With all the attention and pure hype that Web Services have been receiving in the past year, there are remarkably few sources of information about testing Web Services. Of course, if Web Services were nothing more than traditional software components with standard XML interfaces, then there wouldn't be much to write about. Today's mature software testing tools and techniques would be sufficient. It's true, after all, that the primary use for Web Services currently is to simplify and reduce the cost of integration within the enterprise. Such straightforward applications of Web Services -- wrapping components with SOAP interfaces so that they can exchange XML-based RPC messages with other components -- are relatively straightforward to test with today's software testing tools and techniques. However, if you've read my previous articles on Web Services for The Rational Edge then you know that Web Services have far more potential than a simplification of existing component programming techniques. Over the next few years, Web Services promise to fundamentally change the distributed computing landscape. This new landscape will present new testing scenarios and problems that Web Service producers and consumers will have to understand.

The Two Dimensions of Web Services Testing

To understand the issues facing Web Services testing from now into the future, it is important to understand the current spectrum of testing techniques and how they apply to Web Services -- what might be called the "first dimension" of Web Services testing. The second dimension takes us into the future, as enterprises move beyond today's simple applications of Web Services to a broad, Service-oriented environment that contains large numbers of dynamically described and discovered Services.
Combining these two dimensions of Web Services testing with a roadmap that lays out the enterprise adoption of Web Services over the next few years provides a clear picture of how the area of Web Services testing should develop over time.

First Dimension

Fundamentally, Web Services are software components that expose their functionality via SOAP (Simple Object Access Protocol) interfaces. A WSDL (Web Services Description Language) metadata file provides information about the Web Service's interface, including the port type, binding, and so forth. On this fundamental level, therefore, testing a Web Service is a simple matter of exchanging the appropriate SOAP messages with it. Such messages can be "hardwired" into the testing tool, or the testing tool can also access the WSDL file in order to dynamically create the SOAP messages necessary for testing the Service. To accomplish this basic testing need, testers can employ the current range of software testing techniques, which includes:

- **"Black box" functional testing.** Verifying that the functionality of a component matches the functional specification for that component. The term "black box" refers to the fact that this kind of testing is concerned only with the component's inputs and outputs.

- **"White box" structural testing.** Analyzing the structure and organization of the programming code within a component in order to verify the quality of that component. White box testing is also often referred to as code profiling.

- **Regression testing.** A form of black box testing in which a component's functionality is compared to the functionality of a previous version of that component, to verify that changes to the component haven't broken anything that worked previously.

- **Load testing.** A set of testing techniques that includes capacity testing, stress testing, and tuning. Simply put, load testing involves emulating a certain number of users (or, in general, volume of input traffic) for a component in order to verify that the component performs properly under the required load.

- **Unit testing.** Testing individual components separately from the rest of the system in which they participate, typically as part of the day-to-day work of the developer.

- **System testing.** Testing systems of components to ensure that the system as a whole meets the specifications set out for it. Part of system testing is integration testing, which focuses specifically on the exchange of messages and function calls between components.

Second Dimension
As enterprises take advantage of the dynamic discovery and invocation capabilities of Web Services to build loosely coupled Service-oriented architectures (SOAs), many testing issues will arise that the simplistic approach to Web Services testing outlined above do not cover. These issues lead to the "second dimension" of Web Services testing, which involves the following Web Services and SOA capabilities:

- **Testing SOAP messages.** This means not only using SOAP as the interface to the Web Service, but also testing the format of the messages themselves.

- **Testing WSDL files and using WSDL files for test plan generation.** Web Services are special, in that they have files containing metadata about their interfaces that follow open standards. Testing tools use these WSDL files to generate black box test plans automatically. WSDL files can also help with white box testing.

- **Web Service consumer and producer emulation.** When a testing tool generates test messages that it sends to a component and then analyzes the results, it is essentially emulating the client for that component. When the component is a Web Service, the testing tool emulates the consumer of the Service in order to test the Service. With Web Services, however, the consumer of the Service also sends and receives SOAP messages. Therefore, testing tools also emulate the Web Service producer (namely the Service itself) in order to test the Web Service consumer.

- **Testing the publish, find, and bind capabilities of an SOA.** A fundamental characteristic of SOAs is the "publish, find, and bind" triangle: A Web Service provider first publishes the WSDL file for the Service in a UDDI registry, where Web Services consumers can find it. The consumers then bind to the Service based upon the WSDL file in the registry. Web Services testing tools test each leg of this triangle.

- **Testing the asynchronous notification and alert capabilities of Web Services in addition to their synchronous RPC capabilities.** Today's early uses of Web Services are often simplified remote procedure calls (RPCs). Using SOAP as an RPC format is quite powerful, but it does not utilize the full capabilities of the specification. In addition to this straightforward synchronous use, SOAP also supports asynchronous messages, including notification (from consumer to producer) and alert (from producer to consumer) messages. Web Services testing tools must test each kind of SOAP message.

