zapthink
white paper

DESIGN & VALIDATE SOA IN A HETEROGENEOUS ENVIRONMENT

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

DESIGN & VALIDATE SOA IN A HETEROGENEOUS ENVIRONMENT

THE FOUNDATIONS OF SOA

July 2008

Analyst: David Linthicum

Abstract

The promised business value of Service-Oriented Architecture (SOA) is driving many enterprises to purchase key SOA-related technologies, such as an Enterprise Service Bus (ESB), in advance of the forthcoming architecture. However, the road to SOA is not always clear, and many enterprise architects, developers, and project leaders require guidance to understand the steps to SOA success. Such guidance must include clear, interactive, and adaptive process that ensures that team members understand all requirements, document all details, and plan all testing and implementation tasks.

The purpose of this manual is to walk those charged with designing and building their SOA implementation through the issues, steps, and procedures they require to create their first iteration of SOA, and how to build on that experience to drive a more systemic change within their enterprise. We're going to take you through this journey assuming that you have the technology on hand, whether that is an integration server, application server, or an ESB, and focus on the best practices and architectural patterns for that particular category of technology. Since SOA is by nature heterogeneous, strategies for resolving architectural challenges earlier in the lifecycle using well-defined semantics will be discussed.

Systemic to this manual is how to approach testing in the context of SOA. While many consider testing as something that is at the end of the process, it's actually a part of each step. Thus, as part of this manual we will teach you about testing approaches and key enabling technology.

I. The Value of SOA

Typical SOA implementations deliver a strategic framework of technology that allows all relevant systems, inside and outside of an organization, to expose and access well defined services, and information bound to those services, in order to abstract them as business services in support of orchestration layers and/or composite applications for solution development.

The primary benefits of SOA include reuse, agility, visibility, and extended reach.

The primary benefits of SOA include reuse, agility, visibility, and extended reach.

Reuse of services/behaviors, or the ability to leverage application behavior from application to application without a significant amount of re-coding or integration. In other words, using the same application functionality (behavior) over and over again, without having to port the code…leveraging application behavior in a location independent manner.

Agility, or the ability to change business processes on top of existing services and information flows, quickly, and as needed to support a changing business.

Visibility, or the ability to monitor points of information and points of service, in real time, to determine the well being of an enterprise or trading community. Moreover, the ability to adjust processes for the benefit of the organization in real time.

Extend reach, or the ability to expose certain enterprises' processes to other external entities for the purpose of inter-enterprise collaboration or shared processes. This benefit is in essence the next generation of supply chain integration for information.

The notion of SOA is not at all new. Attempts to share common processes, information, and services have a long history, one that began more than ten years ago with multi-tier client/server—a set of shared services on a common server that provided the enterprise with the infrastructure for reuse and now provides for integration—and the distributed object movement. A common set of services among enterprise applications invites reusability and, as a result, significantly reduces the need for redundant application services.

II. Evaluating your Existing Technology

So, what do you have lying around that will allow you to implement SOA? Typically, many enterprises have already made an investment in technology, and thus need to consider the use of that technology within the context of SOA as an option. You must still consider functional fit, and just because you already own a particular technology does not mean you should leverage it.

Let's examine a few types of SOA-compliant technology that may already exist within your organization.

So, you have an Integration Server

Integration servers, such as those that have morphed from the Enterprise Application Integration (EAI) days, have many common features, including:

- Schema/structure transformation
- Intelligent routing
- Rules processing

- Message warehousing
- Flow control
- Repository services
- Directory services
- Management
- APIs and adapters

The ability of integration servers to leave systems "where they are"—minimizing change while allowing for the sharing of data—lends tremendous value to SOA where integration is a core design pattern. Integration servers more closely resemble the way many business processes actually work—providing greater efficiency and flexibility by automating functions currently performed manually, functions such as sending sales reports through inter-office mail or walking data down the hall on disk. In essence, they automate how all of these internal systems or processes communicate one to another, and thus solve the integration challenge that underlies SOA.

Integration servers indeed have a place in SOA.

