Introduction to SOA governance

Governance: The official IBM definition, and why you need it

Bobby Woolf

July 2007
(First published June 13, 2006)

Explore how IBM defines Service-Oriented Architecture (SOA) governance—learn what it is and why it's critical to the success of your SOA project.

The need for governance and management

Service-Oriented Architecture (SOA) is a compelling technique for developing software applications that best align with business models. However, SOA increases the level of cooperation and coordination required between business and information technology (IT), as well as among IT departments and teams. This cooperation and coordination is provided by SOA governance, which covers the tasks and processes for specifying and managing how services and SOA applications are supported.

In this article, discover what governance and management are and why they're important. We'll then review the following important aspects of SOA governance:

- Service definition
- Service deployment life cycle
- Service versioning
- Service migration
- Service registries
- Service message model
- Service monitoring
- Service ownership
- Service testing
- Service security

With this information, you should have a good idea of why your SOA efforts need SOA governance. You will also have some ideas about how your organization can build its own SOA governance scheme.
Life without governance

Before we talk about what governance is, let's think about how things typically work in a corporate IT department; that is, how things work when there is no governance.

So you want to provide a service

Let's say you develop some nifty little service to convert a monetary amount in one currency into another currency. You need it in a couple of different places in an order-processing program you're working on, so you write the code as a reusable function you can invoke from anywhere in your program. You need it in another program, so you put the code in a Java .jar that you can add to the classpath of any program that needs it. But one problem with the service is that it takes a long time to start up because it needs to be initialized with currency-exchange rates, like those published on Yahoo's currency converter. This step takes too much time to initialize every time you need to convert one monetary amount. So you host your converter in its own little program that starts up, initializes, and is always running and can always be called from any of your programs through a remote API. Maybe the API is implemented as a SOAP-over-HTTP Web service, or maybe it's just a remote Enterprise JavaBeans (EJB) interface that supports RMI-over-IIOP.

What you've got now is a monetary currency converter service. Not only do a couple of your programs use it, but some of your coworkers in your department like it and start invoking it from their programs. Before long, unbeknownst to you, programs in other departments in your company you've never even heard of are using it. The converter is getting run so much that response times are slow, so you persuade your manager to buy a more powerful machine to host your service. The manager isn't happy about spending money from his budget for yet another machine that you told him is doing something simple, but you convince him.

One weekend, the machine crashes, and you find out about it because someone from your company that you never heard of calls you at home, asking that you come into the office to get the converter running again. Later, your manager says he's received complaints from another department about your converter not updating to the current exchange rates in a timely enough manner. He wants you to fix it, but doesn't want you to take time away from your real work. How'd you become responsible for all this?

One day you say, "To heck with this." You're not going to be responsible for the converter anymore, and you shut it down. Lots of complaint e-mails start circulating though the company from people trying to find out who's responsible for the converter and why it's not running anymore. Many of the e-mails complain that very valuable programs no longer run without the converter. Customers are angry, your company is loosing money -- and with any luck, when everyone finds out it was you, you may actually get fired. How did this happen?

So you want to consume a service

Now let's consider the flip side. You're a different employee in this company, working on a product catalog application. Users in different countries want to be able to see what the products cost in their currency. A coworker tells you about this service you can call that converts each product's
price to the user's currency. You try it out and sure enough it works, so you implement your application using it. Your manager is happy that you got the new feature working in record time, customers love the feature, and sales on your Web site improve remarkably.

Then one weekend, your Web site stops working: it can't display prices in other currencies. Your manager calls you at home and tells you to fix your program pronto. You call your coworker at home who tells you about the friend of a friend that told him about the service. Meanwhile, the assistant to the vice president of sales calls to inform you that customers are complaining. You tell him about the friend of a friend who apparently wrote the service. The assistant calls that guy and tells him to get into the office and get that service working again right now. Meanwhile, you’re in trouble because your catalog application stopped working even though there was nothing wrong with your application, just the service it was calling. How did this become your fault?

Welcome to the world of SOA governance. Or in this case, the lack of any effective governance. Both the provider of the service and the consumer of the service became responsible for a lot more than they bargained for. How do you use services without having them spin out of control like this?

