EMC White Paper

EMC² where information lives

DFC Operations

Umair Nauman

Abstract: This document discusses fundamental concepts regarding DFC Operations.

Date 7/30/2007

Introduction to DFC

DFC is a key part of the Documentum software platform. While the main user of DFC is other Documentum software, you can use DFC in any of the following ways:

- Access Documentum functionality from within one of your company's enterprise applications. For example, your corporate purchasing application can retrieve a contract from your Documentum system.
- Customize or extend products like Documentum Desktop or Webtop. For example, you can modify Webtop functionality to implement one of your company's business rules.
- Write a method or procedure for Content Server to execute as part of a workflow or document lifecycle.

DFC is Java based. As a result, client programs that are Java based can interface directly with DFC. DFC provides the Documentum Java-COM bridge (DJCB) to make most DFC interfaces available in Microsoft's Component Object Model (COM) environment. The DFC primary interop assembly (PIA) provides access to DFC from the Microsoft .NET environment. When application developers use DFC, it is usually within the customization model of a Documentum client, though you can also use DFC to develop the methods associated with intrinsic Content Server functionality, such as document lifecycles.

DFC runs on a Java virtual machine (JVM), which can be on:
- The machine that runs Content Server. For example, to be called from a Docbasic method as part of a workflow or document lifecycle.
- A middle-tier system. For example, on an application server to support WDK or to execute server methods.
- An end user's computer. For example, to support Documentum Desktop.

For more details about DFC please refer to the DFC User Guide. The rest of this paper will concentrate on DFC Operations.

Introduction to operations

To manipulate documents in Documentum, you must understand operations. Operations are like the document carriers you use to put small pieces of paper through the feeder of a copying machine. Operations provide interfaces and a processing environment to ensure that Documentum can handle a variety of documents and collections of documents in a standard way. You obtain an operation of the appropriate kind, place one or more documents into it, and "run it through the machine," that is, execute the operation. All of the examples in this chapter pertain only to documents, but operations apply to objects of type IDfSysObject, not just the subtype IDfDocument. For example, to check out a document, take the following steps:

1. Obtain a checkout operation.
2. Add the document to the operation.
3. Execute the operation.

Standard Documentum Functionality like a checkin operation requires several API calls in a certain order. Operations hide this complexity and may execute several API calls in the correct order. The main purpose of Operations package is to provide a standard framework for processing common repository actions. Without the facilities provided in the Operation interface, a programmer might accidentally ignore a vital step for processing certain kinds of documents. By using the Operations interfaces, you ensure that all vital steps are performed and achieve the greatest level of portability and performance, especially when it involves compound and XML documents.

The Operations package provides support for any combination of simple documents, Virtual Documents, XML Documents, compound documents, Assemblies, related documents, and even deep folder contents. Operations manage compound document linked both in the repository and in the file system. They account for and manage object relationships. They ensure adherence to client conventions such as the registration of checked out objects.

Operations provide seamless distributed support. For example, you can set the destination folder of the IDfCopyOperation to a folder in another repository, and the operation will perform as expected, as long as the user is authenticated in both repositories.

DFC carries out the behind-the-scenes tasks associated with checking out a document. For a virtual document, for example, DFC adds all of its components to the operation and ensures that links between them are still valid when it stores the documents into the checkout directory on the file system. It corrects filename conflicts, and it keeps a local record of which documents it checks out. This is only a partial description of what DFC does when you check out a document. Because of the number and complexity of the underlying tasks, DFC wraps seemingly elementary document-manipulation tasks in operations.

An IDfClientX object provides factory methods for creating operations. Once you have an IDfClientX object (say cX) and a SysObject (say doc) representing the document,
the code for the checkout looks like this:

```
// Obtain a checkout operation
IDfCheckoutOperation checkout = cX.getCheckoutOperation();
// Add the document to the checkout operation
checkout.add(doc); //This might fail and return a null
// Check the document out
checkout.execute(); //This might produce errors without
//throwing an exception
```

In a real application, you would add code to handle a null returned by the add method
or errors produced by the execute method.

