Realizing service-oriented solutions with the IBM Rational Software Development Platform

A. W. Brown
M. Delbaere
P. Eeles
S. Johnston
R. Weaver

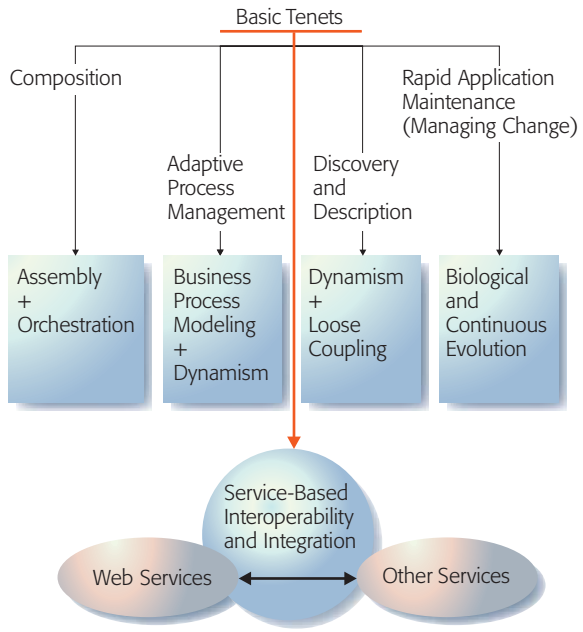
Creating service-oriented architecture (SOA) solutions means rethinking the practices currently in use to build systems, reconsidering the skills in an organization, and redefining the ways in which team members collaborate. A service orientation contributes to the development of solutions that are assembled from disparate applications, and SOA is an architectural style that emphasizes loose coupling of independent service providers. This perspective on service orientation is known as service-oriented development of applications (SODA). SODA encompasses composition, adaptive process management, service-based interoperability and integration, discovery and description, and rapid application maintenance. In this paper, we focus on how IBM supports SODA, the relationship of SODA to the IBM Rational Software Development Platform (RSDP), and how IBM's core approach to design and construction—model-driven development—is an essential element in creating effective and efficient services and service-oriented solutions. We explore the concepts behind these approaches and illustrate their realization with illustrative examples from customer experiences.

INTRODUCTION

Building enterprise-scale software solutions has never been easy. The difficulties of understanding highly complex business domains are typically compounded with all the challenges of managing a development effort involving large teams of engineers over multiple phases of a project spanning many months. The time-to-market pressures inherent in many of today's product development efforts only serve to exacerbate these problems.

In addition to the scale and complexity of many of these efforts, there is also great complexity in the software platforms for which enterprise-scale software is targeted. Most large IT (information technology) organizations rely on a complicated assortment of infrastructure technologies that have evolved over many years, consist of a variety of middleware acquired from many vendors, and have been assembled through various poorly documented integration efforts of varied quality.

To develop applications in this context requires an approach to software architecture that helps architects evolve their solutions in flexible ways, reusing existing assets in the context of new capabilities that



Reprinted with permission of Gartner, Inc.

Figure 1
Service-oriented development of applications

implement business functionality even as the target infrastructure itself is evolving. Service-oriented architecture is an important idea that is central to addressing these issues.

Service-oriented architecture

An approach gaining support in the industry today is based on viewing enterprise solutions as federations of services connected by well-specified contracts that define their service interfaces. The resulting system designs are frequently called service-oriented architectures (SOAs). Many organizations now express their solutions in terms of services and their interconnections. The ultimate goal of adapting an SOA is to achieve flexibility for the business and IT.

For purposes of this paper, we use the following definition of a service: it 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.

A number of important technologies have been defined to support an SOA approach, most notably when the services are distributed across multiple machines and connected over the Internet or an intranet. For example, Web Services approaches rely on intraservice communication protocols, such as the Simple Object Access Protocol (SOAP); enable the Web Services interfaces expressed in WSDL (Web Services Description Language) to be registered in public directories and searched in UDDI (Universal Description, Discovery and Integration) repositories; and share information in documents defined in XML (Extensible Markup Language) and described in standard schemas.

Of course, SOA is more than a set of standards and service descriptions in the same way that object-orientation is more than a set of class hierarchies. The essence of SOA is that it emphasizes loose coupling of independent service providers. In fact, it may be implemented by using a range of different technology choices. Indeed, it is possible to create an SOA that does not use Web Services technology, and it is possible to use Web Services technology in a way that would not be considered service-oriented. More important, a number of fundamental concepts become primary when implementing a solution using SOA (e.g., the concept of "service leasing" to support flexibility between service provider and consumer).[1] There is a great deal more that needs to be explored to understand why a service-oriented viewpoint adds value to a business and how service-oriented solutions are designed, implemented, deployed, and managed.

Service-oriented development of applications

Creating solutions for SOA means rethinking the kinds of systems being built today, reconsidering the skills in an organization, and redefining the ways in which members of teams collaborate. Most important, adopting a service orientation to the development of solutions requires a broader review of the impact of this orientation on how solutions are designed, what it means to assemble them from disparate services, and how deployed service-oriented applications are managed and evolved.

Gartner refers to this broader context of service orientation as Service-Oriented Development of Applications (SODA).[2] Gartner considers the five key areas (or "tenets") of SODA to be composition, adaptive process management, service-based interoperability and integration, discovery and description, and rapid application maintenance. In *Figure 1*, we see the primary elements of SODA as defined by Gartner. From a tool vendor perspective,

these areas relate to technology support offered in three areas:

1. *The SOA life cycle*—The "discovery and description" and "rapid application maintenance" tenets refer to the life cycle of services and how they are found, applied, evolved, and maintained. Tool vendors increasingly offer ways to store, catalog, search, and retrieve services. Support for the ongoing evolution of services is a critical aspect of this process, leading to multiple versions of services.
2. *SOA platform and programming model*—The "service-based interoperability and integration" tenet refers to the way services can be connected, deployed, and managed within a specific runtime platform. The major platform vendors are supporting service-oriented capabilities directly as part of their middleware runtimes and evolving their runtime programming models to present service concepts as first-class elements. As a result, solutions may be conceived, designed, implemented, and managed from a single service-based perspective.
3. *SOA practices and tools*—The "composition" and "adaptive process management" tenets refer to how services are created and assembled in the context of changing business needs. Tool vendors support the mining of existing applications to discover potential services, wrapping existing functionality to make those capabilities accessible as services, creation of new services, and "wiring together" of services by connecting behavior exposed through their interfaces. Fundamental to this is the availability of clear guidance and best practices for designing service-oriented solutions in repeatable, predictable ways.

All three of these areas are important for success in developing service-oriented solutions. They must all be addressed to meet an organization's needs in efficiently creating more flexible solutions that better align with the goals of the business.

IBM's role in services and SOA

IBM has a leading role in the definition and application of SOA and SODA. IBM recognizes the value that SOA brings to organizations that require greater flexibility and control of the solutions they deploy. IBM demonstrates its support for service-oriented approaches to development by driving standards activities, delivering tools and technologies, and documenting best practices.

Driving standard activities

IBM has taken a leading role in defining the key standards that allow services (in particular Web services) to be defined, registered, managed, and discovered. For example, standards such as WSDL, UDDI, the Business Process Execution Language for Web Services (BPEL4WS or BPEL for short), and related standards for security (WS-Security) and interoperability (WS-I) provide a greater measure of interoperability among service-oriented applications and encourage a healthy marketplace of tools and technologies. In addition, IBM is driving standards for modeling services and service assemblies, such as the Unified Modeling Language** (UML**), and working on numerous Java** standards that support realization of services and service-oriented solutions in the context of the J2EE** (Java 2 Enterprise Edition) programming model, such as the Java ServerFaces (JSF) standard and the emerging Service Data Object (SDO) standard.

Delivering tools and technologies

As these standards are defined, they are supported in commercially available IBM products. For example, the IBM Rational* Application Developer includes support for a number of key Web Services standards, such as WSDL, UDDI, WS-I and WS-Security. Creating services and service-oriented solutions is automated through a wide collection of rapid application-development features based on wizards, patterns, and reusable assets. These help to automate efficient delivery of services and service-oriented applications in conformance with applicable standards.

Documenting best practices

IBM has substantial experience with the application of service-oriented approaches across many business domains. This knowledge is being captured and exploited through IBM's service offerings and engagements, guidebooks, tutorials, and reusable samples. For example, IBM's developerWorks*, Redbooks*, and the Rational Unified Process* contain a wealth of materials for those using SOA or practicing SODA.[3] This guidance is continually being refined and updated as IBM's experience base grows.

Although there is much to be discussed in all of these areas, in this paper we focus on how IBM

supports SODA, the relationship of SODA to the IBM Rational Software Development Platform (RSDP), and how IBM's core approach to design and construction—model-driven development—is an essential element in creating efficient and effective services and service-oriented solutions. For more details, see the general references at the end of this paper.

TOWARD SERVICE-ORIENTED SOLUTIONS

Adopting an SOA and applying SODA techniques is important in creating the kinds of solutions that will drive the next generation of business process improvements. A growing number of organizations are looking to adopt a different kind of software development platform that recognizes a new approach to the role software plays in their business. Many of these organizations face the pressure of quantitatively showing the value that IT provides to their business.

There are three imperatives influencing the way organizations are looking at the platforms of their next-generation solutions: a service view, rapid assembly and reassembly, and a focus on asset management and reuse.

A service view

Additional insight and understanding of how business processes can be realized in IT solutions can be obtained by viewing the collection of capabilities offered throughout an IT infrastructure as a set of services that are assembled to meet specific business needs. System architectures are designed as collections of services governed by interservice protocols and explicit SLAs (service level agreements).

Rapid assembly and reassembly of solutions

Greater flexibility can be offered by treating an IT organization as a "software factory" for creating solutions to meet evolving business goals. To achieve this, it must be possible to readily assemble and reassemble pieces and parts of the solutions as business and market conditions demand. This requires close relationships between business analysts and IT architects, tools to promote collaboration, and a disciplined approach to managing the elements that are assembled.

A focus on asset management and reuse

As organizations seek to obtain greater business efficiency, there has been increased emphasis on reuse as a principle that applies throughout the software development life cycle. In particular, the limited impact of reuse through shared code libraries has been broadened to include reuse of business processes, requirements definitions, architectural design elements, test scripts, and so on. This view changes the solution life cycle in substantial ways, altering the roles of individuals in the organization and creating different project practices, as well as creating and managing assets throughout the life cycle. For practical reasons, this is accompanied by strong governance practices for reusable assets tied to a flexible asset management infrastructure.

Supporting this new view of software development requires tools and platforms that take a service perspective and provide a business focus that ties the business and IT practitioners together more effectively. Rather than extending object models, practitioners must think in terms of "wiring of services." They must take advantage of exposed service-based middleware capabilities and facilitate greater management and reuse of solution fragments. The result is different kinds of solutions, different roles, different development processes, and different expectations from the tooling.

Tool vendors and middleware software vendors have recognized these requirements and are offering capabilities to help fulfill them. In this regard, one of the most important developments from IBM has been the consolidation of a collection of these capabilities as the IBM Rational Software Development Platform (RSDP), a comprehensive set of offerings for developing, deploying, and managing service-oriented solutions.