**What is SOA governance?**

Now we've seen what can happen when governance is ineffective. So how would IT work better if its governance were better? First, we need to understand what governance is, and how it impacts IT and services.

In general, governance means establishing and enforcing how a group agrees to work together. Specifically, governance is the establishment of:

- Chains of responsibility to empower people
- Measurement to gauge effectiveness
- Policies to guide the organization to meet its goals
- Control mechanisms to ensure compliance
- Communication to keep all required parties informed

Governance determines who is responsible for making decisions, what decisions need to be made, and policies for making decisions consistently.

Governance is different from management. Governance plans for what decisions will need to be made, whereas management is the process of making and implementing the decisions. Governance sets policies, whereas management follows them.

IT governance is, well, governance for IT; namely: The application of governance to an IT organization, its people, processes and information to guide the way those assets support the needs of the business. SOA governance is a specialization of IT governance that puts key IT governance decisions within the context of the lifecycle of service components, services, and business processes. It is the effective management of this lifecycle that is the key goal of SOA governance.
IT governance is broader than SOA governance. IT governance covers all aspects of IT, including issues that affect SOA like data models and security, as well as issues beyond SOA like data storage and desktop support. SOA governance addresses aspects of the service life cycle such as: planning, publishing, discovery, versioning, management, and security.

Governance becomes more important in SOA than in general IT. In SOA, service consumers and service providers run in different processes, are developed and managed by different departments, and require a lot of coordination to work together successfully. For SOA to succeed, multiple applications need to share common services, which means they need to coordinate on making those services common and reusable. These are governance issues, and they're much more complex than in the days of monolithic applications or even in the days of reusable code and components.

As companies use SOA to better align IT with the business, they can ideally use SOA governance to improve overall IT governance. Employing SOA governance is key if companies are to realize the benefits of SOA. For SOA to be successful, SOA business and technical governance is not optional, it is required.

**SOA governance in practice**

In practice, SOA governance guides the development of reusable services, establishing how services will be designed and developed and how those services will change over time. It establishes agreements between the providers of services and the consumers of those services, telling the consumers what they can expect and the providers what they're obligated to provide.

SOA governance doesn't design the services, but guides how the services will be designed. It helps answer many thorny questions related to SOA: What services are available? Who can use them? How reliable are they? How long will they be supported? Can you depend on them to not change? What if you want them to change, for example, to fix a bug? Or to add a new feature? What if two consumers want the same service to work differently? Just because you decide to expose a service, does that mean you're obligated to support it forever? If you decide to consume a service, can you be confident that it won't be shut down tomorrow?

SOA governance builds on existing IT governance techniques and practices. A key aspect of IT governance when using object-oriented technologies like Java 2 Platform, Enterprise Edition (J2EE) is code reuse. Code reuse also illustrates the difficulties of IT governance. Everyone thinks reusable assets are good, but they're difficult to make work in practice: Who's going to pay to develop them? Will development teams actually strive to reuse them? Can everyone really agree on a single set of behavior for a reusable asset, or will everyone have their own customized version which isn't really being reused after all? SOA and services make these governance issues even more important and thus, their consequences even more significant.

Governance is more of a political problem than a technological or business one. Technology focuses on matching interfaces and invocation protocols. Business focuses on functionality for serving customers. Technology and business are focused on requirements. While governance gets involved in those aspects, it focuses more on ensuring that everyone is working together and that
separate efforts are not contradicting each other. Governance does not determine what the results of decisions are, but what decisions must be made and who will make them.

The two parties, the consumers and the providers, have to agree on how they're going to work together. Much of this understanding can be captured in a service-level agreement (SLA), measurable goals that a service provider agrees to meet and that a service consumer agrees to live with. This agreement is like a contract between the parties, and can, in fact, be a legal contract. At the very least, the SLA articulates what the provider must do and what the consumer can expect.

SOA governance is enacted by an SOA center of excellence (COE), a board of knowledgeable SOA practitioners who establish and supervise policies to help ensure an enterprise's success with SOA. The COE establishes policies for identification and development of services, establishment of SLAs, management of registries, and other efforts that provide effective governance. COE members then put those policies into practice, mentoring and assisting teams with developing services and composite applications.