So, while more adapted for traditional EAI, integration servers do solve the integration issues that are an early SOA challenge. Thus, they may indeed have a place in the architecture. The trick is to figure out the proper fit, using the requirements of the architecture (see the procedure we're suggesting below).

When considering SOA, integration servers have the following supporting features:

1. Service externalization
2. Core information mediation capabilities
3. Management of information flow
4. Support for registry and repository
5. Connector, adapters, and APIs.

So, you have an Application Server

Application servers not only provide a location for application logic and interface processing, they also coordinate many resource connections, and are able to provide service externalization and service creation. Application servers also serve as tools to create Service-Oriented Business Applications, or SOBAs. SOBAs are in essence composite applications providing a place to create applications that leverage multiple services, brought together to form solutions.

Application servers take many existing enterprise systems and expose them through a single user interface, typically a Web browser. For example, application servers can easily externalize information contained in mainframes, ERP applications, and even middleware without a user interface. As a result, developers can gain all the application development capabilities they require, including a programming language and an integrated development environment. This capability makes application servers ideal for SOBA development.

Where there are strengths, there are also weaknesses. Application servers are not strong in providing back-end integration or application-to-application integration, when the information is rarely externalized through a user interface. This weakness is a direct result of the need to code all parts of the information extraction, transformation, and update process, typically through traditional transaction semantics.

Many application servers support rudimentary process integration capabilities.

However, many application servers do support rudimentary process integration capabilities. In some instances they support orchestration services through an embedded Business Process Execution Language (BPEL) engine, or a proprietary process integration engine. Indeed, over the years many application server providers, including IBM, BEA Systems, and Oracle, have added functionality useful for SOA, including integration services, messaging services, and core process integration features.

When considering SOA, application servers have the following supporting features:

1. Service externalization
2. Service development
3. Some core integration capabilities
4. Support for registry and repository
5. SOBA development
6. Some process integration capabilities

So, you have an ESB

ESBs provide an abstraction layer on top of an implementation of an enterprise messaging system, which allows integration architects to exploit the value of messaging through the use of services. For our purposes, you can consider ESBs as messaging systems that are able to internalize and externalize information via services, where in the past these message queuing systems leveraged proprietary APIs.

There is no single definition of ESB, thus many middleware systems calling themselves an ESB have very different patterns from vendor to vendor.

Moreover, it's important to note that there is no single definition of ESB, thus many middleware systems calling themselves an ESB have very different patterns from vendor to vendor. Thus, you need to check in with your vendor as to what capabilities are available.

The common use of ESB is as an information message and movement engine where the information flows from one peer to another, using a queuing type of infrastructure. Some ESBs go beyond basic queuing, providing many of the services found in traditional integration servers and application servers (see above), including transformation, routing, flow control, and even process integration and orchestration.

While many ESB vendors sell themselves as complete SOA solutions, how complete they are for your enterprise depends on your requirements and needs within the architecture. What is true is that in some instances ESBs provide a good starting point for SOA implementations, allowing you to implement basic features and thus prove the value in an incremental way, but in many other cases, organizations make the fatal mistake of purchasing an ESB without any architecture, and then are left disillusioned when their SOA initiative founders.

When considering SOA, ESBs have the following supporting features:

1. Service externalization
2. Some core integration capabilities
3. Message processing and management
4. Support for registry and repository
5. Some process integration capabilities

6. Connectors, adapters, and APIs.

III. Defining Your Domain

So, how do you implement SOA? First, you need to define the domain within the enterprise in which you're going to work. It is usually best to implement SOA in small iterations, such as moving a single division, or portion of a division, to SOA, if needed, instead of an entire enterprise all at once. Small successes lead to larger more strategic successes over time, and you need to establish the demarcation lines at the beginning of the project to provide better focus and understanding.

There are really not hard and fast rules here. However, as a rule of thumb your domain should be less than 6 systems, and deal with less than 300 core services. To better understand how to define your domain, you may want to go through a complexity analysis: in essence counting the function points of all participating systems.

