Chapter 4

Service-Orientation

Having covered some of the basic elements of service-oriented computing, we now narrow our focus on service-orientation. The next set of sections establish the paradigm of service-orientation and explain how it is changing the face of distributed computing.

4.1 Introduction to Service-Orientation

In the every day world around us, services are and have been commonplace for as long as civilized history has existed. Any person carrying out a distinct task in support of others is providing a service (Figure 4.1). Any group of individuals collectively performing a task is also demonstrating the delivery of a service.

dispatcher
driver
bookkeeper
"I take calls and arrange deliveries"
"I make deliveries"
"I take care of the accounting"

Figure 4.1
Three individuals, each capable of providing a distinct service.

Similarly, an organization that carries out tasks associated with its purpose or business is also providing a service. As long as the task or function being provided is well-defined and can be relatively isolated from other associated tasks, it can be distinctly classified as a service (Figure 4.2).

Certain baseline requirements exist to enable a group of individual service providers to collaborate in order to collectively provide a larger service. Figure 4.2, for example, displays a group of employees that each provide a service for ABC Delivery. Even though each individual contributes a distinct service, for the company to function effectively, its staff also needs to have fundamental, common characteristics, such as availability, reliability, and the ability to communicate using the same language. With all of this in place, these individuals can be composed into a productive working team. Establishing these types of baseline requirements is a key goal of service-orientation.

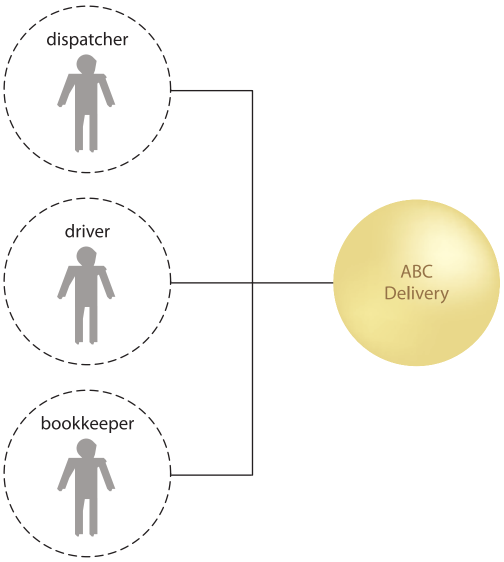

Figure 4.2
A company that employs these three people can compose their capabilities to carry out its business.

Services in Business Automation

In the world of SOA and service-orientation, the term "service" is not generic. It has specific connotations that relate to a unique combination of design characteristics. When solution logic is consistently built as services and when services are consistently designed with these common characteristics, service-orientation is successfully realized throughout an environment.

For example, one of the primary service design characteristics explored as part of this study of service-orientation is reusability. A strong emphasis on producing solution logic in the format of services that are positioned as highly generic and reusable enterprise resources gradually transitions an organization to a state where more and more of its solution logic becomes less dependent on and more agnostic to any one purpose or business process. Repeatedly fostering this characteristic within services eventually results in wide-spread reuse potential.

Consistently realizing specific design characteristics requires a set of guiding principles. This is what the service-orientation design paradigm is all about.

Services Are Collections of Capabilities

When discussing services, it is important to remember that a single service can provide a collection of capabilities. They are grouped together because they relate to a functional context established by the service. The functional context of the service illustrated in Figure 4.3, for example, is that of "shipment." Therefore, this particular service provides a set of capabilities associated with the processing of shipments.

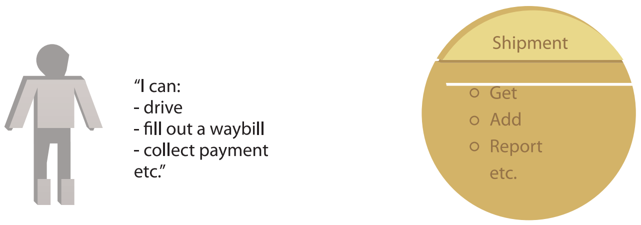

Figure 4.3
Much like a human, an automated service can provide multiple capabilities.

A service can essentially act as a container of related capabilities. It is comprised of a body of logic designed to carry out these capabilities and a service contract that expresses which of its capabilities are made available for public invocation.

