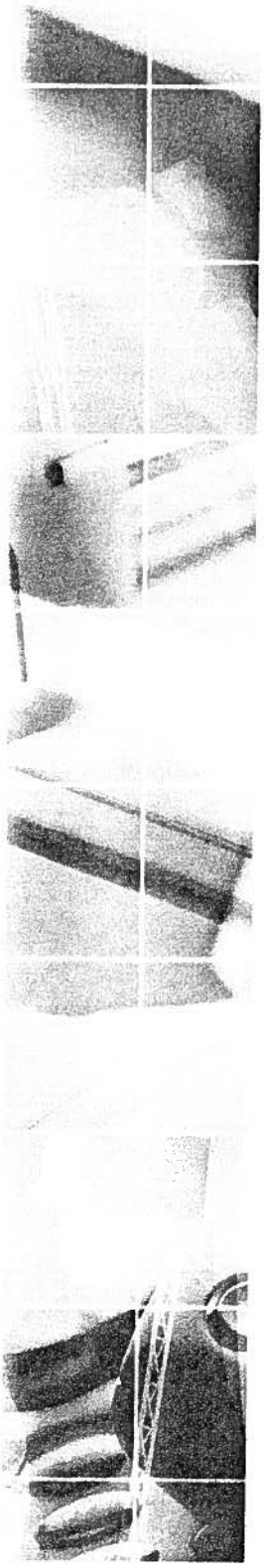


31 Service-oriented software engineering

Objectives

The objective of this chapter is to introduce service-oriented software engineering, an increasingly important approach to business application development. When you have read this chapter, you will:

- understand the basic notions of a web service and web service standards and how these can support inter-organisational computing;
- 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

In Chapter 12, I introduced the notion of service-oriented architectures as a means of facilitating inter-organisational computing. Essentially, service-oriented architectures (SOA) are a way of developing distributed systems where the components of these systems are stand-alone services. These services may execute on geographically distributed computers. Standard protocols have been designed to support service communication and information exchange. Consequently, services are platform and implementation-language independent. Software systems can be constructed using services from different providers with seamless interaction between these services.

Figure 31.1 illustrates how web services are used. Service providers design and implement services and specify these services in a language called WSDL (discussed later). They also publish information about these services in a generally accessible registry using a publication standard called UDDI. Service requestors (sometimes called service clients), who wish to make use of a service, search the UDDI registry to discover the specification of that service and to locate the service provider. They can then bind their application to that specific service and communicate with it, usually using a protocol called SOAP.

Service-oriented architecture is now generally recognised as a significant development, particularly for business application systems. It allows flexibility as services can be provided locally or outsourced to external providers. Services may be implemented in any programming language. By wrapping legacy systems (see Chapter 21) as services, companies can preserve their investment in valuable software and make this available to a wider range of applications. SOA allows different platforms and implementation technologies that may be used in different parts of a company to inter-operate. Most importantly, perhaps, building applications based on services allows companies and other organisations to cooperate and to 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.

Perhaps the key reason for the success of service-oriented architectures is the fact that, from the outset, there has been an active standardisation process working alongside technical developments. All of the major hardware and software companies are committed to these standards. As a result, service-oriented architectures

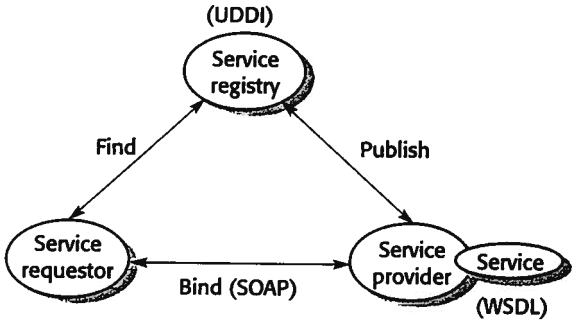


Figure 31.1 Service-oriented architecture

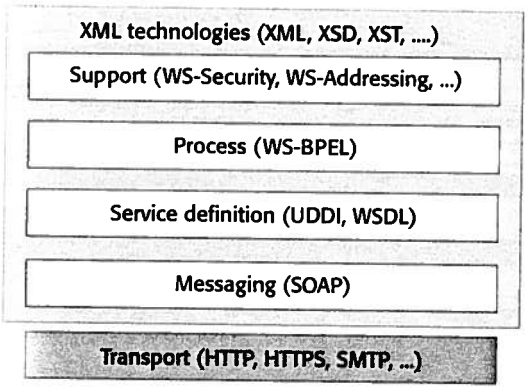


Figure 31.2 Web service standards

have not suffered from the incompatibilities that normally arise with technical innovations, where different suppliers maintain their proprietary version of the technology. Hence, problems, such as the multiple incompatible component models in CBSE that I discussed in Chapter 19, have not arisen in service-oriented system development.

