Repositories in the SAP Java Connector (JCo)

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's BIT525 and BIT526 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 40)

to know a thing and not expresse it,
is all one as if he knewe it not.
Robert Burton, The Anatomy of Melancholy

The SAP Java Connector (JCo) is the premier middleware for connecting non-SAP components written in Java to ABAP-based SAP systems like R/3. The non-SAP components can be clients (Java calls ABAP) or servers (ABAP calls Java). In both cases, you need a repository object that represents the metadata (parameters and exceptions) for the functions to be invoked. Using JCo repositories properly is a key contributor to stable and performance-optimized applications. This article will:

- Show you the recommended use of repositories in client components
- Explain why it is important to have only one repository object for each SAP system you connect to
- Explain the value of subclassing the standard JCo repository class (*JCO.Repository*)
- Show you how to avoid hard-coded repositories in server components
- Introduce a concept for managing repositories in web server applications
- Discuss two utility classes[1] that manage repositories for you

[1] The complete source code for these classes is contained in the appendices to this article.

Listing 1: Creating a JCO.Repository Object

```
static final String POOL_NAME = "Pool";
IRepository mRepository;
mRepository = new JCO.Repository("ARAsoft", POOL_NAME);
```

This article presupposes that you already know Java. Some familiarity with JCo would be beneficial.[2]

The Role of the Repository in JCo Client Programming

JCo provides an encapsulation of the SAP Remote Function Call (RFC) protocol for Java developers. Functions to be invoked in an SAP system are called RFC-enabled Function Modules (RFMs) and are represented by an object of type *JCO.Function*. In order to create a function object you need a repository that knows the metadata of the RFM.

JCo defines an interface, *IRepository*, that must be implemented by all repository classes. There are two such repository classes available in JCo, *JCO.BasicRepository* and its subclass, *JCO.Repository*.

JCO.BasicRepository is an abstract class that provides basic capabilities such as metadata caching and persistence. The class — being abstract — cannot be instantiated and only serves as a starting point for *JCO.Repository* or custom-developed subclasses.

JCO.Repository is the standard repository used in most JCo applications. It retrieves the required RFM metadata from an SAP system dynamically and caches them to avoid additional roundtrips to SAP. **Listing 1** shows a code snippet that creates a *JCO.Repository* object with a connection pool.

Unless you develop a single-user desktop application with JCo, it is highly recommended that you always use a connection pool for the repository, although direct connections are also supported. Using a pool ensures that multiple connections are available for metadata retrieval and thus improves the performance of multi-user applications.

As you can see, the constructor for *JCO.Repository* takes two parameters. The first one is an arbitrary name assigned to the repository; the second one is the name of the connection pool to be used for the actual metadata retrieval. The name you assign to a repository is stored in the repository and can be accessed using the *getName()* method, but JCo makes no further use of the name. You can even create multiple repositories with the same name without JCo complaining, in other words the name of the repository is totally devoid of any socially redeeming value. Particularly, JCo does not provide a capability to access a repository by its name[3], which has the following implications:

[2] For a general introduction to client JCo programming see my JCo tutorial that ships with JCo itself. If you are interested in the latest version of this tutorial, send me an email.

[3] This is different from the way in which JCo treats connection pools. The *JCO.PoolManager* class allows you to access any connection pool by its name.

Listing 2: Creating a JCO.Function Object

```
JCO.Function function = createFunction("DDIF_FIELDINFO_GET");
```

Listing 3: Utility Method to Create a JCO.Function Object

```
public JCO.Function createFunction(String name)
       throws ARAsoftException {
  try {
    IFunctionTemplate ft =
      mRepository.getFunctionTemplate(name.toUpperCase());
    if (ft == null)
      return null;
    return ft.getFunction();
  }
  catch (Exception ex) {
    throw new ARAsoftException
      ("Problem retrieving JCO.Function object.", ex);
  }
}
```

- The application you are building must keep a reference to the repository in order to be able to use it later.

- Checking whether a repository already exists for a given SAP system — useful in web applications connecting to the same SAP system — cannot be achieved without a suitable utility class[4].

- Unless proper steps are taken, applications could easily create additional, unnecessary repositories. This would lead to a severe performance degradation since the individual repositories would potentially all request the same metadata from the SAP system.

[4] See the section "Repositories in Web Applications" later in this article for a discussion of such a utility class.

After you have created the repository, you can create function objects. This is best encapsulated in a utility method so that proper error handling is guaranteed. **Listing 2** shows source code that makes use of the utility method *createFunction()*. **Listing 3** contains the utility method itself.

Creating a function object is a two-step process:

1. First you access the function template (defined in interface *IFunctionTemplate*) by invoking the repository's *getFunctionTemplate()* method. A function template contains the RFM's metadata (parameters and exceptions), but not the data (the parameters themselves). JCo caches the metadata after retrieving it from SAP, so there is no need to provide your own caching scheme. Method

Figure 1 Required Authorizations for the Repository Access

In order to retrieve the necessary information from the SAP system's data dictionary, the repository needs to call a number of RFMs for which the access rights have to be granted (Authorization Object: S_RFC, ACTVT: 16, FUGR).

