Overview of SOA Implementation Methodology

Enterprise SOA defines a set of business-aligned IT services (available to participants throughout the enterprise across multiple lines of business or even outside of the enterprise) that collectively address an organization's business processes and goals. These services can be combined in a variety of different ways to support enterprise business processes and business solutions. By ensuring that there is a business focus of its main constituents (business services and business processes), the SOA architectural style promotes alignment of business requirements and technology solutions. Both processes and services are driven by the business architecture and can be traced back to the business outcomes that they help to realize. The major forces shaping the SOA architecture and its major elements are shown in Figure 3-1 and discussed in the following list:

- The forces that drive the business and SOA — the enterprise business drivers — are at the top. These are things like strategy, competition, market forces, regulatory forces, and so on. They all combine to drive the business architecture (model) and to shape the measurement and feedback for enterprise-wide performance management.
- The business model is the representation of the business resources and processes that are required to meet enterprise operational, tactical, and strategic business goals. Having a business model is critical to the successful alignment of services with business goals and objectives, and consequently to the overall SOA implementation's success.
- The semantic information model defines the common business information for a given enterprise (such as customer, agreement, etc.). These objects effectively create an ontology of the enterprise data by defining common concepts (and their content) that describe the operations of the enterprise. Using the semantic information model to define business service interfaces leads to the creation of semantically interoperable services — a semantic SOA.
- Other aspects that enable SOA to provide value are: key performance indicators (KPIs) and portfolio rationalization. The KPIs enable quantitative assessment of the impact of SOA and allow business processes and services to be measured and optimized. Portfolio rationalization enables

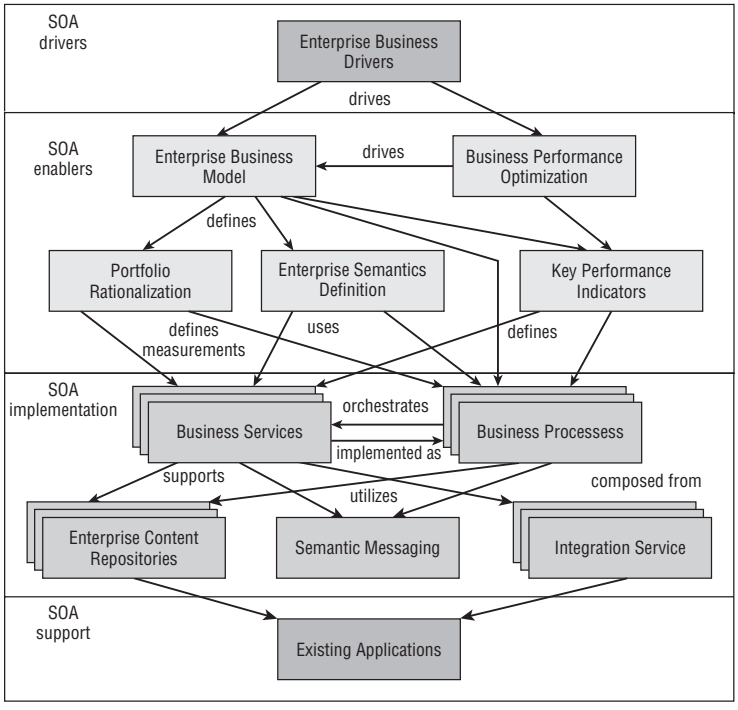


Figure 3-1 Major elements of enterprise SOA

the enterprise to simplify and consolidate infrastructure, applications, and data, where SOA plays a leading role in the implementation of the consolidation activities.

- In terms of implementation, the primary aspects are business processes and services. The business processes orchestrate the execution of business services to implement enterprise capabilities as specified in the business model — for example, order processing or claims processing. Business processes are usually associated with operational objectives and business goals (such as insurance claims processing or engineering development processing) in the form of specific outcomes that can be measured against KPIs. These KPIs are collected as part of the process implementation and are usually used to evaluate organizational performance.
- The services implement specific enterprise business functions and access the business data and resources. Well-defined, business-aligned services are a critical ingredient of a flexible, extensible enterprise SOA implementation. The structure of services allows them to be independently developed and deployed. Correctly defining and aligning services with

the business and semantic models results in plug-and-play implementations that can effectively be combined into different enterprise-wide business processes and/or solutions.

- Information represents the data resources of the organization. Data resides in a variety of different stores, applications, and formats. Different levels of data are used by different levels of SOA constructs. The semantic information model defines the data for business processes and services. The information passed in business processes in the form of documents is based on the semantic information model. The documents provide a form of semantic message between processes and services. The SOA defines the mechanisms for transforming data from its native operational format to the semantic data required for the business processes.
- Documents can represent legal entities (such as financial documents, insurance policies and claims, and government regulations) that define the obligations of the enterprise and its partners. Documents are a vital part of modern enterprises and have to be included in the SOA implementations (along with the rest of the enterprise information) as first-class citizens.
- Information from existing systems and applications is made available to processes and services through a data virtualization layer.
- Functions from existing systems and applications are made available to services through integration services that expose the existing functionality through new service interfaces.

