A language-neutral, component-based programming model for service-oriented architecture (SOA) facilitates the implementation of Web services and their assembly into solutions. The programming model enables non-programmers to use existing IT assets without mastering intricate technologies. It meets the needs of solution designers and business analysts, providing a higher level of abstraction that conceals differences between implementation technologies, yet enables business accountability.

Why a component-based programming model?

The vision that motivated IBM's SOA programming model rests on two central observations, aptly captured by the following quotations:

"Design-by-buzzword is a common occurrence. At least some of this behavior within the software industry is due to a lack of understanding of why a given set of architectural constraints is useful. In other words, the reasoning behind good software architectures is not apparent to designers when those architectures are selected for reuse."
-- Roy Thomas Fielding, "Architectural Styles and the Design of Network-based Software Architectures" (See Resources for a link to this dissertation.)

This problem can be overcome by embodying the experience of those who have the detailed knowledge of the reasons behind the architectural constraints into a set of patterns, programming artifacts, and tools.

The second key observation pertains to human beings and how they interact with technology:

"Creating service-oriented architecture (SOA) solutions means rethinking the practices currently in use to build systems, the skills in an organization, and the ways in which team members
collaborate. A services orientation contributes to the development of solutions, which are assembled from disparate applications, and SOA is an architectural style that emphasizes loose coupling of independent service providers."
-- A.W. Brown, et. al., "Realizing service-oriented solutions with the IBM software development platform" (See Resources for a link to this paper.)

The Web services standards, based on XML, already suggest aspects of a component-based programming model. Standards such as Web Services Interoperability (WS-I), Web Services Description Language (WSDL), and Web Services Policy (WS-Policy) attempt to create a platform-agnostic abstraction and a universal framework for business software integration. Conversely, the value of Web services is derived from their use within an SOA.

Most literature on Web services focuses on the interoperability protocols and service interfaces and their use. Instead, this article focuses on the programming model for implementing services and assembling them into solutions. A component model simplifies the process of building and assembling services.

A well-defined component model should enable a broad ecosystem of user roles -- such as the business analyst, integration developer, adapter developer, and solution administrator -- to create service-oriented applications by instantiating, using, assembling, and customizing different component types that match the user's goals, skills, and conceptual framework. (Note: Programming still exists as a profession and an important software development role, but not everybody has to be a professional programmer to work productively with SOA artifacts.)

Our component model, founded on the Web Services standards, adds a crisper definition of components and component types, and how components are defined and structured for reuse and for manipulation by role-appropriate tools, in response to our observations about the human side of technology use.

A component-based programming model has many benefits, most noticeably reduced complexity. No single role needs to understand all the possible ways of implementing a function or all of a system's interfaces. The complexity exposed to each role is bounded, and those in different developer roles contribute to the solution in well-defined ways, using tools appropriate to their tasks.

A component model must be abstract and language neutral, because its role is to hide technology details and differences.

It must also facilitate assembly and tailoring of "solution pieces" by non-programmers. Thus, all aspects of a component (or a collection of components) that pertain to assembly and tailoring are explicitly declared in a language-neutral way, so they can be manipulated by tools without a programmer modifying source code. We use XML to express these declarations.

A precise characterization of SOA may still be debatable, yet there is wide acceptance on some key aspects:
1. SOA is a *distributed component* architecture. SOA components are transparently located inside or outside the enterprise and universally accessible as services through a stack of universally supported, interoperable remote procedure call and messaging protocols. Interface definition standards enable interoperability between developer tools. On the wire protocol interoperability -- as opposed to code portability -- is the centerpiece of SOA component interactions, enabling universal access and platform independence. Callers are unaware of a component's underlying implementation technology, such as Java™ 2 Platform, Enterprise Edition (J2EE), Microsoft® .Net, and PHP.

2. SOA components encapsulate functionality and enable reuse at the level of abstraction that is modeled by business analysts and business models. This can improve accountability by mapping more directly between an IT function and the business function it supports.