R/3 Release	Function Groups
since 3.1H	RFC1, SG00, SRFC, SUNI, SYST
since 4.0A	RFC1, SDIF, SG00, SRFC, SYST, SYSU, SUNI
since 4.6A	RFC1, SDIF, SG00, SRFC, SYST, SYSU
since 4.6D	RFC1, SDIFRUNTIME, SG00, SRFC, SYST, SYSU

getFunctionTemplate() returns null if the specified RFM does not exist in the SAP system. An exception is thrown if JCo has problems while trying to retrieve the metadata (possible reasons include communication problems and insufficient authorizations).

2. You then use the function template's *getFunction()* method to create a new function object. It is highly recommended that you create a fresh function object for each function invocation. Otherwise you might easily forget to clear some parameters from the previous invocation and get different results than you expect.

Authorizations and Repository Access

The userid defined for the pool that the repository object uses to access the metadata in SAP must have sufficient authorizations. The specific authorizations required in the different SAP releases are listed in the Javadoc for class *JCO.Repository*. **Figure 1** is a screenshot of the pertinent text.

Many customers are of the opinion that normal users should not have these authorizations and hence use a separate pool based on a special userid for the repository. I believe that this is an excellent idea, but would like to extend it. I would also use the repository pool for other purposes of a more technical nature like retrieving extended metadata, retrieving Helpvalues, converting values between the internal and external formats, etc.

Summary of Recommendations for Clients

☑ Only use one repository per connected SAP system.[5] Otherwise you should not be surprised when performance suffers.[6]

☑ Always create a fresh function object for each function call.

---

[5] You do not need one repository per SAP client (here in the meaning of the three-digit number SAP uses to logically partition one physical database, as opposed to a JCo Client connection object) since the metadata is client-independent.

[6] We have even seen cases where a customer created a new repository before each function call. This is a surefire way to make your hardware vendor very happy.

Listing 4: Using Extended Metadata

```
private static final String POOL_NAME = "ARAsoft";
JCoRepository mRepository;
mRepository = new JCoRepository(POOL_NAME);
JCO.Function function =
  mRepository.createFunction("BAPI_CUSTOMER_GETDETAIL2");
JCO.Field custField =
  function.getImportParameterList().getField("CUSTOMERNO");
XField xCustField = (XField) custField.getExtendedFieldMetaData();
String screenLabel = xCustField.getFieldTexts().getLabelLong();
String conversionExit = xCustField.getConversionExit();
boolean mixedCase = xCustField.isMixedCaseSupported();
```

Subclassing the JCO.Repository Class

As explained before, class *JCO.Repository* retrieves the RFM's metadata from SAP. This includes the exceptions and all parameter metadata. Parameters can be simple fields, structures, and tables, so JCo needs the metadata for each field. This is accomplished by invoking the RFM `DDIF_FIELDINFO_GET`. This RFM delivers very extensive metadata for each field, but JCo is only interested in the metadata required for the RFC protocol. The additional metadata is "lost". An application can, of course, call `DDIF_FIELDINFO_GET` itself in order to obtain the other metadata, some of which is very useful for providing state-of-the-art user interfaces (e.g., the various texts available for each field in all installed languages allow you to build a multi-lingual application without having to administer your own texts).

If you want to use this approach, you should build a component to encapsulate access to `DDIF_FIELDINFO_GET`, cache the retrieved values, and provide an easy-to-use object model for the actual application. Still, you would make additional calls to SAP! Fortunately, the designers of JCo were very clever people and foresaw the need for additional metadata.

Class *JCO.Field* has a method *getExtendedFieldMetaData()*. This method allows you to access any additional metadata stored via the *setExtendedFieldMetaData()* method available on class *JCO.Metadata*. The trick to avoid additional roundtrips to SAP for retrieving the extended metadata is to subclass *JCO.Repository* and exploit the values obtained by its call to `DDIF_FIELDINFO_GET`. Writing a subclass for *JCO.Repository* is not exactly trivial, and certainly beyond the scope of this article, but if you are a JCo expert and have a few dull weekends to kill, you might give it a try.

Listing 4 shows code utilizing a subclass of *JCO.Repository* called *JCoRepository*. This class provides its own *createFunction()* method (the source code for this is Listing 3). Class *XField* contains the extended metadata. The final three statements in Listing 4 retrieve the long screen label (SAP maintains three different sizes in its Data Dictionary), the name of the conversion exit, and whether mixed case is supported, respectively, all without additional roundtrips to SAP.

Figure 2 Extended Metadata Available in Class XField

Access Method	Description
String getName()	Returns the field name.
String getStructureName()	Returns the structure name.
String getCheckTable()	Returns the check table name.
String getConversionExit()	Returns the GUI conversion routine name.
String getDataElementName()	Returns the data element name.
String getDataTypeABAP()	Returns the ABAP 1-byte data type.
String getDataTypeDD()	Returns the Data Dictionary data type.
int getDecimals()	Returns the number of decimals.
String getDomainName()	Returns the domain name.
FieldTexts getFieldTexts()	Returns the texts for this field.
int getInternalLength()	Returns the internal field length.
int getLength()	Returns the number of characters without formatting.
int getOutputLength()	Returns the number of characters with formatting.
boolean isFixedValuesListDefined()	Checks whether this field (i.e., its domain) has a fixed values list.
boolean isMixedCaseSupported()	Checks whether this field supports mixed case, not just upper case.
boolean isSearchHelpSupported()	Checks whether this field has search help (F4) in SAPGUI.
boolean isSignedNumber()	Checks whether this field is a signed number.

