A Rational software whitepaper from IBM
10/03

Rational. software

Developing Applications for the Web Service Era

Jim Conallen
Senior Engineer

ABSTRACT 1

WHAT IS A WEB SERVICE? 1

CLIENT/SERVER INTERACTION, WEB SERVICES STYLE 2

Use-Cases and UML: Common Semantics for Web Services Developers and Client Developers 3

CHALLENGES FOR WEB SERVICES DESIGNERS 3

Using Behavioral Diagrams 4

Using Automated Tools 5

Web-Related Issues 6

Using OO Best Practices 7

DESIGNING APPLICATIONS THAT USE WEB SERVICES 7

SUMMARY 8

Abstract

Web Services, a new architectural mechanism available to enterprise software development teams, result from a confluence of emerging and established technologies, including XML, HTTP, and SOAP. This paper will discuss some of the challenges inherent in developing both Web Services applications and the consumer applications that call on them.[1]

What Is a Web Service?

Simply put, a Web Service is a collection of functions packaged together and published on a network for use by other client programs. At a very high level of abstraction, we might think of a Web Service as a type of Remote Procedure Call (RPC) or messaging system. The idea of a client application requesting a service from a server application is not new; the difference is all in the plumbing. A Web Service, as its name implies, is based on common, Web-related technologies: TCP/IP, HTTP, FTP, SMTP, and XML. In addition, three new specifications -- Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description Discovery and Integration (UDDI) -- together form the foundation set of technologies that define Web Services. Figure 1 shows the layered relationship of these technologies.

Service directory Service publication	UDDI
Service interface definition Service implementation	WSDL
XML Messaging	SOAP
Network Services	HTTP, FTP, SMTP, messaging
	TCP/IP

Figure 1: Layered Relationship of Web Service and Related Technologies.

A Web Service runs on a server connected to a network and "listens" for incoming service requests. Typically, clients send those requests in the form of an XML document and use HTTP[2] to get the message to the server. The structure of the XML document is typically defined by the Simple Object Access Protocol (SOAP),[3] although other protocols can be used instead.[4] We can think of a SOAP document as a message or function call expressed in XML. Unlike function calls between components that rely on the frameworks of a common operating system or distributed object system, SOAP defines how XML can be used to express all the information of the function. SOAP messages specify details of the encodings used and how data types are handled. Despite the apparent simplicity of SOAP, however, there are "hidden" issues development teams should consider as they develop or use Web Services. We describe some of these later in the paper.

---

[1] For a good introduction to Web Services architectures, see Alan Brown, Simon Johnston, and Kevin Kelly, "Using Service–Oriented Architecture and Component-Based Development to Build Web Service Applications." Rational Software Whitepaper, 2002.

[2] Although other protocols can be used (e.g., FTP, SMTP, message oriented middleware), HTTP is most commonly used today.

[3] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000. http://www.w3.org/TR/SOAP/

[4] XML-RPC is a simple alternative to SOAP as a mechanism for expressing remote procedure calls with XML. At this writing, however, SOAP is by far the preferred format.

Rooted in component-based development, Web Services are evolutionary, not revolutionary. Their evolution was driven in part by two desires on the part of the software industry: to make software reusable, and to offer potential customers programmatic access to resources and business services. These same desires have driven interoperability standards in the past. Today, two additional forces are shaping the development of Web Services as a technology: the need for platform independence and a desire to leverage existing Web technologies.

All of these motivating factors align with the viewpoints of the development and business communities. Application architects are interested in reuse as a means to help achieve their goal of improving application quality and time to completion. Large organizations often have a wide variety of systems and processes, and components can help architects eliminate duplicate functionality and promote consistent and efficient use of critical resources within their companies. In the larger view, businesses recognize that they can profit from services that offer customers efficient access to business resources, and by efficiently using the resources of other companies.

Any Web Service can interact with any other Web Service; Web services can also aggregate to provide higher-level functions. After all, they are just software components, and all software components have the potential to invoke Web Services. By implication, some Web Services will have dependencies on other services that are either invisible to the client or require client participation. For example, a Web Service might require that the client use another Web Service to create some token or value that it must then pass on to the next service. Or, a Web Service might internally invoke other services before returning a result to the client.

This paper will focus on issues related to Web Services development on both the consumer and producer sides, which have very distinct concerns. To invoke the Web Services of a business system, a consumer application needs to "know" how and what to send, but the developer of such an application need not be concerned with how the message is processed. The Web Service developer, on the other hand, must attend to issues that span everything from design and implementation to publishing and explaining how to use the services.

