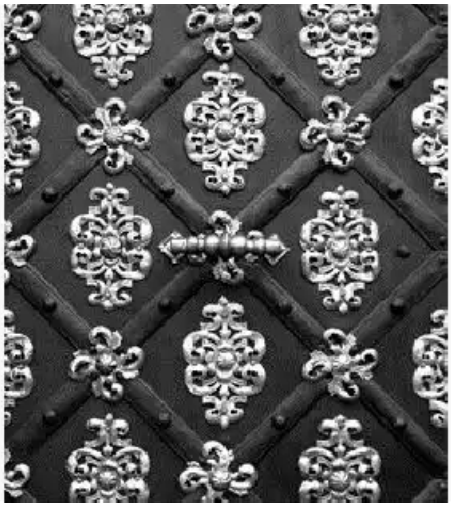
Chapter 9. Service Layers



While the service-orientation concepts we covered in the previous chapter are what fundamentally define SOA and distinguish it from other architectural platforms, they are still just theory. To bring service-orientation into a real-life automation solution, we need to provide an environment capable of supporting its fundamental principles.

As we've already established, the Web services framework provides us with the technology and the design paradigm with

which these principles can be realized. To then implement service-orientation in support of manifesting the contemporary SOA characteristics and benefits we identified back in Chapter 3, we need a means of coordinating and propagating service-orientation throughout an enterprise. This can be accomplished by service layer abstraction.

This chapter forms an approach to structuring and delivering specialized service layers around the delivery of key contemporary SOA characteristics.

How case studies are used: Both RailCo and TLS environments are revisited to identify which of the existing services correspond to the service layers discussed in this chapter.

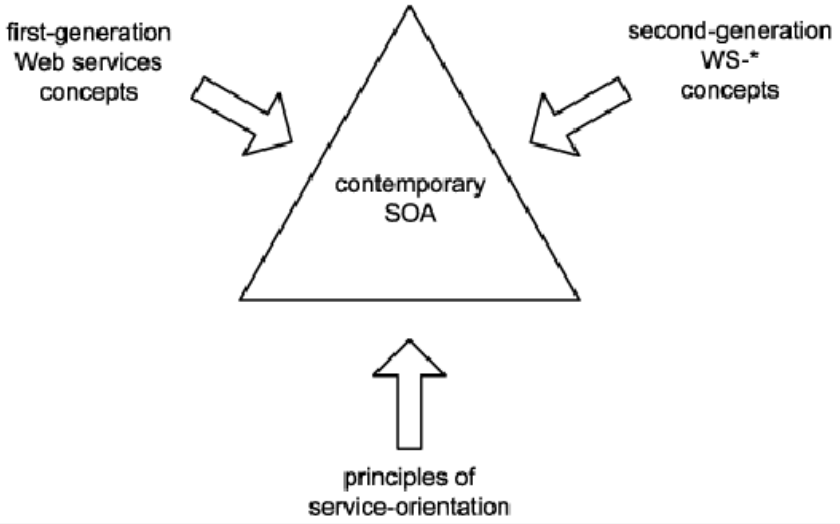
9.1. Service-orientation and contemporary SOA

Contemporary SOA is a complex and sophisticated architectural platform that offers significant potential to solve many historic and current IT problems. In part, it achieves this potential by leveraging existing paradigms and technologies that support its overall vision. However, the majority of what it has to offer only can be harnessed thorough analysis and targeted design.

Listed here (and shown in Figure 9.1) are three of the primary influences of contemporary SOA discussed so far in this book:

- first-generation Web services concepts (explained in Chapter 5)
- second-generation (WS-*) Web Services concepts (explained in Chapters 6 and 7)
- principles of service-orientation (explained in Chapter 8)

Figure 9.1. External influences that form and support contemporary SOA.



There are, of course, many other influences. Fundamental XML concepts, for example, have driven and shaped SOA and Web services on a fundamental level. For now, though, let's focus on these three.

As previously stated, to fulfill the potential of this architecture, we need to understand the origins of its characteristics. This understanding equips us with a healthy knowledge of what intrinsically drives fundamental aspects of SOA. More importantly, though, it allows us to pinpoint exactly which parts of contemporary SOA must be manually bolted on to this architecture. This allows us to focus on analysis and design efforts that ensure that these parts of SOA are properly implemented and their benefits realized.

9.1.1. Mapping the origins and supporting sources of concrete SOA characteristics

In Chapter 3 we created a list of concrete characteristics commonly associated with contemporary SOA. These qualities represent the current state and expectations surrounding this

architectural platform and are also a reflection of the technology being developed in its support.

We've reached a stage in this book where we've discussed each of the three contemporary SOA influences identified in the previous section as being either responsible for or that in some way relating to a number of these characteristics.

We want to ensure that we identify those characteristics not supported by these external influences so that we can discuss how they also can be realized. So let's take this opportunity to revisit our original list, with the intention of striking off the ones that already have been addressed.

Note

The WS-* specifications referenced in Table 9.1 are only those covered by this book. Additional specifications exist.

Table 9.1. A review of how contemporary SOA characteristics are influenced by Web services specifications and service-orientation principles.

