EMC® Documentum® Foundation Classes

Version 6

Development Guide

P/N 300-005-247

EMC Corporation
Corporate Headquarters:
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Published August 2007

Table of Contents

List of Figures

List of Tables

Preface

This manual describes EMC Documentum Foundation Classes (DFC). It provides overview and summary information.

For an introduction to other developer resources, refer to DFC developer support, page 14.

Intended audience

This manual is for programmers who understand how to use Java and are generally familiar with the principles of object oriented design.

Revision History

The following changes have been made to this document.

Revision History

Revision Date	Description
August 2007	Initial publication

Chapter 1

Getting Started with DFC

This chapter introduces DFC. It contains the following major sections:

What Is 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.

  For example, the procedure that runs when you promote an XML document might apply a transformation to it and start a workflow to subject the transformed document to a predefined business process.

You can view Documentum functionality as having the following elements:

Repositories	One or more places where you keep the content and associated metadata of your organization's information. The metadata resides in a relational database, and the content resides in various storage elements.
Content Server	Software that manages, protects, and imposes an object oriented structure on the information in repositories. It provides intrinsic tools for managing the lifecycles of that information and automating processes for manipulating it.
Client programs	Software that provides interfaces between Content Server and end users. The most common clients run on application servers (for example, Webtop) or on personal computers (for example, Desktop).
End Users	People who control, contribute, or use your organization's information. They use a browser to access client programs running on application servers, or they use the integral user interface of a client program running on their personal computer.

In this view of Documentum functionality, Documentum Foundation Classes (DFC) lies between Content Server and clients. Documentum Foundation Services are the primary client interface to the Documentum platform. Documentum Foundation Classes are used for server-side business logic and customization.

DFC is Java based. As a result, client programs that are Java based can interface directly with DFC.

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.

In the Java application server environment, Documentum client software rests on the foundation provided by the Web Development Kit (WDK). In the Microsoft personal computer environment, many customers use Documentum Desktop, an integration with Windows Explorer. Each of these clients has a customization model that allows you to modify the user interface and also implement some business logic. However, the principal tool for adding custom business logic to a Documentum system is to use the Business Object Framework (BOF).

BOF enables you to embody business rules and patterns in reusable elements, called modules. The most important modules for application developers are type based objects (TBOs) and service based objects (SBOs). BOF makes it possible to extend some of DFC's implementation classes. As a result, you can introduce new functionality in such a way that unmodified existing programs begin immediately to deliver the new functionality. Aspect modules are similar to TBOs, but enable you to attach properties and behavior on an instance-by-instance basis, independent of the target object's type.

The *Documentum Content Server Fundamentals* manual provides a conceptual explanation of the capabilities of Content Server and how they work. DFC provides a framework for accessing those capabilities. Using this framework makes your code much more likely to survive future architectural changes in the Documentum system.

Where Is DFC?

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 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.

For client machines, Documentum 6 now provides Documentum Foundation Services (DFS) as the primary support for applications communicating with the Documentum platform.

Note: Refer to the DFC release notes for the supported versions of the JVM. These can change from one minor release to the next.

The *DFC Installation Guide* describes the locations of files that DFC installs. The config directory contains several files that are important to DFC's operation.

Programming with DFC

This chapter provides an overview of the resources available to application developers to help them use DFC. It also provides an overview of DFC programming patterns.

DFC documentation set

The following documents pertain to DFC or to closely related subjects

- *Documentum System Migration Guide*

  Refer to the migration guide if you are upgrading from an earlier version of DFC.
- *DFC Installation Guide*

  This guide explains how to install DFC. It includes information about preparing the environment for DFC.
- *DFC Development Guide*

  A programmers guide to using DFC to access Documentum functionality. This guide focuses on concepts and overviews. Refer to the Javadocs for details.
- *DFC Online Reference (Javadocs)*

  The Javadocs provide detailed reference information about the classes and interfaces that make up DFC. They contain automatically generated information about method signatures and interclass relationships, explanatory comments provided by developers, and a large number of code samples.

- *Documentum Content Server Fundamentals*

  A conceptual description of the capabilities of Documentum Content Server and how to use them. The material in this manual is a key to understanding most DFC interfaces.
- *XML Application Development Guide*

  A description of the XML-related capabilities of the Documentum server, and an explanation of how to design applications that exploit those capabilities.

DFC developer support

Application developers using DFC can find additional help in the following places:

- CustomerNet

  The CustomerNet website (http://softwaresupport.emc.com/support) is an extensive resource that every developer should take some time to explore and become familiar with.

  The Developer section contains tips, sample code, downloads, a component exchange, and other resources for application developers using DFC, WDK, and other tools.

  The Support section provides a knowledge base, support notes, technical alerts and papers, support forums, and access to individual cases. The DFC support forum is a good place to go when you need answers.
- Yahoo! groups

  There are several Yahoo! groups dedicated to aspects of Documentum software. The ones focused on DFC, WDK, and XML might be especially interesting to application developers who use DFC.

Interfaces

Because DFC is large and complex, and because its underlying implementation is subject to change, you should use DFC's public interfaces.

Tip: DFC provides factory methods to instantiate objects that implement specified DFC interfaces. If you bypass these methods to instantiate implementation classes directly, your programs may fail to work properly, because the factory methods sometimes do more than simply instantiate the default implementation class. For most DFC programming, the only implementation classes you should instantiate directly are DfClientX and the exception classes (DfException and its subclasses).

DFC does not generally support direct access to, replacement of, or extension of its implementation classes. The principal exception to these rules is the *Business Object Framework* (BOF). For more information about BOF, refer to Chapter 5, Using the Business Object Framework (BOF).

Client/Server model

The Documentum architecture generally follows the client/server model. DFC-based programs are client programs, even if they run on the same machine as a Documentum server. DFC encapsulates its client functionality in the IDfClient interface, which serves as the entry point for DFC code. IDfClient handles basic details of connecting to Documentum servers.

You obtain an IDfClient object by calling the static method DfClientX.getLocalClient().

An IDfSession object represents a connection with the Documentum server and provides services related to that session. DFC programmers create new Documentum objects or obtain references to existing Documentum objects through the methods of IDfSession.

To get a session, ~~firs~~ create an IDfSessionManager by calling IDfClient.newSessionManager(). Next, get the session fromt he session manager using the procedure described in the sections and session managers below. For more information about sessions, refer to .

IDfPersistentObject

An IDfPersistentObject corresponds to a persistent object in a repository. With DFC you usually don't create objects directly. Instead, you obtain objects by calling factory methods that have IDfPersistentObject as their return type.

Caution: If the return value of a factory method has type IDfPersistentObject, you may cast it to an appropriate interface (for example, IDfDocument if the returned object implements that interface). Do not cast it to an implementation class (for example, DfDocument). Doing so produces a ClassCastException.

Processing a repository object

The following general procedure may help to clarify the DFC approach. Don't worry if you don't completely understand the references to the interface hierarchy. Subsequent sections deal with that subject.

To process a repository object:

1. Obtain an IDfClientX object. For example, execute the following Java code:

   ```
   IDfClientX cx = new DfClientX();
   ```

2. Obtain an IDfClient object by calling the getLocalClient method of the IDfClientX object. For example, execute the following Java code:

```
IDfClient c = cx.getLocalClient();
```

The IDfClient object must reside in the same process as the Documentum client library, DMCL.

3. Obtain a session manager by calling the newSessionManager method of the IDfClient object. For example, execute the following Java code:

```
IDfSessionManager sm = c.newSessionManager();
```

4. Use the session manager to obtain a session with the repository, that is, a reference to an object that implements the IDfSession interface. For example, execute the following Java code:

```
IDfSession s = sm.getSession();
```

Refer to for information about the difference between the getSession and newSession methods of IDfSessionManager.

5. If you do not have a reference to the Documentum object, call an IDfSession method (for example, newObject or getObjectByQualification) to create an object or to obtain a reference to an existing object.
6. Use routines of the operations package to manipulate the object, that is, to check it out, check it in, and so forth. For simplicity, the example below does not use the operations package. (Refer to Chapter 4, Working with Document Operations for examples that use operations).
7. Release the session.

Example 1-1. Processing a repository object

The following fragment from a Java program that uses DFC illustrates this procedure.

```
IDfClientX cx = new DfClientX();             //Step 1

 IDfClient client = cx.getLocalClient();        //Step 2

 IDfSessionManager sMgr = client.newSessionManager();   //Step 3
 IDfLoginInfo loginInfo = clientX.getLoginInfo();
 loginInfo.setUser( "Mary" );
 loginInfo.setPassword( "ganDalF" );
 loginInfo.setDomain( "" );
 sMgr.setIdentity( strRepositoryName, loginInfo );

 IDfSession session = sMgr.getSession( strRepoName );   //Step 4



 try {
   IDfDocument document =
     (IDfDocument) session.newObject( "dm_document" ); //Step 5

   document.setObjectName( "Report on Wizards" );    //Step 6
   document.save();

 }
 finally {
   sMgr.release( session );                //Step 7
 }
```

Steps 1 through 4 obtain an IDfSession object, which encapsulates a session for this application program with the specified repository.

