Editor: Siobhán Clarke • siobhan.clarke@cs.tcd.ie

How BPEL and SOA Are Changing Web Services Development

James Pasley • *Cape Clear Software*

As the use of Web services grows, organizations are increasingly choosing the Business Process Execution Language for modeling business processes within the Web services architecture. In addition to orchestrating organizations' Web services, BPEL's strengths include asynchronous message handling, reliability, and recovery. By developing Web services with BPEL in mind, organizations can implement aspects of the service-oriented architecture that might previously have been difficult to achieve.

Every organization faces the challenge of integrating diverse IT systems. Developers must first solve communication-level integration issues, ensuring that systems using different transport protocols and data formats can exchange information. Once these issues are resolved, organizations must decide how their various IT systems can interact to support business processes. Business process modeling (BPM) environments seek to solve these issues. Historically, however, these systems have been proprietary, locking an organization — and sometimes its business partners — into a single product. BPMs also have had limited interoperability with various IT systems, creating further integration problems.

Increasingly, developers are using the Business Process Execution Language (ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf) for modeling business processes within the Web services architecture.[1] BPEL is an XML-based standard for defining business process flows. In addition to facilitating the orchestration of synchronous (client-server) and asynchronous (peer-to-peer) Web services, BPEL provides specific support for long-running and stateful processes. BPEL is an open standard, making it interoperable and portable across many environments. BPEL is ideally suited to the service-oriented architecture, a set of guidelines for integrating disparate systems by presenting each system as a service that implements a specific business function. BPEL provides an ideal way to orchestrate services within SOA into complete business processes.

Here, I describe how BPEL fits into the Web services stack, some of BPEL's key benefits, and how targeting Web services for use with BPEL makes the creation of an SOA easier than ever. To illustrate some of the SOA principles and how BPEL affects Web service design, I'll use a sample integration project in which a phone company wants to automate its sign-up process for new customers. This process involves four separate systems based on different technologies:

- *Payment gateway*: a third-party system that handles credit-card transactions and is already exposed as a Web service.
- *Billing system*: hosted on a mainframe, this system uses a Java Message Service (JMS) queuing system for communication.
- *Customer-relationship management (CRM) system*: a packaged off-the-shelf application.
- *Network administration system*: a packaged off-the-shelf application implemented in Corba.

 Published by the IEEE Computer Society 1089-7801/05/$20.00 © 2005 IEEE

Uniting these systems into a single business process involves several tasks. First, developers must solve various integration issues by exposing each system as a Web service. They can then use BPEL to combine the services into a single business process.

Solving the Integration Issues

SOA advocates a general approach to integrating diverse systems. I outline that approach here, highlighting details pertinent to our four sample systems.

Service-Oriented Architecture

SOA advocates that developers create distributed software systems whose functionality is provided entirely by services. SOA services

- can be invoked remotely,
- have well-defined interfaces described in an implementation-independent manner, and
- are self-contained (each service's task is specific and reusable in isolation from other services).

Service interoperability is paramount. Although researchers have proposed various middleware technologies to achieve SOA, Web services standards better satisfy the universal interoperability needs. Services will be invoked using SOAP typically over HTTP and will have interfaces described by the Web Services Description Language (WSDL).[2]

By using SOA and ensuring that each of the four systems complies with SOA's service definitions, our phone company's development team can solve the integration problem. Each system already complies with some definitions. The billing system, for example, is an asynchronous message-based system that performs specific business functions based on particular messages sent to it. However, the message formats are not defined in a machine-readable form. The network administration system is Corba-based, so its interface is defined using IDL, but the system is based on an object-oriented, rather than a message-based, approach. To proceed with integration, the company needs a system to fill these gaps and raise each system to SOA standards.