Operation Entity Definitions

This section will attempt to provide a clearer definition for entities and concerns surrounding operations

Operation – the embodiment of content-centric processing. An operation acts upon one or more nodes according to specified operational parameters—for the entire operation and/or for a specific node. An operation executes one or more steps in a specified order.

- Inbound operation – an operation that applies to content entering a Documentum-controlled repository (e.g. import, checkin)

- Outbound operation – an operation that applies to content leaving a Documentum-controlled repository (e.g. export, checkout)

Node – An Operation "Node" is a "thing" that is operated upon during the operation's execute() method and is created at the point of adding an object to the Operation. The node interfaces provide access to the properties and settings to allow a programmer to parameterize the processing of the same operational steps to individual nodes. These interfaces allow the node to set properties that are particular to the corresponding Operation for that node.

A key point to notice is that for each Operation type there is a corresponding Node type. When you add objects to an operation of one type, the operation generates nodes of its corresponding type. For example, all of the objects you add to an IDfImportOperation object will ALWAYS generate one or more IDfImportNode objects. There is no way that the IDfImportOperation.add() method could return any other type than IDfImportNode (e.g. it will never return IDfExportNode, or IDfDeleteNode).

A node is an abstract representation of a sysobject or a file (i.e. the actionable essence of an operation, which contains a structure of nodes and descendants). More than one node can point to the same sysobject or file. For an import operation, node representation corresponds to a file; otherwise, it corresponds to sysobject.

Each Operation has one or more root nodes. Each node can have zero or more child nodes, and thus an Operation can incorporate multiple "graphs" of objects. The Operations account for an object appearing more than once through different paths, for example an object appearing more than once in a Checkin Operation will only actually check in once.

A single add() can create many Nodes. For example, adding once Virtual Document will create a Node for each child in the Virtual Document.

Code example of Import Node: IDfImportNode node = (IDfImportNode) operation.add(srcFileOrDir);

Step – a logical unit of execution. An operation is executed as a series of steps, and each step may have a corresponding node action.

Node action – a logical unit of execution. A node action is independently performed on each node in an operation. Some steps may not have corresponding node action (e.g. DfPreDelete). In such cases, the step will go through all the nodes and perform the necessary action.

Populator – mechanism that supports the creation of nodes against which an operation is applied. An operation contains a structure of nodes and descendants. When you obtain an operation, it has no nodes (i.e. DFC does not regard the operation itself as a node). When you use an operation's add method to add documents (or other content items) to the operation, it creates new root nodes. The add method is basically the populator for the operation. There are various populators for adding sysobjects, virtual documents, files, etc. In the current implementation, populators are passive, not active, entities.

Code example of adding documents to and operation:

```
IDfCheckinNode node = (IDfCheckinNode)operation.add(sysObj);
```

Package (and package item) – mechanism that supports more efficient ("one shot" per se) operations by supplying all the necessary information up front (versus in two phases). For example, prior to the notion of content packages and content package items, there was no mechanism to specify the attributes that one would want to set on the newly created object during import operation. As a result, user had to obtain the newly created object after the import operation had been executed, set the necessary attributes and then save the object. With content package APIs, one can create an import package (which will result in creation of import package item(s)) and specify all the attributes information up front. The information is then set on the newly created object as part of the execution of the import operation.

Content packages have affinity to operations (e.g. IDfImportContentPackage for IDfImportOperation). Once the content package is created, a file or a sysobject (depending on the type of operation) is then added to the content package. A corresponding package item is created for the added file or sysobject. One can then set all the necessary information on the content package and/or content package item(s). Once all the information is set, content package should be added to the operation. The populator for content package iterates over all the content package items in the content package and creates node(s) for the corresponding content package item(s). Operation then executes on the nodes.

