A Rational software whitepaper from IBM
04/03

Rational. software

Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications

Alan Brown,
Simon Johnston,
Kevin Kelly

Contents

Introduction

Building an enterprise-scale software system is a complex undertaking. Despite decades of technological advances, the demands imposed by today's information systems frequently stretch to breaking point a company's ability to design, construct, and evolve its mission-critical software solutions. In particular, few new systems are designed from the ground up. Rather, a software architect's task is commonly that of extending the life of an existing solution by describing new business logic that manipulates an existing repository of data, presenting existing data and transactions through new channels such as an Internet browser or handheld devices, integrating previously disconnected systems supporting overlapping business activities, and so on.

To assist software developers, commercial software infrastructure products are now available from vendors such as Microsoft and IBM. They form the centerpiece of the approaches to software development they advocate in their .NET and WebSphere product lines, respectively. Both approaches focus on assembly of hsystems from distributed services. However, is there anything new about building enterprise-scale solutions from services? How do the lessons of component-based systems apply to construction of service-based architectures (SOA)? What are the best approaches for building high quality systems for deployment to this new generation of software infrastructure products? These important questions are the topic of this paper.

In recent years, much of the attention in the software engineering community has focused on design approaches, processes, and tools supporting the concept that large software systems can be assembled from independent, reusable collections of functionality. Some of the functionality may already be available and implemented in-house or acquired from a third party, while the remaining functionality may need to be created. In these cases, the whole system must be conceived and designed to bring together all these elements into a single, coherent whole. Today this is exemplified in *component-based development (CBD)*, a concept that is realized in technological approaches such as the Microsoft .NET platform and the Java 2 Enterprise Edition (J2EE) standards supported by products such as IBM's WebSphere and Sun's iPlanet..

An additional consideration is that operational systems will typically be distributed across many machines to improve performance, availability, and scalability. An enterprise solution has to coordinate functionality executing on a collection of hardware. One way to conceive of such a system is to consider it to be

composed of a collection of interacting *services*. Each service provides access to a well-defined collection of functionality. The system as a whole is designed and implemented as a set of interactions among these services. Exposing functionality as services is the key to flexibility. This allows other pieces of functionality (perhaps themselves implemented as services) to make use of other services in a natural way regardless of their physical location. A system evolves through the addition of new services. The resulting *service-oriented architecture (SOA)* defines the services of which the system is composed, describes the interactions that occur among the services to realize certain behavior, and maps the services into one or more implementations in specific technologies.

While the services encapsulate the business functionality, some form of inter-service infrastructure is required to facilitate service interactions and communication. Different forms of this infrastructure are possible because services may be implemented on a single machine, distributed across a set of computers on a local area network, or distributed more widely across several companies' networks. A particularly interesting case is when the services use the Internet as the communication mechanism. The resulting *Web services* share the characteristics of more general services, but they require special consideration as a result of using a public, insecure, low-fidelity mechanism for inter-service interactions.

Much of the software industry's focus so far has been on the underlying technology for implementing Web services and their interactions. However, additional concerns arise around the question of the most appropriate way to design Web services for ease of assembly into enterprise-scale solutions. Conversely, there has been a surprising lack of attention on appropriate practices and tools for architecting enterprise-scale software solutions composed of Web services. As with the design of any complex structure, high-quality solutions are the result of early architectural decisions supported by a number of well-understood design techniques, structural patterns, and styles. These patterns address common service issues such as scalability, reliability, and security.

This paper provides the context for a deeper understanding of services and service-oriented architectures for enterprise-scale software solutions. In particular, it explores services in relationship to the more established concept of software components, and it describes how current component-based development practices provide a tried and tested foundation for the implementation of a service-oriented architecture. Interface-based design is highlighted as the key to both service and component design, and it is argued that the interfaces exposed by both have certain constraints and criteria that distinguish them. The Unified Modeling Language (UML)[1] is used as a tool to describe both logical and implementation designs, as well as specific patterns for both component and service design.

Finally, the paper focuses on Web services as a vehicle for exploring issues related to the implementation of services in general. Specifically, the paper investigates how to interpret the general guidance for service-oriented architecture as specific design patterns for Web services. All of the patterns described in this paper have been implemented using the unique pattern and code template capabilities of Rational® XDE™ and will be published through the Rational Developer Network[8].

What Is a Service-Oriented Architecture?

So what is a service-oriented architecture (SOA)? In essence, it is a way of designing a software system to provide services to either end-user applications or other services through published and discoverable interfaces. In many cases, services provide a better way to expose discrete business functions and therefore an excellent way to develop applications that support business processes.

Service-oriented architecture is not a new notion; it is important at this time because of the emerging Web services technology. For example, here is a quote from a book published in 2000 describing the value of a service-oriented architecture:

> *Service-Oriented Solutions… Applications must be developed as independent sets of interacting services offering well-defined interfaces to their potential users. Similarly, supporting technology*

must be available to allow application developers to browse collections of services, select those of interest, and assemble them to create the desired functionality. [2]

For the purposes of this document, we shall consider the following definition of a service:

A service is generally implemented as a course-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model.

