Toward a pattern language for Service-Oriented Architecture and Integration, Part 2: Service composition

Ali Arsanjani, Ph.D.

December 02, 2005

Explore the realm of patterns for Service-Oriented Architecture (SOA) and Service-Oriented Integration (SOI) and examine some of the fundamental concepts behind SOA and some key architectural decisions that you can make in creating a robust and flexible SOA. The author discusses the architectural decisions related to the notion of service composition, which is when the design of the service composition helps achieve flexibility through the use of services.

View more content in this series

Introduction

One way to explore the value of Service-Oriented Architecture (SOA) is from the perspective of the flexibility that is provided to the business and IT sides of the value equation. Business value is created when a set of recomposable business process elements are defined as the key capabilities of a business. Then the business model that is created as a result of composing those business elements into processes can more effectively be supported by a flexible IT infrastructure with services that support those specific processes elements.

Another way to look at flexibility is in terms of future extensibility. This is achieved by introducing a service layer into an enterprise architecture. Packaged applications and existing systems can be integrated through SOA by a loosely coupled set of service interfaces that collaborate to achieve business objectives. Thus packaged applications can be mixed and matched in a more flexible manner than if the packages and applications were integrated point to point; rather they conform to SOA by implementing functionality required by the enterprise service model, which defines a set of services.

This flexibility of SOA is reminiscent of a pattern, namely, the Strategy pattern. In this pattern, services are used to decouple an interface from its implementation and, using Remote Service Strategy, that implementation is separated from bindings to individual communication protocols that occur within distributed systems architectures. Another insight that you can explore in this paper is the application of David Parnas' 1972 article entitled "On the Criteria To Be Used in Decomposing Systems Into Modules" (please see Resources section) to the world of services in SOA.

© Copyright IBM Corporation 2005

Toward a pattern language for Service-Oriented Architecture and Integration, Part 2: Service composition
In this seminal article, Parnas describes the criteria by which the partitioning of large and complex computer applications can be broken down into a set of constituent components or modules that can be interconnected in order to weave the fabric of the underlying functional requirements of the system. Although this article is often cited as advocating information hiding, which itself is the precursor to object-oriented programming, in reality this article goes far beyond the specific application of the technique of information hiding and deep into what I call externalization. Externalization is about encapsulating change. The criteria and techniques for externalization can be found in Variation-Oriented Analysis (VOA -- see Resources). In Variation-Oriented Analysis and Design (VOAD) it has been shown that the fundamental and underlying fabric of requirements that are used to drive system functionality and the specification of its nonfunctional requirements are contingent upon not only a set of common components (or set of components that support the functionality of the application), but more importantly, on how the variations in the structure perform their functions. This is in addition to the intention of the requirements that drive the application -- these variations in structure and behavior can be seen as variations in an underlying theme of functionality that is driven by business intention.

Variation-Oriented Analysis and Design

In VOAD, where appropriate, you seek to separate the more changing elements of the application from its more fundamentally stable and longer lasting parts. You can consider these "change types" to be focused on three fundamental kinds of elements within the application and its architecture. These include:

- The policies and business rules that are used to drive the application (rules for handling overdrafts for platinum customers)
- The process by which the flow of the application passes from control point to control point guiding the execution or flow of the application often by invoking services over functionality at each critical point of its flow (processing of a loan for a platinum customer)
- The type hierarchies by which types of elements into business are related to one another (Customer Type with Platinum, Gold, and Silver).

In reality, the catalyst in the adoption of Web services and SOA is the flexibility it can provide. This flexibility manifests itself in multiple dimensions. The key element is the ability to request the invocation of the piece of functionality which today may be provided internally within the organization by a given business line, and yet reserve the right to choose an alternative service provider in a later period in time. This new service provider can be someplace else in the organization (another business line), a business partner in the organization's ecosystem (see how to build service eco-systems in Part 1 of this series -- see Resources), or a third party vendor who provides the services as an IT capability.

