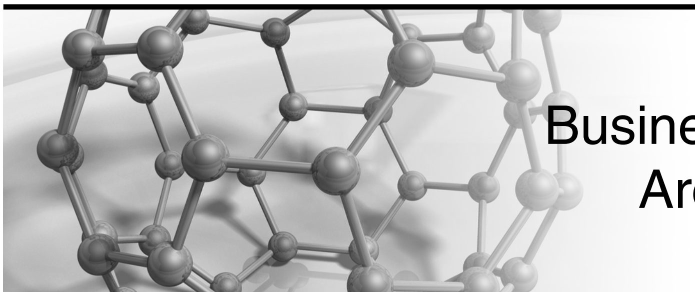

Chapter 2

Business Integration Architecture and Patterns

A typical business integration project involves coordinating several different IT assets, potentially running on different platforms, and having been developed at different times using different technologies. Being able to easily manipulate and exchange information with a diverse set of components is a major technical challenge. It is best addressed by the programming model used to develop business integration solutions.

This chapter explores the fundamentals of the business integration programming model. It introduces the Service Component Architecture (SCA) and discusses patterns related to business integration. Patterns seem to permeate our lives. Sewing patterns, think-and-learn patterns for children, home construction patterns, wood-carving patterns, flight patterns, wind patterns, practice patterns in medicine, customer buying patterns, workflow patterns, design patterns in computer science, and many more exist.

Patterns have proven successful in helping solution designers and developers. Therefore, it is not surprising that we now have business integration patterns or enterprise integration patterns. In the referenced literature, you will find a wide array of patterns that are applicable to business integration, including patterns for request and response routing, channel patterns (such as publish/subscribe), and many more. Abstract patterns provide a template for resolving a certain category of problems, whereas concrete patterns provide more specific indications of how to implement a specific solution. This chapter focuses on patterns that deal with data and service invocation, which are at the foundation of the programming model of the IBM software strategy for WebSphere business integration. dW A.2.1

Business Integration Scenarios

Enterprises have many different software systems that they use to run their business. In addition, they have their own ways of integrating these business components. The two most prevalent business integration scenarios are as follows:

- **Integration broker:** In this use case, the business integration solution acts as an intermediary located among a variety of "back-end" applications. For example, you might need to ensure that when a customer places an order using the online order management application, the transaction updates relevant information in your Customer Relationship Management (CRM) back end. In this scenario, the integration solution needs to be able to capture and possibly transform the necessary information from the order management application and invoke the appropriate services in the CRM application.
- **Process automation:** In this scenario, the integration solution acts as the glue among different IT services that would otherwise be unrelated. For example, when a company hires an employee, the following sequence of actions needs to occur:
  - The employee's information is added to the payroll system.
  - The employee needs to be granted physical access to the facilities, and a badge needs to be provided.
  - The company might need to assign a set of physical assets to the employee (office space, a computer, and so on).
  - The IT department needs to create a user profile for the employee and grant access to a series of applications.

  Automating this process is also a common use case in a business integration scenario. In this case, the solution implements an automated flow that is triggered by the employee's addition to the payroll system. Subsequently, the flow triggers the other steps by creating work items for the people who are responsible for taking action or by calling the appropriate services.

In both scenarios, the integration solution needs to do the following:

- Work with disparate sources of information and different data formats, and be able to convert information between different formats
- Be able to invoke a variety of services, potentially using different invocation mechanisms and protocols

Throughout this book, we illustrate how the foundational programming model of IBM's WebSphere Process Server (WPS) addresses these requirements.

Business Integration: Roles, Products, and Technical Challenges

Successful business integration projects require a few basic ingredients:

- A clear separation of roles in the development organization to promote specialization, which typically improves the quality of the individual components that are developed
- A common business object (BO) model that enables business information to be represented in a canonical format
- A programming model that strongly separates interfaces from implementations and that supports a generic service invocation mechanism that is totally independent of the implementation and that only involves dealing with interfaces
- An integrated set of tools and products that supports development roles and preserves their separation

The following sections elaborate on each of these ingredients.