Characteristic	Origin and/or Supporting Source
fundamentally autonomous	Autonomy is one of the core service-orientation principles that can be applied to numerous parts of SOA. Pursuing autonomy when building and assembling service logic supports other SOA characteristics.
based on open standards	This is a natural by-product of basing SOA on the Web services technology platform and its ever-growing collection of WS-* specifications. The majority of Web services specifications are open and vendor-neutral.
QoS capable	The quality of service improvements provided by contemporary SOA are, for the most part, realized via vendor implementations of individual WS-* extensions.
architecturally composable	While composability, on a service level, is one of our service-orientation principles, for an architecture to be considered composable requires that the technology from which the architecture is built support this notion. For the most part, the specifications that comprise the WS-* landscape fully enable architectural composability by allowing a given SOA to only implement extensions it actually requires.

vendor diversity	This is really more of a benefit of SOA than an actual characteristic. Regardless, it is primarily realized through the use of the open standards provided by the Web services platform.
intrinsic interoperability	The standardized communications framework provided by Web services establishes the potential to foster limitless interoperability between services. This is no big secret. To foster intrinsic interoperability among services, though, requires forethought and good design standards. Although supported by a number of WS-* specifications, this characteristic is not directly enabled by our identified influences.
discoverability	Service-level discoverability is one of our fundamental principles of service-orientation. Implementing discoverability on an SOA level typically requires the use of directory technologies, such as UDDI (one of the first-generation Web services specifications).
promotes federation	Federation is a state achieved by extending SOA into the realm of service-oriented integration. A number of key WS-* extensions provide feature-sets that support the attainment of federation. Most notable among these are the specifications that implement the concepts of orchestration and choreography.
inherent reusability	Reusability is one of the primary principles of service-orientation and one that can be applied across service-oriented solution environments. SOA promotes the creation of inherently reusable service logic within individual services and across service compositions—a benefit attainable through quality service design.

extensibility	Given that Web services are composable and based on open standards, extensibility is a natural benefit of this platform. Several WS-* extensions introduce architectural mechanisms that build extensibility into a solution. However, for the most part, this is a characteristic that must be intentionally designed into services individually and into SOA as a whole.
service-oriented business modeling	This key characteristic is supported by orchestration, although not automatically. WS-* specifications, such as WS-BPEL, provide a dialect capable of expressing business process logic in an operational syntax resulting in a process definition. Only through deliberate design, though, can these types of process definitions actually be utilized to support service-oriented business modeling.
layers of abstraction	Service-orientation principles fully promote black box-type abstraction on a service interface level. However, to coordinate logic abstraction into layers, services must be designed and organized according to specific design standards.
enterprise-wide loose coupling	Loose coupling is one of the fundamental characteristics of Web services. Achieving a loosely coupled enterprise is a benefit expected from the coordinated proliferation of SOA and abstraction layers throughout an organization's business and application domains.
organizational agility	Though the use of Web services, service-orientation principles, and WS-* specifications support the concept of increasing an organization's agility, they do not directly enable it. This important characteristic requires dedicated analysis and design and relies on the realization of other SOA characteristics.

9.1.2. Unsupported SOA characteristics

Having removed the concrete SOA characteristics that receive support from our identified external influences, we are now left with the following six:

- intrinsic interoperability
- extensibility
- enterprise-wide loose coupling

- service-oriented business modeling
- organizational agility
- layers of abstraction

The first two are somewhat enabled by different WS-* extensions. However, realizing these characteristics within SOA is a direct result of standardized, quality service design. The design guidelines provided in Chapter 15 provide recommendations for fostering these qualities. As a result, we'll take interoperability and extensibility off our list for now.

This leaves us with four remaining characteristics of contemporary SOA that are not directly supported or provided by the external influences we identified. These characteristics have been numbered here only to allow for easier referencing in later sections.

1. enterprise-wide loose coupling
2. support for service-oriented business modeling
3. organizational agility
4. layers of abstraction

What is most interesting about our brief study is that these four characteristics actually provide some of the most crucial benefits of this architecture. The caveat, though, is that they require a conscious effort for us to realize. This translates into extra up-front work that simply comes with the territory of building contemporary SOA.

Incorporating these key qualities into SOA requires that some very fundamental decisions be made, long before the building process of individual services actually can commence. The remaining sections in this chapter explore how structuring SOA

around the creation of specialized service layers directly determines the extent to which these characteristics can be manifested.

SUMMARY OF KEY POINTS

- The primary external influences that shape and affect contemporary SOA are first- and second-generation Web services specifications and the principles of service-orientation.
- Many of the characteristics that define contemporary SOA are, in fact, provided by these external influences.
- Those characteristics not directly supplied by these influences must be realized through dedicated modeling and design effort.
- These unique characteristics represent some of SOA's most important features and its broadest benefit potential.

9.2. Service layer abstraction

In our familiar enterprise model, the service interface layer is located between the business process and application layers. This is where service connectivity resides and is therefore the area of our enterprise wherein the characteristics of SOA are most prevalent. To implement the characteristics we just identified in an effective manner, some larger issues need to be addressed.

Specifically, we need to answer the following questions:

- What logic should be represented by services?
- How should services relate to existing application logic?
- How can services best represent business process logic?

- How can services be built and positioned to promote agility?

Typically, these questions are studied and eventually answered during the service-oriented analysis phase, where services are carefully modeled in accordance with and in response to external business and project drivers. Before we delve into specific recommendations on how to succeed at the art of service modeling (as explained in Chapters 11 and 12), let's first provide some preliminary answers to these questions.

9.2.1. Problems solved by layering services

What logic should be represented by services?

In the previous chapter we established that enterprise logic can be divided into two primary domains: business logic and application logic. Services can be modeled to represent either or both types of logic, as long as the principles of service-orientation can be applied.

However, to achieve enterprise-wide loose coupling (the first of our four outstanding SOA characteristics) physically separate layers of services are, in fact, required. When individual collections of services represent corporate business logic and technology-specific application logic, each domain of the enterprise is freed of direct dependencies on the other.