Figure 31.2 shows the stack of key standards that have been established to support web services. In principle, a service-oriented approach may be applied in situations where other protocols are used; in practice, web services are dominant. Web services do not depend on any particular transport protocol for information exchange although, in practice, the HTTP and HTTPS protocols are commonly used.

Web service protocols cover all aspects of service-oriented architectures 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 (Erl, 2004) provides a good summary of XML technologies and their role in web services.

Briefly, the key standards for web service-oriented architectures are:

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) standard defines the way in which service providers should define the interface to these services. Essentially, it allows the interface of a service (the service operations, parameters and their types) and its bindings to be defined in a standard way.
3. *UDDI* The UDDI (Universal Description, Discovery and Integration) standard defines the components of a service specification that may be used to discover the existence of a service. These include information about the service provider, the services provided, the location of the service description (usually expressed

in WSDL) and information about business relationships. UDDI registries enable potential users of a service to discover what services are available.

4. *WS-BPEL* This standard 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 31.3.

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

1. WS-Reliable Messaging is a standard for message exchange that ensures messages will be delivered once and once only.
2. WS-Security is 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 defines how address information should be represented in a SOAP message.
4. WS-Transactions defines how transactions across distributed services should be coordinated.

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

As I discuss in the following section, a service can be considered simply as a reusable abstraction and hence this chapter complements Chapters 18 and 19 that discuss issues of software reuse. There are therefore two themes to the chapter:

1. Service engineering. This concerns the development of dependable, reusable services. Essentially, the concern is software development for reuse.
2. Software development with services. This concerns the development of dependable software systems that use services either on their own or in conjunction with other types of component. Essentially, the concern is software development with reuse.

Service-oriented architectures and service-oriented software engineering are, currently, a 'hot topic'. There is an enormous amount of business interest in adopting a service-oriented approach to software development but, at the time of writing, practical experience with service-oriented system is limited. Hot topics always generate ambitious visions and often promise more than they finally deliver. For example, in their book on SOA, Newcomer and Lomow (2005) state:

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.

I believe that service-oriented software engineering is as important a development as object-oriented software engineering. However, the reality is that it will take many years to realise these benefits and for the vision of SOA to become a reality. Because service-oriented software development is so new, we do not yet have well-established software engineering methods for this type of system. I therefore focus here on general issues of designing and implementing services and building systems using service composition.

31.1 Services as reusable components

In Chapter 19, I introduced component-based software engineering (CBSE) where software systems are constructed by composing software components that are based on some standard component model. Services are a natural development of software components where the component model is, in essence, the set of standards associated with web services. A service can therefore be defined as:

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 provides 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 may also be distributed over the Internet. They communicate by exchanging messages, expressed in XML, and these messages are distributed using standard Internet transport protocols such as HTTP and TCP/IP. 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 message to the requesting service. This service then parses the reply to extract the required information. Unlike software components, services do not 'call' methods associated with other services.

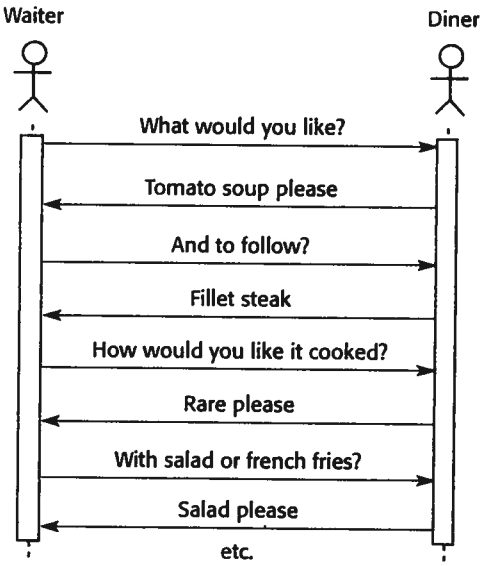


Figure 31.3 Synchronous interaction when ordering a meal

To illustrate the difference between communication using method calls and communication using message passing, consider a situation where you are ordering a meal in a restaurant. When you have a conversation with the waiter, you are involved in a series of synchronous interactions that define your order. This is comparable to components interacting in a software system, where one component calls methods from other components. The waiter writes down your order along with the order of the other people with you, then passes this message, which includes details of everything that has been ordered, to the kitchen to prepare the food. Essentially, the waiter service is passing a message to the kitchen service defining the food to be prepared.