Let's begin by looking at the basic mechanism of Web Services.

Client/Server Interaction, Web Services Style

As we have noted, in its simplest form, a Web Service is nothing more than a client sending SOAP-encoded messages to a server, which accepts and processes the messages. If the client and the server applications were developed by one team, then little more would be needed to make this happen; the client team would have all the information it needs to move ahead. In reality, however these applications are typically developed not only by different teams, but also often by different organizations or parts of an organization.

Depending upon which part you are developing, your understanding of the other system is limited to the protocol or specification that defines the Web Service. If you're a client application developer, you need something that provides all the information necessary to establish a connection to the server, and to explain exactly what should be in the SOAP messages. That "something" is a Web Services Description Language (WSDL)[5] document. A WSDL document is an XML formatted document that is like an interface definition for a class. It describes how a client can connect to, and invoke, a service.

Discovering a suitable Web Service is another task for the client application developer, and the Universal Description Discovery and Integration (UDDI)[6] specification provides the necessary support by defining a standard mechanism for publishing and locating businesses and the services they provide. Typically, client developers determine ahead of time (i.e., during application design) which Web Services and providers the application will use. Some applications, however, are designed to discover and use Web Services at runtime. An even more sophisticated client application might query a UDDI registry for a compatible Service, retrieve and process the Service's WSDL document, and then, based on the information in the WSDL document, invoke the Service. This depth of sophistication, though, is not typical. The majority of Web Service clients use WSDL documents during development, not during runtime.

---

[5] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001. http://www.w3.org/TR/wsdl.

[6] For more information see http://www.uddi.org/

Use-Cases and UML: Common Semantics for Web Services Developers and Client Developers

Although WSDL is fine for describing the messaging and contact details, it is a little lacking when it comes to expressing the higher-level semantics of the Service. Both Web Services developers and client developers need to be just as concerned with the semantics as with the mechanics of the messages being exchanged. Just because a WSDL document defines a concept called "address" doesn't automatically mean that the client and server will interpret the concept the same way. Web Services developers and client developers alike need to ensure that the messaging details and the semantic details of the service are compatible.

Use cases, a powerful modeling and requirements technique defined in the Unified Modeling Language (UML),[7] provide a common set of semantics for Web Services developers and client developers (see Figure 2). Each use case describes the dialog between an Actor (i.e., a client) and a system (i.e., a Web Service). An accompanying use-case specification is a textual document written in the language of the domain -- that is, it uses the same vocabulary used to define the business goals the Service is providing.

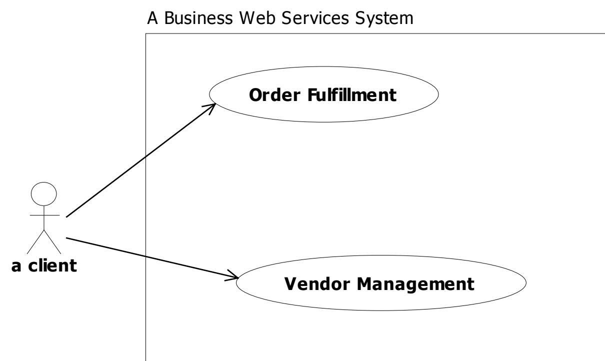

Figure 2: Use-Case Model of a Business's Web Services

In use-case terminology,[8,9] the client and server both see each other as an external Actor, albeit an automated one. The client Actor initiates the Web Service invocation to accomplish some goal, which might require a sequence of Web Service invocations. Use cases describe Web Services interactions in the same manner they describe human /computer interactions. Although Web Services use cases might contain more technically detailed descriptions than other use cases, the purpose is still to capture and express the high-level semantics of the exchange, not the details of the implementation. This makes use cases a good tool for ensuring a common semantic interpretation of Web Services interactions.

Challenges for Web Services Designers

The first and most important message for teams designing a Web Service is: "Web Services are different!" Web Services are based on Service-Oriented Architectures; older paradigms for component-based development cannot be blindly applied to Web Services. With the automation available today to produce object wrappers around Web Services, there is a temptation to treat Web Services like any normal component; in fact, some might think that you can port existing components to act as Web Services just by creating these wrappers and leaving the underlying component untouched. Unless the nature of the component makes it suitable for use as a Web Service (most are not), however, it takes serious thought and redesign to properly deliver a component's functionality through a Web Service. Insufficient respect for the Web Services infrastructure can lead to poorly designed and poorly performing services.