The following example shows how to obtain an IDfClient object in Visual Basic:

```
Dim myclient  As IDfClient           'Steps 1-2
Dim myclientx As new DfClientX
Set myclient  =  myclientX.getLocalClient
```

Step 5 creates an IDfDocument object. The return type of the newObject method is IDfPersistentObject. You must cast the returned object to IDfDocument in order to use methods that are specific to documents.

Step 6 of the example code sets the document object name and saves it.

Note that the return type of the newObject method is IDfPersistentObject. The program explicitly casts the return value to IDfDocument, then uses the object's save method, which IDfDocument inherits from IDfPersistentObject. This is an example of interface inheritance, which is an important part of DFC programming. The interfaces that correspond to repository types mimic the repository type hierarchy.

Step 7 releases the session, that is, places it back under the control of the session manager, sMgr. The session manager will most likely return the same session the next time the application calls sMgr.getSession.

Most DFC methods report errors by throwing a DfException object. Java code like that in the above example normally appears within a try/catch/finally block, with an error handler in the catch block.

Tip: When writing code that calls DFC, it is a best practiceto include a finally block to ensure that you release storage and sessions.

Using dfc.properties to configure DFC

You can adjust some DFC behaviors. This section describes the dfc.properties file, which contains properties compatible with the java.util.Properties class.

The dfc.properties file enables you to set preferences for how DFC handles certain choices in the course of its execution. The accompanying dfcfull.properties file contains documentation of all recognized properties and their default settings. Leave dfcfull.properties intact, and copy parts that you want to use into the dfc.properties file. The following sections describe the most commonly used groups of properties. Refer to the dfcfull.properties file for a complete list.

The DFC installer creates a simple dfc.properties file and places it in the classpath. Other EMC installers for products that bundle DFC also create dfc.properties file in the appropriate classpath. At a minimum, the dfc.properties file must include the following entries:

dfc.docbroker.host[0]

dfc.globalregistry.repository

dfc.globalregistry.username

dfc.globalregistry.password

BOF and global registry settings

All Documentum installations must designate a global registry to centralize information and functionality. Refer to Global registry, page 92 for information about global registries.

Connecting to the global registry

The dfc.properties file contains the following properties that are mandatory for using a global registry.

- dfc.bof.registry.repository

  The name of the repository. The repository must project to a connection broker that DFC has access to.
- dfc.bof.registry.username

  The user name part of the credentials that DFC uses to access the global registry. Refer to Global registry user, page 92 for information about how to create this user.
- dfc.bof.registry.password

  The password part of the credentials that DFC uses to access the global registry. The DFC installer encrypts the password if you supply it. If you want to encrypt the password yourself, use the following instruction at a command prompt:

  ```
  java com.documentum.fc.tools.RegistryPasswordUtils password
  ```

The dfc.properties file also provides an optional property to resist attempts to obtain unauthorized access to the global registry. For example, the entry

```
dfc.bof.registry.connect.attempt.interval=60
```

sets the minimum interval between connection attempts to the default value of 60 seconds.

Performance tradeoffs

Based on the needs of your organization, you can use property settings to make choices that affect performance and reliability. For example, preloading provides protection against a situation in which the global registry becomes unavailable. On the other hand, preloading increases startup time. If you want to turn off preloading, you can do so with the following setting in dfc.properties:

```
dfc.bof.registry.preload.enabled=false
```

You can also adjust the amount of time DFC relies on cached information before checking for consistency between the local cache and the global registry. For example, the entry

```
dfc.bof.cacheconsistency.interval=60
```

sets that interval to the default value of 60 seconds. Because global registry information tends to be relatively static, you might be able to check lees frequently in a production environment. On the other hand, you might want to check more frequently in a development environment. The check is inexpensive. If nothing has changed, the check consists of looking at one vstamp object. Refer to the *Content Server Object Reference* for information about vstamp objects.

Diagnostic settings

DFC provides a number of properties that facilitate diagnosing and solving problems.

Diagnostic mode

DFC can run in diagnostic mode. You can cause this to happen by including the following setting in dfc.properties:

```
dfc.resources.diagnostics.enabled=T
```

The set of problems that diagnostic mode can help you correct can change without notice. Here are some examples of issues detected by diagnostic mode.:

- Session leaks
- Collection leaks

DFC catches the leaks at garbage collection time. If it finds an unreleased session or an unclosed collection, it places an appropriate message in the log.

Configuring docbrokers

You must set the repeating property dfc.docbroker.host, one entry per docbroker. For example,

```
dfc.docbroker.host[0]=docbroker1.yourcompany.com
dfc.docbroker.host[1]=docbroker2.yourcompany.com
```

These settings are described in the *dfcfull.properties* file.

dfc.data.dir

The dfc.data.dir setting identifies the directory used by DFC to store files. By default, it is a folder relative to the current working directory of the process running DFC. You can set this to another value.

Tracing options

DFC has extensive tracing support. Trace files can be found in a directory called *logs*under dfc.data.dir. For simple tracing, add the following line to dfc.properties:

```
dfc.tracing.enable=true
```

That will trace DFC entry calls, return values and parameters.

For more extensive tracing information, add the following lines to dfc.properties.

```
dfc.tracing.enable=true
dfc.tracing.verbose=true
dfc.tracing.include_rpcs=true
```

Thsi will include more details and RPCs sent to the server. It is a good idea to start with the simple trace, because the verbose trace produces much more output to scan and sort through.

XML processing options

DFC's XML processing is largely controlled by configuration files that define XML applications. Two properties provide additional options. Refer to the *XML Application Development Guide* for information about working with content in XML format.

Search options

DFC supports the search capabilities of Enterprise Content Integration Services (ECIS) with a set of properties. The ECIS installer sets some of these. Most of the others specify diagnostic options or performance tradeoffs.

Storage policy options

Some properties control the way DFC applies storage policy rules. These properties do not change storage policies. They provide diagnostic support and performance tradeoffs.

Performance tradeoffs

Several properties enable you to make tradeoffs between performance and the frequency with which DFC executes certain maintenance tasks.

DFC caches the contents of properties files such as dfc.properties or dbor.properties. If you change the contents of a properties file, the new value does not take effect until DFC rereads that file. The dfc.config.timeout property specifies the interval between checks. The default value is 1 second.

DFC periodically reclaims unused resources. The dfc.housekeeping.cleanup.interval property specifies the interval between cleanups. The default value is 7 days.

Some properties described in the BOF and global registry settings, page 18 and Search options, page 20 sections also provide performance tradeoffs.

Registry emulation

DFC uses the dfc.registry.mode property to keep track of whether to use a file, rather than the Windows registry, to store certain settings.

The DFC installer sets this property to *file* for Unix systems and *registry* for Windows systems.

You can set the property to *file* for a Windows system. This is helpful in environments in which you want to control access to the Windows registry.

Setting the property to *file* is incompatible with Documentum Desktop. If you use Documentum Desktop, do not set dfc.registry.mode to *file*.

Client file system locations

Several properties record the file system locations of files and directories that DFC uses. The DFC installer sets the values of these properties based upon information that you provide. There is normally no need to modify them after installation.

Using DFC logging

DFC provides diagnostic logging. The logging is controlled by a log4j configuration file, which is customarily named log4j.properties. The dmcl installer as well as other EMC installers place a default version of this file in the same directory where the dfc.properties file is created. Though you generally will not need to change this file, you can customize your logging by consulting the log4j documentation.

In past releases, DFC used the DMCL library to communicate with the server and provided support for integrating DMCL logging and DFC logging. Since DMCL is no longer used, the features to integrate its tracing into the DFC log are no longer needed.

Using DFC from application programs

You can use Java to interface directly to DFC. On a Windows system you can also use the Microsoft component object model (COM). From the .NET environment, you can use the primary interop assembly (PIA) supplied with DFC. The Developer section of the CustomerNet website contains an extensive ASP.NET example.

Using DFC from ASP or ASP.NET requires you to set appropriately high permission levels on some folders for the anonymous user account. Support notes 23274 and 24234 in the Support section of the CustomerNet website and the ASP.NET sample in the Developer section provide more information about this.

The DFC installer installs dfc.dll and, for backward compatibility, dfc.tlb. Always import dfc.dll rather than dfc.tlb into your COM programs.

The most convenient way to access DFC on a Windows system depends on the programming language.

Java

From Java, add dctm.jar to your classpath. This file contains a manifest, listing all files that your Java execution environment needs access to. The javac compiler does not recognize the contents of the manifest, so for compilation you must ensure that the compiler has access to dfc.jar. This file contains most Java classes and interfaces that you need to access directly. In some cases you may have to give the compiler access to other files described in the manifest. In your Java source code, import the classes and interfaces you want to use.

Ensure that the classpath points to the config directory.

Packages

DFC comprises a number of packages, that is, sets of related classes and interfaces.

- The names of DFC Java classes begin with Df (for example, DfCollectionX).
- Names of interfaces begin with IDf (for example, IDfSessionManager).

Interfaces expose DFC's public methods. Each interface contains a set of related methods. The Javadocs describe each package and its purpose.

