

Chapter 4

A Methodology for Service Modeling and Design

When the programming model shifted from the traditional procedural model to that of object-orientation, a major paradigm shift occurred in the world of IT development. The focus was on encapsulating the state and behavior of entities and calling that encapsulation a class. Instances of a class were called objects, which occupied some space in the memory. Object orientation (OO) brought in concepts of inheritance, encapsulation, and polymorphism that could be applied to define relationships between classes. With the prevalence of the use of OO in the programming world, developers and architects started noticing some patterns that can be applied to the usage of OO principles to solve similar types of problems. The patterns depicted the deconstruction of a problem into multiple class entities, together with their interrelationships using the basic concepts of OO, to provide a solution to the problem. The seminal work in this field was done by the Gang of Four authors in the book called *Design Patterns: Elements of Reusable Object-Oriented Software*. (See the “References” section.) Whereas in OO the first-class constructs were objects and classes, the next-generation methodology for building software applications was called component-based development (CBD). In CBD, the first-class constructs were components, where a component was defined by its external specification, which could be used without any knowledge of its internal implementation. As such, the same external specification could be implemented in different programming language (for example, Java, C#). The internal implementation of a component may use multiple classes that collectively provide the implementation of the external specification. The classes could use one or more design patterns, thereby leveraging the advantages of OO principles.

In SOA, the main emphasis is on the identification of the right services followed by their specification and realization. Although some might argue that object-oriented analysis and design (OOAD) techniques can be used as a good starting point for services, its main emphasis is on microlevel abstractions. Services, on the other hand, are business-aligned entities and therefore are at a much higher level of abstraction than are objects and components.

The main first-class constructs in an SOA are *services*, *service components*, and *process flows*. For the sake of brevity, we refer to process flows as just flows. These are at a level of abstraction that is higher than that of objects, classes, and components. Hence, there needs to be a higher level of modeling and design principles that deal with the first-class constructs of an SOA. Service-oriented modeling and design is a discipline that provides prescriptive guidance about how to effectively design an SOA using services, service components, and flows. Rational Software, now a part of IBM, has provided an extension to Rational Unified Process (RUP) called *RUP-SOMA* (see the “References” section), which is built on a service-oriented analysis and design technique developed by IBM called Service Oriented Modeling and Architecture (SOMA). The rest of this chapter takes you through the SOMA technique and explains how it helps in the identification, specification, and realization of services, service components, and flows.

4.1 An SOA Reference Architecture



A.4.1

When defining a service-oriented solution, it makes sense to keep a reference architecture in context—an architecture that establishes the building blocks of SOA: *services*, *service components*, and *flows* that collectively support enterprise business processes and the business goals. The reference architecture provides characteristics and definitions for each layer and the relationships between them and assists in the placement of the architectural building blocks onto each layer. This layering facilitates the creation of architectural blueprints in SOA and helps in reusability of solutions and assets within an industry and potentially across industry verticals. Figure 4-1 shows a sample logical SOA reference architecture.

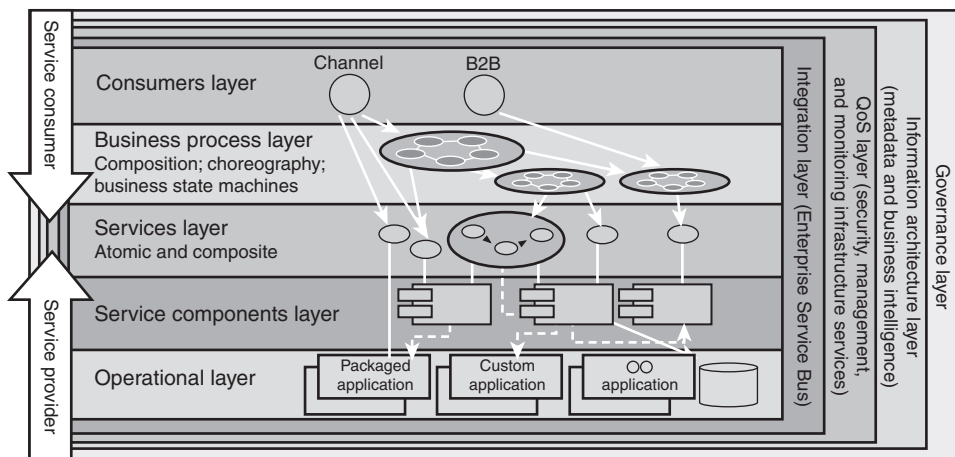


Figure 4-1 Logical view of SOA Reference Architecture

The figure shows a nine-layered architecture with five horizontal layers and four vertical layers. The horizontal layers follow the basic principle of a layered architecture model in which

architecture building blocks (ABB) from layers above can access ABBs from layers below, whereas layers below may not access ABBs from layers above. The vertical layers usually contain ABBs that are cross-cutting in nature, which implies that they may be applicable to and used by ABBs in one or more of the horizontal layers. This can also be called a partial layered architecture because any layer above does not need to strictly interact with elements from its immediate lower layer. For example, a specific access channel can directly access a service rather than needing to go through a business process. The access constraints, however, are dictated by the architectural style, guidelines, and principles that apply to a given SOA solution. This view of the SOA reference architecture is independent of any specific technology implementation, and hence is a logical view. Instances of this logical architecture can be developed for a specific platform and technology. Following are definitions of each of the layers:

- **Layer 1: Operational systems**—This layer includes the operational systems that exist in the current IT environment of the enterprise, supporting business activities. Operational systems include all custom applications, packaged applications, legacy systems, transaction-processing systems, and the various databases.
- **Layer 2: Service component layer**—Components in this layer conform to the contracts defined by services in the services layer. A service component may realize one or more services. A service component provides an implementation façade that aggregates functionality from multiple, possible disparate, operational systems while hiding the integration and access complexities from the service that is exposed to the consumer. The consumer thus is oblivious of the service component, which encapsulates the implementation complexities. The advantage of this façade component comes from the flexibility of changing operational systems without affecting the service definition. The service component provides an enforcement point for service realization to ensure quality of service (QoS) and compliance to service level agreements.
- **Layer 3: Services layer**—This layer include all the services defined in the enterprise service portfolio. The definition of each service, which constitutes both its syntactic and semantic information, is defined in this layer. Whereas the syntactic information is essentially around the operations on each service, the input and output messages, and the definition of the service faults, the semantic information is around the service policies, service management decisions, service access requirements, and so on. The services are defined in such a way that they are accessible to and invocable by channels and consumers independent of implementation and the transport protocol. The critical step is the identification of the services using the various techniques that can be employed for the same. The methodology that we focus on in this chapter addresses such identification techniques.
- **Layer 4: Business process layer**—Business processes depict how the business runs. A business process is an IT representation of the various activities coordinated and collaborated in an enterprise to perform a specific high-level business function. This layer represents the processes as an orchestration or a composition of loosely coupled services—leveraging the services represented in the services layer. The layer is also responsible for the entire lifecycle management of the processes along with

their orchestration, and choreography. The data and information flow between steps within each process is also represented in this layer. Processes represented in this layer are the connection medium between business requirements and their manifestation as IT-level solutions using ABBs from other horizontal and vertical layers in the architecture stack. Users, channels, and B2B partner systems in the consumer layer uses the business processes in this layer as one of the ways to invoke application functionality.

