Web Services and a New Approach to Software Development

by Jason Bloomberg
Senior Technologist
Ashton Services

Web Services is by far the hottest topic in software circles since the dotcom implosion, but sometimes it's hard to see what all the fuss is about. The most immediate promise of Web Services technologies like SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language) is that they will simplify the integration process. Now, there's no question that integration -- especially on the enterprise level -- can do with some simplification. After all, up to 75 percent of the expense in many large-scale software projects often goes toward simply integrating new systems with old. Anybody who's been in the trenches in any EAI (Enterprise Application Integration) project knows that the hard work is in building the wrappers -- the special code that allows disparate systems to communicate. The promise of Web Services is to simplify this wrapper code by boiling it down to XML-based standardized interfaces. That means that all wrapper code would follow the same standards, and integration would become simpler, and hence, less expensive.

If, in the end, Web Services do nothing more than reduce the cost of integration, they are still sure to be worth the trouble. However, if that's all they're good for, then they will be merely a relatively small evolutionary step in the discipline of software engineering. Not to worry. If you read my article in the March issue of The Rational Edge, then you know I believe Web Services have a much brighter future than this, although they are still in an early phase. Whereas my last article focused on business changes that might be wrought by Web Services, this article focuses on the world of software. As you read, keep in mind that today's new Web Services-capable software development tools, while clearly more mature and sophisticated than their predecessors, still only scratch the surface when it comes to building powerful Web Services applications. This limitation is not the fault of the tools; rather, it is due to the limitations of the entire discipline of software development.
The Seven Principles of Service-Oriented Development

Software development has come a long way since the early procedural programming days. Object-oriented analysis and design led to distributed object frameworks, which in turn led to today’s distributed component frameworks. Service-oriented development is the next logical step in this progression. However, just as developers had to "unlearn" procedural programming in order to fully grasp object-oriented techniques, so too must they adapt to the new service-oriented development principles that Web Services will require.

To break free from the old ways of thinking, software professionals must have both an understanding of the capabilities of the new tools, as well as the big picture of what can be built with them. The principles of service-oriented development that follow -- expressed as transitions from the old way of thinking to the new -- are, effectively, guidelines for reaching this new level of understanding.

**Principle #1: Static Components -> Dynamic Services**

Traditional component-based software development naturally focuses on building and integrating components. To build a component, the developer takes the specifications and puts together various pieces of code in an attempt to make some software that satisfies the specifications. In other words, the developer is done if the software does what it's supposed to do.

If the developer is building a Web Service, however, it is not enough to build the Service with set functionality as specified. Instead, the Web Service should be defined by a dynamic description of its functionality, as shown in Figure 1. Such dynamic descriptions are currently expressed in a WSDL file for the Web Service. The developer, therefore, need only indicate where the WSDL file is to be found, and the software that seeks to invoke the Web Service can find the functional description at runtime.

![Figure 1: Dynamic Web Service Represented as a Component with a Dynamic Description](image)

One of the most significant ramifications of this principle is that it mandates the separation of the presentation logic from the business and persistence logic within a system that uses Web Services. When a developer builds a Web Service, he or she may have no idea whatsoever how that Service will be invoked, or what the user interface on the invoking software will be. Now,
n-tier systems traditionally separate presentation logic onto a different tier from the business logic and persistence tiers, but in the case of n-tier architectures, such separation is motivated by the benefits of manageability and reuse. Therefore, an n-tier architect has the option of mixing up the functionality of the tiers somewhat if there is a benefit that outweighs the loss of reusability and manageability. Not so with Web Services.

**Principle #2: System Integration -> Service Exposure and Reflection**

Today's approach to integration begins with system-level requirements. Depending on what the system is supposed to do, the architect plans out the various components and how they should be integrated. Instead of taking this top-down approach, service-oriented development takes a bottom-up approach. Each component in a system should be exposed as a Web Service before any system architecture is in place. Then, each service uses reflection to provide external systems with the information they need to access the data and functionality encapsulated in the service. (*Reflection* means that the Service allows the invoking software to query the Service about its interface.)

![Figure 2: A Service Assembly for a Service-Oriented System](image)

When building a system using service-oriented development, then, the architect starts with the system requirements and performs a *service assembly* step, as shown in Figure 2. Service assembly corresponds to identifying the appropriate components for the system, except that in the Web Services environment, the architect accesses the dynamic descriptions of the Services (as described in Principle #1). Once the required dynamic Service descriptions are identified, the architect can specify the structure of the system, even though the individual components and their interfaces are not fully described, and won't be until runtime.

There are two important aspects to Principle #2. First, when each data source or business object is exposed as a Web Service, the developer need not know how systems will access or utilize that Service. Second, when the system architect plans the system, the actual nature of the components (which application, platform, etc.) is irrelevant.

