The IBM® programming model for Service-Oriented Architecture (SOA) enables non-programmers to create and reuse IT assets without mastering IT skills. The model includes component types, wiring, templates, application adapters, uniform data representation, and an Enterprise Service Bus (ESB). This is the first in a series of articles about the IBM SOA programming model and what is required to select, develop, deploy, and recommend programming model elements. The content presented here takes into account that developers come to this model with different skill levels and roles.

The SOA programming model series

It has become increasingly difficult for any individual programmer, much less a non-programmer, 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 non-programmers need to use existing IT assets to carry out their duties and cannot expect to learn the excruciating details of the underlying technologies. This article series describes a new Service-Oriented Architecture (SOA) programming model that achieves a separation of concerns so that persons with different skill levels and 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.

Introduction

IBM products increasingly implement a SOA and programming model. Programmers build services, use services, and develop solutions that aggregate services. We use the term "programmer" loosely here because a key aspect of the SOA programming model is the expansion of "programming" to a broader set of non-traditional developer roles and skills, such as business analysts and scripting language users.
Most of the literature on Web services focuses mainly on service interfaces and their use. To complement interface standards and best practices, IBM introduces a programming model for implementing services and assembling them into solutions. Extending the IBM software platform's reach to a broader community of users -- including non-traditional programmers -- this model offers new component types that match the user's role, goals, skills, and conceptual framework. These component types enable more intuitive development tools. Another major theme is consumability through progressive disclosure of the programming model's features and capabilities.

This is the first of a series of articles about the IBM SOA programming model that specifically addresses software development professionals. In this series, we present several new programming model elements that address these goals. We show you how you can take advantage of them to make the software you select, develop, deploy, recommend, or administer easier to develop, reuse, and consume. Software that is structured as services is especially valuable to the on demand enterprise because it can be "wired" into a solution by a less skilled developer, or orchestrated into a business process choreography flow to meet rapidly changing business needs. Whether you're a developer at a large enterprise or a small business, an independent software vendor (ISV), an applications provider, or a middleware vendor, you can benefit from structuring your software this way. So, let's start applying SOA principles right now.

**Highlights of the SOA programming model**

Let's start by highlighting some of the major features of the SOA programming model.

*Service Data Objects* (SDOs) are a fundamental concept in the IBM SOA. SDOs make developers more productive and free you from the technical details of how to access particular back-end data sources, applications, or services. They provide a simplifying abstraction that allows programmers to concentrate principally on business logic. SDOs also provide a uniform representation for messages that interact with services, replace the confusing labyrinth of technologies for data representation, and access with a single uniform model.

The SOA programming model also needs a unified paradigm for creating and accessing business logic. For easy consumability, a service should hide the differences between implementation technologies and be at a higher level of abstraction than existing programming constructs, such as Enterprise Java™Beans (EJBs). Consider that services can be implemented by components that are assembled into modules, which might in turn be composed into solutions. Components expose services that can be invoked using addressable interfaces. You can use Web Services Description Language (WSDL), Java, or other languages to describe interfaces. This implementation style can have unresolved references to required services that will be satisfied prior to execution by wiring components together.

This 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", Business Process Execution Language (BPEL) processes, structured Query Language (SQL) services, Adaptive Business Objects, CICS® programs accessed through Java Connector Architecture (J2C) resource adapters, applications using SAP's business application programming interface, Java 2 Enterprise Edition (J2EE) stateless session beans, and MQSeries® applications.
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, security, and so forth -- 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.

New process languages reduce the gap between IT concepts and business artifacts. A key one is *BPEL*. Although a process can be defined using graphical tools that business analysts can relate to, it's also an executable program. Processes are coming to play an important role in on demand business transformation, for example in describing an executable long-running process for an extended value chain. By extended value chain, we mean a business arrangement that can span multiple suppliers and IT domains, such as a retailer and its many individual suppliers, an insurance company and its numerous third party adjusters, an IT outsourcing situation, and so forth.

A *business state machine* is another programming metaphor that a business analyst can create with graphical tools, yet executes in a process choreography engine. A state machine can represent a business artifact -- such as a purchase order, an insurance claim, and so forth -- that transitions through several well-defined states in response to specific life cycle "events".

A component that is intended for reuse can be packaged as a template with *points of variability* that are intended to 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.

Another area of innovation is a new *solution model* that lets deployers, administrators, and other business users assemble components into solutions. At development time, you can associate a service implementation with the topology that hosts it (the deployment topology that the system architect models). System requirements and environment assumptions captured by the model are verified against the implementation early, reducing application life cycle costs while greatly improving reliability and accountability. Late bindings, mediations to transform data, and adapters for existing applications allowing service-oriented interactions through an Enterprise Service Bus, are features of this model.