Note:

- The com.documentum.operations package and the IDfSysObject interface in the com.documentum.fc.client package have some methods for the same basic tasks (for example, checkin, checkout). In these cases, the IDfSysObject methods are mostly for internal use and for supporting legacy applications. The methods in the operations package perform the corresponding tasks at a higher level. For example, they keep track of client-side files and implement Content Server XML functionality.
- The IDfClientX is the correct interface for accessing factory methods (all of its get*Xxx* methods, except for those dealing with the DFC version or trace levels).

The DFC interfaces form a hierarchy; some derive methods and constants from others. Use the Tree link from the home page of the DFC online reference (see DFC online reference documentation, page 23) to examine the interface hierarchy. Click any interface to go to its definition.

Each interface inherits the methods and constants of the interfaces above it in the hierarchy. For example, IDfPersistentObject has a save method. IDfSysObject is below IDfPersistentObject in the hierarchy, so it inherits the save method. You can call the save method of an object of type IDfSysObject.

DFC online reference documentation

The public API for DFC is documented in the Javadocs that ship with the product. You have the option of deploying the Javadocs during DFC installation, or you can download the Javadocs from the EMC Developer Connection web site (http://developer.emc.com — search for “DFC Javadocs”). Direct access to undocumented classes or interfaces is not supported.

Chapter 2

Sessions and Session Managers

This chapter describes how to get, use, and release sessions, which enable your application to connect to a repository and access repository objects.

Note: If you are programming in the WDK environment, be sure to refer to Managing Sessions in *Web Development Kit Development Guide* for information on session management techniques and methods specific to WDK.

This chapter contains the following major sections:

Sessions

To do any work in a repository, you must first get a session on the repository. A session (IDfSession) maintains a connection to a repository, and gives access to objects in the repository for a specific logical user whose credentials are authenticated before the session can connect to the repository. The IDfSession interface provides a large number of methods for examining and modifying the session itself, the repository and its objects, as well as for using transactions (refer to IDfSession in the javadoc for a complete reference).

Sharable and Private Sessions

A sharable session can be shared by multiple users, threads, or applications. A private session is a session that is not and will not be shared. To get a shared session, use IDfSessionManager.getSession. To get a private session, use IDfSessionManager.newSession.

Session Managers

A session manager (IDfSessionManager) manages sessions for a single user on one or more repositories. You create a session manager using the DfClient.newSessionManager factory method.

The session manager serves as a factory for generating new IDfSession objects using the IDfSessionManager.newSession method. Immediately after using the session to do work in the repository, you should release the session using the IDfSessionManager.release method in a *finally* clause. The session initially remains available to be reclaimed by session manager instance that released it, and subsequently will be placed in a connection pool where it can be shared.

The IDfSessionManager.getSession method checks for an available shared session, and if one is available uses it instead of creating a new session. This makes for efficient use of content server connections, which are an extremely expensive resource, in a web programming environment where a large number of sessions are required.

Getting session managers and sessions

The following sections describe how to instantiate a session manager, set its identities, and use it to get and release sessions.

Instantiating a Session Manager

The following sample method instantiates and returns a session manager object without setting any identities.

Figure 1. Instantiating a session manager without identities

```
import com.documentum.com.DfClientX;
import com.documentum.fc.client.IDfClient;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.common.DfException;
.  .  .
/**
```

```
 * Creates a IDfSessionManager with no prepopulated identities
 */
public static IDfSessionManager getSessionManager() throws Exception
{
  // Create a client object using a factory method in DfClientX.

  DfClientX clientx = new DfClientX();
  IDfClient client = clientx.getLocalClient();

  // Call a factory method to create the session manager.

  IDfSessionManager sessionMgr = client.newSessionManager();
  return sessionMgr;
}
```

In this example, you directly instantiate a DfClientX object. Based on that object, you are able to use factory methods to create the other objects required to interact with the repository.

Setting Session Manager Identities

To set a session manager identity, encapsulate a set of user credentials in an IDfLoginInfo object and pass this with the repository name to the IDfSessionManager.setIdentity method. In simple cases, where the session manager will be limited to providing sessions for a single repository, or where the login credentials for the user is the same in all repositories, you can set a single identity to IDfLoginInfo.ALL_DOCBASES (= *). This causes the session manager to map any repository name for which there is no specific identity defined to a default set of login credentials.

```
/**
 * Creates a simplest-case IDfSessionManager
 * The user in this case is assumed to have the same login
 * credentials in any available repository
 */
public static IDfSessionManager getSessionManager
  (String userName, String password) throws Exception
{
  // create a client object using a factory method in DfClientX

  DfClientX clientx = new DfClientX();
  IDfClient client = clientx.getLocalClient();

  // call a factory method to create the session manager

  IDfSessionManager sessionMgr = client.newSessionManager();
    // create an IDfLoginInfo object and set its fields
  IDfLoginInfo loginInfo = clientx.getLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(password);

  // set single identity for all docbases
  sessionMgr.setIdentity(IDfSessionManager.ALL_DOCBASES, loginInfo);
  return sessionMgr;
}
```

If the session manager has multiple identities, you can add these lazily, as sessions are requested. The following method adds an identity to a session manager, stored in the session manager referred to by the Java instance variable sessionMgr. If there is already an identity set for the repository name, setIdentity will throw a DfServiceException. To allow your method to overwrite existing identities, you can check for the identity (using hasIdentity) and clear it (using clearIdentity) before calling setIdentity.

```
public void addIdentity
  (String repository, String userName, String password) throws DfServiceException
{
  // create an IDfLoginInfo object and set its fields

  IDfLoginInfo loginInfo = this.clientx.getLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(password);

  if (sessionMgr.hasIdentity(repository))
  {
  sessionMgr.clearIdentity(repository);
  }
  sessionMgr.setIdentity(repository, loginInfo);
}
```

Note that setIdentity does not validate the repository name nor authenticate the user credentials. This normally isn't done until the application requests a session using the getSession or newSession method; however, you can authenticate the credentials stored in the identity without requesting a session using the IDfSessionManager.authenticate method. The authenticate method, like getSession and newSession, uses an identity stored in the session manager object, and throws an exception if the user does not have access to the requested repository.

Getting and releasing sessions

To get and assume ownership of a managed session, call IDfSessionManager.getSession (to get a sharable session) or newSession (to get a private session), passing a repository name as a String. If there is no identity set for the repository, getSession throws a DfIdentityException. If a user name and password stored in the identity fail to authenticate, getSession will throw a DfAuthentication exception.

To release a session, call IDfSessionManager.release(), passing it the session reference. You should release a session as soon as the work for which it is immediately required is complete, and you should release the session in a finally block to ensure that it gets released in the event of an error or exception. This discipline will help you avoid problems with session leaks, in which sessions are created and remain open. It is safer and much more efficient to release sessions as soon as they are no longer required and get new sessions as needed than to store sessions for later use.

It is important to note that this pattern for releasing sessions applies only when the session was obtained using a factory method of IDfSessionManager.

```
public static void demoSessionGetRelease
  (String repository, String userName, String password) throws DfException
{
```

```
  IDfSession mySession = null;

// Get a session manager with a single identity.

  DfClientX clientx = new DfClientX();
  IDfClient client = clientx.getLocalClient();
  mySessMgr = client.newSessionManager();
  IDfLoginInfo logininfo = clientx.getLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(password);
  mySessMgr.setIdentity(repository, loginInfo);

// Get a session using a factory method of session manager.

  IDfSession mySession = mySessMgr.getSession(repository);
  try
  {
/*
 * Insert code that performs tasks in the repository.
 */
  }
  finally
  {
    mySessMgr.release(mySession);
  }
}
```

Some legacy applications might still be getting sessions from IDfClient directly, which is now discouraged in favor of the method above. In those cases, you should call IDfSession.disconnect.

When you can and cannot release a managed session

You can only release a managed session that was obtained using a factory method of the session manager; that is IDfSessionManager.getSession or IDfSessionManager.newSession. Getting a session in this way implies ownership, and confers responsibility for releasing the session.

If you get a reference to an existing session, which might for example be stored as a data member of a typed object, no ownership is implied, and you cannot release the session. This would be the case if you obtained the session using IDfTypedObject.getSession.

The following snippet demonstrates these two cases:

```
// session is owned
IDfSession session = sessionManager.getSession("docbase");
IDfSysObject object = session.getObject(objectId);
mySbo.doSomething(object);
sessionManager.release(session);

public class MySbo
{
  private void doSomething( IDfSysObject object )
  {
  // session is not owned
  IDfSession session = object.getSession();
```

```
  IDfQuery query = new DfClientX().getQuery();
  query.setDQL("select r_object_id from dm_relation where ...");
  IDfCollection co = query.execute(session, IDfQuery.DF_READ_QUERY );
  ...
  }
}
```

Note that the session instantiated using the session manager factory method is released as soon as the calling code has finished using it. The session obtained by object.getSession, which is in a sense only borrowed, is not released, and in fact cannot be released in this method.

