Service-oriented architecture: Programming model and product architecture

D. F. Ferguson
M. L. Stockton

IBM products increasingly implement a service-oriented architecture (SOA), in which programmers build services, use services, and develop solutions that aggregate services. IBM Software Group middleware products and tools support the development and deployment of SOA solutions, and increasingly make functional interfaces between components and products visible through a service model. Software Group components will increasingly use SOA standards for intracomponent communications. Our move to SOA encompasses both the programming model and lower-level infrastructure software, for example, systems-management and storage-management application programming interfaces and functions. This paper concisely defines the IBM SOA programming model and the product architecture that supports it. We provide the motivation for our programming-model and design decisions. This paper also focuses on the architectural concepts that underlie our programming model and product architecture.

INTRODUCTION

This paper provides an overview of IBM's programming model and product architecture in support of service-oriented architecture (SOA). The profound implications of SOA and Web services for IBM products and programmers who use them are too sweeping for a single paper to cover in detail. Instead, this paper focuses on a broad overview of the concepts and architecture. We refer the reader to other sources, in this issue and elsewhere,[1,2] for more detail.

The programming model concept

A programming model defines the concepts and abstractions that developers build and use. In this paper, we use the terms *developer* and *programmer* loosely. A key element of our SOA programming model and supporting development tools is to enable nontraditional roles to implement services and assemble solutions by using services. A business analyst defining business processes and a marketing specialist defining policies that classify customers and compute product discounts illustrate what we mean by *role*.

Runtime products, such as WebSphere* Application Server, DB2* and CICS* (Customer Information Control System), run or "host" the programming model artifacts. Development tools support the modeling and implementation of programming model artifacts, their assembly into applications (solutions), and their deployment into the runtimes. Finally, systems management products, agents, and instrumentation support the administration of the runtimes and the programming model artifacts they host.

Although there is no generally accepted definition for a programming model, for the purposes of this paper we define it to be a set of part types that programmers build and a set of roles grouping members of the development and administrative community who have similar skills and knowledge. Part types encompass the diversity of programming model artifacts: Hypertext Markup Language (HTML) files, database stored procedures, Java** classes, XML (Extensible Markup Language) Schema definitions, C structs (C programming language syntax for defining data structures) defining MQSeries* messages, and so forth.

Categorizing developers by role helps us produce role-appropriate tools that enable nonprogrammers to implement services and assemble solutions from services. This enables the participation of new kinds of developers, such as a business analyst defining business processes and a marketing specialist defining policies that classify customers and compute product discounts. For each role, a set of skills is defined, for example, a user interface developer develops interfaces presenting the functional artifacts of the application or solution. This role is assumed to know the application under development and its business goals, to understand the application's users and their tasks, to be an expert in several user-interface design methods, and to create easy-to-use user interfaces by choosing the right kind for each task.

Each role is associated with part types and application interfaces with which the role interacts (consumes or produces). For example, those in the role of designers of dynamic pages produce the part type JavaServer Pages** (JSPs**) and consume the part type JavaBeans**. These part types wrap existing sources of information and applications. Each role is also associated with the tools that the role uses; for example, a role-appropriate tool for a Web developer is a "what-you-see-is-what-you-get" page design tool for building dynamic pages, using controls associated with HTML and JSP tag libraries, and wiring the controls to JavaBeans.

This paper focuses primarily on the part types comprising the SOA programming model. Incremental extension of a person's existing skills and knowledge is the key to making Web services easy to implement and use. A service in the form of CICS COBOL transaction programs bears little resemblance to one written in the Business Process Execution Language for Web Services (BPEL4WS or BPEL, for short).[3] Calling a service from a database stored procedure differs from calling it from a JSP; the skills and expectations are different. We offer an assortment of tools to adapt the part types to various skills and to the stages of the development process.

Product architecture

Products supporting IBM's service-oriented architecture fall into two broad categories: service endpoints and the message transport fabric interconnecting them. This general architecture, populated by many products, which jointly constitute the delivery vehicle for IBM's SOA, is illustrated in *Figure 1*.

At the core is an Enterprise Service Bus (ESB) supplying connectivity among services. The ESB is a multiprotocol bus and supports "point-to-point" and "publish/subscribe"-style communication, as well as mediation services that process messages in flight. IBM WebSphere MQ, WebSphere MQ Integrator Broker, and WebSphere's support for Web services and Java Message Services (JMS)[4] are all in the first category.

A service resides in an abstract hosting environment known as a *container* and provides a specific programming metaphor. The container loads the service's implementation code, provides connectivity to the ESB, and manages service instances. Different types of services reside in different containers. (In a notable example of design recursion, the ESB itself is considered a container for mediation services.) *Table 1* lists some of IBM's major SOA hosting environments and the kinds of components hosted.

The evolution of SOA will bring access, through the bus, to an increasingly rich set of distinguished (i.e.,

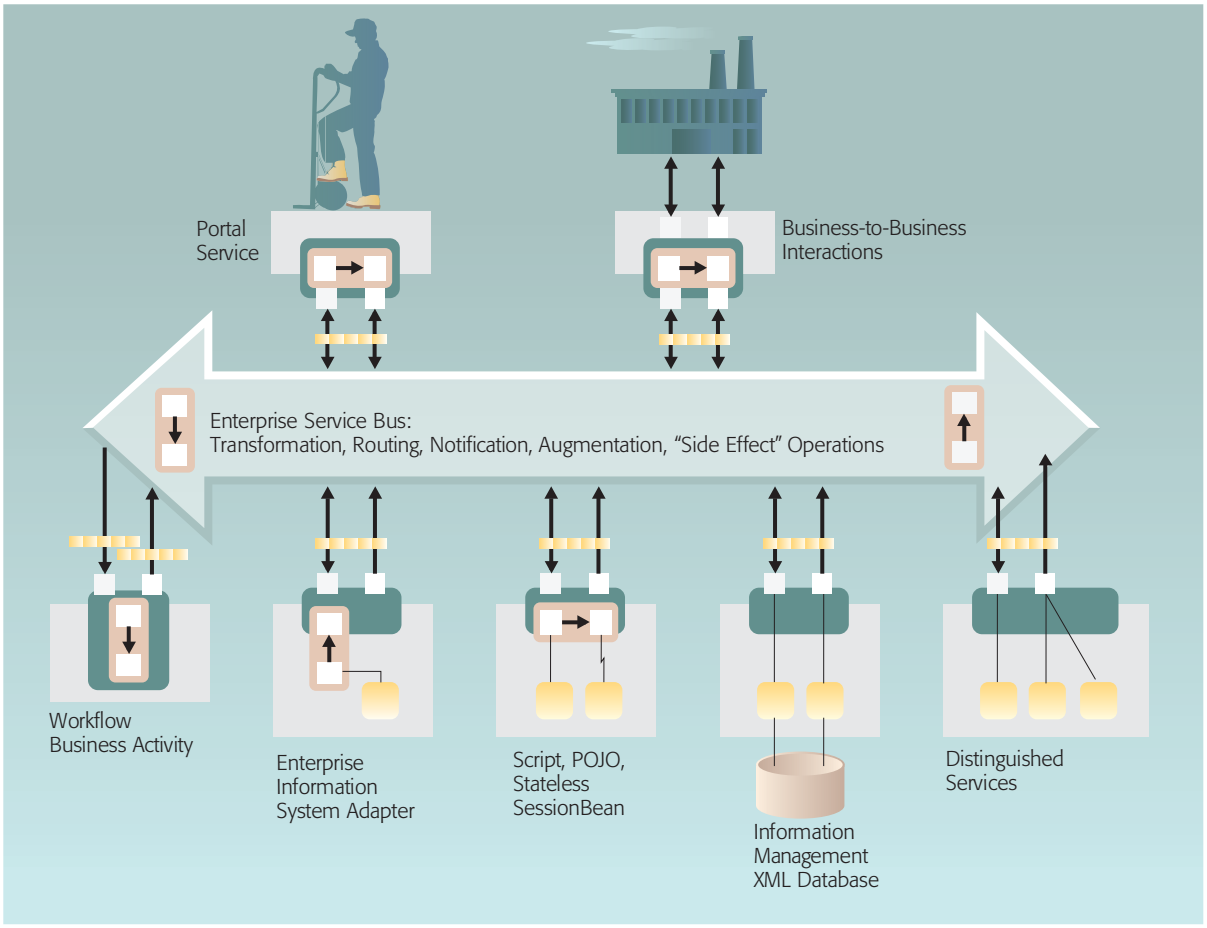

Figure 1
Product architecture

well-known) services for use by applications and containers. These services could include directory services; for example, Universal Description, Discovery, and Integration (UDDI) for locating and binding to service instances,[7] authentication services by using WS-Security secure token services,[8] coordination services provided by WS-Coordination,[9,10] WS-AtomicTransaction,[9,11] and WS-Business Activity[9,12] to manage the outcome of multiservice computations and management and monitoring.

The Model-View-Controller (MVC) paradigm underlies most modern user interface application frameworks.[13] SOA operations provide the model layer. WebSphere's Web container provides the view and controller functions through its support for Java servlets, JSPs and Apache Struts.[14] WebSphere Portal Server builds on this capability.

The remainder of this paper is organized as follows. The next section introduces concepts fundamental to the programming model and explains how they simplify the development experience. It covers Web Services as a component model, Service Data Objects (SDOs), codified design patterns for services, and the association between component types and hosting containers.

This section also introduces the most basic component types—POJOs (plain old Java objects), Enterprise JavaBeans** (EJB**) and adapters—and several simple component types for other environments and languages, for example COBOL transaction programs running in CICS or IMS** (Information Management System).

The section "Service composition and customization" describes the programming model for aggre-

Table 1 Containers hosting various component and service types

Service/Component type	Container
Transaction programs written in COBOL, PL/I and other languages	CICS or IMS. Programmers can use SOAP/HTTP, WebSphere MQ and J2EE (Java 2 Enterprise Edition) Connector Architecture connections to access the services.[5,6]
Business Process Choreography	WebSphere Business Integration Server Foundation (WBISF). This container supports long-lived workflow processes that implement Web Services interfaces and invoke operations on other Web services. It also supports long-running business activity transactions.
Application adapters, providing an SOA/Web service façade for existing applications and systems	Application adapter container provided by WBISF. An adapter converts from SOA protocols and formats to those of existing applications and systems. For example, an adapter for SAP converts from SOA-encoded XML-over-Hypertext Transport Protocol to SAP's existing business application programming interface formats and remote function calls.
Services implemented by pre-defined SQL or XML queries or as database stored procedures	DB2 in conjunction with WebSphere Application Server. Parameters for the query come from an SOA operation's input message, and the result provides the output message.
Services implemented using Java classes and EJBs	WebSphere Application Server

gating individual services into composite services and solutions. It introduces the business process component type with two realizations: BPEL and business state machines (BSMs). Communication between services is intrinsic to service composition. In our architecture, the ESB is responsible for mediating these message exchanges. This section covers the ESB architecture and product realization and explains our programming model and architecture for customizing business services. Business policies (for example, the definition of a "gold customer") change over time. This section documents our approach to customizing and evolving business rules without source code changes or application redeployment.

Much of the literature on SOA ignores the data dimension. Referring to services as "stateless" is quite common. SDOs are one aspect of the intersection of SOA and data. The section "Services and data" explains the integration of SOA with database management: how to publish data through services and integrate services into database operations. The section "Services and user interfaces" outlines the programming model and product architecture for user access to services. SOA also underpins the Web Services Remote Portlet specification, which uses Web services to integrate portal systems.

IBM's integration technology is based on SOA concepts; our SOA strategy includes both management using Web services and management of Web services. The section "Services and management" describes our architecture for building distributed systems and application management solutions from SOA and Web services. Once SOA applications become pervasively deployed, it is necessary to manage them; this section discusses architectural approaches and evolving standards for this. The section "Development tools" summarizes SOA support in our development tools; a thorough treatment of this topic would require a separate paper. The concluding section, "Advanced concepts," examines some of the current areas for research, standards, and advanced development.

THE BASICS: WHAT IS A SERVICE?