The effective implementation of service-oriented solutions is a complex undertaking that must take all of these different aspects into account. This requires cooperation among many groups within an enterprise, including management, business leaders, architecture, development organization, operations, and so forth. At an enterprise level, this would not be possible without a well-defined methodology, describing the major steps and work products, and the roles and responsibilities of each participating group. In the remainder of this chapter, we lay out a high-level methodology for enterprise SOA solutions. This methodology is shown in Figure 3-2.

The methodology consists of the following major activities:

- **SOA reference architecture** — Define the important aspects of the SOA reference architecture, in particular what a service is, the types of services and their relationships, design and implementation concepts and processes, and relationships to other architectures and communications.
- **Business architecture definition** — The first step is to define the enterprise business architecture. This influences the processes, services, information, and enterprise solutions that will be built.

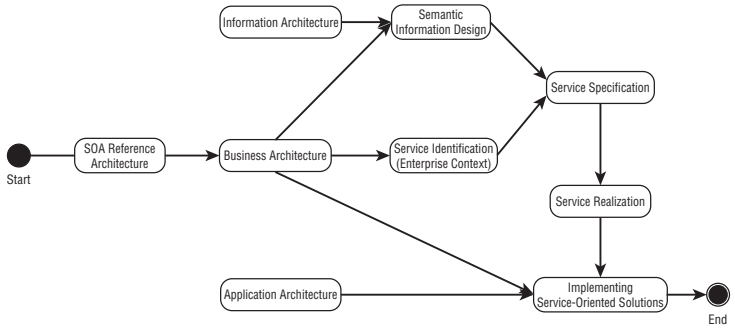


Figure 3-2 SOA methodology

- **Service identification** — Define a set of services within the enterprise context that supports the business architecture. The overall set of services makes up the service inventory.
- **Semantic information model definition** — Create an enterprise information model that defines the shared semantics of processes and services. This activity is often done in parallel with service identification. Note that the semantic model is influenced both by the business architecture and by the information architecture.
- **Service specification** — Create service contracts that can be used at design time for the selection of appropriate services in solutions. The service specification includes the service interface as well as other usage and dependency information.
- **Service realization** — Design and implement services.
- **Implementation of service-oriented solutions** — Build enterprise solutions from services. Also notice that the service-oriented solutions are influenced by the application architecture. It is important to note that this is not a linear, waterfall process. You do not need to have a complete business architecture or a completely specified service inventory before you can start designing and implementing services. The process is iterative and incremental. You start by creating a high-level business architecture and service inventory. Then you go about implementing the first set of services to support specific business goals. As you learn from this process, you update your SOA architecture, business architecture, service inventory, standards, governance, and the like. Then, you start building your next set of services.

Also notice that the structure of this book mirrors this process:

- The SOA reference architecture is covered in Chapter 2.
- Business architecture is covered in Chapter 4.

- Service identification is covered in Chapters 4 and 5.
- The semantic information model is covered in Chapter 5.
- Service specification and interface design are covered in Chapter 6.
- Service realization is covered in Chapters 7 and 8. Chapter 7 describes service implementation design, and Chapter 8 covers service composition.
- Service-oriented solutions are covered in chapters 9–12. Chapter 9 covers the overall issues and architecture related to enterprise solutions. Chapter 10 covers integration. Chapter 11 is on security, and Chapter 12 is on governance.

SOA Reference Architecture

One of the first things that needs to be done before embarking on enterprise SOA solutions is to initiate the SOA reference architecture as described in Chapter 2 and detailed in Figure 2-14. In reality, it takes some time to complete the reference architecture (if architecture is ever really finished). That is to be expected. It is not important to have everything worked out before you start or to have complete models, documentations, standards, and governance in place before allowing the first service to be designed and built. But, it is important to have an idea of what you're doing. It is important to have a high-level vision of the architecture and the context that the architecture provides in terms of the service hierarchy, service inventory, and semantic information model, before you create very many services.

We recommend creating what we call a minimum architecture. The minimum architecture determines the few things that absolutely must be standardized in order to meet the enterprise goals and clearly specifies them. Then, it puts an architectural vision in place for how the rest of the architecture might be defined, and a process for continual, incremental enhancement and improvement of the architecture. For enterprise SOA, those crucial things are the service definitions, service inventory, and semantic information model. We provided our vision of the SOA reference architecture in the previous chapter. It is based on our extensive experience with proven implementations, and we encourage you to adopt it. It is up to you to define the inventory and information models for your particular business, but we do explain the techniques for creating them.