Sessions can be released only once

Once a session is released, you cannot release or disconnect it again using the same session reference. The following code will throw a runtime exception:

```
IDfSession session = sessionManager.getSession("docbase");
sessionManager.release(session);
sessionManager.release(session); // throws runtime exception
```

Sessions cannot be used once released

Once you have released a session, you cannot use the session reference again to do anything with the session (such as getting an object).

```
IDfSession session = sessionManager.getSession("docbase");
sessionManager.release(session);
session.getObject(objectId); // throws runtime exception
```

Objects disconnected from sessions

There may be cases when it is preferable to release a session, but store the object(s) retrieved through the session for later use. DFC 6 handles this for you transparently. If you attempt to do something that requires a session with an object after the session with which the object was obtained has been released, DFC will automatically reopen a session for the object. You can get a reference to this session by calling object.getSession.

Users of earlier versions of DFC should note that you no longer need to call setSessionManager to explicitly disconnect objects, nor is there a need to use beginClientControl/endClientControl to temporarily turn off management of a session.

Related sessions (subconnections)

It is sometimes necessary to get a session to another repository based on an existing session. For example, you may be writing a method that takes an object and opens a session on another repository using the user credentials that were used to obtain the object. There are two recommended approaches to solving this problem.

The first is to get a session using the session manager associated with the original session. This second session is a peer of the original session, and your code is responsible for releasing the both sessions.

```
IDfSession peerSession =
  session.getSessionManager().getSession(repository2Name);
try
{
  doSomethingUsingRepository2(peerSession);
}
finally
{
  session.getSessionManager().releaseSession(peerSession);
}
```

A second approach is to use a related session (subconnection), obtained by calling IDfSession.getRelatedSession. The lifetime of the related session will be dependent on the lifetime of the original session; you cannot explicitly release it.

```
IDfSession relatedSession = session.getRelatedSession(repository2Name);
doSomethingUsingRepository2(relatedSession);
```

Both of these techniques allow you to make use of identities stored in the session manager.

Users of earlier versions of DFC should note that using setDocbaseScope for creating subconnections is no longer recommended. Use one of the preceding techniques instead.

Original vs. object sessions

The original session is the session used by an application to obtain an object.

The object session is a session to the object's repository that DFC used internally to get the object.

Generally, the original session and object session are the same.

IDfTypedObject.getOriginalSession() returns the original session.

IDfTypedObject.getObjectSession() returns the object session.

IDfTypedObject.getSession() is provided for compatibility purposes and returns the original session. It is equivalent to IDfTypedObject.getObjectSession().

Transactions

DFC supports transactions at the session manager level and at the session level. A transaction at the session manager level includes operations on any sessions obtained by a thread using IDfSessionManager.newSession() or IDfSessionManager.getSession after the transaction is started (See IDfSessionManager.beginTransaction() in the DFC Javadoc) and before it completes the transaction (see IDfSessionManager.commitTransaction() and IDfSessionManager.abortTransaction()).

A transaction at the session level includes operations on the session that occur after the transaction begins (see IDfSession.beginTrans()) and occur before it completes (see IDfSession.commitTrans() and IDfSession.abortTrans()). Previous versions of DFC did not support calling beginTrans() on a session obtained from a session manager. This restriction has been removed. The code below shows how a TBO can use a session-level transaction.

```
public class MyTBO
{
  protected void doSave() throws DfException
  {
    boolean txStartedHere = false;
    if ( !getObjectSession().isTransactionActive() )
    {
      getObjectSession().beginTrans();
      txStartedHere = true;
    }
    try
    {
      doSomething();        // Do something that requires transactions
      if ( txStartedHere )
        getObjectSession().commitTrans();
    }
    finally
    {
      if ( txStartedHere && getObjectSession().isTransactionActive())
      getObjectSession().abortTrans();
    }
  }
}
```

Configuring sessions using IDfSessionManagerConfig

You can configure common session settings stored in the session configuration objects using the IDfSessionManagerConfig interface. You can get an object of this type using the IDfSessionManager.getConfig method. You can set attributes related to locale, time zone, dynamic groups, and application codes. Generally the settings that you apply will be applied to future sessions, not to sessions that exist at the time the settings are applied. The following snippet demonstrates the use IDfSessionManagerConfig:

```
// get session manager
IDfClient client = new DfClientX().getLocalClient();
IDfSessionManager sm = client.newSessionManager();

// configure session manager
// settings common among future sessions
sm.getConfig().setLocale("en_US");
sm.getConfig().addApplicationCode("finance");

// continue setting up session manager
IDfLoginInfo li = new DfClientX().getLoginInfo();
li.setUser(user);
li.setPassword(password);
sm.setIdentity("bank", li);

// now you can get sessions and continue processing
IDfSession session = sessionManager.getSession("docbase");
```

Getting sessions using login tickets

A login ticket is a token string that you can pass in place of a password to obtain a repository session. You generate a login ticket using methods of an IDfSession object. There are two main use cases to consider.

The first use is to provide additional sessions for a user who already has an authenticated session. This technique is typically employed when a web application that already has a Documentum session needs to build a link to a WDK-based application. This enables a user who is already authenticated to link to the second application without going through an additional login screen. The ticket in this case is typically embedded in a URL. For documentation of this technique, see Ticketed Login in the *Web Development Kit Development Guide.*

The second use is to allow a session authenticated for a superuser to grant other users access to the repository. This strategy can be used in a workflow method, in which the method has to perform a task on the part of a user without requesting the user password. It is also used in JEE principal authentication support, which allows a single login to the Web server and the Content Server. For information on how to use this technique in WDK applications, see J2EE Principal Authentication in *Web Development Kit Development Guide.* For information on building support for principal authentication in custom (non-WDK) web applications, see Principal authentication support, page 35.

For further information on login ticket behavior and server configuration dependencies, see Login Tickets in *Content Server Fundamentals.* For information on related server configuration settings see Configuring Login Tickets in *Content Server Administrators Guide.*

Methods for getting login tickets

The IDfSession interface defines three methods for generating login tickets: getLoginTicket, getLoginTicketForUser, and getLoginTicketEx.

- The getLoginTicket method does not require that the session used to obtain the ticket be a superuser, and it generates a ticket that can be used only by the same user who owned the session that requested the ticket.
- The getLoginTicketEx requires that the session used to obtain a ticket be a superuser, and it can be used to generate a ticket for any user. It has parameters that can be used to supplement or override server settings: specifically the scope of the ticket (whether its use is restricted to a specific server or repository, or whether it can be used globally); the time in minutes that the ticket will remain valid after it is created; whether the ticket is for a single use; and if so, the name of the server that it is restricted to.
- The getLoginTicketForUser requires that the session used to obtain a ticket belong to a superuser, and it can be used to generate a ticket for any user. It uses default server settings for ticket scope, ticket expiration time, and whether the ticket is for single use.

Generating a login ticket using a superuser account

The IDfSession.getLoginTicketEx method allow you to obtain tickets that can be used to log in any user using a superuser account. The following sample assumes that you have already instantiated a session manager (IDfSessionManager adminSessionMgr), and set an identity in adminSessionMgr for a repository using credentials of a superuser account on that repository. The sample then obtains a session for the repository from adminSessionMgr, and returns a login ticket. For information on instantiating session managers and setting identities, see Getting session managers and sessions, page 26.

```
/**
 * Obtains a login ticket for userName from a superuser session
 *
 */
public String dispenseTicket(String repository, String userName)
  throws DfException
{

// This assumes we already have a session manager (IDfSessionManager)
// with an identity set for repository with superuser credentials.
  session = adminSessionMgr.getSession(repository);
  try
  {
    String ticket = session.getLoginTicketForUser(userName);
    return ticket;
  }
  finally
  {
    adminSessionMgr.release(session);
```

```
  }
}
```

To get a session for the user using this login ticket, you pass the ticket in place of the user's password when setting the identity for the user's session manager. The following sample assumes that you have already instantiated a session manager for the user.

```
public IDfSession getSessionWithTicket
  (String repository, String userName) throws DfException
{
  // get a ticket using the preceding sample method

  String ticket = dispenseTicket(repository, userName);

  // set an identity in the user session manager using the
  // ticket in place of the password

  DfClientX clientx = new DfClientX();
  IDfLoginInfo loginInfo = clientx.getLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(ticket);

  // if an identity for this repository already exists, replace it silently

  if (userSessionMgr.hasIdentity(repository))
  {
    userSessionMgr.clearIdentity(repository);
  }
  userSessionMgr.setIdentity(repository, loginInfo);

  // get the session an return it

  IDfSession sess = userSessionMgr.getSession(repository);
  return sess;
}
```

Principal authentication support

Java Enterprise Edition (JEE) principal authentication allows you to use the authentication mechanism of a JEE application server to authenticate users of a web application. However, a separate mechanism is required to obtain a session for the user on the repository, once the user is authenticated by the application server. WDK provides built-in support for this, so the information presented in this section is useful primarily for developers of custom web applications not derived from WDK. For more information on using principal authentication in WDK-based applications, see J2EE Principal Authentication in *Web Development Kit Development Guide.*

Implementing principal support in your custom application

To implement principal support, you must provide your own implementation of the IDfPrincipalSupport interface and, optionally, of the IDfTrustManager interface. The sole method to implement in IDfPrincipalSupport is getSession, which takes as arguments the repository and principal names, and returns an IDfSession. The purpose of this method is to obtain a session for the principal (that is, the user) on the specified repository using a login ticket generated using the login credentials of a trusted authenticator (who is a superuser).

```
public IDfSession getSession
  (String docbaseName,String principalName) throws DfPrincipalException
```

The intent of the IDfTrustManager interface is to obtain login credentials for the trusted authenticator from a secure data source and return them in an IDfLoginInfo object to be used internally by the IDfPrincipalSupport object. This is done in the IDfTrustManager.getTrustCredential method:

```
public IDfLoginInfo getTrustCredential(String docbaseName)
```

To obtain sessions using DFC principal support, use the IDfClient.setPrincipalSupport method, passing it an instance of your IDfPrincipalSupport implementation class. This changes the behavior of any session manager generated from the IDfClient, so that it will delegate the task of obtaining the session to the IDfPrincipalSupport.getSession method. You then set the principal name in the session manager using the setPrincipalName method. Thereafter, when a session is requested from this session manager instance, it will use the principal support object getSession method to obtain a session for the principal on a specified repository.