Despite the fact that many customers and independent software vendors have been implementing SOA-based applications, integration layers, and solutions for years, there is still no generally accepted definition of "service" or "service-oriented." This paper employs a very narrow, technical

definition; a service has a well-defined interface (with a set of messages that the service receives and sends and a set of named operations or verbs), an implementation of the interface, and, if deployed, a binding to a documented network address. Examples of services falling within the scope of our definition include a message-driven application that processes WebSphere MQ messages, a set of CICS or IMS transaction programs, and a Java class.

A Web service is a service that, at minimum, defines its interface by using the Web Services Description Language (WSDL)[15] and is accessible by using a protocol that is compliant with Web Services Interoperability (WS-I).[16,17] Because automatic transformation between Web-service constructs and more traditional approaches to defining services (for example COBOL, C, and Java) is a feature of the IBM runtimes and tools, the terms "service" and "Web service" are often used interchangeably.

Our programming model and architecture do not burden programmers with the complexity of writing WSDL or the overhead of using SOAP (Simple Object Access Protocol) and HTTP (HyperText Transport Protocol). Programmers using Java can build and use Web services relying only on Java interfaces and classes; COBOL programmers can do the same while relying solely on COBOL transaction programs. The runtime architecture optimizes bindings for service access, using Java Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) or JMS. These optimizations are transparent to programmers. Application development tools automate the generation of WSDL from COBOL, C, Java, and so forth. The IBM SOA supports standards, however. The WSDL is available for exchanging interface information, and a WS-Interoperability binding is available for communication.

An evolving component model

Most of the literature on Web services, especially standards, focuses on service interfaces and their use; this paper focuses instead on the programming model for implementing services and assembling them into solutions. A component model simplifies the process of building and assembling services. Logically, a component is defined by the set of six values listed in *Table 2*.

The programming model offers two formats for component definition. The first is a *control file*: this is a document that, by reference, associates all the parts of the component. For example, referring again to the six values in Table 2, the file references the WSDL definition (i.e., the interface provided), the Java class that implements the component (the implementation artifact), the associated policy documents (policy assertions), and so forth. These can be references to the file system, class path, source code control system, or Web URLs (uniform resource locators). The control file format gathers several individual programmer-developed artifacts into a collection that comprises the component. Application development tools aid in defining the control file. The second format uses *pragmas*: these are structured comments specifying the same information, but contained within the body of a single source file. Each component type has an associated source file format for its implementation artifact, for example a Java file or an SQL (Structured Query Language) file. WebSphere Rapid Deployment is a tool that simplifies defining a service in Java by using the pragma format. The annotations supported in WebSphere Rapid Deployment can generate all the individual elements comprising a component from a source file containing pragmas. For example, structured comments in a Java source file can indicate which Java methods will become Web-service operations in the generated WSDL defining the component's service interfaces. We will illustrate this concept further in the discussion of individual component types.

Component types and simplifying development

Before the architecture described herein, our programming model and tool experience was focused on the infrastructure, as (the tongue-in-cheek) *Figure 2* illustrates. Our tools would demand, "Which type of Enterprise JavaBean do you want to build?" This exasperating question evaded the programmer's true intent: to implement an element of a business solution, for example, converting documents from one format to another. The programming model presented here enables developers to define business logic without being concerned with what the business logic becomes upon deployment.

To achieve this transparency, we introduced an extensible set of service component types, each suited for a developer with a given set of skills who performs a specific task by using a certain tool optimized for that task. For queries, the programmer

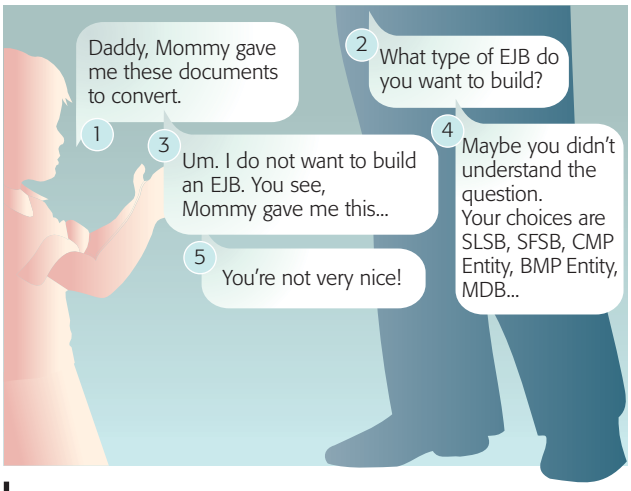

Figure 2
Programmer impasse

implements an SQL file; for document conversion, Extensible Stylesheet Language Transformations (XSLTs), and so forth. There is no need for the developer to know that a Web service, EJB, or other artifact is generated upon deployment. Each service component type also supports templates: that is, recurring design patterns for implementing services within a type. The programming model and tools support extension of a set of templates.

Figure 3 lists some service component types, showing the relationship between more specific types (at the bottom of the tree) and more general types (at the top of the tree). Programmers build the leaf elements of this tree, concentrating on the problem to be solved and the tool for doing so, not on the resulting artifacts. The focus is thus on the skills of the developers and the concepts they understand. The remainder of this paper elaborates on this theme and provides detail on elements of the taxonomy.

The basic types of service component

This section presents several typical component types by way of illustrating the extensible set of service component types just mentioned.

POJO and stateless SessionBeans

The most basic type of service component implementation is a POJO. JSR (Java Specification

Table 2 Six values that define a component

Interfaces Provided	How one invokes the component; typically WSDL, although the programming model and tools also support other languages.
Implementation Artifact	The component's executable to be hosted in a container at runtime; for example, a Java file, BPEL document, SQL file, and so forth.
Policy Assertions	Declaration of the services that the component expects the infrastructure (container) to provide. Each Web Services standard (such as WS-ReliableMessaging[18] or WS-AtomicTransactions) enables a service to document its requirements via WS-Policy extensions.[19] The container reads the policy assertions and automates their implementation in a manner analogous to container-managed transactions and security in J2EE. Programmers using the service may also examine the policy assertions to determine how to correctly call the service. For example, the policy assertions may document expectations about message signing and acceptable certificate authorities.
Interfaces required (optional)	The component's dependencies on external services. Although a service's implementation calls the interface in the native way (for example, via a BPEL *invoke* or a JAX-RPC[20] *stub*), documenting its dependencies aids in application and solution assembly.
Resource Type Managed (optional)	Support for WS-ResourceFramework (WSRF), expressed by an XML Schema definition associated with the component's WSDL-defined interface; may also support other WSRF interfaces.[21] All Web services, even stateless ones, manage state. Coupled with WS-Addressing,[22] WSRF enables support for state within the service itself, mirroring the J2EE EntityBean model.
Valid Operation Sequences (optional)	An *abstract process* defining supplemental machine-readable information about a service's correct usage, for example the order in which to invoke its WSDL-defined operations. For example, `modifyPurchaseOrder` must follow `createPurchaseOrder` and cannot occur after `submitPurchaseOrder`. Although this concept was introduced by BPEL, an abstract process can be associated with any service.

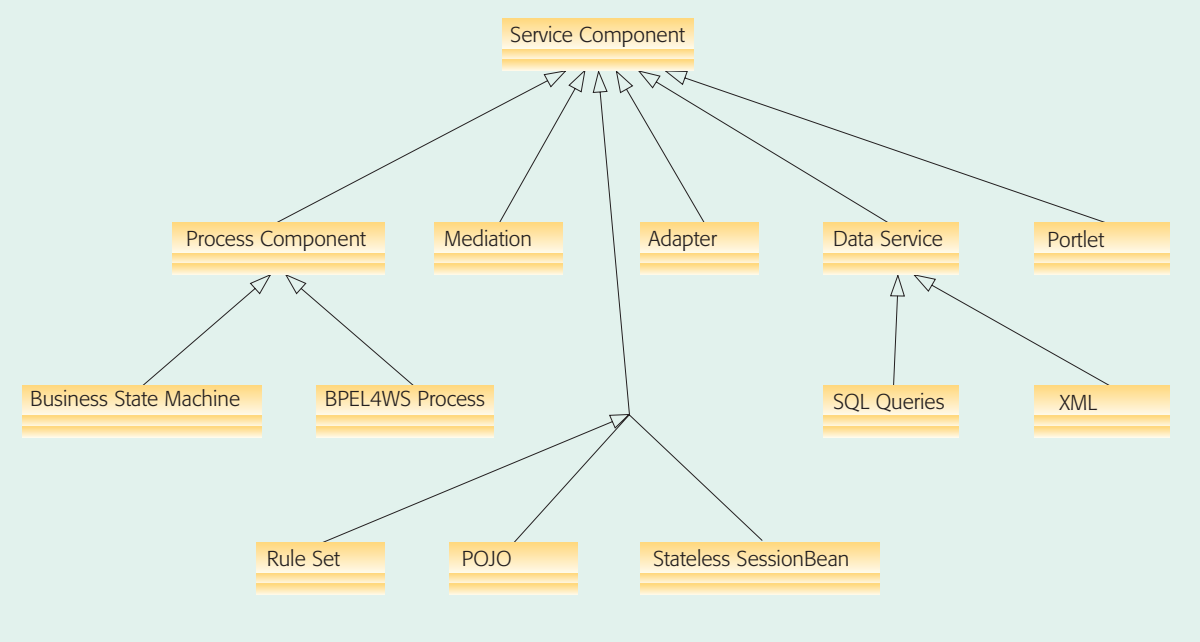

Figure 3
Some service component types

Request) 109 defines the model and architecture for implementing Web services in J2EE** (Java 2 Enterprise Edition).[23] WebSphere Studio can publish a Java class through a Web-service abstraction. The Java class runs in the Web container and has full access to the J2EE programming model's facilities. The WebSphere tools and runtime automate the conversion from SOA-encoded XML to the Java interface and operations of the POJO and vice versa. Programmers may also use stateless SessionBeans to implement services. WebSphere Studio tools automate publishing a stateless SessionBean through a WSDL/SOA abstraction.

WebSphere Rapid Deployment is a tool that simplifies defining a service in Java by using the pragma format described previously. Using an editor, a programmer annotates the Java source file with control tags derived from the XDoclet model.[24] These tags specify whether the component is a POJO or a stateless SessionBean, the values for deployment descriptors (e.g., for a transaction model), and operations that become part of the remote interface and WSDL. Placing the file into a certain directory causes "rapid deployment" of the service defined by the annotations. The model is similar to the tool support for JSPs and Java Web Start.[25]

Application adapters

An application adapter is another very common service component type. WebSphere Business Integration (WBI), WebSphere Portal Server, and WebSphere Information Integrator (WII) exploit a common programming model and adapter portfolio. An application adapter makes an existing system or application look like a Web service or a JavaBean. The functions performed by an application adapter fall into three categories, which are described in the following: protocol and connection adaptation, message format adaptation, and sequence or operation adaptation.

Protocol and connection adaptation. Most existing systems invoke applications or transaction programs through remote procedure call (RPC) or messaging protocols. For example, CICS uses the External Call Interface and Advanced Program-to-Program Communication (APPC); IMS uses various APPC and other Systems Network Architecture (SNA) protocols; and SAP uses Remote Function Call and MQSeries messaging interfaces.

In the most primitive case, the application adapter must simulate the inputs expected by an existing terminal user interface: a terminal and a user. For existing protocols, the application adapter imple-

ments a connection manager and connection pool following the J2EE Connector Architecture (J2C). The connection manager pools connections for efficiency and manages reuse across users and transactions. It also provides global sign-on by integrating with the application server's support for user credential mapping; in addition, it integrates the legacy protocol's transaction model with that of the application server.

Message format adaptation. The existing system expects inputs in a specific format, for example 3270 screen layouts, C "structs" (i.e., C programming language syntax for defining data structures), COBOL records, or a vector of name/value pairs. The J2C model with WebSphere or Rational* tools can import message or structure definitions from existing systems.[26] The tools generate a transformation artifact that converts from XML (or Java) to the back-end system's binary format. The set of transformation artifacts can be deployed as a SessionBean or Web service. The following example typifies this message format adaptation logic.

