IT2401 SERVICE ORIENTED ARCHITECTURE  

L T P C  3 0 0 3

OBJECTIVES:
- To gain understanding of the basic principles of service orientation
- To learn service oriented analysis techniques
- To learn technology underlying the service design
- To learn advanced concepts such as service composition, orchestration and Choreography
- To know about various WS-* specification standards

UNIT I
9

UNIT II
9
Web services – Service descriptions – Messaging with SOAP –Message exchange Patterns – Coordination –Atomic Transactions – Business activities – Orchestration – Choreography - Service layer abstraction – Application Service Layer – Business Service Layer – Orchestration Service Layer

UNIT III
9

UNIT IV
9
SOA platform basics – SOA support in J2EE – Java API for XML-based web services (JAX-WS) - Java architecture for XML binding (JAXB) – Java API for XML Registries (JAXR) - Java API for XML based RPC (JAX-RPC)- Web Services Interoperability Technologies (WSIT) - SOA support in .NET – Common Language Runtime - ASP.NET web forms – ASP.NET web services – Web Services Enhancements (WSE)

UNIT V
9
WS-BPEL basics – WS-Coordination overview - WS-Choreography, WS-Policy, WSSecurity

TOTAL = 45 PERIODS

TEXT BOOKS:

REFERENCES:
4.3. The roots of SOA (comparing SOA to past architectures)

4.3.1. What is architecture?

For as long as there have been computerized automation solutions, technology architecture has existed. However, in older environments, the construction of the solution was so straightforward that the task of abstracting and defining its architecture was seldom performed. With the rise of multi-tier applications, the variations with which applications could be delivered began to dramatically increase. IT departments started to recognize the need for a standardized definition of a baseline application that could act as a template for all others. This definition was abstract in nature, but specifically explained the technology, boundaries, rules, limitations, and design characteristics that apply to all solutions based on this template. This was the birth of the application architecture.

Application architecture

Application architecture is to an application development team what a blueprint is to a team of construction workers. Different organizations document different levels of application architecture. Some keep it high-level, providing abstract physical and logical representations of the technical blueprint. Others include more detail, such as common data models, communication flow diagrams, application-wide security requirements, and aspects of infrastructure.

It is not uncommon for an organization to have several application architectures. A single architecture document typically represents a distinct solution environment. For example, an organization that houses both .NET and J2EE solutions would very likely have separate application architecture specifications for each.

A key part of any application-level architecture is that it reflects immediate solution requirements, as well as long-term, strategic IT goals. It is for this reason that when multiple application architectures exist within an organization, they are almost always accompanied by and kept in alignment with a governing enterprise architecture.

Enterprise architecture

In larger IT environments, the need to control and direct IT infrastructure is critical. When numerous, disparate application architectures co-exist and sometimes even integrate, the demands on the underlying hosting platforms can be complex and onerous. Therefore, it is common for a master specification to be created, providing a high-level overview of all forms of heterogeneity that exist within an enterprise, as well as a definition of the supporting infrastructure.

Continuing our previous analogy, an enterprise architecture specification is to an organization what an urban plan is to a city. Therefore, the relationship between an urban plan and the blueprint of a building are comparable to that of enterprise and application architecture specifications.

Typically, changes to enterprise architectures directly affect application architectures, which is why architecture specifications often are maintained by the same group of individuals. Further, enterprise architectures often contain a long-term vision of how the organization plans to evolve its technology and environments. For example, the goal of phasing out an outdated technology platform may be established in this specification.

Finally, this document also may define the technology and policies behind enterprise-wide security measures. However, these often are isolated into a separate security architecture specification.
Service-oriented architecture

Put simply, service-oriented architecture spans both enterprise and application architecture domains. The benefit potential offered by SOA can only be truly realized when applied across multiple solution environments. This is where the investment in building reusable and interoperable services based on a vendor-neutral communications platform can fully be leveraged. This does not mean that the entire enterprise must become service-oriented. SOA belongs in those areas that have the most to gain from the features and characteristics it introduces.

Note that the term "SOA" does not necessarily imply a particular architectural scope. An SOA can refer to an application architecture or the approach used to standardize technical architecture across the enterprise. Because of the composable nature of SOA (meaning that individual application-level architectures can be comprised of different extensions and technologies), it is absolutely possible for an organization to have more than one SOA.

Note that, as explained in the previous chapter, the Web services platform offers one of a number of available forms of implementation for SOA. It is the approach exclusively explored by this book, but other approaches, such as those provided by traditional distributed platforms, also exist. An important aspect of the terminology used in the upcoming sections and throughout this book is that our use of the term "SOA" implies the contemporary SOA model (based on Web services and service-orientation principles) established in Chapter 3.

3.2. Common characteristics of contemporary SOA

Numerous recent and ongoing industry trends and developments have shaped the real world look of SOA. Its founding principles remain, but many have been expanded primarily because the opportunity to do so has been readily acted upon.

Major software vendors are continually conceiving new Web services specifications and building increasingly powerful XML and Web services support into current technology platforms. The result is an extended variation of service-oriented architecture we refer to as contemporary SOA.

Contemporary SOA builds upon the primitive SOA model by leveraging industry and technology advancements to further its original ideals. Though the required implementation technology can vary, contemporary SOAs have evolved to a point where they can be associated with a set of common characteristics.

Specifically, we explore the following primary characteristics:

- Contemporary SOA is at the core of the service-oriented computing platform.
- Contemporary SOA increases quality of service.
- Contemporary SOA is fundamentally autonomous.
- Contemporary SOA is based on open standards.
- Contemporary SOA supports vendor diversity.
- Contemporary SOA fosters intrinsic interoperability.
- Contemporary SOA promotes discovery.
- Contemporary SOA promotes federation.
- Contemporary SOA promotes architectural composability.
- Contemporary SOA fosters inherent reusability.
- Contemporary SOA emphasizes extensibility.
Contemporary SOA supports a service-oriented business modeling paradigm.
Contemporary SOA implements layers of abstraction.
Contemporary SOA promotes loose coupling throughout the enterprise.
Contemporary SOA promotes organizational agility.
Contemporary SOA is a building block.
Contemporary SOA is an evolution.
Contemporary SOA is still maturing.
Contemporary SOA is an achievable ideal.

Note the absence of traditional architectural qualities such as "secure," "transactional," "reliable," and so on. These have been grouped into the "Contemporary SOA increases quality of service" characteristic. Chapters 6 and 7 explain how the evolving landscape of Web services specifications addresses typical quality of service (QoS) requirements.

As we step through the following sections we elaborate on each of the characteristics in our list and discuss their overall meaning to SOA. In doing so, we also build a formal definition of contemporary SOA.

3.2.1. Contemporary SOA is at the core of the service-oriented computing platform

Before we get into the actual meaning behind contemporary SOA, let's first discuss how the term "SOA" has been tossed about within the IT industry. Many argue that the manner in which SOA is used to qualify products, designs, and technologies elevates this term beyond one that simply relates to architecture. SOA, some believe, has become synonymous with an entire new world application computing platform.

Past terms used to identify distinct application computing platforms were often suffixed with the word "architecture" when the architecture was actually being referenced. The terms "client-server" or "n-tier," for example, can be used to classify a tool, an administration infrastructure, or an application architecture.

With SOA, however, the actual acronym has become a multi-purpose buzzword used frequently when discussing an application computing platform consisting of Web services technology and service-orientation principles. Because the acronym already represents the word "architecture" we are unfortunately subjected to statements that can be confusing.

Perhaps the best way to view it is that if a product, design, or technology is prefixed with "SOA," it is something that was (directly or indirectly) created in support of an architecture based on service-orientation principles. Along those same lines, this book, though partially titled "Service-Oriented Architecture," goes well beyond architectural boundaries to explore the contemporary service-oriented platform.

Because we positioned contemporary SOA as building upon and extending the primitive SOA model, we already have a starting point for our definition:

Contemporary SOA represents an architecture that promotes service-orientation through the use of Web services.

3.2.2. Contemporary SOA increases quality of service

There is a definite need to bring SOA to a point where it can implement enterprise-level functionality as safely and reliably as the more established distributed architectures already do.
This relates to common quality of service requirements, such as:

- The ability for tasks to be carried out in a secure manner, protecting the contents of a message, as well as access to individual services.
- Allowing tasks to be carried out reliably so that message delivery or notification of failed delivery can be guaranteed.
- Performance requirements to ensure that the overhead imposed by SOAP message and XML content processing does not inhibit the execution of a task.
- Transactional capabilities to protect the integrity of specific business tasks with a guarantee that should the task fail, exception logic is executed.

Contemporary SOA is striving to fill the QoS gaps of the primitive SOA model. Many of the concepts and specifications we discuss in Part II SOA and WS-* Extensions provide features that directly address quality of service requirements. For lack of a better term, we'll refer to an SOA that fulfills specific quality of service requirements as "QoS-capable."

3.2.3. Contemporary SOA is fundamentally autonomous

The service-orientation principle of autonomy requires that individual services be as independent and self-contained as possible with respect to the control they maintain over their underlying logic. This is further realized through message-level autonomy where messages passed between services are sufficiently intelligence-heavy that they can control the manner in which they are processed by recipient services.

SOA builds upon and expands this principle by promoting the concept of autonomy throughout solution environments and the enterprise. Applications comprised of autonomous services, for example, can themselves be viewed as composite, self-reliant services that exercise their own self-governance within service-oriented integration environments.

Later we explain how by creating service abstraction layers, entire domains of solution logic can achieve control over their respective areas of governance. This establishes a level of autonomy that can cross solution boundaries.

3.2.4. Contemporary SOA is based on open standards

Perhaps the most significant characteristic of Web services is the fact that data exchange is governed by open standards. After a message is sent from one Web service to another it travels via a set of protocols that is globally standardized and accepted.

Further, the message itself is standardized, both in format and in how it represents its payload. The use of SOAP, WSDL, XML, and XML Schema allow for messages to be fully self-contained and support the underlying agreement that to communicate, services require nothing more than a knowledge of each other's service descriptions. The use of an open, standardized messaging model eliminates the need for underlying service logic to share type systems and supports the loosely coupled paradigm.

Contemporary SOAs fully leverage and reinforce this open, vendor-neutral communications framework (Figure 3.5). An SOA limits the role of proprietary technology to the implementation and hosting of the application logic encapsulated by a service. The opportunity for inter-service communication is therefore always an option.
3.2.5. Contemporary SOA supports vendor diversity

The open communications framework explained in the previous section not only has significant implications for bridging much of the heterogeneity within (and between) corporations, but it also allows organizations to choose best-of-breed environments for specific applications.

For example, regardless of how proprietary a development environment is, as long as it supports the creation of standard Web services, it can be used to create a non-proprietary service interface layer, opening up interoperability opportunities with other, service-capable applications (Figure 3.6). This, incidentally, has changed the face of integration architectures, which now can encapsulate legacy logic through service adapters, and leverage middleware advancements based on Web services.

3.2.6. Contemporary SOA promotes discovery

Even though the first generation of Web services standards included UDDI, few of the early implementations actually used service registries as part of their environments. This may have to do with the fact that not enough Web services were actually built to warrant a registry. However, another likely reason is that the concept of service discovery was simply not designed into the architecture. When utilized within traditional distributed architectures, Web services were more often employed to facilitate point-to-point solutions. Therefore, discovery was not a common concern.

SOA supports and encourages the advertisement and discovery of services throughout the enterprise and beyond. A serious
SOA will likely rely on some form of service registry or directory to manage service descriptions (**Figure 3.7**).

**Figure 3.7.** Registries enable a mechanism for the discovery of services.

3.2.7. Contemporary SOA fosters intrinsic interoperability

Further leveraging and supporting the required usage of open standards, a vendor diverse environment, and the availability of a discovery mechanism, is the concept of intrinsic interoperability. Regardless of whether an application actually has immediate integration requirements, design principles can be applied to outfit services with characteristics that naturally promote interoperability.