In the next few sections, we describe a sample architectural roadmap. Your particular roadmap depends on your own requirements and circumstances, but this example illustrates the basic concepts and contents of a roadmap for an SOA architecture.

Minimum Architecture

The minimum architecture should specify:

- **What a service is** — The types and granularities of services. For example, business, domain, utility, integration, external, and foundation services.
- **Required interfaces and functions** — Interfaces or other functions that services are required to use or support. For example, all services must support the management interface and use the logging service.
- **Technical infrastructure** — What technology services use to communicate. For example, Web Services conforming to the WS-I Basic Profile v1.1 and Security Profile v1.0.
- **High-level semantic information model** — Identify the major enterprise business entities and documents. What information do they need in common to meet enterprise goals? What information needs to be shared between services? For example, a consolidated customer entity supports the business goal of having a single customer view. The high-level model should identify 20–40 business entities and documents.
- **Initial service inventory** — Identify the major service groups and services needed to support enterprise goals and processes. Determine an organizational structure (such as line-of-business or functional domain). Integrate appropriate industry standards or patterns. The initial inventory should identify 30–50 services and service groups.
- **High-level business model** — Identify the major enterprise business processes and the common processes that occur across enterprise domains. Identify the underlying capabilities needed to support those processes. The high-level business model should identify 10–20 major processes and 20–40 capabilities.
- **Service identification, specification, and design process** — This describes how the architecture and enterprise context fit into and support the development process.
- **Architecture life cycle process** — This is a feedback mechanism for the constant updating and enhancement of the architecture.
- **Roadmap** — The roadmap addresses at least two areas. The first is a rough priority order of service implementation based on dependencies, commonality, and usefulness. This doesn't specify a timeline, nor take into account other business drivers, but it provides an initial vision for building out the service inventory. The second is a high-level plan for building out the architecture.

The minimum architecture should take between 4 and 8 weeks to produce, depending on the size and complexity of the enterprise, and the experience, capability and number of architects.

9-Month Checkpoint

Once the architectural vision (minimum architecture) is in place, you can start to implement services and use them in enterprise solutions. Often, this begins with a small-scale or pilot project to really figure out how to do it, and then expand from there. The architecture and process needs to be updated based on the knowledge gained from this process. After 6–9 months, the following additional architecture aspects should have been developed:

- **Governance** — Processes for design-time and deploy-time governance are put in place.
- **Metrics** — Measurements to demonstrate the usage and value of SOA are defined. Implementation of metrics is started.
- **Services metamodel** — A formalized service definition is created in the form of a metamodel.
- **Integration services** — Patterns and techniques for how to implement integration services are in place.
- **Updated business and information models** — The models are updated to include prior implementations.
- **Updated service inventory and roadmap** — The service inventory and roadmap are updated to include existing services and to factor in new business models and other forces.

18-Month Checkpoint

Typically around the next checkpoint, the architecture and the organization are ready for a larger-scale rollout of SOA. For this to be effective, the architecture and processes need to be complete and clear enough for a broader audience of developers. At this point, the following aspects should have been introduced:

- **Updated architecture** — The architecture is updated based on past experience and projects. It is also documented more completely.
- **Formalized process** — Governance and development processes are enhanced, formalized, documented, and measured.
- **Design-time repository** — A design-time repository is introduced and integrated with the service inventory.
- **Versioning** — Versioning policies, procedures, and infrastructure are in place.

- **BPM** — Business processes are constructed using services to implement process tasks. The rules and constraints are clearly defined.
- **SaaS** — Services provided by external vendors or software-as-a-service providers make up a portion of the overall service inventory. Integration techniques, rules, and constraints are clearly defined.
- **Reporting** — Information from metrics is collected and reported on. Process and architectural improvements can be identified and measured. The SOA's value can be measured and demonstrated.
- **Integration with enterprise architecture** — SOA and EA activities are well coordinated.
- **Updated business and information models** — The models are updated to include prior implementations.
- **Updated service inventory and roadmap** — The service inventory and roadmap are updated to include existing services and to factor in new business models and other forces.

Long Term

Long term, there are many things you can do to continue to enhance the value of the architecture and improve organizational effectiveness and business agility. These are the more advanced aspects of the reference architecture. The ability to implement them and benefit from them depends on the maturity and capability of business and IT. Many organizations do not get as far as this with their architecture program, but we have seen the benefits of these activities when they are implemented and believe it is important to at least mention the possibilities:

- **Model-based development (MBD)** — Integrate the architecture into a model-based development process and tool.
- **Formal metamodels and perspectives** — Formalize the architecture in terms of metamodels and perspectives that support both MBD and EA.
- **Tool and framework integration** — Create tools and frameworks to automate compliance and implementation.

