Reduce Development and Maintenance Time Using an Object-Oriented Framework for Deploying Application Logic in SAP JCo Server Environments

W. Patrick Tunney

W. Patrick Tunney has worked for the SAP Research organization since 2000. In 2003 he moved to SAP Labs Canada to establish a new research team. His current research includes the integration of sensor network data with ERP systems, the application of semantic Web technologies, and the logistics of software development and deployment.

(complete bio appears on page 110)

While isolating application logic from the mechanics of the underlying system infrastructure is a well-established concept in most object-oriented software design circles, there has been no common model that allows developers to easily implement Java-side application logic independent of the SAP Java Connector (JCo) server infrastructure. The result is that many Java programmers do not follow this venerable discipline when leveraging the JCo API, and this, as we all know, can lead to untold consequences for developers and administrators alike. Java developers who leverage JCo for building custom applications can't escape the inescapable: changing market demands, bug fixes, and feature requests all conspire to confound software development, deployment, and ongoing maintenance efforts.

Few running systems go untouched for long, so I have devised some simple Remote Function Call (RFC) implementation principles that will help you separate your application logic from your JCo server infrastructure coding, and in so doing simplify your software's development lifecycle over the short and long term. Imagine in the months or years ahead being able to easily add a new application logic object to your production system, or add or remove function module objects dynamically without the need to restart the JCo server.[1] You can, if you leverage the JCo API to set up a Java-side deployment framework that clearly separates the mechanics of the RFC invocation from the application logic. In this article, I will show you how.

My goal in presenting this deployment framework is twofold. First, I hope to help you reduce your application development time.

[1] For a detailed look at how to leverage the JCo API, see the article "Server Programming with the SAP Java Connector (JCo)," published in the September/October 2003 edition of *SAP Professional Journal*.

Figure 1 Standard Process for Handling an Incoming RFC Request in a Java Program

```
 1 import com.sap.mw.jco.*;
 2 import java.util.Properties;
 3
 4 /** The simplest RFC Server ever. */
 5 public class BasicServer extends JCO.Server {
 6
 7     public BasicServer(Properties properties, IRepository repository){
 8         super(properties, repository);
 9     }
10
11     protected void handleRequest(JCO.Function function){
12
13         if(function.getName().equals("HELLO_WORLD")){
14             JCO.Structure returnCode =
                 function.getExportParameterList().getStructure("RETURN");
15             returnCode.setValue("S", "TYPE");
16             returnCode.setValue("MESSAGE", "Hello World!");
17         }
18
19     }
20
21 }
```

Second, I hope to help you simplify downstream code modification activities by enabling you to add new application logic without affecting previously deployed application logic.

How Does This Deployment Framework Ease Code Maintenance?

A look at the way most developers now approach JCo server coding will provide you with an understanding of where we are today and how the deployment framework outlined in this article improves our lot. At present, setting up a Java program to handle incoming RFC requests is fairly straightforward. Using the JCo API, a Java programmer can have a Java RFC server up and running in no time.

The standard process for handling incoming RFC requests in a Java program is to simply extend the *JCO.Server* parent class and override the *handleRequest* method. One such example[2] is provided in **Figure 1**.[3] This simple JCo server (*BasicServer*) handles a single RFC request called *HELLO_WORLD* and sets a message in its return structure.

In this particular example, there is no clear distinction between the code that manages the JCo server infrastructure and the code that implements the application logic. The direct consequence of mixing the two is that in order to make changes to the implementation of *HELLO_WORLD*, you must modify the server source code. In practice, this isn't "the end

[2] While the code segments in this article provide a working solution, keep in mind that they are intentionally simplified for illustrative purposes. For example, several code snippets create and use a *JCo.Client* object directly. This is not something I would advise doing on a production system, which would be better served by a connection pooling strategy.

[3] Note that line numbers have been added to the code for your convenience.

of the world," if you'll excuse the pun. If, for example, you change the *HELLO_WORLD* function signature[4] in a way that removes the *RETURN* structure, *handleRequest* will throw a runtime exception that aborts the RFC invocation. Thanks to some clever coding by the SAP JCo development team, the aborted invocation is only a temporary inconvenience and won't "crash" your server. However, if we take a closer look at the design approach illustrated in Figure 1, with an eye toward maintaining the code in the future, two fundamental deficiencies become apparent:

1. Every time application logic is changed, there is an inherent risk that the changes may affect the system at large, so even the most mundane modifications require that the infrastructure source code be "checked out" of the local concurrent versioning system and modified. An unnoticed mistake could have ramifications not just for the application logic being modified, but also for all the units of application logic that follow. One simple mistake in the control flow logic, and you could find yourself with a bug that is difficult to isolate because the affected functionality wasn't changed.