For those of you who want to build your own subclass of *JCO.Repository*, **Figure 2** contains a table of the metadata properties that I would recommend.

Putting all information concerning the texts available for a field in its own class, *FieldTexts* (see **Figure 3** for its methods), makes things easier for the application programmer.

Method *getText()* is a convenience method because there is no guarantee that all texts are available in all languages for all fields. Using this method ensures that the application gets a text at all as long as any of the texts have been maintained.

Repositories in JCo Server Components

While the majority of all applications developed with JCo are client applications, more and more customers need to access Java components from ABAP, i.e., develop JCo server components. When you instantiate your server class (which must be a subclass of *JCO.Server*) you need to pass a repository object so that JCo can interpret the function call from the SAP system correctly. This repository object must contain the metadata for all functions that your server component wants to be able to process.

JCo provides two sample programs for server

Figure 3 Class FieldTexts

Access Method	Description
String getColumnHeading()	Returns the column heading.
int getColumnHeadingMaxLength()	Returns the column heading maximum length.
String getDescription()	Returns the Data Dictionary description.
String getLabelLong()	Returns the long screen label.
int getLabelLongMaxLength()	Returns the maximum length of the long screen label.
String getLabelMedium()	Returns the medium screen label.
int getLabelMediumMaxLength()	Returns the maximum length of the medium screen label.
String getLabelShort()	Returns the short screen label.
int getLabelShortMaxLength()	Returns the maximum length of the short screen label.
String getText()	Returns the first non-empty text using the following priority list: • getLabelLong() • getLabelMedium() • getLabelShort() • getColumnHeading() • getDescription() Returns an empty string if all texts are empty.

programming, Example5.java and Example7.java. Example5.java uses a hard-coded repository while Example7.java uses a client connection to SAP to retrieve the required metadata dynamically. I highly recommend the latter approach, for several reasons:

- Setting up a hard-coded repository is difficult and error-prone.

- You have two, potentially different, definitions of the function interface (one in the SAP system, and one in your server component). If the function interface changes in the SAP system, your server component will most likely fail, unless you remember to change the source code for the hard-coded repository.

- When you use an SAP system configured to support Unicode you will need to change your source code for the hard-coded repository.

Using a normal client repository with a connection pool solves all these issues, but raises two new questions:

1. What do I do if the function called from ABAP is not defined in the SAP Function Builder (transaction code SE37)?

2. How does using a client connection from my server component affect the restart behavior of my server?

The answer to the first question is that I consider it bad practice to make function calls where the function interface is not defined in the SAP Function

Builder. It is too easy to change the source code of the function call without taking into account the consequences in the server component. Adding the function definition in the Function Builder does not take very much time. Remember that you do not have to implement the function (i.e., write ABAP source code). All you do is define the parameters and exceptions.

To answer the second question, I first need to explain what happens if a server thread loses its connection to the SAP system. (Remember that you have two types of connections if you employ the approach I suggested earlier: server connections waiting for SAP to call, and client connections only used to retrieve function metadata.) JCo automatically tries to reconnect the failed server connection, starting one second after it recognizes the failure, and doubling the interval before each subsequent reconnect attempt (up to a maximum value[7]).

But what happens to our client connection pool for the repository? Does it also reconnect automatically? Well, sort of. Without going into gruesome implementation details, the answer is that it is possible, albeit not likely, that a metadata retrieval request by JCo on behalf of a call from SAP to a server will fail. Do not despair, though, there is an easy work-around for this issue: Before starting your server threads, but after the instantiation of the repository, retrieve the metadata for all functions supported by your server by calling *getFunctionTemplate()* for each function. This ensures that no further client calls to SAP will ever be required again for as long as your server is running.

Summary of Recommendations for Servers

☑ Define the interfaces of all the functions your servers implement in the SAP Function Builder.

☑ Use a client connection to SAP to dynamically retrieve the required metadata for your servers from SAP.

☑ Always retrieve the function templates of all functions available in your server at start-up time.

Repositories in Web Applications

Many customers use JCo to build HTML-based front-ends to SAP systems. A web application written in Java usually contains a mixture of Servlets and JavaServer Pages.[8] One application could use one or more Servlets. How do you ensure that you only create one repository to be shared by all Servlets? As mentioned earlier, for connection pools this is easy to accomplish since JCo contains a pool manager that controls all connection pools. **Listing 5** shows source code that first checks whether a pool with a given name already exists and, if not, creates a new one.

A similar feature for repositories is sorely lacking from JCo. Time to rectify this and build our own repository manager!