When building an SOA application from the ground up, services with intrinsic interoperability become potential integration endpoints (**Figure 3.8**). When properly standardized, this leads to service-oriented integration architectures wherein solutions themselves achieve a level of intrinsic interoperability. Fostering this characteristic can significantly alleviate the cost and effort of fulfilling future cross-application integration requirements.

**Figure 3.8.** Intrinsically interoperable services enable unforeseen integration opportunities.
3.2.8. Contemporary SOA promotes federation

Establishing SOA within an enterprise does not necessarily require that you replace what you already have. One of the most attractive aspects of this architecture is its ability to introduce unity across previously non-federated environments. While Web services enable federation, SOA promotes this cause by establishing and standardizing the ability to encapsulate legacy and non-legacy application logic and by exposing it via a common, open, and standardized communications framework (also supported by an extensive adapter technology marketplace).

Obviously, the incorporation of SOA with previous platforms can lead to a variety of hybrid solutions. However, the key aspect is that the communication channels achieved by this form of service-oriented integration are all uniform and standardized (Figure 3.9).

![Figure 3.9. Services enable standardized federation of disparate legacy systems.](image)

3.2.9. Contemporary SOA promotes architectural composability

Composability is a deep-rooted characteristic of SOA that can be realized on different levels. For example, by fostering the development and evolution of composable services, SOA supports the automation of flexible and highly adaptive business processes. As previously mentioned, services exist as independent units of logic. A business process can therefore be broken down into a series of services, each responsible for executing a portion of the process.

A broader example of composability is represented by the second-generation Web services framework that is evolving out of the release of the numerous WS-* specifications. The modular nature of these specifications allows an SOA to be composed of only the functional building blocks it requires.

What provides this flexibility is the fact that second-generation Web services specifications are being designed specifically to leverage the SOAP messaging model. Individual specifications consist of modular extensions that provide one or more specific features. As the offering of WS-* extensions supported by a given vendor platform grows, the flexibility to compose allows you to continue building solutions that only implement the features you actually need (Figure 3.10). In other words, the WS-* platform allows for the creation of streamlined and optimized service-oriented architectures, applications, services, and even messages.
3.2.10. Contemporary SOA fosters inherent reusability

SOA establishes an environment that promotes reuse on many levels. For example, services designed according to service-orientation principles are encouraged to promote reuse, even if no immediate reuse requirements exist. Collections of services that form service compositions can themselves be reused by larger compositions.

The emphasis placed by SOA on the creation of services that are agnostic to both the business processes and the automation solutions that utilize them leads to an environment in which reuse is naturally realized as a side benefit to delivering services for a given project. Thus, inherent reuse can be fostered when building service-oriented solutions (Figure 3.11).
3.2.11. Contemporary SOA emphasizes extensibility

When expressing encapsulated functionality through a service description, SOA encourages you to think beyond immediate, point-to-point communication requirements. When service logic is properly partitioned via an appropriate level of interface granularity, the scope of functionality offered by a service can sometimes be extended without breaking the established interface (Figure 3.12).

Figure 3.12. Extensible services can expand functionality with minimal impact.

Extensibility is also a characteristic that is promoted throughout SOA as a whole. Extending entire solutions can be accomplished by adding services or by merging with other service-oriented applications (which also, effectively, "adds services"). Because the loosely coupled relationship fostered among all services minimizes inter-service dependencies, extending logic can be achieved with significantly less impact.

Time to revisit our original definition to add a few adjectives that represent the characteristics we've covered.

Contemporary SOA represents an open, extensible, federated, composable architecture that promotes service-orientation and is comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services.

3.2.12. Contemporary SOA supports a service-oriented business modeling paradigm
In our description of a primitive SOA, we briefly explored how business processes can be represented and expressed through services. Partitioning business logic into services that can then be composed has significant implications as to how business processes can be modeled (Figure 3.13). Analysts can leverage these features by incorporating an extent of service-orientation into business processes for implementation through SOAs.

Figure 3.13. A collection (layer) of services encapsulating business process logic.

In other words, services can be designed to express business logic. BPM models, entity models, and other forms of business intelligence can be accurately represented through the coordinated composition of business-centric services. This is an area of SOA that is not yet widely accepted or understood. We therefore spend a significant portion of this book exploring the service-oriented business modeling paradigm.

3.2.13. Contemporary SOA implements layers of abstraction

One of the characteristics that tends to evolve naturally through the application of service-oriented design principles is that of abstraction. Typical SOAs can introduce layers of abstraction by positioning services as the sole access points to a variety of resources and processing logic.

When applied through proper design, abstraction can be targeted at business and application logic. For example, by establishing a layer of endpoints that represent entire solutions and technology platforms, all of the proprietary details associated with these environments disappear (Figure 3.14). The only remaining concern is the functionality offered via the service interfaces.

Figure 3.14. Application logic created with proprietary technology can be abstracted through a dedicated service layer.
It is the mutual abstraction of business and technology that supports the service-oriented business modeling paradigm we discussed and further establishes the loosely coupled enterprise model explained in the following section.

3.2.14. Contemporary SOA promotes loose coupling throughout the enterprise

As we've established, a core benefit to building a technical architecture with loosely coupled services is the resulting independence of service logic. Services only require an awareness of each other, allowing them to evolve independently.

Now, let's take a step back and look at the enterprise as a whole. Within an organization where service-orientation principles are applied to both business modeling and technical design, the concept of loose coupling is amplified.

By implementing standardized service abstraction layers, a loosely coupled relationship also can be achieved between the business and application technology domains of an enterprise (Figure 3.15). Each end only requires an awareness of the other, therefore allowing each domain to evolve more independently. The result is an environment that can better accommodate business and technology-related change—a quality known as organizational agility.

Figure 3.15. Through the implementation of service layers that abstract business and application logic, the loose coupling paradigm can be applied to the enterprise as a whole.
3.2.15. Contemporary SOA promotes organizational agility

Whether the result of an internal reorganization, a corporate merger, a change in an organization's business scope, or the replacement of an established technology platform, an organization's ability to accommodate change determines the efficiency with which it can respond to unplanned events.

Change in an organization's business logic can impact the application technology that automates it. Change in an organization's application technology infrastructure can impact the business logic automated by this technology. The more dependencies that exist between these two parts of an enterprise, the greater the extent to which change imposes disruption and expense.

By leveraging service business representation, service abstraction, and the loose coupling between business and application logic provided through the use of service layers, SOA offers the potential to increase organizational agility (Figure 3.16).

Figure 3.16. A loosely coupled relationship between business and application technology allows each end to more efficiently respond to changes in the other.
Other benefits realized through the standardization of SOA also contribute to minimizing dependencies and increasing overall responsiveness to change: notably, the intrinsic interoperability that can be built into services and the open communications framework established across integration architectures that enable interoperability between disparate platforms. Change imposed on any of these environments is more easily facilitated for the same reasons a loosely coupled state between services representing either ends of the communication channel.

Organizational agility is perhaps the most significant benefit that can be realized with contemporary SOA.

### 3.2.16. Contemporary SOA is a building block

A service-oriented application architecture will likely be one of several within an organization committed to SOA as the standard architectural platform. Organizations standardizing on SOA work toward an ideal known as the service-oriented enterprise (SOE), where all business processes are composed of and exist as services, both logically and physically.

When viewed in the context of SOE, the functional boundary of an SOA represents a part of this future-state environment, either as a standalone unit of business automation or as a service encapsulating some or all of the business automation logic. In responding to business model-level changes, SOAs can be augmented to change the nature of their automation, or they can be pulled into service-oriented integration architectures that require the participation of multiple applications.
What this all boils down to is that an individual service-oriented application can, in its entirety, be represented by and modeled as a single service. As mentioned earlier, there are no limits to the scope of service encapsulation. An SOA consists of services within services within services, to the point that a solution based on SOA itself is one of many services within an SOE.

This past set of characteristics has further broadened our definition. Let's append the definition with the following:

SOA can establish an abstraction of business logic and technology that may introduce changes to business process modeling and technical architecture, resulting in a loose coupling between these models. These changes foster service-orientation in support of a service-oriented enterprise.

### 3.2.17. Contemporary SOA is an evolution

SOA defines an architecture that is related to but still distinct from its predecessors. It differs from traditional client-server and distributed environments in that it is heavily influenced by the concepts and principles associated with service-orientation and Web services. It is similar to previous platforms in that it preserves the successful characteristics of its predecessors and builds upon them with distinct design patterns and a new technology set.

For example, SOA supports and promotes reuse, as well as the componentization and distribution of application logic. These and other established design principles that are commonplace in traditional distributed environments are still very much a part of SOA.

### 3.2.18. Contemporary SOA is still maturing

While the characteristics described so far are fundamental to contemporary SOA, this point is obviously more of a subjective statement of where SOA is at the moment. Even though SOA is being positioned as the next standard application computing platform, this transition is not yet complete. Despite the fact that Web services are being used to implement a great deal of application functionality, the support for a number of features necessary for enterprise-level computing is not yet fully available.

Standards organizations and major software vendors have produced many specifications to address a variety of supplementary extensions. Additionally, the next generation of development tools and application servers promises to support a great deal of these new technologies. When SOA platforms and tools reach an adequate level of maturity, the utilization of Web services can be extended to support the creation of enterprise SOA solutions, making the ideal of a service-oriented enterprise attainable.

If you needed to provide an accurate definition of SOA today, you would not be out of line to mention the state of its underlying technology. Considering the rate at which the IT industry as a whole is adopting and evolving the SOA platform, though, it should not be too long before an accurate definition will no longer require this statement.

### 3.2.19. Contemporary SOA is an achievable ideal

A standardized enterprise-wide adoption of SOA is a state to which many organizations would like to fast-forward. The reality is that the process of transitioning to this state demands an enormous amount of effort, discipline, and, depending on the size of the organization, a good amount of time. Every technical environment will undergo changes during such a migration, and various parts of SOA will be phased in at different stages and to varying extents. This will likely result in
countless hybrid architectures, consisting mostly of distributed environments that are part legacy and part service-oriented. Further supporting this prediction is the evolving state of the technology set that is emerging to realize enterprise-level SOAs. As companies adopt SOA during this evolution, many will need to retrofit their environments (and their standards) to accommodate changes and innovations as SOA-related specifications, standards, and products continue to mature.

However, the majority of the contemporary SOA characteristics we just covered are attainable today. This book provides a series of tutorials and step-by-step process descriptions that explain how to manifest them.

3.2.20. Defining SOA

Now that we've finished covering characteristics, we can finalize our formal definition.

Contemporary SOA represents an open, agile, extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services.

SOA can establish an abstraction of business logic and technology that may introduce changes to business process modeling and technical architecture, resulting in a loose coupling between these models.

SOA is an evolution of past platforms, preserving successful characteristics of traditional architectures, and bringing with it distinct principles that foster service-orientation in support of a service-oriented enterprise.

SOA is ideally standardized throughout an enterprise, but achieving this state requires a planned transition and the support of a still evolving technology set.

Though accurate, this definition of contemporary SOA is quite detailed. For practical purposes, let's provide a supplementary definition that can be applied to both primitive and contemporary SOA.

SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and automation domains of an enterprise.

3.2.21. Separating concrete characteristics

Looking back at the list of characteristics we just covered, we can actually split them into two groupscharacteristics that represent concrete qualities that can be realized as real extensions of SOA and those that can be categorized as commentary or observations. Collectively, these characteristics were useful for achieving our formal definition. From here on, though, we are more interested in exploring the concrete characteristics only.

Let's therefore remove the following items from our original list:

- Contemporary SOA is at the core of the service-oriented computing platform.
- Contemporary SOA is a building block.
- Contemporary SOA is an evolution.
- Contemporary SOA is still maturing.
• Contemporary SOA is an achievable ideal.

By trimming these items, along with some superfluous wording, we end up with the following set of concrete characteristics.