A caller invokes an operation on a Web service implementing the adapter pattern. The adapter does the following:

- Calls the transformation artifact, for example, a generated Java transformation class, to convert from the XML input message to the back-end system's binary format, for example, a byte array overlay for a C structure.
- Reads configuration information to determine the transaction program (interaction specification) to invoke by using the converted data.
- Accesses the connection manager to obtain a connection to the back-end system. The connection manager returns the optimal connection, supporting affinity and reuse for the user and the transaction as well as other policies.
- Invokes the back-end application through the connection, passing the verb and converted message.
- Receives the response and invokes a transformation artifact to convert from the back-end message format to the XML (or Java) format for the response.

The result is returned to the caller.

Sequence or operation adaptation. In some cases, neither protocol nor message format adaptation will suffice. The adapter might require a highly customized approach, tailored to the existing system's nuances. It may modify the sequence of operations to match that of the existing system, or it may emit multiple messages to the back-end system in response to a single input message carrying multiple parameters. This level of adaptation is distinguished by mappings that are more complex than one-to-one.

CICS and IMS transactions

The abundance of transaction-oriented business application programs (and data) for the CICS and IMS environments can be rendered as service components. New IBM-provided functionality and tools unlock significant business value by weaving these existing programs into the service-oriented paradigm. The transactional style typical of IMS and CICS programs lends itself to publication of these programs as services and operations. Because these applications are usually structured around verb-like transaction programs, each of which receives a message and responds with a message, it is natural and intuitive to map a transaction to an SOA operation and a message to XML.

CICS

Most existing CICS applications can be exposed as Web services, provided they have a well-established "commarea"-type interface. A commarea is a formatted message buffer which programmers typically define for messages that a transaction program receives and returns using COBOL, PL/I, or C. At the protocol level, CICS Transaction Server for z/OS* Version 3.1 supports SOAP 1.1 and 1.2[27] for Web Services Interoperability (WS-I); optional plug-ins add WS-Security (SOAP Message Security), and WS-AtomicTransactions. Interaction styles including synchronous interactions over Hypertext Transport Protocol (HTTP) or Secure HTTP, WebSphere MQ-based asynchronous interactions, and one-way asynchronous interactions are supported.

Infrastructure components can examine and transform entire SOAP messages, or specific SOAP headers, through an easy-to-use runtime interface. A complementary tool generates converters to map between SOAP payloads and commarea structures. It can also generate XSDs (W3C** XML Schema definition language) describing the interface. New XML-aware CICS applications can benefit from the high-performance z/OS XML parser of the Enterprise COBOL and PL/I compilers.

To let applications written in any CICS-supported language access Web services offered on other servers, the familiar EXEC CICS application programming interface (API) is extended in a manner that is very natural to a CICS programmer. The tools enable CICS-hosted Web services to be published as standard WSDL-described services, enabling support for standard Web-service requestor patterns. It will be possible to provide standard WSDL descriptions of all significant services provided by CICS applications.

IMS

The IMS SOAP gateway affords the ability to seamlessly expose existing and newly created IMS application assets as Web services, in conjunction with IMS Connect capabilities in IMS Version 9. The rollout of the gateway will start with SOAP server support for synchronous interactions over HTTP and HTTPS (to enable the IMS application to receive inbound service requests). Additional functions such as SOAP client outbound support and additional Web Services protocols such as WS-Security, WS-Atomic transaction, and WS-Endpoint Support are expected. Future additions may include the use of a WebSphere MQ-based asynchronous transport and the ability for IMS to act as an ESB endpoint.

The mapping of an IMS transaction to a Web-service operation is implemented by a collection of several files: an XML-COBOL converter, a WSDL Web-service interface definition, and an XML correlator. The correlator relates the Uniform Resource Name (URN) of the application to the name of the associated XML-COBOL converter. The URN specifies the appropriate data conversion for each incoming SOAP message. The correlator also contains protocol details enabling connection establishment between the SOAP runtime and IMS Connect. An XML enablement utility in WebSphere Studio Enterprise Developer generates these file artifacts to repurpose IMS COBOL applications as Web services.

A gateway tool automatically deploys server- and client-side artifacts. From information in the WSDL file, the tool automatically generates and deploys a Java application, including an internal service file and all Java beans in the SOAP Gateway server, to invoke the IMS transaction. From the same WSDL file, the tool also generates a Java SOAP client that can run the IMS transaction by invoking the Web service.

Enterprise Generation Language and other languages

The Rational software development platform defines the Enterprise Generation Language (EGL) and a supporting tool suite.[28] EGL is a classic fourth-generation language that simplifies business application development through abstract concept definition.[29] EGL generates Java code, and from Java, it obtains its support for building Web applications and Web services.

WebSphere supports JavaScript** within JSPs through the extensible Bean Scripting Framework (BSF).[30] BSF transforms scripting functions and their parameters into Java bean operations within the Java runtime, and vice versa. Because Java beans inherently support Web services (via POJO and JSR 109), one can program in a scripting language and publish the result as a Web service.

Simplified data access through SDOs

SDOs are a fundamental concept in IBM's SOA.[31] SDOs make developers more productive by freeing them from technical details concerning how to access particular back-end data sources, so that they can focus on business logic. Currently, the programming models for accessing Java Data Base Connectivity (JDBC**),[32] a WSDL service, an EJB, and so forth, from a Java program are similar, but different enough to create difficulties. SDOs replace these diverse data access models with a uniform abstraction for creating, retrieving, updating, and deleting business data used by service implementations.

SDOs define a uniform paradigm of data graphs to access and manipulate data from heterogeneous sources, including relational databases, XML data sources, Web services, and enterprise information systems. A *data graph* is a collection of tree-structured objects that may be disconnected from the data source. With SDOs, an application does not connect to a data source directly. Instead, it accesses an intermediary called a data access service (DAS) and receives a data graph in response.

A DAS is an adapter that handles the technical details for a particular kind of data source. It transforms the data into an SDO graph for the client. The client application interacts with the data graph to get and change data. To apply an update to the original data source, the application returns the

updated graph to the DAS, which in turn interacts with the data source. In general, the runtime provides the implementations of the DASes, and application development tools provide support for the data graphs.

In addition, SDOs offer a meta-data API enabling applications, tools, and frameworks to introspect the data model (i.e., programmatically examine its meta-data to determine its structure) in a uniform way, regardless of its origin. The DAS translates back-end meta-data to the standard SDO format.

Implementors can define SDO types using Java interfaces, XML schema, or the Unified Modeling Language** (UML**).[33] Simple Java types are valid SDO types, saving a step for the Java implementor. SDOs support both dynamic and static data access.

The dynamic model for SDOs (which is the default) lets programmers get and set data elements in the data graph by name. This is particularly useful when the type of the SDO is not known at compilation time. The client program or service queries the SDO to learn its structure, then reads and updates any element by name. For example, one could write a generic SDO-access function and then populate it with element-specific meta-data in order to access individual SDOs. The static model employs named, typed Java interfaces. Each data element has its own individual "getter" and "setter" method. A tool generates static interfaces from dynamic ones.

SDOs are important for data representation even if there is no classic data source present. Examples of this kind of usage include XML messages exchanged with Web services, JMS messages, XML files, and many others.

Figure 4 (in XML) shows the basis for the SDO type. The Java interface in *Figure 5*, generated from the preceding XML, illustrates the use of static interfaces.

The following examples—defining a data object containing customer data—illustrate how easy it is to define SDOs and use them with either Java or XML. Storage is allocated for the data objects by passing the SDO type definition to the SDO data factory, a runtime component that instantiates SDO data objects from SDO type definitions. The following two examples show the creation of an SDO by passing an XML schema namespace and complex type name (Example 1) as the argument or a Java interface class (Example 2) as the argument.

```
DataObject customer=DataFactory.INSTANCE.
  create("http://www.myvalue.com",
  "Customer");                                    (1)

Customer customer = (Customer)DataFactory.
  INSTANCE.create(service.customerinfo.
  Customer.class);                                (2)
```

After SDO instantiation, an implementation can access the SDO. The following code sample shows dynamic access to the customer SDO.

```
DataObject customer = ...;
customer.setString("customerID", customerID);
...

customer.setInt("stockQuantity", 100);
... = customer.getString("customerID");
...
... = customer.getInt("stockQuantity");
```

This code sample shows static access to the customer SDO:

```
Customer customer = ...;
customer.setCustomerID(customerID);
...
customer.setStockQuantity(100);
... = customer.getCustomerID();
...
... = customer.getStockQuantity();
```

In *Table 3* and *Table 4*, we further illustrate the simplicity of the programming model promoted by SDOs with examples of access to an XML file service and to a relational database. These applications can be seen to be quite similar, despite technology differences. The application developer can focus on business logic and let the service handle the implementation details of updating a persistent data store.

The simple example shown in Table 3 loads data from an XML file into an SDO data graph, prints and updates the data, then writes it back to a file. (The business goal is to change "Adam" to "Kevin".)

Although complex relational-to-SDO mappings are possible, the example shown in Table 4 uses a very

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.myvalue.com">
    <element name="customer" type="Customer"></element>
    <complexType name="Customer">
        <sequence>
            <element name="customerID" type="string"></element>
            <element name="firstName" type="string"></element>
            <element name="lastName" type="string"></element>
            <element name="stockSymbol" type="string"></element>
            <element name="stockQuantity" type="int"></element>
        </sequence>
    </complexType>
</schema>
```

Figure 4
An SDO type definition in XML

simple one: each database table is an SDO type, each row in the table is an SDO data object, and each column is an SDO property. The application logic is the same: execute a predefined query to read the database, print and update the data (change "Adam" to "Kevin"), and save the changes to the database. The database query returns two rows from the CUSTOMER table.

What if another application had accessed the database and changed values after our example application had obtained its data graph? On a write operation, the data access service examines the change summary to determine how to apply that update to the data source. The database can use optimistic concurrency control to ensure that the database last contained the value "Adam" before this change (otherwise, another application might have changed the data first, possibly requiring some error recovery in the application). Some services implement more advanced forms of optimistic concurrency; the change history provides the original values needed for those algorithms.

SERVICE COMPOSITION AND CUSTOMIZATION

Our programming model offers several ways to compose new services from existing ones. *Structural composition* is the assembly of modules and solutions from existing services. Interfaces that a service needs are "wired" to interfaces that other services provide. This wiring metaphor is similar to defining UML collaboration diagrams.

Behavioral composition is the definition of a composite service, called a *process*, through a classic procedural programming metaphor. The services to call, the order, and the aggregation of the results are defined. Processes are well-suited for business workflows because of their state model, lifetime, and transaction model. BPEL processes and BSMs (to model complex, stateful business process concepts like purchase orders or trouble tickets) are examples of process components.

It is commonly thought that the main purpose of SOA is to enable reuse. The composition model allows programmers to find services that have the desired interfaces and infrastructure policies and aggregate them into new services. These new services can themselves be composed. It is unlikely, however, that a service can always be reused as is, without customization or tailoring. When change is needed, the current state of the art involves source