References to service capabilities in this book are specifically focused on those that are defined in the service contract. For a discussion of how service capabilities are distinguished from Web service operations and component methods, see the *Principles and Service Implementation Mediums* section in Chapter 5.

Service-Orientation as a Design Paradigm

As established in Chapter 3, a design paradigm is an approach to designing solution logic. When building distributed solution logic, design approaches revolve around a software engineering theory known as the *separation of concerns*. In a nutshell, this theory states that a larger problem is more effectively solved when decomposed into a set of smaller problems or *concerns*. This gives us the option of partitioning solution logic into capabilities, each designed to solve an individual concern. Related capabilities can be grouped into units of solution logic.

The fundamental benefit to solving problems this way is that a number of the solution logic units can be designed to solve immediate concerns while still remaining agnostic to the greater problem. This provides the constant opportunity for us to reutilize the capabilities within those units to solve other problems as well.

Different design paradigms exist for distributed solution logic. What distinguishes service-orientation is the manner in which it carries out the separation of concerns and how it shapes the individual units of solution logic. Applying service-orientation to a meaningful extent results in solution logic that can be safely classified as "service-oriented" and units that qualify as “services.” To understand exactly what that means requires an appreciation of the strategic goals covered in Chapter 3 combined with knowledge of the associated design principles documented in Part II.

For now, let’s briefly introduce each of these principles:

Standardized Service Contract

Services express their purpose and capabilities via a service contract. The Standardized Service Contract design principle is perhaps the most fundamental part of service-orientation in that it essentially requires that specific considerations be taken into account when designing a service’s public technical interface and assessing the nature and quantity of content that will be published as part of a service’s official contract.

A great deal of emphasis is placed on specific aspects of contract design, including the manner in which services express functionality, how data types and data models are defined, and how policies are asserted and attached. There is a constant focus on ensuring that service contracts are both optimized, appropriately granular, and standardized to ensure that the endpoints established by services are consistent, reliable, and governable.

Chapter 6 is dedicated to exploring this design principle in detail.

Service Loose Coupling

Coupling refers to a connection or relationship between two things. A measure of coupling is comparable to a level of dependency. This principle advocates the creation of a specific type of relationship within and outside of service boundaries, with a constant emphasis on reducing (“loosening”) dependencies between the service contract, its implementation, and its service consumers.

The principle of Service Loose Coupling promotes the independent design and evolution of a service’s logic and implementation while still guaranteeing baseline interoperability with consumers that have come to rely on the service’s capabilities. There are numerous types of coupling involved in the design of a service, each of which can impact the content and granularity of its contract. Achieving the appropriate level of coupling requires that practical considerations be balanced against various service design preferences.

Chapter 7 provides an in-depth exploration of this principle and introduces related patterns and concepts.

Service Abstraction

Abstraction ties into many aspects of service-orientation. On a fundamental level, this principle emphasizes the need to hide as much of the underlying details of a service as possible. Doing so directly enables and preserves the previously described loosely coupled relationship. Service Abstraction also plays a significant role in the positioning and design of service compositions.

Various forms of meta data come into the picture when assessing appropriate abstraction levels. The extent of abstraction applied can affect service contract granularity and can further influence the ultimate cost and effort of governing the service.

Chapter 8 covers several aspects of applying abstraction to different types of service meta data, along with processes and approaches associated with information hiding.

Service Reusability

Reuse is strongly advocated within service-orientation; so much so, that it becomes a core part of typical service analysis and design processes, and also forms the basis for key service models. The advent of mature, non-proprietary service technology has provided the opportunity to maximize the reuse potential of multi-purpose logic on an unprecedented level.

The principle of Service Reusability emphasizes the positioning of services as enterprise resources with agnostic functional contexts. Numerous design considerations are raised to ensure that individual service capabilities are appropriately defined in relation to an agnostic service context, and to guarantee that they can facilitate the necessary reuse requirements.

Variations and levels of reuse and associated agnostic service models are covered in Chapter 9, along with a study of how commercial product design approaches have influenced this principle.

Service Autonomy

For services to carry out their capabilities consistently and reliably, their underlying solution logic needs to have a significant degree of control over its environment and resources. The principle of Service Autonomy supports the extent to which other design principles can be effectively realized in real world production environments by fostering design characteristics that increase a service's reliability and behavioral predictability.