In many ways, the terminology for services is much the same as the terminology used to describe component-based development; however, there are specific terms used to define elements within Web services, as shown in Figure 1 below.

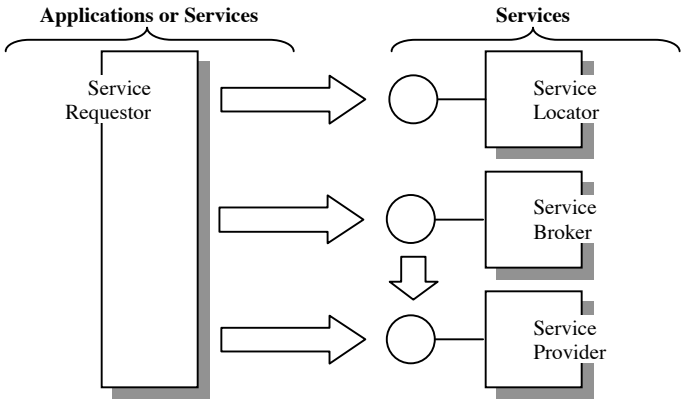

Figure 1 – Service terminology

- **Service** — A logical entity; the contract defined by one or more published interfaces.
- **Service provider** — The software entity that implements a service specification.
- **Service requestor** — The software entity that calls a service provider. Traditionally, this is termed a "client"; however, a service requestor can be an end-user application or another service.
- **Service locator** — A specific kind of service provider that acts as a registry and allows for the lookup of service provider interfaces and service locations.
- **Service broker** — A specific kind of service provider that can pass on service requests to one or more additional service providers.

This description of services, and the context of their use, imposes a series of constraints. Furthermore, efficient use of services suggests a few high-level best practices. Here are some key characteristics for effective use of services:

- **Coarse-grained** — Operations on services are frequently implemented to encompass more functionality and operate on larger data sets, compared with component-interface design.
- **Interface-based design** — Services implement separately defined interfaces. The benefit of this is that multiple services can implement a common interface and a service can implement multiple interfaces.
- **Discoverable** — Services need to be found at both design time and run time, not only by unique identity but also by interface identity and by service kind.
- **Single instance** — Unlike component-based development, which instantiates components as needed, each service is a single, always running instance that a number of clients communicate with.
- **Loosely coupled** — Services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods such as XML document exchanges.
- **Asynchronous** — In general, services use an asynchronous message passing approach; however, this is not required. In fact, many services will use synchronous message passing at times.

Some of these criteria, such as interface-based design and discoverability, are also used in component-based development; however, it is the sum total of these attributes that differentiate a service-based application from an application developed using component architectures such as a J2EE or .NET.

Interface-Based Design

In both component and service development, the design of the interfaces is done such that a software entity implements and exposes a key part of its definition. Therefore, the notion and concept of "interface" is key to successful design in both component-based and service-oriented systems. The following are some key interface-related definitions:

- **Interface** — Defines a set of public method signatures, logically grouped but providing no implementation. An interface defines a contract between the requestor and provider of a service. Any implementation of an interface must provide all methods.
- **Published interface** — An interface that is uniquely identifiable and made available through a registry for clients to dynamically discover.[3]
- **Public interface** — An interface that is available for clients to use but is not published, thus requiring static knowledge on the part of the client.
- **Dual interface** — Frequently interfaces are developed as pairs such that one interface depends on another; for example, a client must implement an interface to call a requestor because the client interface provides some callback mechanism. This concept was introduced by Web services.

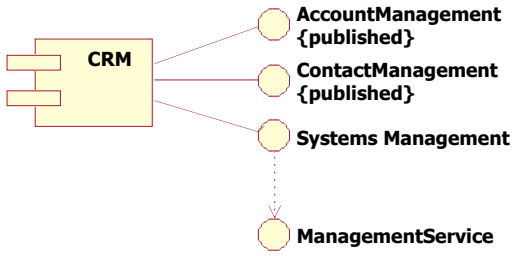

Figure 2 – Implemented services

Figure 2 shows the UML definition of a customer relationship management (CRM) service, represented as a UML component, that implements the interfaces AccountManagement, ContactManagement, and SystemsManagement. Only the first two of these are published interfaces, although the latter is a public interface. Note that the SystemsManagement interface and ManagementService interface form a dual interface. The CRM service can implement any number of such interfaces, and it is this ability of a service (or component) to behave in multiple ways depending on the client that allows for great flexibility in the implementation of behavior. It is even possible to provide different or additional services to specific classes of clients. In some run-time environments such a capability is also used to support different versions of the same interface on a single component or service.