- Inbound package – a package entering a Documentum-controlled repository
- Outbound package – a package leaving a Documentum-controlled repository

Edge – an abstract representation that depicts the relationship between two nodes (e.g. in the case of VDM, an edge corresponds to dmr_containment)

Monitor – mechanism that supports operation progress indication. An operation monitor provides access to operation errors as they occur and allows for abort/continue decisions in real time.

Types of operations

DFC provides operation types and corresponding nodes (to be explained in subsequent sections) for many tasks you might wish to perform on documents or, where appropriate, files or folders. The following table summarizes these.

Table 2-1. DFC operation types and nodes

Task Operation	Type Operation	Node Type
Import into a repository	IDfImportOperation	IDfImportNode
Export from a repository	IDfExportOperation	IDfExportNode
Check into a repository	IDfCheckinOperation	IDfCheckinNode
Check out of a repository	IDfCheckoutOperation	IIDfCheckoutNode
Cancel a checkout	IDfCancelCheckoutOperation	IDfCancelCheckoutNode
Delete from a repository	IDfDeleteOperation	IDfDeleteNode
Copy from one repository location to another	IDfCopyOperation	IDfCopyNode
Move from one repository location to another	IDfMoveOperation	IDfMoveNode
Validate an XML document against a DTD or Schema	IDfValidationOperation	IDfValidationNode

Setting parameters for the operation

Different operations accept different parameters to control the way they carry out their tasks. Some parameters are optional, some mandatory. **Note:** You must use the setSession method of

IDfImportOperation or IDfXMLTransformOperation to set a repository session before adding nodes to either of these types of operation. No other operation type has a setSession method.

Adding documents to the operation

An operation contains a structure of nodes and descendants. When you obtain the operation, it has no nodes (DFC does not regard the operation itself as a node). When you use the operation's add method to add documents to the operation, it creates new root nodes. The add method returns the node as an IDfOperationNode object. You must cast it to the appropriate operation node type to use any methods the type does not inherit from IDfOperationNode. **Note:** If the add method cannot create a node for the specified document, it returns a null argument. Be sure to test for this case, because it does not usually throw an exception. DFC may add additional nodes to the operation. For example, if you add a repository folder, DFC adds nodes for the documents linked to that folder, as children of the folder's node in the operation. Each node can have zero or more child nodes. If you add a virtual document, the add method creates as many descendant nodes as necessary to create an image of the virtual document's structure within the operation. You can add objects from more than one repository to an operation. You can use a variety of methods to obtain and step through all nodes of the operation. You may wish to set parameters on individual nodes differently from the way you set them on the operation.

Executing the Operation

The operations package processes the objects in an operation as a group, possibly invoking many DFC calls for each object. Operations encapsulate Documentum client conventions for registering, naming, and managing local content files. DFC executes the operation in a predefined set of steps, applying each step to all of the documents in the operation before proceeding to the next step. It processes each document in an operation only once, even if the document appears at more than one node. Once DFC has executed a step of the operation on all of the documents in the operation, it cannot execute that step again. If you wish to perform the same task again, you must construct a new operation to do so. Normally, you use the operation's execute method and let DFC proceed through the execution steps. DFC provides a limited ability for you to execute an operation in steps, so that you can perform special processing between steps. Documentum does not recommend this approach, because the number and identity of steps in an operation may change with future versions of DFC.

Processing the results

If DFC encounters an error while processing one node in an operation, it continues to process the other nodes. For example, if one object in a checkout operation is locked, the operation checks out the others. Only fatal conditions cause an operation to throw an exception. DFC catches other exceptions internally and converts them into IDfOperationError objects. The getErrors method returns an IDfList object containing those errors, or a null if there are no errors. The calling program can examine the errors, and decide whether to undo the operation, or to accept the results for those objects that did not generate errors. Once you have checked the errors you may wish to examine and further process the results of the operation.