Also, it's helpful at this stage to define the testing domains, meaning functional divisions within the architecture where isolated testing can occur. Typically this means service, process, and persistence tiers, and perhaps subdivided domains on top of that. We're doing this to logically understand how we will divide the architecture up for testing, and then divide those domains into subcomponents. Moreover, this is also a good time to create initial strategy documents around testing those domains at a conceptual level.

IV. Defining Your Approach to SOA Governance

SOA governance is one of those topics that means different things to different people in the world of SOA. However, we can define it as an emerging discipline which enables organizations to provide guidance and control of their SOA initiatives and programs within the context of their IT governance processes, as well as leveraging SOA for better IT governance overall. Drilling down a bit there are really two flavors emerging: design time and runtime. It's important to understand the differences, and that you may indeed need two SOA governance products at the end of the day.

Design Time SOA governance, as the name implies, typically provides an integrated registry/repository that attempts to manage a service from its design to its deployment, but typically not during runtime execution of the services, albeit some do.

Key components of design time SOA governance include:

1. A registry and/or repository for the tracking of service design, management, policy, security, and testing artifacts.
2. Design tools, including service modeling, dependency tracking, policy creation and management, and other tools that assist in the design of services.
3. Deployment tools, including service deployment, typically through binding with external development environments.
4. Links to testing tools and services, providing the developer/designer the ability to create a test plan and testing scenarios, and then leverage service testing technology.

In essence, design time SOA governance works up from the data to the services, gathering key information as it goes. Thus, you typically begin by defining the underlying data schema, and turning that into metadata, and perhaps an abstraction of the data. Then working up from there, you further define the services that interact with the data, data services, and then transactional services on top of that. You can further define that into processes or orchestration. All this occurring, with design time information managed within the design time SOA governance system.

Runtime SOA governance works and plays in the world of SOA management, and should be linked with design time SOA governance, but often is not. Thus we have design time, which is all about defining the policies that need to be enforced by the services and implemented by the consumer that's going to consume the services. Therefore, runtime governance is the process of enforcing and implementing those policies at service runtime, but may do other things as well.

Runtime SOA governance typically supports:

1. Service discovery
2. Service delivery
3. Security
4. Setting and maintaining appropriate service levels
5. Managing errors and exceptions
6. Enabling online upgrades and versioning
7. Service validation
8. Auditing and logging

In creating a SOA governance approach you need to define your run time and design time SOA governance strategy, inclusive of approaches you wish to employ and how the SOA governance infrastructure will service the design, creation, testing, deployment, and management of the services and thus the architecture.

You should define a testing and quality strategy around your SOA governance planning.

Again, you should define a testing and quality strategy around your SOA governance planning. You need to consider SOA quality management approaches and tools that work and play well with your SOA governance strategy and enabling technology. This means allowing the tester to interact with and store tests as validation points within the SOA governance tools you select.

Moreover, most SOA governance systems leverage policies, and like any other SOA component should be tested. Part of this process is to identify which policies should be tested, how they are to be tested, and how to leverage the SOA testing technology.

V. Defining Your Approach to SOA Testing

When dealing with SOA testing, you need to learn how to invoke, then verify intended results across technologies, ensuring that things work at the service, persistence, and process layers. The foundation of SOA testing is to select the right tool for the job, having a well thought out plan, and spare no expense in testing cycles – or else risk that your SOA implementation will not meet the business requirements set out for it.

Within the SOA abstraction, services are the building blocks, and are found at the lowest level of the stack. Services become the base of SOA, and while some abstract existing legacy services, others are new and built for specific purposes.

Moving up the stack, we then find composite services, or services made up of other services, and all services abstract up into the business process or orchestration layer, which provides the agile nature of SOA since you can create and change solutions using a configuration metaphor. Also, it's noteworthy that, while many of the services tested within SOA will be Web service-based, there are still underlying technologies where the integration and business logic resides such as ESB messaging queues. SOA by nature leverages existing technologies, from databases, CORBA, J2EE, and often custom or proprietary approaches, so validation needs to happen at all of these relevant layers.