Interface Behavior

An interface definition in languages such as Java or C#, or in languages such as IDL, only provides a set of method signatures. The definition provides the "what" without any guidance on the "how." For example, consider the Security interface in Figure 3. It is clear that the clients calling an implementation of this interface are able to call any of three public methods — or are they?

Figure 3 – Interface in UML

By simply defining the "what" we are not able to portray the fact that the client is unable to call GetUserName( ) or GetUserDomain( ) until the user has logged on. The following state machine demonstrates this dependency, or behavior. This kind of constraint is often included in literature on interface-based design; however, it is not supported in any programming languages, so one cannot ensure that the implementer of an interface is compliant with any behavioral specification.

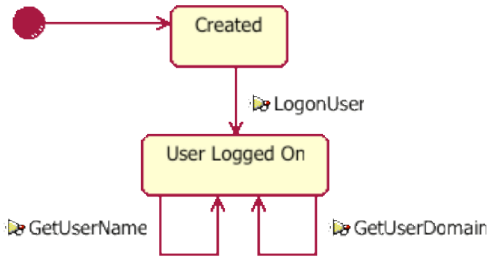

Figure 4 – Interface behavior

However, businesses are moving more and more to service-oriented systems in the hope that they can be more easily integrated and choreographed to realize business processes through collaborations of services. As a result, the notion of defining the behavior of an interface and, more importantly, the behavior of sets of related interfaces is receiving increasing industry attention. Unfortunately, there are currently few standard approaches to this.

One approach might be to use design models such as those introduced through this paper, defined in a standardized language such as the UML to document the interdependencies between service interfaces. Such models can be shared, socialized, and used to drive specific standards when they emerge.

Additionally, Rational has sponsored the Reusable Asset Specification (RAS), which provides a mechanism for packaging and sharing assets that could be applied to this problem. For example, when using the RAS mechanism to distribute the details for a service, one could package the model describing its behavior as well. Within such a model, a sequence diagram may then be used to show the required interaction between the calls on the interface.

Architecting Service-Oriented Systems

In any new development in software engineering, it is very easy to assume that one can apply the same techniques and tools that have worked in previous projects. We have already mentioned that components and services, although similar, are not the same; they have differing design criteria and different design patterns. This section will discuss an important practical consequence. *The bottom line is that not every good component transformed into a service makes a good service.*

Layering Application Design

This tendency to solve new problems with outdated solutions is not new. Similarly, as developers began to create component-based systems, they tried to bring to bear their experience with object-oriented development, with similar problems. With more experience, it was understood that object-oriented technology and languages are great ways to implement components, though one has to understand the trade-offs made through decisions and implementation. Trade-offs concerning inheritance vs. aggregation for implementing polymorphic behavior, or redesign of class libraries to be able to use in sets of components rather than as the base for a monolithic C++ application.

In a similar way, we see *components as the best way to implement services*, though one has to understand that an exemplary component-based application does not necessarily make an exemplary service-oriented application. We see a great opportunity to leverage your company's component developers and existing

components, once the role played by services in application architecture is understood. The key to making this transition is to realize that a service-oriented approach implies an additional application architecture layer. Figure 5 below demonstrates how technology layers can be applied to application architecture to provide more coarse-grained implementations as one gets closer to the consumers of the application. The term coined to refer to this part of the system is "the application edge," reflecting the fact that a service is a great way to expose an external view of a system, with internal reuse and composition using traditional component design.

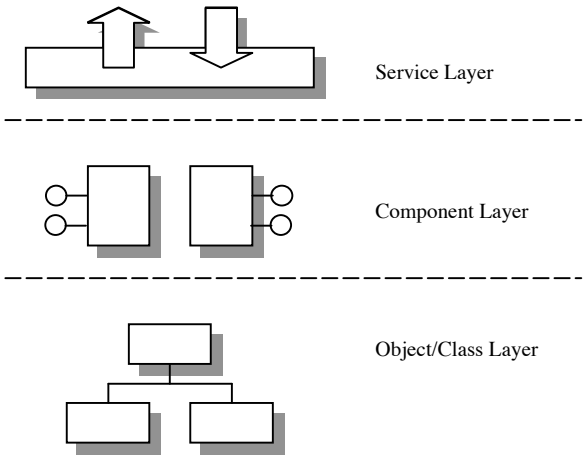

Figure 5 – Application implementation layers

In general, the move from object-oriented to component-based thinking took between 6 and 18 months as developers learned about this new technology and the requirements it placed on them. Hopefully, the move to service-oriented systems can happen more quickly. To this end, developers will have to understand the challenges, trade-offs, and design decisions that allow for the development and reuse of components in support of service-oriented applications.

Example Customer Model

To illustrate how components and services interact in realizing an application, consider an example that manages some customer relationship information, as defined in the UML class diagram in Figure 6 below.