Working with nodes

This section shows how to access the objects and results associated with the nodes of an operation. **Note:** Each operation node type (for example, IDfCheckinNode) inherits most of its methods from IDfOperationNode. The getChildren method of an IDfOperationNode object returns the first level of nodes under the given node. You can use this method recursively to step through all of the descendant nodes. Alternatively, you can use the operation's getNodes method to obtain a flat list of descendant nodes, that is, an IDfList object containing of all of its descendant nodes without the structure. These methods return nodes as objects of type IDfOperationNode, not as the specific node type (for example, IDfCheckinNode). The getId method of an IDfOperationNode object returns a unique identifier for the node, *not the object ID*

of the corresponding document. IDfOperationNode does not have a method for obtaining the object ID of the corresponding object. Each operation node type (for example, IDfCheckinNode) has its own getObjectID method. You must cast the IDfOperationNode object to a node of the specific type before obtaining the object ID.

Operations and Transactions

Operations do not use session transactions because operations:

- Support distributed operations involving multiple repositories.
- May potentially process vast numbers of objects.
- Manage non-database resources such as the system registry and the local file system.

You can undo most operations by calling an operation's abort method. The abort method is specific to each operation, but generally undoes repository actions and cleans up registry entries and local content files. Some operations (for example, delete) cannot be undone.

If you know an operation only contains objects from a single repository, and the number of objects being processed is small enough to ensure sufficient database resources, you can wrap the operation execution in a session transaction. You can also include operations in session manager transactions. Session manager transactions can include operations on objects in different repositories, but you must still pay attention to database resources. Session manager transactions are not completely atomic, because they do not use a two-phase commit.

Operation Examples

In this section I have listed the source code for various operations as a reference. You can also find these Operations Samples in the DFC API Docs.

Operations- Import (Java)

The execute method of an IDfImportOperation object imports files and directories into the repository. It creates objects as required, transfers the content to the repository, and removes local files if appropriate. If any of the nodes of the operation refer to existing objects (for example, through XML or OLE links), it imports those into the repository too.

Notes: Example call: Operations.doImport("c:/DFCDir/test.txt", "/DFCCab", session);

```
public static void doImport( String srcFileOrDir, String destFolderPath, IDfSession session)
throws DfException {
    IDfClientX clientx = new DfClientX();
    IDfImportOperation operation = clientx.getImportOperation();
    //the Import Opertion requires a session
    operation.setSession( session );

    IDfFolder folder = null;
    folder = session.getFolderByPath( destFolderPath );
    //note: the destination folder must exist- or throws nullPointerException
    if( folder == null ) {
        System.out.println("Folder or cabinet " + destFolderPath + " does not exist in the Docbase!");
        return;
```

```
    }
    operation.setDestinationFolderId( folder.getObjectId() );

    //check if file or directory on file system exists
    IDfFile myFile = clientx.getFile( srcFileOrDir );
    if( myFile.exists() == false ) {
       System.out.println("File or directory " + srcFileOrDir + " does not exist in the file system!");
       return;
    }
    //add the file or directory to the operation
    IDfImportNode node = (IDfImportNode) operation.add( srcFileOrDir );

    //can optionally use the following
    // node.setXMLApplicationName("MyXMLApplication");
    // see the docs for IDfImportNode and IDfImportOperation for more options

    //see the Operation- Execute and Check Errors sample code
    executeOperation( operation );

    //work with resulting nodes here
    IDfList myNodes = operation.getNodes();
    int iCount  = myNodes.getCount();
    System.out.println("Number of nodes after operation: " + iCount );
    for( int i = 0; i < iCount; ++i ) {
       IDfImportNode aNode = (IDfImportNode) myNodes.get(i);
       System.out.print("Object ID " + i + ": " + aNode.getNewObjectId().toString() + "  " );
       System.out.println("Object name: " + aNode.getNewObjectName() );
    }
  }
```

Operations- Checkout (Java)