The following sample demonstrates the basic principal support mechanism.

```
public static void demoPrincipalIdentity(String repository, String principalName)
  throws Exception
{
// This assumes that your trust manager implementation
// gets and stores the superuser credentials when it is constructed.
  IDfTrustManager trustManager = new YourTrustManager();

// Initialize the client.
  IDfClientX clientX = new DfClientX();
  IDfClient localClient = clientX.getLocalClient();

// Set principal support on the client. This delegates the task of
// getting sessions to the IDfPrincipalSupport object.
  IDfPrincipalSupport principalSupport =
    new YourPrincipalSupport(localClient, trustManager);
  localClient.setPrincipalSupport(principalSupport);

// Create session manager and set principal.
  IDfSessionManager sessMgr = localClient.newSessionManager();
  sessMgr.setPrincipalName(principalName)(;

// Session is obtained for the principal, not for the authenticator.
  IDfSession sess = sessMgr.getSession(repository);

  try
```

```
  {
    System.out.println("Got session: " + sess.getSessionId());
    System.out.println("Username: " + sess.getLoginInfo().getUser());
  }
// Release the session in a finally clause.
  finally
  {
    sessMgr.release(sess);
  }
}
```

Default classes for demonstrating principal support implementation

DFC includes a default implementation of IDfPrincipalSupport, as well as of the supporting interface IDfTrustManager, which together demonstrate a design pattern that you can use in building your own principal support implementation. Direct use of these classes is not supported, because they do not provide security for the trusted authenticator password, and because they may change in future releases.

The default implementation of IDfPrincipalSupport, DfDefaultPrincipalSupport, gets the session by generating a login ticket for the principal using the credentials of a trusted authenticator. The task of obtaining the credentials for the authenticator is managed by another object, of type IDfTrustManager. The IDfTrustManager.getTrustCredential method returns an IDfLoginInfo object containing the credentials of the trusted authenticator, who must be a superuser. DfDefaultPrincipalSupport, which is passed an instance of the IDfTrustManager implementation class, obtains the trust credentials from the trust manager and uses them to request a login ticket for the principal.

The default implementation of IDfTrustManager, DfDefaultTrustManager, has overloaded constructors that get the authenticator credentials from either a properties file, or from a Java Properties object passed directly to the constructor. These DfDefaultTrustManager constructors do not provide any security for the authenticator password, so it is critical that you do provide your own implementation of IDfTrustManager, and obtain the credentials in a manner that meets your application's security requirements.

Note: Source code for these classes (which should be used as templates only) is available on the developer network: http://developer.emc.com/developer/samplecode.htm.

Maintaining state in a session manager

You can cause the session manager to maintain the state of a repository object. The setSessionManager method of IDfTypedObject accomplishes this. This method copies the state of the object from the

session to the session manager, so that disconnecting the session no longer makes references to the object invalid.

Caution: Use setSessionManager sparingly. It is an expensive operation. If you find yourself using it often, try to find a more efficient algorithm.

However, using setSessionManager is preferable to using the begin/end client control mechanism (refer to the Javadocs for details) to prevent the session manager from disconnecting a session.

Chapter 3

Creating a Test Application

The primary use of DFC is to add business logic to your applications. The presentation layer is built using the Web Development Kit, most often via customization of the Webtop interface. Building a custom UI on top of DFC requires a great deal of work, and will largely recreate effort that has already been done for you.

While you should not create a completely new interface for your users, it can be helpful to have a small application that you can use to work with the API directly and see the results. With that in mind, here is a rudimentary interface class that will enable you to add and test behavior using the Operation API.

These examples were created with Oracle JDeveloper, and feature its idiosyncratic ways of building the UI. You can use any IDE you prefer, using this example as a guideline.

This chapter contains the following sections:

The TutorialSessionManager class

The TutorialSessionManager class provides convenience methods for creating and managing repository sessions.

Figure 2. TutorialSessionManager.java

```
package dfctestenvironment;

import com.documentum.com.DfClientX;

import com.documentum.fc.client.IDfClient;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;

import com.documentum.fc.common.DfException;
```

```
import com.documentum.fc.common.IDfLoginInfo;

public class TutorialSessionManager {

// Member variables used to store session and user information.
    private IDfSessionManager m_sessionMgr;
    private String m_repository;
    private String m_userName;
    private String m_password;

// Constructor
    public TutorialSessionManager(String rep, String user, String pword) {
        try {

// Populate member variables.
            m_repository = rep;
            m_userName = user;
            m_password = pword;

// Call the createSessionManager method.
            m_sessionMgr = createSessionManager();
        }
        catch (Exception e) {
            System.out.println("An exception has been thrown: " + e);
        }
    }

    private IDfSessionManager createSessionManager() throws Exception {

// The only class we instantiate directly is DfClientX.
        DfClientX clientx = new DfClientX();

// Most objects are created using factory methods in interfaces.
// Create a client based on the DfClientX object.
        IDfClient client = clientx.getLocalClient();

// Create a session manager based on the local client.
        IDfSessionManager sMgr = client.newSessionManager();

// Set the user information in the login information variable.
        IDfLoginInfo loginInfo = clientx.getLoginInfo();
        loginInfo.setUser(m_userName);
        loginInfo.setPassword(m_password);

// Set the identity of the session manager object based on the repository
// name and login information.
        sMgr.setIdentity(m_repository, loginInfo);

// Return the populated session manager to the calling class. The session
// manager object now has the required information to connect to the
// repository, but is not actively connected.
        return sMgr;
    }

// Request an active connection to the repository.
    public IDfSession getSession() throws DfException {
        return m_sessionMgr.getSession(m_repository);
```

```
    }

// Release an active connection to the repository for reuse.
    public void releaseSession(IDfSession session) {
        m_sessionMgr.release(session);
    }
}
```

The DfcTestFrame class

DfcTestFrame displays the controls used to set arguments, execute commands, and view results. In this example, it displays a single command button, used to list the content of a directory in the repository.