3. Declarative, machine-processable contracts enable third parties to access the services that an SOA component provides. These contracts explicitly state functional characteristics as well as non-functional (quality-of-service, or QoS) capabilities and requirements. SOA components document their operations using WSDL. Business Process Execution Language for Web Services (BPEL4WS) can also be used to define the valid sequences of operations on a component.

4. SOA components can be dynamically found, selected, and bound by means of their declarative properties, and integrated using composition mechanisms, such as those described in Part 3 in this series, "Process choreography and business state machines" (developerWorks, July 2005).

**Component implementation and specialized component types**

A developer may choose to implement base components using J2EE, PHP, or other tools. SOA as a programming model is more fundamentally concerned with component interactions and their integration into new composite components, applications, and solutions.

Our programming model also introduces well-defined component types that model common kinds of artifacts that developers produce and deploy into solutions. Examples include plain old Java objects (POJOs), business processes (BPEL4WS), Structured Query Language (SQL) services, Adaptive Business Objects, Customer Information Control System (CICS) programs accessed through J2EE Connector (J2C) architecture resource adapters, applications using SAP’s business application programming interface, J2EE stateless session beans, and IBM MQSeries® applications.

Because of the virtual nature of the SOA component model, many SOA components naturally support multiple implementation technologies. On the other hand, different implementation technologies are better suited for different tasks. To improve transparency we introduced the notion of service component types, each suited for a developer with a given set of skills, performing a specific task, and using a certain tool. For queries, the programmer implements a SQL file and a file containing a set of XQuery statements; for document conversion, XSLT style sheets and so forth are implemented using tools optimized for that task. There is no need to know that a Web service, Enterprise JavaBean (EJB), or other artifact is generated upon deployment, just that the overall result will be exposed and made available as a generic SOA component.
Programmers build a specific type of component adapted to the task, concentrating on the problem to be solved and the tool for doing so, not on the resulting artifacts. SOA development tools should focus on the skills of the developer and the concepts he or she understands. A future article will take a brief look at some component types, demonstrating with three very different examples -- Java objects, Information Management System (IMS) transactions, and SQL statements -- how any implementation technologies are mapped to the common SOA component model and, at the same time, address the needs of specific developers.

**Composition**

While SOA composition may be achieved using platform-specific techniques, a new type of SOA-centric composition stands on its own without needing translation into another programming model.

The composition model allows discovery of services having the desired interfaces and infrastructure (QoS) policies and their aggregation into new services, modules, and solutions. These new services can be composed.

Our approach unifies the paradigm for creating and accessing business logic. More abstract than existing programming constructs like EJBs, our SOA programming model hides the differences between implementation technologies. In this model, components are assembled into modules, which may in turn be composed into solutions. Components expose services that can be invoked through addressable interfaces. Interfaces are described using WSDL, Java, or other languages. An implementation can have unresolved references to required services that are satisfied prior to execution by wiring components together. The wiring operation can be done, using a role-appropriate tool, by a solution integrator or solution assembler who can bring to bear knowledge of enterprise policies and Enterprise Service Bus (ESB) deployment topology that might not be known to the person who developed the components initially.

**Customization without programming**

It is unlikely that a service can always be reused as is, without configuration, customization, or tailoring. When change is needed, the current state of the art is source code modification. However, the ability to deliver wide reuse of components depends heavily on the capability to adapt components to the environment in which they are used. An SOA programming model should enable building services and modules that "programmers" can customize without source code modification. This is especially important when the programmer is in a different organization than the programmers who built the components.

The component-based programming model for SOA offers several mechanisms for component customization without programming.