A Repository Manager for JCO.Repository

Our implementation will follow the approach SAP has used for its *JCO.PoolManager* class and employ the singleton pattern[9]. The complete source code for our repository manager, called *StandardRepositoryManager*, can be found in Appendix A on page 41. Before discussing portions of the source code, let us look at the methods offered by this class (see **Figure 4**).

[7] This value can be changed by setting JCo property jco.server.max_startup_delay.

[8] It is, of course, possible to use only Servlets or only JavaServer Pages, if you feel so inclined.

[9] For an excellent introduction to design patterns in Java, see Mark Grand's three-volume series *Patterns in Java*, published by John Wiley & Sons.

Listing 5: Creating a Connection Pool If Necessary

```
JCO.Pool pool;
pool = JCO.getClientPoolManager().getPool(POOL_NAME);
if (pool == null) {
  OrderedProperties logonProperties =
    OrderedProperties.load("/logon.properties");
  JCO.addClientPool(POOL_NAME,         // pool name
                    5,                 // maximum number of connections
                    logonProperties);  // connection properties
  pool = JCO.getClientPoolManager().getPool(POOL_NAME);
}
```

Figure 4 Class StandardRepositoryManager

Access Method	Description
static StandardRepositoryManager getSingleInstance()	Returns the singleton instance of this class.
JCO.Repository createRepository(JCO.Pool pool)	Creates a JCO.Repository object for the SAP system to which the pool is connected.
boolean existsRepository(JCO.Pool pool)	Checks whether a JCO.Repository object for the SAP system to which the pool is connected already exists.
boolean existsRepository(String systemId)	Checks whether a JCO.Repository object for the specified SAP system already exists.
JCO.Repository getRepository(JCO.Pool pool)	Returns the JCO.Repository object for the SAP system to which the pool is connected.
JCO.Repository getRepository (JCO.Pool pool, boolean createIfItDoesNotExist)	Returns the JCO.Repository object for the SAP system to which the pool is connected. If no repository exists and createIfItDoesNotExist is true, a new repository is created; otherwise an exception is thrown.
JCO.Repository getRepository(String systemId)	Returns the JCO.Repository object for the specified SAP system.
void removeRepository(JCO.Repository repository)	Removes the specified JCO.Repository object.

Listing 6: Method createRepository()

```
public synchronized JCO.Repository createRepository(JCO.Pool pool)
       throws ARAsoftException {
  JCO.Client client = null;
  try {
    client = JCO.getClient(pool.getName());
    String name = client.getAttributes().getSystemID();
    JCO.releaseClient(client);
    client = null;
    if ( items.containsKey(name) )
      throw new ARAsoftException
        ("A repository for system '" + name + "' already exists.");
    JCO.Repository repository = new JCO.Repository
      (name, pool.getName());
    items.put(name, repository);
    return repository;
  }
  catch (Exception ex) {
    throw new ARAsoftException(ex);
  }
  finally {
    if ( client != null ) {
      JCO.releaseClient(client);
    }
  }
}
```

Method *getSingleInstance()* accesses the singleton instance of the repository manager. The first time it is called, the instance is created. Subsequent calls just return the reference to the object.

Method *createRepository()* creates a new repository for the pool passed as an argument. There are two aspects in the implementation of this method (**Listing 6**) that I want to draw to your attention.

In order to avoid the creation of more than one repository per SAP system we need to assign a unique name to each repository and keep track of the names used so far. The names are maintained as keys in the *TreeMap* `items`. We use the system ID of the SAP system as the key. This system ID can be obtained by calling the *getSystemID()* method of *JCO.Attributes* (available on a *JCO.Client* object that we get from the connection pool).

Listing 7: Creating a Pool and a Repository If Necessary

```
JCO.Pool pool;
IRepository repository;
pool = JCO.getClientPoolManager().getPool(POOL_NAME);
if (pool == null) {
  OrderedProperties logonProperties =
    OrderedProperties.load("/logon.properties");
  JCO.addClientPool(POOL_NAME,         // pool name
                    5,                 // maximum number of connections
                    logonProperties);  // connection properties
  pool = JCO.getClientPoolManager().getPool(POOL_NAME);
}
repository = StandardRepositoryManager.getSingleInstance()
                                      .getRepository(pool, true);
```

The `finally` clause makes sure that the client connection is returned to the pool even if an error occurs earlier. This is extremely important since we would otherwise run out of usable client connections sooner or later.

The two flavors of *existsRepository()* allow a client program to check whether a repository already exists for a given pool or SAP system ID.

We offer three flavors of the *getRepository()* method. Normal applications will use the one with two parameters (see Figure 4). It will return an existing repository and allows the application to specify whether a new one will be created if none existed before. Using this method in all your applications will ensure that no unnecessary repositories will be created.

Method *removeRepository()* allows you to remove a repository from the repository manager.

We can now extend Listing 5 so that, in addition to the connection pool, the repository is also created if necessary (see **Listing 7**). This code would typically be used in the *init()* method of each Servlet that needs to access SAP.