I have illustrated this in Figure 31.3, which shows the synchronous ordering process, and in Figure 31.4, which shows a hypothetical XML message, which I hope is self-explanatory, that defines an order made by the table of three people. The difference is clear—the waiter takes the order as a series of interactions, with each interaction defining part of the order. However, the waiter has a single interaction with the kitchen where the message passed defines the complete order.

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 (Web Service Description Language). The WSDL specification defines three things about a Web service. It defines *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.

Figure 31.4
A restaurant order expressed as an XML message

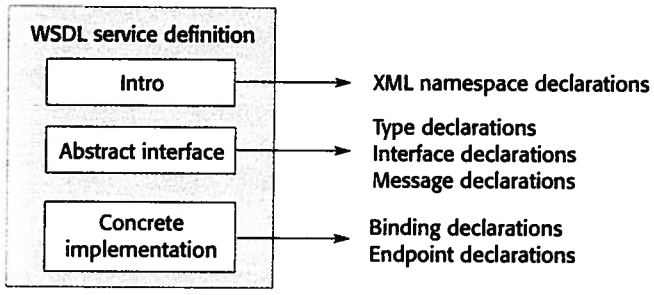
```
<starter>
        <dish name = "soup" type = "tomato" />
        <dish name = "soup" type = "fish" />
        <dish name = "pigeon salad" />
</starter>
<main course>
        <dish name = "steak" type = "sirloin" cooking = "medium" />
        <dish name = "steak" type = "fillet" cooking = "rare" />
        <dish name = "sea bass">
</main>
<accompaniment>
        <dish name = "french fries" portions = "2" />
        <dish name = "salad" portions = "1" />
</accompaniment>
```

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, called (confusingly) a service, describes where to locate a specific Web service implementation.

The WSDL conceptual model (Figure 31.5) shows all the parts 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 which, 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, i.e. the operations that it provides.
4. A description of the input and output messages processed by the service.

Figure 31.5
Organisation 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. It means that it is possible to distinguish between identifiers with the same name that have been defined in different parts of an XML description. I do not want to go into details of namespaces here. To understand this chapter, you need to know only that names can be prefixed with a namespace identifier and that the namespace:name pair should be unique.

I have included an example of a complete service description on the book website. However, as this is very long, I focus here on the description of the abstract interface. This is the part of the WSDL that equates to the 'provides' interface of a software component. Figure 31.6 shows details of the interface for a simple service that, given a date and a place (town and country), returns the maximum and minimum temperature recorded in that place on that date. These temperatures may be returned in degrees Celsius or in degrees Fahrenheit, depending on the location where they were recorded.

In Figure 31.6, 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.

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

Figure 31.6 Part of a WSDL description for a web service

```
Define some of the types used. Assume that the namespace prefixes '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>
```

the service signature and it relies on the user of the service to deduce what the service actually does and what the different fields in the input and output messages mean. While meaningful names and service documentation helps here, there is still scope for misunderstanding and misusing the service.

31.2 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 must ensure that the service is robust and reliable so that it operates dependably in different applications. They have to document the service so that it can be discovered by and understood by potential users.

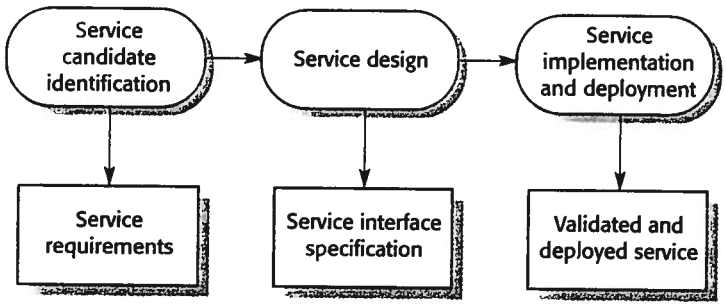


Figure 31.7 The service engineering process

There are three logical stages in the service engineering process (Figure 31.7). These are:

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.

I discuss each of these stages in this section of the book.

31.2.1 Service candidate identification

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

Erl identifies 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 registering 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.