When testing services, you need to keep the following in mind:

Services are not complete applications or systems, and shouldn't be tested with that expectation.

- Services are not complete applications or systems, and shouldn't be tested with that expectation. They are a small part of an application. Nor are they subsystems; they are small parts of subsystems as well. Thus, you need to test them with a high degree of independence, meaning that the services are both able to properly function by themselves, or as a part of a cohesive system. Indeed, services are more analogous to traditional application functions in terms of design, and how they are leveraged to form solutions, whether fine or coarse-grained.
- The best approach to testing services is to first list the use cases for those services. At that point you can design testing approaches for that service including testing harnesses or extensions, and the use of SOA tests that document the use case internally such as iTKO LISA. You also need to consider any services that the service may employ, and thus be tested holistically as a single logical service. In some cases you may be testing a service that calls a service, that in turn calls another service, where some of the services are developed and managed in house, and some of them exist on remote systems that you don't control. You should consider all use cases and configurations.
- You should be able to test services with a high degree of autonomy. If it is possible to test them in isolation of their dependencies, they can be tested as independent units of code using a single design pattern that fits within other systems that use many design patterns. However, in real world integrations this is often not the case, and the service has several dependent services and components that need to be accounted for. Virtualization of the behaviors of services, as well as the "environment" of SOA can alleviate some of these constraints.
- Services should have the appropriate granularity. Don't focus on too fine-grained or too coarse-grained. Focus on that correct granularity for the purpose and use within the SOA. Here the issues related to testing are more along the lines of performance than anything else. Too fine-grained services have a tendency to bog down due to the communications overhead required when dealing with so many services. Too coarse-grained, and they don't provide the proper autonomic values to support their reuse. You need to work with the service designer on this one.

VI. Create a Semantic-, Service-, and Process-Level Understanding

Next, you need to focus on the core set of requirements for your SOA initiative, which means a complete semantic-, service-, and process-level understanding of the domain you previously selected.

You can't deal with information you don't understand, including information bound to services.

Understand all application semantics in your domain. You can't deal with information you don't understand, including information bound to behavior (services). Thus, it is extremely important for you to identify all application semantics—metadata, if you will—that exist in your domain, thus allowing you to properly deal with those data.

The understanding of application semantics establishes the form in which the particular application refers to properties of the business process. For example, the very same customer number for one application may have a completely different value and meaning in another application. Understanding the semantics of an application guarantees that there will be no contradictory information when the application is integrated with other applications at the information or service levels. Achieving consistent application semantics requires an application integration "Rosetta Stone" and, as such, represents one of the major challenges to implementing SOA.

Defining application semantics is a tough job since many of the existing systems you'll be dealing with are older, proprietary, or perhaps both. The first step in identifying and locating semantics is to create a list of candidate systems. This list will make it possible to determine which data repositories exist in support of those candidate systems.

Any technology that can reverse-engineer existing physical and logical database schemas will prove helpful in identifying data within the problem domains. However, while the schema and database model may give insight into the structure of the database or databases, they cannot determine how that information is used within the context of the application or service. That's why we need the next several steps.

For quality assurance purposes it's helpful to create a high level strategy for both understanding and testing the metadata within the SOA. Typically, this will become the physical and abstract database layers for the architecture, and several testing issues should be explored, including:

- Creation of high level service level agreements for data access.
- An understanding of the database model to be tested, and a strategy for testing it (e.g., relational, object, XML, others).
- Monitoring capabilities and integration with SOA testing tool.

Understand all services available in your domain. Service interfaces are quirky. They differ greatly from application to application, and may be custom or proprietary. What's more, many interfaces, despite what the application vendors or developers may claim, are not really service interfaces at all, and you need to know the difference. Services provide behavior as well as information, thus they are service-oriented. There are some services that just produce information; those are information-oriented and should not be included in this step. We are only interested in the former at this point.

It is important to devote time to validating assumptions about services, including:

- Where they exist.
- The purpose of the service.