Clear Separation of Roles

A business integration project requires people in four collaborative, but distinctly separate, roles:

- **Business analyst:** Business analysts are domain experts responsible for capturing the business aspects of a process and for creating a process model that adequately represents the process itself. Their focus is to optimize the financial performance of a process. Business analysts are not concerned with the technical aspects of implementing processes.
- **Component developer:** Component developers are responsible for implementing individual services and components. Their focus is the specific technology used for the implementation. This role requires a strong programming background.
- **Integration specialist:** This relatively new role describes the person who is responsible for assembling a set of existing components into a larger business integration solution. Integration developers do not need to know the technical details of each of the components and services they reuse and wire together. Ideally, integration developers are concerned only with understanding the interfaces of the services that they are assembling. Integration developers should rely on integration tools for the assembly process.
- **Solution deployer:** Solution deployers and administrators are concerned with making business integration solutions available to end users. Ideally, a solution deployer is primarily concerned with binding a solution to the physical resources ready for it to function (databases, queue managers, and so on) and not with having a deep understanding of the internals of a solution. The solution deployer's focus is quality of service (QoS).

A Common Business Object Model

As we discussed, the key aspects of a business integration project include the ability to coordinate the invocation of several components and the ability to handle the data exchange among those. In particular, different components can use different techniques to represent business items such as the data in an order, a customer's information, and so on. For example, you might have to integrate a Java application that uses entity Enterprise Java Beans

(EJBs) to represent business items and a legacy application that organizes information in COBOL copybooks. Therefore, a platform that aims to simplify the creation of integration solutions should also provide a generic way to represent business items, irrespective of the techniques used by the back-end systems for data handling. This goal is achieved in WPS and WebSphere Enterprise Service Bus (WESB) thanks to the *business object framework*.

The business object framework enables developers to use XML Schemas to define the structure of business data and access and manipulate instances of these data structures (business objects) via XPath or Java code. The business object framework is based on the Service Data Object (SDO) standard.

The Service Component Architecture (SCA) Programming Model

The SCA programming model represents the foundation for any solution to be developed on WPS and WESB.

SCA provides a way for developers to encapsulate service implementations in reusable components. It enables you to define interfaces, implementations, and references in a technology-agnostic way, giving you the opportunity to bind the elements to whichever technology you choose.

There is also an SCA client programming model that enables the invocation of those components. In particular, it enables runtime infrastructures based on Java—such as IBM's WebSphere Process Server, BEA's WebLogic Server (with its Aqualogic product family), and Oracle's Application Server (part of Oracle's Fusion Middleware family)—to interact with non-Java runtimes. SCA uses business objects as the data items for service invocation.

Tools and Products

IBM's WebSphere Integration Developer is the integrated development environment that has all the necessary tools to create and compose business integration solutions based on the technologies just mentioned. These solutions typically are deployed to the WPS or, in some cases, to the WESB—the products that are at the center of this book.

Now that you understand the key ingredients of business integration solutions, let's take a look at the business object framework, at SCA, and at some of the key patterns, processes, and qualifiers in more detail.

The Business Object Framework

The computer software industry has developed several programming models and frameworks that enable developers to encapsulate business object information. In general, a BO framework should provide database independence, transparently map custom business objects to database tables, and bind business objects to user interfaces. Of late, XML schemas are perhaps the most popular and accepted way to represent the structure of a business object.

From a tooling perspective, WebSphere Integration Developer (WID) provides developers with a common BO model for representing different kinds of entities from different domains. At development time, WID represents business objects as XML schemas. At runtime, however, those same business objects are represented in memory by a Java instance of an SDO. SDO is a standard specification that IBM and BEA Systems have jointly developed and agreed on. IBM has extended the SDO specification by including some additional services that facilitate the manipulation of data within the business objects. We'll discuss some of these later in this chapter.

Before we get into the BO framework, let's look at the basic types of data that get manipulated:

- **Instance data** is the actual data and data structures, from simple, basic objects with scalar properties to large, complex hierarchies of objects. This also includes data definitions such as a description of the basic attribute types, complex type information, cardinality, and default values.
- **Instance metadata** is instance-specific data. Incremental information is added to the base data, such as change tracking (also known as change summary), context information associated with how the object or data was created, and message headers and footers.
- **Type metadata** is usually application-specific information, such as attribute-level mappings to destination enterprise information system (EIS) data columns (for example, mapping a BO field name to a SAP table column name).
- **Services** are basically helper services that get data, set data, change summary, or provide data definition type access.