Contemporary SOA is generally:

• based on open standards
• architecturally composable
• capable of improving QoS

Contemporary SOA supports, fosters, or promotes:

• vendor diversity
• intrinsic interoperability
• discoverability
• federation
• inherent reusability
• extensibility
• service-oriented business modeling
• layers of abstraction
• enterprise-wide loose coupling
• organizational agility

It is these characteristics that, when realized, provide tangible, measurable benefits.

Note

Though we qualify these as "concrete" here, it is this set of characteristics that we refer to when we use the term "contemporary SOA characteristics" from here on.

SUMMARY OF KEY POINTS

• We distinguish contemporary SOA with a number of common characteristics that build upon and extend the original qualities and principles established by primitive SOA.
• The realization of contemporary SOA characteristics is explored in detail throughout this book.

4.3.2. SOA vs. client-server architecture

Just about any environment in which one piece of software requests or receives information from another can be referred to as "client-server." Pretty much every variation of application architecture that ever existed (including SOA) has an element of client-server interaction in it. However, the industry term "client-server architecture" generally refers to a particular generation of early environments during which the client and the server played specific roles and had distinct implementation characteristics.
Client-server architecture: a brief history

The original monolithic mainframe systems that empowered organizations to get seriously computerized often are considered the first inception of client-server architecture. These environments, in which bulky mainframe back-ends served thin clients, are considered an implementation of the single-tier client-server architecture (Figure 4.2).

Figure 4.2. A typical single-tier client-server architecture.

Mainframe systems natively supported both synchronous and asynchronous communication. The latter approach was used primarily to allow the server to continuously receive characters from the terminal in response to individual key-strokes. Only upon certain conditions would the server actually respond.

While its legacy still remains, the reign of the mainframe as the foremost computing platform began to decline when a two-tier variation of the client-server design emerged in the late 80s.

This new approach introduced the concept of delegating logic and processing duties onto individual workstations, resulting in the birth of the fat client. Further supported by the innovation of the graphical user-interface (GUI), two-tier client-server was considered a huge step forward and went on to dominate the IT world for years during the early 90s.

The common configuration of this architecture consisted of multiple fat clients, each with its own connection to a database on a central server. Client-side software performed the bulk of the processing, including all presentation-related and most data access logic (Figure 4.3). One or more servers facilitated these clients by hosting scalable RDBMSs.

Figure 4.3. A typical two-tier client-server architecture.
Let's look at the primary characteristics of the two-tier client-server architecture individually and compare them to the corresponding parts of SOA.

Application logic

Client-server environments place the majority of application logic into the client software. This results in a monolithic executable that controls the user experience, as well as the back-end resources. One exception is the distribution of business rules. A popular trend was to embed and maintain business rules relating to data within stored procedures and triggers on the database. This somewhat abstracted a set of business logic from the client and simplified data access programming. Overall, though, the client ran the show.

The presentation layer within contemporary service-oriented solutions can vary. Any piece of software capable of
exchanging SOAP messages according to required service contracts can be classified as a service requestor. While it is commonly expected for requestors to be services as well, presentation layer designs are completely open and specific to a solution's requirements.

Within the server environment, options exist as to where application logic can reside and how it can be distributed. These options do not preclude the use of database triggers or stored procedures. However, service-oriented design principles come into play, often dictating the partitioning of processing logic into autonomous units. This facilitates specific design qualities, such as service statelessness and interoperability, as well as future composability and reusability.

Additionally, it is more common within an SOA for these units of processing logic to be solution-agnostic. This supports the ultimate goal of promoting reuse and loose coupling across application boundaries.

**Application processing**

Because most client-server application logic resides in the client component, the client workstation is responsible for the bulk of the processing. The 80/20 ratio often is used as a rule of thumb, with the database server typically performing twenty percent of the work. Despite that, though, it is the database that frequently becomes the performance bottleneck in these environments.

A two-tier client-server solution with a large user-base generally requires that each client establish its own database connection. Communication is predictably synchronous, and these connections are often persistent (meaning that they are generated upon user login and kept active until the user exits the application). Proprietary database connections are expensive, and the resource demands sometimes overwhelm database servers, imposing processing latency on all users. Additionally, given that the clients are assigned the majority of processing responsibilities, they too often demand significant resources. Client-side executables are fully stateful and consume a steady chunk of PC memory. User workstations therefore often are required to run client programs exclusively so that all available resources can be offered to the application.

Processing in SOA is highly distributed. Each service has an explicit functional boundary and related resource requirements. In modeling a technical service-oriented architecture, you have many choices as to how you can position and deploy services. Enterprise solutions consist of multiple servers, each hosting sets of Web services and supporting middleware. There is, therefore, no fixed processing ratio for SOAs. Services can be distributed as required, and performance demands are one of several factors in determining the physical deployment configuration.

Communication between service and requestor can be synchronous or asynchronous. This flexibility allows processing to be further streamlined, especially when asynchronous message patterns are utilized. Additionally, by placing a large amount of intelligence into the messages, options for achieving message-level context management are provided. This promotes the stateless and autonomous nature of services and further alleviates processing by reducing the need for runtime caching of state information.

**Technology**

The emergence of client-server applications promoted the use of 4GL programming languages, such as Visual Basic and PowerBuilder. These development environments took better advantage of the Windows operating system by providing the ability to create aesthetically rich and more interactive user-interfaces. Regardless, traditional 3GL languages, such as C++, were also still used, especially for solutions that had more rigid performance requirements. On the back-end, major database
vendors, such as Oracle, Informix, IBM, Sybase, and Microsoft, provided robust RDBMSs that could manage multiple connections, while providing flexible data storage and data management features.

The technology set used by SOA actually has not changed as much as it has expanded. Newer versions of older programming languages, such as Visual Basic, still can be used to create Web services, and the use of relational databases still is commonplace. The technology landscape of SOA, though, has become increasingly diverse. In addition to the standard set of Web technologies (HTML, CSS, HTTP, etc.) contemporary SOA brings with it the absolute requirement that an XML data representation architecture be established, along with a SOAP messaging framework, and a service architecture comprised of the ever-expanding Web services platform.

Security

Besides the storage and management of data and the business rules embedded in stored procedures and triggers, the one other part of client-server architecture that frequently is centralized at the server level is security. Databases are sufficiently sophisticated to manage user accounts and groups and to assign these to individual parts of the physical data model.

Security also can be controlled within the client executable, especially when it relates to specific business rules that dictate the execution of application logic (such as limiting access to a part of a user-interface to select users). Additionally, operating system-level security can be incorporated to achieve a single sign-on, where application clearance is derived from the user's operating system login account information.

Though one could boast about the advantages of SOA, most architects envy the simplicity of client-server security. Corporate data is protected via a single point of authentication, establishing a single connection between client and server. In the distributed world of SOA, this is not possible. Security becomes a significant complexity directly relational to the degree of security measures required. Multiple technologies are typically involved, many of which comprise the WS-Security framework (explained in Chapters 7 and 17).

Administration

One of the main reasons the client-server era ended was the increasingly large maintenance costs associated with the distribution and maintenance of application logic across user workstations. Because each client housed the application code, each update to the application required a redistribution of the client software to all workstations. In larger environments, this resulted in a highly burdensome administration process.

Maintenance issues spanned both client and server ends. Client workstations were subject to environment-specific problems because different workstations could have different software programs installed or may have been purchased from different hardware vendors. Further, there were increased server-side demands on databases, especially when a client-server application expanded to a larger user base.

Because service-oriented solutions can have a variety of requestors, they are not necessarily immune to client-side maintenance challenges. While their distributed back-end does accommodate scalability for application and database servers, new administration demands can be introduced. For example, once SOAs evolve to a state where services are reused and become part of multiple service compositions, the management of server resources and service interfaces can require powerful administration tools, including the use of a private registry.
Case Study

RailCo's accounting system is a classic two-tier client-server application. Its GUI front-end consists of a single executable designed for deployment on old Windows workstations. It provides user-interfaces for looking up, editing, and adding various accounting records. It also offers a financial reporting facility that can produce a fixed amount of statements with detailed or summarized accounting data.

Considering it's only ever had two to three users, there have never really been performance problems on the database end. The now outdated RDBMS that has been in place for the past decade has been reliable and has required little attention.

However, problems with this application have surfaced:

- Operating system upgrades have introduced erratic behavior on some screens, resulting in unexplainable error messages. It is uncertain if these are caused by other programs that have been installed on the workstations.
- The workstations themselves have been rarely upgraded and have not kept pace with the hardware demands of recent software upgrades. After the accounting system launches, there is little more the user can do with the computer. As a result, employee productivity has been affected somewhat.
- Following a new records management policy and some billing procedure changes, a modification to the overall billing process was imposed on the accounting personnel. Because the accounting system was not designed to accommodate this change, employees are required to supplement the automated billing process by manually filling out supplementary forms.

Fundamentally, this accounting system has been getting the job done. However, the actual accounting tasks performed by the users have become increasingly convoluted and inefficient. This is due to the questionable stability of the workstation environments and also because the system itself is not easily adaptable to changes in the processes it automates.

SOA can address issues such as these, as follows:

- Service-oriented solutions eliminate dependencies on user workstation environments by delegating all processing to the server-side (as regular distributed Internet applications already have been doing).
- SOA establishes an adaptable and extensible architectural model that allows solutions to be enhanced with minimal impact. Services can encapsulate existing legacy logic providing a standardized API that can plug into larger integrated solutions. Further, when building custom service-oriented applications, extensibility can be built into the solution environment, supporting future enhancements, also with minimal impact.

4.3.3. SOA vs. distributed Internet architecture

This comparison may seem like a contradiction, given that SOA can be viewed as a form of distributed Internet architecture and because we established earlier that previous types of distributed architecture also could be designed as SOAs. Though
possible, and although there are distributed environments in existence that may have been heavily influenced by service-oriented principles, this variation of SOA is still a rarity. Consider the comparison provided here as one that contrasts traditional distributed Internet architecture in the manner it was most commonly designed.

**Distributed Internet architecture: a brief history**

In response to the costs and limitations associated with the two-tier client server architecture, the concept of building component-based applications hit the mainstream. Multi-tier client-server architectures surfaced, breaking up the monolithic client executable into components designed to varying extents of compliance with object-orientation.

Distributing application logic among multiple components (some residing on the client, others on the server) reduced deployment headaches by centralizing a greater amount of the logic on servers. Server-side components, now located on dedicated application servers, would then share and manage pools of database connections, alleviating the burden of concurrent usage on the database server (Figure 4.4). A single connection could easily facilitate multiple user.

**Figure 4.4. A typical multi-tier client-server architecture**

These benefits came at the cost of increased complexity and ended up shifting expense and effort from deployment issues to development and administration processes. Building components capable of processing multiple, concurrent requests was more difficult and problem-ridden than developing a straight-forward executable intended for a single user.

Additionally, replacing client-server database connections was the client-server remote procedure call (RPC) connection. RPC technologies such as CORBA and DCOM allowed for remote communication between components residing on client
workstations and servers. Issues similar to the client-server architecture problems involving resources and persistent connections emerged. Adding to this was an increased maintenance effort resulting from the introduction of the middleware layer. For example, application servers and transaction monitors required significant attention in larger environments.

Upon the arrival of the World Wide Web as a viable medium for computing technology in the mid-to-late 90s, the multi-tiered client-server environments began incorporating Internet technology. Most significant was the replacement of the custom software client component with the browser. Not only did this change radically alter (and limit) user-interface design, it practically shifted 100% of application logic to the server (Figure 4.5).

**Figure 4.5. A typical distributed Internet architecture.**

Distributed Internet architecture also introduced a new physical tier, the Web server. This resulted in HTTP replacing proprietary RPC protocols used to communicate between the user's workstation and the server. The role of RPC was limited to enabling communication between remote Web and application servers.

From the late 90s to the mid 2000s, distributed Internet architectures represented the de facto computing platform for custom developed enterprise solutions. The commoditization of component-based programming skills and the increasing sophistication of middleware eventually lessened some of the overall complexity.