Many businesses are intrigued by the ability to reserve the right to choose the implementer of the service based upon a set of potentially changing business agreements and requirements. This flexibility provides and supports immense business agility.

This flexibility in IT, which provides and enables a corresponding flexibility from a business perspective, is an implementation of the strategy pattern in combination with the separation of interface from implementation and implementation from binding to communication protocols. The
strategy pattern is usually implemented in a single address space using a single programming language. However, when faced with a kind of binding that is not bound to the same address space but manifested as a distributed system call or a remote procedure call, in order to fulfill the dynamic binding to the service provider either at compile time or at run-time, this pattern is called the Remote Service Strategy or Distributed Strategy.

Distributed systems require the ability to tap into the resources of the underlying communication protocol between the components that are distributed in different address spaces, but in the most seamless and transparent fashion possible. Web services standards have been particularly successful in providing open standards to enable this kind of fundamental and underlying communication to enable distributed system invocations, especially across high-speed telecommunication networks that are the underlying fabric of the World Wide Web.

**Service composition and flexibility**

In order to better understand how to achieve flexibility using SOA, let's explore the degrees of flexibility afforded through the notion of Service Composition.

But before doing that, let's talk about an important distinction that I think should be made among computation, composition, and collaboration. Computations are the fine-grained activities performed by using general purpose programming languages (such as Java™ or C#) that typically include mathematical calculations, Boolean evaluations, and performance of functions based on these evaluations. Composition languages, such as Piccola, aim at providing a small syntax with the goal of component assembly. Collaborations, on the other hand, are defined by flows (Business Process Execution Language for Web Services (BPEL4WS) or DAML-S, for example) which may be workflows or other forms of purposeful integration of component behavior to achieve a collective goal, for example, for supporting a business. Others have stressed the need for integration services to lead this composition. I'll cover this in more detail in the section on composition, below.

In the world of SOA, services reign supreme (in other words, the fundamental construct is the service, which can be composed into applications). Quality of Service (QoS) and flexibility are two of the most sought-after characteristics of applications developed with this architectural style. One way to enable business agility is through a flexible IT infrastructure built as SOA that performs well, is secure and reliable and, in general, meets the QoS requirements of the business. SOA provides flexibility primarily through its three first-class constructs: services, components (that realize the services), and flows (or processes).

Each of these three main SOA constructs provides a different kind of flexibility. For example, from a services angle, flexibility (and its "sister," reuse) is gained through separation of the interface from implementation and from binding to specific protocols. A special species of components, called enterprise components (see Resources), provide the realization of the service(s). They ensure QoS.

As an organization, and the industry in general, matures through its realization of the promise of SOA, applications will be built more and more out of services rather than being hard-wired from
components. This evolution happens when services are combined in a composition through some flow (process) mechanism. Orchestration and choreography are similar types of composition: loosely-coupled, standards-based (BPEL). However, early adopters can create compositions of services that involve a more hard-wired type of composition where some services (loosely-coupled units of business functionality) are combined with components to produce a realistically well-performing composition. This composition, however, will be less flexible than if the composition were a choreography of services. In this way, you can consider architecting SOA as a journey toward a dynamically re-configurable (DRC) style of architecture, Figure 1 shows.

**Figure 1. Increasing value gained from SOA as its usage matures within the organization**

Therefore, in a fully mature SOA, every unit of invocable functionality is a service, and these are woven together with the golden silk of choreography to create applications out of service composition.

**Composition**, then, is the process of building a larger structure through the assembly of smaller and, often, equally smart building blocks. A composition often manages the state of its services, similar to what a BPEL engine would do for its composed services.

But in the real world as it is seen today (versus what it is imagined to be or should be in the not-too-distant future), you can't always afford to create compositions out of loosely coupled choreography of services alone. In many cases, you have to make trade-offs between flexibility and QoS, so your resulting services, composite services, and the applications they embody or support will perform well and scale.