Checks out a simple or virtual document
Notes: Also applies to XML documents that are Virtual Documents (chunked)
Example call: Operations.doCheckout("/DFCCab/test", session);

```
  public static void doCheckout( String strPath, IDfSession session ) throws DfException {

    IDfClientX clientx = new DfClientX();
    IDfCheckoutOperation operation = clientx.getCheckoutOperation();

    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strPath );
    if( sysObj == null ) {
       System.out.println("Object " + strPath + " can not be found.");
       return;
    }

   if (sysObj.isCheckedOut() == true) {
        System.out.println("Object " + strPath + " is already checked out.");
       return;
    }

    IDfCheckoutNode node;
    if( sysObj.isVirtualDocument() ) {
       IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
```

```
        node = (IDfCheckoutNode)operation.add(vDoc);
    } else {
        node = (IDfCheckoutNode)operation.add(sysObj);
    }

    //see sample: Operations- Execute and Check Errors
    executeOperation( operation );

    System.out.println( "Checkout file path: " + node.getFilePath() );
}
```

Please note that for traversing virtual documents, you must navigate by using a Virtual Document instance and not the IDfOperation node. Please refer to DFC javadocs for code samples of traversing Virtual documents.

Operations- Checkin (Java)

Checks in a document, virtual document or XML document using IDfCheckinOperation
Notes: Example call: Operations.doCheckin("/DFCCab/test", session);

```
public static void doCheckin( String strPath, IDfSession session ) throws DfException {
    IDfClientX clientx = new DfClientX();
    IDfCheckinOperation operation = clientx.getCheckinOperation();

    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strPath );
    if( sysObj == null ) {
        System.out.println("Object " + strPath + " can not be found.");
        return;
    }

    if (sysObj.isCheckedOut() == false) {
         System.out.println("Object " + strPath + " is not checked out.");
        return;
    }

    IDfCheckinNode node;
    if( sysObj.isVirtualDocument() ) {
        IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
        node = (IDfCheckinNode)operation.add(vDoc);
    } else {
        node = (IDfCheckinNode)operation.add(sysObj);
    }

    //Other options for setCheckinVersion: VERSION_NOT_SET, NEXT_MAJOR,
    //  NEXT_MINOR, BRANCH_VERSION
    operation.setCheckinVersion(IDfCheckinOperation.SAME_VERSION);

    //see sample: Operations- Execute and Check Errors
    executeOperation( operation );

    System.out.println("Old object id: " + node.getObjectId().toString() );
    System.out.println("New object id: " + node.getNewObjectId().toString() );
}
```

Operations- CancelCheckout (Java)

The execute method of an IDfCancelCheckoutOperation object cancels the checkout of documents by releasing locks, deleting local files if appropriate, and removing registry entries. This also applies to XML documents that are Virtual Documents (chunked)

If the operation's add method receives a virtual document as an argument, it also adds all of the document's descendants (determined by applying the applicable binding rules), creating a separate operation node for each.

Example call: Operations.doCancelCheckout("/DFCCab/test", session);

```
public static void doCancelCheckout(  String strPath, IDfSession session  ) throws DfException {
    IDfClientX clientx = new DfClientX();
    IDfCancelCheckoutOperation operation = clientx.getCancelCheckoutOperation();
    operation.setKeepLocalFile( false );

    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strPath );
    if( sysObj == null ) {
        System.out.println("Object " + strPath + " can not be found.");
        return;
    }

    if (sysObj.isCheckedOut() == false) {
         System.out.println("Object " + strPath + " is not checked out.");
        return;
    }

    IDfCancelCheckoutNode node;
    if( sysObj.isVirtualDocument() ) {
        IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
        node = (IDfCancelCheckoutNode)operation.add(vDoc);
    } else {
        node = (IDfCancelCheckoutNode)operation.add(sysObj);
    }

    //see sample: Operations- Execute and Check Errors
    executeOperation( operation );

    System.out.println( "Canceled checkout for object " + strPath );
}
```

Operations- Export Single Object (Java)

The execute method of an IDfExportOperation object creates copies of documents on the local file system. If the operation's add method receives a virtual document as an argument, it also adds all of the document's descendants (determined by applying the applicable binding rules), creating a separate node for each. This also applies to XML documents that are Virtual Documents (chunked).You can not export a folder using IDfExportOperation