Business Architecture

The foundation of a business-aligned SOA implementation is an enterprise business model, containing the primary representation of the resources (business, IT, data, etc.) and processes involved in meeting the enterprise's

operational, tactical, and strategic business goals. Business architecture (BA) is an essential component of a successful service-oriented implementation, providing consistency and flexibility of services across the enterprise.

We go into some length to define business architecture in the next chapter, so we're not going to try to define it here. Instead, we'll describe what aspects of BA we're concerned with when implementing an SOA or enterprise solution. BA must answer the following questions:

- What business are you in?
- What are the goals and objectives of this particular business?
- What outcomes are needed to achieve those goals?
- What is the strategy for achieving them?
- How will they be measured?
- What capabilities and information are needed to achieve those outcomes?
- What processes, services, entities, and rules are needed to implement those capabilities?
- What existing applications provide basic capabilities and information?
- How are the applications, processes, and so on, aligned with the business strategies and goals?

All very good questions. Business architecture helps you to understand and answer these questions, and it describes how to provide traceability, from the operational concepts of processes and services, through to the concepts of tactics and objectives, all the way up to business goals and strategy.

Business Processes

Business tactics and objectives are typically defined for particular business processes. A *business process* is a group of logically related (and typically sequenced) activities that use the resources of the organization to provide defined results. Business processes deliver value in the form of products or services, often to an external party such as a customer or partner.

In order to accommodate the needs of both executive management and business process owners, business processes are typically defined at two levels of detail: ''One model, for the executives, contains a set of high-level business scenarios that show the intent and purpose of the organization. The other model, for the business process owners, contains a detailed set of use cases that define how the organization needs to function internally. For each high-level business scenario, you could define one, or several, detailed business use cases representing the same activities in the organization. . . .''

(IBM's Rational Unified Process [RUP] for SOMA). This kind of analysis can be thought of as a type of process decomposition.

The high-level scenarios are the high-level descriptions of what business systems do. This level of processes defines only the highest-level enterprise scenarios and is rarely detailed beyond the narrative. Processes, such as Order to Payment, fit this level. These descriptions typically serve as the input (starting point) for process decomposition. Such decomposition defines business processes (sometimes called level 2 processes), which are the foundation of the enterprise business model. Receive Purchase Order is an example of a process that supports the order to payment scenario. Level 2 processes are also a foundation for the definition of the process activities (steps that make up the processes), which are used for definition of the high-level business services. For example, the Receive Purchase Order process might be composed of Purchase Order, Customer, Inventory, Credit Checking, and other business services. In other words, business process decomposition provides three levels of hierarchy — top-level scenarios, made up of (level 2) processes, composed from business services.

The goal of SOA is to expose an organization's computing assets as reusable business services, implementing basic business capabilities, which can be (re)used and integrated more readily using business processes. The relationship between business services and business processes (shown in Figure 3-3) paves the way to a truly flexible enterprise:

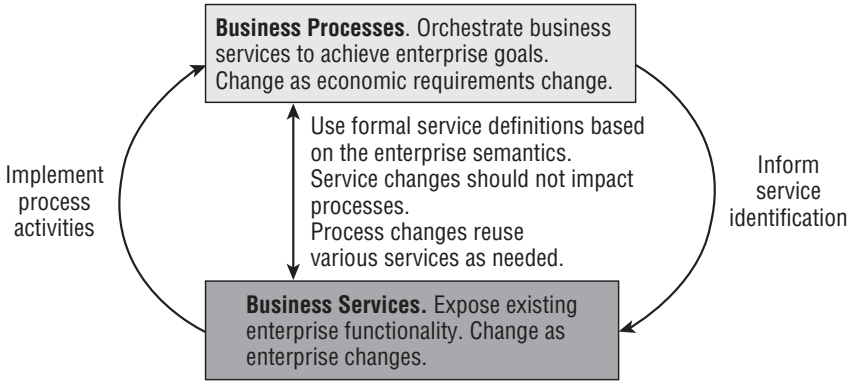


Figure 3-3 Relationship between business services and processes in SOA

- Business services support stable business artifacts, incorporating processing and rules whose interfaces change fairly rarely. (Note though that the service implementations can and typically do change frequently.)
- Business processes support fairly fluid business procedures and rules, which can change every few months or even weeks.

- The interaction between business processes and business services is based on the enterprise semantics, which minimizes the impact of service changes on the business processes and simplifies building processes from business services.

This separation of responsibilities enables business analysts and IT architects to reuse IT capabilities, encapsulated in business services, through the composition of business processes. This simplifies the creation of new processes and optimization of the existing ones. More importantly, once designed, processes can be quickly modified in response to market conditions. All this translates into increased business flexibility and competitiveness, while reducing the incremental costs of making frequent process changes.