Enterprise Service Bus

The enterprise service bus is a new middleware technology that provides SOA-required features. Within the IT industry, it's generally accepted that developers use an ESB to implement applications such as those described in our sample project

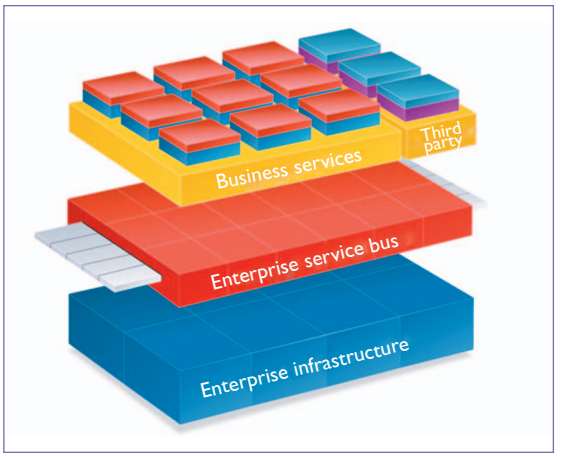

Figure 1. The ESB architecture. The architecture consists of three layers: the existing enterprise infrastructure; the ESB layer, which includes adapters to expose existing systems and provide transport connectivity; and the business services created from the existing IT systems.

(www.gartner.com/regionalization/img/gpress/pdf/gartner_exec_report_sample_WEB.pdf). An ESB provides a hosting environment for Web services, whether they're new and entirely ESB-hosted or Web service front-ends to existing legacy systems. An ESB connects IT resources over various transports and ensures that services are exposed over standards-based transports (such as HTTP) so that any Web-service-aware client can contact them directly. The ESB also provides other features that are essential to services deployment, including enterprise management services, message validation and transformation, security, and a service registry.

In addition to the runtime environment, an ESB must also provide a development environment with tools for creating Web services. Because reusing — rather than replacing — existing systems is fundamental to the ESB concept, these tools should include wizards to automatically create Web services from other technologies such as Corba or Enterprise JavaBeans.

As Figure 1 shows, the resulting ESB architecture consists of three layers. The lowest is the existing enterprise infrastructure, which includes the IT systems that provide much of the functionality to be exposed as Web services. The ESB sits on top of this layer and contains adapters to expose the existing IT systems and provide connectivity to various transports. The top layer consists of business services created from existing IT systems. These services provide essentially the same func-

tionality as the existing systems, but they're exposed as secure and reliable Web services that the organization or its business partners can reuse.

So, how does the ESB architecture support our example integration project? The payment gateway is already implemented as a Web service and requires no further development. Yet, the ESB is useful nonetheless: in addition to handling security and reliable messaging requirements, it offers a single management view of the service.

To expose the remaining systems as Web services requires additional work. The ESB's transport-switching capability lets clients access the services through HTTP (or other transports) and forwards client requests to the billing system via JMS. Project developers can define new message formats using XML Schema[3] and create transformation rules to convert to the existing application's format. The result is a new ESB-hosted Web service that receives requests and transforms them before placing them in the JMS queue.

Next, the development team can use an ESB adapter to expose the CRM application as a Web service. (Because application vendors are increasingly adding Web service interfaces to their applications, the need for such adapters is decreasing.)

Finally, the development team must address the Corba-based network administration system. Although the system's interface is defined in interface definition language (IDL), it is also fine-grained and uses many different objects. The team can use an ESB wizard to automatically create a Web service from the interface description. To create a more course-grained interface, the team members have two primary options. They can define a new interface in IDL, let developers familiar with the Corba system implement it, and then expose it using ESB wizards. Alternatively, they can design the new interface in WSDL and create the Web service from there. The service implementation can act as a client of the Corba system directly or through an ESB-generated Web-service interface. The best option here depends on several criteria, including the developers' skill set.

BPEL Features

BPM introduces a fourth layer to the ESB architecture. Using an SOA, all of an organization's IT systems can be viewed as services providing particular business functions. Because the ESB resolves integration issues, BPEL can orchestrate these individual tasks into business processes.

BPEL expresses a business process's event sequence and collaboration logic, whereas the underlying Web services provide the process functionality. To gain the most from BPEL, developers must understand the dividing line between the logic implemented in the BPEL processes and the functionality that Web services provide.