See the sample: Export Operation- Multiple Objects for a workaround
Example call: Operations.doExport("/DFCCab/test", "c:/DFCDir", "test_exported", "crtext", session);

```
public static void doExport(  String strPath, String destDir,
String exportName, String exportFormat, IDfSession session )
throws DfException {
    IDfClientX clientx = new DfClientX();
    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strPath );
    if( sysObj == null ) {
        System.out.println("Object " + strPath + " can not be found.");
        return;
    }

    sysObj.setObjectName(exportName);

    IDfExportOperation operation = clientx.getExportOperation();

    operation.setDestinationDirectory( destDir );

    IDfExportNode node = (IDfExportNode)operation.add( sysObj );

    node.setFormat( exportFormat );

    //see sample: Operations- Execute and Check Errors
    executeOperation( operation );

    System.out.println( "exported file path: " + node.getFilePath() );
}
```

Operations- Export Multiple Objects (Java)

Exports the documents in a folder to the file system
Notes: Example call: Operations.doExportMultipleObjects("/DFCCab", "c:/DFCDir", session);

```
public static void doExportMultipleObjects(  String strFolderPath, String destDir, IDfSession session )
throws DfException {
    IDfClientX clientx = new DfClientX();
    IDfFolder folder = null;
    folder = session.getFolderByPath( strFolderPath );
    if( folder == null ) {
        System.out.println("Folder or cabinet " + strFolderPath + " does not exist in the Docbase!");
        return;
    }
    IDfExportOperation operation = clientx.getExportOperation();

    operation.setDestinationDirectory( destDir );

    //if the export directory doesn't exist, make it
    //be sure to "import java.io.*;"
    File f = new File(destDir);
    f.mkdir();

    String qualification = "select * from dm_document where FOLDER(ID('" + folder.getObjectId() + "'))";

    IDfCollection col = null; //create a collection for the images to be exported
    try {

        IDfQuery q = clientx.getQuery(); //Create query object
```

```
        q.setDQL(qualification); //Give it the query
        col = q.execute(session, IDfQuery.DF_READ_QUERY); //Execute synchronously

        while (col.next()) {
            String name = col.getString("object_name");
            IDfFile destFile = clientx.getFile( destDir + name );
            //check if the file has been exported (exists) to avoid another export
            if (! destFile.exists()){
                //Create an IDfSysObject representing the object in the collection
                String id = col.getString("r_object_id");
                IDfId idObj = clientx.getId(id);
                IDfSysObject myObj = (IDfSysObject) session.getObject(idObj);
                // add the IDfSysObject to the operation
                IDfExportNode node = (IDfExportNode)operation.add(myObj);
            }
        }

        //see sample: Operations- Execute and Check Errors
        executeOperation( operation );

        System.out.println("The objects in " + strFolderPath + " have been exported to " + destDir);
    }
    finally {
        if (col!= null)
            col.close();
    }
}
```

Operations- Copy (Java)

The execute method of an IDfCopyOperation object copies the *current versions* of documents or folders from one repository location to another. If the operation's add method receives a virtual document as an argument, it also adds all of the document's descendants (determined by applying the applicable binding rules), creating a separate node of the operation for each. If the add method receives a folder (unless you override this default behavior), it also adds all documents and folders linked to that folder(To change this behavior, use setDeepFolders(false);). This continues recursively until the entire hierarchy of documents and subfolders under the original folder is part of the operation. The execute method replicates this hierarchy at the target location.

