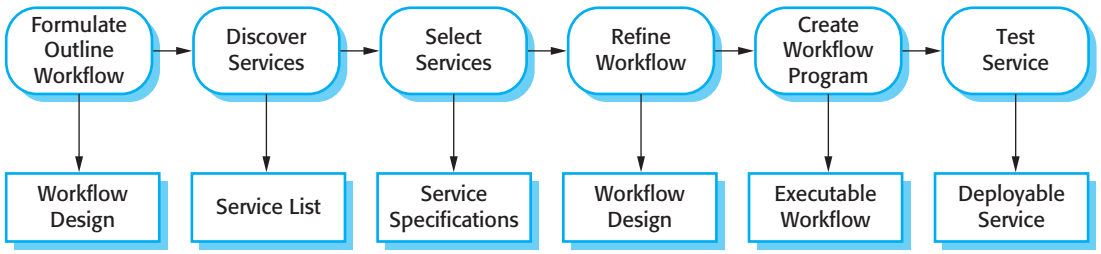

Figure 19.13 Service construction by composition

3. *Select possible services* From the set of possible service candidates that you have discovered, you then select possible services that can implement workflow activities. Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered. You may decide to choose a number of functionally equivalent services, which could be bound to a workflow activity depending on details of cost and quality of service.

4. *Refine workflow* On the basis of information about the services that you have selected, you then refine the workflow. This involves adding detail to the abstract description and perhaps adding or removing workflow activities. You may then repeat the service discovery and selection stages. Once a stable set of services has been chosen and the final workflow design established, you move on to the next stage in the process.

5. *Create workflow program* During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can use a conventional programming language, such as Java or C#, for service implementation or a workflow language, such as WS-BPEL. As I discussed in the previous section, the service interface specification should be written in WSDL. This stage may also involve the creation of web-based user interfaces to allow the new service to be accessed from a web browser.

6. *Test completed service or application* The process of testing the completed, composite service is more complex than component testing in situations where external services are used. I discuss testing issues in Section 19.3.2.

In the remainder of this chapter, I focus on workflow design and testing. In practice, service discovery does not appear to be a major problem. It is still the case that most service reuse is within organizations, where services can be discovered using internal registries and informal communications between software engineers. Standard search engines may be used to discover publicly available services.

19.3.1 Workflow design and implementation

Workflow design involves analyzing existing or planned business processes to understand the different activities that go on and how these exchange information.

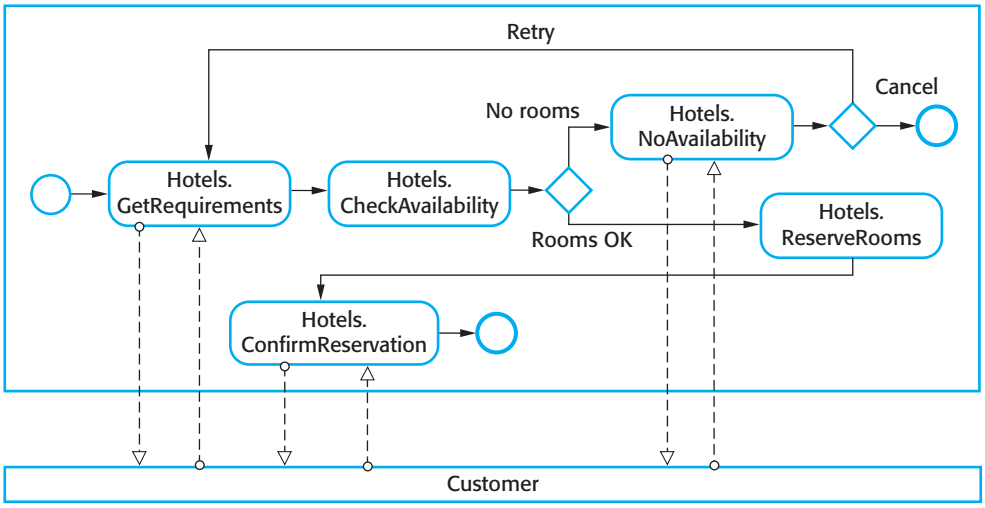

Figure 19.14 A fragment of a hotel booking workflow

You then define the new business process in a workflow design notation. This sets out the stages involved in enacting the process and the information that is passed between the different process stages. However, existing processes may be informal and dependent on the skills and ability of the people involved—there may be no 'normal' way of working or process definition. In such cases, you have to use your knowledge of the current process to design a workflow that achieves the same goals.