2. The implementation makes it difficult to apply unit-testing methodologies. Typical *handleRequest* implementations contain a central *if … else if … else* block along with conditional blocks that each contain the application logic for a single function module. Since every conditional block requires multiple test cases, the number of test cases can quickly become unmanageable. In an ideal scenario, each unit of application logic would be defined as its own unique object so that it can be tested outside the context of the running JCo server. Grouping the application logic in self-contained units would allow you to implement a more thorough and simplified unit-testing methodology, and would make it easier to reuse the code in another environment — in an Enterprise JavaBean as part of a J2EE server, for example.

These problems present themselves not just when modifications to the existing application logic are necessary, but also when function modules need to be added to or removed from the server. While it is fairly easy to maintain code for simple server implementations, maintaining implementations with complex control structures or a large number of function modules, or that require frequent additions and removals of function modules, presents an increased risk of unexpected failure, which can cause all kinds of problems you can't afford, especially in your mission-critical applications.

The object-oriented deployment framework I describe in this article was devised to address these concerns by providing an immutable server instance that allows for the safe addition and removal of function modules that are defined as unique, independent objects. Defining the function modules as unique objects provides a clear separation between the server infrastructure and the application logic, and helps ease maintenance and unit testing. The function module objects are added to and removed from the JCo server through an interface that allows the *handleRequest* method to resolve and execute them at runtime according to the name provided by the *JCO.Function* parameter, which is generated automatically by the JCo server at runtime.

Over the remainder of this article, we will look at the elements of the deployment framework. First, we'll look at the interface that keeps the server infrastructure and the application logic separate yet connected, and defines the actions that can be performed on the function modules. Then, you'll see how to implement the interface in a function module so that it can be added or removed without affecting the server's infrastructure coding, as well as how to design the implementation so that you can easily reuse code between function modules that share functionality. And finally, I will show you how to tie the framework together with one class that starts the server and another that performs the defined actions on the function modules — i.e., adding them to and removing them from the JCo server.

[4] The function signature is either hard-coded as a local repository or referenced via a remote SAP repository. The recommended method is to reference a remote SAP repository (for more on this, see the article "Repositories in the SAP Java Connector (JCo)" in the March/April 2003 edition of *SAP Professional Journal*).

Figure 2 The RFCModule Interface at a Glance

Method Signature	Description	Preconditions	JCo-Specific Comments
`String getFunctionName()`	Identifies a function module object based on the function name provided.	None.	The return value should be unique to the function module object class. In the deployment framework, it exactly matches the name provided by the JCO.Function parameter of handleRequest.
`void process(JCO.ParameterList input, JCO.ParameterList output, JCO.ParameterList tables)`	Allows the function module code to be executed with a single method call.	The parameters should contain all the information required to execute the function module. After execution, the parameters should contain all the information associated with the results of processing the application logic.	Most function modules have a BAPIRET2 structure that indicates whether the application logic executed successfully. Depending on the purpose and success of the function module, additional information is usually available in the output and tables objects.
`boolean assertStructures(IRepository repository)`	Ensures that all metadata descriptions (i.e., structures and tables) required by the function module object are available in the server repository.	If all required metadata descriptions were found, true is returned; otherwise, false is returned.	In most cases, the repository parameter will be linked to a remote repository via a JCO.Client connection, which is the recommended method.
`String getMissingStructure()`	Returns information on missing metadata elements.	assertStructures must be called first in order for this method to do anything useful.	Call this method only if assertStructures returns false.

An Interface for Encapsulating Application Logic

The first step in separating the server infrastructure from the application logic is to pack the application logic into unique function module objects. The deployment framework includes a permanent interface called *RFCModule* that serves as a link between the *JCO.Server* class, which contains the server infrastructure code (i.e., the class that implements *handleRequest),* and the unique function module objects that contain the application logic. The interface is implemented by the function module objects you want to add to or remove from the server. When the *handleRequest* method of the *JCO.Server* class is invoked during runtime, the server simply checks to see if the function module name specified in the incoming RFC request matches that of a function module object added to the server. If a match is found, the server will invoke the function module in question, which causes the application logic to execute.

The *RFCModule* interface includes the features listed in **Figure 2**. The lifecycle of a function module object that implements this interface consists of two phases: initialization and execution. Initialization involves constructing the function module object and calling the *assertStructures* method to make sure that the required metadata for the function module object is available in the server repository. If *assertStructures* fails, a call to the *getMissingStructure* method can be made to determine which metadata element could not be found. Following successful initialization, the execution phase is characterized by calling the *getFunctionName* method to see if the function module specified in an incoming RFC request matches the name of this object and, if so, calling the *process* method to execute the application logic unique to the function module object.