BPEL has several core features. Actions are performed through *activities*, such as invoking a Web service or assigning a new value in an XML document. Activities such as `while` or `switch` offer the developer control over activity execution. Because it was designed to implement only the collaboration logic, BPEL offers only basic activities.

BPEL describes communication with partners using *partner links*, and messages exchanged by partners are defined using WSDL. Web services operate using client-server or peer-to-peer communications. In client-server communication, the client must initiate all invocations on the server, whereas in peer-to-peer communication, partners can make invocations on each other. BPEL extends WSDL with partner link definitions to indicate whether client-server or peer-to-peer communication will be used. In peer-to-peer communication, each partner uses WSDL to define its Web service interfaces; partner links define each partner's role and the interfaces they must implement (WSDL 1.1 alone can't do this satisfactorily).

BPEL supports *asynchronous message exchanges* and gives the developer great flexibility regarding when messages are sent or received. It also gives the developer full control over when incoming messages are processed. Using event handlers, BPEL processes can handle multiple incoming messages as they occur. Alternatively, they can use the `receive` activity to ensure that particular messages are processed only once the business process reaches a given state. These process instances can persist over extended periods of inactivity. A BPEL engine stores such instances in a database, freeing up resources and ensuring scalability.

BPEL provides *fault handlers* to deal with faults that occur either within processes or in external Web services. Developers can also use *compensation handlers* to undo any previous actions, which gives them an alternative approach to providing a two-phase commit based on distributed transaction support. When a business process instance extends over a long period or crosses organizational boundaries, it's impractical to have transactions waiting to commit. The compensation handler approach is more appropriate in this scenario.

Commercial BPEL engines provide *management consoles* that let operators monitor business process states, including processed messages and executed activities. This lets operators see inside the running BPEL processes to a far greater extent than is possible with other technologies. Such tool support, which is easily built around BPEL, is an important benefit to using the language.

Because it is an open standard, developers can use BPEL scripts in different environments and exchange them between organizations. They can use these scripts to provide additional details on message interactions beyond that offered by WSDL, including descriptions of business-process life cycles (the message-exchange order).

In addition, tools can extract correlation information from within BPEL scripts to correlate messages to particular business transactions. For example, a management console could use such information to identify messages of interest to the operator. Developers can also provide sample executable BPEL scripts to show partners how to use their Web services. Business partners can load these examples into their own environments and customize them for their own use.

Impact on Web Service Development

Using Web services to expose applications over the Internet is now a widely accepted practice. Developers typically create Web services individually and expose them either directly over the Internet or within an organization for particular purposes.

Typically, these Web services are implemented in Java and invoked by Java clients. It's thus quick and easy to generate Web service descriptions (in WSDL) from APIs defined in code, and subsequently generate client-side proxies from the WSDL. Similar situations exist for other programming languages such as C#. The disadvantage of this approach is that the Web services tend to reflect the environments in which they're developed. This is why many services are based on synchronous remote procedure calls (RPCs).

BPEL provides a native XML scripting environment that is ideally suited to asynchronous document processing. Designing Web services for use within BPEL, however, requires more thought than the generate-WSDL-from-Java approach.

BPEL-Ready WSDL

When creating services for use within BPEL, it's helpful to use BPEL's WSDL extensions:

```
    <plnk:partnerLinkType
name="PaymentGatewayLinkType">
       <plnk:role name="serviceProvider">
          <plnk:portType
name="tns:PaymentGateway"/>
       </plnk:role>
    </plnk:partnerLinkType>
       <bpws:property name="CardNumber"
type="xsd:string"/>
       <bpws:propertyAlias
messageType="tns:authenticate" part="request"
             propertyName="tns:CardNumber"
query="/xsd1:CreditCardDetails/CardNumber"/>
```

Figure 2. Example of BPEL's WSDL extensions. `PartnerLinkType` *defines business roles and ties them to specific* `portTypes` *(interfaces). The* `CardNumber` *property shows how to extract that value from the* `authenticate` *message.*

- add partner link definitions, especially for peer-to-peer communication, and
- define properties and property aliases for important message and correlation values.