THE IBM RATIONAL SOFTWARE DEVELOPMENT PLATFORM

IBM offers many valuable technologies to help organizations design, build, deploy, and manage service-oriented solutions. Although individual product capabilities are important, the real value to customers is the combination of these capabilities in a robust software development platform for creating a new generation of applications. Many organizations are seeking a set of capabilities for executing IT projects with a level of coordination, accuracy, and clarity that is currently unavailable. In fact, the role of IT in an organization is seen as a core "utility"; investment in IT resources is seen to provide a

predictable, risk-managed impact on the goals and mission of the business.[4]

Consequently, organizations are beginning to view software development as a "business process" that itself must be measurable, predictable, and manageable. This is a compelling vision, and one that can be delivered only through the deep integration of tool and runtime capabilities throughout all aspects of the business in support of a service-oriented view of their solutions. In this regard, the RSDP is a critical step. It offers the tooling and technology infrastructure to realize that vision. With respect to SODA, the RSDP addresses five critical needs: bridging the business-to-IT gap, support of the changing roles in the IT organization, a focus on assets and reuse, increasing levels of collaboration within and across practitioner roles, and simplification of product offerings.

Bridging the business-to-IT gap

The service concept is essential in aligning the business view of activities and processes with the technology that is used to realize them. This alignment includes the ability for business models to drive development and to evolve business models and IT solutions in tandem. Services and service-based thinking form the common ground that ties business analysts, IT architects, integrators, and developers together. Common design practices are essential to ensure that concepts, artifacts, and activities are synchronized across these different perspectives. Having tools that can efficiently transform models which represent business intent into efficient implementations is critical in bridging this gap.

Support of changing roles in the IT organization

The move to a service viewpoint changes the skills and composition of teams in an organization. The focus of development is on finding, defining, managing, and assembling services, with architectural descriptions highlighting SLAs and interservice protocols. The traditional breakdown of tool functions into today's line-up of products is not appropriate to this approach; a different blend of capabilities is required by the various members of IT organizations. For example, the skills required for existing roles, such as "software architect," are changing to include greater emphasis on assembly and management of services across a diverse set of service providers. Similarly, new roles, such as "integration specialists," are emerging, whose focus is on assembling a service-based value chain in support of an organization's key business goals.

A focus on assets and reuse

Considering services as key assets in the design of systems changes an organization's view of the value of reusing these services. A service assembly viewpoint leads to the "software factory" paradigm. As a result, technologies and techniques for management and governance of assets and repeatable ways to capture patterns for combining assets become much more important. In an asset-based development approach, these assets hold critical value for the organization and must be carefully managed and administered. The team infrastructure for managing assets takes on a key role in this approach.

Increasing levels of collaboration within and across practitioner roles

Enterprise application developers have always recognized that software development is a "team sport" and have focused attention throughout the life cycle on shared assets management, artifact traceability, and shared practices and processes. The collaborative nature of software development is increasing with greater geographic distribution of organizations, enhanced real-time communication among individuals in teams, and software being embedded as one part of broader systems development initiatives. Increasingly, software development infrastructures are seen as collaborative development environments for software practitioners that encourage sharing and reuse of services among teams.

Simplification of product offerings

An essential element of success in SODA involves reorganizing tool offerings to support the dynamic view of customer needs and to align product offerings with emerging roles in the IT organization. Repackaging tool capabilities greatly improves the delivery of those offerings to customers, supported by promotion of how the tools address today's customer concerns for greater flexibility and value in delivering service-oriented enterprise solutions.

As illustrated in *Figure 2*, the RSDP supports a business-driven development life cycle aimed at bringing together the tasks and roles that collaborate in any enterprise-scale project. They work together to understand, define, build, and deploy collections of services that support the business. The services

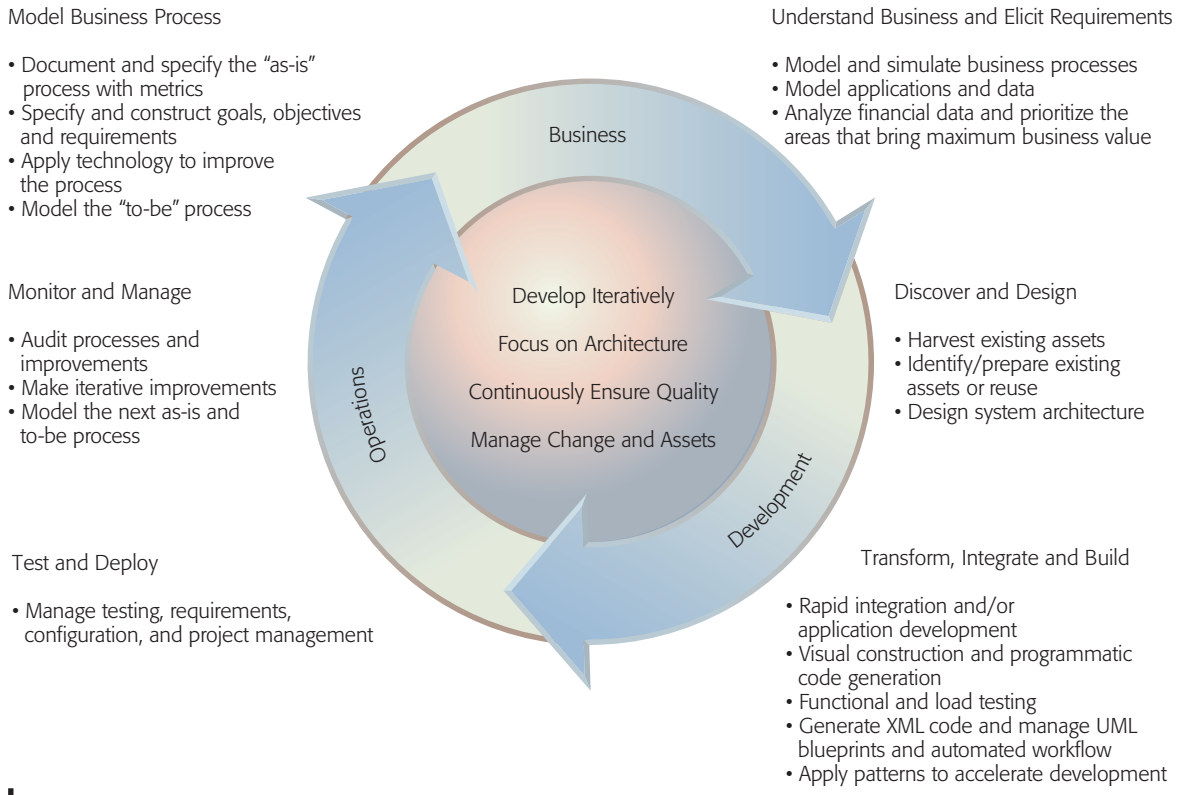


Figure 2
Business-driven life cycle

are also monitored as they are executed, in order to provide feedback that can optimize business operations.

For practical reasons, the creation and delivery of IBM's rich and integrated RSDP has been built upon a common tooling infrastructure, based on a set of shared components. The Eclipse infrastructure (its plug-in architecture, metamodel framework, shared metamodels, and libraries of capabilities) makes this possible.[5] This shared infrastructure enables the use of common components among IBM development teams, so that IBM products can be used together more easily, are open to extension by third parties, have greater consistency and quality, and can be evolved more efficiently.

A service-oriented programming model

A key aspect of the RSDP is the use of a programming model that is influenced strongly by the service-oriented nature of the emerging IBM runtime platforms.[6] The RSDP presents this programming model to practitioners as the means by which to perceive, design, implement, and evolve solutions for the IBM runtime platforms. This programming model is quickly evolving to place services and service-oriented concepts at the heart of how practitioners think about solutions.

A key element of the programming model being driven by the RSDP is a simplified data-access programming model for various resources known as the Service Data Object (SDO), which is in the process of being standardized through the Java Community Process. Practitioners think about persistent information requirements for their solutions in terms of SDOs and how these resources are manipulated by services, not in terms of specific technologies for describing and storing data.

Another key element is the emerging component model for WebSphere* that supports Web-service invocation, wiring, and composition regardless of the type of implementation technology (EJBs** [Enterprise JavaBeans**], stored procedures, RDBMS [relational database management systems], CICS* [Customer Information Control System] transactions, and so on). Based on the J2EE stan-

dard, this work supports modeling, assembly, and runtime interaction among service implementations.

The RSDP uses a service orchestration and component scripting standard that supports workflow and business process integration. BPEL is a maturing industry standard, already widely supported in vendor tools. It also uses a Java framework, Java ServerFaces, that speeds Web application development for developers who are not expert J2EE developers. Applications can be customized by using external policies and rules through a series of emerging standards that are in development for policy definition and enforcement, including WS-Policy.

DESIGN OF SERVICE-ORIENTED SOLUTIONS

In light of the preceding discussion, we can now begin to understand more about what a service is and how services are defined and assembled in an SOA. In essence, a service-oriented approach is a way of designing a software system to provide services either to end-user applications or to 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.

In the following subsections, we explore the idea of services and the design of service-oriented solutions in more detail. In particular, where appropriate, we contrast the design of service-oriented solutions with more widely established approaches for the design of component-based solutions.[7,8]

Service Types

A service is generally implemented as a coarse-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.

Services can take different forms, which are related to the technology used in their implementation. For example, we might be interested in the definition of a specific kind of service, a Web service, as defined by the XML Web Services group in the World Wide Web (W3C**) Consortium:

> A Web service is a software application identified by a universal resource identifier (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.[9]

These two descriptions of services, one focusing on the SOA architectural style and the other on its realization as Web Services, offer a set of characteristics for services related to their nature and applicability. These include:

1. *Granularity*—Operations on services are frequently implemented to encompass more functionality and operate on larger data sets, compared with component-interface design.
2. *Interface-based definition*—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.
3. *Discoverability*—Services need to be found at both design time and runtime, not only by unique identity but also by interface identity and by kind of service.
4. *Single-instance nature*—Unlike component-based development, which instantiates components as needed, each service is a single, always-running instance that a number of clients communicate with.
5. *Loosely coupled nature*—The SOA is a loosely coupled architecture because it strictly separates the interface from the implementation. In addition, runtime discovery further reduces the dependency between service consumers and providers and makes an SOA even more loosely coupled. Services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods, such as XML document exchanges.
6. *Asynchronous nature*—In general, services use an asynchronous message-passing approach; however, this is not required. In fact, many services use synchronous message-passing at times.
7. *Reusability*—Services are assets that can be reused in several contexts, regardless of the component architecture.

Some of these criteria, such as interface-based definition and discoverability, are also used in component-based development. However, the major difference between SOA and component-based

development is the fact that SOA focuses only on the interfaces and their discoverability and emphasizes loose coupling, particularly over network infrastructures. In contrast, component-based development focuses on the component execution environment and the acquisition and deployment of software components in that environment. Collectively, these characteristics differentiate a service-based solution from a component-based solution.

Service design considerations

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. Components and services, although similar, are not the same; they have differing design criteria and design patterns.

Interface-based design

In both component and service development, the design of interfaces is performed such that a software entity implements and exposes a key part of its definition. As a result, the concept of "interface" is essential to successful design in both component-based and service-oriented systems.

An interface definition in languages such as Java or C++, or in languages such as IDL (interface definition language), only provides a set of method signatures. The definition provides the "what," without any guidance on the "how." 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 concept of defining the *behavior* of an interface and, more important, the behavior of sets of related interfaces, is receiving increasing industry attention. Unfortunately, there are currently few standard approaches governing these definitions.

One approach might be to use design models, such as those introduced in this paper, defined in a standardized language such as UML, to document the interdependencies between service interfaces. Such models can be shared, jointly developed, and used to drive specific standards when they emerge. Additionally, IBM has supported the Reusable Asset Specification (RAS) through its standardization within the OMG (Object Management Group, Inc.). RAS provides a mechanism for packaging and sharing assets, and this mechanism could be applied to defining the behavior of sets of interfaces. 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.