A Repository Manager for JCoRepository

If we want to use a subclass of *JCO.Repository* that provides additional metadata, we need a slightly more intelligent repository manager. Class *ExtendedRepositoryManager* (see Appendix B on page 45 for the complete source code) fills this requirement. Since the texts stored for each field are language-dependent we need a different repository for each language/SAP system combination. That is, if the application is at all interested in the texts.

Figure 5 Class ExtendedRepositoryManager

Access Method	Description
static ExtendedRepositoryManager getSingleInstance()	Returns the singleton instance of this class.
JCoRepository createRepository(JCO.Pool pool)	Creates a JCoRepository object for the SAP system to which the pool is connected. An exception is thrown if a repository for this system exists — regardless of the language used.
JCoRepository createRepository (JCO.Pool pool, boolean ignoreLanguage)	Checks whether a JCO.Repository object for the SAP system to which the pool is connected already exists. If ignoreLanguage is true, an exception is thrown if any repository for this system exists. If ignoreLanguage is false, an exception is thrown if a repository for this system and the language used in the pool exists.
boolean existsRepository(JCO.Pool pool)	Checks whether a JCoRepository object for the SAP system to which the pool is connected already exists — regardless of language.
boolean existsRepository(JCO.Pool pool, boolean ignoreLanguage)	Checks whether a JCoRepository object for the SAP system to which the pool is connected — and with the language used by the pool (unless ignoreLanguage is true) — already exists.
boolean existsRepository(String systemId)	Checks whether a JCoRepository object for the specified SAP system exists — regardless of language.
boolean existsRepository(String systemId, String language)	Checks whether a JCoRepository object for the specified SAP system and language exists.

Therefore, we give the application control over whether it needs a language-specific repository or not.

Figure 5 shows the methods offered by the *ExtendedRepositoryManager* class.

The source code in Appendix B should be quite easy to understand, but I would like to point out why we need the private *getLanguage()* method shown in **Listing 8**. When connecting to SAP you do not need to specify a language key. In that case, the default system language defined by the administrator will be used. JCo does not know which language that is, so a call to the *getLanguage()* method of *JCO.Attributes* returns a string with one blank. There are several ways to determine the actual language, but the one I prefer is a call to `DDIF_FIELDINFO_GET` since this RFM exists in all ABAP-based SAP systems; in other words it works not only in R/3, but also in CRM, etc.

Conclusion

Proper use of repositories guarantees high-performance,

Figure 5 (continued)

Access Method	Description
JCoRepository getRepository (boolean ignoreLanguage, JCO.Pool pool)	Returns the JCoRepository object for the SAP system to which the pool is connected. Throws an exception if no repository at all exists for this system. Throws an exception if ignoreLanguage is false and no repository for the language used in the pool exists.
JCoRepository getRepository (boolean ignoreLanguage, JCO.Pool pool, boolean createIfItDoesNotExist)	Returns the JCoRepository object for the SAP system to which the pool is connected. If no repository exists and createIfItDoesNotExist is true, a new repository is created; otherwise an exception is thrown. If ignoreLanguage is true, the language used in the pool is ignored.
JCoRepository getRepository(JCO.Pool pool)	Returns the JCoRepository object for the SAP system to which the pool is connected. Ignores the language.
JCoRepository getRepository (JCO.Pool pool, boolean createIfItDoesNotExist)	Returns the JCoRepository object for the SAP system to which the pool is connected. If no repository exists and createIfItDoesNotExist is true, a new repository is created; otherwise an exception is thrown. Ignores the language.
JCoRepository getRepository(String systemId)	Returns the JCoRepository object for the specified SAP system. Ignores the language and throws an exception if no repository for this system exists.
void removeRepository(JCoRepository repository)	Removes the specified JCoRepository object.

Listing 8: The getLanguage() Utility Method

```
static private String getLanguage(JCO.Client client,
                                  IRepository repository)
       throws Exception {
  String lang = client.getAttributes().getLanguage();
  if ( lang == null || lang.equals("") || lang.equals(" ") ) {
    JCO.Function function =
      repository.getFunctionTemplate("DDIF_FIELDINFO_GET")
                .getFunction();
    function.getImportParameterList().setValue("DFIES", "TABNAME");
    function.getImportParameterList().setValue("LANGU", "FIELDNAME");
    client.execute(function);
    lang = function.getTableParameterList().getTable("DFIES_TAB")
                                           .getString("LANGU");
  }
  return lang;
}
```

stable applications. For all multi-user applications, a repository manager relieves the applications from having to deal with the issue of creating unnecessary repositories.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