This principle raises various issues that pertain to the design of service logic as well as the service's actual implementation environment. Isolation levels and service normalization considerations are taken into account to achieve a suitable measure of autonomy, especially for reusable services that are frequently shared.

Chapter 10 documents the design issues and challenges related to attaining higher levels of service autonomy, and further classifies different forms of autonomy and highlights associated risks.

Service Statelessness

The management of excessive state information can compromise the availability of a service and undermine its scalability potential. Services are therefore ideally designed to remain stateful only when required. Applying the principle of Service Statelessness requires that measures of realistically attainable statelessness be assessed, based on the adequacy of the surrounding technology architecture to provide state management delegation and deferral options.

Chapter 11 explores the options and impacts of incorporating stateless design characteristics into service architectures.

Service Discoverability

For services to be positioned as IT assets with repeatable ROI they need to be easily identified and understood when opportunities for reuse present themselves. The service design therefore needs to take the "communications quality" of the service and its individual capabilities into account, regardless of whether a discovery mechanism (such as a service registry) is an immediate part of the environment.

The application of this principle, as well as an explanation of how discoverability relates to interpretability and the overall service discovery process, are covered in Chapter 12.

Service Composability

As the sophistication of service-oriented solutions continues to grow, so does the complexity of underlying service composition configurations. The ability to effectively compose services is a critical requirement for achieving some of the most fundamental goals of service-oriented computing.

Complex service compositions place demands on service design that need to be anticipated to avoid massive retro-fitting efforts. Services are expected to be capable of participating as effective composition members, regardless of whether they need to be immediately enlisted in a composition. The principle of Service Composability addresses this requirement by ensuring that a variety of considerations are taken into account.

How the application of this design principle helps prepare services for the world of complex compositions is described in Chapter 13.

Service-Orientation and Interoperability

One item that may appear to be absent from the preceding list is a principle along the lines of *"Services are Interoperable."* The reason this does not exist as a separate principle is because interoperability is fundamental to every one of the principles just described. Therefore, in relation to service-oriented computing, stating that services must be interoperable is just about as basic as stating that services must exist. Each of the eight principles supports or contributes to interoperability in some manner.

Here are just a few examples:

- Service contracts are standardized to guarantee a baseline measure of interoperability associated with the harmonization of data models.
- Reducing the degree of service coupling fosters interoperability by making individual services less dependent on others and therefore more open for invocation by different service consumers.
- Abstracting details about the service limits all interoperation to the service contract, increasing the long-term consistency of interoperability by allowing underlying service logic to evolve more independently.
- Designing services for reuse implies a high-level of required interoperability between the service and numerous potential service consumers.
- By raising a service's individual autonomy, its behavior becomes more consistently predictable, increasing its reuse potential and thereby its attainable level of interoperability.
- Through an emphasis on stateless design, the availability and scalability of services increase, allowing them to interoperate more frequently and reliably.

- Service Discoverability simply allows services to be more easily located by those who want to potentially interoperate with them.
- Finally, for services to be effectively composable they must be interoperable. The success of fulfilling composability requirements is often tied directly to the extent to which services are standardized and cross-service data exchange is optimized.

A fundamental goal of applying service-orientation is for interoperability to become a natural by-product, ideally to the extent that a level of intrinsic interoperability is established as a common and expected service design characteristic. Depending on the architectural strategy being employed, this extent may or may not be limited to a specific service inventory.

Of course, as with any other design characteristic, there are levels of interoperability a service can attain. The ultimate measure is generally determined by the extent to which service-orientation principles have been consistently and successfully realized (plus, of course, environmental factors such as the compatibility of wire protocols, the maturity level of the underlying technology platform, and adherence to technology standards).

NOTE

Increased intrinsic interoperability is one of the key strategic goals associated with service-oriented computing (as originally established in Chapter 3). For more detailed information about how service-orientation principles directly support this and other strategic goals, see Chapter 16.

SUMMARY OF KEY POINTS

- The service-orientation paradigm consists of eight distinct design principles, each of which fosters fundamental design characteristics, such as interoperability. These principles are explored individually in subsequent chapters.
- Interoperability is a natural by-product of applying service-orientation design principles.

4.2 Problems Solved by Service-Orientation

To best appreciate why service-orientation has emerged and how it is intended to improve the design of automation systems, we need to compare before and after perspectives. By studying some of the common issues that have historically plagued IT, we can begin to understand the solutions proposed by this design paradigm.