This allows the automated representation of business process logic to evolve independently from the technology-level application logic responsible for its execution. In other words, this establishes a loosely coupled relationship between business and application logic.

How should services relate to existing application logic?

Much of this depends on whether existing legacy application logic needs to be exposed via services or whether new logic is

being developed in support of services. Existing systems can impose any number of constraints, limitations, and environmental requirements that need to be taken into consideration during service design.

Applying a service layer on top of legacy application environments may even require that some service-orientation principles be compromised. This is less likely when building solutions from the ground up with service layers in mind, as this affords a level of control with which service-orientation can be directly incorporated into application logic.

Either way, services designed specifically to represent application logic should exist in a separate layer. We'll therefore simply refer to this group of services as belonging to the *application service layer*.

How can services best represent business logic?

Business logic is defined within an organization's business models and business processes. When modeling services to represent business logic, it is most important to ensure that the service representation of this logic is in alignment with existing business models.

It is also useful to separately categorize services that are designed in this manner. Therefore, we'll refer to services that have been modeled to represent business logic as belonging to the *business service layer*. By adding a business service layer, we also implement the second of our four SOA characteristics, which is support for service-oriented business modeling.

How can services be built and positioned to promote agility?

The key to building an agile SOA is in minimizing the dependencies each service has within its own processing logic.

Services that contain business rules are required to enforce and act upon these rules at runtime. This limits the service's ability to be utilized outside of environments that require these rules. Similarly, controller services that are embedded with the logic required to compose other services can develop dependencies on the composition structure.

Introducing a parent controller layer on top of more specialized service layers would allow us to establish a centralized location for business rules and composition logic related to the sequence in which services are executed. Orchestration is designed specifically for this purpose. It introduces the concept of a process service, capable of composing other services to complete a business process according to predefined workflow logic. Process services establish what we refer to as the *orchestration service layer*.

While the addition of an orchestration service layer significantly increases organizational agility (number three on our list of SOA characteristics), it is not alone in realizing this quality. All forms of organized service abstraction contribute to establishing an agile enterprise, which means that the creation of separate application, business, and orchestration layers collectively fulfill this characteristic.

Abstraction is the key

Though we addressed each of the preceding questions individually, the one common element to all of the answers also happens to be the last of our four outstanding SOA characteristics: layers of abstraction.

We have established how, by leveraging the concept of composition, we can build specialized layers of services. Each

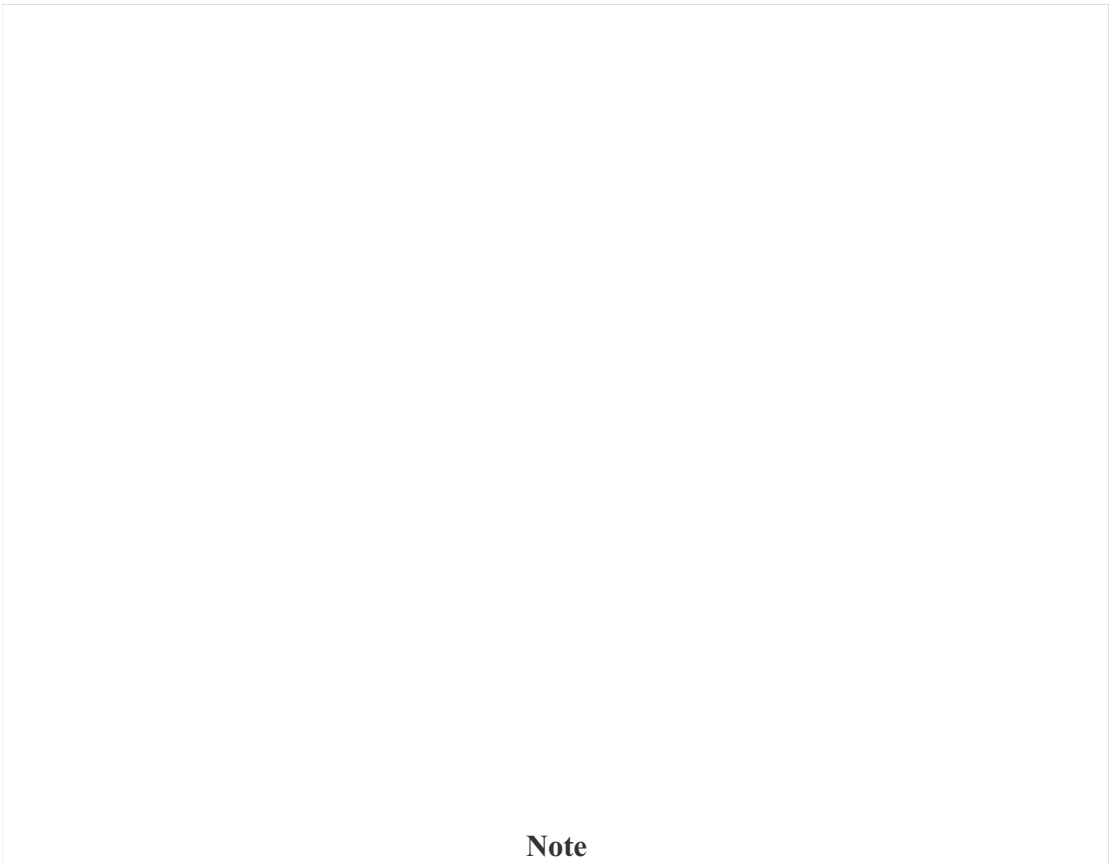
layer can abstract a specific aspect of our solution, addressing one of the issues we identified. This alleviates us from having to build services that accommodate business, application, and agility considerations all at once.