Table 2.1 shows how the basic types of data are implemented in the WebSphere platform.

Table 2.1 Data Abstractions and the Corresponding Implementations

Data Abstraction	Implementation
Instance data	Business object (SDO)
Instance metadata	Business graph
Type metadata	Enterprise metadata
	Business object type metadata
Services	Business object services

Working with the IBM Business Object Framework

As we mentioned, the WPS BO framework is an extension of the SDO standard. Therefore, business objects exchanged between WPS components are instances of the *commonj.sdo.DataObject* class. However, the WPS BO framework adds several services and functions that simplify and enrich the basic *DataObject* functionality.

To facilitate the creation and manipulation of business objects, the WebSphere BO framework extends SDO specifications by providing a set of Java services. These services are part of the package named *com.ibm.websphere.bo*:

- **BOFactory:** The key service that provides various ways to create instances of business objects.
- **BOXMLSerializer:** Provides ways to "inflate" a business object from a stream or to write the content of a business object, in XML format, to a stream.
- **BOCopy:** Provides methods that make copies of business objects ("deep" and "shallow" semantics).
- **BODataObject:** Gives you access to the data object aspects of a business object, such as the change summary, the business graph, and the event summary.
- **BOXMLDocument:** The front end to the service that lets you manipulate the business object as an XML document.
- **BOChangeSummary and BOEventSummary:** Simplifies access to and manipulation of the change summary and event summary portion of a business object.
- **BOEquality:** A service that enables you to determine whether two business objects contain the same information. It supports both deep and shallow equality.
- **BOType and BOTypeMetaData:** These services materialize instances of *commonj.sdo.Type* and let you manipulate the associated metadata. Instances of *Type* can then be used to create business objects "by type."

Chapter 4, "WebSphere Integration Developer," introduces the facilities that WID provides to enable you to quickly formulate your object definitions.

Service Component Architecture

A.2.2

SCA is an abstraction you can implement in many different ways. It does not mandate any particular technology, programming language, invocation protocol, or transport mechanism. SCA components are described using Service Component Definition Language (SCDL), which is an XML-based language. You could, in theory, create an SCDL file manually. In practice, you're more likely to use an integrated development environment (IDE) such as WebSphere Integration Developer to generate the SCDL file.

An SCA component has the following characteristics:

- It wraps an implementation artifact, which contains the logic that the component can execute.
- It exposes one or more interfaces.

- It can expose one or more references to other components. The implementation's logic determines whether a component exposes a reference. If the implementation requires invoking other services, the SCA component needs to expose a reference.

This chapter focuses on the SCA implementation that WPS offers and the WID tool that is available to create and combine SCA components. WPS and WID support the following implementation artifacts:

- Plain Java objects
- Business Process Execution Language (BPEL) processes
- Business state machines
- Human tasks
- Business rules
- Selectors
- Mediations

SCA separates business logic from infrastructure so that application programmers can focus on solving business problems. IBM's WPS is based on that same premise. Figure 2.1 shows the architectural model of WPS.