Example call: Operations.doCopy("/DFCCab/test" , "/DFCCab/copy" , session);

```
public static void doCopy( String strSourcePath, String strDestFolderPath, IDfSession session )
throws DfException {
    IDfClientX clientx = new DfClientX();
    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strSourcePath );
    if( sysObj == null ) {
        System.out.println("Source object " + strSourcePath + " can not be found.");
        return;
    }

    IDfFolder destFolder = null;
    destFolder = session.getFolderByPath( strDestFolderPath );
    if( destFolder == null ) {
```

```
        System.out.println("Destination folder or cabinet " + strDestFolderPath + " does not exist in the Docbase!");
        return;
    }
    //obtain an IDfCopyOperation and set parameters
    IDfCopyOperation operation = clientx.getCopyOperation();
    operation.setDestinationFolderId( destFolder.getObjectId() );
    operation.setDeepFolders( true );

    //add the appropriate object to the operation
    if( sysObj.isVirtualDocument() ) {
        IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
        IDfCopyNode node = (IDfCopyNode)operation.add(vDoc);
    } else {
        IDfCopyNode node = (IDfCopyNode)operation.add(sysObj);
    }

    //see the Operation- Execute and Check Errors sample code
    executeOperation( operation );

    System.out.println("Copy operation complete.");
}
```

Operations- Move (Java)

The execute method of an IDfMoveOperation object moves the *current versions* of documents or folders from one repository location to another by unlinking them from the source location and linking them to the destination. Versions other than the current version remain linked to the original location. If the operation's add method receives a virtual document as an argument, it also adds all of the document's descendants (determined by applying the applicable binding rules), creating a separate node for each. If the add method receives a folder (unless you override this default behavior), it adds all documents and folders linked to that folder. This continues recursively until the entire hierarchy of documents and subfolders under the original folder is part of the operation. The execute method links this hierarchy to the target location.

Notes: Must pass in the source folder (strSourceFolder), source object(s) (strSourceObject) and the destination folder (strDestFolder). To move a folder, set strSourceObject to the folder and strSourceFolder to the folder's parent. If a folder is passed into this method (as strSourceObject), any subfolders will also be moved.

Example call: Operations.doMove("/DFCCab" , "DFCCab/test" , "/DFCCab/dest" , session);

```
public static void doMove( String strSourceFolder, String strSourceObject, String strDestFolder, IDfSession session )
throws DfException {
    IDfClientX clientx = new DfClientX();

    IDfFolder srcFolder = session.getFolderByPath( strSourceFolder );
    if( srcFolder == null ) {
        System.out.println("Source folder or cabinet " + strSourceFolder + " does not exist in the Docbase!");
        return;
    }

    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strSourceObject );
    if( sysObj == null ) {
```

```
        System.out.println("Source object " + strSourceObject + " can not be found.");
        return;
    }

    IDfFolder destFolder = session.getFolderByPath( strDestFolder );
    if( destFolder == null ) {
        System.out.println("Destination folder or cabinet " + strDestFolder + " does not exist in the Docbase!");
        return;
    }

    //obtain an IDfMoveOperation and set parameters
    IDfMoveOperation operation = clientx.getMoveOperation();
    operation.setSourceFolderId( srcFolder.getObjectId() );
    operation.setDestinationFolderId( destFolder.getObjectId() );

    //add the appropriate object to the operation
    if( sysObj.isVirtualDocument() ) {
        IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
        IDfMoveNode node = (IDfMoveNode)operation.add(vDoc);
    } else {
        IDfMoveNode node = (IDfMoveNode)operation.add(sysObj);
    }

    //see the Operation- Execute and Check Errors sample code
    executeOperation( operation );

    System.out.println("Move operation complete.");
}
```

Operations- Delete (Java)

The execute method of an IDfDeleteOperation object removes documents and folders from the repository. If the operation's add method receives a virtual document as an argument, it also adds all of the document's descendants (determined by applying the applicable binding rules), creating a separate node for each. You can use the enableDeepDeleteVirtualDocumentsInFolders method of IDfDeleteOperation to override this behavior.