How then, does this popular and familiar architecture compare with SOA? The following sections contrast distributed Internet architecture and SOA characteristics.

Note
Although multi-tier client-server is a distinct architecture in its own right, we do not provide a direct comparison between it and SOA. Most of the issues raised in the client-server and the distributed Internet architecture comparisons cover those that would be discussed in a comparison between multi-tier client-server and SOA.

**Application logic**

Except for some rare applications that embed proprietary extensions in browsers, distributed Internet applications place all of their application logic on the server side. Even client-side scripts intended to execute in response to events on a Web page are downloaded from the Web server upon the initial HTTP request. With none of the logic existing on the client workstation, the entire solution is centralized.

The emphasis is therefore on:

- how application logic should be partitioned
- where the partitioned units of processing logic should reside
- how the units of processing logic should interact

From a physical perspective, service-oriented architecture is very similar to distributed Internet architecture. Provider logic resides on the server end where it is broken down into separate units. The differences lie in the principles used to determine the three primary design considerations just listed.

Traditional distributed applications consist of a series of components that reside on one or more application servers. Components are designed with varying degrees of functional granularity, depending on the tasks they execute, and to what extent they are considered reusable by other tasks or applications. Components residing on the same server communicate via proprietary APIs, as per the public interfaces they expose. RPC protocols are used to accomplish the same communication across server boundaries. This is made possible through the use of local proxy stubs that represent components in remote locations (Figure 4.6).

![Figure 4.6. Components rely on proxy stubs for remote communication.](image)

At design time, the expected interaction components will have with others is taken into account so much so that actual references to other physical components can be embedded within the programming code. This level of design-time dependence is a form of tight-coupling. It is efficient in that little processing is wasted in trying to locate a required component at runtime. However, the embedded coupling leads to a tightly bound component network that, once implemented, is not easily altered.
Contemporary SOAs still employ and rely on components. However, the entire modeling approach now takes into consideration the creation of services that encapsulate some or all of these components. These services are designed according to service-orientation principles and are strategically positioned to expose specific sets of functionality. While this functionality can be provided by components, it also can originate from legacy systems and other sources, such as adapters interfacing with packaged software products, or even databases.

The purpose of wrapping functionality within a service is to expose that functionality via an open, standardized interface irrespective of the technology used to implement the underlying logic. The standardized interface supports the open communications framework that sits at the core of SOA. Further, the use of Web services establishes a loosely coupled environment that runs contrary to many traditional distributed application designs. When properly designed, loosely coupled services support a composition model, allowing individual services to participate in aggregate assemblies. This introduces continual opportunities for reuse and extensibility.

Another significant shift related to the design and behavior of distributed application logic is in how services exchange information. While traditional components provide methods that, once invoked, send and receive parameter data, Web services communicate with SOAP messages. Even though SOAP supports RPC-style message structures, the majority of service-oriented Web service designs rely on document-style messages. (This important distinction is explored in subsequent chapters.)

Also messages are structured to be as self-sufficient as possible. Through the use of SOAP headers, message contents can be accompanied by a wide range of meta information, processing instructions, and policy rules. In comparison to data exchange in the pure component world, the messaging framework used by SOA is more sophisticated, bulkier, and tends to result in less individual transmissions.

Finally, although reuse is also commonly emphasized in traditional distributed design approaches, SOA fosters reuse and cross-application interoperability on a deep level by promoting the creation of solution-agnostic services.

Application processing

Regardless of platform, components represent the lion's share of application logic and are therefore responsible for most of the processing. However, because the technology used for inter-component communication differs from the technology used to accomplish inter-service communication, so do the processing requirements.

Distributed Internet architecture promotes the use of proprietary communication protocols, such as DCOM and vendor implementations of CORBA for remote data exchange. While these technologies historically have had challenges, they are considered relatively efficient and reliable, especially once an active connection is made. They can support the creation of stateful and stateless components that primarily interact with synchronous data exchanges (asynchronous communication is supported by some platforms but not commonly used).

SOA, on the other hand, relies on message-based communication. This involves the serialization, transmission, and deserialization of SOAP messages containing XML document payloads. Processing steps can involve the conversion of relational data into an XML-compliant structure, the validation of the XML document prior and subsequent to transmission, and the parsing of the document and extraction of the data by the recipient. Although advancements, such as the use of enterprise parsers and hardware accelerators are on-going, most still rank RPC communication as being noticeably faster than SOAP.
Because a network of SOAP servers can effectively replace RPC-style communication channels within service-oriented application environments, the incurred processing overhead becomes a significant design issue. Document and message modeling conventions and the strategic placement of validation logic are important factors that shape the transport layer of service-oriented architecture.

This messaging framework promotes the creation of autonomous services that support a wide range of message exchange patterns. Though synchronous communication is fully supported, asynchronous patterns are encouraged, as they provide further opportunities to optimize processing by minimizing communication. Further supporting the statelessness of services are various context management options that can be employed, including the use of WS-* specifications, such as WS-Coordination and WS-BPEL, as well as custom solutions.

**Technology**

The technology behind distributed Internet architecture went through a number of stages over the past few years. Initial architectures consisted of components, server-side scripts, and raw Web technologies, such as HTML and HTTP. Improvements in middleware allowed for increased processing power and transaction control. The emergence of XML introduced sophisticated data representation that actually gave substance to content transmitted via Internet protocols. The subsequent availability of Web services allowed distributed Internet applications to cross proprietary platform boundaries.

Because many current distributed applications use XML and Web services, there may be little difference between the technology behind these solutions and those based on SOA. One clear distinction, though, is that a contemporary SOA will most likely be built upon XML data representation and the Web services technology platform. Beyond a core set of Internet technologies and the use of components, there is no governance of the technology used by traditional Internet applications. Thus XML and Web services are optional for distributed Internet architecture but not for contemporary SOA.

**Security**

When application logic is strewn across multiple physical boundaries, implementing fundamental security measures such as authentication and authorization becomes more difficult.

In a two-tiered client-server environment, an exclusive server-side connection easily facilitates the identification of users and the safe transportation of corporate data. However, when the exclusivity of that connection is removed, and when data is required to travel across different physical layers, new approaches to security are needed. To ensure the safe transportation of information and the recognition of user credentials, while preserving the original security context, traditional security architectures incorporate approaches such as delegation and impersonation. Encryption also is added to the otherwise wide open HTTP protocol to allow data to be protected during transmission beyond the Web server.

SOAs depart from this model by introducing wholesale changes to how security is incorporated and applied. Relying heavily on the extensions and concepts established by the WS-Security framework, the security models used within SOA emphasize the placement of security logic onto the messaging level. SOAP messages provide header blocks in which security logic can be stored. That way, wherever the message goes, so does its security information. This approach is required to preserve individual autonomy and loose coupling between services, as well as the extent to which a service can remain fully stateless.

**Administration**

Maintaining component-based applications involves keeping track of individual component instances, tracing local and remote communication problems, monitoring server resource demands, and, of course, the standard database administration
tasks. Distributed Internet architecture further introduces the Web server and with it an additional physical environment that requires attention while solutions are in operation. Because clients, whether local or external to an organization, connect to these solutions using HTTP, the Web server becomes the official first point of contact. It must therefore be designed for scalability requirement that has led to the creation of Web server farms that pool resources.

Enterprise-level SOAs typically require additional runtime administration. Problems with messaging frameworks (especially when working with asynchronous exchange patterns) can more easily go undetected than with RPC-based data exchanges. This is because so many variations exist as to how messages can be interchanged. RPC communication generally requires a response from the initiating component, indicating success or failure. Upon encountering a failure condition, an exception handling routine kicks in. Exception handling with messaging frameworks can be more complex and less robust. Although WS-* extensions are being positioned to better deal with these situations, administration effort is still expected to remain high.

Other maintenance tasks, such as resource management (similar to component management), are also required. However, to best foster reuse and composability, a useful part of an administration infrastructure for enterprises building large amounts of Web services is a private registry. UDDI is one of the technologies used for standardizing this interface repository, which can be manually or programmatically accessed to discover service descriptions.

**Case Study**

The TLS accounting system consists of a large, distributed component-based solution. Some 50 odd components host and execute various parts of the application logic. For performance and security reasons, some components have been deployed on separate application servers.

Overall, the execution of a typical accounting task will involve four to five physical layers consisting of:

- A Web server hosting server-side scripts that relay HTTP requests to components on application servers and then relay responses from those components back to the browser clients.
- An application server hosting a controller component that generates a transaction context and manages more specialized components.
- A possible second application server hosting two or more business components that enforce specific business rules and perform various functions related to a particular business context. This server also may host one or more data components that encapsulate the data access logic required to interact with application repositories.
- A database server hosting a complete RDBMS environment.

This enterprise solution has undergone many changes and enhancements over the past few years. Some of the primary issues that have arisen include:

- Initially, many components were custom developed to alter or extend existing functionality. Each redevelopment project has become increasingly expensive. This trend is being blamed on the overhead associated with the amount of testing and redeployment effort required to ensure that all pre-existing dependencies are not affected by any modification to a component's functionality.
- Because state management was never standardized, a design disparity has emerged. Some components manage state information by caching data in memory, while others use application server-deployed
databases. This became an issue when XML was first introduced as a standard data format. Permanent state management designs already had a relational storage format in place that was incompatible with the required XML document structures.

Subsequent chapters explain how SOA addresses these types of problems as follows:

- SOA establishes a loosely coupled relationship between units of processing logic encapsulated as services. This allows the logic within each service boundary to be updated and evolved independently of existing service requestors, as long as the original service contract is preserved.
- SOA promotes the standardization of XML data representation throughout solution environments. Further, service statelessness is emphasized by deferring state management to the message level. This maximizes reuse, availability, and scalability of service logic but also provides a standardized state management approach.

### 4.3.4. SOA vs. hybrid Web service architecture

In the previous section we mentioned how more recent variations of the distributed Internet architecture have come to incorporate Web services. This topic is worth elaborating upon because it has been (and is expected to continue to be) at the root of some confusion surrounding SOA.

First, the use of Web services within traditional architectures is completely legitimate. Due to the development support for Web services in many established programming languages, they easily can be positioned to fit in with older application designs. And, for those legacy environments that do not support the custom development of Web services, adapters are often available.

Note

Although we are focusing on distributed Internet architecture here, there are no restrictions for two-tier client-server applications to be outfitted with Web services.

**Web services as component wrappers**

The primary role of Web services in this context has been to introduce an integration layer that consists of wrapper services that enable synchronous communication via SOAP-compliant integration channels (**Figure 4.7**). In fact, the initial release of the SOAP specification and the first generation of SOAP servers were specifically designed to duplicate RPC-style communication using messages.

**Figure 4.7. Wrapper services encapsulating components.**
These integration channels are primarily utilized in integration architectures to facilitate communication with other applications or outside partners. They also are used to enable communication with other (more service-oriented) solutions and to take advantage of some of the features offered by third-party utility Web services. Regardless of their use or purpose within traditional architectures, it is important to clarify that a distributed Internet architecture that incorporates Web services in this manner does not qualify as a true SOA. It is simply a distributed Internet architecture that uses Web services.

Instead of mirroring component interfaces and establishing point-to-point connections with Web services, SOA provides strong support for a variety of messaging models (based on both synchronous and asynchronous exchanges). Additionally, Web services within SOAs are subject to specific design requirements, such as those provided by service-orientation principles. These and other characteristics support the pursuit of consistent loose coupling. Once achieved, a single service is never limited to point-to-point communication; it can accommodate any number of current and future requestors.

Web services within SOA

While SOAs can vary in size and quality, there are tangible characteristics that distinguish an SOA from other architectures that use Web services. Much of this book is dedicated to exploring these characteristics. For now it is sufficient to state that fundamentally, SOAs are built with a set of Web services designed to collectively automate (or participate in the automation of) one or more business processes and that SOA promotes the organization of these services into specialized layers that abstract specific parts of enterprise automation logic.