---

[7] For more information see http://www.rational.com/uml/

[8] Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

[9] Kurt Bittner, and Ian Spence, *Use Case Modeling.* Addison Wesley, 2002.

Careful attention to granularity is critical to the success of a Web Service: The inherent overhead of each invocation dictates a high level of granularity. That is, each Service invocation should deliver a significant value to the client. For example, you should not have to invoke the Service multiple times to set all the fields of a purchase order request; the Service should set the entire purchase order with a single invocation.

Designing Web Services with a high level of granularity results in fewer, but more complex, functions. Also, the control flow shifts from the client to the server. Instead of the client being responsible for setting each data field individually, they are done *en masse*. So the client loses the opportunity to adjust or respond to events that happen in between setting fields. Of course, good designs can mitigate this loss by providing advanced features that react just like the client would when processing the Service request.

Web Services tend to have fewer and more complex interfaces than other applications, and the messages passed to a Web Service can be quite complex. Fortunately, SOAP is sufficiently expressive to allow complex data structures to be defined and consumed. For traditional, in-process components, the efficiency and reliability of each method call is very high, resulting in component interfaces with many methods and small, simple signatures. Web Services developers, on the other hand, design interfaces with fewer methods that have larger, more complex signatures. Web Services developers are also likely to use more complex data structures in their messages than the data structures in the method parameters of a typical in-process component.

Although efficient use of a Web Service might require complex data structures, however, it is still good practice to keep these structures as simple as possible. Minimize any special or customized encodings, and use only the most common, fundamental data types. In the early releases of SOAP frameworks and kits, special data types such as big integers or dates were sometimes misunderstood. By sticking to the most common and oldest built-in types, you can make your application more robust and better able to interoperate with the latest versions of Web Services toolkits.

Using Behavioral Diagrams

To a client, a Web Service, or set of Web Services, is an implementation of an interface. The actual “instance” of the Service only needs to be known at runtime. Accomplishing a given task (i.e., a use-case goal) might require a number of invocations that will likely be sequenced; the sequencing might be dynamic, depending on the current state of the system. Developing a Web Service for this task requires an understanding of the system and user state, and of the control flow. These issues are not new to software development and can be managed in the same way you would manage them for other applications. UML behavioral diagrams – specifically sequence and activity diagrams -- are excellent tools for expressing the dynamic flow and behavior of systems. They are used in combination with use-case realizations, UML modeling elements that express use cases’ activities in terms of the system’s elements. These same tools can be used to express how a Web Service, or set of Web Services, can be invoked to deliver value to the client.

When designing and developing a Web Service, the public interface should be defined by a separate modeling element, such as a UML Interface (Figure 3). This interface is all that the client sees. Eventually, it will get translated into a WSDL document, but during design this interface is sufficient to begin the process of coming up with a good design. Sequence and activity diagrams are good tools for expressing and examining the flows of control that the client must make (Figure 4). Eventually, these diagrams become invaluable aids for client application developers trying to understand how to use a given set of Web Services.