```
public interface Customer {
    public String getCustomerID();
    public void setCustomerID(String customerID);
    public String getFirstName();
    public void setFirstName(String firstName);
    public String getLastName();
    public void setLastName(String lastName);
    public String getStockSymbol();
    public void setStockSymbol(String stockSymbol);
    public int getStockQuantity();
    public void setStockQuantity(int stockQuantity);
}
```

Figure 5
An SDO type definition in Java

Table 3 An XML file service

`<customers xmlns="http://customers.com">`
`  <customer SN="1" firstName="Adam"/>`
`  <customer SN="2" firstName="Baker"/>`
`</customers>`
	Define the XML file to be read as a root data object corresponding to the root XML element, and a many-valued `customers` property. The `customers` property contains one data object for each customer element in the XML file. Each `customer` has two properties: `SN` and `firstName`.

`DataObject root=xmlService.load(InputStream);`	Read the file data.

`Iterator i=root.getList("customer").iterator();`
`while (i.hasNext()){`
`  DataObject cust=(DataObject) i.next();`
`  String name=cust.getString("firstName");`
`  System.out.println(name);`
`}`
	Walk through the list of customer data objects and print the first name for each.

`DataObject customer1=root.getDataObject("customer[1]");` `customer1.setString("firstName", "Kevin");`	Set the `firstName` property of the first customer data object to `Kevin`. The middleware updates the change summary (not shown) to indicate what data was changed.
`xmlService.save(OutputStream, root);`	Write the data objects to the file.

`<customers xmlns="http://customers.com">`
`  <customer SN="1" firstName="Kevin"/>`
`  <customer SN="2" firstName="Baker"/>`
`</customers>`
	The result is an updated XML document.

code modification. Our SOA programming model also enables building services and modules that programmers can customize without source code modification by using templates, patterns, tailoring, mediations, and the strategy pattern.[34]

In the strategy pattern, a variable that is computed at runtime selects one of several possible conditional execution paths. We extend the strategy pattern so that this computation can be implemented by evaluation of a rule or by execution of separately supplied program logic. This pattern helps us define reusable software components that can be easily tailored by a less-skilled person without having to analyze, change, recompile, and redeploy the source code.

Structural composition: wiring, assembly, and mediation

This section discusses collections of services, their connections, and the ESB.

Connections and modules

In structural composition, services document the required interfaces to be provided by other services (imports) and the interfaces they offer (exports), so that a developer can wire them together. Wiring—defining logical connections from imports to exports—is done by a software tool with an asset view, such as that shown in ***Figure 6***. The wiring approach improves the usability of a visual programming tool. Instead of typing the name of the reference used by an interface, the programmer simply places a wire (draws a line) connecting the output of one interface to the input of another interface.

A collection of services wired together into a bundle is called a *module*. Like a service, a module can declare imports and exports and be wired into a larger assembly. Wires defined at assembly time are not satisfied until, at runtime, they are bound to deployed component instances.

In summary, programmers implement services that define the interfaces they implement and require. Programmers can assemble modules from service components and document the service interfaces that a module exports and imports. The model is recursive; modules can aggregate other modules.

Table 4 Access to a relational database

(A) Database prior to execution of application logic

CUSTOMER ID (int, primary key)	CUSTOMER FIRSTNAME (String)	CUSTOMER LASTNAME (String)
1	Adam	Smith
2	Baker	Street

(B) Application logic

```
DataObject root=rdbService.get();
```
The rdbService queries to obtain data from the database.

```
<ROOT>
  <CUSTOMER ID="1" FIRSTNAME="Adam" LASTNAME
    ="Smith"/>
  <CUSTOMER ID="2" FIRSTNAME="Baker" LASTNAME
    ="Street"/>
</ROOT>
```
The same data could have been equivalently expressed in XML.

```
Iterator i=root.getList("CUSTOMER").iterator();
while (i.hasNext()) {
  DataObject cust=(DataObject) i.next();
  String name=cust.getString("FIRSTNAME");
  System.out.println(name);
}
```
Print each customer's first name.

```
DataObject customer1
  =root.getDataObject("CUSTOMER[1]");
customer1.setString("FIRSTNAME", "Kevin");
```
Set the FIRSTNAME of the first data object to Kevin. The middleware updates the change summary (not shown) to indicate the change.

```
rdbService.update(root);
```
Write the updated data to the database.

(C) Database after execution of application logic. Note that row 1 has been updated.

CUSTOMER ID (int, primary key)	CUSTOMER FIRSTNAME (String)	CUSTOMER LASTNAME (String)
1	Kevin	Smith
2	Baker	Street

Mediations

A mediation service defines the "behavior" of a wire and is invoked by the ESB whenever a message traverses the wire. Mediations typically do one of the following; content-based routing, transformation, augmentation, or "side effect" operations:

1. *Content-based routing*—The mediation routes the message to one or more alternative destinations based on its content. For example, it may route a message to the proper credit card processor, based on message payload.
2. *Transformation*—The mediation transforms messages and maps operations, adapting the required interface to the implemented interface.
3. *Augmentation*—The mediation retrieves additional information to put the message into the form expected by the target service.
4. "*Side effect*" *operations*—The mediation performs an extra operation needed by the infrastructure or by an enterprise policy, beyond that specified in the data payload. For example, it may log financial messages exceeding a certain value. This policy can be implemented at the infra-

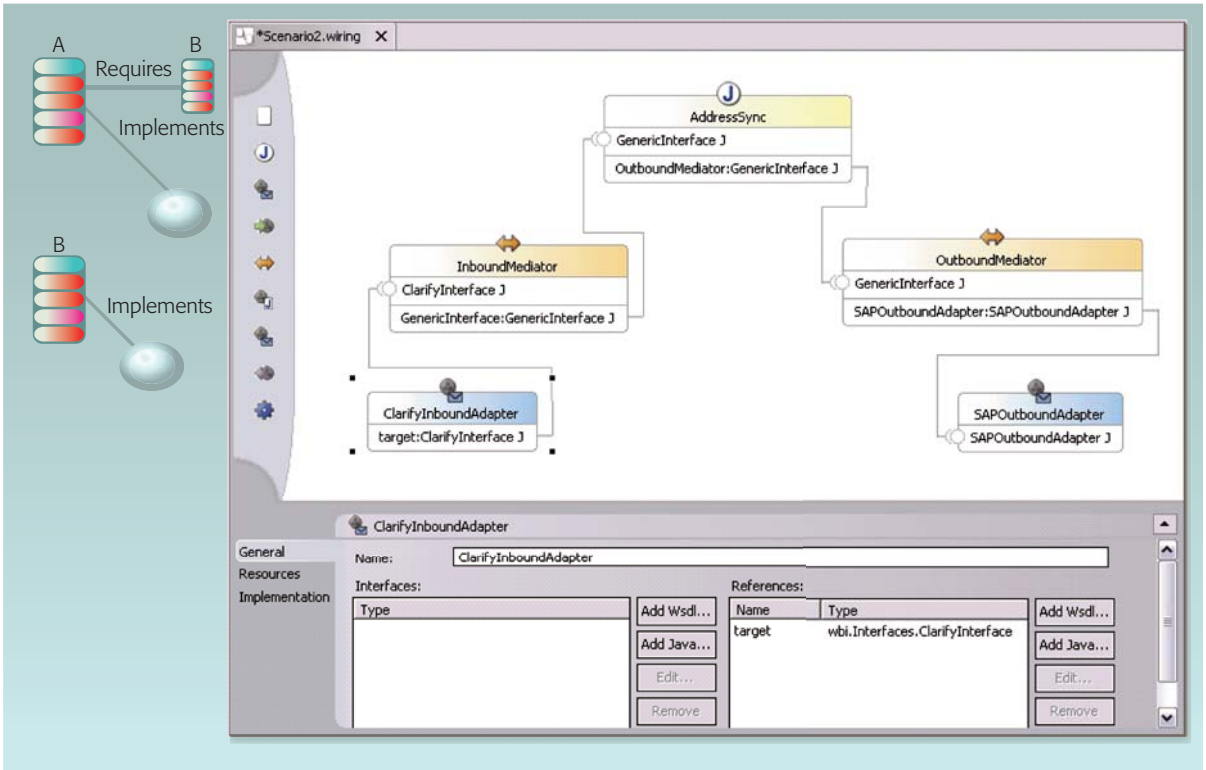

Figure 6
Wiring to connect service components

structure level without revising the affected applications.

Mediations are first-class services with supporting tools. WebSphere MQ Integrator supports powerful, complex mediations; for example, one may chain an augmentation, transformation, and routing mediation. Programmers can also implement mediations using the Web-service capabilities in WebSphere and other products.

THE ENTERPRISE SERVICE BUS

The ESB performs a variety of functions in the programming model, for the programmer, assembler, deployer, or administrator. In structural composition, the designer wires compatible service interfaces together. After deployment, those wires traverse the ESB. Each wire has a mediated destination (for point-to-point messaging) or topic (for publish/subscribe and event-based messaging, including JMS and the evolving WS-Eventing[35] and WS-Notification[36] standards).

The deployer establishes communication paths between services by defining wires and attaching appropriate mediations to them. To create an event-driven service, one connects an import to a topic with a filter. To make a service emit an event when called, one connects an export to a topic. These can be changed by an administrator.

Physically, the ESB backbone consists of WebSphere Platform Messaging, WebSphere MQ, and WebSphere MQ Integrator nodes. ESB endpoints are on WebSphere and other servers. Thus, a mediation "on a wire" can run in an ESB endpoint container with a local quality of service. With its multiple protocols and formats (including WS-I, IIOP, and MQSeries), the ESB supports persistent and transient messages and events, as well as transactional sending and receiving of messages.

Behavioral composition: Process components

This section introduces two typical kinds of process components: BPEL processes and BSMs.

BPEL is a traditional approach to workflow that builds on SOA and ESB. Programmers often think of a workflow process as an action or "verb," for example: `CreatePurchaseOrder` or `OpenAccount`. Execution of the verb may take multiple steps and paths, and it may synchronously or asynchronously invoke many Web services, Java classes, or EJBs.

If a workflow process is a verb, then as a complementary process, a BSM is a noun that identifies a thing, such as a purchase order, trouble ticket, or life-insurance-policy application. Here a verb, such as `createPO` or `cancelPO`, instead of being a separate workflow process, is an operation upon the thing. This model allows BPEL processes to be invoked in the operations on the BSM. Neither approach—BPEL or BSM—is superior. Rather, they are functionally equivalent service abstractions. We offer a choice to ensure a natural fit for the task at hand and the programmer's skills.

BPEL-based business processes

A process is represented by a directed graph of activity nodes representing a single business activity, for example, a "quick loan" service in a banking business. Processes are classified as short-running or long-running. *Short-running* processes have a single transaction per process and can be defined by using basic process choreography. *Long-running* processes persist in their execution state in a database. They require advanced process choreography and support transactions at the activity level. They may include compensations to roll back partially completed work in the event of a failure for long-lived processes that cannot rely on the resource locking mechanisms of transaction managers or for operations that lack transaction support.

The business process choreography container in WBI Server Foundation hosts business processes, that is, workflows, written in BPEL, which is described extensively elsewhere. Here we summarize only a few of its features and extensions:

- *Incorporating people into processes*—Humans perform some of the steps in a typical business process, including complex context-aware situations of assigning work to people and the "four eyes principle," where a second approval step can be performed by any approver except the first approver. The business process choreography engine and the WebSphere Studio Application Developer—Integration Edition tool support incorporating human tasks into the workflow.
- *Embedding processes into J2EE and using Java as a first class language within a process*—IBM and BEA Systems, Inc. are proposing Java extensions for BPEL, including BPELJ, which would let programmers use Java to implement activities, formulate BPEL expressions, and manipulate work data within a process.[37]
- *Quality of service extensions*—These include the ability to fine tune transaction boundaries or produce audit logs needed by production systems.
- *Integrating the business process choreography engine with the transaction engine and activity service in WebSphere*—Future integration activities are planned for WS-Coordination, WS-AtomicTransactions, and WS-Business Activity.

A visual editor in the Rational/WebSphere tool suite can be used to build, test, and deploy BPEL-implemented business processes as services. The model can also import service interfaces into an asset view, making the interface operations available for invocation from the process.

BSMs

A BSM has an associated state machine definition that is in a specific state at any given time. A purchase order, shown in ***Figure 7***, is typical of objects that are readily modeled by a BSM, objects that undergo several well-defined state transitions during their life cycle.

The nodes in Figure 7 (rectangles) represent possible states of the BSM from the time that it is created until it is archived. In this example, the purchase order may be in the state `Ready`, `inApproval`, `Purchased`, `Canceled`, `Shipped` or `Delivered`. The arcs (arrows) represent events that can occur. For a BSM implemented by an EJB, an event is an operation on the EJB's WSDL port type or remote interface. The current state determines which events (operations) are allowed. The runtime throws an exception if a caller attempts to invoke an invalid operation. One can also query the current state to determine an operation's validity.

When an event occurs (i.e., an operation is called), the BSM changes to a new state and invokes the associated operation or method, shown diagrammatically by an arc. *Guards* may prevent exiting or entering a state until some condition is fulfilled, and

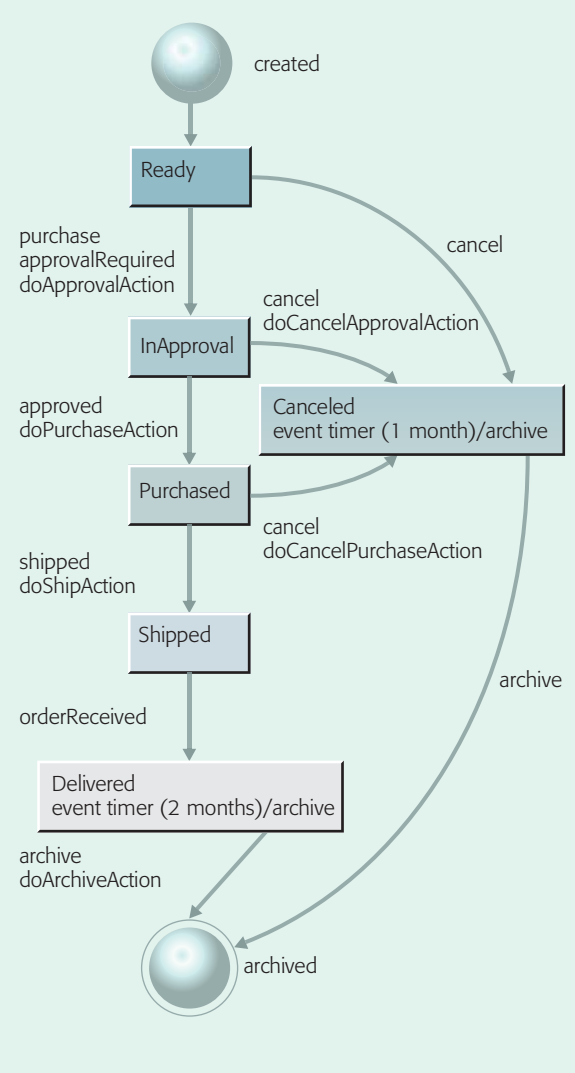

Figure 7
A business state machine

actions may be performed on state entry and exit. For example, a guard could ensure that only purchase orders under $10,000 can change from `Ready` to `Purchased`. In Figure 7, the BSM in the `Ready` state has two possible events (enabled operations): `cancel` and `purchase(approval Required)`.

When a caller invokes the `purchase(ApprovalRequired)` operation, the BSM framework does the following. It determines if the operation is valid for the current state and evaluates a state exit guard, if one exists. The guard is a private operation on the BSM, for example, a Java method. If the guard evaluation returns "true," the BSM can exit the current state. The framework then executes the action associated with the transition, in this case `doApprovalAction()` (a private operation on the BSM). For example, this operation could send e-mail to a sales manager or simply invoke an operation on another SOA component, similar to the BPEL "invoke" activity. It then evaluates the guard associated with state entry, if one exists, and enters the new state.

A BSM is an EJB EntityBean that may use container- or bean-managed persistence and that may be persisted to any supported back-end system. It may have a WSDL-defined Web-service interface. Like BPEL processes, BSM instances are stateful. The runtime provides a stateless SessionBean/WSDL wrapper for instances of a specific type, with the guards and state and private methods executing on the EntityBean.

To implement and test a BSM, a developer creates UML state diagrams, utilizes a visual design tool and wizards, or edits an XML source file defining the state machine, state data, transitions, and embedded Java for operations and guards. Implementation includes defining the state machine, its initial state (e.g., `Created`), terminal state (e.g., `Archived`), transitions, public operations, transition actions (methods and operations), and guards. The transition actions and guards must also be implemented, aided by a simple tool that assists in querying state data.

Customizing services

A *customizable service* is one that can be tailored for reuse in a new context or within an assembly, or adapted to evolving business policies, without changing the source code. A *point of variability* is a first-class programming model construct that defines a location in the code intended for subsequent customization, where the program logic can be varied, for example, by applying a rule or calling an external service that conditionally returns a value specifying one of several execution paths. Our SOA programming model introduces customizable services and points of variability, building on the strategy pattern and mediations, to facilitate the creation and usage of customizable services.

For example, if a programmer wants to implement a decision that determines if adding a line item to a purchase order is valid, then, instead of coding "if . . . then . . . " statements within the purchase

order Java file, the programmer might implement the method as follows:

```
Boolean addLineItemToPO(PurchaseOrder p,
  LineItem li) {

  //Locate the the customizing services
  POPolicies p =
    context.lookUp("lineItem
      ValidityRule");
  if (p.isValidLineItem(li) == false) {
    return false;
  }

}/Implement the addition logic below.
```

In this example, the `POPolicies.isValidLineItem()` logic is moved to the wiring from the main component and invoked by an operation on a service to which the purchase order service is wired. The logic is simply a service look up (perhaps JAX-RPC, CICS COBOL, BPEL, or a business rule written in a rule language) and operation invocation. By being externalized, the policy can be evaluated post-development, for example, by wiring, administration, configuration, or operation of a separate program. This straightforward use of the strategy pattern is a convention for good service design. The customizer, mediator, and service being customized are all arbitrary SOA services of any valid component type. This multiple-component example (customizer, service, and mediation) does not necessarily imply a long path length, as the runtime optimizes execution for co-resident components.

For example, a routing mediation on the wire could choose a variation of POPolicies, based on date and time (to vary policies according to season), purchase order value, customer identity, and so forth. Our tools make such changes easy for a nonprogramming business analyst. The WBI tools support decision tables and "if . . . then . . ." rules. Policies can be defined in IBM Workplace and WebSphere Portal Server, or changed at runtime by means of rule templates. For more information on customizing software behavior with rules, see Reference 38.

SERVICES AND DATA

Immense quantities of business-critical information residing in databases can be incorporated in service-oriented applications. One can publish SQL queries and stored procedures as Web services, federate XML data sources with databases, and invoke Web services from SQL and stored procedures, aided by the capabilities depicted in ***Figure 8***.

WebSphere Information Integrator (WII) enables the database to consume Web services. It can make data sources described by XML schema accessible through standard SQL queries, the form familiar to DB2 programmers. The tools and runtime convert XML data sources to relational tables. A set of adapters provides a common WSDL-described interface for accessing XML information from WII. The basic SQL `SELECT`, `UPDATE`, and `INSERT` commands are integrated with compatible Web-service operations.

DB2 can invoke operations on Web services, both in queries and stored procedures, from SQL. Tools bridge between Web services and SQL data models by exposing Web-service parameters and operations as nicknames or SQL user-defined functions for the SQL programmer.

To enable developers to publish enterprise information as Web services without programming, WebSphere tools expose SQL queries, database stored procedures, and XML Extender as Web services. Some of the supported scenarios are the following:

1. A Web client sends a Web-service request, which WebSphere and DB2 convert to SQL for processing in the database; message conversion is governed by mapping files, or by default, well-defined mappings between SQL and XML.
2. SQL, SQL/XML, and stored procedures are published and invoked as Web services through a feature of DB2 in conjunction with WebSphere Application Server. In the future, XQuery[39] requests will be supported in the same manner.
3. Stored procedures incorporating application code, SQL, SQL/XML requests, and later XQuery are published as Web services and can access Web services.

Figure 9 illustrates a simple Web service (written in DB2's DADX [Document Access Definition Extension]) that generates an SQL request to the database. This example defines an operation, `listDepartments`, on a WSDL Web service. The WebSphere/DB2 tools automate the generation of the WSDL/XML and the mapping between the input WSDL message and the `SELECT` predicate.

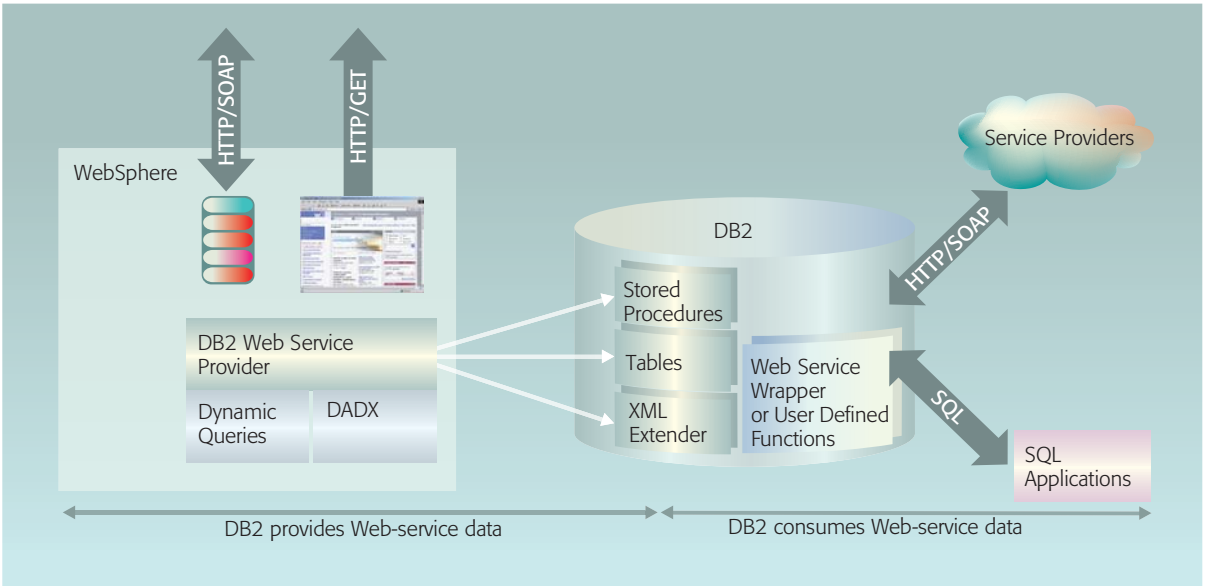

Figure 8
Information management – Web-service overview

Figure 10 illustrates a response with default XML tagging (although explicit formatting is also supported, e.g., through SQL/XML). The tools generate an XSD to define the columns specified by the `SELECT` command and a message expressing the output of `listDepartments` as a set of rows.

In the future, the standards WS-Transaction[40] and WS-Security[8] may be supported. For more information on DB2 Web services, see References 41–43. Standardization is underway for defining Web-service access to XML Metadata Interchange and relational databases and files, taking into consideration the Web Services Resource Framework.[44,45]

SERVICES AND USER INTERFACES

This section highlights a few of the major concepts involved in viewing user interfaces as services. User interfaces (UIs) occupy the "view" layer in the MVC pattern. UI technologies can render information on devices ranging from smart-phones to browsers and rich clients capable of considerable client-side processing. IBM middleware and tools connect view-layer UI technologies to model-layer Web services.

In an SOA, the environments hosting UI components are also abstracted as containers that provide well-known sets of infrastructure services. Our three major UI containers are the basic Web browser, a Web browser augmented with JavaServer Faces (JSF)[46] and dynamic HTML,[47] and a workplace client—the Eclipse rich client,[48] in addition to native WebSphere Application Server client support.

Container services are augmented by supporting technologies such as servlets, JSPs and JSP tags, Apache Struts for page sequencing, JSF for advanced page composition, and portlets to combine views of multiple applications on the same page. UI code can invoke business logic using SDOs, Web services, and so forth.

UI development frameworks can simplify the creation of complex user-facing applications. The Struts project,[14] having a large developer community and exceptional tools support, is an Apache open-source project predating the Java Portlet Specification, JSR 168.[49] Struts is a multipage MVC framework for server-based UI development using the servlet/JSP paradigm. A special version of the Struts Version 1.1 library supports JSR 168 portlets on WebSphere Portal.

JSF,[46] an MVC realization for Java Web applications, was recently standardized through the Java Community Process. JSF builds incrementally on earlier technologies. It is well-suited for portlet