Also by standardizing on SOA across an enterprise, a natural interoperability emerges that transcends proprietary application platforms. This allows for previously disparate environments to be composed in support of new and evolving business automation processes.

Case Study

TLS had the development of a group of custom eBusiness solutions outsourced to a number of consulting firms. With each project, TLS was guaranteed that the latest technologies would be used. In particular, they
were assured that XML and Web services had been incorporated. These specialized applications were even referred to as "service-oriented."

Later, a requirement arose for one solution to integrate with another. A subsequent analysis revealed an alarming degree of inconsistency with regard to how each application managed and represented corporate data and the messaging formats used to package this data. To achieve the level of required interoperability between these two systems, a complex and expensive integration project was needed. Many stakeholders wondered why, if both systems were based on common technologies, sharing data between them was still such a monumental issue.

It turned out that each solution managed corporate data relevant to its application scope in a different way. Some used XML only to represent data in a unique context. Though promoted as service-oriented solutions, Web services were not actually a key part of the application architecture. These "token services" addressed some specific requirements but were not built with future interoperability in mind.

There was no initial concern around this approach, as each application delivered its promised set of features and solved its corresponding business problems. However, because no design principles were applied to ensure that XML and Web services were being implemented in a standardized manner in support of SOA, there was nothing in place to prevent the resulting design disparity.

4.3.5. Service-orientation and object-orientation (Part I)

Note that this section title is "Service-orientation and object-orientation," as opposed to "Service-orientation vs. object-orientation." That distinction was made to stress the fact that the relationship between these two schools of thought is not necessarily a competitive one.

In fact, object-oriented programming is commonly used to build the application logic encapsulated within Web services. However, how the object-oriented programming methodology differs fundamentally from service-orientation is worth exploring. An understanding of their differences will help you make them work together.

Below is a list comparing aspects of these design approaches. (Whereas service-orientation is based on the design of services, object-orientation is centered around the creation of objects. Because comparing services to objects can be confusing, the term "units of processing logic" is used.)

- Service-orientation emphasizes loose coupling between units of processing logic (services). Although object-orientation supports the creation of reusable, loosely coupled programming routines, much of it is based on predefined class dependencies, resulting in more tightly bound units of processing logic (objects).
- Service-orientation encourages coarse-grained interfaces (service descriptions) so that every unit of communication (message) contains as much information as possible for the completion of a given task. Object-oriented programming fully supports fine-grained interfaces (APIs) so that units of communication (RPC or local API calls) can perform various sized tasks.
- Service-orientation expects the scope of a unit of processing logic (service) to vary significantly. Object-oriented units of logic (objects) tend to be smaller and more specific in scope.
- Service-orientation promotes the creation of activity-agnostic units of processing logic (services) that are driven into action by intelligent units of communication (messages). Object-orientation encourages the binding of processing logic with data, resulting in highly intelligent units (objects).
Service-orientation prefers that units of processing logic (services) be designed to remain as stateless as possible. Object-orientation promotes the binding of data and logic, resulting in the creation of more stateful units (objects). (However, more recent component-based design approaches deviate from this tendency.)

Service-orientation supports the composition of loosely coupled units of processing logic (services). Object-orientation supports composition but also encourages inheritance among units of processing logic (objects), which can lead to tightly coupled dependencies.

You may have noticed that we avoided referencing specific object-orientation principles, such as encapsulation, inheritance, and aggregation. Because we have not yet fully described the principles of service-orientation, we cannot compare the respective paradigms on this level. Chapter 8 explains the individual service-orientation principles in detail and then continues this discussion in the Service-orientation and object-orientation (Part II) section.

**SUMMARY OF KEY POINTS**

- SOA is a radical departure from client-server architecture. Current SOAs still employ some of the technologies originally used to build client-server applications. Though more sophisticated, SOAs introduce complexity that sharply contrasts the simplicity of a two-tier client-server architecture.
- Distributed Internet architecture has much in common with SOA, including a significant amount of its technology. However, SOA has distinct characteristics relating to both technology and its underlying design principles. For example, SOA introduces processing and security requirements that differ from distributed Internet architecture, and SOA administration is typically more complex due to its reliance on messaging-based communication.
- Traditional architectures have and can continue to use Web services within their own design paradigms. It’s important to not confuse these architectures with SOA. Non-SOA use of Web services is typically found within distributed Internet architectures, where Web services are employed to mirror RPC-style communication.

### 8.2. Anatomy of a service-oriented architecture

Chapter 5 established the components of the basic (first-generation) Web services framework. This framework can be applied to implement services in just about any environment. For example, services can be appended to traditional distributed applications or used as wrappers to expose legacy system logic. However, neither of these environments resembles a "real" service-oriented architecture.

To best understand what constitutes a true SOA, we need to abstract the key components of the Web services framework and study their relationships more closely. To accomplish this, we begin by revisiting these familiar components and altering our perspective of them. First, we re-label them to reflect terminology more associated with service-orientation. Then we position them into a logical view wherein we subsequently re-examine our components within the context of SOA.

#### 8.2.1. Logical components of the Web services framework

The communications framework established by Web services brings us the foundation technology for what we've classified as contemporary SOA. Because we covered this framework in Chapter 5, we will use it as a reference point for our discussion of service-orientation.

Let's first recap some Web services fundamentals within a logical modeling context. As shown in Figure 8.4, each Web service contains one or more operations. Note that this diagram introduces a new symbol to represent operations separately from the service.
Each operation governs the processing of a specific function the Web service is capable of performing. The processing consists of sending and receiving SOAP messages, as shown in Figure 8.5.

By composing these parts, Web services form an activity through which they can collectively automate a task (Figure 8.6).
8.2.2. Logical components of automation logic

The Web services framework provides us not only with a technology base for enabling connectivity, it also establishes a modularized perspective of how automation logic, as a whole, can be comprised of independent units. To illustrate the inherent modularity of Web services, let's abstract the following fundamental parts of the framework:

- SOAP messages
- Web service operations
- Web services
- activities

The latter three items represent units of logic that perform work and communicate using SOAP messages. To better illustrate this in a service-oriented perspective, let's replace these terms with new ones, as follows:

- messages
- operations
- services
- processes (and process instances)

You'll notice that these are quite similar to the terms we used before. The one exception is the use of "process" instead of "activity." In later chapters we actually use the word "activity" in different contexts when modeling service-oriented business processes.

For now, the one discrepancy to be aware of is that while a Web service activity is typically used to represent the temporary interaction of a group of Web services, a process is a static definition of interaction logic. An activity is best compared to an instance of a process wherein a group of services follow a particular path through the process logic to complete a task.

Regardless, for the purposes of our discussion of service-orientation, we'll continue with our look at how automation logic is comprised of the four identified parts. We can further qualify these parts by relating each to different sized units of logic, as follows:

- messages = units of communication
- operations = units of work
- services = units of processing logic (collections of units of work)
- processes = units of automation logic (coordinated aggregation of units of work)

Figure 8.7 provides us with a primitive view of how operations and services represent units of logic that can be assembled to comprise a unit of automation logic.
Figure 8.7. A primitive view of how SOA modularizes automation logic into units.

Next, in Figure 8.8, we establish that messages are a suitable means by which all units of processing logic (services) communicate. This illustrates that regardless of the scope of logic a service represents, no actual processing of that logic can be performed without issuing units of communication (in this case, messages).

The purpose of these views is simply to express that processes, services, and operations, on the most fundamental level, provide a flexible means of partitioning and modularizing logic. Regardless of the technology platform used, this remains the most basic concept that underlies service-orientation. In being able to derive this view from the Web services framework, we also have demonstrated the suitability of the Web services platform as a means of implementation for SOA.

8.2.3. Components of an SOA

We'll continue to work with our components of automation logic, but we now broaden our discussion to how the characteristics and behaviors of these components are formed within service-oriented architecture.
Each of the previously defined components establishes a level of enterprise logic abstraction, as follows:

- A message represents the data required to complete some or all parts of a unit of work.
- An operation represents the logic required to process messages in order to complete a unit of work (Figure 8.9).

Figure 8.9. The scope of an operation within a process.

- A service represents a logically grouped set of operations capable of performing related units of work.
- A process contains the business rules that determine which service operations are used to complete a unit of automation. In other words, a process represents a large piece of work that requires the completion of smaller units of work (Figure 8.10).

Figure 8.10. Operations belonging to different services representing various parts of process logic.

8.2.4. How components in an SOA inter-relate

Having established the core characteristics of our SOA components, let's now look at how these components are required to relate to each other:

- An operation sends and receives messages to perform work.
- An operation is therefore mostly defined by the messages it processes.
- A service groups a collection of related operations.
- A service is therefore mostly defined by the operations that comprise it.
- A process instance can compose services.
- A process instance is not necessarily defined by its services because it may only require a subset of the functionality offered by the services.
- A process instance invokes a unique series of operations to complete its automation.
- Every process instance is therefore partially defined by the service operations it uses.

Figures 8.11 and 8.12 further illustrate these relationships.

**Figure 8.11. How the components of a service-oriented architecture relate.**

**Figure 8.12. How the components of a service-oriented architecture define each other.**
A service-oriented architecture is an environment standardized according to the principles of service-orientation in which a process that uses services (a service-oriented process) can execute. Next, we'll take a closer look at what exactly the principles of service-orientation consist of.

**SUMMARY OF KEY POINTS**

- The logical parts of an SOA can be mapped to corresponding components in the basic Web services framework.
- By viewing a service-oriented solution as a unit of automation logic, we establish that SOA consists of a sophisticated environment that supports a highly modularized separation of logic into differently scoped units.
- SOA further establishes specific characteristics, behaviors, and relationships among these components that provide a predictable environment in support of service-orientation.

8.3. Common principles of service-orientation

In Chapter 3 we established that there is no single definition of SOA. There is also no single governing standards body that defines the principles behind service-orientation. Instead, there are many opinions, originating from public IT organizations to vendors and consulting firms, about what constitutes service-orientation.

Service-orientation is said to have its roots in a software engineering theory known as "separation of concerns." This theory is based on the notion that it is beneficial to break down a large problem into a series of individual concerns. This allows the logic required to solve the problem to be decomposed into a collection of smaller, related pieces. Each piece of logic addresses a specific concern.

This theory has been implemented in different ways with different development platforms. Object-oriented programming and component-based programming approaches, for example, achieve a separation of concerns through the use of objects, classes, and components.

Service-orientation can be viewed as a distinct manner in which to realize a separation of concerns. The principles of service-orientation provide a means of supporting this theory while achieving a foundation paradigm upon which many contemporary SOA characteristics can be built. In fact, if you study these characteristics again, you will notice that several are (directly or indirectly) linked to the separation of concerns theory.

As previously mentioned, there is no official set of service-orientation principles. There are, however, a common set of principles most associated with service-orientation. These are listed below and described further in this section.

- Services are reusable Regardless of whether immediate reuse opportunities exist, services are designed to support potential reuse.
- Services share a formal contract For services to interact, they need not share anything but a formal contract that describes each service and defines the terms of information exchange.
- Services are loosely coupled Services must be designed to interact without the need for tight, cross-service dependencies.
- Services abstract underlying logic The only part of a service that is visible to the outside world is what is exposed.
via the service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestors.

- Services are composable Services may compose other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.
- Services are autonomous The logic governed by a service resides within an explicit boundary. The service has control within this boundary and is not dependent on other services for it to execute its governance.
- Services are stateless Services should not be required to manage state information, as that can impede their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.
- Services are discoverable Services should allow their descriptions to be discovered and understood by humans and service requestors that may be able to make use of their logic.

Of these eight, autonomy, loose coupling, abstraction, and the need for a formal contract can be considered the core principles that form the baseline foundation for SOA. As explained in the How service-orientation principles inter-relate section later in this chapter, these four principles directly support the realization of other principles (as well as each other).