The three layers of abstraction we identified for SOA are:

- the application service layer
- the business service layer
- the orchestration service layer

Each of these layers (also shown in Figure 9.2) is introduced individually in the following sections.

Figure 9.2. The three primary service layers.



Note

The next three sections reference service models established in previous chapters and also introduce several new service models. Appendix B provides a reference table for all service models covered in this book, including information as to where individual models are discussed.

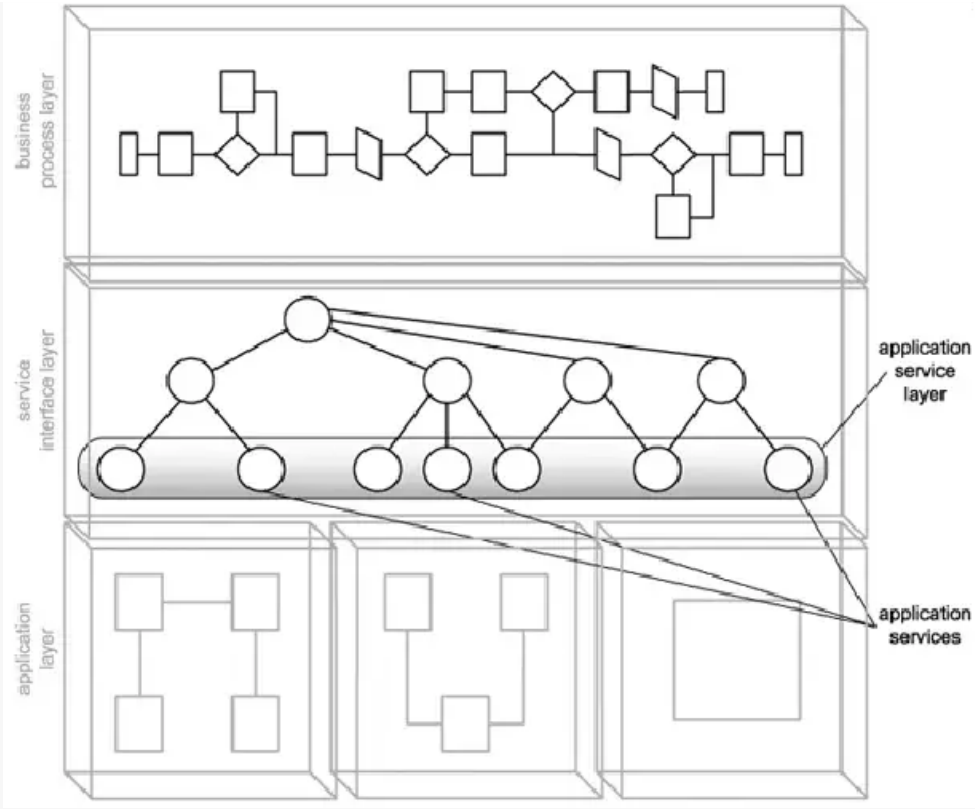
SUMMARY OF KEY POINTS

- Through abstraction implemented via distinct service layers, key contemporary SOA characteristics can be realized—most notably, increased organizational agility.
- The three common layers of SOA are the application service layer, the business service layer, and the orchestration service layer.

9.3. Application service layer

The application service layer establishes the ground level foundation that exists to express technology-specific functionality. Services that reside within this layer can be referred to simply as *application services* (Figure 9.3). Their purpose is to provide reusable functions related to processing data within new or legacy application environments.

Figure 9.3. The application service layer.



Application services commonly have the following characteristics:

- they expose functionality within a specific processing context
- they draw upon available resources within a given platform
- they are solution-agnostic
- they are generic and reusable
- they can be used to achieve point-to-point integration with other application services
- they are often inconsistent in terms of the interface granularity they expose
- they may consist of a mixture of custom-developed services and third-party services that have been purchased or leased

Typical examples of service models implemented as application services include the following:

- utility service (explained in Chapter 5)

- wrapper service (explained shortly)

When a separate business service layer exists (as explained in the *Business service layer* section), there is a strong motivation to turn all application services into generic utility services. This way they are implemented in a solution-agnostic manner, providing reusable operations that can be composed by business services to fulfill business-centric processing requirements.

Alternatively, if business logic does not reside in a separate layer, application services may be required to implement service models more associated with the business service layer. For example, a single application service also can be classified as a business service if it interacts directly with application logic and contains embedded business rules.

Services that contain both application and business logic can be referred to as *hybrid application services* or just *hybrid services*. This service model is commonly found within traditional distributed architectures. It is not a recommended design when building service abstraction layers. Because it is so common, though, it is discussed and referenced throughout this book.

Finally, an application service also can compose other, smaller-grained application services (such as proxy services) into a unit of coarse-grained application logic. Aggregating application services is frequently done to accommodate integration requirements. Application services that exist solely to enable integration between systems often are referred to as *application integration services* or simply *integration services*. Integration services often are implemented as controllers.

Because they are common residents of the application service layer, now is a good time to introduce the *wrapper service* model. Wrapper services most often are utilized for integration purposes. They consist of services that encapsulate ("wrap") some or all parts of a legacy environment to expose legacy functionality to service requestors. The most frequent form of wrapper service is a service adapter provided by legacy vendors. This type of out-of-the-box Web service simply establishes a vendor-defined service interface that expresses an underlying API to legacy logic.

Another variation of the wrapper service model not discussed in this book is the *proxy service*, also known as an *auto-generated WSDL*. This simply provides a WSDL definition that mirrors an existing component interface. It

establishes an endpoint on the component's behalf, essentially allowing it to participate in SOAP communication. The proxy service should not be confused with a *service proxy*, which is used by service requestors to contact service providers (as explained in Chapter 18).