Layering application design

The tendency to solve new problems with outdated solutions was encountered as developers began to create component-based systems. They tried to bring their experience with object-oriented development to bear and encountered problems similar to those endemic to that paradigm. With more experience, it was understood that object-oriented technology and languages are excellent ways to implement components, though one has to understand the trade-offs that are inevitable in this design approach. These trade-offs include using inheritance versus aggregation for implementing polymorphic behavior, and the redesign of class libraries to enable the use of components as the basis of a monolithic C++ application.

In a similar way, we believe that components are the best way to implement services, with the caveat that an exemplary component-based application does not necessarily make an exemplary service-oriented application. There is a great opportunity to leverage a company's component developers and existing components, once the role played by services in an application architecture is understood. The key to making this transition is to realize that a service-oriented approach implies an additional application architecture layer. In particular, 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 advantage is "the application edge," reflecting the fact that a service orientation provides an excellent way to expose an external view of a system with internal reuse and composition using traditional component design.

In our experience, 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. It is to be hoped that the move to service-oriented systems can happen more quickly. To this end,

developers 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.

Service-oriented design

Because there is a single instance that manages a set of resources for a service, they are for the most part stateless. 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 are, in fact, simply serialized states. Thus, if we can define the rules for taking a component definition and transforming it into a service, we can implement this serialization as a pattern. The creation and reuse of such patterns is possible with IBM Rational Software Architect.

This passing of state from provider to requestor implies that rather than using a large number of small operations to retrieve the component state, a single large operation is used. Most services are remote, and this approach has certain implications for network usage for remote services, as well as for the behavior of requestors when dealing with large value objects. It 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 that when we retrieve a stock quote or weather forecast, there is the possibility that it is 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 the architecture described here, the requestors must be conditioned to accept variations in copies of state.

Service design and implementation patterns

Our experience with modeling of services and service-oriented solutions using UML 2.0 have led to a number of observations on effective approaches for designing services and service-oriented solutions. Much of our work has been done in the context of creating Web Services solutions. However, many of these lessons support broad design principles that were first highlighted in earlier techniques supporting object-oriented and component-based design. These approaches reinforce ideas such as separation of interface specification from implementation, coupling and cohesion of interfaces, and so on. However, we believe that a number of practices specific to modeling of service-oriented solutions are also readily identifiable.

To make these lessons more concrete, we focus on Web Services. This focus does not change the analysis of the functional requirements for an application; an insurance claim-processing application, for example, must process insurance claims regardless of the technology used in its implementation. Adopting Web Services introduces a set of constraints and potential issues in the area of nonfunctional requirements. In the following, we highlight some of the more interesting Web Services design practices that we have observed in recent modeling projects.

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 (Hypertext Transport Protocol) and SOAP, which are inherently slow and unreliable. To respond to this criticism, "slow and unreliable" must be defined, and it must be realized that even reliable transports rely on unreliable means. When designing enterprise-scale solutions, one must always bear in mind functional and nonfunctional requirements and ensure that the correct trade-offs and decisions are made to support 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. Nevertheless, an alternative to HTTP might provide a better solution in light of the fact that certain services communicate within the same security or application context.

Consider an example consisting of three services: customer management, customer services, and order management. We may design this system such that all external clients interact with the customer management service; however, it interacts with two internal services, customer services and order management. The decision here is, "Why would we require the flexibility of HTTP and SOAP for these internal service communications?" Let us assume that performance is our key requirement for the interaction between customer management and customer services. If so, we might decide to use a

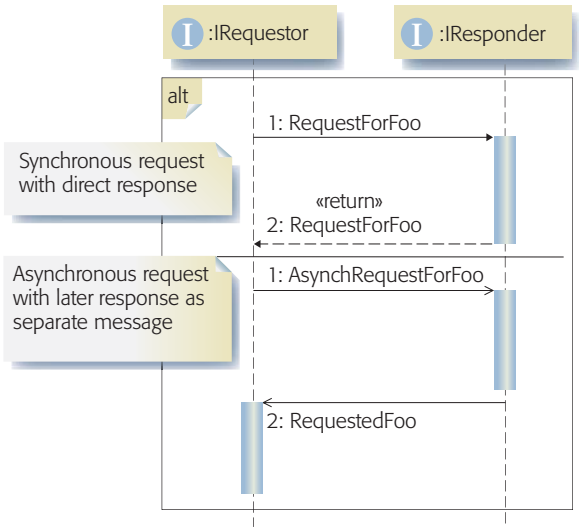


Figure 3
Synchronous and asynchronous behavior

component RPC communication (such as Microsoft .NET** Remoting or Java's RMI over IIOP [Remote Method Invocation over Internet Inter-Orb Protocol]) that provides binary encoding formats and higher performance characteristics. On the other hand, because the key requirement to place an order from customer management to order management is guaranteed delivery, we might use a queuing technology (such as IBM WebSphere MQ or MSMQ [Microsoft Message Queue Server]) to deliver the message, trading performance for a higher level of reliability.

Even though Web Services present a simple model and a set of simple, flexible protocols, one is not restricted to these choices. WSDL has bindings for both SOAP and HTTP `get` and `put` requests, but it is important to provide requestors with additional choices. For example, a single service may expose a message by 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.

Asynchronous behavior and queuing

As mentioned in the introduction to SOA, it is beneficial to make 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 without waiting for the provider to respond. This is not to say that synchronous service behavior is never appropriate; rather, experience has demonstrated that asynchronous service behavior is often preferable, especially where communication costs are high or network latency is unpredictable.

The behavior described in ***Figure 3*** represents a major advance toward implementing highly scalable Web services. By making a service call asynchronous, the provider is enabled to use multiple worker threads to handle multiple client requests. Much more must be done to support an asynchronous mode of operation, aside from returning a response to the client quickly. It is necessary to specify dual interfaces; the requestor will need to pass 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. One may learn about various methods for doing this by looking at the design of Web sessions that are not based on Web Services.

Nevertheless, this solution is scalable only 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 (using Java Message Queue Service [JMS] or message-driven beans for J2EE) is used to decouple a service façade from the service implementation.

Caching in service-oriented design

In the previous section, we introduced the concept 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 three to five characters of data for the security and five to seven characters for the price. This may result in an unacceptable load on the network and service provider. Instead, the contents of the entire portfolio should be requested, either by passing the list of symbols or by passing the portfolio identifier to the

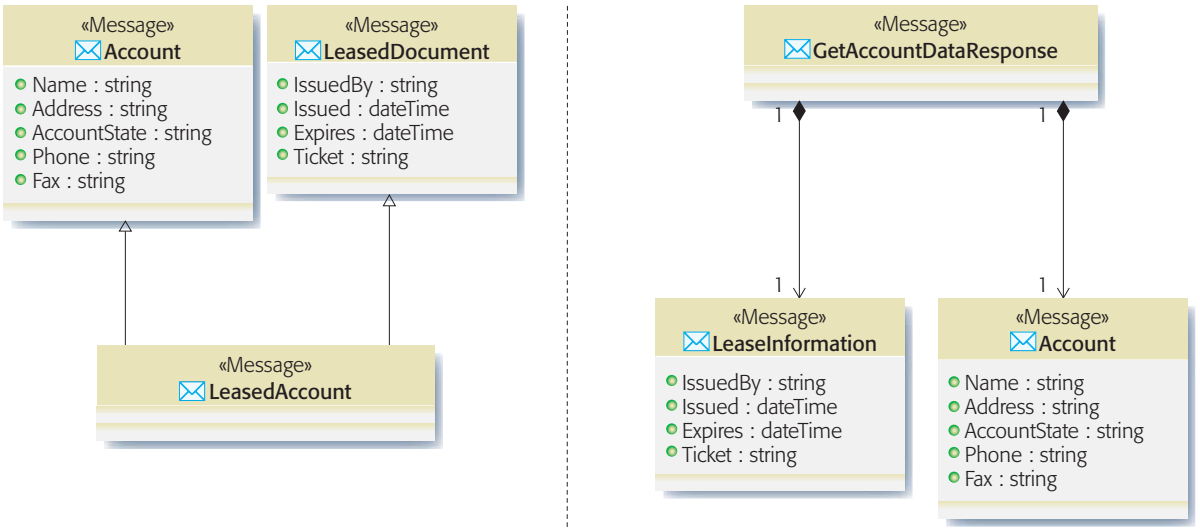


Figure 4
Two implementations of information leasing

service and retrieving all information for each security. If the user simply asks 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 task for the requestor becomes the identification of the "lease" duration of the data. For a portfolio, if it is known that the stock quote service has a 20-minute delay, it may be best to work on a 25 percent margin and cache the results for five minutes.

This pattern is seen frequently 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). It would be desirable for a Web service provider to identify the cache or lease duration as part of its interaction with the requestor. Such issues are well understood in messaging systems such as MSMQ and WebSphere MQ, where message timeouts and expiry times are routinely managed.

Leasing information is viewed in terms of borrowing a book from a library rather 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 a snapshot of state at a given time. This can cause problems, 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.

This is such a fundamental issue that one might expect HTTP, SOAP, or one of the transport protocols would handle it. We could reuse the HTTP caching semantics that allow browsers and firewalls to cache pages, but this is not under the provider's control, and the requestor may not be using HTTP as a transport. One option is to build such support into the document exchange, such that the messages between requestor and provider encode the leasing information for the client, as shown in ***Figure 4***.

Figure 4 illustrates two alternative implementations for the information-leasing paradigm. The first demonstrates the use of inheritance to transform 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. Whereas both of these approaches are equally valid, they result in differently structured data, and the choice is very much one of style, that is, inheritance versus aggregation.

GENERATING SERVICES AND SERVICE-ORIENTED SOLUTIONS FROM MODELS

Modeling services and service designs is important for the purpose of understanding an architectural solution to a given business problem. However, the value of capturing designs in well-defined models and the use of rigorous modeling notation is that they can become the basis for a model-driven approach to implementing an SOA. Generating more concrete models (and code) from abstract models is at the heart of a model-driven approach.

As a fundamental aspect of software engineering, modeling is critical to the success of every enterprise-scale solution; however, there is great variety in what models represent and how models are used. *Model-driven development* refers to a set of approaches in which code is semiautomatically generated from more abstract models, and which employs standard specification languages for describing those models and the transformations between them. It also supports model-to-model transformations.

Models are the stepping stones on the path between a description of business needs and deployable runtime components. As the system under development evolves, the models themselves become more complete, accurate, and consistent with each other. The focus of effort also shifts from the models at the higher level of abstraction to those at lower levels. Ultimately, these models are used to directly create the deployable components.

This approach is equally applicable when the goal is to create services and service-oriented solutions. High-level models representing business concepts can be transformed into logical models of a service-oriented solution, which in turn is transformed into implementations of services and service assemblies that realize the solution. The process of model-driven development can be explored from three perspectives: how models evolve and are related, how transformations are defined and applied, and how automation of these transformations can lead to efficiencies in a software project.[10]

How models evolve

Two main activities occur with models: refinement and transformation. *Model refinement* is the gradual change of a model to better match the desired system. The model is refined as more is known and understood about the system. A model may also be refined for purely internal reasons (i.e., refactoring). As the various models evolve, dependent models need to change in response. By the end of each iteration of the development cycle, however, all the models should be consistent with each other.

Models are refined either manually or through some form of automation or assisted automation. Automation can be in the form of rules for model refinement implemented as executable patterns or assets. When a pattern is applied to a model, it modifies or rearranges the model elements to resemble the pattern. The application of a pattern adds new elements or properties to the model. When a pattern is applied, it may involve some user assistance; for example, prompting the developer for an existing model element with which to bind a pattern parameter, or for other decisions that need to be resolved for the pattern to be executed.