There are other qualities commonly associated with services and service-orientation. Examples include self-descriptive and coarse-grained interface design characteristics. We classify these more as service design guidelines, and they are therefore discussed as part of the design guidelines provided in Chapter 15.

Note

You may have noticed that the reusability and autonomy principles also were mentioned as part of the contemporary SOA characteristics described in Chapter 3. This overlap is intentional, as we simply are identifying qualities commonly associated with SOA as a whole as well as services designed for use in SOA. We further clarify the relationship between contemporary SOA characteristics and service-orientation principles in Chapter 9.

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

### 8.3.1. Services are reusable

Service-orientation encourages reuse in all services, regardless if immediate requirements for reuse exist. By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased. Inherently reusable services also reduce the need for creating wrapper services that expose a generic interface over top of less reusable services.

This principle facilitates all forms of reuse, including inter-application interoperability, composition, and the creation of cross-cutting or utility services. As we established earlier in this chapter, a service is simply a collection of related operations. It is therefore the logic encapsulated by the individual operations that must be deemed reusable to warrant representation as a reusable service (Figure 8.13).

Figure 8.13. A reusable service exposes reusable operations.
Messaging also indirectly supports service reusability through the use of SOAP headers. These allow for messages to become increasingly self-reliant by grouping metadata details with message content into a single package (the SOAP envelope). Messages can be equipped with processing instructions and business rules that allow them to dictate to recipient services how they should be processed.

The processing-specific logic embedded in a message alleviates the need for a service to contain this logic. More importantly, it imposes a requirement that service operations become less activity-specific in other words, more generic. The more generic a service's operations are, the more reusable the service.

**Case Study**

RailCo delivered the Invoice Submission Service for the sole purpose of being able to connect to TLS's new B2B system. This Web service’s primary function therefore is to send electronic invoice documents to the TLS Accounts Payable Service. The service contains the following two operations: SubmitInvoice and GetTLSMetadata (Figure 8.14).
The SubmitInvoice operation simply initiates the transmission of the invoice document. You might recall in the Metadata exchange section of Chapter 7 that an operation was added to periodically check the TLS Accounts Payable Service for changes to its service description. This new operation is GetTLSMetadata.

Because they were built to meet immediate and specific business requirements, these operations have no real reuse potential. The SubmitInvoice operation is designed to forward SOAP messages containing specific headers required by TLS and containing an invoice XML document structured according to a schema also defined by TLS. By its very name, the GetTLSMetadata operation identifies itself as existing for one reason: to query a specific endpoint for new metadata information.

The TLS Accounts Payable Service, on the other hand, provides a series of generic operations related to the processing of accounts payable transactions. This service is therefore used by different TLS systems, one of which is the aforementioned B2B solution.

In Chapters 11 and 12 we will submit the RailCo Invoice Submission Service to a modeling exercise in an attempt to reshape it into a service that implements actual service-orientation principles, including reusability.

In Plain English

One day, a government inspector stops by our car washing operation. Not knowing who he is, I ask if he would like his car washed and waxed or just washed. He responds by asking a question of his own. "Do you have a business license for this operation?"

A subsequent conversation between the inspector and our team results in the revelation that we have indeed
been operating without a business license. We are therefore ordered to cease all work until we obtain one. We scramble to find out what needs to be done. This leads us to visit the local Business License Office to start the process of acquiring a license.

The Business License Office provides a distinct service: issuing and renewing business licenses. It is not there to service just our car washing company; it is there to provide this service to anyone requesting it. Because its service is designed to facilitate multiple service requestors, the logic that enables the service can be classified as being reusable.

8.3.2. Services share a formal contract

Service contracts provide a formal definition of:

- the service endpoint
- each service operation
- every input and output message supported by each operation
- rules and characteristics of the service and its operations

Service contracts therefore define almost all of the primary parts of an SOA (Figure 8.15). Good service contracts also may provide semantic information that explains how a service may go about accomplishing a particular task. Either way, this information establishes the agreement made by a service provider and its service requestors.

Figure 8.15. Service contracts formally define the service, operation, and message components of a service-oriented architecture.
Because this contract is shared among services, its design is extremely important. Service requestors that agree to this contract can become dependent on its definition. Therefore, contracts need to be carefully maintained and versioned after their initial release.

As explained in Chapter 5, service description documents, such as the WSDL definition, XSD schemas, and policies, can be viewed collectively as a communications contract that expresses exactly how a service can be programmatically accessed.

Case Study

From the onset, RailCo and TLS agreed to each other's service contracts, which enabled these two companies to interact via the TLS B2B system. The rules of the contract and the definition of associated service description documents all are provided by TLS to ensure a standardized level of conformance that applies to each of its online vendors.

One day, RailCo is informed that TLS has revised the policy published with the Accounts Payable Service. A new rule has been added where TLS is offering better payment terms to vendors in exchange for larger discounts. RailCo has the choice to continue pricing their products at the regular amounts and face a payment term of 60 days for their invoices or reduce their prices to get a payment term of 30 days.

Both of these options are acceptable contract conditions published by TLS. After some evaluation, RailCo decides not to take advantage of the reduced payment terms and therefore does not adjust its product prices.

In Plain English

For us to get a business license, we must fill out an application form. This process essentially formalizes our request in a format required and expected by the Business License Office.

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

8.3.3. Services are loosely coupled

No one can predict how an IT environment will evolve. How automation solutions grow, integrate, or are replaced over time can never be accurately planned out because the requirements that drive these changes are almost always external to the IT environment. Being able to ultimately respond to unforeseen changes in an efficient manner is a key goal of applying service-orientation. Realizing this form of agility is directly supported by establishing a loosely coupled relationship between services (Figure 8.16).
Figure 8.16. Services limit dependencies to the service contract, allowing underlying provider and requestor logic to remain loosely coupled.

Loose coupling is a condition wherein a service acquires knowledge of another service while still remaining independent of that service. Loose coupling is achieved through the use of service contracts that allow services to interact within predefined parameters.

It is interesting to note that within a loosely coupled architecture, service contracts actually tightly couple operations to services. When a service is formally described as being the location of an operation, other services will depend on that operation-to-service association.

**Case Study**

Through the use of service contracts, RailCo and TLS services are naturally loosely coupled. However, one could say that the extent of loose coupling between the two service provider entities is significantly different.

TLS services are designed to facilitate multiple B2B partners, as well as internal reuse and composition requirements. This makes TLS services very loosely coupled from any of its service requestors.

RailCo's services, on the other hand, are designed specifically to interact with designated TLS services that are part of the overall TLS B2B solution. No attempt was made to make these services useful for any other service requestors. RailCo services are therefore considered less loosely coupled than TLS services.

**In Plain English**

After we have submitted our form, we are not required to remain at the Business License Office, nor do we need to stay in touch with them. We only need to wait until the application is processed and a license is
(hopefully) issued.

This is much like an asynchronous message exchange, but it is also a demonstration of a loosely coupled relationship between services or between service provider and requestor. All we need to interact with the Business License Office is an application form that defines the information the office requires to process our request. Prior to and subsequent to the submission of that request, our car washing team (service requestor) and the Business License Office (service provider) remain independent of each other.

8.3.4. Services abstract underlying logic

Also referred to as service interface-level abstraction, it is this principle that allows services to act as black boxes, hiding their details from the outside world. The scope of logic represented by a service significantly influences the design of its operations and its position within a process.

There is no limit to the amount of logic a service can represent. A service may be designed to perform a simple task, or it may be positioned as a gateway to an entire automation solution. There is also no restriction as to the source of application logic a service can draw upon. For example, a single service can, technically, expose application logic from two different systems (Figure 8.17).

Figure 8.17. Service operations abstract the underlying details of the functionality they expose.
Operation granularity is therefore a primary design consideration that is directly related to the range and nature of functionality being exposed by the service. Again, it is the individual operations that collectively abstract the underlying logic. Services simply act as containers for these operations.

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

**Case Study**

Because both RailCo and TLS employ Web services to communicate, each environment successfully implements service interface-level abstraction. On RailCo’s end, this abstraction hides the legacy systems involved with generating electronic invoice documents and processing incoming purchase orders. On the TLS side, services hide service compositions wherein processing duties are delegated to specialized services as part of single activities (Figure 8.18).

![Figure 8.18. Neither of RailCo's or TLS's service requestors require any knowledge of what lies behind the other's service providers.](View full size image)
In Plain English

The tasks required for the Business License Office to process our request include:

- A name check to ensure that the name of our company "Oasis Car Wash" isn't already taken.
- A background check of the company principals to ensure that none of us have had past bankruptcies.
- A verification of our sub-lease agreement to ensure that we are, in fact, allowed to operate at the gas station we have been using.

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

8.3.5. Services are composable

A service can represent any range of logic from any types of sources, including other services. The main reason to implement this principle is to ensure that services are designed so that they can participate as effective members of other service compositions if ever required. This requirement is irrespective of whether the service itself composes others to accomplish its work (Figure 8.19).

Figure 8.19. The UpdateEverything operation encapsulating a service composition.

A common SOA extension that underlines compositability is the concept of orchestration. Here, a service-oriented process (which essentially can be classified as a service composition) is controlled by a parent process service that composes process participants.

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

Case Study

As with RailCo’s Invoice Submission Service, its Order Fulfillment Service was created to meet a specific requirement in support of communication with TLS’s B2B solution.

The Order Fulfillment Service contains just one public operation called ProcessTLSPO (Figure 8.20). This operation is designed in compliance with TLS vendor service specifications so that it is fully capable of receiving POs submitted by the TLS Purchase Order Service. Part of this compliance requires the operation to be able to process custom SOAP headers containing proprietary security tokens.

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

Though the Order Fulfillment Service is capable of acting as a composition member, its potential for being useful to any future compositions is limited. Composition support is similar to reusability in that generic functionality exposed by operations make a service more composable. This RailCo service provides one operation that performs a very specialized function, customized to processing a specific document from a specific source. It will likely not be a suitable composition member, but it can act as a controller service, composing other services to complete its PO processing tasks.

The TLS Accounts Payable Service already establishes a well-defined composition, wherein it acts as a controller service that composes the Vendor Profile and Ledger Services (Figure 8.21). Because they each expose a complete set of generic operations, all three of these services are capable of participating in other composition configurations. Figure 8.21. The TLS Accounts Payable Service Composition.
In Plain English

Given that the services provided by the Business License Office are distinct and reusable, it can be asked to assist other government offices to participate in the completion of other services. For example, the Business Relocation Office manages all administrative paperwork for businesses that need to be moved when their location is scheduled for demolition.

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

By reusing the services offered by the Business License Office, the Business Relocation Office has effectively composed services, much like a controller service reuses and composes other service providers.

8.3.6. Services are autonomous

Autonomy requires that the range of logic exposed by a service exist within an explicit boundary. This allows the service to execute self-governance of all its processing. It also eliminates dependencies on other services, which frees a service from ties that could inhibit its deployment and evolution (Figure 8.22). Service autonomy is a primary consideration when deciding how application logic should be divided up into services and which operations should be grouped together within a service context. Figure 8.22. Autonomous services have control over underlying resources.
Deferring the location of business rules is one way to strengthen autonomy and keep services more generic. Processes generally assume this role by owning the business rules that determine how the process is structured and, subsequently, how services are composed to automate the process logic. This is another aspect of orchestration explored in the Orchestration service layer section in Chapter 9.

Note that autonomy does not necessarily grant a service exclusive ownership of the logic it encapsulates. It only guarantees that at the time of execution, the service has control over whatever logic it represents. We therefore can make a distinction between two types of autonomy.

- Service-level autonomy Service boundaries are distinct from each other, but the service may share underlying resources. For example, a wrapper service that encapsulates a legacy environment that also is used independently from the service has service-level autonomy. It governs the legacy system but also shares resources with other legacy clients.
- Pure autonomy The underlying logic is under complete control and ownership of the service. This is typically the case when the underlying logic is built from the ground up in support of the service.