- Information bound to the service.
- Dependencies (e.g., if it's a composite service).
- Security issues.

The best place to begin with services is with the creation of a services directory. As with other directories, this is a repository for gathered information about available services, along with the documentation for each service, including what it does, information passed to a service, information coming from a service, etc. Use this directory, along with the now-understood application semantics, to define the points of integration within all systems in the domain.

As with the metadata defined previously, it's helpful to define any issues around service testing here, including:

- The enabling technology of the service.
- Service level agreements.
- Dependencies on the database, or other services.

Understand all processes in your domain. You need to define and list all business processes that exist within your domain, either automated or not. This step is important because, now that we know which services and information sources and sinks are available, we must define higher level mechanisms for interaction, including all high-level, mid-level, and low level processes. In many instance, these processes have yet to become automated or are only partially automated.

For example, if an application integration architect needs to understand all the processes that exist within an inventory application, he or she will either read the documentation or the code to determine which processes are present. Then, the architect will enter the business processes into the catalog and determine the purpose of the process, who owns it, what exactly it does, and the technology it employs (e.g., Java or C++). These processes are later bound to new processes—or, metaprocesses—providing orchestration of encapsulated processes or services to meet some business need.

You should also consider the notion of shared versus private processes. Some processes are private, and thus not shared with outside entities (or, in some cases, they are not even shared with other parts of the organization). Other processes are shared, meaning that others leverage the same processes in order to automate things inter-enterprise. Private and shared processes can exist in the same process space with the process integration technology managing security among the users.

For the purposes of testing, you can consider all processes in the domain as services, and in essence these have many of the same issues when it comes to testing. However, processes have a tendency to leverage many more services per process, than per composite. Thus, the test approaches should be adjusted accordingly. Moreover, you need to consider the location where the processes execute (e.g., BPEL engine), and consider that within the testing strategy and integration with the testing tools.

VII. Developing Core Services

When building core services, keep in mind:

Each service you have the opportunity to design ideally has a specific purpose, and is not dependent upon other services. This makes it easier to abstract services into composite applications, in essence, leveraging these services as if they are functions local to the composite. Furthermore, as we stated above, services should exist with a high degree of autonomy. They should execute without dependencies, if at all possible. This allows you to leverage the service by itself, and design the service with this in mind, no matter how coarse- or fine-grained the service is.

Service Design Principles

So, how do you design a service? First, it's important to follow a few basic principles. While following these principles does not ensure success, it will send you down the right path.

First and foremost, many services should be designed for ***reuse***. Services become a part of any number of other applications, and thus must be designed to provide behavior and information, but not be application specific. This is a difficult paradigm for many developers since custom one-off software is what they've been doing for most of their careers. Thus, the patterns must be applicable to more than a single problem domain or application, meaning you must have use for your reusable service. Without this, the exercise is in vain.

In addition, services have to be designed for ***heterogeneity***. Web services should be built so that there are no calls to native interfaces or platforms. This requirement is due to the fact that a Web service, say, one built on Linux, may be leveraged by applications on Windows, Macs, and even mainframes. Those that leverage your service should do so without regard for how it was created, and should be completely platform independent.

Among other benefits, ***abstraction*** allows access to services from multiple, simultaneous consumers; hiding technology details from the service developer. The use of abstraction is required to get around the many protocols, data access layers, and even security mechanisms that may be in place, thus hiding these very different technologies behind a layer that can emulate a single layer of abstraction.

Also, when we build or design services we need to account for ***aggregation***. Many services will become parts of other services, thus composite services leveraged by an application, and you must consider that in their design. For instance, a customer validation service may be part of a customer processing service, which is part of the inventory control systems. Aggregations like SOBAs are clusters of services bound together to create a solution.

Services are not applications and should have ***limited scope***, as we discussed above. In other words, they do simple things such as check inventory or calculate reorder points. If your needs are more complex, you simply write more services instead of overloading a single service with too much functionality. Services with too much functionality are considered heavy, and are difficult to reuse since you may deploy a service where you're only leveraging 10 percent or less of its functions.