The third-party developers providing the phone company's payment gateway, for example, might not be using BPEL themselves. Yet, realizing that the purpose of providing a Web services interface is to simplify integration and reuse, they might add the BPEL extensions to their WSDL and follow the principles described below.

Figure 2 shows the BPEL constructs that developers can add to WSDL files. The `partnerLinkType` defines the roles involved in the business relationship and ties them to given `portTypes` (the WSDL term for interfaces). In this example, the presence of a single role implies a client-server relationship.

The figure also shows the `CardNumber` property definition and a property alias that describes how to extract that property's value from the `authenticate` message.

The WSDL-First Approach

Two factors are motivating Web services' move from RPC to a more document-based approach:

- SOA's requirements for loose coupling of services, and
- the rise of standards that define common document formats.

Transitioning from RPC to a document approach

```
<xsd:element name="CardNumber">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{16}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Figure 3. Validation within the XML Schema. The data-type definition for credit-card details states that the card number is a 16-digit string.

```
<plnk:partnerLinkType
   name="BillingSystemLinkType">
    <plnk:role name="ServiceProvider">
        <plnk:portType
           name="tns:BillingSystem"/>
    </plnk:role>
    <plnk:role name="Client">
        <plnk:portType
           name="tns:BillingSystemCallback"/>
    </plnk:role>
</plnk:partnerLinkType>
```

Figure 4. A peer-to-peer `partnerLinkType`*. The* `ServiceProvider` *and* `Client` *roles describe communication between the client and billing system.*

requires a move away from Java-based WSDL generation. Instead, developers should describe document formats with XML schemas and then incorporate them into WSDL. The proper use of XML schemas will let developers take full advantage of the standard's features, including validation and extensible document descriptions. When developing the Web services to process these documents, developers can generate Java code from the WSDL, which lets them easily manipulate the XML. Alternatively, they can program against XML APIs, such as the Java binding for the Document Object Model (www.w3.org/DOM).

This document-based approach can also increase service granularity, so that services perform specific business functions on documents and thus fit more naturally into business process models. This is a significant improvement over situations in which several RPC-style operations must be invoked to achieve a single task. Using such RPC-style operations results in a business process model that is both less clear and less manageable.

The WSDL-first approach also permits the use of a common set of data types or documents across all Web services. This was less an issue when Web services were deployed and used individually. As multiple services increasingly act together, however, orchestrating them is much easier if they share a set of data types. Developers can resolve data-structure mismatches using mapping technologies such as Extensible Stylesheet Language transformations (XSLT).[4] However, a business process that has to handle several different data types and continually map between them will be complex and less clear to the operators.

Adopting a WSDL-first approach requires more familiarity with XML Schema than the code-first approach, but the benefits are well worth the investment. For our phone company, using this approach means that, rather than expose each system independently on the ESB, the developers would first analyze the systems' data types. Next, they'd create an XML schema defining a new XML format for the shared data types. Each system reuses this XML schema by creating a WSDL description to define the operations it performs on these data types. The format this shared schema defines will be the starting point for creating transforms to convert from XML to the billing system's format. When exposing the Corba network-administration system, the developers create a WSDL file defining the operations performed on the shared data types. They then use this to create a Java Web service skeleton using ESB tools and implement each operation by invoking the existing Corba system.

The WSDL-first approach also lets developers add validation rules that wouldn't normally be present in WSDL generated from programming languages. As Figure 3 shows, for example, XML schema's data-type definition for credit-card details might state that the card number is a 16-digit string.

Defining data types in this way gives business partners additional details on how to use services and lets the ESB automatically validate messages before they're delivered to the applications.

Peer-to-Peer Communication

Until recently, most Web services used client-server communications because organizations generally thought peer-to-peer communication made service access too complex. This reliance on client-server communications has prevented organizations from creating truly asynchronous or event-based systems. As a result, problems can arise in many

areas, including scalability, because connections must remain open during message processing, or when clients need to poll the server to check for results availability.

BPEL lets organizations offer peer-to-peer services, knowing that clients and partners using BPEL can easily make use of them. For instance, the billing system is an asynchronous messaging system that is now exposed as a Web service. The WSDL describing this Web service contains two `portTypes`: one contains the operations that the client invokes, and the other contains the operations that the client must provide so the billing system can send it results. As Figure 4 shows, the partner link definition describes this relationship by defining two roles: `ServiceProvider` and `Client`.

The service provided by the BPEL script must appear synchronous: the client will wait for the response to indicate that the account has been created. The billing system, however, is asynchronous and might not respond quickly enough. To address this, the BPEL script sends the request to the billing system, but doesn't wait for the reply. Instead, it completes its other work and then sends the reply message back to the client. It then waits for the billing system's response and updates the CRM system when it arrives. Figure 5 shows this event sequence.

The `createAccount` operation is a one-way operation, so it can be invoked asynchronously. After the BPEL script replies to the customer, a receive activity makes it wait for the billing system's response message. Once it receives the response, the script updates the CRM system and thus completes the process.