Figure 3 API for the RFCModule Interface

```
 1 import com.sap.mw.jco.JCO;
 2 import com.sap.mw.jco.IRepository;
 3
 4
 5 public interface RFCModule {
 6
 7     /** Returns the name of the function implemented */
 8     public String getFunctionName();
 9
10     /** Implements the application logic*/
11     public void process(JCO.ParameterList input, JCO.ParameterList output,
         JCO.ParameterList tables);
12
13     /** Verifies that all required metadata structures are available in
           repository.
14      * @return true if all metadata elements are available, false otherwise. */
15 public boolean assertStructures(IRepository repository);
16
17     /** If the assertStructures method returns false, the name of the missing
18      * metadata element is returned, otherwise "" is returned.*/
19     public String getMissingStructure();
20
21 }
```

The code segment in **Figure 3** shows the API for the *RFCModule* interface. See the sidebar below for a summary of the requirements that must be in place in order to use the interface.

Requirements for Using the RFCModule Interface

The basic "contract" for using the RFCModule interface can be summarized as follows:

- ☑ All function module objects that implement the RFCModule interface are identified by a unique String, which is obtained by calling the getFunctionName method (lines 7-8 in Figure 3).
- ☑ The function module object is invoked by calling the process method with the appropriate parameters (lines 10-11 in Figure 3).
- ☑ Prior to invoking the process method, the assertStructures method is called to ensure that the server repository contains all the required metadata descriptions for the functions, structures, and tables used by the function module (lines 13-15 in Figure 3).
- ☑ Should assertStructures fail, the name of the metadata element identified as missing is available by calling the getMissingStructure method (lines 17-19 in Figure 3). (For simplicity, the example code will list only the first element identified as missing, rather than all missing elements.)

Figure 4 The JCo.Server Class RFCServer at a Glance

Method Signature	Description	Preconditions	JCo-Specific Comments
`RFCServer(Properties properties, IRepository repository)`	Constructs a new RFCServer.	The Properties parameter contains all the key/value pairs required to start the server. The repository object contains the metadata descriptions the server needs to successfully handle incoming RFC requests.	See the JCo documentation concerning valid Properties key/value pairs.
`void addHandler(RFCModule handler)`	Enables a function module object to be added to the server.	The function module object must be successfully initialized; it cannot be null (see below).	Make sure that no objects share the same function name; otherwise, the name mapping is ambiguous.
`void removeHandler(RFCModule handler)`	Enables a function module object to be removed from the server.	The RFCModule parameter must not be null; otherwise, a runtime exception is thrown.	This approach requires that the external code used to add and remove function module objects keep a reference to all added modules. Another approach is to use the function name as a parameter.
`void handleRequest(JCO.Function function)`	Handles incoming RFC requests.	This method is called by the JCo framework.	This method should be called by the parent class rather than directly to ensure that the parameters are correct.

Through the *RFCModule* interface, a *JCo.Server* class can implement the *handleRequest* method such that the core server infrastructure code never changes. **Figure 4** provides an overview of the methods that make this possible, using an example *JCo.Server* class called *RFCServer*. Here's how it works: A new *RFCServer* object is created by calling its constructor with the appropriate *Properties* key/value pairs (these are JCo-specific and are defined by the general JCo framework) and a metadata repository. Function module objects that implement the *RFCModule* interface are added by calling *addHandler* with the function module object as the parameter. The server is started by calling *start* (inherited from *JCO.Server*). When incoming RFCs are received, *handleRequest* is called by the JCo framework, once per RFC. A check is performed to see if the requested RFC name matches the name of a function module object previously added with *addHandler*. If a match is found, the *process* method of the appropriate function module object is invoked. Finally, the server is shut down by calling *stop* (also inherited from *JCO.Server*).

The API shown in **Figure 5** illustrates how the example *JCo.Server* class *RFCServer* uses the *RFCModule* interface and an internal *Hashtable* object to add or remove a function module object without changing the server infrastructure code.

First let's examine the server's constructor, which does just two things:

Figure 5 API for the JCo.Server Class RFCServer

```
 1 import com.sap.mw.jco.JCO;
 2 import com.sap.mw.jco.IRepository;
 3 import java.util.Properties;
 4 import java.util.Hashtable;
 5
 6 public class RFCServer extends JCO.Server {
 7
 8     /** A list of function handlers that implement the RFCModule interface. */
 9     private Hashtable functionHandlers;
10
11     public RFCServer(Properties properties, IRepository repository){
12         super(properties, repository);
13         this.functionHandlers = new Hashtable();
14     }
15
16     /** Adds a function module object to an internal list of function module
           objects.
17     public synchronized void addHandler(RFCModule handler) {
18         this.functionHandlers.put(handler.getFunctionName(), handler);
19     }
20
21     /** Removes a function module object from an internal list of function
           module objects.
22     public synchronized void removeHandler(RFCModule handler) {
23         this.functionHandlers.remove(handler.getFunctionName());
24     }
25
26
27     protected synchronized void handleRequest(JCO.Function function){
28
29         JCO.ParameterList input  = function.getImportParameterList();
30         JCO.ParameterList output = function.getExportParameterList();
31         JCO.ParameterList tables = function.getTableParameterList();
32
33         RFCModule update = (RFCModule)functionHandlers.get(function.getName());
34
35         try{
36          update.process(input, output, tables);
37         }catch(NullPointerException npe){
38                throw new JCO.AbapException("FUNCTION_NOT_SUPPORTED", "Requested
                    function not supported.");
39         }
40     }
41
42 }
```