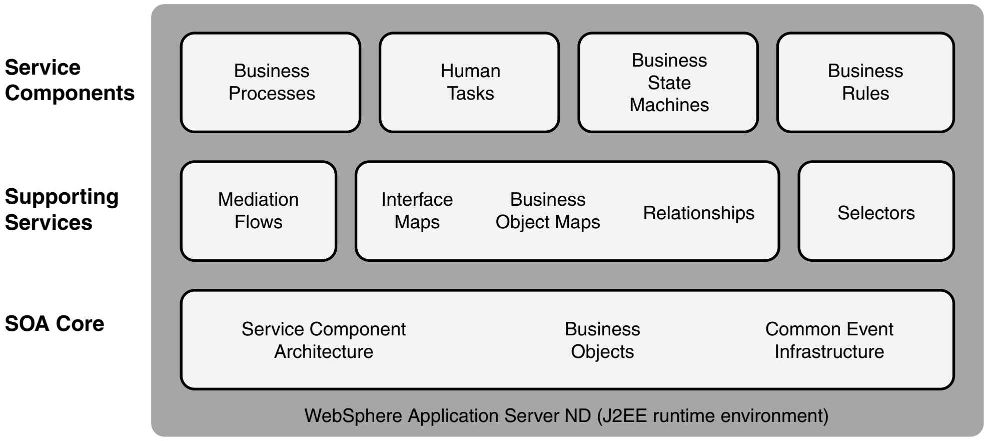

Figure 2.1 Architectural model for WPS

In the WebSphere environment, the SCA framework is based on the Java 2 Platform, Enterprise Edition (J2EE) runtime environment of WebSphere Application Server. The overall WebSphere Process Server framework consists of SOA Core, Supporting Services, and the Service Components. The same framework with a subset of this overall capability, targeted more specifically at the connectivity and application integration needs of business integration, is available in WESB.

The interface of an SCA component, as illustrated in Figure 2.2, can be represented as one of the following:

- A Java interface
- A WSDL port type (in WSDL 2.0, port type is called interface)

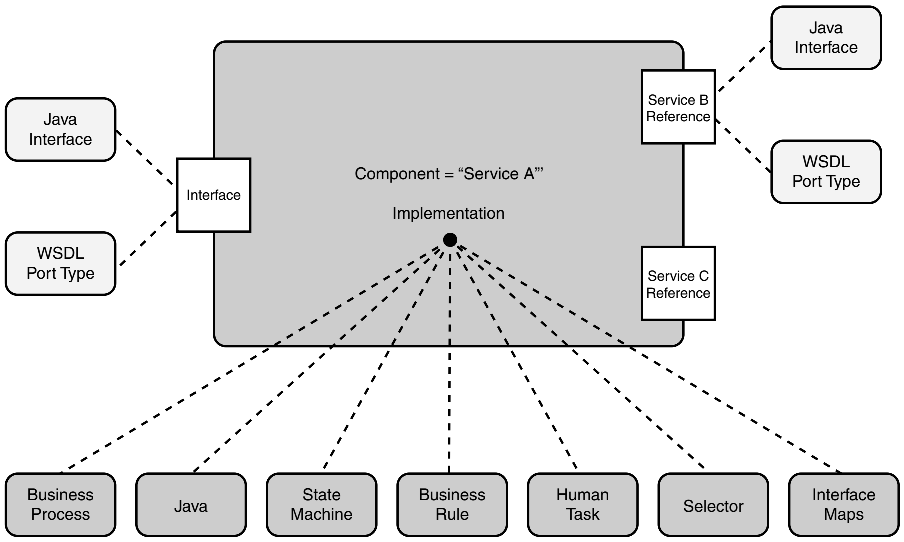

Figure 2.2 SCA in WPS

An SCA module is a group of components wired together by directly linking references and implementations. In WID, each SCA module has an *assembly diagram* associated with it, which represents the integrated business application, consisting of SCA components and the wires that connect them. One of the main responsibilities of the integration developer is to create the assembly diagram by connecting the components that form the solution. WID provides a graphical Assembly Editor to assist with this task. When creating the assembly diagram, the integration developer can proceed in one of two ways:

- **Top-down** defines the components, their interfaces, and their interactions before creating the implementation. The integration developer can define the structure of the process, identify the necessary components and their implementation types, and then generate an implementation skeleton.
- **Bottom-up** combines existing components. In this case, the integration developer simply needs to drag and drop existing implementations onto the assembly diagram.

The bottom-up approach is more commonly used when customers have existing services that they want to reuse and combine. When you need to create new business objects from scratch, you are likely to adopt the top-down approach. Chapter 4 introduces the various wizards in WID and lays out the six phases of creating a simple module using the top-down approach.

The SCA Programming Model: Fundamentals

The concept of a software *component* forms the basis of the SCA programming model. As we mentioned, a component is a unit that implements some logic and makes it available to other components through an interface. A component may also require the services made available by other components. In that case, the component exposes a *reference* to these services.