Figure 3. Code listing — DfcTestFrame.java

```
package dfctutorialenvironment;

import com.documentum.fc.client.IDfCollection;
import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfTypedObject;

import java.awt.Button;
import java.awt.Frame;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.Label;
import java.awt.List;
import java.awt.Rectangle;
import java.awt.SystemColor;
import java.awt.TextField;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.StringTokenizer;
import java.util.Vector;



public class DfcTestFrame extends Frame {

// Generate UI elements

    private Label label_results = new Label();
    private TextField textField_arguments = new TextField();
    private List list_id = new List();
    private Button button_directory = new Button();
    private GridBagLayout gridBagLayout1 = new GridBagLayout();
```

```
// This Vector is used to store the list of internal document IDs
// returned by the directory query.
    private Vector m_fileIDs = new Vector();

    public DfcTestFrame() {
        try {
            jbInit();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

// Initialize UI components

    private void jbInit() throws Exception {
        this.setLayout(gridBagLayout1);
        this.setTitle( "DFC Test Frame" );
        this.setBackground( SystemColor.control );
        this.setBounds(new Rectangle(10, 10, 660, 500));

        textField_arguments.setText("Arguments go here.");
        label_results.setText("Messages appear here.");
        button_directory.setLabel("Directory");
        button_directory.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        button_directory_actionPerformed(e);
                    }
                });
        this.add(textField_arguments,
           new GridBagConstraints(
                0, 0, 3, 1, 1.0, 0.0,
                GridBagConstraints.WEST,
                GridBagConstraints.HORIZONTAL,
                new Insets(5, 10, 0, 15), 527, 0)
            );
        this.add(list_id,
            new GridBagConstraints(
                 0, 2, 3, 1, 1.0, 1.0,
                 GridBagConstraints.CENTER,
                 GridBagConstraints.BOTH,
                 new Insets(80, 10, 0, 15), 372, 165)
            );
        this.add(button_directory,
            new GridBagConstraints(
                 0, 1, 1, 1, 0.0, 0.0,
                 GridBagConstraints.CENTER,
                 GridBagConstraints.NONE,
                 new Insets(10, 10, 0, 0), 14, 8)
            );
        this.add(label_results,
            new GridBagConstraints(
                0, 3, 2, 1, 0.0, 0.0,
                GridBagConstraints.WEST,
                GridBagConstraints.NONE,
                new Insets(0, 10, 7, 0), 507, 11)
            );
    }
```

```
// Handler for the Directory button. To use the button, enter the arguments
// repository_name, user_name, password, directory_path in the arguments field.

 private void button_directory_actionPerformed(ActionEvent e) {

 // Get the arguments, trim whitespace, and assign variable values.

     String temp = textField_arguments.getText();
     StringTokenizer st = new StringTokenizer(temp, ",");
     String repository = st.nextToken().trim();
     String userName = st.nextToken().trim();
     String password = st.nextToken().trim();
     String directory = st.nextToken().trim();

// Create an empty session.
     IDfSession mySession = null;

// Create a TutorSessionManager object.
     TutorialSessionManager mySessMgr = null;

     String docId = "";
     String docName = "";

     try {

         label_results.setText("Accessing directory information....");

// Populate the custom session manager object.

         mySessMgr = new TutorialSessionManager(repository, userName, password);

// Get a session and assign it to the mySession variable.
         mySession = mySessMgr.getSession();

// Create a folder object based on the directory_path argument.
         IDfFolder myFolder = mySession.getFolderByPath(directory);

// Get the contents of the folder as a collection.
         IDfCollection folderList = myFolder.getContents(null);

// Empty the file IDs member variable.
         m_fileIDs.clear();

// Empty the file list display.
         list_id.removeAll();

// Cycle through the collection getting the object ID and adding it to the
// m_listIDs Vector. Get the object name and add it to the file list control.
         while (folderList.next())
         {
             IDfTypedObject doc = folderList.getTypedObject();
             docId = doc.getString("r_object_id");
             docName = doc.getString("object_name");
             list_id.add(docName);
             m_fileIDs.addElement(docId);
         }
```

```
// Display a success message for 1 or more objects, or a different message if
// the command succeeds but there are no files in the directory.
         if (list_id.getItemCount()>0) {
             label_results.setText("Directory query complete.");
         }
         else {
             label_results.setText("Directory query complete. No items found.");
         }
     }

// Handle any exceptions.
     catch (Exception ex) {
         label_results.setText("Exception has been thrown: " + ex);
         ex.printStackTrace();
     }

// Always, always, release the session in the finally clause.
     finally {
         mySessMgr.releaseSession(mySession);
     }
 }

}
```

The DfcTutorialApplication class

This is the runnable class with the main() method, used to start and stop the application.

Figure 4. DfcTutorialApplication.java

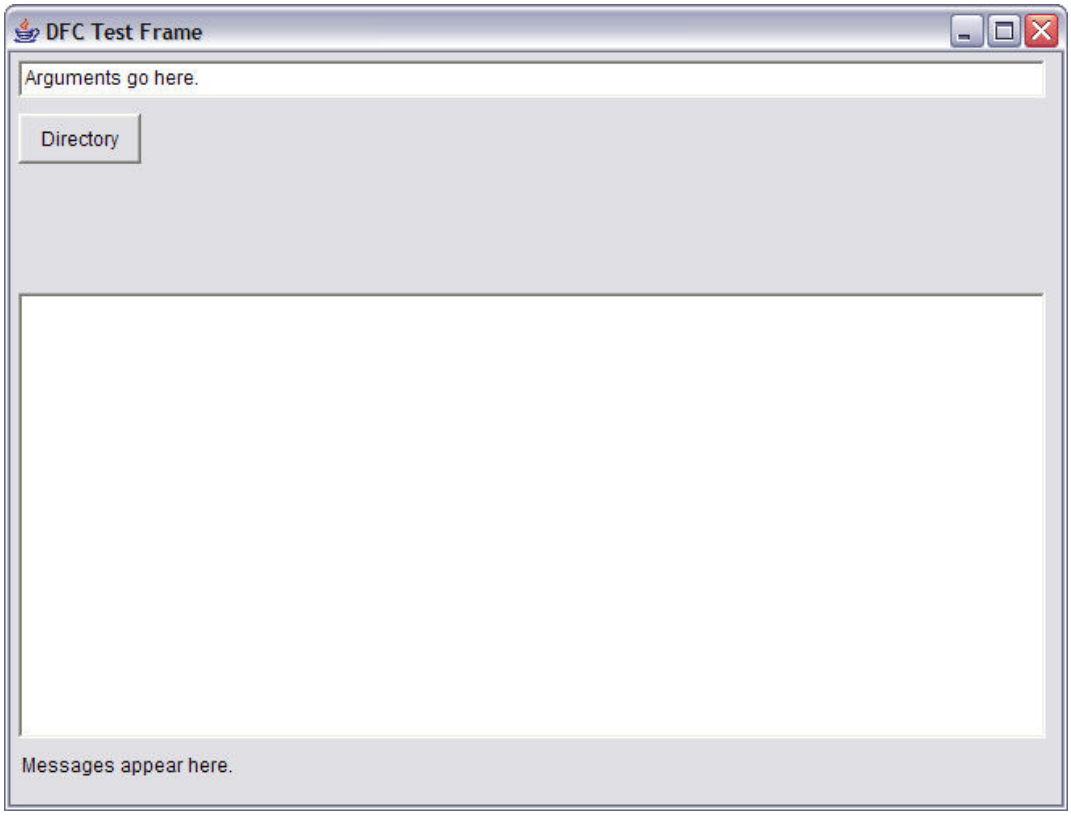
```
package dfctutorialenvironment;

import java.awt.Dimension;
import java.awt.Frame;
import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.UIManager;

public class DfcTutorialApplication {
    public DfcTutorialApplication() {
        Frame frame = new DfcTestFrame();
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        Dimension frameSize = frame.getSize();
        if (frameSize.height > screenSize.height) {
            frameSize.height = screenSize.height;
        }
        if (frameSize.width > screenSize.width) {
            frameSize.width = screenSize.width;
        }
        frame.setLocation(
```

```
            ( screenSize.width - frameSize.width ) / 2,
            ( screenSize.height - frameSize.height ) / 2 );
        frame.addWindowListener( new WindowAdapter()
            { public void windowClosing(WindowEvent e) { System.exit(0); } });
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            e.printStackTrace();
        }
        new DfcTutorialApplication();
    }
}
```

Running the tutorial application

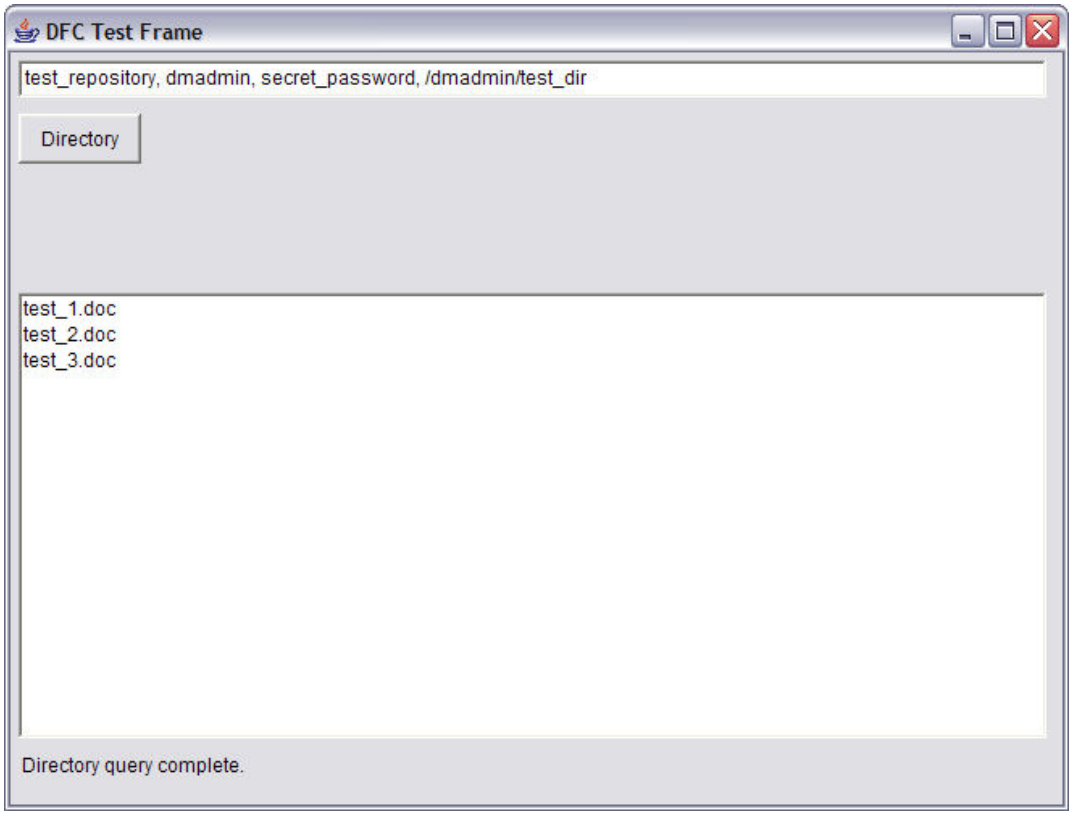
Compile the three base classes and run DfcTutorialApplication.class. The application interface appears.