Model transformation, on the other hand, involves two or more models. The most typical example is the transformation of a high-level abstraction model (a platform-independent model [PIM]) into a low-level abstraction model that is technology-dependent (a platform-specific model [PSM]). For example, a UML PIM could represent a logical data model and consist of a number of entity classes, each with a number of persistent attributes. This model could be transformed through automation into a UML data model that captures the same underlying entities, but from the viewpoint of database tables. The data model could in turn be used to directly generate SQL (Structured Query Language) scripts that define the database and could be directly executed on a specific database management system (DBMS).

Model transformations are not necessarily unidirectional; some model transformations can be bidirectional. For example, a platform-specific UML model of several Enterprise JavaBean (EJB) classes could be "synchronized" with the source code implementing these EJBs. New elements (i.e., methods, attributes, and associations) defined in the model would generate appropriate elements in the source, and any new elements created (or removed) in the source would cause appropriate elements in the model to be generated or removed.

Understanding model transformations

Defining and applying model transformations are critical techniques within any model-driven style of

development. Model transformations involve using a model as one of the inputs in the automation process. Possible outputs include another model or varying levels of executable code. In practice, there are three common model transformations, as described in the following:

1. *Refactoring transformations* reorganize a model based on some well-defined criteria. In this case, the output is a revision of the original model and is called the refactored model. An example could be as simple as renaming all the instances where a UML entity name is used, or something more complex, such as replacing a class with a set of classes and relationships in both the metamodel and all diagrams displaying those model elements.
2. *Model-to-model transformations* convert information from one or more models to another model or set of models, typically where the flow of information is across abstraction boundaries. An example would be the conversion of one type of model into another, such as the transformation of a set of entity classes into a matched set of database schema, "plain old Java objects" (POJOs), and XML-formatted mapping descriptor files.
3. *Model-to-code transformations* are familiar to anyone who has used the code generation capability of a UML modeling tool. These transformations convert a model element into a code fragment. This is not limited to object-oriented languages such as Java and C++, nor is it limited to programming languages. Configuration, deployment, data definitions, message schemas, and other kinds of files can also be generated from models expressed in notations such as UML. Model-to-code transformations can be developed for nearly any form of programming language or declarative specification. An example is generating Data Definition Language (DDL) code from a logical data model expressed as a UML class diagram.

Applying model transformations

In practice, there are several ways in which model transformations can be applied. In model-driven approaches, there are four categories of techniques for applying model transformations. In the *manual* approach, the developer examines the input model and manually creates or edits the elements in the transformed model. The developer interprets the information in the model and makes modifications accordingly. Apart from raw speed, the significant difference between manual and automated transformations is that automation is ensured to be consistent and a manual approach is not.

A *prepared profile* is an extension of the UML semantics in which a model type is derived. Applying a profile defines rules by which a model is transformed.

A *pattern* is a particular arrangement of model elements. Patterns can be applied to a model, and this results in the creation of new model elements in the transformed model.

Automatic transformations apply a set of changes to one or more models, based on predefined transformation rules. These rules may be implicit to the tools being used or may have been explicitly defined, based on domain-specific knowledge. This type of transformation requires that the input model be sufficiently complete, both syntactically and semantically, and may require models to be marked with information specific to the transformations being applied.

The use of profiles and patterns usually involves developer input at the time of transformation, or requires the input model to be "marked." A marked model contains extra information not necessarily relevant to the model's viewpoint or level of abstraction. This information is only relevant to the tools or processes that transform the model. For example, a UML analysis model containing entities of type `String` may be marked variable or fixed length, or it may be marked to specify its maximum length. From an analysis viewpoint, the identification of the `String` data type is usually sufficient. However, when transforming an attribute of this type into, for example, a database column type, the additional information is required to complete the definition.

Models and transformations

Transformations such as these can be used to enable efficient development, deployment, and integration of services and service-oriented solutions. Practitioners create models specific to their viewpoint and needs, and these are used as the basis of analysis, consistency checking, integration, and automation of routine tasks. Model-driven approaches allow developers to create services and service-oriented

solutions by focusing on logical design of services and to apply transformations to the underlying SOA technologies. Furthermore, as illustrated in the examples later in this paper, substantial improvements in the quality and productivity of delivered solutions is possible by automating substantial aspects of these transformations to service implementations.

SERVICES, SOA, AND THE RATIONAL UNIFIED PROCESS

As experience in developing services and service-oriented solutions increases, a growing consensus concerning best practices for designing an SOA is emerging. It is essential that these emerging practices augment and support existing software engineering methods, rather than serve as a separate thread of development experience. In this way, an SOA can be seen as a natural evolution of established approaches, and a channel for introducing service-oriented techniques is made available. In this section we discuss how SOA design fits into the broader context of software-engineering processes.

The Rational Unified Process (RUP*) is the de facto standard software engineering process in use today.[11] It provides a disciplined approach to assigning tasks and responsibilities within a development organization and has been applied to projects of varying size and complexity, with small teams and large, on small efforts lasting a few weeks to large-scale programs lasting years. The goal of the RUP is to ensure the production of high-quality software that predictably meets the needs of its end users on schedule and within budget.

Not surprisingly, the RUP has most recently been applied to projects aimed at creating services and SOA solutions. In these projects, we have found that many of the core principles of the RUP remain essential to the success of such projects. However, we also have encountered areas where updates and additions to the RUP are valuable in support of service-oriented approaches.

The Rational Unified Process

The RUP is a software development process that has, as its foundation, a set of best practices that represent commercially proven approaches to software development. When used in combination, these practices ensure the success of a software development project by striking at the root causes of typical software development problems. The RUP was explicitly designed to support the implementation of six best practices:

1. *Develop iteratively*—The functionality of the system should be delivered in a successive series of releases of increasing completeness. Each release is termed an iteration. The selection of which requirements are developed within each iteration is driven by the desire to mitigate project risks, with the most critical risks being addressed first.
2. *Manage requirements*—A systematic approach should be used to elicit and document the system requirements and then manage changes to those requirements, including assessing the impact of those changes on the rest of the system. Effective management of requirements involves maintaining a clear statement of the requirements, as well as maintaining traceability from these requirements to the other project work products.
3. *Use component architectures*—The software architecture should be designed using components. A component-based development approach to architecture tends to reduce the complexity of the solution and results in an architecture that is more robust and resilient and enables more effective reuse.
4. *Model visually*—A set of visual models of the system should be produced, each of which emphasizes specific details and ignores others. These models promote a better understanding of the system that is being developed and provide a mechanism for unambiguous communication among team members. "A picture is worth a thousand words."
5. *Continuously verify quality*—The quality of the system should be continuously assessed with respect to its functional and nonfunctional requirements. Testing should be performed as part of every iteration. It is much less expensive to correct defects found early in the software development life cycle than to fix defects found later.
6. *Manage change*—A disciplined and controlled approach for managing change should be established (such as changing requirements, technology, resources, products, platforms). The way in which changes are introduced into the project work products should be controlled: who introduces the changes and when those changes are

introduced. A means should be provided to efficiently synchronize those changes across the different development teams, releases, products, platforms, and so forth.

These best practices are the result of IBM Rational's experience in developing its software products, together with the experience of IBM Rational's many customers. Implementing these best practices puts a software development organization in a much better position to deliver quality software in a repeatable and predictable fashion.

The RUP can be described in terms of two dimensions: time and content. ***Figure 5*** provides a graphical representation of these dimensions. The horizontal axis represents time and shows the life-cycle aspects of the process. This dimension is described in terms of phases and iterations. The vertical axis represents content and shows the disciplines that logically group the process content.

As the maxima in Figure 5 illustrate, the relative importance of the disciplines changes over the life of the project. For example, in early iterations more time is spent on requirements; in later iterations more time is spent on implementation. Configuration and change management, environment, and project management activities are performed throughout the project. It is important to note that all disciplines are considered within every iteration.

The RUP as a process framework

Although the RUP is often viewed as a process that is used "as is," it is best thought of as a process framework that is intended to be customized. Organizations that adopt the RUP typically remove process elements that are not relevant to them and add their own best practices, extend existing practices, and introduce the organization's specific nomenclature, standards, and concepts, as appropriate. The result is known as an "RUP configuration."

In particular, IBM Rational provides a number of process plug-ins that allow an organization to import process content not provided with the RUP. For example, plug-ins are available for extreme programming, IBM WebSphere Application Server, J2EE, .NET, and Web design. As a result, the initial phases of a project using the RUP would typically involve the creation of a particular customization of the RUP as the basis for the project. This would involve customizing the RUP content, extending the content with organization-specific information, and selecting the appropriate RUP plug-ins.

The RUP and enterprise-wide initiatives

The RUP has also been applied beyond the execution of a single project. For example, the RUP has been applied to systems engineering (developing systems that comprise hardware and people, as well as software), enterprise architecture (using frameworks such as the Department of Defense Architecture Framework [DoDAF]), and asset-based development (focused on strategic reuse programs).

One of the themes of such enterprise-wide initiatives is the application of an architectural pattern known as the "system of interconnected systems."[12,13,14] This pattern helps control the complexity inherent in a system of systems. One of these systems represents overall capability and is referred to in the pattern as the *superordinate* system. The other systems represent a part of this overall capability, and each is referred to as a *subordinate* system. These systems are shown in ***Figure 6***. The RUP is then used as a process framework that supports the development of the superordinate system and each subordinate system. An important characteristic of the "system of interconnected systems" pattern is that it is recursive, meaning that a subordinate system may also have subsystems of its own and be superordinate in relation to them.

The RUP and SOA

The nature of the RUP makes it well suited to building projects and assembling services. The RUP is founded upon software-engineering best practices, offers a configurable process framework, and is scalable to support enterprise initiatives. Hence, it is a viable choice when considering the development of an SOA because all of these aspects of the RUP apply. This subsection gives some specific examples of where the RUP provides support for an SOA initiative.

First and foremost, the RUP can be applied to support the development of an SOA (a superordinate system) and each individual service (subordinate systems in the context of the SOA). However, there are differences (and many similarities) between the development of an SOA and the development of a service. For example, in developing an SOA, a particular concern is the identification of services and an understanding of how business processes are realized through the execution of these services. The RUP provides a systematic approach for bridging