Finally, services should be ***standards-based***. While in the world of Web services this seems like a no-brainer, many developers and architects ignore compliance with standards and thus limit interoperability.

Steps to Service Design

So, now that we understand the common design patterns we must follow, the question is: How does one design a service? Also, what tools are available?

There are certain steps architects and developers can follow. Here are some suggestions, assuming new service design.

1. You need to define the purpose of the service. What will the service do, and who is the intended user; human, application, and other services?
2. You need to determine the information to be bound to the service, including both metadata and schemas. This means you need to understand how information is leveraged by the service, and what functions require what data.
3. You need to determine the functions (methods) encapsulated inside the service; in other words, the behaviors you would like to expose. Also at this step we define each function, including how the function breaks down using a traditional functional decomposition chart.
4. You need to define any interfaces into the service; both machine and human. This means we need to determine how the service will interact with the calling applications, and through what mechanisms.
5. You need to define how the service is to be tested, using the suggestions above. This is a very important but often neglected step where you define how those leveraging the service will test the service within the context of their usage pattern. You need to define test information, service invocation, and validity of results.

For testing purposes, the design of the service should feed into the test planning.

For testing purposes, the design of the service should feed into the test planning, including issues such as:

- The quality of the service design.
- The development technology leveraged to build the service.
- Integration with other components.
- Service granularity.
- Service composites.
- Links with the underlying data, and a process engine.

VIII. Service Externalization

So, how do you approach service design for services that are externalized from existing interfaces? It's really a matter of understanding, defining, and designing.

First, you need to figure out what you're dealing with. There are many types of interfaces that can potentially become services. For instance, APIs, user interfaces, transactions, and direct-to-hardware interfaces (e.g., impeded interfaces). Each type if interface requires different approaches, techniques, and enabling technology. Here, it's best to work back from the interface to a level of abstract data and behavior.

Next, you need to define the interface at a high level. Typically, they produce or consume information, but may provide more direct access to behavior as well. Thus, define the structure of the information, and the functions that act upon the data. In other words, what are the raw structures, and what behaviors are externalized along with the data?

Once you understand the mechanisms to deal with the interface, and the meta-service information (data and behavior) around it, it's time to design the service. You'll find that the interface mechanisms really control how your services are

defined, designed, and implemented. For instance, it's difficult to normalize services that are bound to interfaces...one interface-one service. Therefore, it's difficult to break apart the services as autonomous units since they in essence reflect an abstraction of the interfaces, and are indeed tightly coupled to the interfaces.

Virtualized Service Endpoints and Virtual Services

Virtualization, for the purposes of this guide, refers to the abstraction of computer resources, in this case services. Thus, we are hiding the physical characteristics of computing resources from the way in which other systems, applications, or end users interact with those resources. This includes making a single physical service appear to function as multiple logical resources; or it can include making multiple services appear as a single logical resource.

Virtual Endpoints allow the SOA to define virtual locations for Services that need to be invoked, when in fact you're completely shielded from the actual end point of the service itself. What's core to this concept is the dynamic processes inherent in SOA applications, as the physical address (or URL) of a Service may need to change depending upon when and how it is used. Often the Virtual Endpoint and necessary lookup information is stored as a definition in a centralized UDDI Registry, so the address can be routed according to the specific needs of the workflow.

Virtual Services are very useful when considering "skip testing" while you are in development against a changing environment. Virtual Services simulate the functional responses, and performance characteristics of a Service and its underlying implementation, preventing the design, development and testing teams from needing access to live or completed versions of Services.

Since the endpoints and the services are virtual, all of the services in a process under test do not have to be active for the test to complete. This technology is able to route those test steps to the Virtual Service versions by looking up the endpoint.

Since the endpoints and the services are virtual, all of the services in a process under test do not have to be active for the test to complete. This technology is able to route those test steps to the Virtual Service versions by looking up the endpoint. Thus, failures, lack of access or service outages do not stop testing.