Once the governance COE works out the policies, technology can be used to manage those policies. Technology doesn't define an SLA, but it can be used to enforce and measure compliance. For example, technology can limit which consumers can invoke a service and when they can do so. It can warn a consumer that the service has been deprecated. It can measure the service's availability and response time.

A good place for the technology to enforce governance policies is through a combination of an enterprise service bus (ESB) and a service registry. A service can be exposed so that only certain ESBs can invoke it. Then the ESB/registry combination can control the consumers' access, monitor and meter usage, measure SLA compliance, and so on. This way, the services focus on providing the business functionality, and the ESB/registry focuses on aspects of governance.

Governance can become a scapegoat for any ill in SOA. As with performance, governance may become an overwhelming concern and an excuse for every problem and justification for every questionable solution. All it takes is a single loaded comment tossed into any SOA discussion (which then becomes a rhetorical hand grenade), and you can watch all useful conversation grind to a halt. A challenge for SOA is using governance judiciously to make SOA work better without letting concerns about governance overwhelm everything else.

**Governance lifecycle and methodology**

Service development follows a lifecycle which IBM calls the SOA lifecycle. SOA governance also follows a lifecycle, the SOA governance lifecycle. These two lifecycles go together, run together, and are used together to produce an SOA composite applications and their services. The governance lifecycle produces a governance model which is used to manage the SOA lifecycle. The SOA governance lifecycle is shown in Figure 1.
IBM’s SOA Governance and Management Method (SGMM) is a full process for performing the SOA governance lifecycle so that governance can be applied to the SOA lifecycle. The four SGMM phases are:

- Plan Determine the governance focus
- Define Define the SOA governance model
- Enable Implement the SOA governance model
- Measure Refine the SOA governance model

A product to help perform the SOA governance lifecycle in general and SGMM specifically is the SOA Governance plug-in for IBM Rational Method Composer. IBM Global Business Services (GBS) can perform an IBM SOA/Web Services Center of Excellence engagement to help your organization establish an SOA center of excellence and develop SOA governance practices.

Aspects of SOA governance

SOA governance is not just a single set of practices, but many sets of practices coordinated together. These aspects of SOA governance each deserve discussion in greater detail in subsequent articles. This discussion is just a brief overview. More information about some of these aspects can be found in the References section at the conclusion of the article.

Service definition

The most fundamental aspect of SOA governance is overseeing the creation of services. Services must be identified, their functionality described, their behavior scoped, and their interfaces designed. The governance COE may not perform these tasks, but it makes sure that the tasks are being performed. The COE coordinates the teams that are creating and requiring services, to make sure needs are being met and to avoid duplicate effort.
Often, it is not obvious what should be a service. The function should match a set of repeatable business tasks. The service's boundaries should encapsulate a reusable, context-free capability. The interface should expose what the service does, but hide how the service is implemented and allow for the implementation to change or for alternative implementations. When services are designed from scratch, they can be designed to model the business; when they wrap existing function, it can be more difficult to create and implement a good business interface.

An interesting example of the potential difficulties in defining service boundaries is where to set transactional boundaries. A service usually runs in its own transaction, making sure that its functionality either works completely or is rolled back entirely. However, a service coordinator (a.k.a. orchestrator or choreographer) may want to invoke multiple services in a single transaction (ideally through a specified interaction like WS-AtomicTransactions). This task requires the service interface to expose its transaction support so that it can participate in the caller's transaction. But such exposure requires trust in the caller and can be risky for the provider. For example, the provider may lock resources to perform the service, but if the caller never finishes the transaction (it fails to commit or roll back), the provider will have difficulty cleanly releasing the resource locks. As this scenario shows, the scope of a service and who has control is sometimes no easy decision.

**Service deployment life cycle**

Services don't come into being instantaneously and then exist forever. Like any software, they need to be planned, designed, implemented, deployed, maintained, and ultimately, decommissioned. The application life cycle can be public and affect many parts of an organization, but a service’s life cycle can have even greater impact because multiple applications can depend on a single service.

The life cycle of services becomes most evident when you consider the use of a registry. When should a new service be added to the registry? Are all services in a registry necessarily available and ready for use? Should a decommissioned service be removed from the registry?