Information Design

The next step in the process definition is creation of the enterprise semantics (semantic information model) — a definition of the standard business entities for the enterprise; for example, insurance policy, claim, and so on. A common semantic definition ensures that:

- Each term throughout the enterprise has a clear and concise definition.
- All enterprise terms are used consistently (mean the same thing and use the same definitions) throughout the enterprise.
- Each term is used in at least one process/activity definition.
- Only terms defined in the enterprise semantic information model are used by process/activity definitions.

The semantic information model is influenced by both the business architecture and the information architecture. The business architecture identifies the processes required to support the business goals and objectives. The semantic information model defines the information, concepts, and meanings that must be common throughout those processes to effectively pass information between the process steps. This corresponds to the information architecture concepts of semantic data as illustrated in Figure 2-6.

The semantic data is not the same as the domain data. It does not define all of the details of the information needed within each step of a process. Rather, it defines the information that must be common between then. Each individual process's step (implemented by a business service) provides any transformation required between the semantic information model and its own internal domain model.

NOTE **In this context, objects and entities refer to business "things." We are using these terms without the connotations associated with object-oriented or entity-relationship modeling. In other words, business semantics described here are used only as a foundation for service interactions (messaging model), not for service implementation.**

Although the semantic information model seems similar to a standardized enterprise data model, the two are radically different and should not be confused with each other. The semantic information model defines the messages exchanged by services. The messages implement interservice communication. Thus, they are transient and do not reside in a data store (at least not explicitly). In contrast, the enterprise data model defines the data structure and the relationships between data in the database. Because in practice implementation of the SOA involves service enabling of existing enterprise applications, changing the underlying data model is an extremely expensive proposition that often requires the complete rewriting of applications. In other words, it's probably not happening, so a system that provides interoperability without changing existing models is going to be better.

An SOA implemented, based on the semantic information model, provides a semantically interoperable SOA. Such an implementation offers enhanced interoperability between services. At the interface level, all of them work with the same objects. In effect, this eliminates the need for message transformations between services. Because service interfaces are created according to the standard enterprise semantic information model, it is guaranteed that every service can understand and correctly interpret any message, regardless of who the service consumer is.

THE FUTURE OF THE SEMANTIC INTERFACES

The introduction of semantic data for service contracts also allows for rethinking the design of service interfaces. It is no longer necessary to send specific request/response message pairs between the consumer and provider for each service operation. Because the interface data models for all services are driven by the same semantics, it is possible to introduce the notion of passing the service execution context around as part of the service invocation "thread." In this case, the service interface operations are massively polymorphic and expressed as:

```
Service.method (XML context in, XML context out)
```

The context in this case is a service execution context, expressed as an XML document supporting enterprise semantics. In this implementation, any particular service can extract data that it is interested in from the context.

This solution reverses responsibilities: Instead of the service consumer building a specific interface for a participating service, the service itself is

(continued)

THE FUTURE OF THE SEMANTIC INTERFACES *(continued)*

responsible for accessing the required information from the execution context and updating the context with the results of its execution. Such an approach minimizes the impact of service interface changes, as long as the required puts data is available in the execution context. This approach, of course, puts an additional burden on the service implementations, but it may be negligible compared to the expenses of realigning of the service consumers with the services interface changes.

This approach, however, can lead to significant control and data coupling between consumers and providers where the semantics of the service are hidden in the interpretation of data. This can make services more difficult to reuse, compromises encapsulation, and can make change management more difficult. (A provider interprets the data differently, changing the service, and consumers don't see this as a change in the service interface.)

There are plenty of industry (and cross-industry) consortiums today, defining data semantics for a particular industry, such as ACORD for insurance, or HL7 for healthcare. Their semantic dictionaries (if they exist) should be considered a starting point for the creation of enterprise semantic information models.

Service Identification

One of the most important tasks during implementation of a solution based on service-oriented principles is the proper definition of business services, based on the decomposition of the problem domain (see the sidebar ''SOA and Decomposition'').

SOA AND DECOMPOSITION

Decomposition is a well-known (and widely adopted) technique for dealing with complexity. The first software decomposition approach (introduced in the early 1960s) was splitting applications into separate jobs, each implemented by a separate program. Later, as more insight into program internals was gained, each program itself was split into modules or subroutines, according to its various functions.