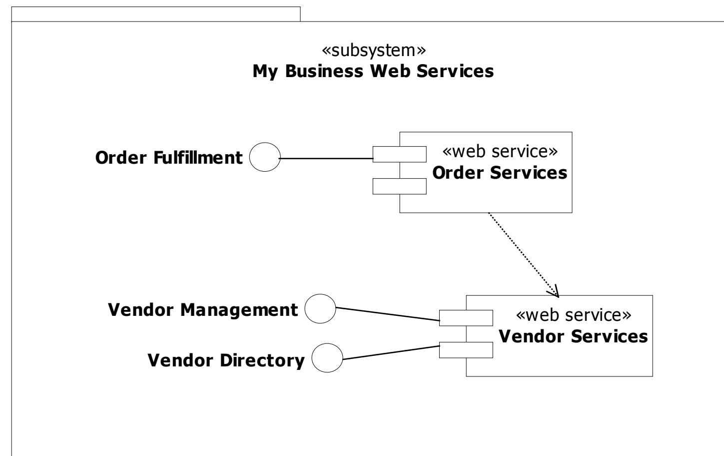

Figure 3: Modeling a Set of Web Services with Interfaces and Components

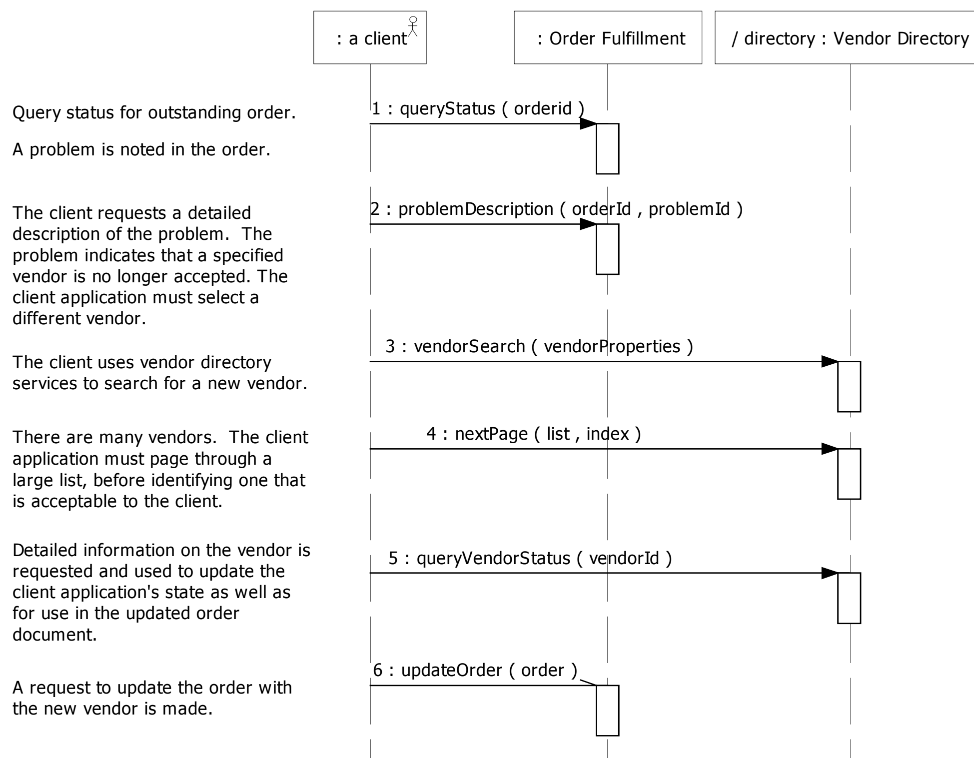

Figure 4: Behavioral Diagram Demonstrating the Sequencing and Usage of Multiple Services

The development team uses these diagrams as the basis for other behavioral diagrams that include not only the Service's public interfaces but also the key elements of their internal implementation. Using these diagrams, in combination with the class and state diagrams of the Web Service implementation, developers can elaborate and design Web Services to meet the needs and requirements expressed in the use cases. Detailing a Web Services implementation is no different than detailing other components. Web Services are, after all, just software; we can apply the tools we use to develop other software components equally effectively to Web Services implementations.

Using Automated Tools

From a developer's point of view, the plumbing and infrastructure that goes into managing SOAP messages is best left to the set of frameworks and tools chosen for the project.

Most of the tools available today provide the following key functions:

- Create wrapper classes from WSDL documents.
- Create WSDL documents from class or interface definitions.
- Manage the creation and usage of SOAP messages.

This functionality reduces a significant amount of the work involved in setting up a Web Service. The tools and frameworks provide insulation from the underlying infrastructure, allowing developers to focus on the business objectives of the service. The wrapper classes produced by the tools, and the framework classes they rely on are, of course, normal classes (C#, Java, etc.), and are therefore modeled in UML -- just like any other class.

This means that design models of a Web Service can focus on just the classes and objects that manage the business functions and not expose too many details of the infrastructure. Of course, there is always a danger that the design will begin to treat the Web Service as a normal, in-process component. Therefore, it is important to annotate the model with notes or by using a simple UML stereotype to identify Web Services implementations.

Theoretically, the excellent tools and frameworks that IBM, Microsoft, and Sun offer can all be used to create Web Services that can be discovered and invoked by applications built with another vendor's tools. In practice, however, there are a few incompatibilities between some vendors' implementations of SOAP frameworks, and these tend to become more pronounced as the frameworks evolve over time. For example, messages created with Apache SOAP always use the xsi:type attribute to help specify data types in a message. Microsoft SOAP messages don't use this attribute, however, so the Apache SOAP implementation rejects Microsoft SOAP messages.