### Case Study

Given the distinct tasks they perform, the following three RailCo services all are autonomous:

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

Each represents a specific boundary of application logic that does not overlap with the boundary of any other services.

Autonomy in RailCo's services was achieved inadvertently. No conscious effort was made to avoid application overlap, as the services were delivered to simply meet specific connectivity requirements. As shown in Figure 8.23, the Invoice Processing and Order Fulfillment Services encapsulate legacy logic. The legacy accounting system also is used by clients independently from the services, which makes this service-level autonomy. The TLS Notification Service achieves pure autonomy, as it represents a set of custom components created only in support of this service.

**Figure 8.23. RailCo's services luckily encapsulate explicit portions of legacy newly added application**
In environments where a larger number of services exist and new services are built on a regular basis, it is more common to introduce dedicated modeling processes so pure service autonomy is preserved among individual services. At TLS, for example, services undergo a service-oriented analysis to guarantee autonomy and avoid encapsulation overlap. (Service-oriented analysis is explained in Chapters 11 and 12.)

**In Plain English**

Let's revisit the three tasks performed by the Business License Office when processing an application for a new business license:

- name check
- background check
- location verification

The Business License Office owns the corporate name database required to perform a name check. Also the office has personnel dedicated to visiting and verifying business site locations. When completing these two tasks, the Business License Office therefore has complete self-governance. However, when having to
perform a background check, the office must share a database system with the Revenue Office. When it gets access, it can retrieve an abbreviated credit history for each of the company principals listed on the application.

The Business License Office's reliance on the shared database reduces its independence somewhat. However, its overall ability to perform the tasks within its own boundary give it a degree of autonomy.

8.3.7. Services are stateless

Services should minimize the amount of state information they manage and the duration for which they hold it. State information is data-specific to a current activity. While a service is processing a message, for example, it is temporarily stateful (Figure 8.24). If a service is responsible for retaining state for longer periods of time, its ability to remain available to other requestors will be impeded.

![Figure 8.24. Stateless and stateful stages a service passes through while processing a message.](image)

Statelessness is a preferred condition for services and one that promotes reusability and scalability. For a service to retain as
little state as possible, its individual operations need to be designed with stateless processing considerations.

A primary quality of SOA that supports statelessness is the use of document-style messages. The more intelligence added to a message, the more independent and self-sufficient it remains. Chapters 6 and 7 explore various WS-* extensions that rely on the use of SOAP headers to carry different types of state data.

**Case Study**

As with loose coupling, statelessness is a quality that can be measured in degrees. The RailCo Order Fulfillment Service is required to perform extra runtime parsing and processing of various standard SOAP header blocks to successfully receive a purchase order document submitted by the TLS Purchase Order Service. This processing ties up the Order Fulfillment Service longer than, say, the Invoice Submission Service, which simply forwards a predefined SOAP message to the TLS Accounting Service.

**In Plain English**

During the initial review of the application, our company was briefly discussed by personnel at the Business License Office. But after the application was fully processed, no one really retained any memory of our request.

Though the details of our application have been logged and recorded in various repositories, there is no further need for anyone involved in the processing of our request to remember further information about it once the application processing task was completed. To this extent, the Business License Office simulates a degree of statelessness. It processes many requests every day, and there is no benefit to retaining information about a completed request.

**8.3.8. Services are discoverable**

Discovery helps avoid the accidental creation of redundant services or services that implement redundant logic. Because each operation provides a potentially reusable piece of processing logic, metadata attached to a service needs to sufficiently describe not only the service's overall purpose, but also the functionality offered by its operations.

Note that this service-orientation principle is related to but distinct from the contemporary SOA characteristic of discoverability. On an SOA level, discoverability refers to the architecture's ability to provide a discovery mechanism, such as a service registry or directory. This effectively becomes part of the IT infrastructure and can support numerous implementations of SOA. On a service level, the principle of discoverability refers to the design of an individual service so that it can be as discoverable as possible.

**Case Study**

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

Similarly, the RailCo Order Fulfillment Service was registered manually with the TLS B2B solution so that it would be placed on the list of vendors that receive purchase orders. This service provides no reusable functionality and is therefore considered to have no immediate requirement for discovery.

Due to the reusable nature of TLS services and because of the volume of services that are expected to exist in TLS technical environments, an internal service registry was established (as shown in Figure 8.25 and originally explained in Chapter 5). This piece of TLS infrastructure promotes discoverability and prevents accidental redundancy. It further leverages the existing design standards used by TLS that promote the creation of descriptive metadata documents in support of service discoverability.

Figure 8.25. RailCo's services are not discoverable, but TLS's inventory of services are stored in an internal registry.

TLS is not interested in making its services publicly discoverable, which is why it does not register them with a public service registry. Vendors that participate in the TLS B2B system only are allowed to do so after a separate negotiation, review, and registration process.
In Plain English

After some time, our business license is finally issued. Upon receiving the certificate in the mail, we are back in business. Looking back at how this whole process began, though, there is one step we did not discuss. When we first learned that we were required to get a business license, we had to find out where the Business License Office was located. This required us to search through the phone book and locate a listing with contact information.

A service registry provides a discovery mechanism very much like a phone book, allowing potential requestors to query and check candidate service providers. In the same manner in which a registry points to service descriptions, the phone book listing led us to the location at which we were able to obtain the original business license application form.

More relevant to the principle of service discoverability is the fact that steps were taken to make the Business License Office itself discoverable. Examples include signs in the lobby of the high-rise in which the office is located, a sign on the office entrance door, brochures located at other offices, and so on.

SUMMARY OF KEY POINTS

- Different organizations have published their own versions of service-oriented principles. As a result, many variations exist.
- The most common principles relate to loose coupling, autonomy, discoverability, composability, reuse, service contracts, abstraction, and statelessness.
UNIT – II

5.1. The Web services framework

A technology framework is a collection of things. It can include one or more architectures, technologies, concepts, models, and even sub-frameworks. The framework established by Web services is comprised of all of these parts.

Specifically, this framework is characterized by:

- an abstract (vendor-neutral) existence defined by standards organizations and implemented by (proprietary) technology platforms
- core building blocks that include Web services, service descriptions, and messages
- a communications agreement centered around service descriptions based on WSDL
- a messaging framework comprised of SOAP technology and concepts
- a service description registration and discovery architecture sometimes realized through UDDI
- a well-defined architecture that supports messaging patterns and compositions (covered in Chapter 6)
- a second generation of Web services extensions (also known as the WS-* specifications) continually broadening its underlying feature-set (covered in Chapters 6 and 7)

Another recommended addition to this list is the WS-I Basic Profile (introduced in Chapter 4 and further explained in later chapters). It provides standards and best practices that govern the usage of WSDL, SOAP, and UDDI features. Therefore, much of what the Web services framework is comprised of can be standardized by the Basic Profile.

In its entirety this technology framework is conceptually in alignment with the principles of service-orientation. To further explore this synergy, the next three sections are intentionally labeled to mirror the three sub-sections from Chapter 3 in which we first defined the parts of primitive SOA (Figure 5.1).
SUMMARY OF KEY POINTS

- First- and second-generation technologies, along with design-agnostic concepts and implementation-neutral architectures, form an abstract Web services framework.
- The fundamentals of this framework are in alignment with the core characteristics of primitive SOA.

5.2. Services (as Web services)
In Chapter 3 we introduced the concept of services and how they provide a means of encapsulating various extents of logic. Manifesting services in real world automation solutions requires the use of a technology capable of preserving fundamental service-orientation, while implementing real world business functionality.

Web services provide the potential of fulfilling these primitive requirements, but they need to be intentionally designed to do so. This is because the Web services framework is flexible and adaptable. Web services can be designed to duplicate the behavior and functionality found in proprietary distributed systems, or they can be designed to be fully SOA-compliant. This flexibility has allowed Web services to become part of many existing application environments and has been one of the reasons behind their popularity. It also reveals the fact that Web services are not necessarily inherently service-oriented.

Note
We use the terms "Web services" and "services" interchangeably throughout this book.

Let's start with an overview of the most basic Web services design concepts. Fundamentally, every Web service can be associated with:

- a temporary classification based on the roles it assumes during the runtime processing of a message
- a permanent classification based on the application logic it provides and the roles it assumes within a solution environment

We explore both of these design classifications in the following two sections:

- service roles (temporary classifications)
- service models (permanent classifications)

**In Plain English**

Bob and Jim are two old friends I occasionally meet. During our last get-together we talked about Chuck, another friend we used to hang out with, but with whom we lost touch several years ago. None of us know where he went off to, and we think it would be fun to see if we could contact him again. Bob says that he's heard of an agency called "Reconnect" that specializes in this particular type of service.

For us to make use of any of the services offered by Reconnect, we must:

1. Find out (discover) how we can contact this agency.
2. Compile the information about Chuck that the agency will require to perform the search (formulate a request).
3. Forward this information (issue the request) to the agency.
(These steps are explained individually in the subsequent In Plain English sections.)

Reconnect is providing a service and can therefore be seen as fulfilling the role of service provider. We are requesting a service and are therefore considered to be acting as service requestors.

5.2.1. Service roles

A Web service is capable of assuming different roles, depending on the context within which it is used. For example, a service can act as the initiator, relayer, or the recipient of a message. A service is therefore not labeled exclusively as a client or server, but instead as a unit of software capable of altering its role, depending on its processing responsibility in a given scenario.

It is not uncommon for a Web service to change its role more than once within a given business task. It is especially not uncommon for a Web service within an SOA to assume different roles in different business tasks.

Provided here are descriptions of the fundamental service roles.

Service provider

The service provider role is assumed by a Web service under the following conditions:

- The Web service is invoked via an external source, such as a service requestor (Figure 5.2)

![Figure 5.2. As the recipient of a request message, the Web service is classified as a service provider.](image)

- The Web service provides a published service description offering information about its features and behavior. (Service descriptions are explained later in this chapter.)
The service provider role is synonymous with the server role in the classic client-server architecture. Depending on the type of message exchange used when invoking a service provider, the service provider may reply to a request message with a response message. (Types of message exchanges are categorized as "message exchange patterns," which are explained in the next chapter.)

The term "service provider" also is used to identify the organization (or individual) responsible for actually providing the Web service. To help distinguish the service role from the service's actual provider, the following, more qualified terms are sometimes used:

- service provider entity (the organization or individual providing the Web service)
- service provider agent (the Web service itself, acting as an agent on behalf of its owner)

It is, however, most common to simply refer to the service being invoked as the service provider.

Note

A service provider agent is different from a service agent, which is a small runtime program used to perform generic processing tasks in support of Web services. Service agents are explained in Chapter 18.

Service requestor

Any unit of processing logic capable of issuing a request message that can be understood by the service provider is classified as a service requestor. A Web service is always a service provider but also can act as a service requestor.

Note

Almost all of the service requestors discussed in this book are classified as Web services and are referenced and depicted as such. Chapter 18 provides platform-specific details about how Web services are implemented and how the physical parts of a Web service interact in service provider and service requestor roles.

A Web service takes on the service requestor role under the following circumstances:

- The Web service invokes a service provider by sending it a message (Figure 5.3).

Figure 5.3. The sender of the request message is classified as a service requestor.
The Web service searches for and assesses the most suitable service provider by studying available service descriptions. (Service descriptions and service registries are covered in the Service descriptions (with WSDL) section.)

The service requestor is the natural counterpart to the service provider, comparable to the client in a typical client-server environment. It is important to note that a service provider is not acting as a service requestor when it sends a message in response to a request message. A service requestor is best viewed as a software program that initiates a conversation with a service provider.