While there is no one-size-fits-all life cycle that is appropriate for all services and all organizations, a typical service development life cycle has five main stages:

1. **Planned.** A new service that is identified and is being designed, but has not yet been implemented or still being implemented.
2. **Test.** Once implemented, a service must be tested (more on testing in a moment). Some testing may need to be performed in production systems, which use the service as if it were active.
3. **Active.** This is the stage for a service available for use and what we typically think of as a service. It's a service, it's available, it really runs and really works, and it hasn't been decommissioned yet.
4. **Deprecated.** This stage describes a service which is still active, but won't be for much longer. It is a warning for consumers to stop using the service.
5. **Sunsetted.** This is the final stage of a service, one that is no longer being provided. Registries may want to keep a record of services that were once active, but are no longer
available. This stage is inevitable, and yet frequently is not planned for by providers or consumers.

Sunsetting effectively turns the service version off, and the sunset date should be planned and announced ahead of time. A service should be deprecated within a suitable amount of time before it is sunsettled, to programmatically warn consumers so that they can plan accordingly. The schedule for deprecation and sunsetting should be specified in the SLA.

One stage which may appear to be missing from this list is "maintenance." Maintenance occurs while a service is in the active state; it can move the service back into test to reconfirm proper functionality, although this can be a problem for existing users depending on an active service provider.

Maintenance occurs in services much less than you might expect; maintenance of a service often involves not changing the existing service, but producing a new service version.

**Service versioning**

No sooner than a service is made available, the users of those services start needing changes. Bugs need to be fixed, new functionality added, interfaces redesigned, and unneeded functionality removed. The service reflects the business, so as the business changes the service needs to change accordingly.

With existing users of the service, however, changes need to be made judiciously so as not to disrupt their successful operation. At the same time, the needs of existing users for stability cannot be allowed to impede the needs of users desiring additional functionality.

Service versioning meets these contradictory goals. It enables users satisfied with an existing service to continue using it unchanged, yet allows the service to evolve to meet the needs of users with new requirements. The current service interface and behavior is preserved as one version, while the newer service is introduced as another version. Version compatibility can enable a consumer expecting one version to invoke a different but compatible version.

While versioning helps solve these problems, it also introduces new ones, such as the need to migrate.

**Service migration**

Even with service versioning, a consumer cannot depend on a service -- or more specifically, a desired version of that service -- to be available and supported forever. Eventually, the provider of a service is bound to stop providing it. Version compatibility can help delay this "day of reckoning" but won't eliminate it. Versioning does not obsolete the service development life cycle, but it enables the life cycle to play out over successive generations.

When a consumer starts using a service, it is creating a dependency on that service, a dependency that has to be managed. A management technique is for planned, periodic migration to newer versions of the service. This approach also enables the consumer to take advantage of additional features added to the service.
However, even in enterprises with the best governance, service providers cannot depend on consumer migration alone. For a variety of reasons -- legacy code, manpower, budget, priorities -- some consumers may not migrate in a timely fashion. Does that mean the provider must support the service version forever? Can the provider simply disable the service version one day after everyone should have already migrated?

Neither of those extremes is desirable. A good compromise is a planned deprecation and sunsetting schedule for every service version, as described in service deployment life cycle.

**Service registries**

How do service providers make their services available and known? How do service consumers locate the services they want to invoke? These are the responsibilities of a service registry. It acts as a listing of the services available and the addresses for invoking them.

The service registry also helps coordinate versions of a service. Consumers and providers can specify which version they need or have, and the registry then makes sure to only enumerate the providers of the version desired by the consumer. The registry can manage version compatibility, tracking compatibility between versions, and enumerating the providers of a consumer's desired version or compatible versions. The registry can also support service states, like test and (as mentioned before) deprecated, and only make services with these states available to consumers that want them.

When a consumer starts using a service, a dependency on that service is created. While each consumer clearly knows which services it depends on, globally throughout an enterprise these dependencies can be difficult to detect, much less manage. Not only can a registry list services and providers, but it can also track dependencies between consumers and services. This tracking can help answer the age-old question: Who's using this service? A registry aware of dependencies can then notify consumers of changes in providers, such as when a service becoming deprecated.