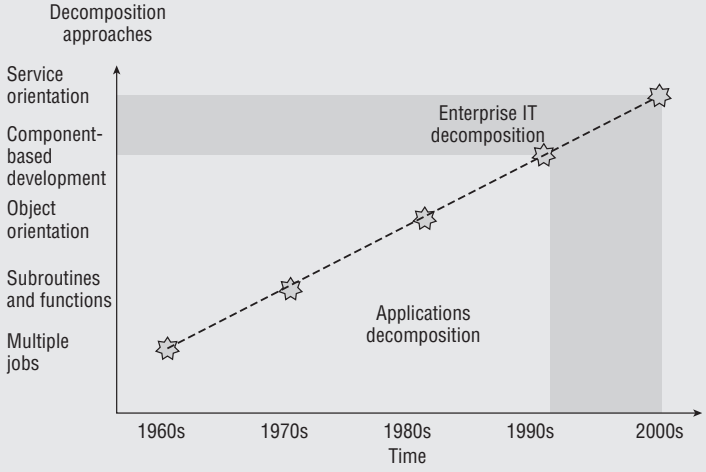
The object-oriented (OO) paradigm introduced by Simula and Smalltalk in the 1970s strengthened the adoption of decomposition by introducing objects: modules of code, each of which implemented a model of a real thing. The idea

was to represent in software the "things of the problem domain," for example customer, order, or trade. However the abstractions provided by objects turned out to be too fine-grained and intertwined with technical concepts to have a meaning on the business level. For various reasons, many object-oriented developers wound up spending most of their time dealing with technical constructs such as collections, graphical widgets, and so on. As a result, in most cases the objects of the problem domain disappeared inside amorphous modules, which no longer represented anything recognizable by domain experts. An additional problem with OO was the fact that although objects are an important decomposition approach during design and implementation time, they are not visible at either deployment or run times and consequently do not directly support either deployment- or run-time decomposition.

In the continued search for a better design paradigm, a different approach to decomposition was introduced in the late 1990s — components. The idea was to fix the problems of object orientation by raising the level of abstraction, increasing granularity, and creating a tighter linkage with the business "things."

Introduction of software components improved the creation of flexible, better structured, and more manageable software applications. Part of the improvement came from removing the object-reference-based coupling that was common in distributed object systems (there's that loose coupling thing again). However it did not solve the main enterprise IT problem: its application-centric nature. Both objects and components provide better design and development approaches for individual applications.

SOA brings decomposition to a higher level, as shown in the following figure. Instead of attempting to decompose applications, it decomposes the entire enterprise IT functionality.



Evolution of Decomposition Approaches

It seems like the simplest approach to decomposition (and consequently service definition), is to directly expose the existing application's functionality as a set of services (decomposition based on the existing application portfolio) — similar to the traditional enterprise application integration (EAI) practice. Unfortunately, such an approach rarely works. It ''is in essence technology first approach and is a recipe for disaster and/or serious over-engineering'' (Gary Booch, ''SOA Best Practices,'' Software architecture, software engineering, and Renaissance Jazz blog [March 11, 2006]). A better decomposition approach is based on the decomposition of the enterprise-wide business model: designing a set of services that define the enterprise architecture blueprint supporting the current business goals of the enterprise and providing capabilities for future changes. It requires you ''to start with the scenarios/business needs, play those out against the existing/new systems, zero in on the points of tangency, and there plant a flag for harvesting a meaningful service'' (ibid.).

Such an approach leads to the creation of a set of business-aligned IT services (available to participants throughout the enterprise across multiple lines-of-business or even outside of the enterprise) that collectively fulfill an organization's business processes and goals. The resulting business services are independent from the current enterprise application portfolio and support the ''ideal'' enterprise architecture.

Hierarchical decomposition, based on the enterprise business model is typically not sufficient for proper service identification. Although it provides an alignment between business and IT, it does not guarantee that resulting services will adhere to the basic service tenets. The service characteristics defined in Chapter 2 need to be considered in the design process.

But still this is not enough. The services need to be defined within the context of the overall enterprise. To do this, you need two things. First, you need to think about the way you design systems and decomposition differently. To overuse a phrase, you need a paradigm shift in design practice. Then, to support the new paradigm, you need an easy way to find the existing services.

For example, a typical approach to SOA design might incorporate this sequence:

- For each business domain, identify and analyze the processes.
- Break the processes down into tasks that are implemented by services.
- Look for existing services that perform the specified tasks.
- Use existing services when possible.
- Design and implement new services.

This probably seems like a pretty reasonable approach, but let's look at an SOA-focused sequence and compare:

- For each business domain, identify and analyze the processes.
- Understand what services currently exist (or are planned) and their responsibilities.
- Use existing services to frame the design, and break the process down into tasks that are implemented by services.
- Use existing services when possible.
- Design and implement new services where necessary.