```
<?xml version="1.0" encoding="UTF-8"?>
<DADX xmlns="http://schemas.ibm.com/db2/dxx/dadx" >
<documentation> Simple DADX example that accesses the SAMPLE database. </documentation>
    <operation name="listDepartments">
    <documentation> Lists the departments. </documentation>
    <query> <SQL_query>SELECT * FROM DEPARTMENT</SQL_query> </query>
    </operation>
</DADX>
```

Figure 9
Simple SQL query from a Web-service interface

development, offering portlets and servlets, state handling, validation, and "eventing" (asynchronous notification of events to interested parties). A JSF page has one or more local models that interact with UI controls on the page. These controls render outputs based on UI properties; sophisticated logic ensures their presentation at the right location. The client-side model can be wired into the ESB to send and receive events. The JSF controls process each model event affecting a page and update the rendering. A mechanism routes user input to the right UI control to cause an appropriate model event. JSF also includes a library of predefined UI controls (notebook, tree, table, graph, etc.) and a "what-you-see-is-what-you-get" tool. WebSphere Studio includes additional JSF widgets and a JSF-based visual layout wizard for portlets that connect JSF controls to SDOs. Local caching of SDO data graphs improves the user experience.

Java Widget Library (JWL), an extended widget set usable by portal and portlet programmers, adds JavaScript client-side processing to JSF and will be supported by Rational Studio. Updating the view locally on the client saves round trips to the server, shortens response time by orders of magnitude, and dramatically improves the user experience. Portlets using JWL can run on WebSphere Portal just like any other portlet.

Portals provide first-class UI support in the SOA. Portlets, their basic building blocks, let developers focus on the unique aspects of their application, while the middleware handles common functions