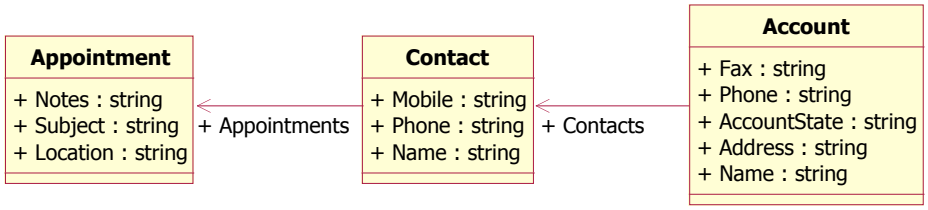

Figure 6 – Logical customer model

In the sections that follow, we will describe how developers would transcribe such a logical model into an implementation model for component-based applications and then for service-based applications (bearing in mind that this model does not describe the behavior of the system in any way). It will become clear that many of these translation steps can be automatically generated. Rational Software has tools for the modeling of the architecture of applications, the harvesting and application of such patterns, and the management of these model/code artifacts through the complete development lifecycle.

A Component-Based Design

How would our example logical model be translated into a component design (such as for COM or J2EE)? A component-based design for the model is presented below, in Figure 7.

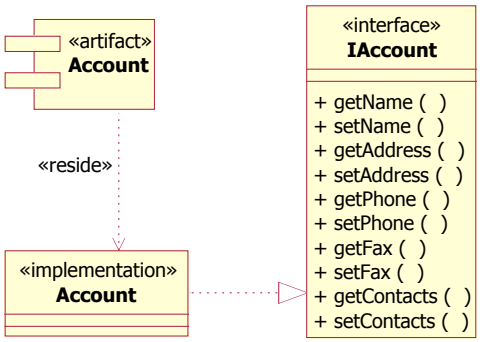

Figure 7 – Generic component diagram

There is one *key* design feature of the interface above. The key is that for existing component platforms there is a common pattern; for each attribute in the analysis class we have to provide two operations — one operation that sets the value and one that returns the value. For local components the overhead of a method call is negligible and for remote objects the Remote Procedure Call (RPC) mechanisms are optimized to minimize overhead. In many cases in an application the client only needs a subset of the properties and so can access them as needed.

A Service-Oriented Design

The design model above demonstrates the correct way to think about component implementations where each component instance represents a single object; for example, for each individual contact in our CRM database becomes, logically, a separate component. So component identity is tied to the identity of the contact.

However, for a service there is a single instance that manages a set of resources, so they are for the most part stateless. This means that we need to view a service as a *manager* object that can create and manage instances of a type, or set of types. This yields a design pattern that makes use of *value objects* (a common pattern in distributed systems where state persists for transfer between components) that represent the instance state, objects that are in fact simply serialized state. One interesting result of this is that, if we can define the rules for taking a component definition, such as the model in Figure 7, and transforming it into a service, we can implement this serialization as a pattern. The creation and reuse of such patterns is possible using Rational XDE.

This passing of state from provider to requestor implies that rather than a large number of small operations to retrieve the component state, a single large operation is used. Now this has certain implications for network usage for remote services (and most services are remote), as well as the behavior of requestors when dealing with large value objects. This also has another implication; the requestor is being provided with a copy of the state of some entity, but is this copy stale? We know when we retrieve a stock quote or weather forecast that there is the possibility that these are out of date, but we are conditioned to accept this. We are also conditioned by the type of data; stock quote data becomes stale faster than weather data. In such an architecture as described here, the requestors must be conditioned to accept copies of state. For details, see the “Impact of Coarse-Grained Interactions” section later in this paper.

Given these requirements, how would we expect a service to look? The model fragment in Figure 8 shows how such a pattern can be described at a design level, demonstrating the interfaces published by the component and the value objects that the interface manipulates.