1. It initializes the server by passing a *Properties* object and an *IRepository* object to the parent constructor (line 12). The *Properties* object defines basic key/value pairs that specify various server parameters. The *IRepository* object is an SAP system repository object that must contain all of the metadata descriptions required by the server in order to handle incoming RFC requests.

> ✓ ***Note!***
>
> *The RFCServer class in Figure 5 makes use of the "synchronized" keyword (line 27). This is an important addition, since the handleRequest method is executed in its own thread in response to an incoming RFC request. Because the thread that invokes handleRequest is invoked internally by the JCo framework, it is independent of any threads that add or remove function module objects by calling addHandler or removeHandler. Given the nature of the Java Memory Model, the "synchronized" keyword must be used when the member Hashtable object is accessed, so that the invoking thread sees the most up-to-date version of the hashtable. Because there are performance implications associated with the use of the "synchronized" keyword, be judicious with its use in a complex implementation.*

2. It creates a new *Hashtable* object (line 13), which will be used to store all function module objects that are added to the server. This hashtable will be used by the *handleRequest* method to locate and execute function module objects.

The methods *addHandler* (lines 16-19) and *removeHandler* (lines 21-24) add and remove function module objects to and from the hashtable using the name of the function module as the hash key. Function module objects are referenced according to the *RFCModule* interface, which a function module must implement in order to be used with the deployment framework.

By using the *RFCModule* interface, the *handleRequest* method (lines 27-40) is now reduced to a simple three-step process of looking up a remote function module object in the hashtable (line 33), casting the object to an *RFCModule* object (line 33), and invoking the object's *process* method (line 36).

Using a *Hashtable* object in conjunction with add and remove methods creates a clean separation between the code that defines the server infrastructure and the application logic, because the server doesn't need to know anything about the function module object except that it implements the *RFCModule* interface. (Note that the actual instantiation and initialization of function module objects that implement the *RFCModule* interface, and the process of adding and removing them to and from the server, is handled by external code. Be patient, we'll get to this shortly.) The core of the server remains constant, independent of the application logic, which is encapsulated in the *RFCModule* implementation and is independent of the number of function module objects added to or removed from the server. The results are cleaner, and the design is more stable.

Implementing the Interface in a Function Module Object

Now it's time to turn our attention to implementing the deployment framework interface in function module objects so that they can be added to or removed from the server. **Figure 6** shows the source code of

Figure 6 Source Code for Function Module Object HelloWorldHandler

```
 1 import com.sap.mw.jco.IRepository;
 2 import com.sap.mw.jco.IMetaData;
 3 import com.sap.mw.jco.JCO.ParameterList;
 4 import com.sap.mw.jco.JCO;
 5
 6 public class HelloWorldHandler implements RFCModule {
 7
```

(continued on next page)

Figure 6 (continued)

```
 8     /** The function name as defined in the repository. */
 9     private static final String functionName = "HELLO_WORLD";
10
11     /** Structures that belong to the function call */
12     private static final String[] memberStructures = {"BAPIRET2"};
13
14     /** If a structure is found to be missing when assertStructures
15      * is called, then it is stored here.*/
16     protected String missingStructure;
17
18     public HelloWorldHandler(){
19         this.missingStructure = "";
20     }
21
22     public final String getFunctionName() {
23         return functionName;
24     }
25
26     public void process(ParameterList input, ParameterList output,
         ParameterList tables) {
27
28         JCO.Structure returnCode = output.getStructure("RETURN");
29         returnCode.setValue("S", "TYPE");
30         returnCode.setValue("MESSAGE", "Hello World!");
31         return;
32     }
33
34
35
36     public final boolean assertStructures(IRepository repository){
37           for(int i=0, count=memberStructures.length; i<count; i++){
38                 if(repository.getStructureDefinition(memberStructures[i]) ==
                    null){
39                     this.missingStructure = memberStructures[i];
40                     return false;
41                 }
42           }
43
44           if(repository.getFunctionInterface(functionName) == null){
45                 this.missingStructure = functionName;
46                 return false;
47           }
48
49           return true;
50     }
51
52
53
54     public String getMissingStructure(){
55           return this.missingStructure;
56     }
57 }
```

the function module object *HelloWorldHandler*, which implements the *RFCModule* interface (line 6). The *HelloWorldHandler* object provides the application logic associated with the *HELLO_WORLD* function module in its *process* method and fulfills the "contract" for using the *RFCModule* interface outlined in the sidebar on page 97.