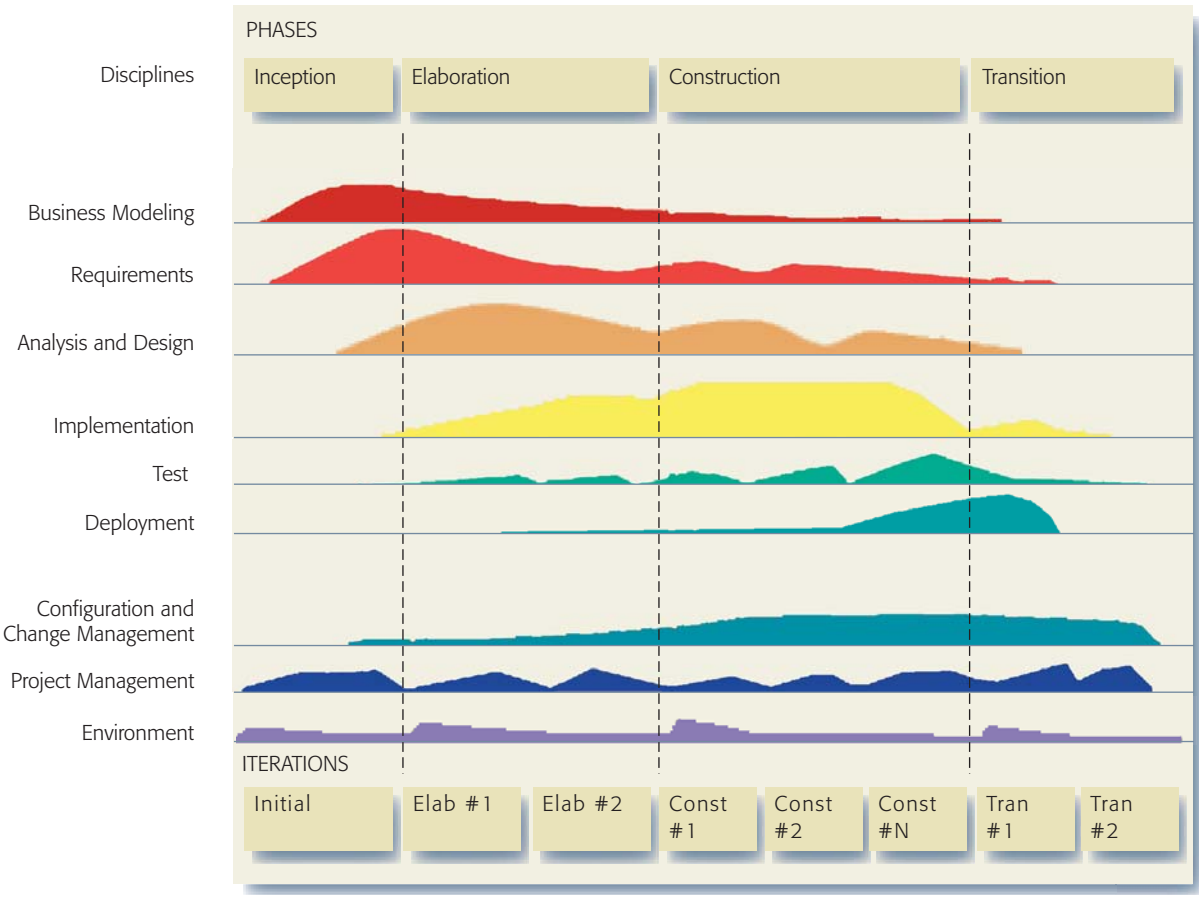


Figure 5
Time versus content in the Rational Unified Process

this gap between business and IT. It contains activities for identifying architectural elements (such as services) known as "architectural analysis" and design-element identification, as well as activities for understanding how these architectural elements collaborate in order to satisfy business requirements (business processes) known as "use case analysis" and "use case design." The RUP also acknowledges any existing design elements that might exist (including legacy systems and packaged applications), and therefore takes both a "top down" and "bottom up" approach to developing an SOA.

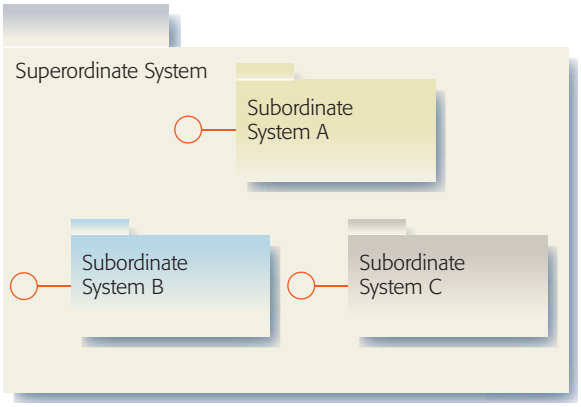


Figure 6
System-of-interconnected-systems pattern

This approach to developing an SOA not only identifies the services and their provided and required interfaces (and associated qualities), but also their relationships and responsibilities. An example of a subset of an SOA that shows structural elements in terms of services, interfaces, and their relationships, is given in ***Figure 7***. This example of a UML component diagram, created by using Rational Software Architect with the UML profile for software services,[15] is from an order-processing system and

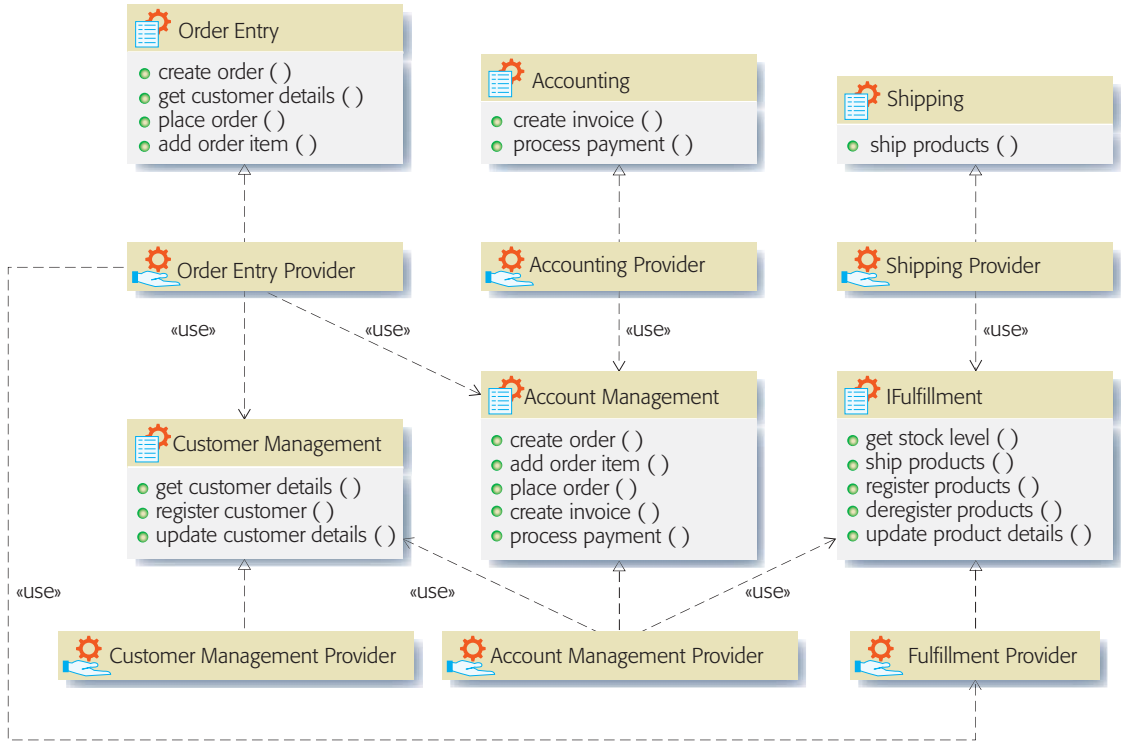


Figure 7
UML representation of an SOA

uses a UML 2.0 component to represent a service. Of course, the behavior associated with the SOA, which, for example, may involve the use of UML sequence diagrams, would also be created as required.

The RUP can also be used in the development of an individual service. In this case, the interfaces that a service provides, the interfaces that it requires, and the qualities that it provides represent the requirements of the service. In this respect, the RUP can be used to provide guidance on how such requirements can be fulfilled by a solution.

There is much more to the RUP than discussed here, including many books on the subject, which provide detailed discussions of the concepts within the RUP that can be applied to the development of an SOA and the services that comprise an SOA.[11,16]

Enhancements to the RUP for SOA

Although the RUP as defined today can be successfully applied to the development of an SOA, there is general agreement that more is needed. The RUP provides a process framework that is, for the most part, technology independent. The base RUP product does not, for example, include any mention of J2EE, .NET, SOA, Web services, and so on. An example of the kind of SOA-specific practices that are useful can be seen in the RUP plug-in for SOA that was recently delivered as an update to the RUP on IBM developerWorks.[15] It contains specific design guidance, heuristics, and tool usage tips for SOA design, based on recent industry experiences developing service-based solutions.

In practice, therefore, an organization applying the RUP to an SOA project would create a specific RUP configuration specialized for the task. The IBM RUP plug-in for SOA defines or refines various process concepts, such as roles, artifacts, and activities, and makes use of relevant standards. In particular, the plug-in defines SOA-related artifacts, SOA-related activities, and SOA-related standards.

Examples of SOA-related artifacts include the "service model," defining a set of services managed as a set of logical service partitions. SOA-related activities might include "harvest services" and "locate service." In defining SOA-related standards, the plugin offers detailed guidance to validate that

defined services comply with the various WS-* standards. In addition, specific tool mentors are provided for IBM Rational Software Architect to illustrate how service models can be developed that comply with the UML profile for software services.

EXAMPLES OF SERVICE-ORIENTED SOLUTIONS

A number of practical lessons in building service-oriented solutions have been gained from using the ideas presented in this paper in specific customer situations. We next outline two example scenarios drawn from real IBM commercial products.

The first example focuses on business-driven development of solutions and the automation of the transformation between the logical design of a service and its realization in a specific set of Web Services technologies.

The second example looks at how a service approach can be applied to an industry domain model to bring together the business and IT views and lead to a high-quality service-oriented implementation. We highlight the role that industry domain models play in guiding the definition and realization of services and service-oriented solutions.

Using a service approach to connect business and IT

One of the primary challenges to be addressed in developing enterprise-scale solutions is to connect the domain-specific requirements expressed by business analysts with the technology-specific solutions designed by the IT organization. Typically, the connection between these two communities is difficult to make because they have very different skills, use different modeling concepts and notations (if at all), and rarely understand the mapping between those concepts. The use of a service-oriented approach is intended to help bridge this gap between the business analysts and line-of-business (LOB) specialists, and the IT specialists (such as system architects, analysts, integrators, designers, and developers). In particular, the integration of process, assets, and deliverables around a core set of services is aimed at connecting these two different aspects of the system in a precise, unambiguous way.

In this example, we consider the vehicle reservation process that is in use in a car rental agency. A project is underway to look at improvements to the vehicle reservation process and the systems that support it. In this example, there are four key steps: (1) modeling the vehicle reservation process, (2) designing the solution to the assign-vehicle task, (3) implementing the assign-vehicle service, and (4) integrating the assign-vehicle service in a choreographed business process.

Modeling the vehicle reservation process

In Step 1, the vehicle reservation process is examined in detail. Each of the key business tasks is described; workflow (manual and automated) among these tasks is defined; and the people, roles, and organizational hierarchies are described.

As shown in *Figure 8*, the process begins by modeling the business process in an intuitive, easy-to-use notation that is accessible to business analysts. A business process model captures the key business services by using the IBM WebSphere Business Integration (WBI) Modeler. This allows the current system of automated and manual steps to be understood and potential changes to the system to be designed and simulated and their costs assessed before the organization commits to any changes to the business process. Various configurations of resources and costs can be examined to optimize revenue from the redesigned process.

Designing the solution to the assign-vehicle task

In Step 2, parts of the vehicle reservation process are identified as candidates for automation and handed off to the IT organization for further elaboration. Decisions can be made about which new business services should be automated in software, and specific services designed and implemented to realize them. This may involve reconfiguring existing implemented services, wrapping existing data and business logic to expose their functionality as services, creating service interfaces to third-party commercial software packages, or designing new services from scratch.