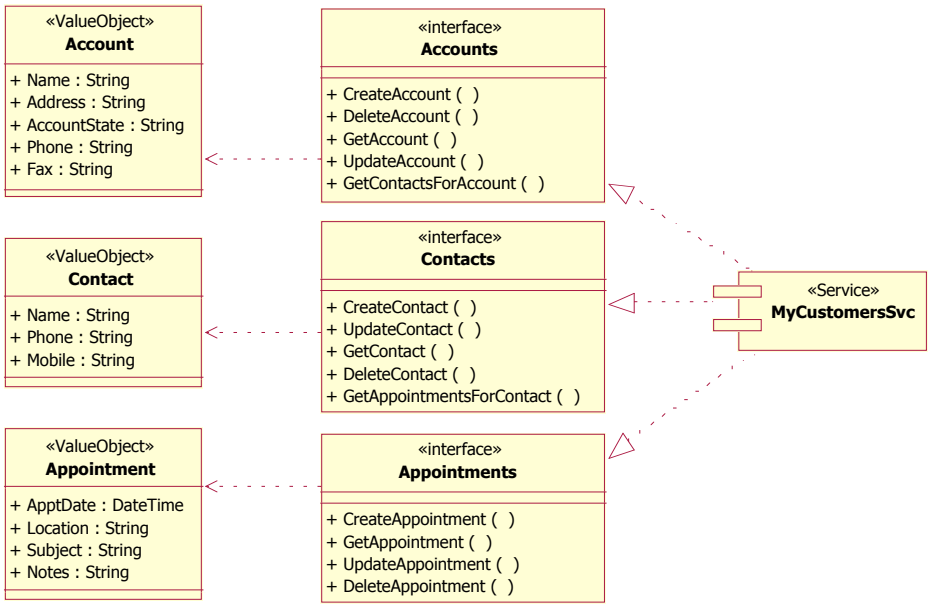

Figure 8 – Generic service-oriented design

Now that we see how services can be designed, let us look at some attributes of this particular design. To begin with, it is clear that there is a lot more information passed in the value objects than a simple "get" or "set" from the provider, MyCustomerSvc, to a requestor for a given interaction, and that this can affect network bandwidth. However, given the nature of Web services, it is clear that the protocols used in the implementation of services differ greatly from those used in component implementations. A platform such as this puts an additional burden on the architect or information engineer to carefully choose the value objects and their composition to maximize the content of each value object while not bringing the network to its knees.

Caching in Service-Oriented Design

Let us reconsider the notion we introduced in the previous section of passing "stale" copies of information from a provider to a requestor. For example, if I am developing a stock portfolio management application, I do not want to ask a Web service for the current price of a security over and over for each security; passing 3-5 characters of data for the security and 5-7 characters for the price back. This may result in an unacceptable load on the network and service provider. What the requestor should do is to request the contents of the entire portfolio, either by passing the list of symbols or by passing the portfolio identifier to the service and retrieving all information for each security. Now, if the user had simply asked for an update to a single symbol this seems like overkill; however, the requestor can now cache the results and, if the user then asks for an update to another symbol, the request can be satisfied from the cache. The work on the requestor is now to identify the "lease" duration of the data. So for a portfolio, if I know the stock quote service has a 20-minute delay, I might like to work on a 25% margin and cache the results for 5 minutes.

This pattern is seen over and over again in information systems. Whenever a user retrieves an order from an order management system, that user is effectively given a copy of the order because another user may be updating it at the same time (unless the system locks out additional access to the order). Now, it would be nice if a Web service provider could actually, as part of its interaction with the requestor, identify the cache or lease duration. Such issues are well understood in messaging systems like Microsoft Message Queue Server (MSMQ) and IBM MQSeries, where message timeouts and expiry times are routinely managed.

We will revisit this problem later in this document and provide some specific guidance on the development of both requestors and providers to cope with this issue.

XML Web Services Application Design

XML Web services are now a part of our vocabulary and there is broad vendor support as well as a growing number of platforms and tools to deploy and develop Web services. The elements of the Web service stack have been well described elsewhere, so we won't go into an exhaustive review in this document. The following definition comes from the World Wide Web Consortium (W3C) Web Services Architecture Working Group:

> *A web service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet-based protocols.*[4]

Let us briefly consider how some of the technologies in Web service use today are applied to the picture we drew in Figure 1 when we described service-oriented architecture terminology.

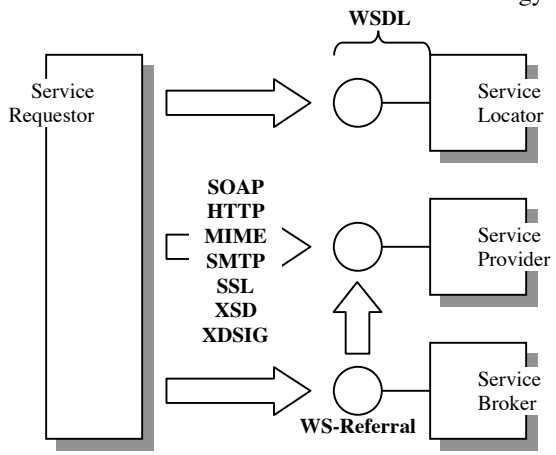

Figure 9 – XML Web service standards

For a start, there is a misconception that all Web services use XML messages, with Simple Object Access Protocol (SOAP) over HTTP — the de facto protocol of the Web. This is not quite true. A Web service message can use XML, but it might transport binary data; it generally uses SOAP headers, but it is not required to use SOAP encoding for message bodies; it may use HTTP as a transport, but it could well use SMTP or other means. For the description of, and discovery of, Web services there are two well-defined standards: WSDL (Web Services Definition Language) and UDDI (Universal Discovery, Description and Integration).