```
<bpws:invoke partnerLink="BillingSystem"
   portType="ns3:BillingSystem"
  operation="createAccount"/>

<!– Some activities skipped –>

<bpws:reply partnerLink="customer"
   portType="ns:ProvisioningInterface"
  operation="provisionNewSubscriber"/>

<bpws:receive partnerLink="BillingSystem"
   portType="ns3:BillingCallback"
  operation="accountConfirmation"/>

<bpws:invoke partnerLink="CRMSystem"
   portType="ns4:CRMSystem"
  operation="registerBillingAccount"/>
```

Figure 5. Communicating with the billing system. Rather than wait for the billing system's response, the BPEL script completes its work, sends the reply message to the client, and then updates the CRM once the billing system's response arrives.

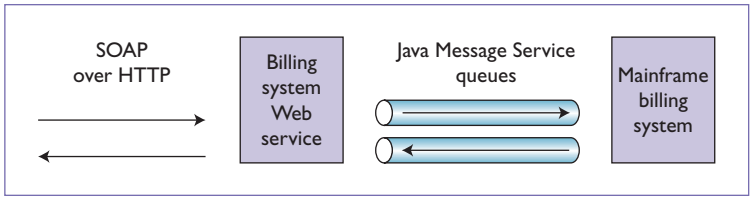

Figure 6. The billing system Web service. The Java Message Service queues are exposed as a Web service and perform data transformation.

State Storage

Traditionally, the Web services community has advocated making all Web services stateless. This is good advice for simplifying service development and deployment. However, some business processes need to store state. The BPEL specification acknowledges this, and compliant BPEL engines must automatically store process-associated state to a database. The BPEL engine ensures that all database access occurs within the appropriate transaction and that system failures will never cause inconsistency in a process's internal state. When failures occur, the BPEL engine provides automatic recovery, removing a significant burden from Web service developers.

Using BPEL, Web service developers can ensure stateless Web services, letting the BPEL process store any state a process requires. A Web service's tasks can thus be focused on the documents it receives. If a service performs many tasks — some of which depend on previously stored state — developers should consider returning this state to the BPEL process for storage with the document. BPEL can then pass it to the Web service again as part of the document when necessary. Reducing the number of services that need to store state offers several benefits in the overall architecture's reliability and flexibility.

In our phone company scenario, the service that exposes the billing system is such an example (see Figure 6). This service is essentially two JMS queues exposed as a Web service that performs data transformation. Requests are received via SOAP over HTTP, transformed, and passed on to a JMS queue. The service receives billing sys-

```
<bpws:correlationSets>
  <bpws:correlationSet name="instanceID" properties="ns:identifier"/>
</bpws:correlationSets>