IBM’s WebSphere Service Registry and Repository is a product for implementing service registries. It acts as a repository for service definitions, and registry for providers of those services. It provides a centralized directory for developers to find the services available for reuse, as well as use at runtime for service consumers and enterprise service buses (ESBs) to find providers and the addresses for invoking them.

**Service message model**

In a service invocation, the consumer and provider must agree on the message formats. When separate development teams are designing the two parts, they can easily have difficulty finding agreement on common message formats. Multiply that by dozens of applications using a typical service and a typical application using dozens of services, and you can see how simply negotiating message formats can become a full-time task.

A common approach for avoiding message format chaos is to use a canonical data model. A canonical data model is a common set of data formats that is independent of any one application and shared by all applications. In this way, applications don't have to agree on message formats,
they can simply agree to use existing canonical data formats. A canonical data model addresses the format of the data in the message, so you still need agreement around the rest of the message format -- such as header fields, what data the message payload contains, and how that data is arranged -- but the canonical data model goes a long way toward reaching agreement.

A central governance board can act as a neutral party to develop a canonical data model. As part of surveying the applications and designing the services, it can also design common data formats to be used in the service invocations.

**Service monitoring**

If a service provider stops working, how will you know? Do you wait until the applications that use those services stop working and the people that use them start complaining?

A composite application, one that combines multiple services, is only as reliable as the services it depends on. Since multiple composite applications can share a service, a single service failure can affect many applications. SLAs must be defined to describe the reliability and performance consumers can depend on. Service providers must be monitored to ensure that they're meeting their defined SLAs.

A related issue is problem determination. When a composite application stops working, why is that? It may be that the application head, the UI that the users interface with, has stopped running. But it can also be that the head is running fine, but some of the services it uses, or some of the services that those services use, are not running properly. Thus it's important to monitor not just how each application is running, but also how each service (as a collection of providers) and individual providers are also running. Correlation of events between services in a single business transaction is critical.

Such monitoring can help detect and prevent problems before they occur. It can detect load imbalances and outages, providing warning before they become critical, and can even attempt to correct problems automatically. It can measure usage over time to help predict services that are becoming more popular so that they can run with increased capacity.

**Service ownership**

When multiple composite applications use a service, who is responsible for that service? Is that person or organization responsible for all of them? One of them; if so, which one? Do others think they own the service? Welcome to the ambiguous world of service ownership.

Any shared resource is difficult to acquire and care for, whether it's a neighborhood park, a reusable Java framework, or a service provider. Yet a needed pooled resource provides value beyond any participant's cost: Think of a public road system.

Often an enterprise organizes its staff reporting structure and finances around business operations. To the extent that an SOA organizes the enterprise's IT around those same operations, the department responsible for certain operations can also be responsible for the development and run time of the IT for those operations. That department owns those services. Yet the services and
composite applications in an SOA often don't follow an enterprise's strict hierarchical reporting and financial structure, creating gaps and overlap in IT responsibilities.

A related issue is user roles. Because a focus of SOA is to align IT and business, and another focus is enterprise reuse, many different people in an organization have a say in what the services will be, how they will work, and how they'll be used. These roles include business analyst, enterprise architect, software architect, software developer, and IT administrator. All of these roles have a stake in making sure the services serve the enterprise needs and work correctly.

An SOA should reflect its business. Usually this means changing the SOA to fit the business, but in cases like this, it may be necessary to change the business to match the SOA. When this is not possible, increased levels of cooperation are needed between multiple departments to share the burden of developing common services. This cooperation can be achieved by a cross-organizational standing committee that, in effect, owns the services and manages them.

**Service testing**
The service deployment life cycle includes the test stage, during which the team confirms that a service works properly before activating it. If a service provider is tested and shown to work correctly, does the consumer need to retest it as well? Are all providers of a service tested with the same rigor? If a service changes, does it need to be retested?

SOA increases the opportunity to test functionality in isolation and increases the expectation that it works as intended. However, SOA also introduces the opportunity to retest the same functionality repeatedly by each new consumer who doesn't necessarily trust that the services it uses are consistently working properly. Meanwhile, because composite applications share services, a single buggy service can adversely affect a range of seemingly unrelated applications, magnifying the consequences of those programming mistakes.