In summary, the SOA programming model separates development and deployment activities into separate phases that can occur at different times and can be done using different skills, by different individuals. This yields a true separation of concerns, enabling the repurposing of software components. It also tailors the software experience to an individual user's business role, skill, and task. Finally, it supports a software life cycle to fit the needs of on demand enterprises as they seek increased profitability by reengineering IT processes for business agility.

**The programming model concept**

A *programming model* is central to IBM SOA and IBM products in general. It defines the concepts and abstractions that developers build and use. Runtime products, such as WebSphere®
Application Server, DB2®, and CICS, 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.

What is a programming model? Although there is no generally accepted definition, we like to define it as:

- A set of **part types** that programmers build. Part types encompass the diversity of programming model artifacts: Hypertext Markup Language (HTML) files, database stored procedures, Java classes, Extensible Markup Language (XML) Schema definitions, C structs defining MQSeries messages, and so forth.
- A set of **roles** that groups members of the development and administrative community who have similar skills and knowledge. Categorizing developers in this way helps produce role-appropriate tools that enable non-programmers to implement services and assemble solutions from services. A business analyst defining business processes and a marketing specialist defining policies that classify customers and compute product discounts illustrate new kinds of developers that you want to reach. Each role contains:
  - **Skills** that the role possesses. For example, a user interface developer develops interfaces that present 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 sufficiently, to be expert in several user interface design methods, and to be able to create easy-to-use user interfaces by choosing the right kind for each task.
  - **Part types and application interfaces** with which the role interacts (consumes or produces). For example, designers of dynamic pages -- a role -- produce JavaServer Pages (JSPs) and consume EJBs -- part types -- that wrap existing sources of information and applications.
  - **Tools** that the role uses. An example of 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 EJBs.

The key to making Web services easy to implement and use is incremental extension of a person's existing skills and knowledge, thus making the SOA consumable. A service in the form of CICS COBOL transaction programs bears little resemblance to one written in BPEL. Calling a service from a database stored procedure differs from calling it from a JSP; the skills and expectations are different. You achieve consumability by offering an assortment of tools to adapt the part types to various skills, and to the stages of the development process.

Further articles in this series explore some of the part types comprising the SOA programming model in more detail.
Product architecture

Figure 1. Product architecture

Products supporting the IBM view of a SOA fall into two broad categories: service endpoints and the message transport fabric interconnecting them. This general architecture -- populated by many products, none of which individually is the sole delivery vehicle for the IBM SOA -- is illustrated in Figure 1.

At the core is an ESB supplying connectivity among services. The ESB is multi-protocol, supports point-to-point and publish-subscribe styles of communication, and mediation services that process messages in flight. IBM WebSphere MQ, IBM WebSphere MQ Integrator Broker, and WebSphere support for Web services and Java Message Services (JMS) 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 implementation code of the service, 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 the major IBM SOA hosting environments and the kinds of components hosted.

Table 1. Containers hosting various component and service types

<table>
<thead>
<tr>
<th>Service/component type</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction programs written in COBOL, PL/1, and other languages</td>
<td>CICS or IMS (Information Management System -- an enterprise transaction processing system). Programmers can use SOAP/HTTP, WebSphere MQ, and J2EE J2C connections to access the services.</td>
</tr>
<tr>
<td>Business process choreography</td>
<td>WebSphere Business Integration Server Foundation. This container supports long-lived workflow processes that implement Web service interfaces and invoke operations on other Web services. It also supports long-running business activity transactions.</td>
</tr>
<tr>
<td>Application adapters -- providing a SOA/Web service facade for existing applications and systems</td>
<td>Application adapter container provided by WebSphere Business Integration Server Foundation. An adapter converts between SOA protocols and formats, and 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...</td>
</tr>
<tr>
<td>Services implemented by pre-defined SQL queries, XML queries, or as database stored procedures</td>
<td>application programming interface formats and Remote Function Call (RFC). DB2 in conjunction with WebSphere Application Server. Parameters for the query come from a SOA operation's input message and the result provides the output message.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Services implemented using Java classes and EJBs</td>
<td>WebSphere Application Server.</td>
</tr>
</tbody>
</table>

**Summary**

This first part of our series on the IBM SOA programming model provided an overview of how IBM tools and products fit with the model and how developers can effectively use it in their application development.

Future topics in this series will include:

- Simplified data access using SDOs
- Definition of a service and an introduction to the evolving component model
- Component types to simplify development
- Basic component types
- Service composition and customization
- Process components: BPEL and business state machines
- Customizing services: design patterns, templates, and points of variability
- Service-oriented user interfaces
- SOA approach to management
- Development tools for the SOA software life cycle
- Security in the SOA
Related topics

- You'll want to check out the **Java Message Service specification** in its various iterations.
- The **J2EE Connector architecture** provides a Java technology solution to the problem of connectivity between the many application servers and today's enterprise information systems.
- IBM developerWorks has a tutorial, **Introduction to the J2EE Connector Architecture**, which gives a great overview of the J2EE Connector architecture.