Figure 31.8 Service classification

	Utility	Business	Coordination
Task	Currency convertor 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	

Erl also suggests that services can be considered 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 some business entity such as, for example, a job application form. Figure 31.8 suggests some examples of services that are task or entity-oriented. While services can be utility and business services, 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, just as the processes of object and component identification are difficult, so too is service candidate identification. 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 help you to identify 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?
2. For a task-oriented service, is the task one that is carried out by different people in the organisation? Will they be willing to accept the inevitable standardisation 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? If so, will a database be used for state maintenance? In general, systems that rely on internal state are less reusable than those where state can be externally maintained.
5. Could the service be used by clients outside of the organisation? For example, an entity-oriented service associated with a catalogue may be accessed both internally and externally?
6. Are different users of the service likely to have different non-functional 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 candidate 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.

Assume that 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 catalogue service that will allow customers to select the equipment that they need. Unlike a consumer catalogue, however, orders are not placed directly, through a catalogue interface, but are made through the web-based procurement system of each company. 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 catalogue service is an example of an entity-oriented service that supports business operations. The functional catalogue service requirements are:

1. A specific version of the catalogue 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 catalogue items.
2. The catalogue shall allow a customer employee to download a version of the catalogue for off-line browsing.
3. The catalogue shall allow users to compare the specifications and prices of up to six catalogue items.
4. The catalogue shall provide browsing and searching facilities for users.
5. Users of the catalogue shall be able to discover the predicted delivery date for a given number of specific catalogue items.
6. Users of the catalogue 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 catalogue has a number of non-functional requirements:

1. Access to the catalogue service shall be restricted to employees of accredited organisations.
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 catalogue shall be available without disruption of service from 0700 GMT to 1100 GMT.

4. The catalogue 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 catalogue service. This depends on the size of the catalogue 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.

31.2.2 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 about how the operations and messages of the service can be designed to minimise 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 require synchronous service interactions.

You should also remember that services are stateless and managing a 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, the inputs and outputs of these operations 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 associated with the service. At this stage, you should also define the exceptions that may arise when a service operation is invoked. Figures 31.9 and 31.10 show the operations that implement the requirements and the inputs, outputs and exceptions for each of the catalogue 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

Figure 31.9 Functional descriptions of catalogue service operations

Operation	Description
MakeCatalogue	Creates a version of the catalogue tailored for a specific customer. Includes an optional parameter to create a downloadable PDF version of the catalogue.
Compare	Provides a comparison of up to six characteristics (e.g. price, dimensions, processor speed, etc.) of up to four catalogue items for comparison.
Lookup	Displays all of the data associated with a specified catalogue item.
Search	This operation takes a logical expression and searches the catalogue 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 it is ordered today.
MakeVirtualOrder	Reserves the number of items to be ordered by a customer and provides item information for the customer's own procurement system.

Figure 31.10 Catalogue interface design

Operation	Inputs	Outputs	Exceptions
MakeCatalogue	*mcIn* Company id PDF-flag	*mcOut* URL of the catalogue for that company	*mcFault* Invalid company id
Compare	*compIn* Company id Entry attribute (up to 6) Catalogue number (up to 4)	*compOut* URL of page showing comparison table	*compFault* Invalid company id Invalid catalogue number Unknown attribute
Lookup	*lookIn* Company id Catalogue number	*lookOut* URL of page with the item information	*lookFault* Invalid company id Invalid catalogue 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 Catalogue number Number of items required	*gdOut* Catalogue number Expected delivery date	*gdFault* Invalid company id Invalid catalogue number No availability Zero items requested
PlaceOrder	*poIn* Company id Number of items required Catalogue number	*poOut* Catalogue number Number of items required Predicted delivery date Unit price estimate Total price estimate	*poFault* Invalid company id Invalid catalogue number Zero items requested

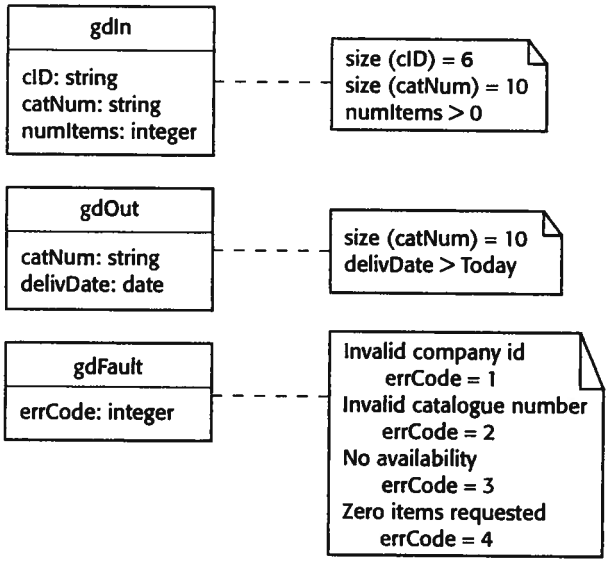


Figure 31.11 UML definition of input and output messages

used and 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 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 31.11 is a UML diagram that shows the structure of the input and output messages for the **getDelivery** operation in the catalogue 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 catalogue item, 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. 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.