The difference comes at the breakdown of processes into tasks and services. The difference may seem subtle, but the effect is huge. In the first approach, you are free to come up with almost any reasonable sequence of tasks to implement your process. There could be dozens of possibilities. Then, you look for existing services that do things your way, but probably don't find very many. Instead, you implement new services, but ones that overlap with existing services. In the SOA approach, you factor in the existing services first and then design around them. They provide a design constraint that limits the possible solutions to a few, instead of dozens. Now, when you use existing services, they've already been designed in, and they work with your new solutions and support your enterprise. Instead of promoting new services, you facilitated reusing existing ones.

The crux is this. You are not designing a solution or process from scratch. Instead, you are starting with an existing base and building your solution on top of it. You are extending and reusing, adding value to what exists, not duplicating responsibilities and adding inconsistencies. But to make this work, you need to be able to find the existing services. This requires an easy way to search for and find services at design time, and an organization of services that makes it easy to understand the overall set of services. We call the overall set of services the *service inventory*.

The service inventory lays out the overall set of services and their relationships to each other and the overall enterprise goals. You can think of the service inventory as a *responsibility map* of service interfaces. It should clearly describe the overall set of services, and what responsibilities the different service groups perform, and don't perform. The service inventory helps you in two important service design activities.

First, the inventory allows you to quickly scan the overall set of services at a high level and then to dig deeper into groups of services within a given area. This helps you to locate the services to support your look-first, design-later approach.

But at least as important, the inventory helps you to make decisions about what functions to include within your service implementations, and what functions you should expect to be performed by another service. If you need to implement a new service, you have to make sure that it doesn't duplicate functions that are already (or plan to be) implemented by other services. This is where the responsibility map aspect of the inventory is important. It must clearly define the boundaries of responsibility for services and service groups.

Service Specification

Once services and their corresponding semantic models are identified, they need to be described (specified) correctly. The complexity of proper service specification stems from the fact that there are two very distinct groups of service users that require information about services: business users (business analysts), who need to decide whether a particular service can be used in the solution that they are designing, and technical users (developers), who need to know how to write the code, invoking a particular service.

Business users need to understand what a service does in business terms, which requires answers to the following questions:

- What does the service provide for prospective clients? This includes a description of what is accomplished by the service, limitations on service applicability and quality of service (QoS), and requirements that the service requester must satisfy to use the service successfully.
- How is the service used? This includes a detailed definition of the content of service requests and responses, the conditions under which particular outcomes occur, and, when necessary, a step-by-step description of processes leading to those outcomes.

Technical users need to know how to implement service operations that require answering the following questions:

- How to interact with services? This specifies a communication protocol, message formats, including serialization techniques and service locations, for example, the service endpoint URL.
- What are the service invocation policies? This defines specific requirements for service invocation, for example, security requirements, required SOAP headers, and so on.
- What are service QoS guarantees? This specifies the quality-of-service characteristics that the service provides, including response time, throughput, availability, planned maintenance, and the like.

CURRENT PRACTICES FOR SERVICE SPECIFICATIONS

The notion of the service specification is widely recognized as one of the prerequisites for successful service usage. The problem is usually not the fact that a specification does not exist, but rather what the specification contains. Based on experience with object-oriented and component-based development, many architects and developers consider the service interface to be equivalent to the service contract. In the best cases, the service interface is supplemented by a free-form text document that captures some additional service information. Although this approach can significantly help, free-form documents are imprecise, hard to validate for completeness, and virtually impossible to process automatically.

For example, the popular web site `www.webservicex.net` provides a LloydsRiskCodeService service1 with the following contract:

> **Textual description of the functionality** — "This service returns Lloyds risk code details for a given risk code or description."
>
> **Textual definition** — "The following operations are supported:
>
> - `GetLloydsRiskCodeDetailsByRiskCode` — This method returns Lloyds Risk Code details for a given risk code.
> - `GetLloydsRiskCodeDetailByRiskCodeDescription` — This method returns Lloyds Risk Code details for a given risk code description."

The formal definition is in the form of the service WSDL and sample XML payloads (not shown here for brevity).

At first glance, the information seems sufficient to successfully use the service. However, let's take a closer look at how this contract can be used by different people.

On the business side, in order to decide whether the service is appropriate for solving a problem, the following questions must be answered:

- What functionality does the service provide? In our example, the information is supposed to be provided by the textual description of the functionality, but unless the user is acquainted with risk codes' definitions (`www.lloyds.com/Lloyds_Market/Tools_and_reference/Risk_codes.htm`) and can figure out which ones are really supported by the service, he or she can't decide whether it is appropriate.
- What are the limitations of the service? The textual definition does not provide any information about this. Examination of the service WSDL answers this question to some degree, but it's rare that business users ever look at it.
- Which SLAs does the service support? This is not specified in the service definition.