Notice that the *getFunctionName* method always returns *HELLO_WORLD* (lines 22-24), since it returns the *functionName* variable, which is marked *final*. This allows the server infrastructure to map the *HelloWorldHandler* object to the appropriate function call at runtime. Since our server implementation refers to all objects by their interface type and not by their class type, this is the only hint the server receives concerning internal implementation details (without having to delve into the deep, dark world of the Java reflection API).

The implementation of the *process* method is virtually identical to the hard-coded version of the application logic found in the *handleRequest* method of Figure 1. The application logic remains the same as before. The only material difference between the two implementations is the additional layer of abstraction that decouples the application logic from the server infrastructure. Should future modification to the application logic of *HELLO_WORLD* be necessary, the *process* method can be altered without affecting other function modules, as would be the case for implementations of *handleRequest* that make use of the previously mentioned *if … else if … else* approach.

The last thing to note is the *assertStructures* method (lines 36-50), which verifies that all metadata descriptions required by the function module are available in the server metadata repository, represented by the *IRepository* parameter. In this case, *HelloWorldHandler* needs only two such metadata elements: the *BAPIRET2* structure (lines 11-12) and the *HELLO_WORLD* function interface (lines 8-9). The server needs information about these two elements in order to handle incoming *HELLO_WORLD* RFC requests. To ensure that your server has this information, you must either hard-code the metadata yourself or retrieve it dynamically by linking to an existing SAP system that contains the definitions. The preferred approach is to link to an existing SAP system in which you define custom function module interfaces with the Function Builder (transaction code *SE37*).[5]

The *assertStructures* method explicitly checks to make sure the *IRepository* object contains both a structure definition for *BAPIRET2* and a function interface definition for *HELLO_WORLD*. Should either definition be missing from the local *IRepository* object, the name of the missing structure is assigned to the *missingStructure* member variable, and *false* is returned (lines 40 and 46). There are three reasons for performing the checks proactively:

1. The consequence of decoupling the application logic from the server infrastructure is that you can no longer make any assumptions about the environment in which the function module is deployed. By proactively polling your environment during initialization, you can programmatically determine if the operating environment meets the implicit prerequisites of the function module. This helps reduce debugging time by immediately identifying any inconsistencies and even helps you track down the cause of the problem by inspecting *missingStructure*.

2. If you're using the recommended method of linking to an existing SAP repository over a remote connection, each time you poll the remote repository for metadata definitions, the definitions are added to the cache of the local repository. So, in the unlikely event that your repository loses its connection to the remote SAP system, all the metadata definitions required to handle incoming RFC requests will be available in the local cache. Note, however, that since this is an internal implementation detail, there are no guarantees that this won't change in future JCo versions.

3. The *assertStructures* initialization is important if

[5] For more details on using repositories in a JCo server, see the section "Repositories in JCo Server Components" in the article "Repositories in the SAP Java Connector (JCo)," published in the March/April 2003 edition of *SAP Professional Journal*.

you need to deploy your application logic in multiple physical environments (e.g., from a sandbox system to a production system). In this situation, the problem with linking to a remote SAP repository is that you need to make sure that you add all of the custom function definitions you created manually with the Function Builder to the new SAP system. Granted, a strong change management process will prevent these types of issues, but sometimes, when push comes to shove, you need to take matters into your own hands. In those cases, *assertStructures* will prove especially valuable.

The motivation for the third reason may not be immediately apparent and lies in the internal details of how a JCo server handles low-level RFC requests. When a low-level RFC request is received, *JCO.Server* uses the function signature and associated data values of the incoming low-level RFC to generate a *JCO.Function* object. This handling of an incoming RFC involves triggering a Java event that causes the *handleRequest* method to execute. If a required function interface, structure, or table definition is missing or incorrectly defined in the server's *IRepository*, the server will not have all of the information it needs to handle the incoming RFC, and the *handleRequest* method will not be called. The JCo framework rejects the incoming RFC because it contains structures or table types that the server doesn't recognize. This is an extremely painful lesson to learn the hard way, because as a developer you don't get any special debugging information about why your newly deployed application logic doesn't execute. Once you figure out that the repository is missing metadata information, identifying which element is missing or incorrectly defined can only be determined through manual inspection. By using a preemptive initialization step that checks for and reports any missing structures, metadata discrepancies can be easily detected and addressed, minimizing debugging time and associated frustration.

The only disadvantage of using this proactive checking approach is that you need to make sure that you correctly list the names of the structures and tables that the function module uses in the function module source code. Given that there is no way for developers to control where function modules are deployed, this inconvenience is justified. Also, if you forget to list a table or structure in the source code, it won't prevent the function module from functioning properly — you just won't be able to ensure that the table or structure in question is a part of the function interface in the remote SAP system.