31.2.3 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 services using a standard programming language such as Java or C#. Both of these languages now include libraries with extensive support for service development.

Alternatively, services may be developed by using existing components or, as I discuss below, 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. New services may also be developed by defining compositions of existing services. I discuss development by service composition in section 31.3.

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 discussed in Chapter 23), creating input messages that reflect these input combinations and 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. Various testing tools are now 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 behaviour is as specified.

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 to write a UDDI description so that potential users can discover the service. Erl (2004) provides a useful summary of UDDI in his book.

There are now a number of public registries for UDDI descriptions and businesses may also maintain their own private UDDI registries. A UDDI description consists of a number of different types of information:

1. Details of the business providing the service. 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 users to check a provider's credentials.
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. Information on where to find the WSDL specification associated with the service.
4. Subscription information that allows users to register for information about updates to the service.

A potential problem with UDDI specifications is that the functional behaviour of the service is specified informally as a natural language description. As I have discussed in Chapter 6, which covers software requirements, 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 ontology-based description where the specific meaning of terms in a description is specified in an ontology. A language called OWL-S has been developed for describing web service ontologies (OWL_Services_Coalition, 2003). At the time of writing, these techniques for semantic service specification are still immature but they are likely to become more widely used over the next few years.

31.2.4 Legacy system services

In Chapter 18, I discussed the possibility of implementing reusable components by providing a component interface to existing legacy systems. In essence, the functionality of the legacy systems could be reused. The implementation of the component was simply concerned with providing a general interface to that system. One of the most important uses of services is to implement such 'wrappers' for legacy systems. 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 maintenance database. 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 centre 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 centre 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 31.12, where I have used a UML stereotype to indicate a service. New applications simply exchange messages with these services to access the legacy system functionality.

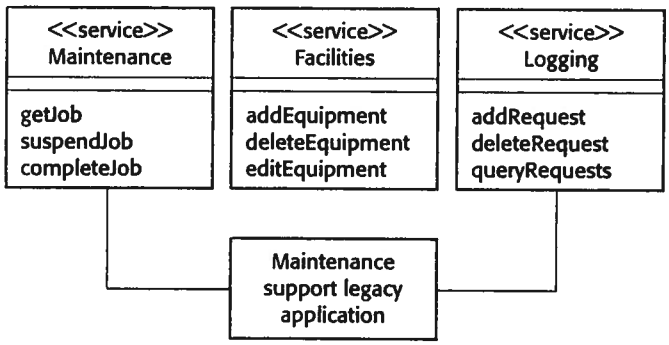


Figure 31.12 Services providing access to a legacy system

Some of the services provided are:

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. It also supports an operation to allow maintenance that has started but is incomplete to be suspended.
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.

The existing legacy system is not simply represented as a single service. Rather, the services that are developed are coherent and support a single area of functionality. This reduces their complexity and makes them easier to understand and reuse in other applications. I do not have space to discuss the details of the messages that might be exchanged by these services—their design is left as an exercise for the reader.

31.3 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 web user interface 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 some external provider.

Many companies are now concerned with converting applications that are used within an enterprise into service-oriented systems. This opens up the possibility of more widespread reuse within the company. The next stage will be the development of inter-organisational applications between trusted suppliers. The final realisation of the long-term vision of service-oriented architectures will rely on the development of a 'services market'. I think it is unlikely that this will emerge during the lifetime of this book. At the time of writing, only a relatively small number of business services that might be included in business applications are publicly available.

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 travellers. As well as booking their flights, travellers can also book hotels in their preferred location, arrange car hire 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 services with services offered by a hotel booking agency, car hire and taxi companies and the reservation services offered by the providers of the local attractions. The result is a single service that integrates these different services from different providers.

You can think of this process as a sequence of separate steps as shown in Figure 31.13. Information is passed from one step to the next—for example, the car hire 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. You can think of a workflow as a model of a business process—the steps involved in reaching some 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 reality, 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 traveller was disabled and required a wheelchair to be rented and delivered to the airport.