Appendix A: Class StandardRepositoryManager

```
package de.arasoft.sap.jco;

import java.util.TreeMap;
import com.sap.mw.jco.*;
import de.arasoft.java.ARAsoftException;

/*
 * Copyright (c) 2002 ARAsoft GmbH
 * All Rights Reserved.
 */

/**
 * A singleton object that manages JCO.Repository objects.
 *
 * @author ARAsoft GmbH
 * @version 2.5
 * @since 2.5
 */

public class StandardRepositoryManager {

  static private StandardRepositoryManager repositoryManager = null;
  static private TreeMap items = null;

  protected StandardRepositoryManager() {
    items = new TreeMap();
  }

/**
 * Returns the singleton instance of this class.
 * @return The singleton instance.
 */
  static public synchronized StandardRepositoryManager getSingleInstance() {
    if ( repositoryManager == null )
      repositoryManager = new StandardRepositoryManager();
    return repositoryManager;
  }

/**
```

```
 * Creates a JCO.Repository object for the SAP system to which the pool is
 * connected.
 * Throws an exception if a repository for this system already exists.
 * @return The created repository object.
 * @param pool The JCO.Pool object.
 */
  public synchronized JCO.Repository createRepository(JCO.Pool pool)
         throws ARAsoftException {
    JCO.Client client = null;
    try {
      client = JCO.getClient(pool.getName());
      String name = client.getAttributes().getSystemID();
      JCO.releaseClient(client);
      client = null;
      if ( items.containsKey(name) )
        throw new ARAsoftException
                  ("A repository for system '" + name + "' already exists.");
      JCO.Repository repository = new JCO.Repository(name, pool.getName());
      items.put(name, repository);
      return repository;
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Checks whether a JCO.Repository object for the specified SAP system already
 * exists.
 * @return Does a repository for the specified SAP system exist?
 * @param systemId The system ID of the SAP system.
 */
  public boolean existsRepository(String systemId) {
    JCO.Repository repository = (JCO.Repository) items.get(systemId);
    return ( repository != null );
  }

/**
 * Checks whether a JCO.Repository object for the SAP system to which the pool
 * is connected already exists.
 * @return Does a repository for this system exist?
 * @param pool The JCO.Pool object.
 */
  public boolean existsRepository(JCO.Pool pool) throws ARAsoftException {
```



```
    JCO.Client client = null;
    try {
      client = JCO.getClient(pool.getName());
      String name = client.getAttributes().getSystemID();
      JCO.releaseClient(client);
      client = null;
      return this.existsRepository(name);
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Returns the JCO.Repository object for the SAP system to which the pool
 * is connected.
 * Throws an exception if no repository exists.
 * @return The repository object.
 * @param pool The JCO.Pool object.
 */
  public synchronized JCO.Repository getRepository(JCO.Pool pool)
         throws ARAsoftException {
    return this.getRepository(pool, false);
  }

/**
 * Returns the JCO.Repository object for the SAP system to which the pool is
 * connected. If no repository exists and <code>createIfItDoesNotExist</code>
 * is <code>true</code>, a new repository is created, otherwise an exception
 * is thrown.
 * @return The repository object.
 * @param pool The JCO.Pool object.
 * @param createIfItDoesNotExist Should a new repository be created if none
 *        exists?
 */
  public synchronized JCO.Repository getRepository
         (JCO.Pool pool, boolean createIfItDoesNotExist)
         throws ARAsoftException {
    JCO.Client client = null;
    try {
      client = JCO.getClient(pool.getName());
      String name = client.getAttributes().getSystemID();
      JCO.releaseClient(client);
      client = null;
```

```
      try {
        return this.getRepository(name);
      }
      catch (ARAsoftException ax) {
        if ( createIfItDoesNotExist ) {
          return this.createRepository(pool);
        } else {
          throw ax;
        }
      }
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Returns the JCO.Repository object for the specified SAP system.
 * If no repository exists an exception is thrown.
 * @return The repository object.
 * @param systemId The system ID of the SAP system.
 */
  public synchronized JCO.Repository getRepository(String systemId)
         throws ARAsoftException {
    JCO.Repository repository = (JCO.Repository) items.get(systemId);
    if ( repository == null )
      throw new ARAsoftException
                ("No repository exists for system '" + systemId + "'.");
    return repository;
  }

/**
 * Removes the specified JCO.Repository object.
 * @param repository The repository to be removed.
 */
  public synchronized void removeRepository(JCO.Repository repository) {
    String name = repository.getName();
    if ( items.containsValue(repository) ) {
      items.remove(name);
    }
  }
}
```