<bpws:invoke partnerLink="MainframeBillingSystem" portType="ns3:SendQueue"
operation="sendMessage">
  <bpws:correlations>
    <bpws:correlation set="instanceID" initiate="yes"/>
  </bpws:correlations>
</bpws:invoke>

<bpws:receive partnerLink="MainframeBillingSystem" portType="ns3:ReceiveQueue"
operation="receiveMessage">
  <bpws:correlations>
    <bpws:correlation set="instanceID" initiate="no"/>
  </bpws:correlations>
</bpws:invoke>
```

Figure 7. Declaration of a correlation set. Using correlation sets, the BPEL script lets developers use WSDL-defined properties to define correlation sets and reference them during billing system communications.

```
<bpws:scope name="CreateAccounts">
  <bpws:sequence>
    <bpws:scope name="BillingSystem">

      <bpws:faultHandlers>
        <bpws:catchAll>
              <!– Report error –>
        </bpws:catchAll>
      </bpws:faultHandlers>

      <bpws:compensationHandler>
        <!– Delete account –>
        <bpws:invoke partnerLink="BillingSystem"
           operation="deleteAccount"/>
      </bpws:compensationHandler>

      <bpws:invoke partnerLink="BillingSystem"
         operation="createAccount"/>

    </bpws:scope>

     <bpws:invoke partnerLink="CRMSystem"
        operation="createCustomer"/>

  </bpws:sequence>
</bpws:scope>
```

Figure 8. Implementing a compensation handler. As this abbreviated example indicates, if a fault occurs within the CRM system the billing system operation is also reversed.

tem responses from the reply queue and forwards them to the client via SOAP over HTTP. Although the Web service client specifies a reply-to address, the mainframe system doesn't use this addressing mechanism. As a result, the address must be stored so that the Web service can pass the reply to the correct address. So, in addition to hosting the transformations, this Web service must perform two other tasks: correlate the request and response messages, and store client callback addresses. This is an ideal service for BPEL implementation.

Within the billing system, messages contain an identifier used to correlate request and response messages. This identifier is defined as a property within WSDL. Another set of WSDL definitions, `propertyAliases`, shows how that property can be found in each message. Whenever state is stored within BPEL process instances, correlation sets must be used to ensure that subsequent messages are routed to the correct process instance. As Figure 7 illustrates, developers can use this property to define a correlation set and reference it when communicating with the billing system.

When the message is sent to the billing system, the `invoke` operation initializes the correlation set. The same correlation set is then referenced within the `receive` activity. This acts as a filter to ensure that only a message with a value that matches the correlation property will be passed to this instance of the BPEL process.

When Things Go Wrong