This flexibility in formats and transport protocols is one of the current issues with Web services — interoperability. How, given the choices in SOAP formats, envelopes, transport protocols, and so on, can two implementations transfer information? Stepping into this is the Web Services Interoperability (WS-I) group, which is attempting to provide the industry with guidance on the usage of the current and emerging standards. Vendors are also helping here by providing flexible Web services development environments such as IBM WebSphere Studio Application Developer Integration Edition, which creates Web services with multiple formats and transport protocols so that the fastest or correct set can be used as required.

Web Service Design and Implementation Patterns

Web services really do not change the analysis and design process around the functional requirements for an application — an insurance claim processing application still has to process insurance claims! What we are introducing are a set of constraints and potential issues in the area of nonfunctional requirements. The following sections describe some of these possible pitfalls.

Performance and Reliability

The question is often asked whether the capabilities required for Web service performance, reliability, and scalability can be provided by an architecture based on HTTP and SOAP, which are inherently slow and unreliable. First, "slow and unreliable" must be defined, then it must be realized that even reliable transports at the end of the day rely on unreliable means. When architecting and designing enterprise-scale solutions, we must always bear in mind the functional and nonfunctional requirements and ensure that the correct trade-offs and decisions are made to support the business goals.

For example, when using SOAP over HTTP it is always possible to build application-level protocols and interactions that provide additional capabilities for message acknowledgements and security. However, if one were to consider that certain services communicate within the same security or application context, we might consider using different means than HTTP anyway. Consider the example in Figure 10.

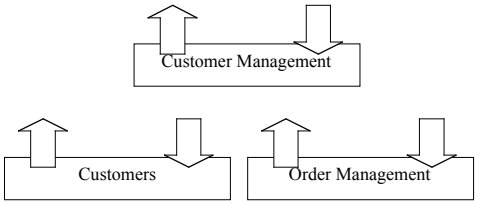

Figure 10 – Service dependencies

Basically, all external clients interact with the Customer Management service; however, it interacts with two internal services. The decision here is, Why would we require the flexibility of HTTP and SOAP for these internal service communications? Let us assume that performance was our key requirement for the interaction between Customer Management and Customers. If so, we might decide to use a component RPC communication (such as Microsoft .NET Remoting or Java's RMI) that provides binary encoding formats and higher performance characteristics. On the other hand, the key requirement to place an order from Customer Management to Order Management is for guaranteed delivery, so we might use some queuing technology (such as IBM MQSeries or MSMQ) to deliver the message where performance is traded for a higher level of reliability.

It is very important to realize that even though Web services present a simple model and a set of simple, flexible protocols, you are not restricted to these choices. Just as WSDL already has bindings for both SOAP and HTTP GET/PUT, it is important to provide requestors with additional choices. For example, a single service may expose a message using a message queue binding *and* a SOAP binding, so that the requestor can then choose which is the more appropriate binding to use. In this case, the provider may also provide incentives, such as a guaranteed service level if the message queue is used but no service guarantees for an HTTP conversation.

Another design decision that one should make early on is whether message exchanges will be idempotent, commutative, or both. To be *idempotent* implies that if the same message arrives more than once, and *is acted on*, in each case there are no ill effects. To be *commutative* means that two related messages can arrive in any order with no ill effects. If the design of a service can be such that message exchanges are identified as at least idempotent, then less reliable transports are a more attractive (and cheaper) option.

In the same way that security (not discussed in this paper) is actually a set of choices, ranging from simple and cheap to complex and expensive, the design goals of performance, reliability, and scalability come down to a set of decisions: How much do you need? How much can you afford? Services provide as many solutions, and yet as many choices, as existing development approaches.

Scalability Through Asynchronous Behavior and Queuing

As mentioned in the introduction to service-oriented architecture, it is beneficial to make your Web services asynchronous in nature. Because of the additional transport overhead associated with Web services and the expectation that services will, by their nature, be remote, it is important to reduce the time a

requestor spends waiting for responses. By making a service call asynchronous, with a separate return message, we allow the requestor to continue execution while the provider has a chance to respond. This is not to say that synchronous service behavior is wrong, just that experience has demonstrated that asynchronous service behavior is desirable, especially where communication costs are high or network latency is unpredictable.

Consider the example in Figure 11.

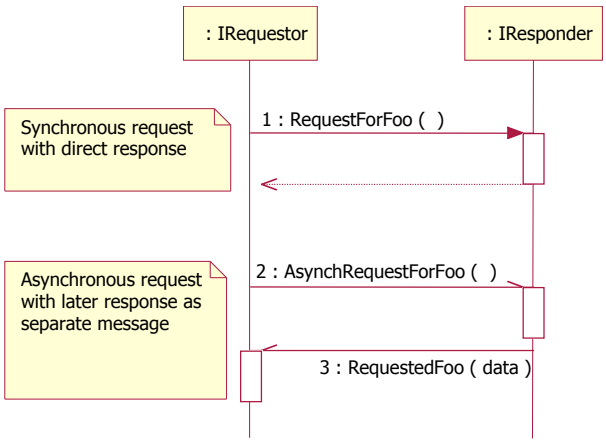

Figure 11 – Synchronous vs. asynchronous

The behavior described in Figure 11 is a big step toward implementing highly scalable Web services. By making a service call asynchronous, it allows the provider to use multiple worker threads to handle multiple client requests. There is a lot more to be done to support an asynchronous mode of operation than just returning to the client quickly. First, you need to specify dual interfaces; the requestor will need to pass in a return address to a service that implements an interface that can accept the returned message. This implies a need to manage state in the conversation between the parties. To learn about various methods for doing this, look at the design of Web sessions that are not based on Web services.