**Principle #3: Coding for Reusability -> Coding for Broad Applicability**

Coding for reusability is one of the pillars of object-oriented programming.
Write the code once and use it many times, thus saving both money and time. In practice, however, writing reusable code can be far more difficult than writing code for a single-use application. There are three main problems with attempting to write reusable code:

1. It takes more time than one-off code.
2. There's no guarantee that the code will actually be reusable.
3. What the developer produces may not be consistent with customer requirements. In many cases, the developer doesn't even know when to stop coding.

For these reasons, agile software methodologies like Extreme Programming (XP) shun reusability. In XP, developers code only what the customer actually needs. The customer actively participates on the development team, providing user testing and guidance. If extraneous functionality creeps in, then developers refactor the code, which means they rework the code until it's as simple as possible.

When developers refactor, they streamline the code until it is general enough to handle all the cases in which there was redundant code before. In essence, refactoring means building for a kind of reuse. However, refactoring is different from traditional coding for reusability, because its aim is to create code that is flexible and broadly applicable.

The difference between reusability and broad applicability is subtle, but essential to the Service-oriented development process. The first two principles imply a level of developer agnosticism with respect to the functionality of a particular Web Service: The description of the Service is dynamic, and the developer need not know how the service will be used. What, then, keeps the developer on task and scope creep in check? The answer is to couple the architectural principles of service-oriented development with the engineering principles of agile development.

Development of Web Services should be an ongoing, iterative process that actively involves the opinions of the users. The services themselves, then, should be constructed to be as simple as possible, and developers can continually refactor them so that they are as broadly applicable as practical.

Principle #4: Disruptive Upgrades -> Ad hoc Upgrades

Modularity, along with reusability, is another fundamental tenet of object-oriented programming, and hence component architectures in general. If a system is modular, that means the individual components that make up the system can be upgraded or replaced without affecting the rest of the system. There is no question that modularity makes such good sense that even the most non-technical business user can understand it. Unfortunately, in today's enterprise component architectures, modularity is largely a myth.

The problem is that replacing or upgrading components in a complex system is never as simple or inexpensive as people would like. Many times, system components are not fully encapsulated: Despite the architect's carefully laid plans, there is often some internal hack that provides some essential piece of functionality which the carefully designed interfaces do not support. In other
cases, the API (application programming interface) includes some semantic
ambiguities. For example, the old component's `getQuantity()` method
returns the number of boxes, but the new component returns the number of
pallets. And of course, new APIs often differ from the old ones they're
replacing, which then necessitates substantial reintegration work.

One of the primary goals of service-oriented development is to solve this
morass of modularity issues. Instead of simply exposing APIs, components
wrapped in Web Services expose dynamic service descriptions. If the
underlying API changes, then the service description adjusts automatically,
and the other components of the system can adjust to the changes at
runtime. And because the separation of the different logic tiers is now
mandated instead of recommended, hacks that break the encapsulation of
the Web Services are much less likely. (Of course, developers will still be
able to break the rules and code badly, but following the principles of service-
oriented development will eliminate the need for such hacks).

Once the dust settles and service-oriented development begins to work the
way it is supposed to, enterprises will realize that upgrades of component-
based architectures can take place on an *ad hoc* basis. Unlike today's legacy
upgrade project, which can require multiple years, millions of dollars, and
many hours or days of downtime, upgrading a component exposed as a Web
Service will be a low-risk part of regular maintenance. From a business
perspective, then, enterprise architectures can remain current, smoothly
upgrading themselves as necessary.

**Principle #5: Top-Down Scalability -> Bottom-Up Scalability**

Any architect who has been called upon to scale up an e-business system
that is experiencing unexpected surges in traffic knows full well that
scalability is far more complex than adding an additional server here and
there. The fact of the matter is, in today's world, scalability must be carefully
planned far ahead of time. Architects will begin with predicted usage
patterns (both average and peak) and use those estimates to plan for load
balancing and failover. They must be especially careful to watch out for
bottlenecks in the system; a site of 1,000 pages can run smoothly for
months, but if one page is featured on the news, then the resulting surge in
traffic can bring the entire site down.

In a fully realized Web Services-enabled environment, scalability can be
handled in a bottom-up fashion, instead of via the fallible, top-down
approach used today. UDDI (Universal Description, Discovery, and
Integration) registries can be set up to provide lists of backup Web Services,
solely for the purposes of scalability and failover. If a system is experiencing
unexpected traffic, it can automatically find backup services in a registry,
obtain their service descriptions (also stored in the registry), and bind to the
supplemental services on the fly. Furthermore, registries providing access to
supplemental Web Services can also be configured recursively: A system can
query a registry for supplemental Web Services, and even if that registry has
no information about a suitable service, it can contain information about
other such registries. The needful system can therefore hop from one
registry to another until it finds the extra Web Services it needs.
Principle #6: Platform Dependence -> Platform Irrelevance

Established service-oriented architectures such as CORBA and Windows DNA have never offered much in the way of platform independence. Naturally, Windows DNA makes no apologies about being a Windows-only architecture. And CORBA, while theoretically platform neutral, in practice has typically been able to achieve cost-effectiveness only in single-platform implementations.