- **Layer 5: Consumer layer**—This layer depicts the various channels through which the IT functions are delivered. The channels can be in the form of different user types (for example, external and internal consumers who access application functionality through access mechanisms like B2B systems, portals, rich clients, and other forms). The goal of this layer is to standardize on the access protocol and data format to enable the quick creation of front ends to the business processes and services exposed from the layers below. Some such standards have emerged in the form of portlets, service component architecture (SCA) components, and Web Services for Remote Portlets (WSRP). The adherence to standard mechanisms for developing the presentation layer components for the business processes and services helps in providing template solutions in the form of standard architecture patterns, which helps the developer community to adopt common front-end patterns for service consumption.
- **Layer 6: Integration layer**—This layer provides the capability for service consumers to locate service providers and initiate service invocations. Through the three basic capabilities of mediation, routing, and data and protocol transformation, this layer helps foster a service ecosystem wherein services can communicate with each other while being a part of a business process. The key nonfunctional requirements such as security, latency, and quality of service between adjacent layers in the reference architecture are implemented by the architecture building blocks in this layer. The functions of this layer are typically and increasingly being collectively defined as the enterprise service bus (ESB). An ESB is a collection of architecture patterns that uses open standards and protocols to implement the three basic capabilities of this layer and provide a layer of indirection between the service consumers and the service provider by exposing the services only through the ESB. ESB products usually add some specialized features to provide differentiated capabilities in the marketplace.

The integration capabilities are most commonly used by ABBs residing between Layer 2 through Layer 5. As an example, in Layer 5 there can be many consumers accessing enterprise services through different channel types. Each channel type can use different protocols—HTML, WML (for mobile users), and Voice XML (for IVR users), to name a few. Each of these protocols and message formats may be passed through an Extensible Stylesheet Language Transformations (XSLT) engine before the actual service is invoked. This XSLT transform is usually an ESB-provided feature. The beauty of the ESB-based integration layer is that any feature or function that can be exposed in a manner that follows open standards and protocols for access can be plugged into the ESB so that it is enabled to take part in a service-based ecosystem. Figure 4-2 depicts a logical view of the ESB.

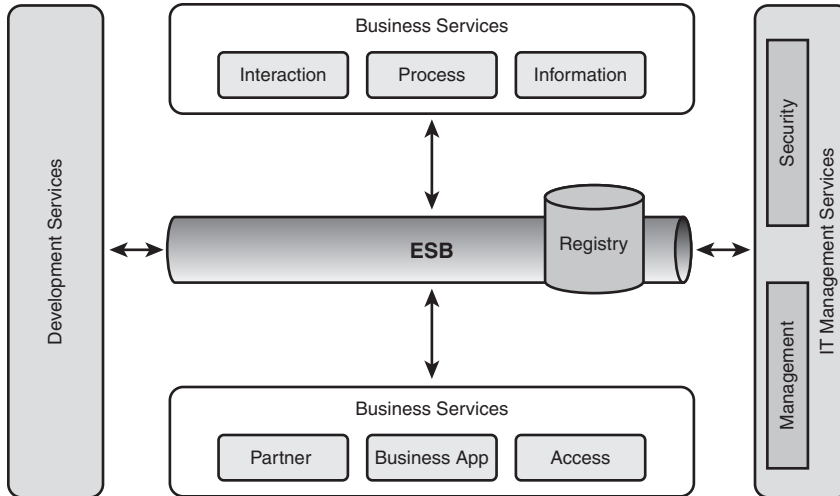


Figure 4-2 Logical view of an ESB in the integration layer

As Figure 4-2 suggests, the ESB provides capabilities for service consumers and providers to connect to each other, for services to be discovered using the registry, for services to be managed and for secure invocations, and provides application programming interfaces (API) to facilitate the development of service connectivity.

- **Layer 7: QoS layer**—This layer focuses on implementing and managing the non-functional requirements (NFR) that the services need to implement. Although SOA brings some real value proposition through the new architectural style, the programming model that supports the building of the first-class SOA constructs adds some inherent challenges that are nontrivial to address. The challenges arise while trying to comply with the essential tenets of SOA: abstraction, open standards and protocols, distributed computing, heterogeneous computing infrastructures, federated service ecosystem, and so on. Adherence to these compliance requirements often makes the implementation of the NFRs that much more complicated. This layer provides the infrastructure capabilities to realize the NFRs. It captures the data elements that provide the information around noncompliance to NFRs at each of the horizontal layers. Standard NFRs that it monitors for noncompliance is security, availability, scalability, and reliability.
- **Layer 8: Information architecture layer**—This layer ensures a proper representation of the data and information that is required in an SOA. The data architecture and the information architecture representation (along with its key considerations and guidelines for its design and usage) at each specific horizontal layer are the responsibilities of this layer.

Industry models (for example, ACORD, IAA, JXDD) and their usage to define the information architecture, along with business protocols used to exchange business

data, are addressed in this layer. It also stores the metadata required for data mining and business intelligence. Refer to the “References” section for the details of some industry models.

- **Layer 9: Governance layer**—This layer ensures the proper management of the entire lifecycle of the services. It is responsible for prioritizing which high-value services should be implemented, for each of the layers in the architecture, and for providing a rationalization based on how the service satisfies a business or IT goal of the enterprise. Enforcing both design-time and runtime policies that the services should implement and conform to is one of the key responsibilities of this layer. Essentially, this layer provides a framework that efficiently oversees the design and implementation of services so that they comply with the various business and IT regulatory policies and requirements.

Chapter 3, “SOA Governance,” discusses SOA governance and the responsibilities of a governance council in detail. Those responsibilities all feature in this layer of the reference architecture.

It is worth noting that one of the primary reasons for the SOA solution stack representation is that it helps to communicate, to the business and IT stakeholders, the evolution and realization of the enterprises SOA vision and roadmap through iterative implementation. Communicating with the stakeholders is key to ensure that the business commitment is pervasive across the various phases of an SOA initiative.

The methodology that we discuss in this chapter will help in identifying, specifying, and realizing the first-class constructs of an SOA and their placement in the various layers of the architecture stack. This logical view of the SOA reference architecture is also known as the SOA solution stack or just the solution stack. Therefore, the terms *SOA reference architecture*, *SOA solution stack*, and *solution stack* all refer to the same concept and hence can be used interchangeably.

4.2 Service Oriented Modeling and Architecture



A.4.2

Service Oriented Modeling and Architecture (SOMA) is a modeling and design technique developed by IBM that provides prescriptive steps for how to enable target business processes by defining and developing a service-based IT solution. SOMA provides the communication link between the business requirements and the IT solution. It provides guidance on how to use business model and information as inputs to derive and define a service-based IT model. SOMA, as a methodology, addresses the gap between SOA and object orientation. This methodology approach provides modeling, analysis, design techniques, and activities to define the foundations of an SOA. It helps defining the elements in each of the SOA layers (see Figure 4-2) and also to take architectural decisions at each level.