Let's continue to explore some of the common scenarios that are encountered in making decisions about those trade-offs.

**Composition and composability**

Let's set some ground rules by putting forth some relevant concepts and defining them. Figure 2 shows composition.
Figure 2. Service composition

![Service composition diagram](image)

The service composition may contain the component that provides a portion of functionality and which may aggregate services along with the component’s functionality to satisfy a business process. It can do this by creating a loosely coupled flow between loosely coupled services. But with the component, there is hard coupling, so quality of service is dependent on that of the weakest link: the service or component that provides the lowest quality of service. Thus, for those elements of a composition (services or components) that require very high performance, or for which there is another constraint such as dependency of other lines of business on this component, you should be comfortable in hard-coding its invocation within the composition. In time, with the continued maturing of tools and platforms, perhaps you can externalize the component, create a service interface to it, and call it as a proper service, with all the flexibility that that brings to the table.

Let’s move on and look at scenarios based on a banking example as I walk you through the realization phase of Service-Oriented Modeling and Architecture (SOMA -- see Resources). Here I will focus on service realization.

Service realization decisions

A question I commonly hear from my customers is: "Shall I expose a particular function as a service (flexible) or as a direct call to an existing component (tight coupling or possible better QoS)?" Let’s take a look at doing just that in Figure 3 below.

Let’s explore this question through a scenario that is actually pretty common: you have a bank with batch processing, legacy systems, multiple databases, and multiple applications for each line of business (such as government loans, retail loans, commercial loans).

Figure 3. Scenario context: New and old systems in fragile co-existence

![Scenario context diagram](image)
Descriptions of common types of composability

Figure 4 shows the various types of composability through examples. First let's see what these ellipses and lines mean. The ellipses indicate services (taken from the UML use-case icon) and the lines are flow of control and messages (roughly). Each service provides a distinct portion of functionality and has an associated quality of service associated with it that is accessible through the policy that governs or is assigned to it.

Let's take a closer look at the various types of composability:

**Substitution.** In this type, the business stakeholder can use the service within all processes where the functionality and service-level agreement of the service are required. Each of S(x) represents a service (functionality and quality of service).

**Consolidation or single-point of access.** Here, the service can be re-used as a single point of access for a recurrent business function, such as loan processing services for different customer types or a payment processing service.

In the simple case, consolidation or single point of access and substitution become more or less the same. Also recall that all of these are facets of composability that entail different design decisions on whether a given service is a "good service" or not -- in other words, should I recommend its funding to the business and how should I decide to realize this service? This question of the "goodness" of a service is answered by what is called the Service Litmus Tests, which I will discuss in a later part of this series.

Conducting a VOA for this bank shows that there are several loan processing systems overlapping in fundamental functionality, with this overlap being the commonality that is to be refactored out of these loan processing systems. In this case, a single loan processing service could effectively handle all loan processing requests to and from partners, customers, and organizations.

This would provide a single point of access for the disparate loan processing systems that can now be virtually merged and gradually sunset to reduce costs of maintaining redundant functionality. This virtualization provides an incremental capability afforded by the introduction of a service layer.
The services layer provides a service as a single point of access and can help the gradual physical consolidation of the implementers of that service over time. This consolidation is driven by budget availability and technological priorities, such as those of cost reduction, ease of maintenance, and decreased complexity. This is often achieved by the introduction of the Service Integrator pattern (Part 1 of this series -- also see Resources).

The pattern Service Integrator is made up of additional, more granular patterns, namely the service router (SR). This pattern can be used in conjunction with a Gateway (Web Services Gateway or ESB Gateway as the case may be) to trap and map requests to the appropriate destination service; deciding which system to send the request. Figure 6 shows this below. It then processes the results and provides the service consumer with the results of its operations. The Service Routher (SR) connects to the back-end systems or to other services. The SR can access the legacy directly through its underlying Service Realizer; in other words, the service component that realizes its functionality. This direct access from the underlying service component is recommended only in special cases where enabling access by wrapping each legacy system with a service adaptor is not possible. In either case, the service router supports the Service Integrator, which is now a single point of access for that business function or process.