However, this is only scalable to a certain degree. For services that expect a very high load, we would need to decouple the part that listens to the requestor and the part that services the request itself. This is already a well-known pattern, in which a message queue is used to decouple a service façade from the service implementation. The diagram in Figure 12 shows how the provider is implemented using a queue to decouple the request from the implementation.

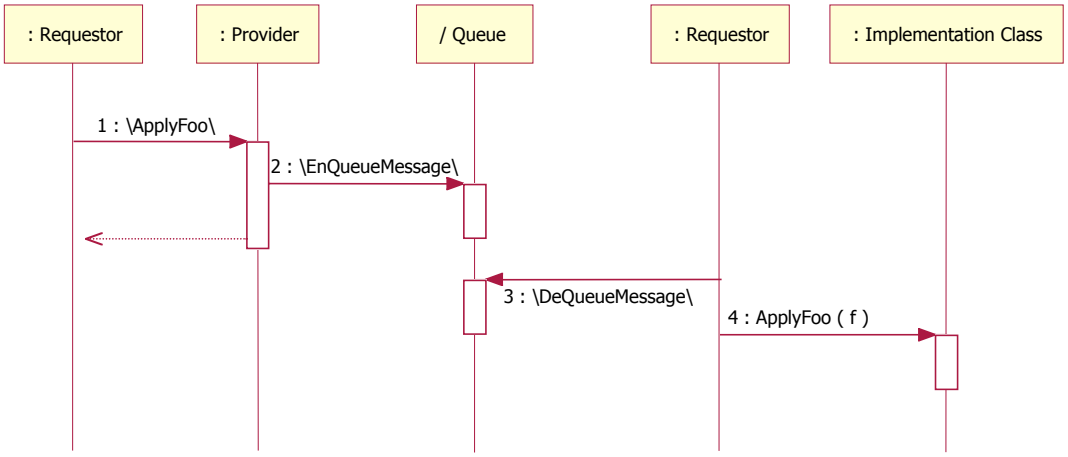

Figure 12 – Queued implementation

Such a pattern can be easily implemented in both .NET and J2EE using services provided by those platforms: MSMQ for .NET and Java Message Queue Service (JMS) or message-driven beans for J2EE. This provides the developer with a simpler scalability model; rather than handling a set of threads with synchronization for the requests, the implementation can simply add additional queue listeners to pick messages from the queue, even across multiple machines.

Information Leasing Revisited

When we think about leasing information, we view this more in terms of the lending of a book from a library than the leasing of property such as a house or car. Implicitly, whenever a requestor makes a request of a service it is asking for a copy of some information; it is always provided with only a snapshot of state at a given time. Now this can be a problem unless it is explicitly understood and accounted for. One strategy is to have the provider give the *expiration* time together with the information. Alternatively, the requestor may get a "ticket" with the lease (like a library book) that would allow it to potentially extend the lease by asking if the information is still valid, and then have the server reset the lease without having to retrieve the data again.

Now, this seems like such a fundamental issue that one might expect that HTTP, SOAP, or one of the transport protocols would handle this for us. We could reuse the HTTP caching semantics that allow browsers and firewalls to cache pages, but it really isn't under the provider's control, and the requestor may not be using HTTP as a transport. One option is to build such support into your document exchange, such that the messages between requestor and provider encode the leasing information for the client, as shown in Figure 13.