```
<?xml version="1.0" ?>
<xsd1:listDepartmentsResponse  xmlns:xsd1="http://schemas.ibm.com/sample/department.dadx/XSD"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<return>
<xsd1:listDepartmentsResult xmlns:xsd1="http://schemas.ibm.com/sample/department.dadx/XSD"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<listDepartmentsRow>
        <DEPTNO>A00</DEPTNO>
        <DEPTNAME>SPIFFY COMPUTER SERVICE </DEPTNAME>
</listDepartmentsRow>
<listDepartmentsRow>
        <DEPTNO>E21</DEPTNO>
        <DEPTNAME>SOFTWARE SUPPORT</DEPTNAME>
</listDepartmentsRow>
</xsd1:listDepartmentsResult>
</return>
</xsd1:listDepartmentsResponse>
```

Figure 10
Results of DB2 Web service for an SQL request with default XML tagging

for life-cycle events, per-user customization, aggregation, and integration with other components.

A portal's powerful integration of the UIs of several back-end services into a centrally managed UI can unify the fractured IT (information technology) infrastructure and give users a single view of IT services with a single UI to master. This type of integration is sometimes called "integration on the glass," that is, integration of what is presented to the end user, as opposed to integration at the application layer. Applications originally designed separately can be wired together to enable new functions. For example, an e-mail portlet wired to a collaboration portlet could filter the "in" box to display received e-mail only when the sender is online and available for a chat, a capability which might be absent from both original applications.

A surprising consequence of the portal model is improved agility for on demand businesses. Administrators become application integrators who create new applications without programming, by defining new pages, adding portlets to them, wiring the portlets together, and setting entitlements (i.e., what services a portlet is allowed to access.) A self-service portal lets users adapt their work environment to their unique needs. The portal architecture frees application developers to concentrate on building new business value.

The service interface and protocol for a local portlet is defined by the Java Portlet Specification.[49] Web Services Remote Portlet (WSRP) is the standard for remote rendering of portlets, enabling a portal to aggregate content from multiple sources.[50]

WSRP extends the integration capabilities of Web services to presentation-oriented components and enables the view layer to be shared across platforms, implementation languages, and vendors. Content and application provider services can be discovered and plugged into standards-compliant applications without any extra programming effort.

Instead of deploying each application or portlet on every server that intends to use it, there are obvious advantages to sharing applications across network boundaries. WSRP enables easier administration. Instead of managing local deployments of pluggable components, portal administrators can browse a registry for WSRP services to offer; users benefit from timely availability of new services and content integration on demand. Load is distributed across multiple servers and infrastructure cost is reduced because applications can share hosting infrastructure. For example, distributing just the presentation layer (via WSRP) of a back-end banking application preserves the application provider's secured computing environment, while enabling users to interact with the shared UI.

An additional advantage to sharing applications is control over content presentation. Content and application providers can vastly expand their reach to new users, as portals redistribute content.

SERVICES AND MANAGEMENT

This section discusses management using Web services (MUWS) and management of Web services (MOWS). The Web Services Distributed Management (WSDM) technical committee in OASIS is defining the architecture for both.[51,52] MUWS is the application of Web service technology to manage distributed systems and applications. MOWS involves the management of Web service-based applications and services.

MUWS

MUWS views a systems management agent as a kind of application adapter that wraps existing management APIs, artifacts, and protocols (analogous to how SOA integrates distributed applications). Subsystems typically support multiple standard management protocols to interact with their resources. Application adapters encapsulate these APIs in a WS-I abstraction. A MUWS agent could, for example, surface the management scripts that start or stop a system, add or remove users, and so forth, as Web-service operations on a WSDL interface. The main benefits are interoperability and simplification. Normalizing diverse management approaches into a single WSDL XML type space and communication protocol lets programmers skilled in generic Web services build software to interact with managed systems without mastering all the unique type spaces and protocols.

MUWS can also simplify complex business systems when both business and management events are invoked by Web services. Some typical IT events that occur when an employee joins an enterprise are "process employee contact information," "publish revised organization chart," and "update outsourced

Web services for payroll and direct deposit." These are business events. "Issue an X509 certificate," "create an e-mail account," and "allocate workspace on a file server" are management events. It is obviously advantageous to orchestrate all of these events from a single business process.

WSDM management, using Web Services, builds on the standards WS-ResourceFramework, WS-Eventing and WS-Notification by introducing manageability capabilities. These include integrated sets of operations, properties, events, meta-data and policy. The operations include Web-service operations for managing the resource, for example, a printer, application server, or operating system. Properties include reading or reading and writing XML elements that describe the state of the resource, such as up, down, average CPU utilization, and average response time. Events include notifications the resource may emit, such as "CPU critical" or "maximum logons exceeded." Meta-data and policy include additional information on how to interact with the service, (e.g., using WS-ReliableMessaging and WS-Security), and on its properties, operations, and events (e.g., "CPU utilization is between 0 and 100 and the averaging interval is the past 10 minutes").

WSDM defines common base functions that all managed systems and resources must support and a standard event format, based on Common Base Events (CBE), for interoperable, correlatable management events. The evolving WSDM standards will become increasingly prevalent in MUWS.

The Integrated Solution Console (ISC), built on WebSphere Portal Server, offers an environment for building systems and application management workspaces.[53] Portlets running in ISC can use Web services to interact with the Web-service interfaces of managed systems.

Tivoli* products are moving to business process choreography for complex management processes such as software change management and user identity provisioning. Evolving standards in this area include the IT Infrastructure Library, which is codifying a set of best practices for IT management.[54,55]

The Common Event Infrastructure (CEI) provides a common base schema and CBE taxonomy (now standardized in WSDM), a toolkit for adapting existing IT event logs to the CBE format, and integration with the ESB to publish events. Monitoring products are adopting the CEI to gain an integrated, correlated view of IT infrastructure, application, and business events.[56,57] CEI is a key element of business performance management.[58]

Management of Web services

All containers that host Web services provide systems management interfaces for configuring, operating, and monitoring the services they contain. In most cases, the container exposes the service management capability (an API or user interface) within the management of the container as a whole. For example, WebSphere Application Server exposes the Mbeans (management beans) and UI functions of JMX** (Java Management Extensions) in the WebSphere Console for managing Web services in hosts. This allows system administrators familiar with managing the environment to extend their skills and tools to include Web services, which in many cases are defined "bottom-up" from the existing artifacts that the administrators manage.

Tivoli monitoring and management products are evolving to provide an end-to-end view of Web-service solutions, the services they combine, and communication between the services. WSDM defines a common set of capabilities—interfaces, events, and properties—that all Web services or their containers should support in order for all Web services to have the same core set of functions. This approach also enables the composition of business and management functionality into a single endpoint, eliminating the need for additional service discovery.

DEVELOPMENT TOOLS

IBM provides tools for the entire software life cycle to help realize the SOA vision. Within the broader context depicted in *Figure 11*, this paper highlights several tools with particular SOA affinity, and *Table 5* maps specific tools to the on demand software life cycle.

Business process monitoring

Agile businesses need the ability to monitor and visualize business activities. For example, a factory manager may want to compare new orders with fulfillment or monitor inventory levels and idle capacity; a financial officer may want to scrutinize

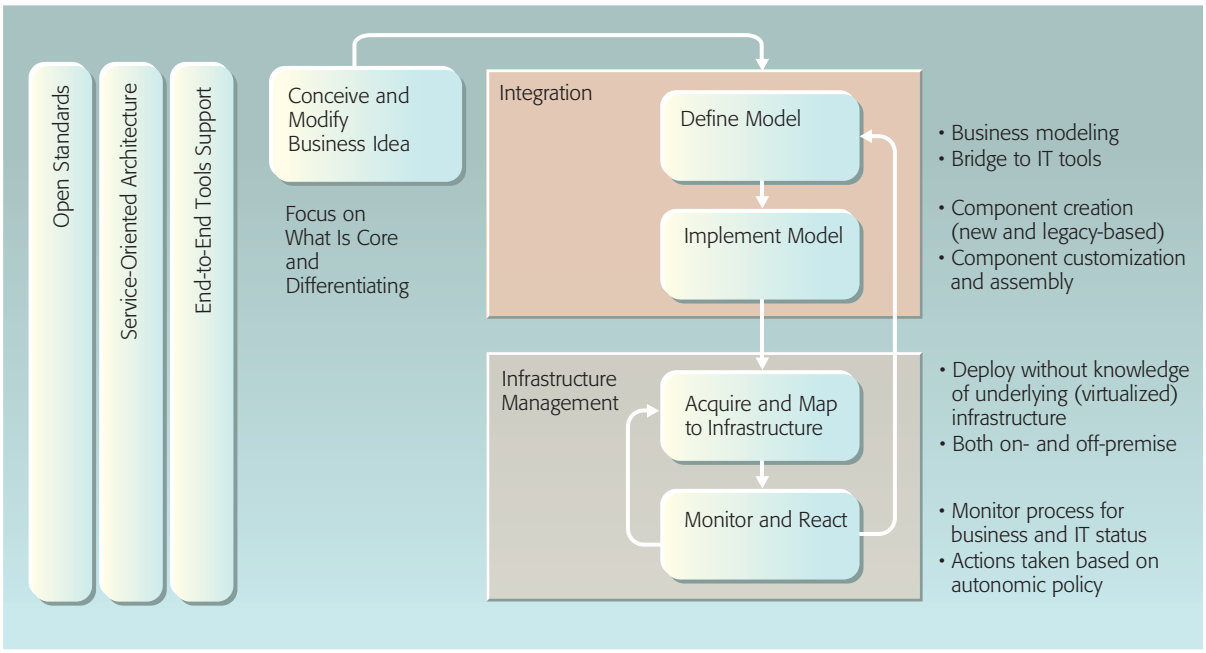

Figure 11
Life cycle of an on demand business solution

receivables, payables, and capital expenditures. In the past, this has been quite difficult. The long lag between obtaining key financial metrics and being able to act upon them has hampered profitability or made it impossible to take needed actions in a timely manner.

An SOA can help address these needs by moving the focus to higher-level business processes in a discipline called *business performance management*.[58] The objectives of this discipline are to define service-level objectives in business terms and then automate their IT realization. A key IBM offering for business performance management is WBI and its tool, the WBI Modeler.

WBI Modeler is a visual process editor that helps build a choreographed business process in five simple steps, shown in ***Figure 12***. This editor allows visual debugging of local or remote process instances. One can view and change process variables, set breakpoints before or after execution of an activity, and debug Java code. The resulting artifacts, in the BPEL language, are service compo-

Table 5 Tools and technologies for the on demand software life cycle

Phase	Tools and Technologies
Define Model	WebSphere Business Integration Modeler, Rational Architect
Implementation	Component types and patterns, BPEL, BPEL4J, JSR 170/Content Model, and Business Rules
Acquire and Map to Infrastructure	Solution Packaging, Solution Change Manager, Solution Configuration, Tivoli Intelligent Orchestrator, Tivoli Provisioning Manager, Tivoli Identity Manager, Tivoli Directory Integrator, WebSphere Identity Manager, WebSphere Business Integrator, and DB2 Information Integrator
Monitor and React	Common Event Infrastructure, Active Correlation Technology, Business Process Management, and Business Workload Management

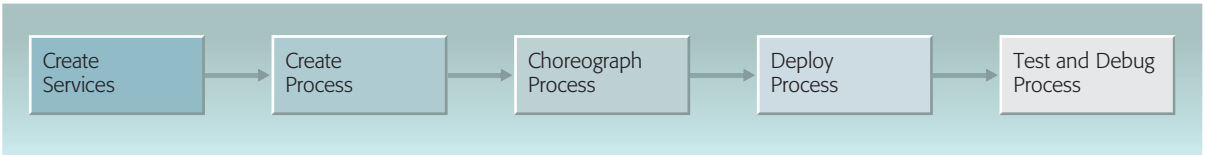

Figure 12
Five simple steps to build a choreographed business process

nents of the process type, previously described in the section "BPEL-based business processes."

Rational tools

Spanning the spectrum of developer preferences and conceptual styles, IBM's Rational tools support two modes for developing services: bottom-up and top-down. An example of the bottom-up approach is the new service wizard in WebSphere Studio Application Developer, Integrated Edition. Here, the developer first decides what kind of technology to use to implement a service: for example, an EJB, a JavaBean, or an Enterprise Information System connected via J2C resource adapters. The bottom-up approach saves time because the developer need not be concerned with the specifics of service details and can incorporate existing artifacts into a service.

The wizard is extensible: any new source of services can be imported and made available to developers. Thus, a software vendor can rapidly incorporate a new application into the SOA by creating a J2C resource adapter and the corresponding resource adapter description (RAR) file. IBM offers J2C adapters for many common services, including CICS-ECI, CICS-EPI, Host on Demand, and IMS. These include a development license and communicate through IMS Connect and the CICS Transaction Gateway, respectively. IBM also offers an adapter for SAP (mySAP.com). WebSphere supports any J2C adapter that uses the J2C Version 1.0 API.

In addition, numerous partner-developed J2C adapters are available. iWay, for example, offers 200 tested and certified adapters for use with WebSphere.[59] Fifty additional adapters for WBI are available that communicate through WebSphere MQ messaging. Future support for J2C Version 1.5 is anticipated.

A new service can also be created by using a top-down approach. Here, an abstract service is first created by using the service interface wizard to generate an empty WSDL file. Focusing on the abstract service rather than on software artifacts or program code, the developer then uses a WSDL editor to define the interface. The editor's visual mode makes it easy to visualize relationships—services and their bindings, messages and their parts, port types and their operations—or to view the WSDL source. Finally, one creates an implementation. Here too, a tool called the service skeleton wizard aids in the creation of an implementation, such as a Java or EJB implementation. This tool reduces the opportunity for introducing coding errors by generating a skeleton in Java that matches the previously defined service interfaces. The programmer can then use a Java editor to implement the operation in Java by filling in the skeleton.

ADVANCED CONCEPTS

This section discusses the infrastructure services of the Web Services standards and the modeling of stateful Web-service interactions.

Container and infrastructure services

A full treatment of the Web Services infrastructure standards would require a complete paper on that topic. This section provides some detail on the abstract model that brings together infrastructure services, WS-Policy, endpoint functions, and distinguished services.

The evolving set of Web Services standards (WS-*) that build on WSDL, WS-Policy, and WS-Interoperability/SOAP include WS-ReliableMessaging, WS-Security, and WS-AtomicTransactions. Each specification introduces several optional elements, including the following:

- *Headers*—These augment a message with information for a specification; for example, a WS-ReliableMessaging header identifies a conversa-

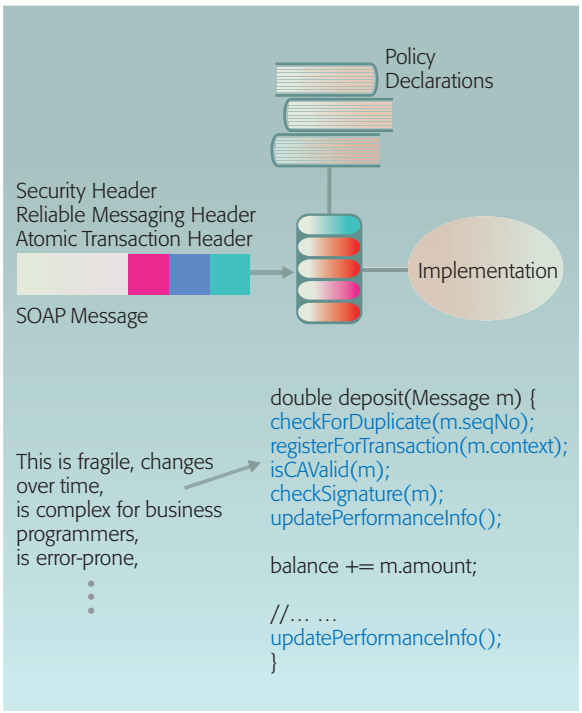

Figure 13
Typical application logic for infrastructure services

tion and sequence number within it, and a WS-Security header contains authentication tokens.

- *Endpoint protocols*—These may define extra operations and specify the endpoint's model for processing them. For example, for reliable messaging, it can specify what action to take on receiving a duplicate or out-of-sequence message. An extra operation could be a request for retransmission, which is "extra," relative to the main business interface of the sending service.
- *Distinguished services*—These are specification-defined Web Services that implement a well-defined interface and semantics. For example, WS-Coordination, WS-BusinessActivity, and WS-AtomicTransactions define the behavior of a coordinator with whom transaction participants interact to begin and end transactions and participate in two-phase commit protocols. WS-Security introduces a Security Token Server (STS) that issues and validates security tokens.
- *WS-Policy extensions*—These are typically schema for specifying expectations and requirements for infrastructure-provided services. A policy document for security, for example, would indicate which STS an endpoint supports, and a document for transactions would specify endpoint support for atomic transactions, business agreements, or both.

Figure 13 illustrates the logic that might be found in a typical business application to check for the accidental double-posting of a bank transaction to a bank account. Some of the housekeeping logic could have been delegated to the middleware but is nevertheless embedded in the application code. Such applications are obviously very common, yet fragile and prone to breakage as changes occur in the computing infrastructure.

The implementor of a simple Web-service solution would have to code numerous functions, including header processing, endpoint protocols and extra operations, and interactions with distinguished servers. This naïve approach has several disadvantages. First, the resulting code is complex and requires detailed, low-level understanding of WS-* specifications. Burdening business-application programmers with this task decreases their productivity and code quality. Second, placing infrastructure code into applications reduces flexibility. Adding or modifying the infrastructure services associated with a service requires modification of the application and retesting and redeploying the service or solution.