These business service descriptions can be automatically transformed into an initial set of use cases for the proposed system that defines the requirements of the system. In this example, there is a direct correspondence between the assign-vehicle task and the creation of a new service to support the execution of this task. This service is provided by a rental software component. It automates the task of finding the particular vehicle from those available

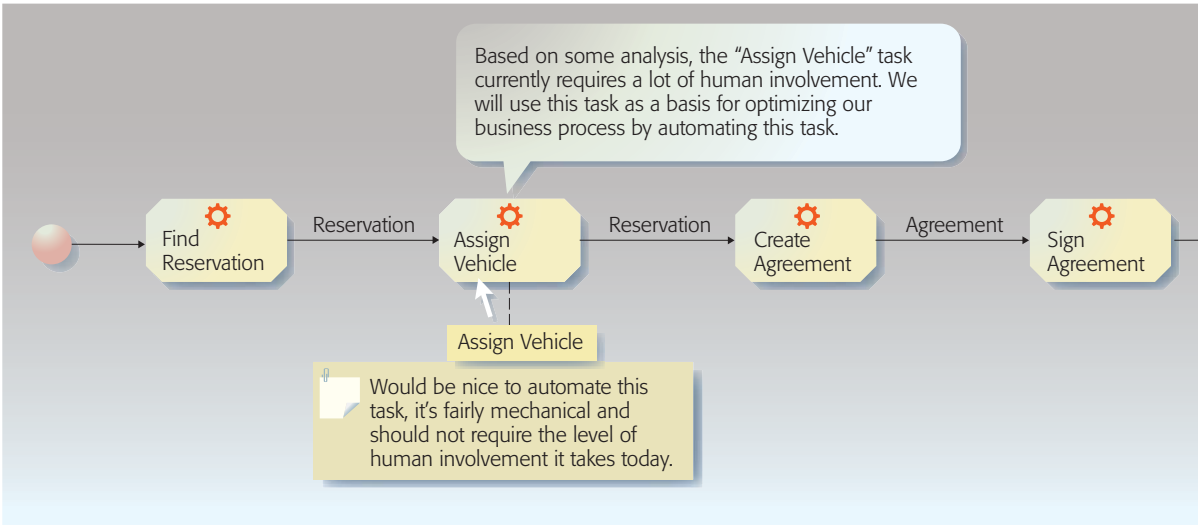


Figure 8
Segment of process model for vehicle reservation system

and assigning that vehicle to a specific customer, such that the vehicle reservation has already been made when the customer arrives at the rental desk with his or her reservation. Using IBM Rational Software Architect, an architect in the IT organization can import the WBI Modeler project and, applying a model-driven development approach, automatically transform the vehicle reservation process model into a UML model, as seen in the tree browser shown on the lefthand side of ***Figure 9***. This transformation ensures that a familiar representation in UML is made available to the IT organization. In the transformed model we find a rental software-component interface with one assigned operation, assign vehicle. This is the interface that will be implemented and made available as a Web service.

Now that we have analyzed the business model, we create a new design model to contain the design for the assign-vehicle service. We add the rental software component to our design model to provide traceability from the design model back to the original business model, and create the new interface we will design and implement. In this example, we model an `IVehicleAssignment` interface with an `AssignVehicle` operation that takes a reservation as input and returns the updated reservation as output. At this point, we have the design for our assign-vehicle service. We can now implement this interface. We start by transforming this design model into the interface definition in Java. The design model and the action to transform the interface into Java are shown in Figure 9.

Implementing the assign-vehicle service

In Step 3, the `IVehicleAssignment` Java interface has been generated. This interface represents a contract between the architect who designed the interface and a developer who will implement the interface. The developer implements this interface as a service and registers the availability of the service in a UDDI registry. Again, by using an automated transformation, the Java implementation class can be generated and all the appropriate business logic implemented as part of this service, employing many of the Web-service patterns and guidance points discussed earlier. From this Java class, a Web service is generated. A series of dialogs is used to capture the data about the Web service, including deployment details. The Web-service generation creates the WSDL description of the service that can now be stored in the UDDI registry to make it available for use by other service consumers.

Integrating the assign-vehicle service in a choreographed business practice

In Step 4, the services are choreographed as part of a business process workflow. This is the role of the integration specialist, bringing together the overall workflow based on the business process model defined earlier with the service implementations that automate key business process tasks. This

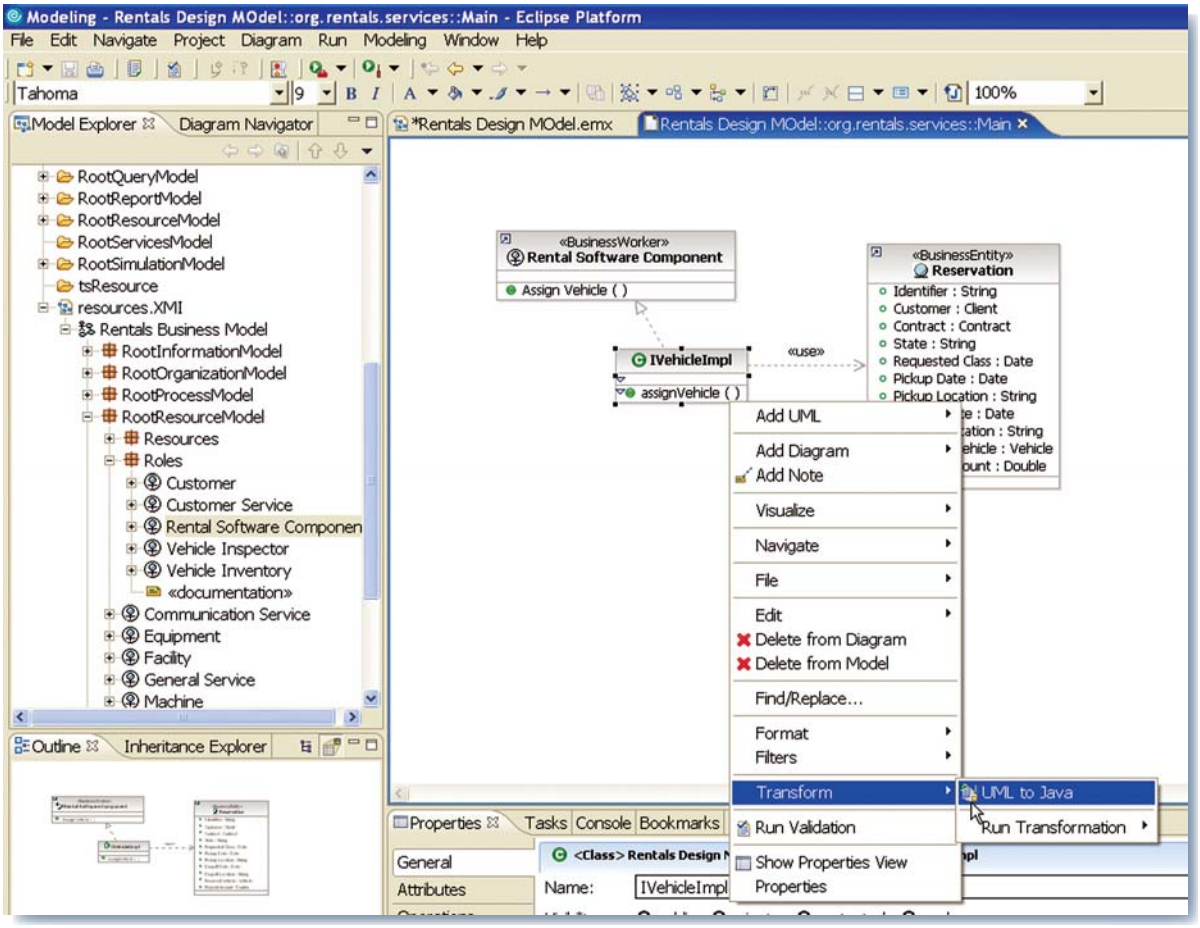


Figure 9
Design model for the assign vehicle interface

business process is exported from the IBM WBI Modeler model in the form of a BPEL script. The IBM WebSphere Studio Application Developer Integration Edition can import, create, enact, and manage business processes described in BPEL. The BPEL implementation of the vehicle-rental-pickup process is shown in *Figure 10*.

One way to implement these business processes is to link to an implementation provided by a Web service. Hence, the vehicle assignment service is found in the UDDI registry and is included as a service provider to the overall car-rental business workflow. At this point, the business process can be deployed to a process execution engine, in this case the IBM WBI Server Foundation runtime.

This example covers only a subset of the RSDP. For instance, although not discussed here, it is possible to expose CICS transactions as Web services by using IBM's WebSphere Enterprise Developer, and these, too, can then participate within an SOA. As one would expect, it is possible, in the vast majority of cases, for all IBM-supported technology to participate in some way in an SOA initiative.

In summary, this example has illustrated how services can be defined and constructed by connecting domain-specific business services into a technology-specific solution for deployment to an SOA infrastructure, following a repeatable, predictable process.

Deploying business-driven SOA solutions with industry models

Industry models, such as IBM's Insurance Application Architecture (IAA) or IBM's Information Framework (IFW) for the banking industry, are a set of business and IT-level technology-independent models, rich in business content, that can be

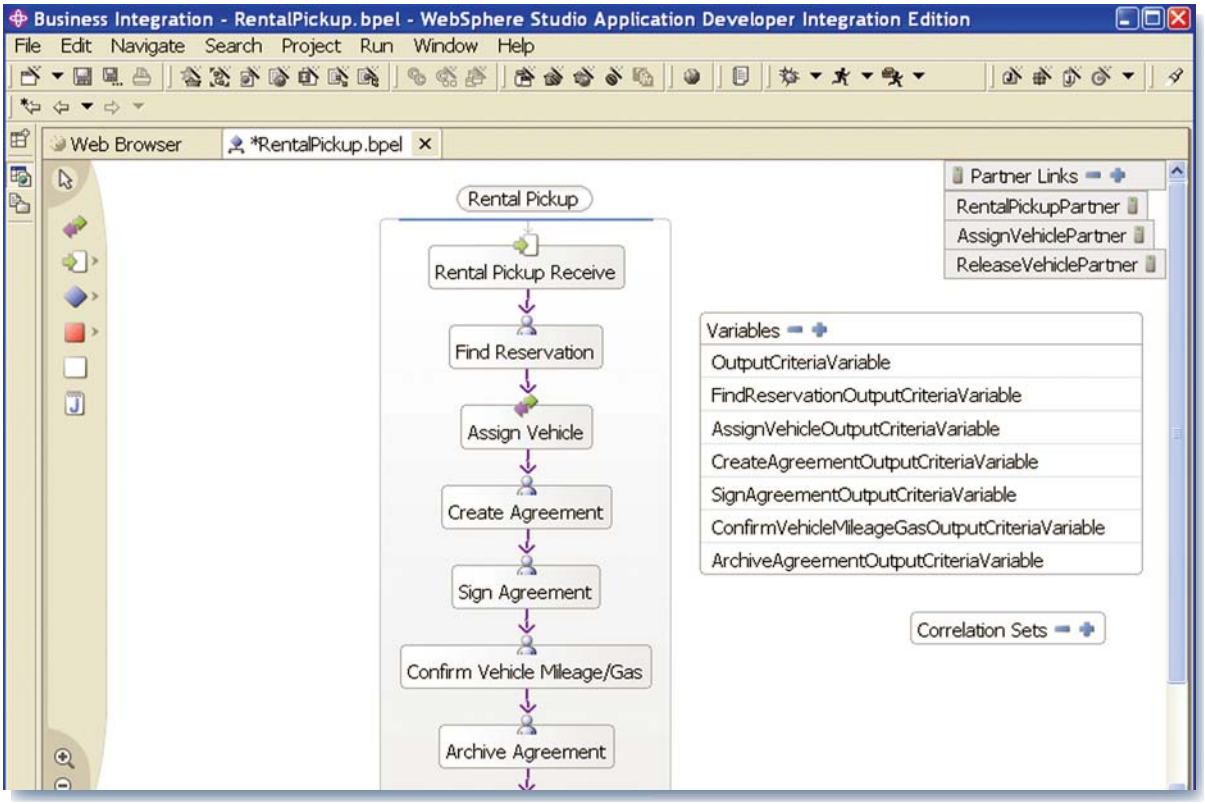


Figure 10
BPEL representation of the process for vehicle rental pickup

customized to reflect specific companies' needs. They capture industry best practices and offer business content for use in developing complex business applications on top of the RSDP. They apply all the principles of separation of concerns, and as such, are particularly well-geared to deploying business-driven SOA solutions.