Grouping Shared Functionality with an Abstract "Skeletal Implementation"

When creating multiple function modules that make use of the *RFCModule* interface, it would be helpful to be able to reuse some of the code — like the *assertStructures* code that checks for the required metadata elements, for example — so that you don't have to keep reentering it in each function module you create. One approach is to cut and paste the code from the *HelloWorldHandler* class into a new implementation. Unfortunately, in addition to risking occasional errors, this approach increases the maintenance commitment — for example, should a bug or performance deficiency be found in one implementation, the same bug or performance deficiency must be corrected in every single implementation that uses the copied code segment.

Another approach is to do away with using the *RFCModule* interface and instead define *RFCModule* as an abstract base class from which all implementations are derived. This way, you would only need to write the code for *assertStructures* once in the base class, and all subclasses would get the benefit of performance improvements and bug fixes. While this would be a valid approach, it needlessly forces all implementations to conform to a rigid class hierarchy. This approach throws out the main benefit of Java interfaces — multiple inheritance through the use of interface APIs instead of through subclassing. Due to the simplistic nature of the *RFCModule* interface, committing to a single class hierarchy is probably not a deal-breaker; nonetheless, since requirements can

Figure 7 Source Code for the AbstractRFCModule Base Class

```
 1 import com.sap.mw.jco.JCO;
 2 import com.sap.mw.jco.IRepository;
 3
 4 public abstract class AbstractRFCModule implements RFCModule {
 5
 6     /** The function name as defined in the repository. */
 7     protected String functionName;
 8
 9     /** Structures and tables that belong to the function call */
10     protected String[] memberStructures;
11
12     /** If a metadata element is found to be missing
13      * it is stored here.*/
14     protected String missingStructure;
15
16     /** Sub-classes must implement this method to perform
17      * useful function handling. */
18     public abstract void process(JCO.ParameterList input, JCO.ParameterList output,
         JCO.ParameterList tables);
19
20     /** Returns the name of the function module. */
21     public final String getFunctionName() {
22         return functionName;
23     }
24
25     /** Default constructor. */
26     public AbstractRFCModule() {
27         this.missingStructure = "";
28     }
29
30     /** Checks for all required metadata */
31     public final boolean assertStructures(IRepository repository){
32           for(int i=0, count=memberStructures.length; i<count; i++){
33                 if(repository.getStructureDefinition(memberStructures[i]) == null){
34                       this.missingStructure = memberStructures[i];
35                       return false;
36                 }
37           }
38
39           if(repository.getFunctionInterface(functionName) == null){
40                 this.missingStructure = functionName;
41                 return false;
42           }
43
44           return true;
45     }
46
47     public String getMissingStructure(){
48           return this.missingStructure;
49     }
50 }
```


Figure 8 Source Code for the GoodbyeWorldHandler Class

```
 1 import com.sap.mw.jco.JCO.ParameterList;
 2 import com.sap.mw.jco.JCO;
 3
 4 public class GoodbyeWorldHandler extends AbstractRFCModule implements RFCModule {
 5
 6     /** Repository name of the function handled by this class. */
 7     public static final String FUNCTION_NAME = "GOODBYE_CRUEL_WORLD";
 8
 9     /** Only requires structure BAPIRET2. */
10     public static final String MEMBER_STRUCTURES[] = {
11                                                 "BAPIRET2"
12                                                       };
13
14     public GoodbyeWorldHandler() {
15         this.functionName = FUNCTION_NAME;
16         this.memberStructures = MEMBER_STRUCTURES;
17     }
18
19     public void process(ParameterList input, ParameterList output, ParameterList
         tables) {
20
21         JCO.Structure returnValue = output.getStructure("RETURN");
22         returnValue.setValue('S', "TYPE");
23         returnValue.setValue("Goodbye Cruel World!","MESSAGE");
24         output.setValue(returnValue, "RETURN");
25         return;
26     }
27
28 }
```

and frequently do change, it's nice to keep your options open if you can.

Fortunately, due to the inherent flexibility of the Java programming language, you can combine the benefits of both interfaces and abstract classes by using an interface to define the type API and an abstract base class to define common default behavior. This is known as "skeletal implementation" design. **Figure 7** and **Figure 8** illustrate how the functionality of function module object *GoodbyeWorldHandler* (essentially the same as *HelloWorldHandler*) is distributed across two classes: an abstract base class named *AbstractRFCModule* that defines common variables and behavior shared by most *RFCModule* implementations (Figure 7), and the *GoodbyeWorldHandler* class that extends the abstract base class and implements the application logic that defines it as a unique function module (Figure 8).