- **Testing the SOAP intermediary capability.** The SOAP specification provides for message intermediaries, or actors. A particular SOAP message will typically have a designated recipient, but it may also have one or more actors that take one or more actions based upon the instructions provided to them in the header of the SOAP message. Not only must Web Services testing tools verify the proper functionality of these intermediaries, but they must also
verify that such intermediaries do not present security holes.

- Web Services orchestration testing. As companies gain momentum in adopting Web Services, they will combine many fine-grained Web Services into larger, coarse-grained Services. Such orchestration can be either recursive (Web Services made up of several Web Services that may also be made up of Web Services) or sequential (a series of Web Services arranged into a business process), or some combination of the two. Such orchestrations of Web Services typically involve more than one company, for example, when they represent a B2B transaction.

- Service-level agreement (SLA) and Quality of Service (QoS) monitoring. Moving from design time to runtime, some Web Services testing tools verify that Web Services, individually and in concert, are performing the way they should.

- Web Services versioning and "agile architecture" testing. As enterprise IT environments move from the tightly coupled, component-oriented world of today to the loosely coupled, Service-oriented environment made possible by Web Services, enterprises will be able to upgrade individual Web Services on an ad hoc basis, instead of as part of a formal software development lifecycle rollout. For such ad hoc upgrades to be successful, however, the enterprise must have testing tools that can test the new versions of individual Web Services, either in production or in an environment that parallels the production environment as closely as possible.

One end of this second dimension for Web Services testing is firmly rooted in the present use of Web Services, namely, SOAP messages used for RPCs. The other end of this dimension is several years off, in a transformed IT environment characterized by large numbers of loosely coupled Services. Connecting the dots from one end to the other is an important part of the Web Services adoption roadmap.

The Web Services Adoption Roadmap

The two dimensions of Web Services testing form a grid of functionality that Web Services testing vendors should implement as enterprises adopt Web Services more extensively. The following Web Services adoption roadmap can help vendors understand which functionality they should implement and when. The roadmap has three phases.

Phase One

The first phase of Web Services adoption -- the phase we're in today -- is internally focused. Enterprises use Web Services primarily to simplify and reduce the cost of integration within the enterprise. Some companies are exploring the use of Web Services to communicate outside the firewall, but these instances are often pilot projects with trusted business partners who have known IT organizations. The primary roadblocks that slow adoption of Web Services during the first phase are security, Web Services
management, and transactions. The primary Web Services invocation style enterprises use during this phase are static bindings to static Web Services.

It is during this first phase of Web Services adoption that Web Services are, for all practical purposes, software components with SOAP interfaces. Web Services testing tools can perform black box testing simply by adding XML support to their list of supported interfaces. WSDL files can provide some insight into the internal workings of a Web Service, so therefore a testing tool can use the WSDL file to create automated test plans that exercise various internal code structures.

Load testing Web Services in Phase One is also relatively straightforward. The primary difference between load testing Web Services and, say, load testing Web pages, is that there is no user interface to emulate when testing a Web Service. As a result, load testing tools that mimic a user to the extent of interacting with a real or emulated browser will not be as appropriate for testing Web Services as those tools that mimic browser behavior on a lower (i.e., HTTP) level. The goal of a Web Services load testing tool is to accurately emulate the behavior of Web Services consumers who will exchange messages with the Service.

In Phase One, Web Services do have consumers, and it is therefore important to test those consumers. Web Services testing tools should typically emulate a Web Service in order to test the consumers who access that Service. Only by testing both the producers and consumers of Web Services can an enterprise perform system testing that includes Web Services.

**Phase Two**

As new products and services enter the market and resolve most of the issues with Web Services security, management, and transactions, companies will be able to exchange Web Service messages with other companies (customers, suppliers, and partners) in a much freer, loosely coupled manner. The UDDI specification will be more mature, and many UDDI implementations will be available, as well as other possible Web Service registry protocols. Such registries won't be particularly useful for enabling business relationships between strangers; instead, enterprises and established groups of business partners will find that Service registries are a critical enabler of the dynamic discovery of Web Services within controlled environments. The Services themselves will still typically be static, but Web Service consumers will find and bind to them dynamically. The primary roadblocks to further adoption of Web Services during Phase Two are: orchestration, business process automation, and billing and metering of Web Services.

It is the dynamic, "publish, find, and bind" capabilities of Web Services that characterize this phase of the adoption of Service-oriented architectures. Testing these capabilities, while not strictly speaking a part of black box functional testing for individual Web Services, will become an ever greater part of testing the functionality of Web Services from a Service-oriented perspective, as opposed to simply a component-oriented view. Testing Web Service consumers will also become more complex,
because they must be able to find the appropriate Web Service as well as bind to it.