Appendix B: Class ExtendedRepositoryManager

```
package de.arasoft.sap.jco;

import java.util.Hashtable;
import java.util.TreeMap;
import java.util.Vector;
import com.sap.mw.jco.*;
import de.arasoft.java.ARAsoftException;
import de.arasoft.sap.jco.JCoRepository;

/*
 * Copyright (c) 2002 ARAsoft GmbH
 * All Rights Reserved.
 */

/**
 * A singleton object that manages JCoRepository objects.
 *
 * @author ARAsoft GmbH
 * @version 2.5
 * @since 2.5
 */

public class ExtendedRepositoryManager {

  static private ExtendedRepositoryManager repositoryManager = null;
  static private TreeMap items = null;
  static private Hashtable systems = null;

  static private final String EXCEPTION_1A =
    "No repository exists for system '";
  static private final String EXCEPTION_1B =
     "'.";

  protected ExtendedRepositoryManager() {
    items = new TreeMap();
    systems = new Hashtable();
  }

/**
```

```
 * Returns the singleton instance of this class.
 * @return The singleton instance.
 */
  static public synchronized ExtendedRepositoryManager getSingleInstance() {
    if (repositoryManager == null)
      repositoryManager = new ExtendedRepositoryManager();
    return repositoryManager;
  }

/**
 * Creates a JCoRepository object for the SAP system to which the pool is
 * connected. An exception is thrown if a repository for this system exists -
 * regardless of the language used.
 * @return The created repository object.
 * @param pool The JCO.Pool object.
 */
  public synchronized JCoRepository createRepository(JCO.Pool pool)
         throws ARAsoftException {
    return this.createRepository(pool, true);
  }

/**
 * Creates a JCoRepository object for the SAP system to which the pool is
 * connected. If <code>ignoreLanguage</code> is <code>true</code>, an
 * exception is thrown if any repository for this system exists.
 * If <code>ignoreLanguage</code> is <code>false</code>, an
 * exception is thrown if a repository for this system and the language used
 * in the pool exists.
 * @return The created repository object.
 * @param pool The JCO.Pool object.
 * @param ignoreLanguage Should the language be ignored?
 */
  public synchronized JCoRepository createRepository
         (JCO.Pool pool, boolean ignoreLanguage) throws ARAsoftException {
    JCO.Client client = null;
    JCoRepository repository = null;
    try {
      client = JCO.getClient(pool.getName());
      String name = client.getAttributes().getSystemID();
      if ( ignoreLanguage ) {
        if ( systems.containsKey(name) ) {
          throw new ARAsoftException
                    ("A repository for system '" + name + "' already exists.");
        }
        repository = new JCoRepository(pool);
        String lang = getLanguage(client, repository);
        JCO.releaseClient(client);
        client = null;
        Vector languages = new Vector(1);
```



```
        languages.add(lang);
        systems.put(name, languages);
        items.put(createKey(name, lang), repository);
        return repository;
      } else { // language not ignored
        if ( systems.containsKey(name) ) {
          Vector languages = (Vector) systems.get(name);
          repository = (JCoRepository) items.get
            (createKey(name, (String)languages.get(0)));
          String lang = getLanguage(client, repository);
          JCO.releaseClient(client);
          client = null;
          if ( containsLanguage(languages, lang) )
            throw new ARAsoftException
            ("A repository for system '" + name + "' and language '"
            + lang + "' already exists.");
          else {
            languages.add(lang);
            repository = new JCoRepository(pool);
            items.put(createKey(name, lang), repository);
            return repository;
          }
        } else { // no repository for this system at all
          repository = new JCoRepository(pool);
          String lang = this.getLanguage(client, repository);
          JCO.releaseClient(client);
          client = null;
          Vector languages = new Vector(1);
          languages.add(lang);
          systems.put(name, languages);
          items.put(createKey(name, lang), repository);
          return repository;
        }
      }
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Checks whether a JCoRepository object for the specified SAP system exists -
 * regardless of language.
 * @return Does a repository for this system exist?
```

```
 * @param systemId The system ID of the SAP system.
 */
  public boolean existsRepository(String systemId) {
    return systems.containsKey(systemId);
  }

/**
 * Checks whether a JCoRepository object for the specified SAP system and
 * language exists.
 * @return Does a repository for this system and language exist?
 * @param systemId The system ID of the SAP system.
 * @param language The 2-byte language identifier.
 */
  public boolean existsRepository(String systemId, String language) {
    boolean found = systems.containsKey(systemId);
    if ( ! found ) return false;
    return containsLanguage((Vector)systems.get(systemId), language);
  }

/**
 * Checks whether a JCoRepository object for the SAP system to which the pool
 * is connected already exists - regardless of language.
 * @return Does a repository for this system exist?
 * @param pool The JCO.Pool object.
 */
  public boolean existsRepository(JCO.Pool pool) throws ARAsoftException {
    return this.existsRepository(pool, true);
  }

/**
 * Checks whether a JCoRepository object for the SAP system to which the pool
 * is connected - and with the language used by the pool
 * (unless <code>ignoreLanguage</code> is <code>true</code>) - already exists.
 * @return Does a repository for this system exist?
 * @param pool The JCO.Pool object.
 * @param ignoreLanguage Should the language be ignored?
 */
  public boolean existsRepository(JCO.Pool pool, boolean ignoreLanguage)
         throws ARAsoftException {
    JCO.Client client = null;
    try {
      client = JCO.getClient(pool.getName());
      String name = client.getAttributes().getSystemID();
      boolean found = existsRepository(name);
      if ( ! found ) return false;
      if (ignoreLanguage) {
        return true;
      } else {
        JCoRepository repository = getRepository(true, pool, false);