Case Study

TLS has a well-defined application services layer. Of the TLS services we've discussed so far in our case study, the following are considered application services:

- Load Balancing Service
- Internal Policy Service
- System Notification Service

Each is a utility service that provides a set of generic, reusable features, and each is capable of acting as a composition member, fulfilling a specific task within a larger unit of automation.

All of the following RailCo services incorporate processing akin to application services:

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

Both the Invoice Submission and Order Fulfillment Services are somewhat hybrid, in that each also contains embedded business logic (as described further in the *Business service layer*example). The TLS Subscription Service can be classified as a pure application service, as it performs a simple, application-centric task. It's questionable whether any RailCo services would be considered utility services because none were designed with any real reusability in mind.

SUMMARY OF KEY POINTS

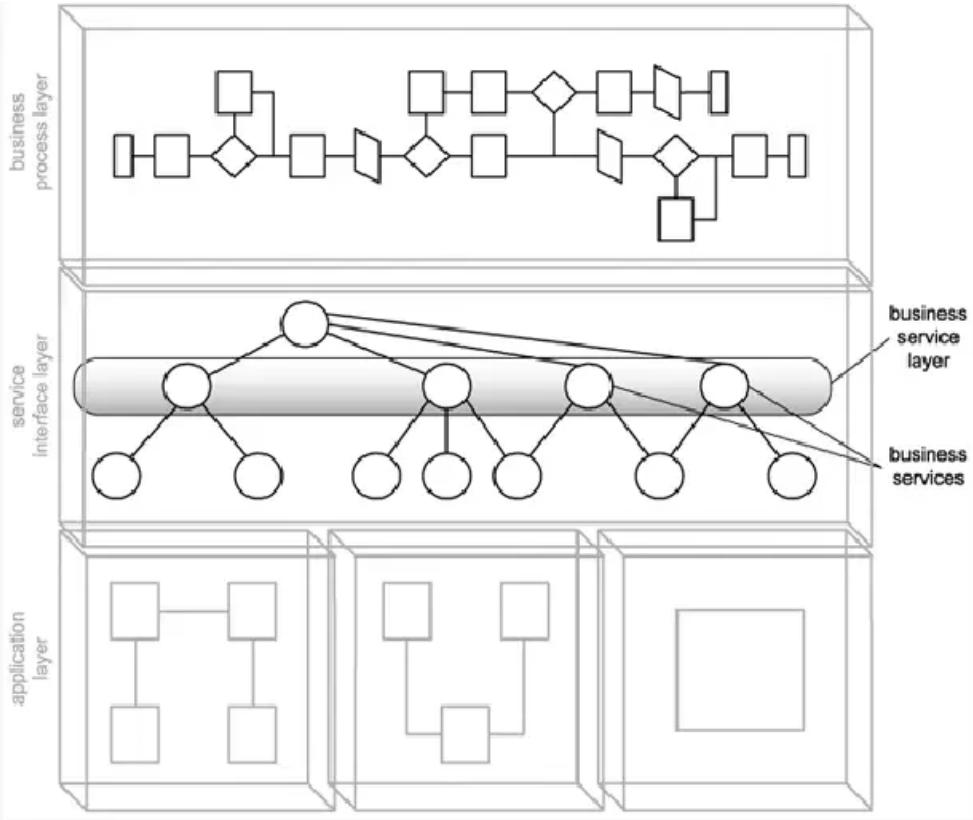
- The application service layer consists of application services that represent technology-specific logic.
- Typical incarnations of application services are the u tility and wrapper models.

• Application services are ideally reusable utility services composed by business services, but also can exist as hybrid services that contain both business and application logic.

9.4. Business service layer

While application services are responsible for representing technology and application logic, the business service layer introduces a service concerned solely with representing business logic, called the *business service* (Figure 9.4).

Figure 9.4. The business service layer.



Business services are the lifeblood of contemporary SOA. They are responsible for expressing business logic through service-orientation and bring the representation of corporate business models into the Web services arena.

Application services can fall into different types of service model categories because they simply represent a group of services that express technology-

specific functionality. Therefore, an application service can be a utility service, a wrapper service, or something else.

Business services, on the other hand, are always an implementation of the business service model. The sole purpose of business services intended for a separate business service layer is to represent business logic in the purest form possible. This does not, however, prevent them from implementing other service models. For example, a business service also can be classified as a controller service and a utility service.

In fact, when application logic is abstracted into a separate application service layer, it is more than likely that business services will act as controllers to compose available application services to execute their business logic.

Business service layer abstraction leads to the creation of two further business service models:

- *Task-centric business service*—A service that encapsulates business logic specific to a task or business process. This type of service generally is required when business process logic is not centralized as part of an orchestration layer. Task-centric business services have limited reuse potential.
- *Entity-centric business service*—A service that encapsulates a specific business entity (such as an invoice or timesheet). Entity-centric services are useful for creating highly reusable and business process-agnostic services that are composed by an orchestration layer or by a service layer consisting of task-centric business services (or both).

When a separate application service layer exists, these two types of business services can be positioned to compose application services to carry out their business logic. Task and entity-centric business services are explained in more detail in the *Deriving business services* section in Chapter 11.

Note that the hybrid service we introduced previously is actually a service that contains both business and application logic. It is therefore often referred to as a type of business service. For the purpose of establishing specialized service layers, we consider the business service layer reserved for services that abstract business logic only. We therefore classify the hybrid service as a variation of an application service, making it a resident of the application service layer.