Workflows represent business process models and are usually represented using a graphical notation such as UML activity diagrams or BPMN, the Business Process Modeling Notation (White, 2004a; White and Miers, 2008). These offer similar features (White, 2004b). I think it is probable that BPMN and UML activity diagrams will be integrated in the future and a standard for workflow modeling defined will be based on this integrated language. I use BPMN for the examples in this chapter.

BPMN is a graphical language that is reasonably easy to understand. Mappings have been defined to translate the language to lower-level, XML-based descriptions in WS-BPEL. BPMN is therefore conformant with the stack of web service standards that I showed in Figure 19.2.

Figure 19.14 is an example of a simple BPMN model of part of the above vacation package scenario. The model shows a simplified workflow for hotel booking and assumes the existence of a Hotels service with associated operations called GetRequirements, CheckAvailability, ReserveRooms, NoAvailability, ConfirmReservation, and CancelReservation. The process involves getting requirements from the customer, checking room availability, and then, if rooms are available, making a booking for the required dates.

This model introduces some of the core concepts of BPMN that are used to create workflow models:

1. Activities are represented by a rectangle with rounded corners. An activity can be executed by a human or by an automated service.

2. Events are represented by circles. An event is something that happens during a business process. A simple circle is used to represent a starting event and a darker circle to represent an end event. A double circle (not shown) is used to represent an intermediate event. Events can be clock events, thus allowing workflows to be executed periodically or timed out.
3. A diamond is used to represent a gateway. A gateway is a stage in the process where some choice is made. For example, in Figure 19.14, there is a choice made on the basis of whether rooms are available or not.
4. A solid arrow is used to show the sequence of activities; a dashed arrow represents message flow between activities. In Figure 19.14, these messages are passed between the hotel booking service and the customer.

These key features are enough to describe the essence of most workflows. However, BPMN includes many additional features that I don't have space to describe here. These add information to a business process description that allows it to be automatically translated into an executable service. Therefore, web services, based on service compositions described in BPMN, can be generated directly from a business process model.

Figure 19.14 shows the process that is enacted in one organization, the company that provides a booking service. However, the key benefit of a service-oriented approach is that it supports interorganizational computing. This means that a computation involves services in different companies. This is represented in BPMN by developing separate workflows for each of the organizations involved with interactions between them.

To illustrate this, I use a different example, drawn from high-performance computing. A service-oriented approach has been proposed to allow resources such as high-performance computers to be shared. In this example, assume that a vector processing computer (a machine that can carry out parallel computations on arrays of values) is offered as a service (VectorProcService) by a research laboratory. This is accessed through another service called SetupComputation. These services and their interactions are shown in Figure 19.15.

In this example, the workflow for the SetupComputation service requests access to a vector processor and, if a processor is available, establishes the computation required and downloads data to the processing service. Once the computation is complete, the results are stored on the local computer. The workflow for VectorProcService checks if a processor is available, allocates resources for the computation, initializes the system, carries out the computation, and returns the results to the client service.

In BPMN terms, the workflow for each organization is represented in a separate pool. It is shown graphically by enclosing the workflow for each participant in the process in a rectangle, with the name written vertically on the left edge. The workflows defined in each pool are coordinated by exchanging messages; sequence flow between the activities in different pools is not allowed. In situations where different parts of an organization are involved in a workflow, this can be shown by