These industry models contain a variety of artifacts. From an SOA context, the following artifacts are particularly relevant and can be customized instead of having to be created from scratch: business process models (analysis and design level), service definitions, and service choreography. Both of the latter are defined within a UML model called the interface design model. This model is a PIM, which focuses on the separation of the interface from the implementation and can be transformed easily into PSMs (XML for messaging, J2EE for component-based development, or WSDL for SOA, for example).

In order to illustrate these principles, we consider an example in the area of claim notification in a typical insurance organization. The steps followed in using the IAA model in this context to develop an SOA solution are illustrated in *Figure 11*.

Analyzing the business process

Analyzing the business process is typically an activity performed by business analysts in conjunction with subject matter experts. The goal is to obtain a representation of how the business is (or should be) run. At this stage, the formalism is typically not rigorous enough to make it possible to deploy a business process through service choreography in the runtime.

This task is the first step in any business-driven SOA project. The advantage here is that it is possible to customize predefined industry processes rather than create them. This accelerates the analysis, and more importantly, ensures that the processes are defined consistently throughout the enterprise. This latter point is particularly relevant in the context of SOA, as it drives the analysis from the very beginning in the direction of reusable enterprise services, a key goal of any SOA solution.

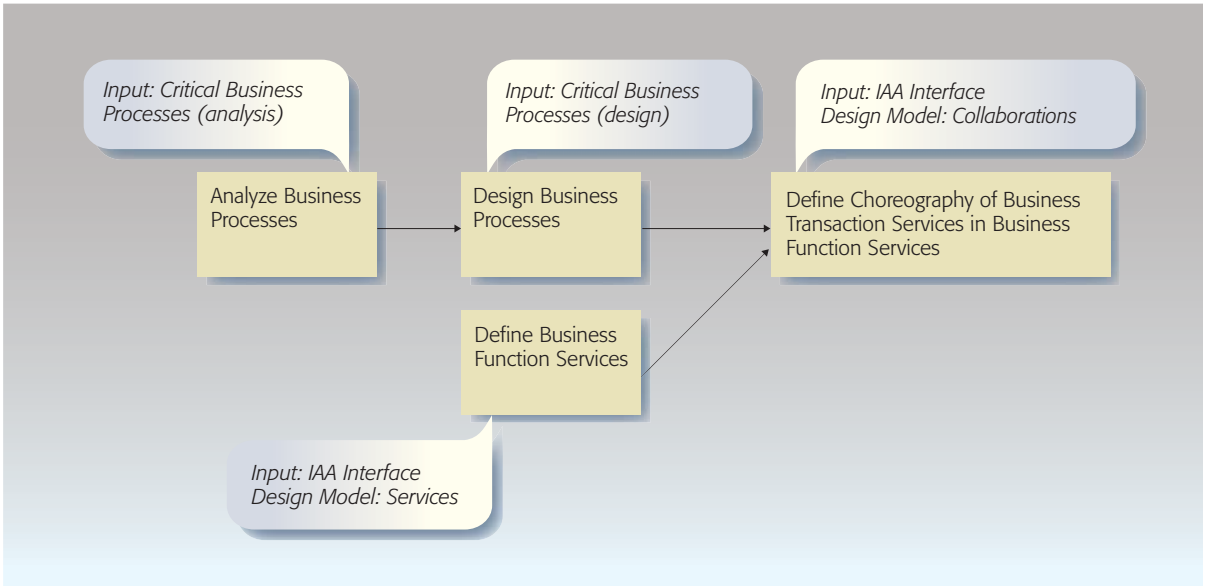


Figure 11
Using the IAA interface to create a service-oriented solution

To illustrate the kinds of business processes that are defined in the IAA, ***Figure 12*** shows a small subset from the IAA of a business process in the IBM WBI Modeler describing a small number of steps in the claims process.

Designing business processes

The primary path to create an executable business workflow is to use BPEL generation, which is available in the WBI Modeler. However, before doing so, it is essential to properly design the processes for that purpose. In particular, three tasks must be carried out to refactor the business process model in order to facilitate transformation from the WBI Modeler business process models into executable BPEL. The data containers must be formalized (transforming multiple data inputs and outputs into structured data containers), loops must be resolved,

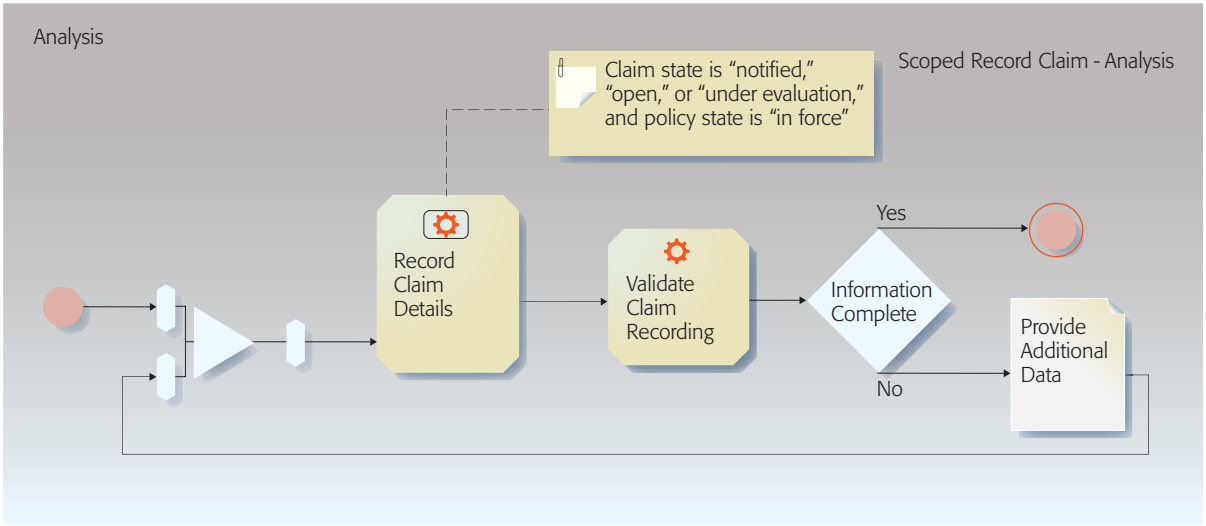


Figure 12
A sample business process in IAA

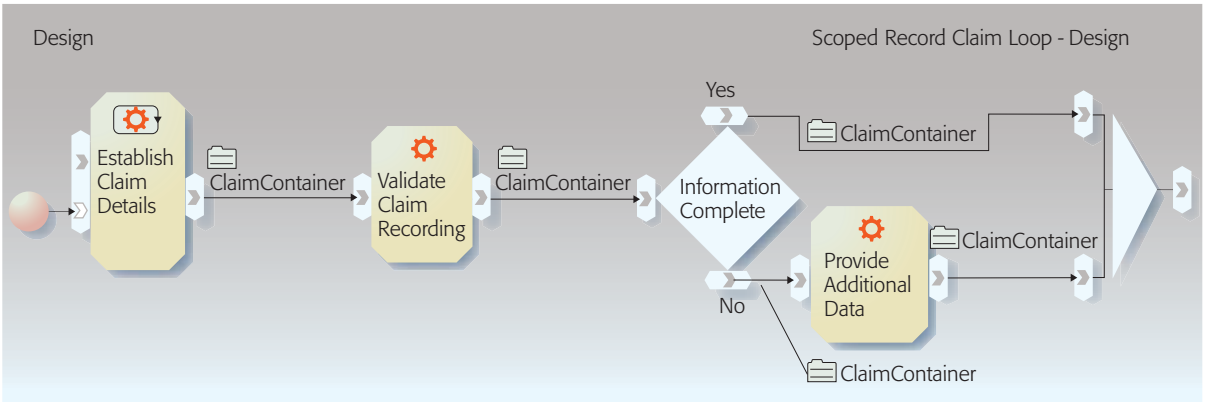


Figure 13
A refactored business process

and decisions must be made on the combination of tasks.

The definition of proper repeating groups is typically very hard to grasp for business analysts who tend to define backward connectors in the process flows. In order to generate valid BPEL, these loops need to be resolved and properly designed. Technology is currently being developed by the IBM Zurich Research laboratory to automate the resolution of most of the loops.

Because of the different realities of business and IT, it is very common to have different groupings of functionality in business and IT terms. The functionality grouping of the design-level business processes must reflect the IT view. *Figure 13* represents a subset of the design-level process after performing the refactoring of the business process.

Following refactoring, the BPEL-generation capabilities of the WBI Modeler can be used to create the implementation of the business process services. A standardized mapping is applied in which the automated activities (tasks) in the process flow correspond to business transaction services. The nonautomated activities are represented as staff activities in the WBI Modeler and typically are documented as manual procedures. Consequently, as part of this step, three elements are defined: the business process services, the business transaction services, and the choreography between the business process services and the business transaction services. *Figure 14* shows the BPEL representation generated from the design level process representation in the WBI Modeler.

Defining business function services

The definition of business function services is the intersection where the top-down and bottom-up approaches meet. In essence, this level provides means to automate the business transaction services. In the IAA approach, the business function services are defined within the interface design model as interfaces with operations. The interface design model has been built over time by combining two different methods: top-down use-case-driven modeling and bottom-up legacy functionality wrapping. The result is a set of enterprise-wide services that satisfy the industry business requirements.

Defining the choreography of business transaction services into business function services

From a platform-independent viewpoint, the collaboration between higher- and lower-level services is key to addressing how lower-level services collaborate to implement higher-level services. This can be expressed by using UML collaboration diagrams (or sequence diagrams).

A variety of technological choices are possible for implementing the business transaction services. For example, BPEL can be used, and the function services can be defined as collaborating services. Alternatively, the collaboration can be implemented as Java code. As often experienced, the trade-off is between performance and maintainability.