Any system that modifies resources must be able to roll back changes when faults occur. Web developers should consider the faults that a service might return and ask: "What action do I expect the client to take as a result of receiving this fault?" If clients might need to reverse previous actions, developers should provide operations on the Web service API accordingly.

BPEL's compensation mechanism is quite sophisticated, and lets developers associate a compensation handler with any activities set. This gives them a lot of flexibility in how they provide operations to undo previous actions. However, developers should ensure that all fault- and compensation-handler operations are easy to use. Otherwise, the code these handlers execute might be as complex (or more complex) than that for normal operation.

BPEL executes a fault handler when a fault occurs and terminates the fault handler's associated scope. In contrast, a compensation handler can be executed only when its associated scope has successfully completed. The BPEL engine stores the required state associated with the completed scope so that the compensation handler can execute. In our sample process, an account is created in both the billing system and the CRM system. If a fault occurs with one system, the operation on the other should be reversed. Figure 8 shows an abbreviated example of the BPEL script that implements this logic.

If a fault occurs in the `createAccount` operation, BPEL invokes the report-error fault handler. If the `createAccount` operation succeeds and a fault occurs in the `createCustomer` operation, BPEL invokes the delete-account compensation handler. If the `BillingSystem` scope were inside a loop, BPEL would invoke each successive iteration's compensation handler instance in reverse order.

The code samples here show the raw XML syntax of BPEL and XML Schema. With graphical tools, however, developers need never deal with this syntax directly. Figure 9 shows the sample scenario as seen through the Cape Clear Orchestrator tool. For the scenario's full source code and instructions on how to run it in a BPEL engine, see www.capescience.com/articles/IEEESamples.

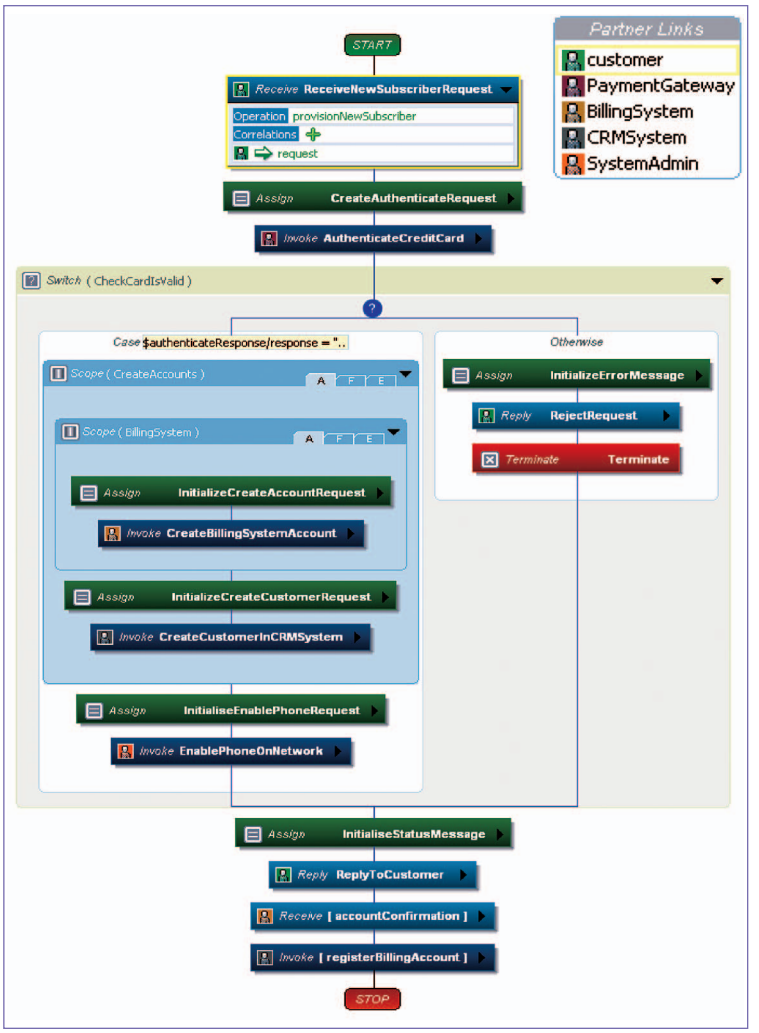

Figure 9. Graphical representation of the example scenario. Using graphical tools such as the Cape Clear Orchestrator, developers can avoid dealing directly with XML syntax.

References

1. D. Booth et al., "Web Services Architecture," W3C note, Feb. 2004; www.w3.org/TR/ws-arch/.
2. E. Christensen et al., "Web Services Description Language (WSDL) 1.1," W3C note, Mar. 2001; www.w3.org/TR/wsdl.html.
3. D.C. Fallside and P. Walmsley, *XML Schema Part 0: Primer, Second Edition*, W3C recommendation, Oct. 2004; www.w3.org/TR/xmlschema-0.
4. J. Clark, *XSL Transformations (XSLT), Version 1.0*, W3C recommendation, Nov. 1999; www.w3.org/TR/xslt.

James Pasley is chief architect at Cape Clear Software, where he oversees the development of Cape Clear's product suite. He has a BA Mod. in computer science from Trinity College, Dublin. Contact him at james.pasley@capeclear.com.