```



```
        String lang = getLanguage(client, repository);
        return existsRepository(name, lang);
      }
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Returns the JCoRepository object for the SAP system to which the pool
 * is connected. Ignores the language.
 * Throws an exception if no repository exists.
 * @return The repository object.
 * @param pool The JCO.Pool object.
 */
  public synchronized JCoRepository getRepository(JCO.Pool pool)
         throws ARAsoftException {
    return this.getRepository(true, pool, false);
  }

/**
 * Returns the JCoRepository object for the SAP system to which the pool is
 * connected. If no repository exists and <code>createIfItDoesNotExist</code>
 * is <code>true</code>, a new repository is created, otherwise an exception
 * is thrown. Ignores the language.
 * @return The repository object.
 * @param pool The JCO.Pool object.
 * @param createIfItDoesNotExist Should a new repository be created if none
 *        exists?
 */
  public synchronized JCoRepository getRepository
         (JCO.Pool pool, boolean createIfItDoesNotExist)
         throws ARAsoftException {
    return this.getRepository(true, pool, createIfItDoesNotExist);
  }

/**
 * Returns the JCoRepository object for the SAP system to which the pool is
 * connected. Throws an exception if no repository at all exists for this
 * system. Throws an exception if <code>ignoreLanguage</code> is
 * <code>false</code> and no repository for the language used in the pool
 * exists.
 * @return The repository object.
```

```
   * @param ignoreLanguage Should the language be ignored?
   * @param pool The JCO.Pool object.
   */
    public synchronized JCoRepository getRepository
           (boolean ignoreLanguage, JCO.Pool pool) throws ARAsoftException {
      return this.getRepository(ignoreLanguage, pool, false);
    }

  /**
   * Returns the JCoRepository object for the SAP system to which the pool is
   * connected. If no repository exists and <code>createIfItDoesNotExist</code>
   * is <code>true</code>, a new repository is created, otherwise an exception
   * is thrown. If <code>ignoreLanguage</code> is <code>true</code>, the
   * language used in the pool is ignored.
   * @return The repository object.
   * @param ignoreLanguage Should the language be ignored?
   * @param pool The JCO.Pool object.
   * @param createIfItDoesNotExist Should a new repository be created if none
   *        exists?
   */
    public synchronized JCoRepository getRepository
           (boolean ignoreLanguage, JCO.Pool pool, boolean createIfItDoesNotExist)
           throws ARAsoftException {
      JCO.Client client = null;
      try {
        client = JCO.getClient(pool.getName());
        String name = client.getAttributes().getSystemID();
        boolean found = systems.containsKey(name);
        if ( found && ignoreLanguage ) {
          return this.getRepository(name);
        }
        if ( ! found ) {
          if ( createIfItDoesNotExist ) {
            return this.createRepository(pool, false);
          } else {
            throw new ARAsoftException(EXCEPTION_1A + name + EXCEPTION_1B);
          }
        }
        JCoRepository repository = this.getRepository(name);
        String lang = getLanguage(client, repository);
        Vector languages = (Vector) systems.get(name);
        found = containsLanguage(languages, lang);
        if ( found ) {
          return (JCoRepository) items.get(createKey(name, lang));
        } else if ( createIfItDoesNotExist ) {
          return this.createRepository(pool, false);
        } else {
          throw new ARAsoftException(EXCEPTION_1A + name + EXCEPTION_1B);
        }
```



```
    }
    catch (ARAsoftException ax) {
      throw ax;
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Returns the JCoRepository object for the specified SAP system. Ignores the
 * language and throws an exception if no repository for this system exists.
 * @return The repository object.
 * @param systemId The system ID of the SAP system.
 */
  public synchronized JCoRepository getRepository(String systemId)
         throws ARAsoftException {
    boolean found = systems.containsKey(systemId);
    if ( ! found )
      throw new ARAsoftException(EXCEPTION_1A + systemId + EXCEPTION_1B);
    Vector languages = (Vector)systems.get(systemId);
    return (JCoRepository) items.get
      (createKey(systemId, (String)languages.get(0)));
  }

/**
 * Removes the specified JCoRepository object.
 * @param repository The repository to be removed.
 */
  public synchronized void removeRepository(JCoRepository repository)
         throws ARAsoftException {
    JCO.Client client = null;
    String poolName = repository.getPoolNames()[0];
    try {
      client = JCO.getClient(poolName);
      String name = client.getAttributes().getSystemID();
      String lang = getLanguage(client, repository);
      if ( items.containsValue(repository) ) {
        items.remove(createKey(name, lang));
        Vector languages = (Vector)systems.get(name);
        if (languages.size() == 1) {
          systems.remove(name);
        } else {
          languages.remove(lang);
```

```
        }
      }
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

  static private boolean containsLanguage(Vector languages, String lang) {
    String s = null;
    int size = languages.size();
    for (int i = 0; i < size; i++) {
      s = (String) languages.get(i);
      if ( s.equals(lang) ) return true;
    }
    return false;
  }

  static private String createKey(String systemId, String lang) {
    return systemId + ":" + lang;
  }

  static private String getLanguage(JCO.Client client, IRepository repository)
         throws Exception {
    String lang = client.getAttributes().getLanguage();
    if ( lang == null || lang.equals("") || lang.equals(" ") ) {
      JCO.Function function =
        repository.getFunctionTemplate("DDIF_FIELDINFO_GET").getFunction();
      function.getImportParameterList().setValue("DFIES", "TABNAME");
      function.getImportParameterList().setValue("LANGU", "FIELDNAME");
      client.execute(function);
      lang = function.getTableParameterList().getTable("DFIES_TAB")
                                             .getString("LANGU");
    }
    return lang;
  }
}
```