(continued)

CURRENT PRACTICES FOR SERVICE SPECIFICATIONS *(continued)*

- **What are the requirements that the service requester must satisfy to invoke the service successfully? The service definition does not specify any requirements on the input parameters.**
- **What are the detailed definitions of the content of service requests and responses? Some of this information is provided by the formal definition in the form of WSDL and XML samples. This definition assumes that the business analyst can understand XML, and that WSDL correctly represents the data semantics.**

Similarly, on the technical side, the following questions must be answered:

- **What are the communication protocols, message formats, including serialization techniques, and service location? This information is provided by the formal definition in the WSDL.**
- **What are the errors that service invocation can produce? This information is provided by the formal definition in the WSDL.**
- **What are the service invocation policies such as security requirements, required SOAP headers, and so on? Some of this information (SOAP headers) is provided by the formal definition in the WSDL. Other characteristics such as invocation policies theoretically could be added to WSDL, but they rarely are.**
- **Which SLAs does the service support? This is not specified in the definition above.**

So, you can see from this example (which is comparatively good) that much information needs to be provided in security specifications.

The service specification should define all of the relevant aspects of a service required by potential service consumers, including the *service expectations*, *interaction model*, *service constraints*, and the *service location*.

Service Expectations

The expectations define the result desired by the consumer who is using the service. This is also known as the real-world effect of using a service. For example, invoking the claims-processing service allows customers to get insurance payments. When potential customers invoke the service, they are not interested in a response indicating that their insurance company has merely recorded an application. Rather, they are interested in whether it will reimburse their losses.

Of course, the service provides encapsulation: The insurance company's internal systems record the claim without exposing this fact to the consumer. However, minimizing the client's assumptions about how the insurance company processes their claim increases the potential for smooth interaction.

Expectations associated with a service interaction are usually described in terms of the message traffic exchanged with the service. In some sense, similar to a service interface, it is possible to define expectations in terms of the kind of information that is provided by a service, as opposed to the information that is required for a current interaction.

Interaction Model

The interaction model defines the interaction between service consumer and provider through the service interface. Three key concepts are important in understanding what it is involved in interacting with services: information model, process model, and execution context.

- The information model defines the information that is exchanged with service consumers. This model should conform to the enterprise semantic information model. The scope of the information model includes the message semantics and their format (encoding). The message format defines the structure of the messages used for service invocation and response.
- The process (behavioral) model of the service defines the actions that consumers can execute on a service, the responses to these actions, and temporal dependencies between them. Temporal dependencies are mostly applicable to a conversational composite service, where interactions between the service consumer and provider can involve multiple service invocations.
- The service execution model defines the behavior resulting from interactions with the service. Some of this behavior can be private, and some public. The publicly visible portion of the service behavior is defined by the service execution model. The private behavior should never be made visible to service consumers.

Service Constraints

Service constraints describe rules, limitations, and facts about a service and its operations. Service constraints are usually expressed as policies. A policy is a statement of the obligations, constraints, or other conditions that either define service characteristics or have to be fulfilled by service consumers when invoking the service. There are two major types of policies that can be defined for a service:

- **Business-oriented policies such as hours of operation, return policies, and so on** — Business-oriented policies usually apply to the service operations, regardless of where and how these operations are deployed. For example, in order to invoke the claim processing service, a consumer must have a valid insurance policy.

- **Infrastructure-oriented policies such as security, privacy, manageability, performance, and the like** — These policies are defined for a particular service endpoint address. This means that there can be multiple service deployments, adhering to different infrastructure policies. For example, an appraisal service can be exposed through two different URIs. One guarantees a two-business-day appraisal response time, while the second guarantees fulfillment in five business days. Typically, the service provider charges differently for using these different endpoints.

Service Location

Invocation of a service requires its location, that is, the endpoint address. The same service can have several endpoint addresses. Multiple endpoint addresses may be employed for several reasons. As in the dual-URI appraisal service example, each endpoint address could support different policies. Often, multiple endpoint addresses are also required for different service methods. For example, withdrawal and inquiry methods on a bank account service expose completely different QoS requirements. On the one hand, the withdrawal operation requires guaranteed (once and only once) service delivery, reliability, and transactionality. These involve fairly expensive infrastructure support. On the other hand, the inquiry operation has less strict requirements. In case of failure, its execution can be retried. Since the frequency of inquiry is, on average, 5–10 times higher than that of withdrawal, it is not cost-effective to use the same expensive infrastructure for both methods. Such situations require that the service specification support different endpoint addresses for different service methods. Additionally multiple endpoint addresses can be used to support multiple versions and different infrastructure constraints that a given service can have.

To summarize, a service specification should provide information about the service's behavior, interface, and policies. This information covers service expectations, the interaction model, service constraints, and the service location. It provides the basis for implementing service consumers, as well as for dynamically binding consumers to the service provider(s).