At the enterprise level, EAI (Enterprise Application Integration) has become the predominant approach to building cross-platform solutions. Each EAI solution is typically a "bus" that lies between the application servers and other components that support the user interfaces on the one hand, and the back-end legacy data sources and various business components on the other. Each EAI bus may support both synchronous and asynchronous communication, and these buses often contain sophisticated transaction processing capabilities as well. However, although EAI enables enterprises to build cross-platform architectures, these architectures are all very expensive. The EAI software alone often runs into the hundreds of thousands of dollars, and even that amount pales in comparison to the integration costs involved in an EAI implementation. The problem is that the real work (read: money) that goes into an EAI implementation project is in building the wrappers that enable the various component systems to interface with the EAI bus. Therefore, most of the work involved in EAI is actually platform-dependent development work after all.

Web Services have gained attention in large part because they offer a way to dodge the expense of EAI. The big irony today is that the most contentious issues in the Web Services world are those of platform dependence. Microsoft's .NET initiative, their replacement for Windows DNA, is again, unapologetically, a Windows-based approach. The Java/J2EE camp, this time led by IBM, is circling its wagons and getting ready for a major platform brawl with the folks in Redmond. What's so ironic about this brewing battle is that service-oriented development promises to make the entire platform issue completely irrelevant.

The component-based way of thinking is to have components built on platforms that expose their functionality via their interfaces. Components then work with each other by accessing each other's interfaces. In essence, the components are soccer players who kick a ball to each other, and the platform is the field. Needless to say, it's difficult for two players to interact if they are on different fields.
The service-oriented way of thinking is that the entire software environment consists of dynamically described services that can be located and invoked on the fly when necessary, as illustrated in Figure 3. Instead of a platform in the true sense, the soccer field is now XML -- nothing more than a markup language whose extensibility allows all players to join in. Now, platforms aren't going away any time soon -- Web Services are still software, after all, and there still need to be systems that provide the user interfaces to the service-oriented environment. J2EE and .NET will therefore move to the periphery where the users are, providing the infrastructure necessary to write Web Services as well as the desktop, browser, and other interface support that users need to access those Services. In the middle of the fray, however, will be Web Services alone, playing on a field of XML.

**Principle #7: Dictatorship Software Model -> Federation Software Model**

Web Services' loose coupling solves many of the problems that result from the tight coupling found in component-based architectures with rigid APIs, including integration issues (Principle #2) and modularity issues (Principle #4). However, loose coupling and the other principles of Service-oriented development will have other, even more profound effects on the world of software. Traditional enterprise software (Siebel's core CRM offering, for example) is integrated at the database level, providing for an extraordinarily tightly coupled application suite. As component technologies have matured, enterprise software vendors have been more likely to build their software following object-oriented principles -- still tightly coupled, but now communicating via exposed APIs.

Taking this progression one step further, architects will be able to create enterprise software by assembling dynamically described collections of Web Services. Now, this involves more than simply taking a component-based architecture and using SOAP as the RPC (remote procedure call) or messaging protocol, which is as far as many companies are taking Web Services today. Software vendors who understand all seven principles of Service-oriented development will publish enterprise software that consists of a loosely coupled federation of Services, many found through registries on the fly, some not even developed in-house. As time goes on, the various Web Services that make up such an enterprise package will change from day to day, as developers perform *ad hoc* upgrades, and systems scale up and down at runtime by assembling supplemental Web Services as necessary. In
essence, tomorrow's enterprise software vendors will follow entirely different models -- both architectural and business models -- than today's vendors.

**Where Will We Go from Here?**

There's no question that the adoption of many of the principles we discussed above is still on the horizon, maybe as many as five years away. Some of them, however, are much more imminent. Software vendors who are planning their next product release must look a year or more out, because of the development time needed to bring a product to market. Looking a year out essentially means predicting the future, a notoriously fallible endeavor. A more reliable approach is to work within the context of a *scenario* -- a logically constructed series of possible events that are likely to take place, given certain assumptions about how the world will change.

The seven principles I've articulated effectively delineate such a scenario. Collectively, they can provide the context necessary for making intelligent predictions about the future of software architecture and development. Will everything I've outlined here come to pass? Probably not. Will this article instigate discussions that help people make decisions about where to concentrate their software efforts? I hope so.

---

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

---

Copyright [Rational Software](#) 2002 | Privacy/Legal Information