Fortunately problems like this are rare and should become rarer still as the standards solidify. But the fact that they do exist emphasizes the need for testing: In addition to the normal testing any application should undergo, Web Services should be tested with a client, using a variety of SOAP implementations and operating systems.[10]

Web-Related Issues

Web Services, as their name implies, are Web applications. Therefore, they pose many of the same architectural issues that developers of other Web applications face, although fortunately not the numerous issues related to HTML and client side JavaScript execution. The following issues, however, are of concern:

- Fragile network reliability
- Client state and session management
- Security
- Performance

Whenever networks are involved, there is always the possibility of temporary outages or inconsistent connection quality, and Web Service and Web application developers are continually aware of this. HTTP, a connectionless (stateless) protocol, is well suited to this type of network environment. The drawback, of course, is that the management of state in any conversation falls on the client and server systems. Using message-oriented middleware is an option for mitigating poor network reliability.

Microsoft, IBM, and Verisign are currently working on a proposal to enhance the security of SOAP base messages.[11] They propose a general-purpose mechanism for associating security tokens with messages, and describe encoding schemes that should improve the security and reliability of Web Service invocations.

---

[10] *Ibid*.

[11] For more information see: http://www-106.ibm.com/developerworks/webservices/library/ws-secure/, Web Services Security (WS-Security)

Using OO Best Practices

When designing a Web Service, it is best to think of it as a separate component or subsystem, rather than as just another type of system interface. Like a component, it has a public interface that hides its implementation. Web Services are also object-like; they encapsulate an implementation that possesses state and behavior. Therefore, designers of Web Services should apply the best practices of object oriented (OO) design. Define the bounds of a Web Service so that it will do one thing and do it well. Grady Booch, *et al*.[12] describe a well-structured class as something that:

- Provides a crisp abstraction of something drawn from the vocabulary of the problem domain.
- Embodies a small, well-defined set of responsibilities and caries them all out very well.
- Provides a clear separation of the abstraction's specification and its implementation.
- Is understandable and simple yet extensible and adaptable.

Making a Web Service understandable and simple places a lot of responsibility on the developer. Typically the developer will not be intimately familiar with all the client applications that might use the Web Service, so there needs to be a good set of accompanying documentation describing how to use the Service. This should be a combination of models and textual documents that explain the Web Service from a consumer perspective.

Another useful approach is to view Web Services as a type of *reusable asset*, which is a broad categorization for any set of software development artifacts that offer a solution to a reoccurring problem. A Web Service is, in fact, not only a set of artifacts (i.e., UDDI and WSDL documents) but also an implementation that can be directly accessed by the consuming application.

Reusable assets can be packaged in a standard format: the Reusable Asset Specification (RAS).[13] This specification is currently being developed by a group of companies that includes IBM, Microsoft, and Rational Software Corporation. It describes how to package and deliver generic software development assets in a consistent, yet extendable way, making them easier to apply now and use in different efforts later on. The RAS format includes a categorization section to help identify a particular asset's characteristics and appropriate contexts. It also includes prescriptive guidance on usage, and can include customized automation support. A well-constructed RAS asset will include UML models, documentation, unit test infrastructure, and the actual artifact files (source code, binaries, configuration files, setup scripts, etc.) necessary to solve the problem the asset addresses.

We encourage development teams to package their Web Services as RAS assets, and to include not only the WSDL document, but also additional information expressing the higher-level semantics required to really understand the Web Service. Also, UML models describing the Web Service at various levels of abstraction (use case, analysis/design, and deployment models) would help client developers better understand the motivations and goals behind the design, and therefore how to use the Web Service as intended.

One final recommendation to Web Service development teams: Provide a mechanism for clients to test or evaluate the Service. Include a few test points to verify 1) that the service is actually reachable, and 2) that the Web Service frameworks used on the client environment are suitable -- before the client has to commit to an actual execution of the service. Also, provide test end points that do not necessarily mimic the active Web Service exactly, but that *do* exercise all the types of data structures and state management functions to which a client application would be exposed.

Designing Applications that Use Web Services

Any type of software application that has network access can use Web Services. Web Services themselves are equally broad in scope and can offer any type of service to their clients. The trick to using Web Services

---

[12] Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley, 1999, pg. 59.