Case Study

Of the TLS services we've discussed so far in our case study examples, the following are true business services:

- Accounts Payable Service
- Purchase Order Service
- Ledger Service
- Vendor Profile Service

Each represents a well-defined boundary of business logic, and whenever one of these service's operations needs to perform a task outside of this boundary, it reuses functionality provided in another business or application service.

As we mentioned earlier, RailCo's Invoice Submission and Order Fulfillment Services are hybrid in that they contain both business rules and application-related processing. This type of service is very common in organizations that only incorporate services peripherally. Either way, for modeling purposes, these two services are classified as hybrid application services. Because business logic is not explicitly abstracted, only an application services layer exists within RailCo.

SUMMARY OF KEY POINTS

- The business service layer is comprised of business services, a direct implementation of the business service model.
- Business services are ideally also controllers that compose application services to execute their business logic.
- Even though hybrid services contain business logic, they are not classified as business services.

9.5. Orchestration service layer

In Chapter 6 we introduced a concept that ties into and actually extends the reach of service-orientation. This concept is orchestration. When incorporated as part of a service-oriented architecture, orchestration assumes the role of the process part we established in the *Anatomy of a service-oriented architecture* section.

Orchestration is more valuable to us than a standard business process, as it allows us to directly link process logic to service interaction within our workflow logic. This combines business process modeling with service-oriented modeling and design. And, because orchestration languages (such as WS-BPEL) realize workflow management through a process service model, orchestration brings the business process into the service layer, positioning it as a master composition controller.

The orchestration service layer introduces a parent level of abstraction that alleviates the need for other services to manage interaction details required to ensure that service operations are executed in a specific sequence (Figure 9.5). Within the orchestration service layer, *process services* compose other services that provide specific sets of functions, independent of the business rules and scenario-specific logic required to execute a process instance.

Figure 9.5. The orchestration service layer.

These are the same process services for which we defined the process service model described in Chapter 6. Therefore, all process services are also controller services by their very nature, as they are required to compose other services to execute business process logic. Process services also have the potential of becoming utility services to an extent, if a process, in its entirety, should be considered reusable. In this case, a process service that enables orchestration can itself be orchestrated (making it part of a larger orchestration).

Also worth noting is that, as explained in Chapter 6, the introduction of an orchestration layer typically brings with it the requirement to introduce new middleware into the IT infrastructure. Orchestration servers are by no means a trivial addition and can impose significant expense and complexity.

Case Study

Neither TLS or RailCo employ process services or a separate orchestration layer. As with many organizations, TLS is interested in orchestration but is uncertain as to the technology that supports it and the impact it would have on their current service architecture. By the time we reach Chapter 16, TLS will have decided to give orchestration a try. As a result, our case study will progress in that chapter to the point where we build a WS-BPEL business process definition for TLS that establishes an official orchestration service layer.

SUMMARY OF KEY POINTS

- The orchestration service layer consists of one or more process services that compose business and application services according to business rules and business logic embedded within process definitions.
- Orchestration abstracts business rules and service execution sequence logic from other services, promoting agility and reusability.

9.6. Agnostic services

A key aspect of delivering reusable services is that they introduce service layers that are not limited to a single process or solution environment. It is important to highlight this one point, as it can blur the architectural boundary of a service-oriented solution.

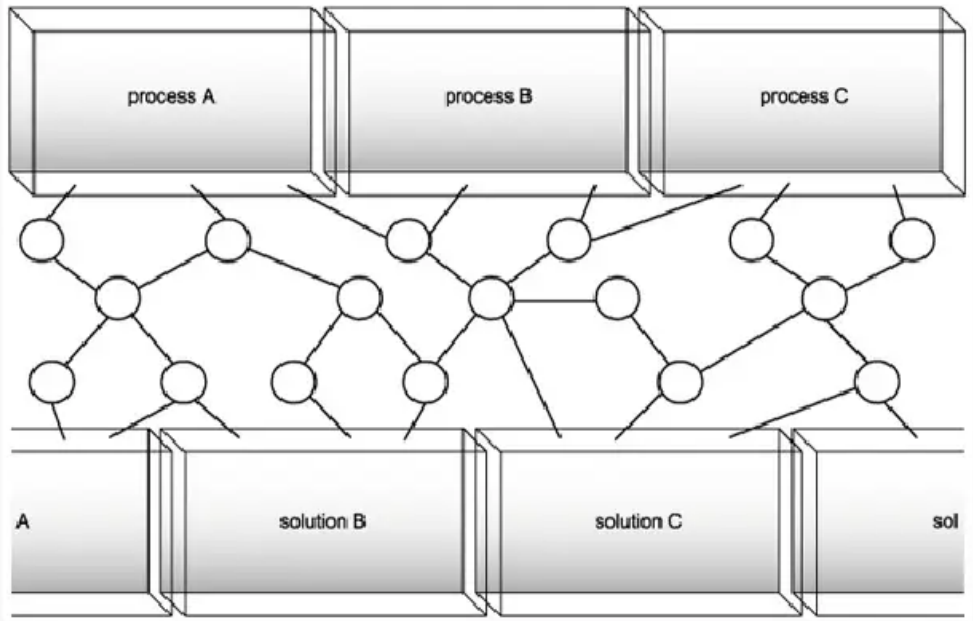
An application-level SOA containing solution-agnostic services does, in fact, extend beyond the application. And, in the same manner, an application-level SOA that depends on the use of existing solution-agnostic services also does not have a well defined application boundary.

To expand on this point, let's take another look at those services more prone to providing reusable logic.

- Entity-centric business services are designed to provide a set of features that provide data management related only to their corresponding entities. They are therefore business process-agnostic. The same entity-centric business services can (and should) be reused by different process or task-centric business services.
- Application services ideally are built according to the utility service model. This makes them highly generic, reusable, and very much solution-agnostic. Different service-oriented solutions can (and should) reuse the same application services.