At the heart of SOMA is the identification and specification of services, components, and process flows. At a high level, SOMA is a three-phased approach to *identify*, *specify*, and *realize* services, components, and flows (typically, choreography of services). The first phase is that of service identification, where various techniques are used to identify an exhaustive

list of candidate services. The second phase is that of service specification, in which a detailed design of services and components is completed. The realization phase focuses on making architectural decisions, justifying the most prudent approach to implement the services.

SOMA focuses directly on the services, service components, and flows. These SOA constructs reside between Layer 2 and Layer 4 of the architecture stack. However, the activities performed as a part of the end-to-end methodology influence the placement of components in the other layers of the stack. Figure 4-3 illustrates the focus of SOMA as it pertains to the solution stack.

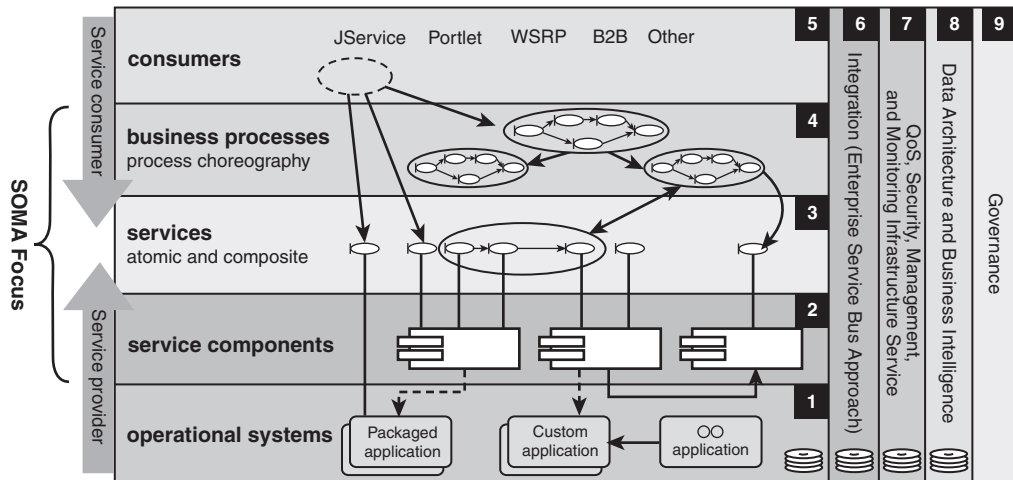


Figure 4-3 The focus of SOMA is on Layer 2 through Layer 4.

One of the main outputs of the SOMA method is a service model. It is recommended that a service model constitute of the following artifacts about services:

- **Service portfolio**—List of all the enterprise services
- **Service hierarchy**—A categorization of services into service groups
- **Service exposure**—An analysis and rationalization of which services should be exposed and which should not
- **Service dependencies**—Representing the dependencies of services on other services
- **Service composition**—How services take part in compositions to realize business process flows
- **Service NFRs**—The nonfunctional requirements that a service must comply with
- **State management**—The various types of states that a service should maintain and implement
- **Realization decisions**—Architectural decisions, for each service, around the most justified mechanism to implement the service

The techniques employed in the various phases of the method address one or more parts of the service model. Although there may be other work products that can be developed by exercising the various phases and activities of this method, we concentrate on the service model in this chapter.

4.2.1 Validation of Inputs

Before we can start identifying, specifying, and realizing service, it is imperative to validate the inputs that the method expects. SOMA takes inputs from the business.

The “to-be” business processes—their definitions and their corresponding process models that are decomposed down to second and third levels—are mandatory inputs. Services that will be summoned for implementation will be used in an orchestration to choreograph business process as a network of collaborating services at runtime.

Acknowledging the fact that SOA is primarily a business initiative where we strive to achieve flexibility, responsiveness, and agility in the business, the emphasis is on using SOA principles to solve business problems by designing and implementing an IT solution aligned with the business goals and strategy. The business goals and drivers of the company, the realization of which is the basis for commissioning an IT project, is a very important input into the method. The business goals need to be supplemented with a mechanism to measure the success of achieving the goal. Key Performance Indicator (KPI) is a metric that provides the business a measure of success of a software service against the attainment criteria for a business goal. Hence, business goals, drivers, and their associated KPIs are very important inputs to the method. These KPIs are used to measure how effective and successful an enterprise SOA initiative has been.

SOA strongly recommends the concept of reuse. Therefore, the traditional and often scary “rip and replace” approach to systems development is the last option in the philosophical premise of SOA. The idea is to reuse as much of the functionality available in currently running enterprise systems as possible. A good and solid understanding of the current IT environment represents a vital input to the method. For instance, the applications and systems, the functionalities they provide, the importance and usage of the functionalities provided, and the roadmap for enhancement or sunset of each of the systems are all key inputs that help in gaining a good understanding of the current IT portfolio.

The current organizational design and the future organization scope and requirements can also prove to be invaluable inputs. These can be used to identify which line of business will be responsible for the ownership and funding of the service lifecycle. However, this is not a mandatory input and can be categorized as “nice to have information.”

We recommend the execution of a method-adoption workshop for any client engagement. Such a workshop allows the consulting team to customize the method to suit the specific requirements of the client. In an SOA-based engagement, and as a part of this method-adoption workshop, one can determine the specific inputs available for use with the SOMA method. The inputs that are available must be carefully validated for completeness. So, what happens if they are incomplete?

The first thing to do is to assess the gaps between the required and the available information, and then a gap-mitigation plan needs to be put in place. A part of the recommended approach is to perform further interview sessions with the stakeholders and subject matter experts (SME) and use that information gathered therein to incorporate the missing information. We also recommend documenting customer pain points and use them to define the customer requirements, business drivers, and priorities. These are by no means the only two ways to address gaps between available and required inputs, and the IT team might have its own gap-mitigation plan as it suits the current customer scenario.

If the mandatory inputs are not available, however, a commitment needs to be made by the business stakeholders to make them available to the degree of completeness as requested by the IT team.

4.2.2 Identification

When the validation of inputs has been completed, the focus shifts to the identification of the services that will ultimately constitute the service portfolio. The aim is to get an exhaustive list of services that are potential candidates for exposure and then categorize the services into some logical grouping. Experience suggests that the general approaches taken to identify services are sometimes too restrictive; typically we do not exploit all possible sources to identify enterprise-level services. To come up with such an exhaustive list of “candidate” services, it is recommended to use a combination of three complementary techniques: domain decomposition, existing asset analysis, and goal service modeling. After we have compiled the list of candidate services, we use a technique that extracts, from the list of candidate services, only those services relevant for exposure. Figure 4-4 illustrates the three techniques for service identification.

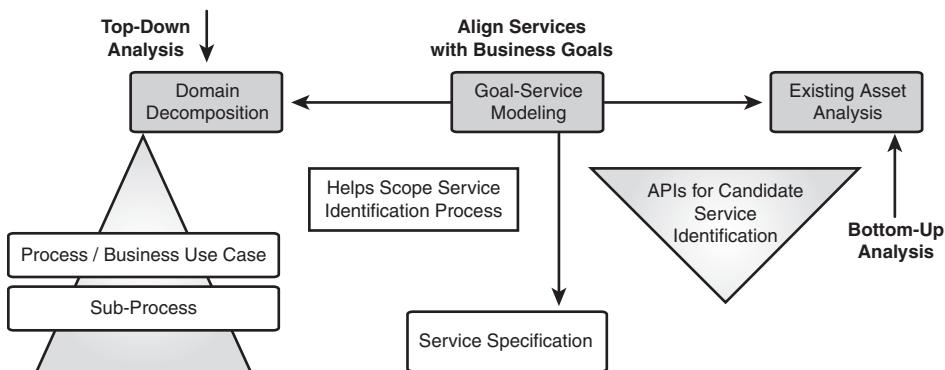


Figure 4-4 The three different techniques for service identification

Let's explore further the various techniques for service identification.