You must to be able to cope with situations where the workflow has to be changed because the normal execution of one of the services results in an incompatibility with some other service execution. For example, say a flight is booked to leave on 1 June and return on 7 June. The workflow then proceeds to the hotel booking stage. However, the resort is hosting a major convention until 2 June so no hotel rooms

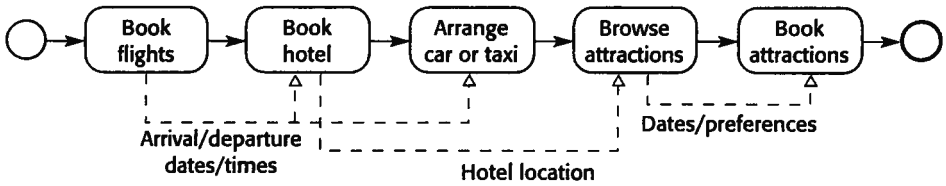


Figure 31.13
Vacation package workflow

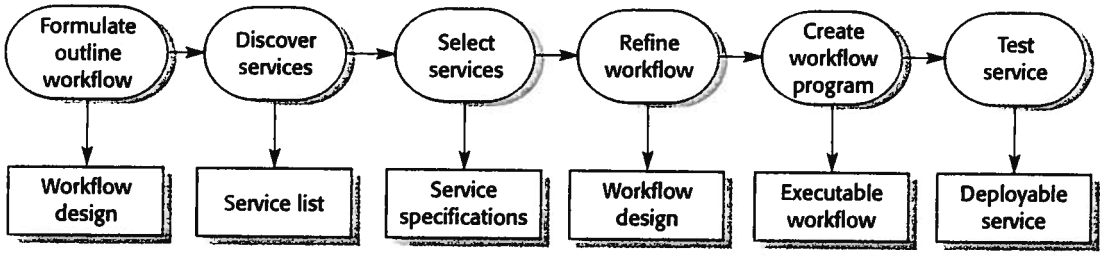


Figure 31.14 Service construction by composition

are available. The hotel booking service reports this lack of availability. This is not a failure: lack of availability is a common situation. You 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, a 'compensating action' is used to undo actions that have already been completed.

The process of designing new services by composing existing services is, essentially, a process of software design with reuse (Figure 31.14). 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 31.14, 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 to discover what services exist, who provides these services and the details of the service provision.
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 then may 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 you can use a more specialised 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 31.3.2.

In the remainder of this chapter, I focus on workflow design and testing. As I discussed in the introduction, a market for services has not yet developed. Although a number of public UDDI registries are available, these are sparsely populated and the service descriptions are sometimes vague and incomplete. For these reasons, service discovery is not yet a major issue. Most services will be discovered within organisations where services can be discovered using internal registries and informal communications between software engineers.

31.3.1 Workflow design and implementation

Workflow design involves analysing existing or planned business processes to understand the different stages of these processes then representing the process being designed in a workflow design notation. This shows 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. In such cases, you have to use process knowledge to design a workflow that achieves the same goals as current business processes.

Workflows represent business process models and are usually represented using a graphical notation such as BPMN (White, 2004) or YAWL (van der Aalst and ter Hofstede, 2005) At the time of writing, the process modelling language which seems most likely to emerge as a standard is BPMN. This is a graphical language which 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 31.2. I use BPMN here to illustrate the notion of business process programming.