In SCA, every component must expose at least one interface. The assembly diagram shown in Figure 2.3 has three components—C1, C2, and C3. Each component has an interface that is represented by the letter I in a circle. A component can also refer to other components. References are represented by the letter R in a square. References and interfaces are then linked in an assembly diagram. Essentially, the integration developer "resolves" the references by connecting them with the interfaces of the components that implement the required logic.

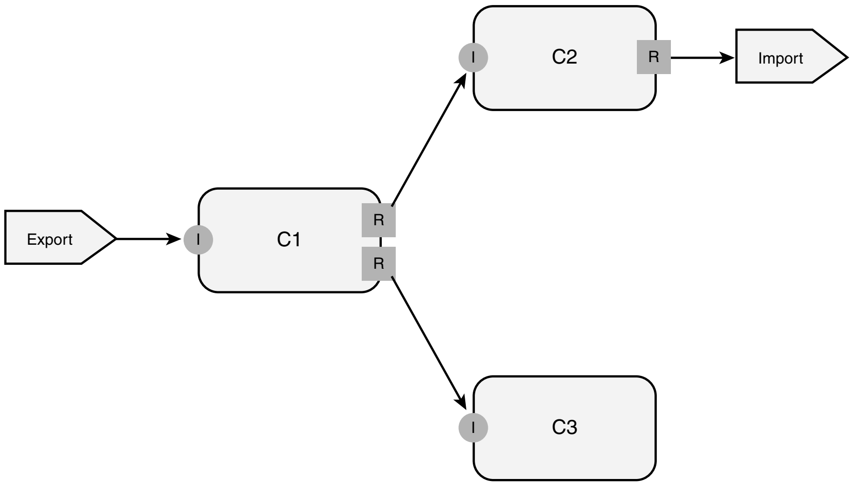

Figure 2.3 Assembly diagram

Invoking SCA Components

To provide access to the services to be invoked, the SCA programming model includes a *ServiceManager* class, which enables developers to look up available services by name. Here is a typical Java code fragment illustrating service lookup. The ServiceManager is used to obtain a reference to the BOFactory service, which is a system-provided service:

```
//Get service manager singleton
ServiceManager smgr = new ServiceManager();
//Access BOFactory service
BOFactory bof =(BOFactory)
        mgr.locateService("com/ibm/websphere/bo/BOFactory");
```

Developers can use a similar mechanism to obtain references to their own services by specifying the name of the service referenced in the *locateService* method. We illustrate in more detail the usage of the SCA programming model in Chapter 11, "Business Integration Programming." For the time being, we want to emphasize that after you have obtained a reference to a service using the *ServiceManager* class, you can invoke any of the available operations on that service in a way that is independent of the invocation protocol and the type of implementation.

SCA components can be called using three different invocation styles:

- **Synchronous invocation:** When using this invocation style, the caller waits synchronously for the response to be returned. This is the classic invocation mechanism.
- **Asynchronous invocation:** This mechanism allows the caller to invoke a service without waiting for the response to be produced right away. Instead of getting the response, the caller gets a "ticket," which can be used later to retrieve the response. The caller retrieves the response by calling a special operation that must be provided by the callee for this purpose.
- **Asynchronous invocation with callback:** This invocation style is similar to the preceding one, but it delegates the responsibility of returning the response to the callee. The caller needs to expose a special operation (the callback operation) that the callee can invoke when the response is ready.

Imports

Sometimes, business logic is provided by components or functions that are available on external systems, such as legacy applications, or other external implementations. In those cases, the integration developer cannot resolve the reference by connecting a reference to a component containing the implementation he or she needs to connect the reference to a component that "points to" the external implementation. Such a component is called an *import*. When you define an import, you need to specify how the external service can be accessed in terms of location and the invocation protocol.

Exports

Similarly, if your component has to be accessed by external applications, which is quite often the case, you must make it accessible. That is done by using a special component that exposes your logic to the "outside world." Such a component is called an *export*. These can also be invoked synchronously or asynchronously.