4.2.1.1 Domain Decomposition

This is a top-down technique that involves the decomposition of the business domain into its functional areas and subsystems, including the decomposition of its business processes into subprocesses and high-level business use cases. These use cases are often good candidates for business services exposed at the edge of the enterprise, or for those used within the boundaries of the enterprise across lines of business. Apart from identifying candidate services, this technique helps to identify functional areas that identify boundaries for subsystems.

This technique includes one step called functional area analysis (FAA). In FAA, we decompose the business domains into logical cohesive functional units and name each unit as a functional area. The resultant set of functional areas provides a modular view of the business and forms the basis of IT subsystem identification, nomenclature, and design. It is not necessary for FAA to be done as a part of SOMA because it can leverage similar work that could have been done as a part of any other project initiative in the same enterprise. The identification of functional areas assists in their usage in service categorization, wherein the identified candidate services can be categorized using the functional areas. The “Service Hierarchy” section of the service model work product is a formalization of the categorization of services.

FAA usually falls under the expertise realm of business analysts and domain experts. One can start with a base set of functional areas, but if and when services are identified and start to be grouped using the functional areas, one can refactor the existing functional areas so that they make sense from a service grouping standpoint. The key point we are trying to make here is that functional areas can be refactored to suit the proper grouping to services.

The next step in this technique is called process decomposition. In process decomposition, we decompose business processes into its constituent subprocesses and further into more atomic activities or tasks. The resultant process model depicts both the business-level and IT-level flow of events that realize a business process. It also forms the basis of candidate service identification. A process is a group of logically related activities that use the resources of the organization to provide defined results in support of the organization’s objectives. Process models describe the work that an organization is involved in and the behavior of systems the organization uses. Each business process in the scope of the business or IT transformation is decomposed into subprocesses and further into leaf-level subprocesses. Each activity in the resultant process model or process breakdown tree is considered a candidate for service exposure. Hence, each is added to a list called the service portfolio. At this point, the service portfolio consists of all the subprocesses, activities, and tasks from the process model definitions for every single process. The “Service Portfolio” section of the service model work product is the recipient of the list of candidate services that are identified in this step.

Decomposition of processes into its activities and tasks also assists in identifying commonalities and variations between multiple business processes. The common activities or subprocesses provide good candidates for services while the points of variability enable the design of the system in a way that it fosters design resiliency and makes the system more adaptive to incorporate future changes. Variations in a system are usually identified across

three aspects: structures, processes, and rules. Externalizing these variability points enables configurable injection of flexibility into system design. Variations may also suggest new services based on types, processes, and rules.

4.2.1.2 Existing Asset Analysis

Existing asset analysis is a bottom-up approach in which we examine assets such as existing custom applications, packaged applications and industry models to determine what can be leveraged to realize service functionality. This analysis is also designed to uncover any services that may have been missed through process decomposition. While you are analyzing existing legacy and custom applications, we recommend performing a coarse-grained mapping in which you map business functionality in the portfolio of existing applications to the business processes and determine which step (as identified through domain decomposition in Section 4.2.1.1) in the process can be potentially realized by some functionality in existing applications. We do not recommend performing a fine-grained mapping to specific transactions and batch processes within legacy application at this stage.

During the coarse-grained mapping activity, a detailed understanding of the application's state and quality is obtained that will allow the assessment of technical risks associated with the services that are going to be realized by the existing system functionality. For the applications that have such technical risks associated with their usage for service implementation, we recommend scoping some technical prototypes to test things like basic connectivity, protocol issues, data structures and formats, and so on. This prototyping will help mitigate the project risks that might otherwise crop up during the later stages, for example, during implementation.

So, with this technique, we can not only start thinking about service realizations using existing assets but also identify new services. These new services will be added to the service portfolio. At this point, the service portfolio consists of potential services derived from both a top-down and a bottom-up approach.

4.2.1.3 Goal Service Modeling

Goal service modeling (GSM) is the third of the three techniques and is used to validate and unearth other services not captured by either top-down or bottom-up service identification approaches. It ensures that key services have not been missed. GSM provides the key link between the business goals and IT through the traceability of services directly to a business goal. The attainment of the goal, through the supporting service, is measured through the KPIs and its metrics that were documented as a part of the inputs from the business. GSM also ensures that stakeholder involvement and accountability is maintained through their consent on the business goals that needs to be achieved. Services directly linked to the business goals would then have a higher probability of being prioritized and funded for subsequent design and implementation. It is worthwhile to point out that GSM may be used as a scoping mechanism that assists in defining the scope of a project by focusing deeper into the problem domain. A problem domain is often too large to be tackled in one iteration and hence narrowing down and identifying an area that provides the highest impact (by realizing one or more business goals) to the business in a reasonable and acceptable timeframe is

a recommended way of scoping a project initiative. Once the scope is defined around a business goal, not only can services be identified through the GSM technique but also the top-down (domain decomposition) and bottom-up (existing asset analysis) techniques may be performed on the given scope of the project.

Identifying business goals is a nontrivial task. It is not uncommon for clients to be grappling for ways to articulate their real business goals. SOA architects are not the ideal consultants who can be of much help to the business people. The business analysts and SMEs are the ones who come to the rescue, helping the clients to clearly articulate their business goals.

The business goals are usually stated in a way that are too lofty and at a very high level. It is difficult and often impossible to try to identify and associate a service to these lofty goals. The recommended approach is to work closely with the business and domain SMEs to decompose the goals into subgoals, and keep decomposing until the point that a subgoal is actionable. *Actionable* here means the attainment of what I call as the “Aha!” factor—that I can identify an IT function that I can use to realize this subgoal. Hence, each business goal is usually decomposed into subgoals, and then services are identified that can realize them. This approach differs radically from the top-down and bottom-up techniques, and therefore you have a high potential of discovering new services. These new services are added back to the service portfolio. Some of the discovered services can be found to be already present in the existing service portfolio. This is a good thing, a validation step that ascertains that more than one technique for service identification has identified the same service!

So, what do we achieve in the service identification phase?

- We have taken a three-pronged approach to identify candidate services.
- Each identified candidate service is added to the service portfolio.
- FAA is performed or leveraged to provide a mechanism to group services—the service hierarchy.
- The service grouping may be iteratively refactored to provide the best categorization of services.
- For functionality in existing applications identified for use to realize service implementations, a technical assessment is performed to assess the viability of reusing the existing application for service implementation.

From a service model standpoint, what have we addressed?

- We are able to provide a service portfolio of candidate services.
- We categorized the services into a service hierarchy or grouping.

With this, we move on to the second phase in the method: specification of services.