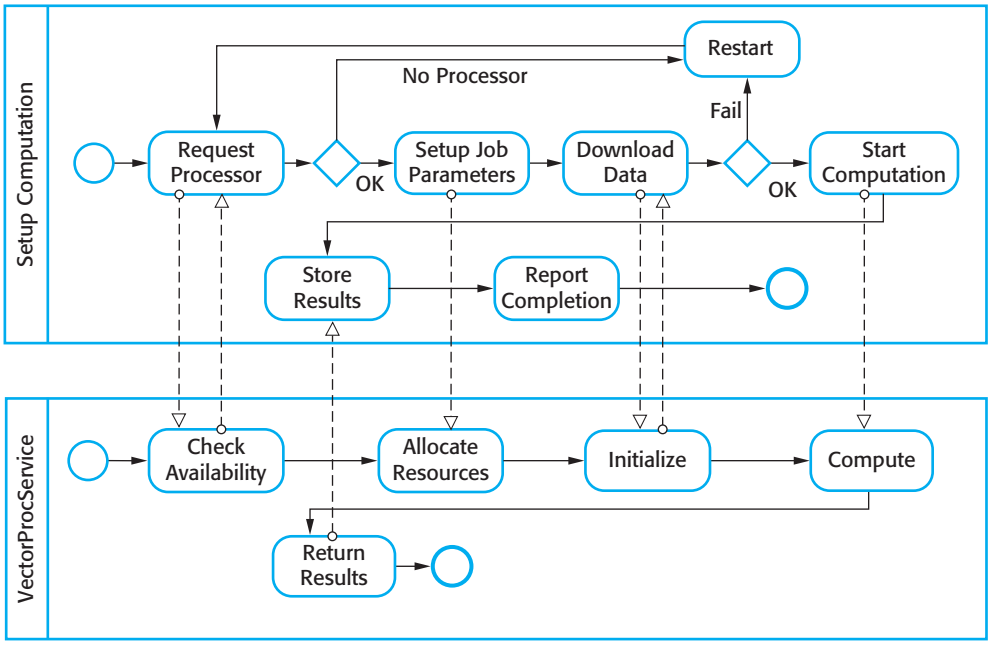

Figure 19.15
Interacting workflows

separating pools into named 'lanes'. Each lane shows the activities in that part of the organization.

Once a business process model has been designed, this has to be refined depending on the services that have been discovered. As I suggested in the discussion of Figure 19.13, the model may go through a number of iterations until a design that allows the maximum possible reuse of available services has been created.

Once the final design is available, it must then be converted to an executable program. This may involve two activities:

1. Implementing the services that are not available for reuse. As services are implementation-language independent, these services can be written in any language. Both Java and C# development environments provide support for web service composition.
2. Generating an executable version of the workflow model. This normally involves translating the model into WS-BPEL, either automatically or by hand. Although there are several tools available to automate the BPMN-WS-BPEL process, there are some circumstances where it is difficult to generate readable WS-BPEL code from a workflow model.

To provide direct support for the implementation of web service compositions, several web service standards have been developed. As I explained in the chapter introduction, the standard XML-based language is WS-BPEL (Business Process Execution Language) which is a 'programming language' to control interactions

between services. This is supported by additional standards such as WS-Coordination (Cabrera et al., 2005), which is used to specify how services are coordinated, and WS-CDL (Choreography Description Language) (Kavantzas et al., 2004), which is a means of defining the message exchanges between participants (Andrews et al., 2003).

19.3.2 Service testing

Testing is important in all system development processes as it demonstrates that a system meets its functional and non-functional requirements and to detect defects that have been introduced during the development process. Many testing techniques, such as program inspections and coverage testing, rely on analysis of the software source code. However, when services are offered by an external provider, source code of the service implementation is not available. Service-based system testing cannot therefore use proven source code–based techniques.

As well as problems of understanding the implementation of the service, testers may also face further difficulties when testing services and service compositions:

1. External services are under the control of the service provider rather than the user of the service. The service provider may withdraw these services at any time or may make changes to them, which invalidates any previous application testing. These problems are handled in software components by maintaining different versions of the component. Currently, however, there are no standards proposed to deal with service versions.
2. The long-term vision of SOAs is for services to be bound dynamically to service-oriented applications. This means that an application may not always use the same service each time that it is executed. Therefore, tests may be successful when an application is bound to a particular service, but it cannot be guaranteed that that service will be used during an actual execution of the system.
3. The non-functional behavior of a service is not simply dependent on how it is used by the application that is being tested. A service may perform well during testing because it is not operating under a heavy load. In practice, the observed service behavior may be different because of the demands made by other service users.
4. The payment model for services could make service testing very expensive. There are different possible payment models—some services may be freely available, some paid for by subscription, and others paid for on a per-use basis. If services are free, then the service provider will not wish them to be loaded by applications being tested; if a subscription is required, then a service user may be reluctant to enter into a subscription agreement before testing the service. Similarly, if the usage is based on payment for each use, service users may find the cost of testing to be prohibitive.