Stand-alone References

In WPS, an SCA service module is equivalent to a J2EE EAR file and contains several other J2EE submodules. J2EE elements, such as a WAR file, can be packaged along with the SCA module. Non-SCA artifacts such as JSPs can also be packaged together with an SCA service module. This lets them invoke SCA services through the SCA client programming model using a special type of component called a stand-alone reference.

The SCA programming model is strongly declarative. Integration developers can configure aspects such as transactional behavior of invocations, propagation of security credentials, whether an invocation should be synchronous or asynchronous in a declarative way, directly in the assembly diagram. The SCA runtime, not the developers, is responsible for taking care of implementing the behavior specified in these modifiers. The declarative flexibility of SCA is one of the most powerful features of this programming model. Developers can concentrate on implementing business logic, rather than focusing on addressing technical aspects, such as being able to accommodate asynchronous invocation mechanisms. All these aspects are automatically taken care of by the SCA runtime. The declarative aspects of SCA programming are discussed in Chapter 11.

Business Integration Patterns

Patterns help architects, designers, and developers use a common vocabulary to efficiently describe their solutions. This section discusses patterns at the business integration level, which fall into two classes: intraenterprise and interenterprise. These break down into Enterprise Application Integration (EAI) scenarios and business-to-business (B2B) scenarios. The EAI patterns deal mainly with application integration and database replication, whereas the B2B patterns deal with data exchange and process integration.

Another classification of business integration or Enterprise Integration Pattern is based on messaging architectures and messaging specifications such as Java Messaging Service (JMS) and Web services. The common categories in this case are channel patterns, endpoint patterns, routing patterns, system management patterns, and transformation patterns. dW A.2.3

Data Exchange Patterns

A data exchange pattern is a rather simple pattern that is predicated on an agreement to use a common data format. The idea behind the SDO standard is to enable Java applications to support the following three common data exchange patterns. Actually, these are mechanisms that you can implement using different design patterns:

- The Plain Business Object pattern
- The Disconnected Object pattern
- The Event pattern

The Plain Business Object Pattern

The Plain Business Object pattern, also known as the Document pattern, is the most common data exchange mechanism. A client component populates a business object and then submits the object to a service by invoking a specific operation. Chapter 4 discusses defining and using plain business objects.

The Disconnected Object Pattern

The Disconnected Object pattern or the Transfer Object pattern is another common data exchange mechanism. Its primary purpose is to minimize the amount of time an application needs to be connected to the back end when making changes to an object. It extends the Transfer Object pattern of the Model-View-Controller architecture.

The Disconnected Object pattern enables an application to get hold of data from a back-end system and manipulate the data without requiring a connection to the back end. Then the application makes the changes persistent in a separate transaction that involves connectivity with the back end. The pattern extends this concept by introducing a mechanism to track the changes made to the business object or to any of the business objects contained by that business object.

In the Disconnected Object pattern, a system uses the change history in addition to the information stored in the object itself. The mechanism to track changes is called a business graph (BG), and it essentially includes four things:

- A *copy of the business object data* that can be manipulated even when no connection exists to the repository that holds the business object itself
- A *change history*, which stores the values of the business object before any manipulation occurred (for example, the original values), with an indication of the operations that modified the state of the business object
- An *event summary*, which contains the identifiers of objects affected by a change and event information recording the actual data involved in the business transaction
- A *verb*, which specifies the kind of operation that was performed (for example, a create or delete operation)

The Event Pattern

The Event pattern is the data exchange mechanism used when an event is produced by an EIS and is injected into WPS through an adapter. Typically, an adapter captures an event that occurred in a back-end system and creates the appropriate business object within WPS. For instance, in a travel reservation application that maintains profiles for travelers, a user might add a new airline company to the list of preferred airlines in his or her traveler's profile. Using the event pattern, that operation is captured as an "update" event that involves a certain traveler ID and a certain airline ID. The appropriate business objects will be materialized in WPS for the benefit of the business processes that can manipulate them.

The Event pattern makes use of the "delta image" information, which is a recording of what was changed in the original object. In our example, the addition of an airline to the list of preferred airlines would be the delta image. The Event pattern becomes important when you design complex relationships between different back ends, especially in integration scenarios that require keeping information about equivalent entities, managed by different applications, synchronized.