4.2.3 Specification

The specification phase helps design the details of the three first-class constructs of SOA: services, service components, and flows. It uses a combination of three high-level activities to determine which services to expose, provides a detailed specification for the exposed services, and specifies the flows (processes) and service components. The three activities are

called service specification, subsystem analysis, and component specification. From a service model work product standpoint, this phase provides the most content: The service exposure, service dependencies, service composition, service NFRs, service messages, and state management are all addressed in this phase. The rest of this section focuses on the three activities.

4.2.3.1 Service Specification

Service specification defines the dependencies, composition, exposure decisions, messages, QoS constraints, and decisions regarding the management of state within a service.

The first task concerns service exposure. The service portfolio had an exhaustive list of services obtained through the three techniques that we used for service identification. It is easy to comprehend that this list may contain too many candidate services; not all of them are at the right level of granularity to be exposed as services. Some of the service candidates may be too coarse grained and might actually be more like business processes or subprocesses rather than individual services (for example, some of the process elements derived from the first level of process decomposition), whereas some others may be too fine-grained IT functions (for example, the process elements in the lowest level of process decomposition and some of the existing system functionality). Deciding to expose the entire list of candidate services is a perfect recipe for following a perfect antipattern in SOA—the service proliferation syndrome (a phenomenon we want to avoid). Some economic and practical considerations limit the exposure of all candidate services. A cost is associated with every service chosen for exposure. The funding of the entire service lifecycle, the governance factor around service lifecycle management, and the added underlying infrastructure requirements to support security, scalability, performance, and other nonfunctional requirements make it impractical to follow the rules of economies of scale when it comes to exposing all candidate services.

Based on these premises, we recommend a service litmus test. The test consists of specific criteria applied to the candidate services. Only those services that meet the criteria are chosen for service exposure. The method provides an initial set of test criteria in the following form:

- 1. Business alignment**—A service must be business aligned. If a service is not, in some shape or form, traceable back to a business goal, it may not be an ideal candidate to be chosen for exposure.
- 2. Composability**—Tests the ability of a service to be used in a context entirely different from the one from which the service was originally identified. A service should be able to participate in multiple business processes without compromising the NFR compliance requirements for the process.
- 3. Feasibility of implementation**—Tests the technical feasibility of implementing the service in a cost- and time-effective manner. Practical considerations limit overly complex services to be commissioned for implementation.
- 4. Redundancy elimination**—Tests whether the service can be used within all processes and applications where its function is required.

This method by no means enforces these four litmus tests and recommends defining litmus test criteria taking the client environment, goals, and other relevant and pertinent client-specific factors into account. The most important point here is that some form of an elimination criterion needs to be defined that will allow the prioritization of services for exposure consideration. It is also recommended to provide a justification for the services that failed the litmus test for exposure. This justification, when documented, provides crucial information which becomes vital when the failed services might be revisited later in the scope of another SOA initiative in the same enterprise. It is quite possible that the business goals, client prerogatives, and other factors as applicable during subsequent projects might expose a service that might not have passed the exposure tests during the current scope!

Now that a filtered list of services has been determined, the services chosen for exposure constitute the refined service portfolio. Each service in this portfolio now needs to be provided with detailed specification. The first specification activity is to identify service dependencies.

Services are ideally dependent on other exposed services. Although in an ideal world of SOA everything is a service, in reality we find that services are more frequently dependent on underlying IT components. This dependency happens typically in situations where QoS requirements such as performance and availability tend to push SOA architects to design a service to be hardwired to one more technology-specific component. A service-to-service dependency is called a processing dependency because a service is dependent on one or more services only in the context of a given business process. Sometimes however, strict nonfunctional requirements mandate that services are more tightly coupled in their dependency on finer grained IT components. This type of dependency is often categorized as a functional dependency. Categorizing service dependencies into processing and functional groupings provides key architectural and design considerations for service implementation. Figure 4-5 provides an example of the two types of dependencies.

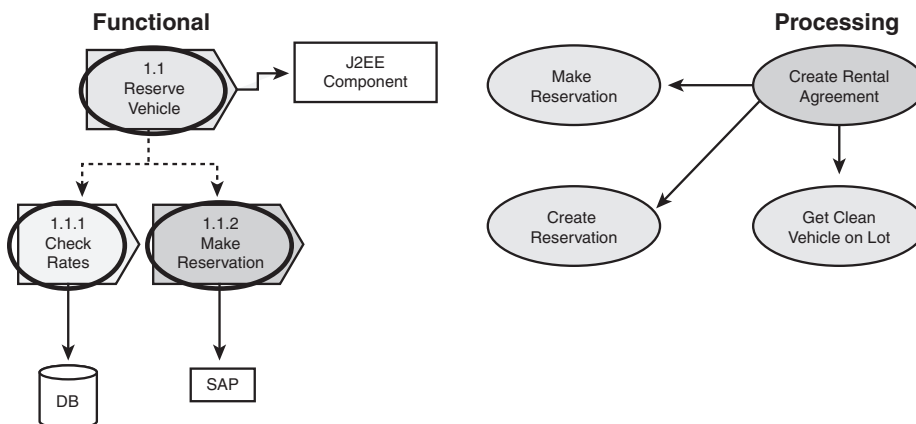


Figure 4-5 The two general types of service dependencies

With service dependencies depicted, the next natural activity is to identify service composition and flows. Services that take part in a composition are a collection of related services to solve a specific high-level business function. A business process can be either represented as a single composite service or may be realized as an orchestration of one or more composites or individual services. Each business process that is in the scope of the transformation initiative is modeled as such an orchestration. These orchestrated services and their specifications and design will influence how they may be implemented as Business Process Execution Language (BPEL) flows at runtime. See the “References” section for more information on BPEL.

The next activity is identification of the NFRs that a service must implement. Each service must, in addition to the business logic that it implements, comply with a set of NFRs. The security mandates for service access, for example, authentication requirements for external consumers or no security for internal consumers; the availability of the service, whether 99.999 or 99.9; the maximum allowable latency for service-invocation turnaround, whether 1 millisecond or 1 minute are examples of NFRs that typically must be implemented by a given service. The method prescribes the documentation of all the required and optional NFRs for each service. This information will influence downstream microdesign and implementation. Note that keeping the complexity of business logic equal, the complexity of the NFRs of a service directly affects the time, cost, and resource requirements for its implementation.

Service message and its specification is one of the most critical and significant activities in this phase. A service message—the input message, the output message, and the exception and faults—typically constitutes the syntactic specification of the service. Service messages are usually defined in XML format for the obvious reasons of portability and interoperability. This method provides some prescriptive guidance for designing service messages.

One of the main tenets of SOA is to provide business flexibility and agility to an enterprise through an IT infrastructure that facilitates the enterprise to participate in a collaborative ecosystem. Collaboration brings in the key requirement for flexible and seamless integration with other collaborating entities. One of the first things you want to do is to standardize on the message format used to define services. Following a standard message format can facilitate a better integration with other partners outside the enterprise perimeter. A growing number of industry-specific consortiums provide standard definitions for business entities and information applicable to a given industry. For example, the insurance industry and the retail industry might define a *customer business entity* differently. The attributes and even some base and common operations on the entities are being standardized per industry. These standard specifications are called industry models. There exist quite a few stable industry models, such as ACORD for insurance, enhanced Telecommunications Operations Map (eTOM) for electronics, Justice XML Data Dictionary (JXDD) for the Department of Justice, Open Travel Alliance (OTA) for travel and transportation, and so on. Refer to the “References” section for more information on eTOM and OTA.