Load testing Web Services in Phase Two will certainly become more complex than in Phase One. The most obvious reason is that the functionality of the registry, as well as the publish and find operations, must support the required traffic loads. But a more subtle difference in Phase Two Web Services load testing will emerge, because the Service-oriented approach to scalability fundamentally rests upon the actions of the registry. Service-oriented architects will use UDDI registries to organize and locate redundant resources on the network. Therefore, load testing these architectures will depend entirely on the proper configuration of such registries.

System testing in Phase Two will probably be the most complex of all, because the systems being tested will no longer be defined at design time. Instead, the individual Web Services may be defined at design time and published to registries. The Web Service consumers will then locate those resources and bind to them at runtime. System tests must therefore take this dynamic nature of Service-oriented architectures into account.

**Phase Three**

As Service-oriented architectures mature and Web Services orchestration and business process automation tools become prevalent, the fundamental invocation style for Web Services will move to **JIT (Just-in-time) integration**. With JIT integration, the Web Services themselves will be dynamically described at runtime. In Phase Three, Web Services consumers will dynamically discover Web Services as in Phase Two, but now those Services might be different every time. Their Service description in the corresponding WSDL file will provide the Web Services consumer with all the information they need to dynamically bind to the Service at runtime.

Furthermore, the concepts of Web Services "consumer" and "producer" will become more arbitrary, because complex orchestrations of Web Services will be the norm rather than the exception. Businesses will expose coarse-grained Web Services to their customers, suppliers, and partners (for example, a "product catalog" Service), and those Web Services will actually consist of a large, dynamic collection of finer grained Services (e.g., "product list," "current inventory," "preferred pricing"), which might themselves consist of even finer grained Web Services. In addition, each of these component Web Services might contain Services provided by other companies (e.g., "credit approval").

In such a fully realized Service-oriented environment, the distinction between design time and runtime will blur, as well. Because individual Services will be dynamically described and published, IT managers will be able to upgrade them on the fly. No longer will enterprises need to roll out upgrades on long, expensive lifecycles. Instead, developers will build, test, and launch individual Services as needed.

Clearly, the ponderous testing processes of the sequential waterfall software development methodology -- first design, then develop, then test, then launch -- will be entirely inappropriate for the fully realized
Service-oriented environment in Phase Three. Instead, software development groups will by necessity have to take what is now known as an agile approach to development and testing: Work with the customer, tackle just what needs to be done, write a test, code as a team, pass the test, repeat until finished. Testing before coding will move from being "extreme" to being the only way of dealing with the dynamic nature of distributed computing.

In fact, the agile approach to testing appears to be the only approach that makes sense in Phase Three. A system test would never be comprehensive and complete if it could be conducted only once the system is in place, because the individual Web Services that make up that system are not determined beforehand. Instead, testing begins with automating the collection of user requirements whenever a new requirement is defined. The test and the resulting code then develop iteratively, so that when the code is complete, the test is as well -- and the code passes the test. If each individual Web Service has the corresponding automated test, then orchestrating Web Services involves orchestrating the tests as well. After all, typical user requirements are quite coarse grained. As a development team works with users to define coarse-grained requirements and write the corresponding coarse-grained tests, the testing tools that the development team uses must be able to incorporate the appropriate tests for all the pre-existing Web Services that, in combination, will make up the coarse-grained, orchestrated Web Service the users require.

Where to Go from Here

If you read my previous articles for The Rational Edge,¹ you'll see that I extrapolate from the current IT environment into the future following a reasoned, step-by-step approach. I base each prediction along the way on fundamental economic realities: People won't buy software unless it meets their needs, and vendors won't create software that doesn't make them money. So, will we ever make it to the dynamic, Service-oriented environment that underlies Phase Three? Well, it might take more or less time -- possibly until 2005 or beyond -- and it might be more or less difficult, but there are clear economic advantages for both the user and vendor communities in moving toward JIT integration. And if there's money to be made, you can rest assured somebody will do it.

Economic forces are changing the world of software testing today. Even the most conservative companies are realizing that the waterfall methodology is no longer a cost-effective approach to software development, and they are moving toward iterative methodologies simply because these methodologies are less expensive and less risky. At the same time, iterative methodologies are becoming increasingly agile, also as a way of reducing the costs and risks inherent in software development. Each of these changes -- from waterfall to iterative, and from iterative to agile -- involves fundamental differences in the approach to software testing.

It's no coincidence that this maturation of the software development process parallels the move toward loosely coupled, open standards-based distributed computing. Just as iterative methodologies became practical only when monolithic application development gave way to component-
oriented development, today's agile methodologies (which are admittedly still on the fringe) will by necessity become the only cost-effective, practical way of conducting the practice of creating software in the fully realized Services-oriented environment.

Notes


For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!