A component that is intended for reuse can be packaged as a template with points of variability that will be tailored when placing it into a solution. This kind of adaptation becomes a first-class part of our programming model with the addition of a rules language and associated tools, offering customization capabilities to new kinds of users.
Mediation focuses on processing in-flight messages. Mediations can often be composed without programming. The enterprise service bus plays a key role as a multi-protocol fabric that weaves service components into a seamless interaction, yet allows enterprise concerns -- such as auditing, logging, routing, adaptation of mismatched interfaces, incremental substitution of equivalent componentry, and security -- to be addressed enterprise wide at the backbone level, by inserting special components called mediations in the path of messages, to broker interactions between services without changing existing endpoints.

Another benefit of an SOA programming model, fostered by the previously mentioned traits, is the ability to substitute one component for another at various times during the software life cycle. This is enabled by the late binding of declared interfaces to implementations supporting those interfaces. There are many business reasons why substituting units of functionality is desirable. Most important of these, perhaps, is to reduce the difficulty of managing change in a large enterprise. Introducing change incrementally and limiting its impact by adhering to defined interfaces confers increased flexibility. It also matches the loose coupling that is often characteristic of large human organizations. Furthermore, service components benefit the organization by enabling groups with different skills, needs, and timetables to work collaboratively on the IT infrastructure in a way that maximizes the efficiency of both human and system resources, allowing the business to respond more rapidly to change at the business level.

Component definition

Our SOA components are defined by the following specifications:

1. A service specification provides the view of the component as a set of services provided and required by the component. It's defined by the following three groups of specifications:
   - Interfaces, which are typically WSDL portTypes.
   - Policies that document QoS properties, such as transactional behavior and security.
   - Behavioral descriptions, such as a BPEL4WS abstract process. Another example might be a Unified Modeling Language, Version 2 (UML2) state model that specifies which operations are valid for different states and state transitions caused by operations. Callers can compute valid sequences of operations from the state model.

2. A service component implementation is defined by the following four groups of specifications:
   - Provided service specifications.
   - Required service specifications.
   - Properties that may be set on the component to tailor or customize its behavior.
   - Properties that provide basic support for this; more complex scenarios use points of variability and outcalls to a customizing component.
   - Container directives (policies) that are invariant for all instances of the implementation.
   - An implementation artifact (such as a Java class, BPEL document, or set of XSLT rules) that defines the implementation of the component.

3. A service component (instance) is defined by the following:
   - A name.
   - A service component implementation.
   - The values of any properties of the implementation that are being set to tailor the instance.
• The specification of any services that resolve the required service specifications of the implementation. These may be “wires” that connect component instances or a “query” that executes to find a component at run time that implements the interface, has the associated QoS policies, and matches the specified behavior (such as an abstract process or state model).

There are two basic approaches to defining an SOA component. The definitions may be generated by development tools or created manually by a developer.

The first is a **control file**, a document that, by reference, associates or joins all the parts of the component. For example, the control file may reference the WSDL definition (interface provided), the Java class that implements the component (implementation artifact), or the associated policy documents (policy assertions). These can be references to the file system, class path, source code control system, or Web URLs. The control file format gathers several individually developed artifacts into a collection that, together, comprises the component. Application development tools aid in defining the control file.

The second format uses **pragmas**, language elements specifying the same information, but contained within the body of a single source file. There is evolving support in Java (for example, XDoclet tags as in JSR 175) to make these annotations part of the Java language. But this approach doesn't yet support other equally valid SOA component implementation technologies, such as a set of SQL or XQuery statements. Each component type has an associated source file format for its implementation artifact, such as a Java file, state machine, or SQL file. The annotations support in IBM WebSphere® Rapid Deployment can generate all the individual elements that comprise a component from a source file containing pragmas. For example, structured comments in a Java source file indicate which Java methods will become Web service operations on the generated WSDL defining the component's service interfaces.

**Summary**

A component-based programming model -- supported by task-oriented tools and a runtime infrastructure -- is key to the rapid adoption of SOA. Expect benefits like a new twist on software reuse that enables business professionals who are not necessarily programmers to use and exploit SOA components to create new business solutions and adapt existing ones as new business needs arise.

© Copyright IBM Corporation 2005
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)