This method recommends using the industry model, if available for the given industry, as a starting point for message specification. Acknowledging that these specifications are often all encompassing, the first level of analysis that needs to be done is to define a subset of the

industry model artifacts that is applicable to the client wherein the SOA project is being undertaken. This subset of the industry model can be the base specifications. In more cases than not, there will be a need to add some specific extensions to the base specifications that incorporates client-specific requirements. The base specifications together with the extensions constitute what we call the Enterprise Message Format (EMF). Defining the EMF is the first step toward service interoperability. Sometimes, a version or a flavor of an EMF may already be present with the client. If so, it needs to be analyzed, validated, and enhanced to support the new and upcoming requirements. The input and output message elements must be compliant with the EMF. The EMF is also a perfect starting point to define the Enterprise Information Model (EIM) and it also influences the Logical Data Model (LDM), both of which, although are not necessarily a part of SOA, are mandatory architectural constructs in any enterprise application development.

NOTE

Note that the domain of information architecture plays a very important role in SOA. The data translation requirements together with information architecture that models the data and information flow from the consumer, through all the layers of the architecture stack right down to the operational systems layer falls under the domain of the Integration and Data architecture layers in the SOA reference architecture.

The amount of work that goes into the definition of the EMF and subsequently into service message specifications is often underestimated and becomes the widest chasm to bridge. Keep in mind that service message specification is closely, if not tightly, linked with the design of the information model and the data models and therefore not a trivial work effort.

Get your EMF well designed and documented; it will have a positive impact on downstream design and specification activities.

The last major activity focuses on analysis of the state management requirements for a service and its operations. As a general design rule, the business logic implemented by a service should not include state-specific logic.

However, requirements often mandate that some services address some state requirements. The most commonly occurring types of state are transactional state, functional state, and security state. Transaction state is required to support transactions spanning multiple messages. If an atomic transaction must include the results of multiple business actions performed as a result of multiple messages from a service requestor to a service provider, the state of the transaction must be maintained until all the messages involved in the transaction are completed and the transaction is committed or rolled back.

Security state addresses how the identity of a consumer may be verified. In a stateless scenario, the client is authenticated each time a message is received. In a stateful scenario, a token is typically passed in the message sent by the consumer.

Functional state refers to state that must be maintained between messages while a business action is accomplished.

We must account for how the specific state requirements must be managed. Sometimes, IT technology components influence how state can be managed. For example, a security state can be managed by a product such as IBM Tivoli Access Manager (see the “References” section of Chapter 6 for more information), and the transactional state requirements between multiple service invocations in a business process may be managed by a BPEL engine. Hence, the documentation of the specific state requirements for each service is imperative. Keep in mind that during service realization, we can come up with the right architectural decisions justifying the best mechanism to implement the state management requirements. So, document them here!

This is just the first major activity during the specification phase; we have two more areas to address. Before we move on, however, we want to mention that all six recommended activities that we discussed as a part of this technique are iterative in nature and we will, depending on the scope and complexity of the project, need to run through this technique multiple times until we get a refined and robust specification of services at this level.

4.2.3.2 Subsystem Analysis

Just like functional areas provide a logical grouping of a business domain, an IT subsystem is a semantically meaningful grouping of logically cohesive IT artifacts, which are in the form of components and classes. When a functional area is too large to grasp, it is broken down into these logical units called subsystems. Although this decomposition can be done top down or bottom up, the method recommends a top-down approach.

A subsystem consists of three types of components: service components, functional components, and technical components. It is the job of the SOA architect to identify a logical grouping of services that can be implemented together by a team that has specific domain knowledge in the area. Subsystem analysis and identification is nothing special to SOA, and it has been practiced from the days of OO; therefore, I call it “architecture as usual” (AAU) work. Let’s focus now on the constituents of a subsystem and how to identify them.

A functional component is an IT component that encapsulates and supplies a single type of business functionality. For example, any customer-related business logic and IT APIs can be encapsulated in a single functional component called, for example, CustomerManager.

A technical component is an IT component that provides generic functionality such as authentication, error handling, and auditing. Their implementation is usually more closely tied with the technology platform that is used.

A service component is an IT component built as a coarse-grained façade on top of more focused and finer-grained functional and technical components. Think of it as a central component in a mediator pattern. A service is usually aligned to a high-level business function. For this business function to be implemented, it might need to call finer-grained IT APIs on some functional and technical components. Let’s consider an example in context. Suppose a service is to provide “details of the last reserved vehicle for a customer.” This requirement will typically necessitate a call to a CustomerManager component to “retrieve the customer profile” information, use some business logic to pick the relevant customer details from the returned result, and then invoke a VehicleManager component to “retrieve

the current reserved vehicle” for the given customer. In the process, it may invoke a technical component called AuditManager to log the service request. Neither the CustomerManager nor the VehicleManager nor the AuditManager has the business logic to control this microflow of steps. This control of the microflow, which component to invoke and in which sequence to realize the service request, is the responsibility of the service component. The service component can be designed to conform to the specifications of service component architecture (SCA). See Chapter 6, “Realization of Services,” for detailed treatment of SCA.

For illustrative purposes only, Figure 4-6 depicts how subsystems are related to service, functional, and technical components.

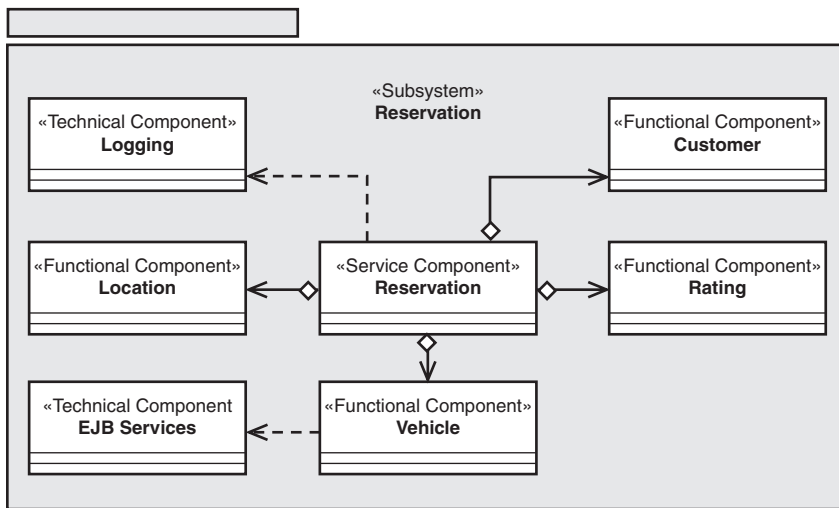


Figure 4-6 The relationship between a subsystem and its constituent service, functional, and technical components

Identifying the various subsystems followed by the derivation of the service components and its constituent functional and technical components that together define a subsystem is the crux of this major step of subsystem analysis.