As with "service provider," this term is subject to some ambiguity. A service requestor can represent both the Web service itself as well as the Web service owner. Therefore, the following extended terms are available (but not really used that often):

- service requestor entity (the organization or individual requesting the Web service)
- service requestor agent (the Web service itself, acting as an agent on behalf of its owner)

Note

Another term frequently used instead of service requestor is service consumer.
Case Study

RailCo is one of many long-time vendors used by TLS. Historically, it was the primary air brake parts supplier TLS relied upon. Until recently, TLS had to order parts from RailCo via phone or fax. When a new air brake supplier surfaced, offering competitive prices and signing up with TLS's B2B solution, there was little need for TLS to continue exclusively with RailCo. In fact, TLS only contacted RailCo again when its new primary vendor could not supply a requested part.

For RailCo to join its competitor as an online partner of TLS, it had to conform to rules and specifications defined by TLS. Specifically, TLS dictates that every supplier must allow TLS to programmatically interface with their inventory control system to submit purchase orders. Additionally, the supplier must be able to connect to TLS's external accounting interface to submit invoices and back-order information.

These policies forced RailCo to build an extension to their accounting system, capable of interacting with TLS's Web service-based B2B solution. After RailCo's application went online, the most common data exchange scenarios were as follows:

- TLS's Purchase Order Service submits electronic POs that are received by RailCo's Order Fulfillment Service.
- Upon shipping the order, RailCo's Invoice Submission Service sends an electronic invoice to TLS's Accounts Payable Service.

Figure 5.4 illustrates these two message exchanges.

Figure 5.4. TLS and RailCo services swapping roles in different but related message exchanges.

In the first scenario, TLS acts as the service requestor entity. Its Purchase Order Service was the service
requestor (or service requestor agent) that initiated the interaction. Being the recipient of the order request, the Order Fulfillment Service is classified as the service provider (or service provider agent). As the owner of this Web service, RailCo is the service provider entity.

The roles are reversed in the second scenario, where RailCo is the service requestor entity because its Invoice Submission Service acts as the service requestor. TLS's Accounts Payable Service receives the invoice message, making that Web service the service provider, and TLS the service provider entity.

In Plain English

Because we need to send a piece of mail to Reconnect for them to initiate the search, we also introduce an intermediary into this scenario. Although the letter is addressed to Reconnect, it is actually first sent to the Post Office. From there it is routed to a carrier that delivers it to the Reconnect office. In this scenario, the Post Office acts as an intermediary that provides a routing service.

Intermediaries

The communications framework established by Web services contrasts the predictable nature of traditional point-to-point communications channels. Though less flexible and less scalable, point-to-point communication was a great deal easier to design. Web services communication is based on the use of messaging paths, which can best be described as point-to-* paths. This is because once a service provider submits a message, it can be processed by multiple intermediate routing and processing agents before it arrives at its ultimate destination. (Message paths are explained at the end of this chapter.)

Web services and service agents that route and process a message after it is initially sent and before it arrives at its ultimate destination are referred to as intermediaries or intermediary services. Because an intermediary receives and submits messages, it always transitions through service provider and service requestor roles (Figure 5.5).

Figure 5.5. The intermediary service transitions through service provider and service requestor roles while processing a message.
There are two types of intermediaries. The first, known as a passive intermediary, is typically responsible for routing messages to a subsequent location (Figure 5.6). It may use information in the SOAP message header to determine the routing path, or it may employ native routing logic to achieve some level of load balancing. Either way, what makes this type of intermediary passive is that it does not modify the message.

**Figure 5.6. A passive intermediary service processing a message without altering its contents.**

![Figure 5.6. A passive intermediary service processing a message without altering its contents.](View full size image)

**Case Study**

After shipping a TLS order, RailCo's Invoice Submission Service transmits a message containing an electronic invoice. The first TLS Web service to receive the message is a passive intermediary called the Load Balancing Service. Its purpose is to provide load balancing logic by checking the current processing statistics of available TLS servers. When the server with the lowest usage is identified, this passive intermediary routes the message accordingly.

Upon receiving the message from the Invoice Submission Service requestor, the passive Load Balancing intermediary acts as the service provider. After it has determined where the message is to be forwarded to, it changes its role to service requestor to forward the invoice document to the destination Accounts Payable Service provider.

**Note**

The Load Balancing Service (and the upcoming Internal Policy Service) is a form of intermediary that can
be explicitly accessed as a Web service through a WSDL or it can act as a service agent. Service agents are intermediaries designed to intercept and process messages en route to their ultimate destinations and are explained further in Chapter 18. TLS opted to develop flexible intermediaries to fulfill requirements specific to their environments.

Like passive intermediary services, active intermediaries also route messages to a forwarding destination. Prior to transmitting a message, however, these services actively process and alter the message contents (Figure 5.7). Typically, active intermediaries will look for particular SOAP header blocks and perform some action in response to the information they find there. They almost always alter data in header blocks and may insert or even delete header blocks entirely. (Header blocks are explained later in this chapter.)

Case Study

TLS employs a number of active intermediaries. The Internal Policy Service, for example, examines the message to determine whether it is subject to any internal policy restrictions. If it is, the active intermediary inserts a new header block containing one or more policy rules used by subsequent service providers. As
with the passive intermediary example, the active intermediary transitions through service provider and service requestor roles before finally forwarding the message to the appropriate TLS service provider.

**Initial sender and ultimate receiver**

Initial senders are simply service requestors that initiate the transmission of a message. Therefore, the initial sender is always the first Web service in a message path. The counterpart to this role is the ultimate receiver. This label identifies service providers that exist as the last Web service along a message's path (Figure 5.8).

**Figure 5.8.** Web services acting as initial sender and ultimate receiver.

Note that intermediary services can never be initial senders or ultimate receivers within the scope of a service activity.

**Case Study**

Expanding on the previous example that demonstrated the use of a passive intermediary, let's take a look at all the services involved in that message exchange. In this scenario, we had the RailCo Invoice Submission...
Service (acting as the service requestor) initiating the message transmission. By receiving the message, the Load Balancing intermediary acts as the service provider. Upon routing the message, the intermediary temporarily assumes the service requestor role and sends the message to the Accounts Payable Service, another service provider (Figure 5.9).

Figure 5.9. The TLS Load Balancing Service acting as an intermediary between the RailCo initial sender and the TLS ultimate receiver.

These three physical services created four logical roles to complete two service requestor-to-service provider transmissions. There was, however, only one Web service that initiated the transmission. This was the Invoice Submission Service, and it is therefore considered the initial sender. Similarly, there was only one Web service that ended the overall activity, which makes the Accounts Payable Service the ultimate receiver.

Service compositions

As the name suggests, this particular term does not apply to a single Web service, but to a composite relationship between a collection of services. Any service can enlist one or more additional services to complete a given task. Further, any of the enlisted services can call other services to complete a given sub-task. Therefore, each service that participates in a composition assumes an individual role of service composition member (Figure 5.10).
Typically, Web services need to be designed with service composition in mind to be effective composition members. Service-orientation principles place an emphasis on composable, allowing some Web services to be designed in such a manner that they can be pulled into future service compositions without a foreknowledge of how they will be utilized.

The concept of service composable is very important to service-oriented environments (and is explained as a service-orientation principle in Chapter 8). In fact, service composition is frequently governed by WS-* composition extensions, such as WS-BPEL and WS-CDL, which introduce the related concepts of orchestration and choreography, respectively (as explained in Chapter 6).

Note

Service compositions also are referred to as service assemblies.

**Case Study**

When the TLS Accounts Payable Service receives an invoice, it invokes a series of additional services to fully process the invoice contents:

1. It first uses the Vendor Profile Service to validate the invoice header data and link the invoice document to a vendor account.
2. Next, the Accounts Payable Service extracts taxes and shipping fees and directly logs all amounts into the appropriate A/P accounts.
3. Finally, the Accounts Payable Service passes the Ledger Service the invoice total, which it uses to
update the General Ledger.

In this scenario our service composition consists of three composition members, spearheaded by the Accounts Payable Service (Figure 5.11).

Figure 5.11. The Accounts Payable Service enlisting other TLS services in a service composition.

Note

A characteristic of this particular composition that is not discussed here is the fact that all three actions we described would very likely be wrapped in a transaction. Should one of them fail, all others would be rolled back. (Concepts relating to service transactions are covered in Chapter 6.)
5.2.2. Service models

The roles we've explored so far are agnostic to the nature of the functionality being provided by the Web service. They are generic states that a service can enter within a generic context. The manner in which services are being utilized in the real world, though, has led to a classification based on the nature of the application logic they provide, as well as their business-related roles within the overall solution. These classifications are known as service models.

The sections that follow describe the basic set of common service models. Additional service models are introduced in later chapters. Appendix B further provides a reference table listing all service models covered in this book. It is important to note that a service can and frequently does belong to more than one service model.

**Business service model**

Within an SOA, the business service represents the most fundamental building block. It encapsulates a distinct set of business logic within a well-defined functional boundary. It is fully autonomous but still not limited to executing in isolation, as business services are frequently expected to participate in service compositions.

Business services are used within SOAs as follows:

- as fundamental building blocks for the representation of business logic
- to represent a corporate entity or information set
- to represent business process logic
- as service composition members

For future reference, when building an SOA around layers of abstraction, the business service model can correspond to the business service layer introduced in Chapter 9. In this case the business service would act as a controller, composing utility application services. (The controller service model is explained shortly.)

**Utility service model**

Any generic Web service or service agent designed for potential reuse can be classified as a utility service. The key to achieving this classification is that the reusable functionality be completely generic and non-application specific in nature.

Utility services are used within SOAs as follows:

- as services that enable the characteristic of reuse within SOA
- as solution-agnostic intermediary services
- as services that promote the intrinsic interoperability characteristic of SOA
- as the services with the highest degree of autonomy

When working with the service abstraction layers described in Chapter 9, a utility service is most commonly associated with the application service layer. As a result, a utility service can be referred to as a utility application service.
Case Study

In the examples we've gone through so far in this chapter, we've described eight Web services. Six of these are business services, while the other two are utility services, as follows:

- Accounts Payable Service = business service
- Internal Policy Service = utility service
- Invoice Submission Service = business service
- Ledger Service = business service
- Load Balancing Service = utility service
- Order Fulfillment Service = business service
- Purchase Order Service = business service
- Vendor Profile Service = business service

The Load Balancing and Internal Policy Services are classified as utility services because they provide generic functionality that can be reused by different types of applications. The application logic of the remaining services is specific to a given business task or solution, which makes them business-centric services.

Controller service model

Service compositions are comprised of a set of independent services that each contribute to the execution of the overall business task. The assembly and coordination of these services is often a task in itself and one that can be assigned as the primary function of a dedicated service or as the secondary function of a service that is fully capable of executing a business task independently. The controller service fulfills this role, acting as the parent service to service composition members.

Controller services are used within SOAs as follows:

- to support and implement the principle of composability
- to leverage reuse opportunities
- to support autonomy in other services

Note that controller services themselves can become subordinate service composition members. In this case the composition coordinated by a controller is, in its entirety, composed into a larger composition. In this situation there may be a master controller service that acts as the parent to the entire service composition, as well as a sub-controller, responsible for coordinating a portion of the composition (Figure 5.12).

Figure 5.12. A service composition consisting of a master controller, a sub-controller, four business services, and one utility service.
The controller service model is used frequently when building SOA with specialized service abstraction layers, as explained later in Chapter 9.

Case Study

In our previous example we demonstrated how the Accounts Payable Service initiated and coordinated a service.
composition consisting of two additional composition members. That would classify the Accounts Payable Service as a controller service. The fact that we already labeled this service as a business service does not conflict with this new classification; a single service can be classified as more than one service model (Figure 5.13).

Figure 5.13. The Accounts Payable Service acting as a business and controller service, composing two other business services.
SUMMARY OF KEY POINTS

- Web services can be labeled using temporary and permanent classifications.
- Temporary classifications relate to roles assumed by a service at runtime. For example, an intermediary service can transition through different roles while processing a message.
- Service models refer to permanent classifications that represent the logic housed by the service, as well as its role within the overall solution. A service can belong to more than one service model.