Making use of an abstract base class greatly reduces the amount of development time required to create a new function module. It also ensures a more stable code base though judicious use of the *final* keyword and common code segments. In fact, the creation of a new function module that derives from *AbstractRFCModule* requires only the following additions:

- Assignment of the *functionName* variable to reflect the runtime *JCO.Function* name
- Assignment of the *memberStructures* variable to reflect the names of all metadata elements

Figure 9 The RFCHost Class at a Glance

Method Signature	Description	Preconditions	JCo-Specific Comments
`RFCHost()`	The default constructor.	This constructor throws a runtime exception that indicates the non-default constructor (i.e., a constructor containing parameters) should be used instead (see below).	Throwing a runtime exception is a common way to prevent the use of the default constructor. You could also declare the default constructor as private, but this would prevent future subclassing.
`RFCHost(Properties initalValues)`	A constructor that uses predefined JCo key/value pairs.	The Properties parameter contains all the key/value pairs required to start the server.	This constructor must be used instead of the default constructor (see above).
`void run()`	Starts the server.	This method can be called only once; otherwise, an IllegalStateException is thrown.	Even though this method defines the standard "Runnable" interface, since this method can only be called once, I chose not to add "implements Runnable" to the class declaration.
`void addHandler(RFCModule handler)`	Adds a function module object to the server.	The run() method must have been called and the RFCModule parameter must not be null.	Invokes assertStructures on the RFCModule parameter. If it fails, the function module object is not added to the underlying server.
`void removeHandler(RFCModule handler)`	Removes a function module object from the server.	The run() method must have been called and the RFCModule parameter must not be null.	Attempting to remove a function module object that was not successfully added to the server has no ill effects.

required by the function module object in the server repository

- Implementation of the *process* method to execute the desired application logic

Tying It All Together with an RFCHost Class and a Main Class

The final pieces of our framework, which tie all the previously discussed elements together into a working system, are an *RFCHost* class and a simple class called *Main* that provides the "public static void main" entry point invoked by the Java Virtual Machine when the program is started. These last two pieces implement the code required to start the server and to actually instantiate and add function modules.

The *RFCHost* class includes the features listed in **Figure 9**.

Figure 10 shows the source code for the *RFCHost* class, which provides the following functionality to complete the system:

- Links to a remote SAP repository that contains the required metadata descriptions (lines 33-42)
- Creates and starts an RFC server by passing on the backend connection settings and the repository object (lines 43-44)
- Provides convenience methods for adding and

Figure 10 Source Code for the RFCHost Class

```
 1 import java.util.Properties;
 2 import com.sap.mw.jco.IRepository;
 3 import com.sap.mw.jco.JCO;
 4
 5 public class RFCHost {
 6
 7     /** The currently active JCO connection*/
 8     private JCO.Client jcoClient;
 9
10     /** The properties used for backend connection initialization */
11     private Properties properties;
12
13     /** The repository. */
14     private IRepository repository;
15
16     /** Server instance that does the actual RFC handling. */
17     private RFCServer rfcServer;
18
19     /** Default constructor - do not use.  */
20     public RFCHost(){
21         throw new RuntimeException("Use non-default constructors.");
22     }
23
24     public RFCHost(Properties initalValues){
25         this.properties = initalValues;
26         this.jcoClient = null;
27         this.repository = null;
28         this.rfcServer = null;
29     }
30
31     /** Start the server */
32     public void run() {
33         if(this.rfcServer == null){
34
35                 if(this.jcoClient == null){
36                 this.jcoClient = JCO.createClient(this.properties);
37                 // system number
38                 // Open the connection
39                 this.jcoClient.connect();
40            }
41            //create and start the server.
42            this.repository = JCO.createRepository("LocalRepository",
                this.jcoClient);
43            this.rfcServer = new RFCServer(this.properties, this.repository);
44            this.rfcServer.start();
45         }else{
46            throw new IllegalStateException("run() can be called only once");
47         }
48
```

(continued on next page)

Figure 10 (continued)

```
49      }
50
51      public void addHandler(RFCModule handler) {
52
53          try{
54                  if(!handler.assertStructures(this.repository)){
55                        System.out.println(""+handler.getFunctionName()+"
                            unable to assert all dictionary structures required.
                            Missing: "+handler.getMissingStructure());
56                          return;
57                  }
58                  rfcServer.addHandler(handler);
59
60          }catch(NullPointerException npe){
61              if(handler == null){
62                          throw new IllegalArgumentException("handler can't be
                              null";
63                   }else{
64                          throw new IllegalStateException("Must start server
                              before adding handlers");
65                   }
66          }
67      }
68
69           public void removeHandler(RFCModule handler) {
70              try{
71                   rfcServer.removeHandler(handler);
72              }catch(NullPointerException npe){
73                   if(handler == null){
74                          throw new IllegalArgumentException("handler can't be
                              null");
75                   }else{
76                          throw new IllegalStateException("Must start server
                              before removing handlers");
77                   }
78          }
79      }
80                      }
```

removing objects that implement the *RFCModule* interface used by the server (lines 51-67 and lines 69-79)

Finally, **Figure 11** illustrates a simple class that launches the *RFCHost* class via its *main* method. This simple class can be modified at will to change both the backend connection parameters and to add or remove function handlers without affecting the underlying server infrastructure. In practice, this will be the only piece of code you will ever need to modify, other than creating new function module objects and modifying existing application logic. If you intend to compile this code, make sure that you adjust the *Properties* key/value pairs to match your system settings.