4.2.3.3 Component Specification

From here on, it is AAU work! There is nothing very special about SOA that we would do. Providing the detailed microlevel design is the focus of this step (that is, the software model in terms of class diagrams, sequence diagrams, and so on). Each service component identified and designed at a high level in the preceding step is further elaborated into a much more detailed treatment. Some typical microdesign steps that apply to a service component are the following:

1. Identify component characteristics, including component attribute and operations together with the policies and rules that they implement.
2. Identify events and messages that the components must sense and respond as a triggering mechanism. Incoming and outgoing component messages are also specified.
3. Identify internal component flow (representing the internal flow of control within the service component and which can be represented as a sequence or collaboration diagram).
4. Create component class diagrams, which are class models that show the static relationships between the functional and technical components.

Similar types of microdesign activities must be performed for each of the functional and technical components so that they are designed and specified to an adequate and unambiguous level of detail to be comfortably handed over to the implementation team for converting into working code. Because these are AAU activities, they are not covered in any further detail in this chapter.

Phew! That was a long section, but we did cover a lot of ground.

Okay, so what did we achieve in the service specification phase?

We were able to filter out only those services that are perfect candidates for exposure, and we did that by applying the service litmus test.

For each of the services tagged for exposure, we provided prescriptive guidance on how to design (at a macro level) a full specification for them, including the following:

- How services are dependent on each other (service dependencies)
- How services are orchestrated together to form composites that enable business processes (flows) (service compositions)
- Identifying and documenting the nonfunctional requirements that each service must implement and comply with (service NFRs)
- Detailed specification of the service messages (service message specification)
- Identifying and documenting the state requirements for each service (service state management)

While developing service message specifications, we acknowledged how these specifications tie in with and influence the information architecture, the integration architecture, and the data architecture of the system. Services are not the only SOA construct that we designed in this phase of SOMA. The processes were further elaborated and their flows were represented as an orchestration of services and IT components. The service component was also designed using their constituent functional and technical components.

From a service model standpoint, what have we achieved?

- Provided a service exposure rationale
- Addressed service dependencies
- Addressed service compositions and how they help in realizing process flows

- Emphasized the need to document service NFRs
- Addressed the recommended approach to define service messages
- Explained why it's necessary to document state management

Having achieved this, we move on to the third and last phase in this method: realization decisions for services.

4.2.3.4 Realization for Services

The method in its first two phases not only demonstrated how to use a three-pronged approach for service identification but they also offered guidance about how to provide detailed specification for the services, service components, and process flows. The main focus of the method in this phase is to provide guidance about how to take architectural decisions that facilitate service realization. It is important to note that SOMA does not, at this point in time, address the actual implementation of services; instead, it provides enough information and detail so that an SOA implementation phase can just concentrate on the development and implementation of the services. Implementation is the phase wherein a specific technology, programming language and platform is chosen and used to transform the design specifications and the realization recipes and patterns into executable code.

This phase has three major activities: component allocation to layers, technical feasibility analysis, and realization decisions. The rest of this section focuses on these three major activities.

4.2.3.5 Component Allocation to Layers

So far, this method has identified services, process flows, service components, functional components, and technical components. It also argued that technical components belong to a genre of components that do not directly provide business functionality but instead focus on delivering infrastructure functionalities that might be used by multiple functional and technical components. Keeping the solution stack in mind, we want to provide architectural recipes to allocate the SOA artifacts that we have identified, to the pertinent layers in the solution stack.

The service components and the functional components are all allocated to Layer 2 in the stack. The services, both atomic and composite, are allocated to Layer 3 in the stack. The process flows orchestrated using services from Layer 3 and functional and technical components from Layer 2, are allocated to Layer 4 of the stack. Technical components are of different types. There can be technical components that encapsulate a persistence framework or a data access layer. This type of technical component is usually allocated to Layer 8 (the data architecture layer). Some technical components encapsulate event management functionality, whereas others might provide queue management functionality, and some may encapsulate transaction management features. These types of components are allocated to Layer 6 (the integration layer). There can be other types of technical components, such as cache management, permissions management, audit management, and so forth. These types of components, which usually assist in complying with QoS requirements, are usually allocated to Layer 7 (the QoS layer). As you can see, based on the characteristics of each layer in the reference architecture, the method assists us to map the various types of software artifacts to the layers.

Without paying specific attention to the names of the components, specifically between Layers 1 and 4, Figure 4-7 depicts how different types of software building blocks, which the method assists us in identifying, are allocated to each layer in the solution stack.

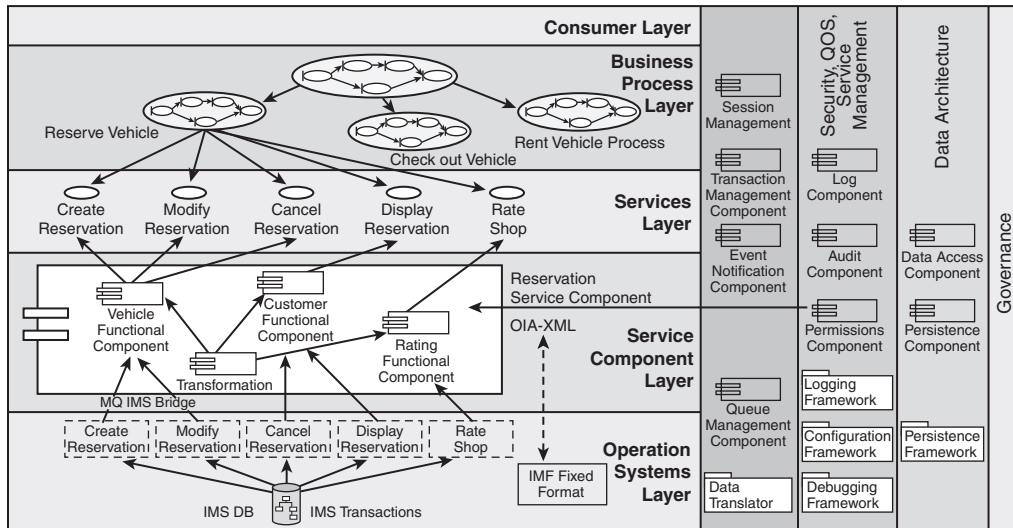


Figure 4-7 Allocation of software artifacts on the layers of the SOA reference architecture

4.2.3.6 Component Allocation to Layers—Technical Feasibility Analysis

Technical feasibility takes input primarily from existing asset analysis and takes into account the services portfolio defined during service specification. The main focus of this activity is to perform a detailed analysis of existing systems to evaluate how much of the existing technology-specific implementation can be leveraged to realize the services. This activity, as is the SOMA method itself, is primarily iterative in nature, and it can be started as early as during the EAA technique during service identification. The functionality, along with the transactions identified during the early stages of service identification, needs to be validated for feasibility for componentization. This type of feasibility analysis results in architectural decisions for either the functional or operational architecture of the system. This is the step in the method where the deep-dive technical analysis of leveraging existing system functionality is actually formalized. You can do as much service specification as you want, but if you do not provide deep insight into the implementation of the service, ably supported by its justification, there still remains that proverbial gap between where architects hand off the design and where developers start with the exact knowledge on what to implement!