Defining Performance and SLAs

When defining the performance of the services and the associated service-level agreement (SLA), we're just putting expectations of performance into a contract, which creates a clear understanding between the producer and the consumer of the services as to how the service will perform in production.

Testing, Validation, and Performance Validation

As we discussed in the in our description around SOA testing above, here is the step where you carry out the complete validation activities, leveraging testing tools such as iTKO LISA.

Service validation, is the process of validating that the services are in good form, and living up to the expectations of the design.

Performance validation is the process of validating that the services live up to the performance expectations set forth in the SLA.

Operational and Runtime Activities

As we covered above when discussing runtime SOA governance, during runtime and operational activities you need to consider:

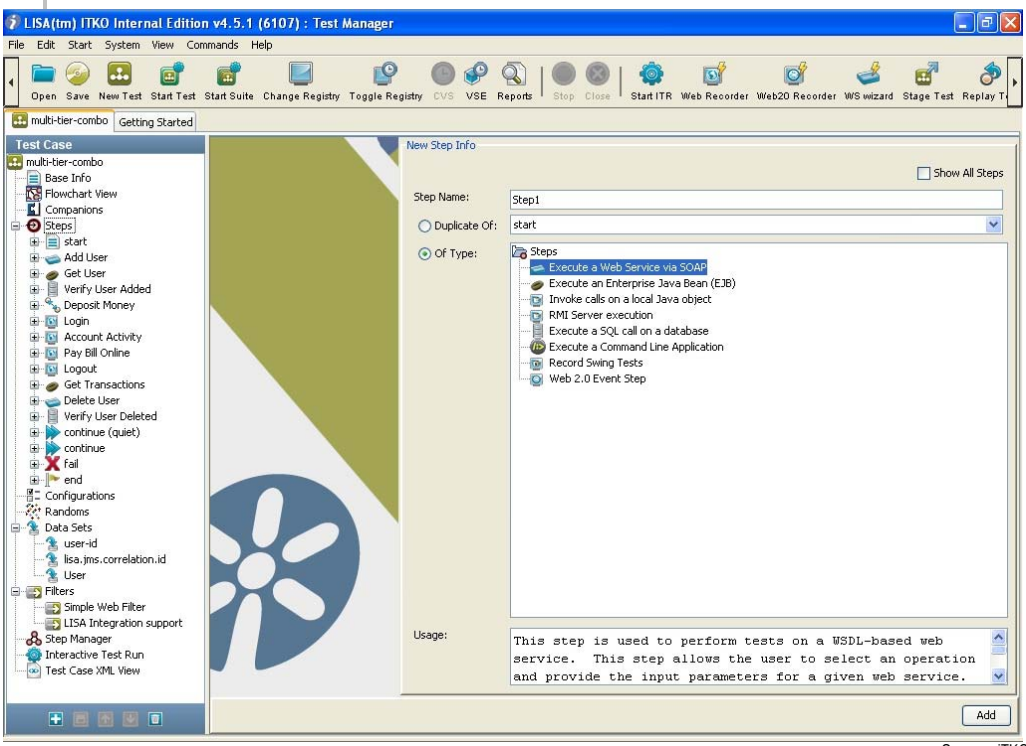
- Security, ensuring that consumers of the service are authenticated and inter-service workflows are interacting safely.
- Setting and maintaining appropriate service levels, referring to the SOA application's ability to live up to the SLAs.
- Managing errors and exceptions, referring to the ability to manage through service and system errors, as well as exceptions that are created.
- Enabling online upgrades and versioning, referring to the ability to update services during runtime without stopping production.
- Auditing and logging, or keeping track of the system during production.

IX. Leveraging iTKO LISA for Service Virtualization

LISA is a SOA Testing, Validation and Virtualization solution set that increases the team's ability to design, build and change services in parallel, while mitigating the risk of change and complexity in service environments, and the constraints of dependency on live services. The LISA Virtual Service Environment (or LISA VSE) creates a "virtual environment" for services, which can capture live or test traffic and model services, ESBs, databases and other systems of record for development, functional validation and performance testing purposes.