Notes: Example call: Operations.doDelete("DFCCab/test" , session);

```
public static void doDelete(String strObject, IDfSession session)
throws DfException {
    IDfClientX clientx = new DfClientX();

    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strObject );
    if( sysObj == null ) {
        System.out.println("Object " + strObject + " can not be found.");
        return;
    }
    //create an IDfDeleteOperation object and set parameters
    IDfDeleteOperation operation = clientx.getDeleteOperation();
    //other options for setVersionDeletionPolicy: ALL_VERSIONS, UNUSED_VERSIONS
    operation.setVersionDeletionPolicy( IDfDeleteOperation.SELECTED_VERSIONS );

    if( sysObj.isVirtualDocument() ) {
```

```
        IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
        IDfDeleteNode node = (IDfDeleteNode)operation.add(vDoc);
    } else {
        IDfDeleteNode node = (IDfDeleteNode)operation.add(sysObj);
    }

    //see the Operation- Execute and Check Errors sample code
    executeOperation( operation );

    System.out.println("Deletion of " + strObject + " complete.");
}
```

Operations- Validate

```
public static void doValidationFromDocbase( String strObject , String tempDir, IDfSession session )
throws DfException {
    IDfClientX clientx = new DfClientX();

    IDfSysObject sysObj = (IDfSysObject) session.getObjectByPath( strObject );
    if( sysObj == null ) {
        System.out.println("Object " + strObject + " can not be found.");
        return;
    }
    //obtain an IDfValidationOperation and set parameters
    IDfValidationOperation operation = clientx.getValidationOperation();
    //specify a directory where the temporary file will be placed
    //  during validation of the XML file
    operation.setDestinationDirectory( tempDir );

    //add the appropriate object to the operation
    if( sysObj.isVirtualDocument() ) {
        IDfVirtualDocument vDoc = sysObj.asVirtualDocument( "CURRENT", false );
        IDfValidationNode node = (IDfValidationNode)operation.add(vDoc);
    } else {
        IDfValidationNode node = (IDfValidationNode)operation.add(sysObj);
    }

    //see the Operation- Execute and Check Errors sample code
    executeOperation( operation );

    System.out.println("Validation operation complete.");
}
```

Setting up the operation

Use the object's setSession method to specify a repository session and the object's setDestinationFolderId method to specify the repository cabinet or folder into which the operation should import documents. You *must* set the session before adding files to the operation. You can set the destination folder, either on the operation or on each node. The node setting overrides the operation setting. If you set neither, DFC uses its default destination folder. You can add an IDfFile object or specify a file system path. You can also specify whether to keep the file on the file system (the default choice) or delete it after the operation is successful. If you add a file system directory to the operation, DFC imports all files in that directory and proceeds recursively to add each subdirectory to the operation. The resulting repository folder hierarchy mirrors the

file system directory hierarchy. You can also control version labels, object names, object types and formats of the imported objects.

Support Notes About Operations

Following is a list of Support Notes you can refer to for more information on DFC Operations:

Support Note 7639 - How do I use IdfOperations to Checkout / Checkin / CancelCheckout a document?
Support Note 19060 - Is there an example of how to delete a (Virtual) document using the IDfDeleteOperation?
Support Note 7195 - How do I use the IDfOperations interface to import documents?
Support Note 15277 - Why do you get an error when running the DFC package "com.documentum.operations" class using Borland JBuilder?
Support Note 24035 - What is an example of exporting a rendition using IDfExportOperation?
Support Note 15454 - Does the IDfImportOperation method scan for OLE links?
Support Note 30117 - Is there an example of how to use the delete operation to delete an object using DFC ?
Support Note 4624 - What functionalities are provided with the IDfOperations Class?
Support Note 16535 - Can you export a folder recursively using the DFC export operation?
Support Note 20901 - Is there a workaround to bug 50524 - Cannot populate the transform operation with a virtual document?

Questionnaire:

1) How would you define an Operation?
2) List some of the entity components that exist within an operations
3) What is the advantage of using Operations?
4) List a few of the common DFC Operations.

References

DFC 6.0 Combined “Functional-Design” Specification Operations Customization Model by Craig Randall

DFC Development Guide

DFC Advanced Training Guide