BPM and SOA

Mike Rosen
Chief Scientist
Wilton Consulting Group

Mike.rosen@wiltonconsultinggroup.com

Creating Flexible Services

SOA is founded on the idea of a service as the fundamental unit of design and construction of business solutions. Numerous articles have described the relationship of business services to business processes. From the BPM perspective, the value proposition is maximized when multiple different processes can be quickly constructed using a relatively stable and flexible service layer. Some articles have gone further to address the challenges of creating a flexible service layer at an enterprise scope, while meeting business requirements for specific processes. No matter how you achieve it, all of the SOA and BPM approaches assume the reuse of business services, and the ROI of services often depends on it.

Anatomy of a Service

Obviously, the more flexible a service is, the more likely it will be able to be used in different processes and scenarios. But before looking into some techniques to address this, let's review the basic structure of a service as illustrated below.

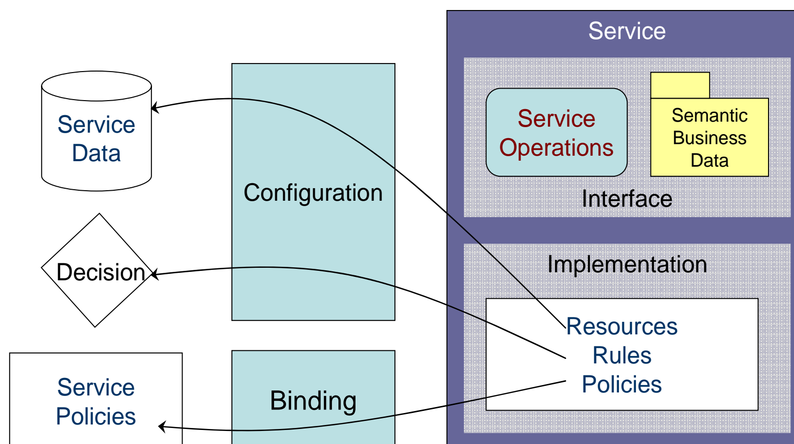

Figure 1. Basic Structure of a Service

Copyright © 2008 Mike Rosen. All Rights Reserved.

There are two main aspects to the service itself. The top part of the service in the diagram is the service interface; the bottom of the service is the service implementation. A service specifically separates the interface from the implementation.

The service interface specifies the service operations, i.e. what the service does, the parameters that are passed into and out of the operation, and the protocols for how those capabilities are used and provided. A service typically contains several different, but related operations. The service implementation is how the service provides the capabilities of its interface. The implementation may be based on existing applications, or on orchestrating other services together, or on code written specifically for the service, or all of the above.

The important point is that how the service is implemented is not visible to the consumers of the service, only what the service does is. The producer of a service is free to change the implementation of a service, as long as they don't change the interface or the behavior. For example, a new service might be completely based on existing functionality in a legacy application. Once the interface contract is finalized, consumers can start to use the service. In the meantime, the producer may create a new, modern implementation, and retire the old legacy application which runs on a platform that is no longer supported. Users (consumers) of the service may never notice the difference as long as the behavior and contract do not change.

There are also two different aspects to both the interface and implementation. These are the functions that are performed, and the information that it is performed on. In other words, a service is a combination of a set of functional service operations, and the corresponding semantic (virtual) business data that is passed into and out of the operations. Semantic business data is an abstraction of business entities (tied-to an enterprise schema) that are independent of data storage or implementation. The service operation signature describes the parameters that are passed in and out of an operation. The information model describes the structure and meaning of the semantic business data passed in and out

Flexible Service Design

When we think about some of the issues in reusing services, we quickly run into the problem that not every consumer of a service has the same configuration, is subject to the same business rules, has the same level of authorization and entitlement, or needs the same service level. So how do we go about making a service flexible enough to accommodate these different factors? Well, we use the time proven technique of indirection, or externalization. Rather than having these things hard coded in the service itself, we design the service to get the information remotely at the appropriate time. The previous figure also illustrates this technique.

In the figure we show that the service implementation requires certain resources and is subject to business rules and policies, all of which we want to be as flexible as possible in order to maximize the reusability of the service.

An external configuration mechanism, such as a file that is read as a service start up, or perhaps a configuration service, is used to point the service to the appropriate set of resources and rules to use. For example, if the service is for managing customer information, we might point it to the 'ACME Customer Database'. We could reuse the same service for the 'FOOBAR' company just by changing the configuration file. Or, perhaps more likely, we might initially partition the database according to customer name, with one partition applying to customers with last names starting with A-L, and a second partition for M-Z. As the customer base grows, we could repartition the database into thirds, A-I, J-R, S-Z, or change to a geographical partition rather than alphabetic. We wouldn't need to change the service for this, just the configuration parameters.

The externalization of rules gives us similar flexibility in two areas. Consider, for example, a service that decides whether or not to underwrite a particular insurance policy. If the rules for policy risk change, the changes are isolated in the lower level risk decision service, and don't

Copyright © 2008 Mike Rosen. All Rights Reserved.

affect the implementation of the underwriting service as a whole. Or, perhaps underwriting automobile insurance for commercial customers uses different rules than those for individual customers. We could potentially use the same service for both types of customers by configuring the underwriting service to use a different risk decision service. The example may seem trivial. Why not just put a conditional statement into the code? Wouldn't that be easier? Easier yes, better no. What happens when we add a third of fourth type of customer with a different risk profile? Simple, just change the configuration, not the code.

Finally, there are many decisions to be made regarding entitlements, authorization, qualities of service, etc. that may affect how a particular consumer of a service is connected to a specific provider. The service's run-time policy allows us to externalize these characteristics and then intelligently apply them at binding time. For example, 'Silver' customers may be given a basic level of service quality, while 'Gold' customers are given a higher level. Rather than hard code these policies in the service, we have the service read them from the externalized run-time policy and apply them. Now, when we decide that we have to add a 'Platinum' customer, with an even higher SLA, we don't need to rewrite the service, just create a new run-time policy.

Don't Do Too Much

This is probably a good time to interject a word of caution. One of the classic problems with other attempts at reuse (remember components and objects) was trying to come up with the perfect, flexible, generic function and interface up front. Of course we know that this didn't work. First, we couldn't really predict what the overall set of requirements and users of a component would be, and even if we could, we couldn't expect the party that was building the component to pay for the additional time or cost of adding functionality that they didn't require.

So with services, rather than trying to predict the future, we plan for the service to evolve. The first time we build the service, we do so to meet a specific set of requirements. But we don't do it in isolation. We do it in the context of our overall service catalog and roadmap. We build the service to be extensible and flexible using the techniques discussed in this report. Then, each new, additional user of the service will have additional requirements. So, we evolve the design and implementation which results in a new version of the service. (Note that the enhancements would typically be paid for by the additional requestor, which should be considerably less than building a new service). Over time, we may have to implement several different versions of the service, but each new user typically has fewer new requirements and takes less time to accommodate. Eventually, most new users can use the existing services as is.

So, to maximize the value of your SOA, make sure that services are designed and built based on a solid service implementation architecture. Then, build in flexibility mechanisms to accommodate change and enhance reuse. Finally, have a process for evolving services that matches your organizational realities. Certainly, these best practices are just the tip of the iceberg. For a more comprehensive discussion of SOA architecture and design, refer to the Author's new book: *Applied SOA: Service Oriented Architecture and Design Strategies.*

Copyright © 2008 Mike Rosen. All Rights Reserved.