Object Mapping Pattern

Every so often, you will hear the term GBO and ASBO. GBO stands for Generic Business Object, and ASBO stands for Application-Specific Business Object. This terminology did not catch on, but we see ASBO-to-GBO and GBO-to-ASBO patterns all the time in business integration solutions.

Many patterns have limitations, which are overcome by the use of a complementary or associated pattern. The Process Integration pattern takes the limitations raised by the Data Exchange pattern and addresses them by providing Business Process Integration (BPI) services. The common underlying entity in both cases is the exchange of XML-based documents, which permits richer, more complex relationships.

Business Processes

Business processes—specifically, BPEL-based business processes—form the cornerstone of service components in the SCA. Whether it is a simple order approval or a complex manufacturing process, enterprises have always had business processes. A business process is a set of activities, related to the business, that are invoked in a specific sequence to achieve a business goal. In the business integration world, a business process is defined using some kind of markup language.

These business processes can invoke other supporting services or contain other service components such as business state machines, human tasks, business rules, or data maps. And, when deployed, these processes can either get done quickly or run over a long period of time. Sometimes, these processes can run for years.

Like most components in the J2EE world, business processes run in a container. In IBM's WebSphere platform, this specific container is called the Business Process Choreographer, which we talk about in Chapter 9, "Business Integration Clients." The container or the BPEL engine provides all the services and process lifecycle requirements.

Qualifiers

In a word, qualifiers are rules. Qualifiers define how much management should be provided by WPS for a component at runtime. A process application communicates its QoS needs to the WPS runtime environment by specifying service qualifiers. The qualifiers govern the interaction between a service client and a target service. Qualifiers can be specified on service component references, interfaces, and implementations and are always external to an implementation. The different categories of qualifiers include the following:

- Transaction, which specifies the way transactional contexts are handled in an SCA invocation
- Activity session, which specifies how Activity Session contexts are propagated. An Activity Session extends the concept of transaction, to encompass a number of related transactions.
- Security, which specifies the permissions
- Asynchronous reliability—rules for asynchronous message delivery

SCA allows these QoS qualifiers to be applied to components declaratively (without requiring programming or a change to the services implementation code). This is done in WID. Table 2.2 lists the different types of qualifiers, along with their qualifying values. Usually, you apply QoS qualifiers when you are ready to consider solution deployment. Look for more details about QoS, especially event sequencing, in Chapter 11.

Table 2.2 Adding References to the Application Deployment Descriptor

Name	Qualifier Values
Reference qualifiers	Asynchronous reliability Suspend transaction Asynchronous invocation Suspend activity session
Interface qualifiers	Event sequencing Join activity session Join transaction Security permission
Implementation qualifiers	Activity session Transaction Security identity

Closing the Link

Imports, exports, references, and so on are new terms in this whole new paradigm called the Service Component Architecture (SCA), which was introduced in this chapter. From the developer's perspective, services are packaged in a service module, which is the basic unit of deployment and administration in an SCA runtime.

Service imports are used in a service module to use external services that are not part of the module itself (for example, services exported by other modules, stateless session EJBs, Web services, EIS services, and so on). External services that are referenced by import declarations are valid targets of service wires. The import binding definition does not need to be finalized at development time. Aspects such as the actual endpoint of services to invoke can be late-bound at deployment, administration, or even runtime. Service exports, on the other hand, are used to offer services from the service module to the outside world, such as services for other service modules or as Web services.

The following chapters build on these concepts and use the tooling and runtime examples to fully explain SCA. It is imperative that you understand the SCA model, because it forms the basis of business integration discussed in this book. It is beneficial to reiterate that patterns are not inventions or edicts; they are harvested from repeated experiences from and use by practitioners. Business integration does not always involve business processes, but in a lot of cases, business processes form the centerpiece of integration.

Links to developerWorks

A.2.1 www.ibm.com/developerworks/podcast/websphere/ws-soa2progmod.html

A.2.2 www.ibm.com/developerworks/websphere/library/techarticles/0610_redlin/0610_redlin.html

A.2.3 www.ibm.com/developerworks/websphere/techjournal/0508_simmons/0508_simmons.html