5. I have discussed the notion of compensation actions that are invoked when an exception occurs and previous commitments that have been made (such as a flight reservation) have to be revoked. There is a problem in testing such actions as they may depend on the failure of other services. Ensuring that these services actually fail during the testing process may be very difficult.

These problems are particularly acute when external services are used. They are less serious when services are used within the same company or where cooperating companies trust services offered by their partners. In such cases, source code may be available to guide the testing process and payment for services is unlikely to be a problem. Resolving these testing problems and producing guidelines, tools, and techniques for testing service-oriented applications remains an important research issue.

KEY POINTS

- Service-oriented architecture is an approach to software engineering where reusable, standardized services are the basic building blocks for application systems.
- Service interfaces may be defined in an XML-based language called WSDL. A WSDL specification includes a definition of the interface types and operations, the binding protocol used by the service and the service location.
- Services may be classified as utility services that provide a general-purpose functionality, business services that implement part of a business process, or coordination services that coordinate the execution of other services.
- The service engineering process involves identifying candidate services for implementation, defining the service interface and implementing, and testing and deploying the service.
- Service interfaces may be defined for legacy software systems that continue to be useful for an organization. The functionality of the legacy system may then be reused in other applications.
- The development of software using services is based around the idea that programs are created by composing and configuring services to create new composite services.
- Business process models define the activities and information exchange that takes place in a business process. Activities in the business process may be implemented by services so that the business process model represents a service composition.

FURTHER READING

There is an immense amount of tutorial material on the Web covering all aspects of web services. However, I found the following two books by Thomas Erl to be the best overview and description of services and service standards. Unlike most books, Erl includes some discussion of software

engineering issues in service-oriented computing. He has also written more specialized books on the design of services and SOA design patterns, although these are generally aimed at readers with experience of implementing SOA.

Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services. The primary focus of this book is the underlying XML-based technologies (SOAP, WSDL, BPEL, etc.) that are a framework for SOA. (T. Erl, Prentice Hall, 2004.)

Service-Oriented Architecture: Concepts, Technology and Design. This is a more general book on the engineering of service-oriented systems. There is a little bit of overlap with the text above but Erl mostly concentrates on discussing how a service-oriented approach may be used at all stages of the software process. (T. Erl, Prentice Hall, 2005.)

'SOA realization: Service design principles'. This short web article is an excellent overview of the issues to be considered in designing services. (D. J. N. Artus, IBM, 2006.) http://www.ibm.com/developerworks/webservices/library/ws-soa-design/.

EXERCISES

19.1. What are the most important distinctions between services and software components?

19.2. Explain why SOAs should be based on standards.

19.3. Using the same notation, extend Figure 19.5 to include definitions for MaxMinType and InDataFault. The temperatures should be represented as integers with an additional field indicating whether the temperature is in degrees Fahrenheit or degrees Celsius. InDataFault should be a simple type consisting of an error code.

19.4. Define an interface specification for the Currency Converter and Check credit rating services shown in Figure 19.7.

19.5. Design possible input and output messages for the services shown in Figure 19.11. You may specify these in the UML or in XML.

19.6. Giving reasons for your answer, suggest two important types of applications where you would *not* recommend the use of service-oriented architecture.

19.7. In Section 19.2.1, I introduced an example of a company that has developed a catalog service that is used by customers' web-based procurement systems. Using BPMN, design a workflow that uses this catalog service to look up and place orders for computer equipment.

19.8. Explain what is meant by a 'compensation action' and, using an example, show why these actions may have to be included in workflows.

19.9. For the example of the vacation package reservation service, design a workflow that will book ground transportation for a group of passengers arriving at an airport. They should be given the option of booking either a taxi or renting a car. You may assume that the taxi and car rental companies offer web services to make a reservation.

19.10. Using an example, explain in detail why the thorough testing of services that include compensation actions is difficult.