As shown in Figure 9.6, services can be process- and solution-agnostic while still being used as part of a service layer that connects different processes and solutions.

Figure 9.6. Services uniting previously isolated business processes and solution environments.



If the services you are delivering collectively represent the logic of an entire solution, then the architectural scope is essentially that of an application-level SOA. However, if you are building services that only extend an existing solution (or are being deployed with immediate reuse in mind), then the architectural scope can vary.

An enterprise that invests heavily in agnostic services easily can end up with an environment in which a great deal of reuse is leveraged. This is the point at which building service-oriented solutions can become more of a modeling exercise and less of an actual development project.

SUMMARY OF KEY POINTS

- Solution-agnostic service layers relate to and tie together multiple business processes and automation solutions.
- These service layers promote reuse but also blur the architectural boundaries of individual solutions.

9.7. Service layer configuration scenarios

So far we've established a relatively clean service layer model for SOA, in which logic is clearly abstracted through three distinct layers that establish a

well-defined composition hierarchy. Unfortunately, real SOAs rarely resemble this state.

Many variations can exist, as the various types of services we've discussed so far can be combined to create different SOA configurations. We provide here some sample configuration scenarios and briefly discuss the characteristics of each.

Just to recap, we will explore scenarios based on the use of the following types of services:

- hybrid application services (services containing both business process and application logic)
- utility application services (services containing reusable application logic)
- task-centric business services (services containing business process logic)
- entity-centric business services (services containing entity business logic)
- process services (services representing the orchestration service layer)

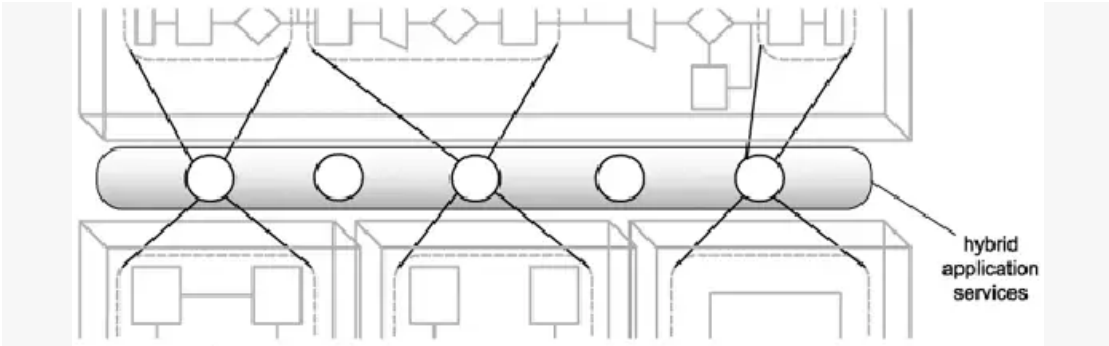
Note

In this section we qualify application services with the terms "hybrid" and "utility." However, in future chapters, utility application services simply are referred to as application services.

9.7.1. Scenario #1: Hybrid application services only

When Web services simply are appended to existing distributed application environments, or when a Web services-based solution is built without any emphasis on reuse or service-oriented business modeling, the resulting architecture tends to consist of a set of hybrid application services (Figure 9.7).

Figure 9.7. Hybrid services encapsulating both business and application logic.



9.7.2. Scenario #2: Hybrid and utility application services

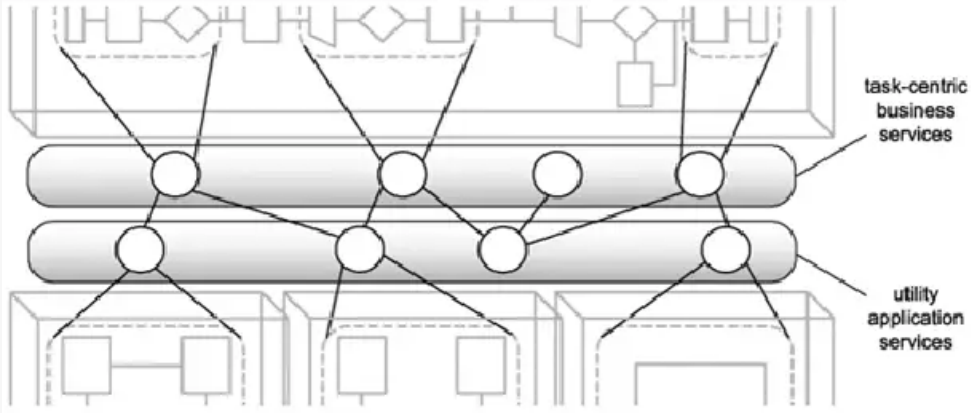
A variation of the previous configuration establishes a Web services-based architecture consisting of hybrid services and reusable application services. In this case, the hybrid services may compose some of the reusable application services. This configuration achieves an extent of abstraction, as the utility services establish a solution-agnostic application layer (Figure 9.8).

Figure 9.8. Hybrid services composing available utility application services.

9.7.3. Scenario #3: Task-centric business services and utility application services

This approach results in a more coordinated level of abstraction, given that business process logic is entirely represented by a layer of task-centric business services. These services rely on a layer of application services for the execution of all their business logic (Figure 9.9).

Figure 9.9. Task-centric business services and utility application services cleanly abstracting business and application logic.



9.7.4. Scenario #4: Task-centric business services, entity-centric business services, and utility application services

Here we've added a further layer of abstraction through the introduction of entity-centric business services. This positions task-centric services as parent controllers that may compose both entity-centric and application services to carry out business process logic (Figure 9.10).