**Figure 6. Service router 1 (service router) and 2 (hard-wired router)**

It is important to note that often a combination of these approaches may be the appropriate architectural decision if QoS issues sway you in that direction. Legacy systems, performance, and other QoS attributes will tend to be constrained to use the most optimal access mechanism: whether pure services, hard-wired, or hybrid.

A hybrid service router (which is used for Process Loan 3) is one in which parts of the connection are hardwired due to legacy constraints, while other parts are enabled for flexible invocation through services in SOA. This is often the case for access to packaged applications.

One of the consequences of this latter hybrid router approach is that probably you now cannot recompose the Process Loan 3 as easily and flexibly as if it were actually a service realized as a Web service.
**Re-composition or context change.** The business stakeholder can make a suggestion to re-compose the service and reuse it in other business processes or applications in order to meet new business goals and to support new changes to business processes.

**Figure 7. Re-composition or context change**

![Diagram](image)

**Self-containment.** In this situation, the architectural decision is around the question: can the service be deployed independently to meet a business goal, although it may cooperate with other services at run-time to perform business processes?

There are no implicit dependencies of the service on other services and all dependent ones are either replaceable or self-contained.

**Figure 8. Self-containment**

![Diagram](image)

If the service I am exposing is dependent upon a component which is itself relying on a database and a batch legacy system, then this service will bring these dependencies along with it. Ultimately, this contributes to increased maintainability and quality of service and helps avoid re-composing this service. In fact, it is often better to keep the scope of visibility of this service hidden behind a service router so that when and if the legacy or Independent Service Vendor (ISV) package is replaced, the interfaces to the service and the application composed using Process Loan 1, for example, do not have to be significantly altered, if at all.

**Business atomicity.** Is the service a software encapsulation of a single step in a business process? If you have done process decomposition and chosen a service based on such a business aligned analysis using Service-Oriented Modeling and Architecture, for example, then the answer is "yes." You can see this in **Figure 9.**
**Figure 9. A process decomposition**

![Diagram](image)

**Notions of composability**

Figure 10 introduces a simple notation. I have found this notation helpful in depicting the architecture and realization decisions about the coupling of services and components within the world of SOA. Block diagrams are often employed in conjunction with more formal notations, such as UML, to aid in understanding and communication between technical and non-technical communities and stakeholders; this is done while each is aware of the decisions made regarding coupling.

**Figure 10. Internal choreography and external dependency types**

![Diagram](image)

In Figure 10, you see Service C to Component C has a loosely-coupled realization connection; Service B has a tightly coupled connection to Component B.

**Re-configurability**

Run-time and dynamic re-configuration are characterized by the ability of underlying software architectures to rapidly respond and dynamically alter the functionalities of its components and services as well as the interconnection between them to suit the problem. They can then re-configure necessary operational aspects (also known as non-functional aspects) to maintain QoS and service-level agreements.

Practical considerations of performance, security, and interoperability make these models currently difficult to realize, although the advent of Web services standards in these areas and research
such as Grammar-Oriented Object Design (GOOD -- see Resources) are promising to alleviate many of the problems.

The above characteristics have been researched and implemented to a large extent within hardware-related systems for a number of years now. We would like to extend this capability to software systems and architectures. But in order to do so, we cannot take the paradigm from the hardware world and use it as is; software components and services are not, despite common (mis)conception, perfectly fitting building blocks; but rather are more like organs in the body which are larger conglomerations of more minute components and provide a distinct function and goal within a context. They also exhibit a certain degree of resilience to changes in their functional and operational capabilities.

Thus, the re-configurability of software services and components is a key success factor in achieving maximum flexibility and thus business agility. You can achieve this by exploiting Web services standards and applying novel methods through methods and techniques for composition, such as GOOD.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)