[13] For more information see: http://www.rational.com/ras

successfully is understanding what they do and how to make them do it. Basically, there are two main reasons a project architect might choose to use a Web Service:

1. The Web Service manages a critical resource that cannot be accessed programmatically in any other way.

2. It is more cost effective to use a particular Web Service or set of Web Services than it would be to develop duplicate functionality in your own application.

If you are considering a Web Service for the first reason, then all effort should go into understanding the service, both mechanically and semantically.

If you are considering a Web Service for the second reason, then it's important to calculate carefully and confirm that using the service would be truly cost-effective over the application's expected lifetime. It would also be wise to conduct a "good fit analysis" to see whether the client application's need (i.e. problem to be solved) really matches up with the solution provided by the service. If the producer has packaged the Web Service as an RAS asset, then all of the information for the fit analysis should be at hand. The client application team should examine the asset's classification section and look for a specified context that matches the client application's environment. A well-packaged RAS asset will also provide additional human readable documentation that explains the semantics related to the Service and includes a glossary of important terms.

If the Web Service involves the use of other services or sequences of invocations, then look for UML behavioral diagrams (sequence, collaboration, and activity) that detail the sequence of Web Service invocations to accomplish common client tasks. Well-constructed models will make it clear what state management functions the client application is responsible for (i.e., which tokens or data should be kept on the client). UML is a very expressive language and well suited to explaining the details of client / Web Service communication at a level of detail and abstraction appropriate for fit analysis.

Once you decide to use a particular Web Service, it is time to start evolving the client application for that purpose. If your application is in the early stages of development, you can begin by simply creating the appropriate wrapper classes and incorporating the Web Service framework into the design models. For existing applications or applications near completion, it might be appropriate to develop an insulating layer between the core of the client application and the Web Services-backed components. In either case, it is important to tag and identify those elements in the design that are implemented by the Web Service. It is critical that designers and developers alike understand which components are in-process or local, and which are Web Services.

In general, client application development is not fundamentally altered by the use of Web Services. Some consideration of performance- and security-related architectural concerns is in order for performing fit analysis. But making Web Service calls is very similar to making other component calls -- except for the cost. The execution costs for a Web Service are much higher than those for typical component functions, so development teams should always try to minimize the client's use of a Web Service and ensure that it does not make unnecessary invocations.

Summary

Web Services are another tool that application developers can use to promote system-to-system connectivity and exchange information across various platforms and systems. Although today's automated tools make Web Services relatively easy to create and consume, Web Services do involve special technologies that require special attention from development teams and pose unique advantages and challenges.

These challenges can be mitigated by applying the best practices for distributed computing and Web applications. In addition to using SOAP, WSDL and UDDI, the foundation specifications for Web Services, developers can make Web Services applications more understandable and consumable by using UML and RAS. UML is an effective tool for expressing the sequencing involved in Web Services, and RAS is an effective mechanism for packaging and conveying vital information about Web Services.

IBM software integrated solutions

IBM Rational supports a wealth of other offerings from IBM software. IBM software solutions can give you the power to achieve your priority business and IT goals.

- *DB2® software helps you leverage information with solutions for data enablement, data management, and data distribution.*

- *Lotus® software helps your staff be productive with solutions for authoring, managing, communicating, and sharing knowledge.*

- *Tivoli® software helps you manage the technology that runs your e-business infrastructure.*

- *WebSphere® software helps you extend your existing business-critical processes to the Web.*

- *Rational® software helps you improve your software development capability with tools, services, and best practices.*

Rational software from IBM

Rational software from IBM helps organizations create business value by improving their software development capability. The Rational software development platform integrates software engineering best practices, tools, and services. With it, organizations thrive in an on demand world by being more responsive, resilient, and focused. Rational's standards-based, cross-platform solution helps software development teams create and extend business applications, embedded systems and software products. Ninety-eight of the Fortune 100 rely on Rational tools to build better software, faster. Additional information is available at www.rational.com and www.therationaledge.com, the monthly e-zine for the Rational community.

(c) Copyright Rational Software Corporation, 2003. All rights reserved.

IBM Corporation
Software Group
Route 100
Somers, NY 10589
U.S.A.

Printed in the United States of America
01-03 All Rights Reserved. Made in the U.S.A.

IBM and the IBM logo are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Rational, and Rational software are trademarks or registered trademarks of Rational software Corporation in the United States, other countries or both.

Microsoft and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a trademark of The Open Group in the United States, other countries or both.

Other company, product or service names may be trademarks or service marks of others.

The IBM home page on the Internet can be found at **ibm.com**

Part No. TP031A