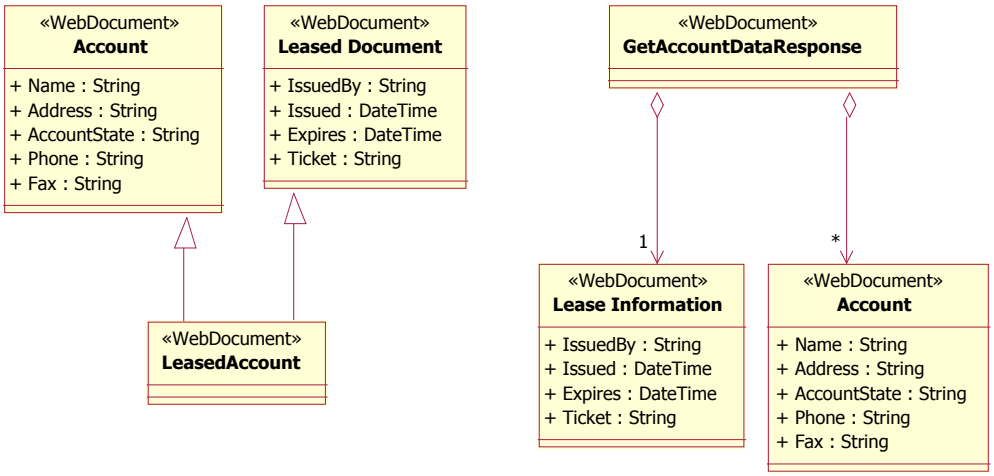

Figure 13 – Two implementations of information leasing

Figure 13 demonstrates two alternative implementations of the information-leasing pattern. The first demonstrates the use of inheritance to specialize the Account XML document into a special form that is not only an Account but also a Leased Document and, therefore, includes the additional information. The second alternative has the leasing information returned alongside the account as a separate part of the response message. Both of these approaches are equally valid, they do result in differently structured data and the choice is very much one of style, inheritance versus aggregation.

Resulting Web Service Design Model

Figure 14 demonstrates how the generic service design in Figure 7 can be modeled using a UML profile specific to Web service design and development. The profile is very simple and introduces only two new *stereotypes* (extensions to the existing UML language), for «WebService» and «WebDocument». By reusing the interface semantics already present in the UML, we can easily visualize the published aspects of a service, as defined in WSDL.

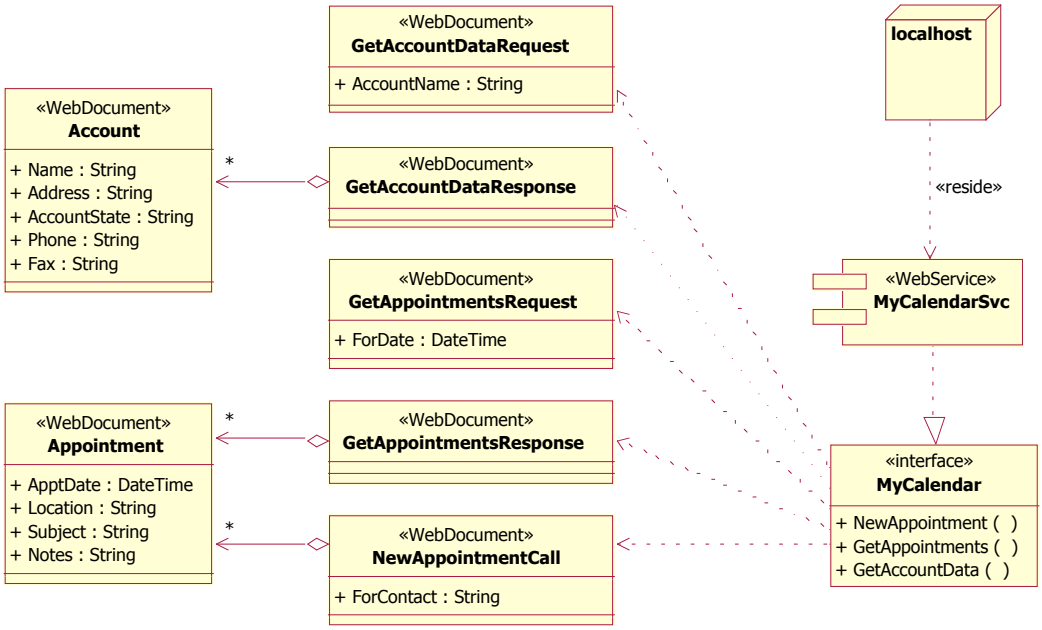

Figure 14 – XML Web service design

The table below demonstrates how the profile elements in Figure 14 are related to the artifacts in the WSDL for the MyCalendar service.

WSDL Artifact	UML Element	Comment
service	«WebService»	The service is represented as a UML component that realizes one or more interfaces and resides at a particular location. The «reside» relationship will capture the actual URL location information.
portType	Interface	Each portType is represented as a UML interface realized by one or services. The realization relationship will capture the binding information.
message	«WebDocument»	Each message is represented as a UML class. A mapping from XML Schema to and from UML is required to model the message and part structure.
part	Attribute or Association End	Each part of the message can be represented either as a UML attribute on the «WebDocument» or as an association to another «WebDocument».
address location	Node	The node represents the server on which the service resides. The node may identify a set of resident services and a service may reside on more than one node.

It is important to note that this method for designing Web services promotes the reuse of both the documents that define the data exchange and the interfaces supported by the services. This is a key capability when designing enterprise-scale solutions. Preferably, all services that interact with the Account document do so based on the same document definition. We also see that operational, unpublished interfaces, such as SystemsManagement seen in Figure 2, can be defined by experts in this area and then made available such that they are implemented in common across the solution.

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.

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.

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