Figure 31.15 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,

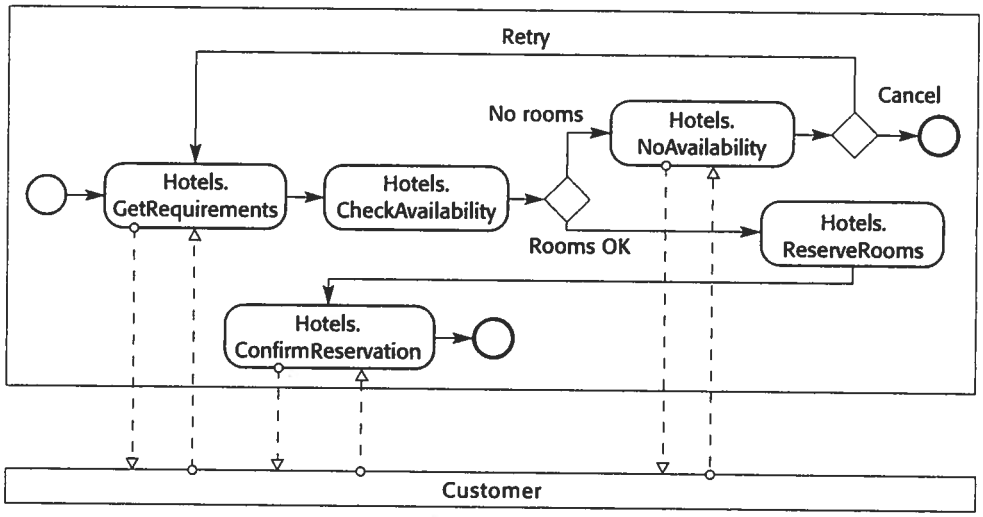


Figure 31.15 Hotel booking workflow

checking room availability 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 a 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 31.15, 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 31.15, 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 do not have space to describe here. These add information to a business process description that allows it to be automatically translated into an executable form. Therefore, web services, based on service compositions described in BPMN can be created from a business process model.

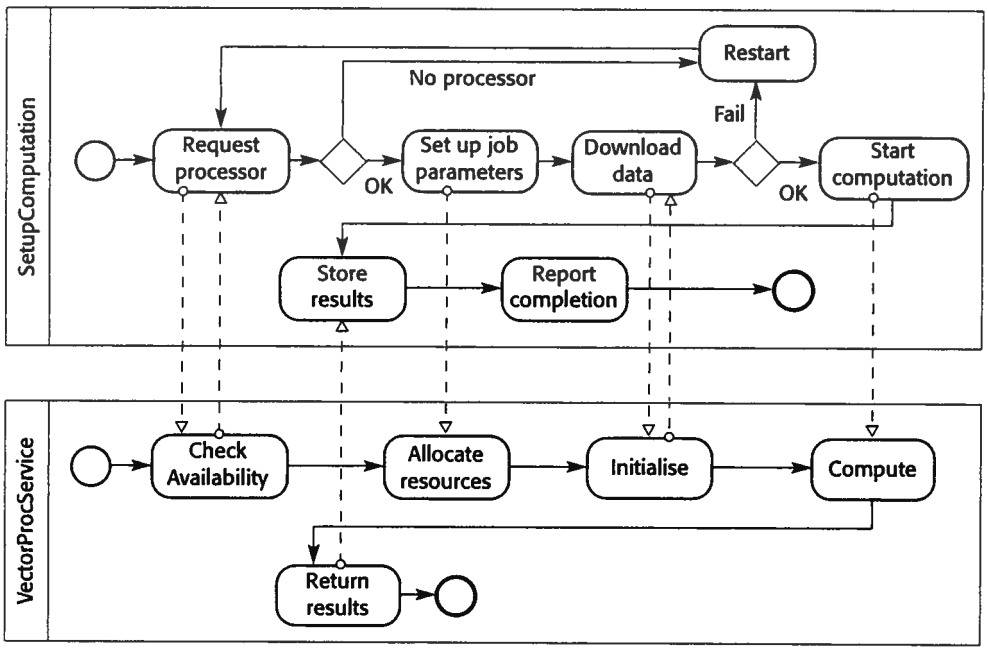


Figure 31.16
Interacting workflows

Figure 31.15 shows the process that is enacted in one organisation, the company that provides a booking service. However, the key benefit of a service-oriented approach is that it supports inter-organisational computing. This means that the total computation involves services in different companies. This is represented in BPMN by developing separate workflows for each of the organisations involved with interactions between them.

To illustrate this, I use a different example, drawn from grid 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 31.16.

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, initialises the system, carries out the computation and returns the results to the client service.

In BPMN terms, the workflow for each organisation 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 organisation are involved in a workflow, this can be shown by separating pools into named 'lanes'. Each lane shows the activities in that part of the organisation.

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 31.14, the model may go through a number of iterations until a design that allows the maximum possible reuse of available services is created. Once such a design is available, the next stage is to convert this to an executable program. As services are implementation-language independent, this can be written in any language and both Java and C# development environments provide support for web service composition.

To provide direct support for the implementation of web service compositions, several web service standards have been developed. The best known of these is WS-BPEL (Business Process Execution Language) which is an XML-based '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).

All of these are XML standards so the resulting descriptions are long and difficult to read. Writing programs directly in XML-based notations is slow and error-prone. I have therefore decided not to go into details of XML-based notations, such as WS-BPEL, as they are not essential for understanding the principles of workflow and service-composition. As support for service-oriented computing matures, these XML descriptions will be generated automatically. Tools will parse a graphical workflow description and generate executable service compositions.