Let's consider an example of assessing the feasibility of using legacy system functionality. We must consider many technology aspects of legacy systems when looking to reuse such an existing asset for service realization. Some notable examples include the following:

- Exception handling in legacy systems is typically in the form of program termination. This might not be acceptable in a real-time SOA-based system.
- Authentication and authorization are often built in to the legacy application code using proprietary technologies and protocols. If legacy functionalities protected via security credentials are considered for reuse, there needs to be a mechanism to integrate the embedded security credentials in a federated environment. Externalizing the legacy security credentials might raise technological issues that will have to be addressed.
- The typical nightly batch processes for data synchronization or request submission may just be too infrequent to be used in real-time scenarios. In such cases, the legacy processing system might need to be amended; in extreme cases, the process might prove unusable.

These examples provide a snippet of the challenges that must be addressed when considering the reuse of existing systems and their functionality. Technical feasibility analysis addresses these types of issues by taking architectural decisions and justifying them for consideration.

4.2.3.7 Realization Decisions

The technical feasibility analysis has a significant influence on how services ought to be realized in the best possible manner. Although technical feasibility analysis is initiated very early in the identification phase of the method and is performed throughout the various phases, considering all the various design and implementation alternatives, this step formalizes the final realization decision for each service and provides justification for the choice. This justification is a key step in the process and helps in maintenance and enhancement of the system in the years to come. The same realization alternatives could well be re-analyzed during enhancement of the system a few years down the road; and while doing so, the then-available technology might justify a different alternative as better suited. Thus, the snapshot in time of the architectural justification often proves invaluable.

In general, this method recommends considering six different high-level realization alternatives, as follows:

1. **Integrate**—Wrap existing legacy applications with SOA technologies and provide a service façade using open standards and protocols, for seamless integration into an SOA. Adapter technology is typically suited for this purpose.
2. **Transform**—Transform parts of legacy applications and expose them. This might involve the extraction of business rules or policies and rewriting the code in a modern SOA-aware programming language. This often falls under the discipline of legacy modernization.
3. **Buy**—Purchase products from independent service vendors (ISV) who provide out-of-the-box functionality that are exposed as services. Note that this option often results in a classic SOA antipattern in which the features of the ISV product often dictate the requirements of an enterprise. So do not fall into this trap and only evaluate the ISV functionality in the context of the project requirements.

4. **Build**—Build applications following the principles and best practices of SOA. This is often called the domain of custom application development.
5. **Subscribe**—Subscribe to external vendors who provide services that meet or exceed specific business functional requirements. Various SOA vendors are trying to find a niche in the market where they can specialize in a specific type of offering. Credit card authorization service is a classic example. Rarely would we see any enterprise developing this functionality indigenously. Instead, they subscribe to the best service provider that suits their needs.
6. **Outsource**—Outsource an entire part of the organization’s functions to a third party. This is not yet considered mainstream because SOA is still undergoing some critical phases in its maturity and adoption. However, there are companies that specialize in, say, HR operations, and we have started seeing big corporations outsourcing an entire department. These third parties will provide services that need to be seamlessly integrated with the enterprise business processes.

This is what realization decisions help us achieve: a justification of the implementation mechanism for the services in the services portfolio.

So, what did we achieve in the realization phase?

- Understood how to allocate the various software building blocks onto the layers of the solution stack.
- Appreciated the justification to perform a detailed technical feasibility analysis before using existing legacy functionality for service realization.
- Identified the various options available for service realization.

And from a service model standpoint, what have we addressed?

- We were able to provide realization decisions for services.

This marks the completion of the three phases of the SOMA method. By now, I hope you can appreciate why a service-oriented analysis and design method, like SOMA, is required in any SOA-based initiative. The treatment provided here has hopefully demonstrated how SOMA is built on top of OOAD, while adding modeling and design techniques specific to SOA.

4.2.4 Using SOMA

The language of SOMA is crisp, clear, and very much focused on, providing a methodology to solve the challenges the IT community faces regarding service-oriented design. However, it is important to keep in mind that tools are integral to processes or methods, and a well-articulated method drives the development of tools in support of them. Rational Unified Process (RUP) has extended its process technique to incorporate service-oriented design, and it has used the SOMA method as the basis for its extension. This method extension is available as a plug-in in a product from Rational Software called the Rational Method Composer (see the “References” section).

SOMA method artifacts can also be expressed as a platform-independent model. Think of SOMA as providing a meta-language that helps in defining a service model. This model representation, defining not only the SOA constructs but also the relationships and constraints between them, is called the meta-model for SOMA. Any design tool that can implement the SOMA meta-model will be able to provide a tooling environment for service-oriented modeling and design based on the SOMA method. Although we do not provide the entire meta-model here, we do offer hints about how to develop one.

Hint: A *business domain* can be decomposed into one or more *functional areas*. A *functional area* can be decomposed into one or more *subsystems*. A *subsystem* contains one or more *service components*. A *service component* uses one or more *functional components* and *technical components*. If you try to enlist all the various constructs of SOMA and then model relationships between them, together with constraints on the relationships, you will be able to create the SOMA meta-model. It is then just a matter of implementing the meta-model in a software modeling tool! IBM has already developed a tool for SOMA and has been using it productively and successfully in multiple client engagements.

4.3 Conclusion

This chapter provided a detailed overview of a service-oriented design methodology, and we used the IBM-developed SOMA methodology as guidance on how to effectively develop an SOA-based system design.

The chapter also covered the SOA reference architecture, also called an SOA solution stack. It described each layer and identified the kind of software building blocks that constitute each layer. It then focused on how the SOMA method assists in the development of the first-class constructs of SOA: service, service components, and flows through the three phases of identification, specification, and realization.

As you finish this chapter, we hope that you now appreciate the need for a service-oriented design methodology and have learned how to execute the same via the SOMA methodology.



4.4 Links to developerWorks Articles

A.4.1 Arsanjani, A. et al. Design an SOA Solution Using a Reference Architecture, IBM developerWorks, March 2007. www-128.ibm.com/developerworks/library/ar-archtemp/.

A.4.2 Arsanjani, A. *Service Oriented Modeling and Architecture*, IBM developerWorks, November 2004. www-128.ibm.com/developerworks/webservices/library/ws-soa-design1/.
www-128.ibm.com/developerworks/architecture/library/ar-archtemp/.

4.5 References

Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

Grossman, B. and J. Naumann. ACORD & XBRL US-XML Standards and the Insurance Value Chain. Acord.org, May 2004. www.arord.org/news/pdf/ACORD_XBRL.pdf.

Deblaere M. et al. IBM Insurance Application Architecture (IAA), White Paper, April 2002. www.baioxian119.com/xiazai/updownload/iaa2002whitepaper.pdf.

The complete and latest release of Justice XML Data Dictionary (JXDD). <http://it.ojp.gov/jxdd/prerelease/3.0.0.3/index.html>.

The official BPEL specifications are maintained by OASIS. www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

The official site that maintains the enhanced Telecom Operations Map (eTOM). www.tmforum.org/browse.aspx?catID=1647

Download the latest Open Travel Alliance (OTA) specifications from the opentravel site. www.opentravel.org/.

Download a trial version of Rational Method Composer. http://www14.software.ibm.com/webapp/download/product.jsp?id=TMMS-6GAMST&s=z&cat=&S_TACT=%26amp%3BS_CMP%3D&S_CMP=

Download the RUP plug-in for SOA. http://www.ibm.com/developerworks/rational/library/05/510_soaplug/