Figure 5. The DFC Test Frame



To list the contents of a directory, a comma-delimited set of values in the arguments field: *repository_name, user_name, password, directory_path.* Click the Directory button to update the file listing.

Figure 6. DFC Test Frame with directory information



This example demonstrated how you can create an application that connects to the content server and retrieves information. Appendix A, Sample Test Interface provides a complete code listing of an expanded version of the DfcTestFrame class that allows you to try out several of the document manipulation operations described in Operations for manipulating documents, page 57

Chapter 4

Working with Document Operations

This chapter describes the way to use DFC to perform the most common operations on documents. Most information about documents also applies to the broader category of repository objects represented by the IDfSysObject interface.

The chapter contains the following main sections:

Introduction to documents

The *Documentum Content Server Fundamentals* manual explains Documentum facilities for managing documents. This section provides a concise summary of what you need to know to understand the remainder of this chapter.

Documentum maintains a repository of objects that it classifies according to a type hierarchy. For this discussion, *SysObjects* are at the top of the hierarchy. A *document* is a specific kind of SysObject. Its primary purpose is to help you manage content.

Documentum maintains more than one version of a document. A *version tree* is an original document and all of its versions. Every version of the document has a unique object ID, but every version has the same *chronicle ID*, namely, the object ID of the original document.

Virtual documents

A *virtual document* is a container document that includes one or more objects, called *components*, organized in a tree structure. A component can be another virtual document or a simple document. A virtual document can have any number of components, nested to any level. Documentum imposes no limit on the depth of nesting in a virtual document.

Documentum uses two sets of terminology for virtual documents. In the first set, a virtual document that contains a component is called the component's *parent*, and the component is called the virtual document's *child*. Children, or children of children to any depth, are called *descendants*.

Note: Internal variables, Javadoc comments, and registry keys sometimes use the alternate spelling *descendent*.

The second set of terminology derives from graph theory, even though a virtual document forms a tree, and not an arbitrary graph. The virtual document and each of its descendants is called a *node*. The directed relationship between a parent node and a child node is called an *edge*.

In both sets of terminology, the original virtual document is sometimes called the *root*.

You can associate a particular version of a component with the virtual document (this is called *early binding*) or you can associate the component's entire version tree with the virtual document. The latter allows you to select which version to include at the time you construct the document (this is called *late binding*).

Documentum provides a flexible set of rules for controlling the way it assembles documents. An *assembly* is a snapshot of a virtual document. It consists of the set of specific component versions that result from assembling the virtual document according to a set of binding rules. To preserve it, you must attach it to a SysObject: usually either the root of the virtual document or a SysObject created to hold the assembly. A SysObject can have at most one attached assembly.

You can version a virtual document and manage its versions just as you do for a simple document. Deleting a virtual document version also removes any containment objects or assembly objects associated with that version.

When you copy a virtual document, the server can make a copy of each component, or it can create an internal reference or pointer to the source component. It maintains information in the containment object about which of these possibilities to choose. One option is to require the copy operation to specify the choice.

Whether it copies a component or creates a reference, Documentum creates a new containment object corresponding to that component.

Note: DFC allows you to process the root of a virtual document as an ordinary document. For example, suppose that *doc* is an object of type IDfDocument and also happens to be the root of a virtual document. If you tell DFC to check out *doc*, it does not check out any of the descendants. If you want DFC to check out the descendants along with the root document, you must first execute an instruction like

```
IDfVirtualDocument vDoc =
    doc.asVirtualDocument(CURRENT, false)
```

If you tell DFC to check out vDoc, it processes the current version of doc and each of its descendants. The DFC Javadocs explain the parameters of the asVirtualDocument method.

Documentum represents the nodes of virtual documents by *containment objects* and the nodes of assemblies by *assembly objects*. An assembly object refers to the SysObject to which the assembly is attached, and to the virtual document from which the assembly came.

If an object appears more than once as a node in a virtual document or assembly, each node has a separate associated containment object or assembly object. No object can appear as a descendant of itself in a virtual document.

XML Documents

Documentum's XML support has many features. Information about those subjects appears in *Documentum Content Server Fundamentals* and in the *XML Application Development Guide*.

Using XML support requires you to provide a controlling XML application. When you import an XML document, DFC examines the controlling application's configuration file and applies any chunking rules that you specify there.

If the application's configuration file specifies chunking rules, DFC creates a virtual document from the chunks it creates. It imports other documents that the XML document refers to as entity references or links, and makes them components of the virtual document. It uses attributes of the containment object associated with a component to remember whether it came from an entity or a link and to maintain other necessary information. Assembly objects have the same XML-related attributes as containment objects do.

Introduction to operations

Operations are used to manipulate documents in Documentum. 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 execute the operation.

All of the examples in this chapter pertain only to documents, but operations can be used to work with 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.

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 your own applications, you would add code to handle a null returned by the add method or errors produced by the execute method.

Types of operation

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

Table 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	IDfCheckoutNode
Cancel a checkout	IDfCancelCheckoutOperation	IDfCancelCheckoutNode
Delete from a repository	IDfDeleteOperation	IDfDeleteNode

Task	Operation Type	Operation Node Type
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
Transform an XML document using XSLT	IDfXMLTransformOperation	IDfXMLTransformNode
Pre-cache objects in a BOCS repository	IDfPredictiveCachingOperation	IDfPredictiveCachingNode

Basic steps for manipulating documents

This section describes the basic steps common to using the facilities of the operations package to manipulate documents. It sets forth the basic steps, then discusses the steps in greater detail.

Steps for manipulating documents

This section describes the basic steps common to document manipulation operations. Details of manipulating documents, page 54 provides more detailed information about these steps.

To perform a document-manipulation task:

1. Use the appropriate factory method of IDfClientX to obtain an operation of the type appropriate to the document-manipulation task.

   For example, if you want to check documents into a repository, start by calling getCheckinOperation.

   Note: Each operation has a type (for example, IDfCheckinOperation) that inherits most of its methods (in particular, its add and execute methods) from IDfOperation.

2. Set parameters to control the way the operation performs the task.

   Each operation type has setXxx methods for setting its parameters.

   The operation behaves in predefined (default) ways if you do not set optional parameters. Some parameters (the session for an import operation, for example) are mandatory.

3. Add documents to the operation:

   a. Use its inherited add method to place a document, file, or folder into the operation.

The add method returns the newly created node, or a null if it fails (refer to Handling document manipulation errors, page 83).

b. Set parameters to change the way the operation handles this item and its descendants.

Each type of operation node has methods for setting parameters that are important for that type of node. These are generally the same as the methods for the corresponding type of operation. If you do not set parameters, the operation handles this item according to the setXxx methods.

c. Repeat the previous two substeps for all items you add to the operation.

4. Invoke the operation's inherited execute method to perform the task.

Note that this step may add and process additional nodes. For example, if part of the execution entails scanning an XML document for links, DFC may add the linked documents to the operation.

The execute method returns a boolean value to indicate its success (true) or failure (false). See Handling document manipulation errors, page 83) for more information.

5. Process the results.

a. Handle errors.

If it detects errors, the execute method returns the boolean value *false*. You can use the operation's inherited getErrors method to obtain a list of failures.

For details of how to process errors, see Processing the results, page 56.

b. Perform tasks specific to the operation.

For example, after an import operation, you may want to take note of all of the new objects that the operation created in the repository. You might want to display or modify their properties.

Details of manipulating documents

This section discusses some issues and background for the steps of the general procedure in Steps for manipulating documents, page 53.

Obtaining the operation

Each operation factory method of IDfClientX instantiates an operation object of the corresponding type. For example, getImportOperation factory method instantiates an IDfImportOperation object.

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.

Adding documents to the operation

An operation contains a structure of nodes and descendants. When you obtain the operation, it has no nodes. When you use the operation's *add* method to include documents in 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 (see Working with nodes, page 56).

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 might include additional nodes in 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 (see Working with nodes, page 56). You might want 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 want 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. If you have a programming hurdle that you cannot get over without using steps, work with Documentum Technical Support or Consulting to design a solution.

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 want to examine and further process the results of the operation. The next section, Working with nodes, page 56, shows how to access the objects and results associated with the nodes 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 for manipulating documents

This section provides sample code and discusses specific details of the following kinds of document manipulation operations:

The examples use the terms *file* and *directory* to refer to entities on the file system and the terms *document* and *folder* to repository entities represented by DFC objects of type IDfDocument and IDfFolder.