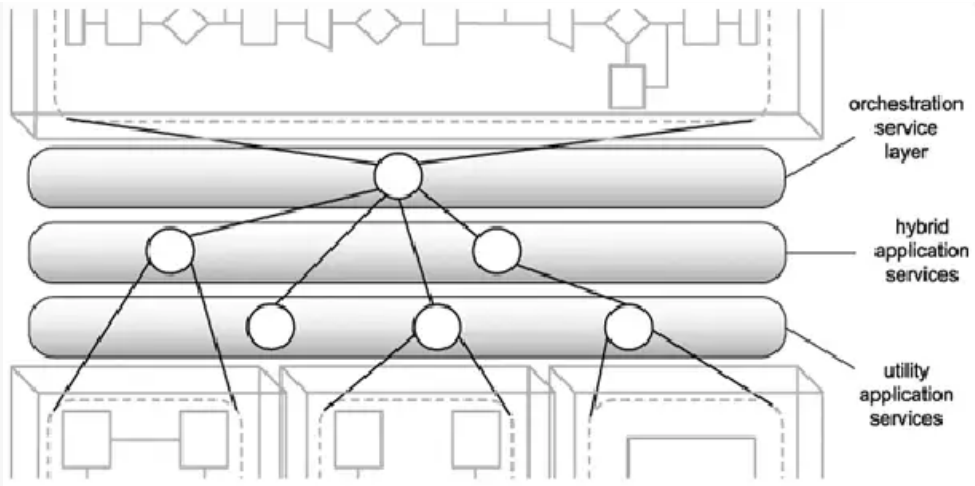
Figure 9.10. Two types of business services composing application services.

9.7.5. Scenario #5: Process services, hybrid application services, and utility application services

In this scenario, a parent process service composes available hybrid and application services to automate a business process. This is a common

configuration when adding an orchestration layer over the top of an older distributed computing architecture that uses Web services (Figure 9.11).

Figure 9.11. An orchestration layer providing a process service that composes different types of application services.

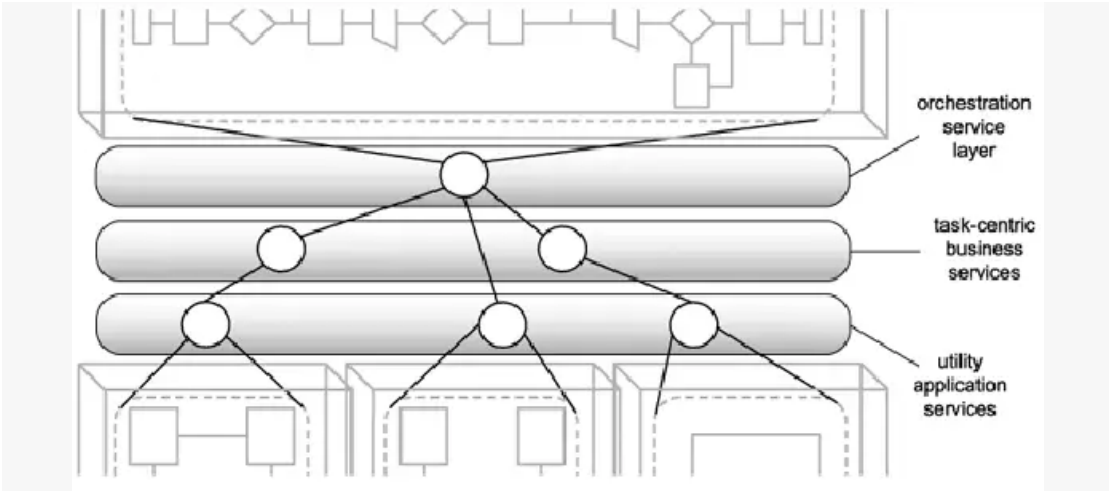


Note that although the hybrid service is being composed by the process service in Figure 9.11, the hybrid service still contains embedded business logic and therefore indirectly represents some of the business logic layer. Note also that the orchestration layer also may compose utility application services directly.

9.7.6. Scenario #6: Process services, task-centric business services, and utility application services

Even though task-centric services also contain business process logic, a parent process service can still be positioned to compose both task-centric and utility application services. Though only partial abstraction is achieved via task-centric services, when combined with the centralized business process logic represented by the process services, business logic as a whole is still abstracted (Figure 9.12).

Figure 9.12. A process service composing task-centric and utility application services.



9.7.7. Scenario #7: Process services, task-centric business services, entity-centric business services, and utility application services

This variation also introduces entity-centric business services, which can result in a configuration that actually consists of four service layers. Sub-processes can be composed by strategically positioned task-centric services, whereas the parent process is managed by the process service, which composes both task-centric and entity-centric services (Figure 9.13).

Figure 9.13. Process and business services composing utility application services.

9.7.8. Scenario #8: Process services, entity-centric business services, and utility application services

This SOA model establishes a clean separation of business and application logic, while maximizing reuse through the utilization of solution and business process-agnostic service layers. The process service contains all business process-specific logic and executes this logic through the involvement of available entity-centric business services, which, in turn, compose utility application logic to carry out the required tasks (Figure 9.14).

Figure 9.14. A process service composing entity-centric services, which compose utility application services.

SUMMARY OF KEY POINTS

- SOAs are configured in different shapes and sizes, depending primarily on the types of services from which they are comprised.
- Hybrid application services are found more commonly when service-oriented environments include legacy distributed application logic.
- By strategically positioning business and process services, an enterprise's business logic can be abstracted successfully from the underlying application logic.