In the current state of technology, it seems more reasonable to implement lower-level service collaborations using component technology, although it

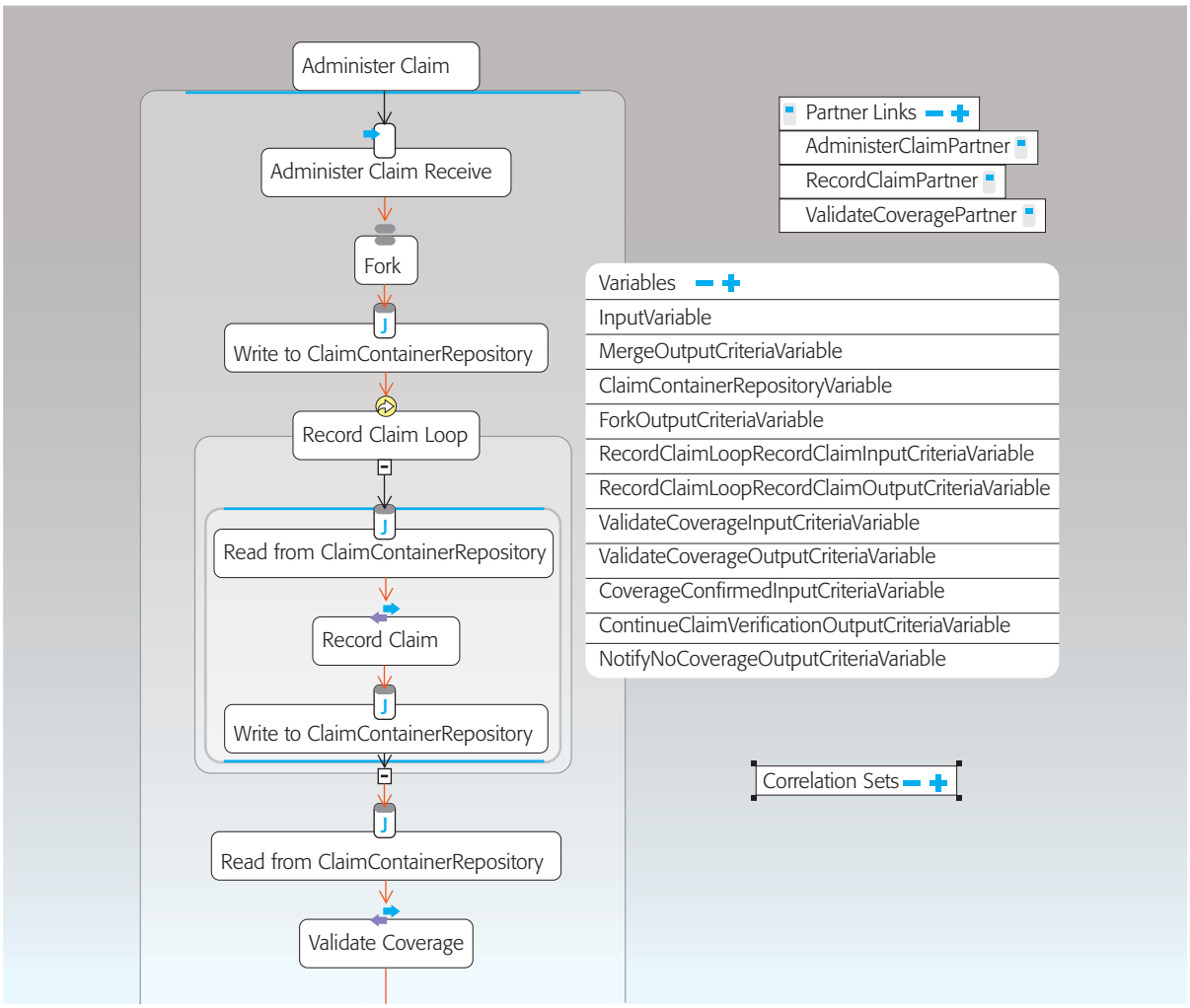


Figure 14
BPEL generated from the business process model

seems reasonable to assume that in the near future it will be possible to "externalize" the lower-level services as collaborations as well. From a platform-independent point of view, the approaches have a similar outcome: a UML collaboration diagram that is readily converted into a BPEL flow or that can be considered as Java pseudo-code. As long as the PIM focuses on interface design and collaboration, insulation from the technology is possible, and the business logic of SOA solutions can be efficiently maintained.

SUMMARY

Flexibility is essential as today's organizations seek to react more quickly to the changing demands of their customers, announcements by competitors, and the evolving business environment. The role of software in many businesses is now seen as central to their ability to compete effectively and efficiently. Having a service orientation to the systems being developed helps to focus businesses on what is essential to them—the services they offer to customers. It also helps IT professionals to look at the systems that support the business in a different way—as composable solution fragments that must be assembled to meet evolving business needs. This view is an important cornerstone of today's highly reactive business environment.

IBM's experience in helping organizations move toward adoption of services and an SOA approach reinforce the lesson that a change in culture and practices goes hand-in-hand with supporting changes in technologies and techniques. There is a

great deal to be said about how service-oriented approaches change an organization's culture. Four aspects of particular relevance to SOA are (1) an enterprise-wide approach and governance, (2) the model-driven approach and architecture, (3) a business-led approach and transformations, and (4) an interface orientation to design.

In this paper, we have focused on the importance of designing for and with services to create quality service-oriented solutions that meet the needs of organizations for flexible and agile enterprise IT systems. Many of the design principles, practices, and tools for service-oriented design are only now beginning to emerge. We have provided a view of these best practices together with practical insights into how service-oriented thinking is having an impact on enterprise software development today.

Creating these service-oriented solutions is far from straightforward. The RSDP plays an important role in helping organizations create a set of services capable of realizing their goals. It combines market-leading products to create a rich, integrated environment for solution development. Through support for model-driven development techniques, the RSDP helps to ensure that customers can efficiently deliver service-oriented solutions that meet their business needs.

CITED REFERENCES AND NOTES

1. M. E. Stevens and H. J. C. Ellis, "Using a Lease to Manage Service Contracts in Service Oriented Architectures," *Proceedings of the Tenth Americas Conference on Information Systems (AMCIS)*, New York (August 2004), http://aisel.isworld.org/proceedings/amcis/2004/track.asp?track_id=243.
2. D. Plummer, *SODA Helps Developers Do Application Integration*, Gartner Research Report (November 2002), http://www.g2r.com/DisplayDocument?doc_cd=111182.
3. For further details and examples, see IBM's developerWorks (http://www.ibm.com/developerworks), IBM Redbooks (http://www.ibm.com/redbooks) and the Rational Unified Process page of IBM developerWorks (http://www-130.ibm.com/developerworks/rational/products/rup/).
4. N. Carr, *Does IT Matter?* Harvard Business School Press, Cambridge, MA (2004).
5. S. Holzner, *Eclipse*, O'Reilly & Associates, Sebastopol, CA (2004).
6. D. Ferguson and M. Stockton, *SOA Programming Model for Implementing Web Services, Part 1: Introduction to the IBM SOA Programming Model*, IBM developerWorks (June 2005), http://www.ibm.com/developerworks/webservices/library/ws-soa-progmodel/index.html.
7. J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, Reading, MA (2000).
8. P. Herzum and O. Sims, *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*, Wiley Press, Hoboken, NJ (2000).
9. As defined by the W3C Web Services Architecture Group. See *Web Services Architecture Requirements*, http://www.w3.org/TR/2002/WD-wsa-reqs-20020429.
10. A. W. Brown, J. Conallen, and D. Tropeano, "Practical Lessons in MDA," Chapter 5 in *Model-Driven Software Development*, S. Beydeda, M. Book, and V. Gruhn, Editors, Springer-Verlag, 2005.
11. P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, Reading, MA (1998).
12. I. Jacobsen, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success*, Addison Wesley, Reading, MA (1997).
13. M. Ericsson, *Developing Large-Scale Systems with the Rational Unified Process*, Rational Software White Paper (2003), ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/sis.pdf.
14. P. Eeles and M. Ericsson, "Modeling for Enterprise Initiatives with the Rational Unified Process," *The Rational Edge* (January 20, 2004).
15. S. K. Johnston, *UML 2.0 Profile for Software Services*, *IBM developerWorks* (2005), http://www.ibm.com/developerworks/rational/library/05/419_soa/.
16. P. Kroll and P. Kruchten, *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Addison-Wesley, Reading, MA (2004).

GENERAL REFERENCES

K. Ahmed, *Developing Enterprise Java Applications with J2EE and UML*, Addison Wesley, Reading, MA (2001).

D. K. Barry, *Web Services and Service Oriented Architectures*, Morgan Kaufman, San Francisco, CA (2003).

G. Booch, I. Jacobsen, and J. Rumbaugh, *The Unified Modeling Language Users Guide*, Addison-Wesley Professional, Reading, MA (1998).

A. W. Brown, "Model Driven Architecture: Concepts and Practice," *Journal of System and Software Modeling* **3**, No. 4 pp. 314–327, Springer Verlag (December 2004).

A. W. Brown, *IBM Rational Software Development Platform*, IBM Corporation (2004), http://www.ibm.com/developerworks/platform/.

K. Brown, G. Craig, G. Hester, R. Stinehour, W. D. Pitt, M. Weitzel, J. Amsden, P. M. Jakab, and D. Berg, *Enterprise Java Programming with IBM WebSphere*, Addison-Wesley Professional, Reading MA (2003).

P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional, Reading, MA (2001).

D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Press, Hoboken, NJ (2003).

IBM Patterns for e-business (2004), http://www.ibm.com/developerworks/patterns.

A. T. Manes, *Web Services: A Manager's Guide*, Addison-Wesley Information Technology Series, Reading, MA (2003).

B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, **20**, No. 5, 19–25 (September 2003).

MDA Guide v1.0.1, J. Miller and J. Mukerji, Editors, Object Management Group (June 2003), http://www.omg.org/docs/omg/03–06–01.pdf.

U. Wahli, M. Tomlinson, O. Zimmerman, W. Deruyck, and D. Hendriks, *Web Services Wizardry with IBM WebSphere Studio Application Developer*, IBM Redbook, SG24-6292-00 (April 2002).

O. Zimmermann, P. Krogdahl and C. Gee, *Elements of Service-Oriented Analysis and Design*, IBM developerWorks (June 2004), http://www-106.ibm.com/developerworks/webservices/library/ws-soad1/.

Accepted for publication June 19, 2005.
Published online October 20, 2005.

Alan W. Brown
IBM Rational Software, 4205 S. Miami Blvd., Durham, NC 27709 (awbrown@us.ibm.com). Dr. Brown is an IBM Distinguished Engineer with the IBM Rational software group. He is responsible for aspects of future product strategy in IBM Rational's design and construction products. He defines technical strategy and evangelizes product direction with customers looking to improve software development efficiency through visual modeling, service-oriented design, generating code from abstract models, and systematic reuse. His current focus is on how service-oriented solutions are created and evolved, with particular interest in software process improvement, model-driven architecture, software design and development, and component-based reuse. He received his Ph.D. degree from the University of Newcastle in the United Kingdom.

Marc Delbaere
IBM Software Group, Industry Solutions, Avenue du Bourget 42, Brussels, Belgium 1130 (delbaere@be.ibm.com). Mr. Delbaere is the development manager for IBM's insurance industry models: IAA (Insurance Application Architecture) and IIW (Insurance Information Warehouse). He has worked for eight years on enterprise-wide model-driven development for the financial services industry. In particular, he engineered the IAA Specification Framework, a generic product and agreement design, the IAA Business Object Model, and the IAA-XML approach to enterprise-wide integration. He has also worked with many insurance companies to help them deploy model-driven solutions in their enterprises. His current work deals with model-driven architectures, model transformations, and service-oriented architectures, and how all these topics can help address concrete business issues.

Peter Eeles
IBM Rational Software, 1 New Square, Bedfont Lakes, Feltham TW14 8HB, Hursley, UK (peter.eeles@uk.ibm.com). Mr. Eeles is an IBM Senior IT Architect, and has spent much of his career designing and implementing large-scale distributed systems. He is based in the United Kingdom and assists organizations in their adoption of the Rational Unified Process and the IBM Rational toolset in architecture-centric initiatives. He is co-author of *Building J2EE Applications with the Rational Unified Process* (Addison-Wesley, 2002) and *Building Business Objects* (John Wiley & Sons, 1998).

Simon Johnston
IBM Rational Software, 4205 S. Miami Blvd., Durham, NC 27709 (skjohn@us.ibm.com). Mr. Johnston is a member of the IBM Rational strategy team and is responsible for the business-level tooling strategy. He has undertaken a number of standards-related activities for both Rational Software and now IBM in the area of XML (W3C™ Schema working group), Web Services (RosettaNet architecture team), and modeling (OMG UML and OCL teams). He was the author of the UML Profile for Software Services and primary author of the RUP Update for SOA.

Rick Weaver
IBM Software Group, 2 Campus Circle, Roanoke, TX 76262 (weaverrw@us.ibm.com). Mr. Weaver is a Senior Consulting Certified Software IT Specialist and has focused on Business Integration and SOA development. He has worked with customers around the world helping them successfully use IBM development tools to solve their business integration challenges. Mr. Weaver is currently a portfolio manager for WebSphere Development tools, helping drive IBM tool strategy in the IBM Software Group. ■