SCA Java Components

One of the component types available in an SCA assembly is a "Java Component". This is a Java class that can be used to implement an Interface. When the assembly calls the component, appropriate entry points in the Java code are called.

A Java component is an SCA component that has been implemented in Java. Thus, if you have an interface and you think the best implementation of that interface would be in Java, you can drag the interface onto your assembly diagram and choose to implement it in Java.

When doing this, a natural question arises: "if I am a component, I can have references. How do I access them?"

Calling a Reference

When a Java component wishes to call a reference defined as WSDL, there are two cases. The first case is the easy one: the WSDL is defined such that the input and output of the operation is a DataObject. In such a case, the programming model is the SCA model we know and love.

```
Service myService =
(Service)ServiceManager.INSTANCE.locateService("Name\_Of\_My\_Reference"); DataObject
inputObject = DataFactory.INSTANCE.create("Namespace\_of\_object",
 "name\_of\_object"); DataObject result = (DataObject)myService.invoke("operation\_name",
inputObject);
But what if the WSDL operation hasn't been defined to take a DataObject but, rather, is
specified as a collection of primitives? In this case, you know you need a DataObject (since
that's what the "invoke" method wants) but you don't have one. The 'Type' support in SCA/SDO
is the answer:
Service myService =
(Service)ServiceManager.INSTANCE.locateService("Name\_Of\_My\_Reference"); Reference ref =
myService.getReference(); OperationType opType =
```

```
ref.getOperationType("Name\_Of\_My\_Operation");
 Type targetType = opType.getInputType(); DataObject input =
DataFactory.INSTANCE.create(targetType);
 // // input now points to a DataObject with an element for each input parameter on the
interface.
 //
```

Java Component Error Handling

Java Components provide the unique ability to leverage the SCA programming model directly. This direct interaction allows a fine grained control of the client to service invocation. SOA can be described as a series of client and service provider interactions where a service can be a client for other services. Consider the following interaction

Component A calls component B.

Component B calls component C.

Component A is a client to the services provided by component B.

Component B is a service provider to component A and a client to component C.

When implementing Java components that promote a comprehensive error handling strategy there are a few factors to consider.

Defensive programming and programming with problem determination in mind encourage system and solution stability.

The problem determination strategy is an important subset of the overall error handling goal and should be consistent across all Java component implementation.

These considerations can be best described in the perspective of the client and the service provider.

Client Programming with Java

Components

The purpose of this section is to understand service interactions and how they relate to the client implementation.

There are two primary aspects of a client and service provider interaction. The operation type that the service provides and the invocation pattern that the client uses to invoke the service. Refer to the section call SCA Programming Model for a more detailed description of operations and invocation patterns.

Additionally, when Java components are implemented there are many opportunities for the component implementation to raise unexpected exceptions. For example, if the Java component implementation uses a variable that is not set this could raise a NullPointerException. The best practice should be to catch all exceptions prior to return. So as not to push the implementation of the Java Component beyond its interface.

Synchronous Invocation Patterns

The try and catch blocks can be used to control the upstream propagation of expected and unexpected exceptions. Failure to handle exceptions can lead to unpredictable results and overall system instability.

Try and catch blocks should be used in the operation implementation. This will prevent the inadvertent propagation of business and run-time exceptions.

```
// Call the request response service and catch any exception
 try {
 DataObject errorBOReturned = eI.reqresp(errorBO);
 } catch (ServiceBusinessException sbe) { System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*
Business exception caught insync/reqresp"); System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* "
+ sbe.toString());
 } catch (ServiceRuntimeException sre) { System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*
Runtime exception caught in sync/reqresp"); System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* "
+ sre.toString());
 } catch (Exception e) { System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* some other exception
caught in sync/reqresp");
 System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* " + e.toString());
 }
```

The code sample above makes a call to the reqresp method of the service via a reference.

Any exception returned from this invocation will be caught and printed to the system out log.

Business exceptions should be thrown when the java component encounters an exception that was defined on the interface. Business exceptions must represent the signature of the fault/throws defined on the service interface.

There are a couple of ways to create and throw a service business exception in a java component.

The first way is to construct he business exception using a string.

ServiceBusinessException sbe = new ServiceBusinessException("********** Service Business Exception constructed by string"); throw sbe;

The second way is to construct a data object that represents the BO defined by the service interface. Populate the data object and throw.

```
BOFactory bof = (BOFactory) sm.locateService("com/ibm/websphere/bo/BOFactory"); DataObject
faultBO = bof.create("http://ErrorHandling/com/ibm/summercamp",
 "FaultBO");
 faultBO.setString("faultString", "faultBO fault string"); ServiceBusinessException sbe =
new ServiceBusinessException(faultBO);
 throw sbe;
```

Service Run-time Exceptions

Run-time exceptions should be thrown when the java component has encountered an exception where the exception cannot be associated to a fault/throws defined in the interface.

Dependent on the invocation style used by the client, the run-time exception will be recorded as a failed event in the case of asynchronous type (one way, deferred response, and callback) or the run-time exception will be returned to the client to catch and handle or propagate further up stream.

Service Run-time Exceptions are handled differently when the client is a BPEL component.

Revision #1
Created 1 year ago by Admin
Updated 1 year ago by Admin