LISA dynamically invokes and gathers responses from a Service, capturing a behavioral model of the service's behavior, including the implementation and data layers underneath it, and creates a Virtual Service asset internal to LISA that allows relevant testing to continue in absence of the implementation.

Test Case in iTKO LISA Test Manager



Source: iTKO

Moreover, LISA supports virtual test beds, providing a proxy-level recording of Web Services, and replay of server SOAP responses. Also, a remote service suite which creates virtual test environments to simulate the effects of testing against third party services that are deployed outside the local implementation. This remote service suite provides virtualization of the supporting infrastructure, and provides read responses back to the tester. For staging, LISA provides an automated virtual test environment staging at build time and change time.

The LISA solution interacts with the leading SOA Governance and integration platforms by allowing the tester to build web service test cases from UDDI (Type 2 and 3) registries such as Software AG CentraSite or HP Systinet. At test case run time, rather than rely on a hard-coded URL, LISA pulls the latest WSDL (t-model), using the Registry to dynamically obtain the most current or relevant version of the service endpoint.

Copyright, Trademark Notice, and Statement of Opinion

About ZapThink, LLC

ZapThink is an Enterprise Architecture (EA) strategy advisory firm. As a recognized authority and master of Service-Oriented Architecture (SOA) and EA, ZapThink provides its audience of IT practitioners, consultants, and technology vendors with practical advice, guidance, education, and mentorship solutions that assist companies in leveraging SOA to meet their business needs and presenting viable SOA solutions to the market. We provide this audience a clear roadmap for standards-based, loosely coupled distributed computing – a vision of IT meeting the needs of the agile business.

ZapThink provides IT practitioners strategic insight and practical guidance for addressing critical agility and change management issues leveraging the latest EA and SOA best practices. ZapThink helps these customers put EA and SOA into practice in a rational, well-paced, and best practices-driven manner and helps to validate or recover architecture initiatives that may be heading down an unknown or incorrect path. ZapThink assists with solution vendor, technology, and consultant selection based on in-depth, objective evaluation of the capabilities, strengths, and applicability of the solutions to meet customer needs as they relate to EA initiatives and as they map against emerging best practices. ZapThink enhances its customer's skills by providing education, credentialing, and training to EAs to develop their skills as architects.

ZapThink helps to augment consulting firms' EA offerings and intellectual property by providing guidance on emerging best practices and access to information that supports those practices. ZapThink provides frameworks for product-based consulting based on ZapThink insight and research, such as SOA Implementation Roadmap guidance, Governance Framework development, and SOA Assessments, and provides a means to endorse and validate consulting firm offerings. ZapThink also accelerates consulting firms' efforts to attract, retain, and enhance the skills of EA and SOA talent by providing education and skills development

For solutions vendors, ZapThink provides retained advisory for guidance on product strategy, as well as marketing, visibility, and third-party endorsement benefits through its marketing activities, lead generation activities, and subscription services. ZapThink enables vendors to leverage ZapThink knowledge to transform their offerings in a cost-effective manner.

ZapThink's Managing Partners are widely regarded as the "go to advisors" and leading experts on SOA, EA, and Enterprise 2.0 by vendors, end-users, and the press. Respected for their candid, insightful opinions, they are in great demand as speakers, and have presented at conferences and industry events around the world. They are among the most quoted experts in the IT industry.

ZapThink was founded in October 2000 and is headquartered in Baltimore, Maryland. Its customers include Global 1000 firms and government organizations, as well as many emerging businesses. Its Managing Partners have worked at such firms as IDC, marchFIRST, and ChannelWave, and have sat on the working group committees for standards bodies such as RosettaNet, UDDI, and ebXML.

Call, email, or visit the ZapThink Web site to learn more about how ZapThink can help you to better understand how SOA and Enterprise 2.0 will impact your business or organization.

ZAPTHINK CONTACT:
ZapThink, LLC
108 Woodlawn Road
Baltimore, MD 21210
Phone: +1 (781) 207 0203
Fax: +1 (815) 301 3171
info@zapthink.com

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com