A better solution, shown abstractly in *Figure 14*, is for the application to exploit infrastructure services provided by a container. Web services deployed in a container are logically wrapped by a container-provided outer shell. The runtime passes incoming messages to the shell, which invokes container-provided code, depending on what message headers are present and what policies are associated with the service. After header processing, endpoint protocols, and interaction with infrastructure services are completed, the shell passes the message to the service implementation. The business logic only receives business messages vetted by the infrastructure, such as those with a valid security token whose signatures have been checked and which are not duplicates or out of order (as per the WS-ReliableMessaging standard). The left path from the implementation outbound shows the implementation before modification, when it does its own message processing. The right path out of the implementation shows the same implementation after modification. In the latter case, the runtime and shell provide "stubs" that are invoked when the

service implementation in the container wishes to send an outbound call or response to an operation. The runtime intercepts the message to the stub and performs the infrastructure protocols required for message delivery. The container approach vastly simplifies the task of building and deploying robust Web services and increases the flexibility of these services over time.

STATEFUL WEB SERVICES

The Web Services standards are evolving to better model stateful interactions. The literature commonly claims that Web services are stateless, but this is misleading or confusing, as we have already shown that the process type of service component (e.g., BPEL, BSM) explicitly models state. Even the most narrowly defined service manages state data when it accesses relational databases or invokes existing applications. Thus most business applications—and the services comprising them—are inherently stateful.

The WS-Addressing, WS-ResourceFramework and WS-ResourceProperties specifications provide first-class support for stateful Web services. WS-Addressing associates XSD with a portType to make the structures managed by a service explicit. For example, if a portType supports creating a Customer object and adding an address to a Customer, a programmer would infer that customers have addresses. WS-Addressing makes this kind of structure explicit, eliminates guesswork, and provides additional information for looking up and binding to services. WS-ResourceProperties introduces WSDL operations for getting one or more properties, setting one or more properties, and querying properties.

Finally, many programming models—systems management, for example—are inherently stateful or resource-oriented. Management protocols such as Simple Network Management Protocol (SNMP) and Common Management Information Protocol (CMIP) model resources that have properties, operations, and events. WS-Addressing and WS-Resource Framework provide a Web Services abstraction to model existing systems management infrastructure.

SUMMARY AND CONCLUSIONS

This paper is a synopsis of IBM's programming model in support of SOA, which is the fundamental principle guiding our programming model, runtime, systems and application management products, and development tools. Programmers build services and assemble them into modules, applications, and solutions, all of which are services. Our runtime products are increasingly built as a set of components that offer their interfaces through services. The intrinsic systems management capability of software products (for example, the management of WebSphere or operating systems) surfaces through a service abstraction, and the end-to-end management tools are solutions that orchestrate and drive the management capabilities to support autonomics, automation, provisioning, problem determination, and so forth.

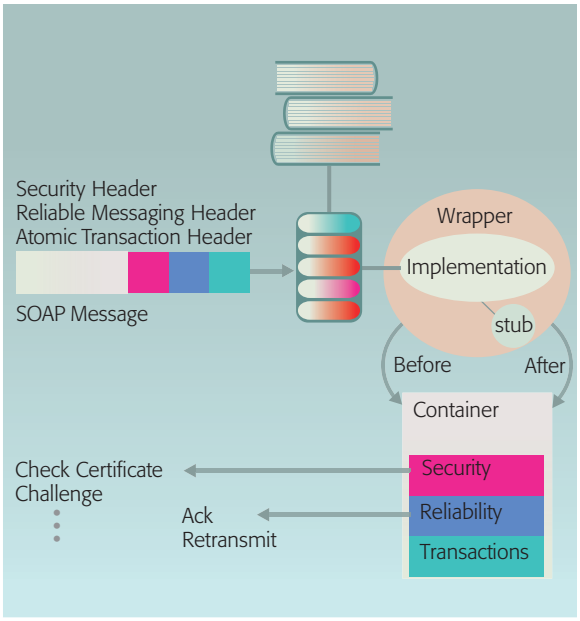

Figure 14
Use of container-provided infrastructure services

The paper has also explored the evolution from an abstract SOA to a pragmatic component model for packaging services, simplifying their implementation, and assembling the components. An SOA describes services and their interfaces. Our programming model defines how to implement services, assemble modules, and build solutions using service components. Supporting tools simplify the building of specific component types. Components are deployed into containers that automate qualities of service, such as security, transactions, and reliable messaging, upon which services rely. Programmers document their quality-of-service expectations and requirements by associating policy

documents with components. This declarative model for quality of services simplifies the development of business services by keeping tedious, error-prone logic out of the business component.

It has become increasingly difficult for any individual programmer, much less a nonprogrammer, to master and apply the alarming proliferation of software technologies, practices, tools, and platforms effectively. Yet if business process transformation is to succeed, a significant number of nonprogrammers will need to use existing IT assets to carry out their duties, and they cannot be expected to learn the excruciating details of the underlying technologies. This paper has described a new SOA programming model that achieves a separation of concerns so that persons with different skill levels and different roles in the enterprise, not necessarily IT professionals, can create and use IT assets throughout every stage of the software development life cycle. The result can be dramatically improved business agility for the on demand enterprise.

CITED REFERENCES AND NOTES

1. *New to SOA and Web Services,* IBM developerWorks, http://www-128.ibm.com/developerworks/webservices/newto/; *Web Services Architecture*, World Wide Web Consortium (W3C) Working Group Note 11 (February 2004), http://w3.org/TR/ws-arch/.
2. N. Bieberstein, S. Bose, M. Fiammante, K. Joones, and R. Shah, *Service-Oriented Architecture Compass—Business Value, Planning and Enterprise Roadmap*, Prentice Hall PTR, 2005.
3. *Business Process Execution Language for Web Services (BPEL4WS) Version 1.1* (February 2005), http://www.ibm.com/developerworks/library/specification/ws-bpel/.
4. *Java Message Service Specification Version 1.0.2*, Sun Microsystems (November 1999), http://docs-pdf.sun.com/816-5904-10/816-5904-10.pdf.
5. *J2EE Connector Architecture Specification Version 1.5*, Sun Microsystems (November 2003), http://java.sun.com/j2ee/connector/download.html.
6. W. Farrell, *Introduction to the J2EE Connector Architecture*, IBM developerWorks (November 2002), http://www-106.ibm.com/developerworks/edu/j-dw-javajca-i.html.
7. *UDDI Version 3.0*, OASIS specification (July 2002), http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.
8. B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, J. Shewchuk, and D. Simon, *Web Services Security (WS-Security)*, (April 2002), http://www.ibm.com/developerworks/webservices/library/ws-secure.
9. *Web Services Transactions Specifications* (November 2004), http://www-128.ibm.com/developerworks/library/specification/ws-tx.
10. L. F. Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, and D. Langworthy, Editor, A. Nadalin, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey, *Web Services Coordination (WS-Coordination)* (November 2004), ftp://www6.software.ibm.com/software/developer/library/ws-coordination200309.pdf.
11. L. F. Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, and D. Langworthy, Editor, A. Nadalin, D. Orchard, I. Robinson, T. Storey, and S. Thatte, *Web Services Atomic Transaction (WS-AtomicTransaction)* (November 2004), ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.
12. L. F. Cabrera, G. Copeland, T. Freund, J. Klein, D. Langworthy, F. Leymann, D. Orchard, I. Robinson, T. Storey, and S. Thatte, *Web Services Business Activity Framework (WS-BusinessActivity)* ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf.
13. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*, Wiley Press, Hoboken, NJ (1996).
14. *Apache Struts Web Application Framework,* Apache Software Foundation, http://struts.apache.org.
15. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*, W3C Working Draft (December 2004), http://www.w3.org/TR/2004/WD-wsdl20-primer-20041221/.
16. K. Ballinger, D. Ehnebuske, C. Ferris, M. Godgin, C. K. Liu, M. Nottingham, and P. Yendluri, *Basic Profile Version 1.1*, Web Services Interoperability Organization (August 2004), http://ws-i.org/Profiles/BasicProfile-1.1.html.
17. J. Liu Y. Lu, *Build Interoperable Web Services with JSR-109*, IBM developerWorks (August 2003), http://www-106.ibm.com/developerworks/library/ws-jsrart/?ca=dnt-431.
18. R. Bilorusets, D. Box, L. F. Cabrera, D. Davis, D. Ferguson, C. Ferris, T. Freund, M. A. Hondo, J. Ibbotson, L. Jin, C. Kaler, D. Langworthy, A. Lewis, R. Limprecht, S. Lucco, D. Mullen, A. Nadalin, M. Nottingham, D. Orchard, J. Roots, S. Samdarshi, J. Shewchuk, and T. Storey, *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*, http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf.
19. *Web Services Policy Framework (WS-Policy)* (September 2004), http://www-106.ibm.com/developerworks/library/specification/ws-polfram/.
20. The Java API for XML-Based Remote Procedure Call (JAX-RPC) is a technology for invoking Web services by using Java classes.
21. K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe, *The WS-Resource Framework Version 1.0* (March 2004), http://devresource.hp.com/drc/specifications/wsrf/WSRF_overview-1-0.pdf.

22. *Web Services Addressing* (March 2004), http://www-106.ibm.com/developerworks/library/specification/ws-add/.

23. *Java 2 Platform Enterprise Edition Specification Version 1.4*, Sun Microsystems (November 2003), http://java.sun.com/j2ee/1.4/download.html#platformspec.

24. R. Hightower, *Enhance J2EE Component Reuse with XDoclets*, IBM developerWorks (May 2003), http://www-106.ibm.com/developerworks/edu/ws-dw-ws-j2x-i.html.

25. S. Kim, *Java Web Start: Developing and Distributing Java Applications for the Client Side*, IBM developerWorks (September 2001), http://www-106.ibm.com/developerworks/java/library/j-webstart/.

26. *Developing the J2C Plugin*, IBM WebSphere Software Information Center (2004), http://publib.boulder.ibm.com/infocenter/adiehelp/index.jsp?topic=/com.ibm.etools.ctc.eab.doc/concepts/cj2cplugn.html.

27. M. Gudgin, M. Hadley, N. Mendelshon, J.-J. Moreau, and H. F. Nielsen, *Simple Object Access Protocol (SOAP) Version 1.2 Part 1: Messaging Framework*, W3C Recommendation (June 2003), http://www.w3.org/TR/soap12-part1/.

28. *Enterprise Generation Language (EGL): An Overview,* IBM Corporation, http://www-128.ibm.com/developerworks/rational/library/04/r-3190/egl_overview2.pdf.

29. *The Role of Enterprise Generation Language (EGL) in a Long History of Innovation on Developer Productivity*, IBM Corporation, http://www-128.ibm.com/developerworks/rational/library/04/r-3190/the_role_of_enterprise_generation_language.pdf

30. *Bean Scripting Framework,* Apache Jakarta Project, http://jakarta.apache.org/bsf/.

31. B. Portier F. Budinsky, *Introduction to Service Data Objects: Next-Generation Data Programming in the Java Environment* (September 2004), http://www-106.ibm.com/developerworks/java/library/j-sdo/.

32. JDBC is a Java interface for executing Structured Query Language (SQL) statements.

33. *Universal Modeling Language 2.0 Superstructure FTF Convenience Document*, Object Management Group (October 2004), http://omg.org/cgi-bin/doc?ptc/2004-10-02.

34. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA (1995).

35. *Web Services Eventing (WS-Eventing)* (August 2004), http://www-128.ibm.com/developerworks/webservices/library/specification/ws-eventing/.

36. *Web Services Notification* (March 2004), http://www-128.ibm.com/developerworks/webservices/library/specification/ws-notification/.

37. IBM and BEA, *BPELJ: BPEL for Java Technology,* www.ibm.com/developerworks/webservices/library/ws-bpelj.

38. M. Linehan, "Enable Dynamic Behavior Changes in Business Performance Management Solutions by Incorporating Business Rules," IBM white paper (December 2004), https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=dw-bpm&S_PKG=dlbrd&S_CMP=DWNL.

39. *XQuery 1.0: An XML Query Language*, W3C working draft (April 2005), http://www.w3.org/TR/xquery/.

40. *Web Services Transactions Specifications* (November 2004), http://www-128.ibm.com/developerworks/library/specification/ws-tx/.

41. *IBM DB2 Information Integrator Application Developer's Guide Version 8.2*, IBM Publication No. SC18-7359-01, http://publibfp.boulder.ibm.com/epubs/pdf/c1873591.pdf.

42. *XML for DB2 Information Integration*, IBM Publication No. SG24-6994-00, http://publib-b.boulder.ibm.com/abstracts/sg246994.html?Open.

43. *DB2 Information Integrator Introduction to Replication and Event Publishing,* IBM Publication No. GC18-7567-00, http://publibfp.boulder.ibm.com/epubs/html/c1875670/c1875670tfrm.htm.

44. *XML Metadata Interchange (XMI Version 2.0)*, Object Management Group (May 2005), http://www.omg.org/technology/documents/formal/xmi.htm.

45. *Project Summary—Database Access and Integration Services Working Group (DAIS-WG),* Global Grid Forum, http://forge.gridforum.org/projects/dais-wg/.

46. *JSR 127:JavaServer Faces*, Java Community Process (May 2004), http://jcp.org/en/jsr/detail?id=127.

47. Dynamic Hypertext Markup Language (dynamic HTML or DHTML) is not a standard defined by the World Wide Web Consortium (W3C) but a combination of technologies used to create dynamic Web sites.

48. *Eclipse Modeling Framework,* eclipse.org, http://www.eclipse.org/emf/.

49. *JSR 168: Portlet Specification*, Java Community Process (October 2003), http://jcp.org/en/jsr/detail?id=168.

50. *Web Services for Remote Portlets (WSRP) Specification Version 1.0*, OASIS (September 2003), http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf.

51. H. Kreger and J. Philips, "Toward Web Services Management Standards—An Architectural Approach to IT Systems Design," *Web Services Journal* **3**, No. 10, pp. 18–22 (October 2003).

52. OASIS Web Services Distributed Management (WSDM) Technical Committee, http://oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm.

53. S. Srivastava, S. Choudhary, and K. Britton, *Enable a Help System within the Integrated Solutions Console*, IBM developerWorks (May 2004), http://www-128.ibm.com/developerworks/autonomic/library/ac-enable/.

54. *IBM Orchestration and Provisioning Automation Library,* http://www.developer.ibm.com/isv/tivoli/workflow.html.

55. *About the IT Infrastructure Library*, Office of Government Commerce http://www.ogc.gov.uk/index.asp?id=1000367.

56. J. Dinger D. Nastacio, *Integrate Event Management with Common Event Infrastructure*, IBM developerWorks (July 2004), http://www-128.ibm.com/developerworks/library-combined/ac-cei/.

57. A. Watkinson, "Using the Common Base Event in the Common Event Infrastructure," *IBM WebSphere Developer Technical Journal* (August 2004), http://www-128.ibm.com/developerworks/websphere/techjournal/0408_watkinson/0408_watkinson.html.

58. M. Schroeck, *CFOs: Rising to the Challenge of Performance Mangement*, IBM Business Consulting Services (2004), http://www-1.ibm.com/services/us/imc/pdf/g510-3618-cfo-performance-management.pdf.

59. IBM WebSphere Application Server Partner Adapters, IBM WebSphere Software, http://www-306.ibm.com/software/webservers/appserv/was/partneradapters/.

Accepted for publication May 25, 2005.
Published online October 20, 2005.

Donald F. Ferguson
IBM Software Group, 294 Route 100, Somers, NY 10589 (dff@us.ibm.com). Dr. Ferguson is one of 55 IBM Fellows, the highest technical position in the IBM engineering community of 200,000 technical professionals. He is the Chief Architect and technical lead for the IBM Software Group (SWG) family of products, which includes Lotus, Rational, WebSphere, DB2, and Tivoli. Dr. Ferguson chairs the SWG Architecture Board, bringing together the SWG's lead product architects. His most recent efforts have focused on Web services, business process management, client platform, outsourcing/hosting platform, grid services, and application development for WebSphere. He was the chief architect for the WebSphere family of products from its inception until assuming the role of SWG Chief Architect in 2003.

Marcia L. Stockton
IBM Software Group, 2827 Poso Flat Road, Bakersfield, CA 93308 (mls@us.ibm.com). Ms. Stockton is a Senior Technical Staff Member and master inventor with the IBM Software Group in Research Triangle Park, North Carolina (residing in California), and a senior member of the Institute of Electrical and Electronics Engineers. She leads the Software Group Architecture Board's Programming Model work group, where she drives horizontal integration initiatives and promotes programming model simplification across Lotus, Rational, WebSphere, DB2, and Tivoli products. Her 73 filed U.S. patents range from networking, Web, security, privacy, multimedia, wireless, and pervasive devices to radio frequency ID. In the 1990s, she was the lead architect for IBM's Advanced Peer to Peer Networking and High Performance Routing network protocols. Ms. Stockton joined IBM in 1988 as a networking software professional. She earned a B.A. degree from Swarthmore College in 1975. ■