31.3.2 Service testing

Testing is important in all system development processes to help demonstrate that a system meets its functional and non-functional requirements and to detect defects that have been introduced during the development process. As I have discussed in Chapters 22–24, a range of different approaches to system validation and testing have been developed to support the testing process. Many of these 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 operation 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 testing experience. 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 service-oriented architectures 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 some particular service but it cannot be guaranteed that that service will be used during an actual execution of the system.

3. As, in most cases, a service is available to different customers, the non-functional behaviour of that 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 behaviour may be different because of the demands made by other 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; 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 some 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 failures 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 is currently an important research issue.

KEY POINTS

- Service-oriented software engineering is based on the notion that programs can be constructed by composing independent services that encapsulate reusable functionality. Services are language independent and their implementation is based on widely adopted XML-based standards.
- Service interfaces are 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 some 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, testing and deploying the service.
- Service interfaces may be defined for legacy software systems that continue to be useful for an organisation. 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 some business process. Activities in the business process may be implemented by services so that the business process model represents a service composition.
- Techniques of software testing based on source-code analysis cannot be used in service-oriented systems that rely on externally provided services.

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.

Erl, T. (2004). *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, Upper Saddle River, NJ: Prentice-Hall.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology and Design*, Upper Saddle River, NJ: Prentice-Hall.

EXERCISES

31.1 What are the important distinctions between services and software components?

31.2 Explain why service-oriented architectures should be based on standards.

31.3 Why is it important to minimise the number of messages exchanged by services?

31.4 Explain why services should always include an exception interface which is used to report faults and exceptions to service clients.

31.5 Using the same notation, extend Figure 31.6 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.

31.6 Define an interface specification for the Currency Converter and Check credit rating services shown in Figure 31.8.

31.7 Design possible input and output messages for the services shown in Figure 31.12. You may specify these in the UML or in XML.

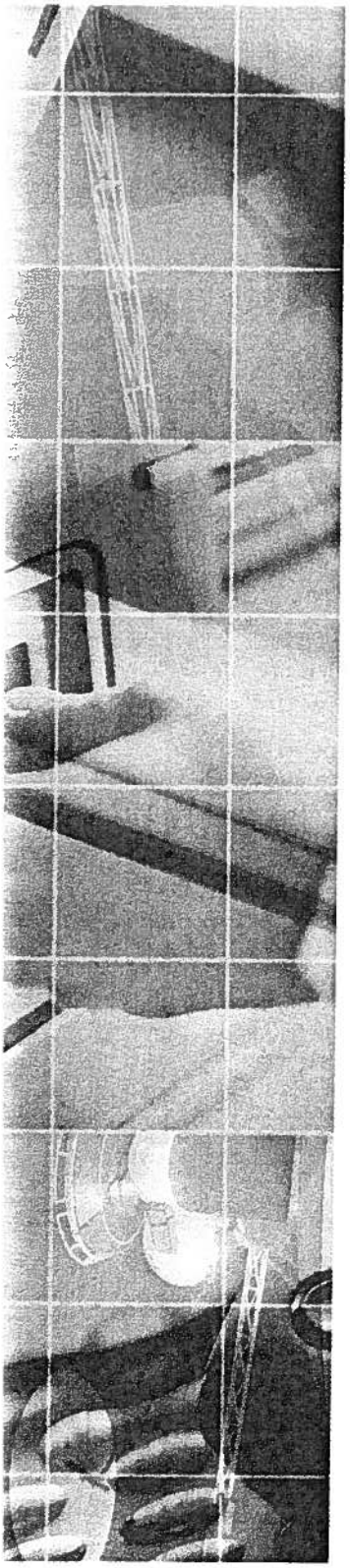
31.8 Giving reasons for your answer, suggest two important types of application that are unlikely to make use of a service-oriented approach.

31.9 In section 31.2.1, I introduced an example of a company that has developed a catalogue service that is used by the web-based procurement systems used by customers. Using BPMN, design a workflow that uses this catalogue service to lookup and place orders for computer equipment.

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

31.11 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 a hire car. You may assume that the taxi and car hire companies offer web services to make a reservation.

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



32 Aspect-oriented software development

Objectives

The objective of this chapter is to introduce you to aspect-oriented software development, which is based on the idea of separating concerns into separate system modules. When you have read this chapter, you will:

- understand why the separation of concerns is a good guiding principle for software development;
- have been introduced to the fundamental ideas underlying aspects and aspect-oriented software development;
- understand how to use an aspect-oriented approach for requirements engineering, software design and programming;
- know the problems of testing aspect-oriented systems.

Contents