To leverage the reuse benefits of SOA, service consumers and providers need to agree on an adequate level of testing of the providers and need to ensure that the testing is performed as agreed. Then a service consumer need only test its own functionality and its connections to the service, and can assume that the service works as advertised.

**Service security**
Should anyone be allowed to invoke any service? Should a service with a range of users enable all users to access all data? Does the data exchanged between service consumers and providers need to be protected? Does a service need to be as secure as the needs of its most paranoid users or as those of its most lackadaisical users?

Security is a difficult but necessary proposition for any application. Functionality needs to be limited to authorized users and data needs to be protected from interception. By providing more access points to functionality (that is, services), SOA has the potential to greatly increase vulnerability in composite applications.

SOA creates services that are easily reusable, even by consumers who ought not to reuse them. Even among authorized users, not all users should have access to all data the service has access
to. For example, a service for accessing bank accounts should only make a particular user's accounts available, even though the code also has access to other accounts for other users. Some consumers of a service have greater needs than other consumers of the same service for data confidentiality, integrity, and nonrepudiation.

Service invocation technologies must be able to provide all of these security capabilities. Access to services has to be controlled and limited to authorized consumers. User identity must be propagated into services and used to authorize data access. Qualities of data protection have to be represented as policies within ranges. This enables consumers to express minimal levels of protection and maximum capabilities and to be matched with appropriate providers who may, in fact, include additional protections.

Summary: Governance critical to SOA success

This article shows you why SOA governance is critical to an enterprise's success with SOA. Governance involves establishing responsibilities and empowering responsible parties, whereas management involves making sure the governance policies actually occur. Technology can be used not to set governance, but to perform management. Governance that is managed during service invocation can be effectively managed by an ESB, simplifying the responsibilities of both the providers and consumers.

SOA governance has many aspects, such as:

- Service definition (the scope, interface, and boundaries of a service)
- Service deployment lifecycle (the lifecycle stages)
- Service versioning (including compatibility)
- Service migration (deprecation and sunsetting)
- Service registries (dependencies)
- Service message model (canonical data models)
- Service monitoring (problem determination)
- Service ownership (corporate organization)
- Service testing (duplicated testing)
- Service security (including ranges of acceptable protection)

Adequate treatment of each of these aspects could become articles unto themselves.

Acknowledgements: The author would like to extend his thanks to fellow IBM colleagues for their input into this article: Steve Graham, Arnauld Desprets, Randy Langel, Kerrie Holley, Ali Arsanjani, Emily Plachy, Bob Vanorsdale, Jon Richter, Mandy Chessell, Mark Cocker, Mark Ernest, Steven Adler, and Fill Bowen.

References

The following is a list of some useful references and resources.

SOA governance

- Get more details about SOA governance in the article, "A case for SOA governance," by Tilak Mitra (IBM developerWorks; August 2005).
- SOA governance and the prevention of service-oriented anarchy.
- IBM SOA/Web Services Center of Excellence (PDF).
- Read the article, "Increase flexibility with the Service Integration Maturity Model (SIMM)," by Ali Arsanjani and Kerrie Holley (IBM developerWorks; September 2005).

### Service versioning

- "Best practices for Web services versioning," by Kyle Brown and Michael Ellis (IBM developerWorks; January 2004)

### Service registries

- WebSphere Service Registry and Repository.
- Introducing IBM WebSphere Service Registry and Repository.
- "Choose an ESB topology to fit your business model," by Chris Nott and Marcia Stockton (IBM developerWorks; March 2006)

### Service message model

- Enterprise Integration Patterns, by Gregor Hohpe and Bobby Woolf (Addison-Wesley; 2003)

### Service ownership

- "SOA programming model for implementing Web services, Part 10: SOA user roles," by Mandy Chessell and Birgit Schmidt-Wesche (IBM developerWorks; February 2006)

### Service security

- "Services security with WebSphere Application Server V6, Part 1: Introduction to security architectures," by Tony Cowan (IBM developerWorks; April 2006)