Figure 11 Source Code for the Main Class

```
 1 import java.util.Properties;
 2
 3 public class Main {
 4
 5     public static void main(String args[]){
 6
 7           Properties serverProps = new Properties();
 8           serverProps.put("jco.client.client","???");
 9           serverProps.put("jco.client.user",  "???");
10           serverProps.put("jco.client.passwd","???");
11           serverProps.put("jco.client.lang","???");
12           serverProps.put("jco.client.ashost","???");
13           serverProps.put("jco.client.sysnr","???");
14           serverProps.put("jco.server.gwhost","???");
15           serverProps.put("jco.server.gwserv","???");
16           serverProps.put("jco.server.progid","???");
17
18        RFCHost obj = new RFCHost(serverProps);
19        obj.run();
20
21        HelloWorldHandler hello = new HelloWorldHandler();
22        GoodbyeWorldHandler goodbye = new GoodbyeWorldHandler();
23
24        obj.addHandler((RFCModule)hello);
25        obj.addHandler((RFCModule)goodbye);
26
27     }
28 }
```

> ✓ ***Note!***
>
> *Because we refer to function modules according to interface type, classes like HelloWorldHandler that do not extend AbstractRFCModule can peacefully coexist with those that do.*

Helpful Hints

☑ The simplest way to create your own *RFCModule* class is to extend *AbstractRFCModule*, assign values to *functionName* and *memberStructures*, and implement the application logic in the *process* method.

☑ Abstract skeletal base classes make your life easier by providing default behavior to subclasses. You don't need to use them as long as you fulfill the "contract" of the interface in question (refer back to the sidebar on page 97 for an example of such a contract).

☑ The lifecycle for function module objects that implement the *RFCModule* objects as presented in this article is one-time construction and assertion followed by repeatable *process* calls.

☑ Don't worry too much about optimizing performance by trying to work around *synchronized* method

calls. If you find that your server is underperforming, first make sure that the *synchronized* calls are causing the bottleneck. If so, one approach might be to restrict the addition of *RFCModule* objects to an explicit one-time initialization phase.

☑ If you intend to compile the code in this article, keep in mind that it is not supported by SAP and that you will need to have the JCo library installed and added to your classpath.

☑ For Java installations that require more than just simple RFC-handling capabilities, consider using SAP NetWeaver, which allows you to maximize the benefits of both the Java and ABAP worlds (see **http://service.sap.com/netweaver** for more information).

Conclusion

The deployment framework presented here for separating application logic from server infrastructure attempts to simplify the jobs of both developers and server administrators through the judicious use of abstraction layers and adherence to simple object-oriented design principles that separate application logic from infrastructure details.

The chief benefit of separating application logic from infrastructure becomes abundantly clear when you look at the simple code presented in Figure 11. In order to add a new application logic object to the server, the server administrator need only instantiate a new object and add it the *RFCHost* class — a grand total of two lines of code. Not only does this greatly simplify the deployment process for new application logic, it also provides a simple add/remove API that could be exploited by ambitious programmers to provide dynamic addition or removal of function module objects at runtime (i.e., addition or removal of function module objects without restarting the server).

While a certain decrease in maintenance is already achieved implicitly through the separation of application logic from infrastructure, this gain is amplified by leveraging the abstract "skeletal implementation" design pattern. This design pattern groups together functionality shared by most application logic objects into a single base class, thus reducing the amount of redundant code and associated maintenance cost without forcing objects into a rigid class hierarchy.

In this article, I've tried to make a clear case for the application of some basic design themes that can help reduce your long-term maintenance commitment. Despite what some people may say, writing good, maintainable, extensible software is hard. Hopefully the techniques illustrated in this article will help make your professional life a little easier.

W. Patrick Tunney has worked for the SAP Research organization since 2000. After spending three years at the Campus-based Engineering Center in Karlsruhe, Germany, in 2003 he moved to SAP Labs Canada to establish a new research team there. Patrick's current research includes the integration of sensor network data with ERP systems, the application of semantic Web technologies, and the logistics of software development and deployment, with a focus on code reuse and object-oriented frameworks. He can be reached at patrick.tunney@sap.com.