Checking out

The execute method of an IDfCheckoutOperation object checks out the documents in the operation. The checkout operation:

- Locks the document
- Copies the document to your local disk
- Always creates registry entries to enable DFC to manage the files it creates on the file system

Example 4-1. TutorialCheckout.java

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfId;
import com.documentum.operations.IDfCheckoutNode;
import com.documentum.operations.IDfCheckoutOperation;

public class TutorialCheckout {
    public TutorialCheckout() { }

// checkoutExample method - pass an existing session and document ID.
    public String checkoutExample(
            IDfSession mySession,
            String docId)
    {
        try {
            String result = "";

// Instantiate a client.
            IDfClientX clientx = new DfClientX();

// Use the factory method to create a checkout operation object.
            IDfCheckoutOperation coOp  = clientx.getCheckoutOperation();

// Set the location where the local copy of the checked out file is stored.
            coOp.setDestinationDirectory("C:\\");

// Get the document instance using the document ID.
            IDfDocument doc = (IDfDocument) mySession.getObject(new DfId(docId));

// Create the checkout node by adding the document to the checkout operation.
            IDfCheckoutNode coNode = (IDfCheckoutNode)coOp.add(doc);

// Verify that the node exists.
            if (coNode == null) result = ("coNode is null");

// Execute the checkout operation. Return the result.
            if ( coOp.execute()) {
                result = "Successfully checked out file ID: " + docId;
            }
            else {
```

```
                result = ("Checkout failed.");
            }
            return result;
        }
        catch (Exception ex) {
            ex.printStackTrace();
            return "Exception hs been thrown: " + ex;

        }
    }
}
```

Special considerations for checkout operations

Follow the steps in Steps for manipulating documents, page 53.

If any node corresponds to a document that is already checked out, the system does not check it out again. DFC does not treat this as an error. If you cancel the checkout, however, DFC cancels the checkout of the previously checked out node as well.

DFC applies XML processing to XML documents. If necessary, it modifies the resulting files to ensure that it has enough information to check in the documents properly.

You can use many of the same methods for setting up checkout operations and processing results that you use for export operations.

Checking out a virtual document

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.

Example 4-2. The TutorialCheckoutVdm class

```
package dfctestenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.common.DfId;
import com.documentum.operations.IDfCheckoutNode;
import com.documentum.operations.IDfCheckoutOperation;

public class TutorialCheckoutVdm {
    public TutorialCheckoutVdm() {
    }
    public String checkoutExample(
```

```
            IDfSession mySession,
            String docId)
    {
    try {
        String result = "";

// Instantiate a client.
        IDfClientX clientx = new DfClientX();

// Use the factory method to create a checkout operation object.
        IDfCheckoutOperation coOp  = clientx.getCheckoutOperation();

// Set the location where the local copy of the checked out file is stored.
        coOp.setDestinationDirectory("C:\\");

// Get the document instance using the document ID.
        IDfDocument doc = (IDfDocument) mySession.getObject(new DfId(docId));


// Create an empty checkout node object.
        IDfCheckoutNode coNode;

// If the doc is a virtual document, instantiate it as a virtual document
// object and add it to the checkout operation. Otherwise, add the document
// object to the checkout operation.
        if (doc.isVirtualDocument()){
            IDfVirtualDocument vDoc = doc.asVirtualDocument( "CURRENT",false);
            coNode = (IDfCheckoutNode)coOp.add(vDoc);
        }
        else {
            coNode = (IDfCheckoutNode)coOp.add(doc);
        }

    // Verify that the node exists.
        if (coNode == null) {
              result = ("coNode is null");
        }

// Execute the checkout operation. Return the result.
        if ( coOp.execute()) {
            result = "Successfully checked out file ID: " + docId;
        }
        else {
            result = ("Checkout failed.");
        }
        return result;
    }
    catch (Exception ex) {
        ex.printStackTrace();
        return "Exception hs been thrown: " + ex;

    }
    }
}
```

Checking in

The execute method of an IDfCheckinOperation object checks documents into the repository. It creates new objects as required, transfers the content to the repository, and removes local files if appropriate. It checks in existing objects that any of the nodes refer to (for example, through XML links).

Example 4-3. TutorialCheckin.java

Check in a document as the next major version (for example, version 1.2 would become version 2.0). The default increment is NEXT_MINOR (for example, version 1.2 would become version 1.3).

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfCheckinNode;
import com.documentum.operations.IDfCheckinOperation;

public class TutorialCheckIn {
    public TutorialCheckIn() {
    }
    public String checkinExample(
        IDfSession mySession,
        String docId)
    {
        try {
// Create a new client instance.
            IDfClientX clientx = new DfClientX();

// Use the factory method to create an IDfCheckinOperation instance.
            IDfCheckinOperation cio = clientx.getCheckinOperation();

// Set the version increment. In this case, the next major version
//
            cio.setCheckinVersion(IDfCheckinOperation.NEXT_MAJOR);

// Create a document object that represents the document being checked in.
            IDfDocument doc =
                (IDfDocument) mySession.getObject(new DfId(docId));

// Create a checkin node, adding it to the checkin operation.
            IDfCheckinNode node = (IDfCheckinNode)cio.add(doc);

// Execute the checkin operation and return the result.
            if(!cio.execute()) {
                return "Checkin failed.";
            }

// After the item is created, you can get it immediately using the
// getNewObjectId method.
```

```
            IDfId newId = node.getNewObjectId();
            return "Checkin succeeded -  new object ID is: " + newId;
            }
        catch (Exception ex) {
            ex.printStackTrace();
            return "Checkin failed.";
        }
    }
}
```

Special considerations for checkin operations

Follow the steps in Steps for manipulating documents, page 53.

Setting up the operation

To check in a document, you pass an object of type IDfSysObject or IDfVirtualDocument, *not the file on the local file system*, to the operation's add method. In the local client file registry, DFC records the path and filename of the local file that represents the content of an object. If you move or rename the file, DFC loses track of it and reports an error when you try to check it in.

Setting the content file, as in IDfCheckinNode.setFilePath, overrides DFC's saved information.

If you specify a document that is not checked out, DFC does not check it in. DFC does not treat this as an error.

You can specify checkin version, symbolic label, or alternate content file, and you can direct DFC to preserve the local file.

If between checkout and checkin you remove a link between documents, DFC adds the orphaned document to the checkin operation as a root node, but the relationship between the documents no longer exists in the repository.

Processing the checked in documents

Executing a checkin operation normally results in the creation of new objects in the repository. If opCheckin is the IDfCheckinOperation object, you can obtain a complete list of the new objects by calling

```
IDfList list = opCheckin.getNewObjects();
```

The list contains the object IDs of the newly created SysObjects.

In addition, the IDfCheckinNode objects associated with the operation are still available after you execute the operation (see Working with nodes, page 56). You can use their methods to find out many other facts about the new SysObjects associated with those nodes.

Cancelling checkout

The execute method of an IDfCancelCheckoutOperation object cancels the checkout of documents by releasing locks, deleting local files if appropriate, and removing registry entries.

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 4-4. TutorialCancelCheckout.java

Cancel checkout of a document.

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.operations.IDfCancelCheckoutNode;
import com.documentum.operations.IDfCancelCheckoutOperation;

public class TutorialCancelCheckout {
    public TutorialCancelCheckout() {
    }
    public String cancelCheckoutExample(
        IDfSession mySession,
        String docId) throws DfException
    {
        try
        {
// Get a new client instance.
            IDfClientX clientx = new DfClientX();

// Use the factory method to create a checkout operation object.
            IDfCancelCheckoutOperation cco =
                clientx.getCancelCheckoutOperation();

// Create an instance of the document using the document ID.
            IDfDocument doc =
                (IDfDocument) mySession.getObject(new DfId(docId));

//
            cco.setKeepLocalFile(true);
            IDfCancelCheckoutNode node = (IDfCancelCheckoutNode)cco.add(doc);
            if (node==null) {return "Node is null";}
            if (!cco.execute()){
                return "Operation failed";
            }
            return "Successfully cancelled checkout of file ID: " + docId;
        }
        catch (Exception e){
            e.printStackTrace();
            return "Exception thrown.";
```

```
        }
    }
}
```

Special considerations for cancel checkout operations

Follow the steps in Steps for manipulating documents, page 53.

If a document in the cancel checkout operation is not checked out, DFC does not process it. DFC does not treat this as an error.

Cancel checkout for virtual document

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 4-5. The TutorialCancelCheckoutVdm class

```
package dfctestenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.operations.IDfCancelCheckoutNode;
import com.documentum.operations.IDfCancelCheckoutOperation;

public class TutorialCancelCheckoutVdm {
    public TutorialCancelCheckoutVdm() {
    }
    public String cancelCheckoutExample(
        IDfSession mySession,
        String docId) throws DfException
    {
        try
        {

// Get a new client instance.
            IDfClientX clientx = new DfClientX();

// Use the factory method to create a checkout operation object.
            IDfCancelCheckoutOperation cco =
                clientx.getCancelCheckoutOperation();
```

```
// Instantiate the document object from the ID string.
            IDfDocument doc =
                (IDfDocument) mySession.getObject(new DfId(docId));

// Indicate whether to keep the local file.
            cco.setKeepLocalFile(true);

// Create an empty cancel checkout node.
            IDfCancelCheckoutNode node;

// If it is a virtual document, instantiate it as a vitual document and add
// the virtual document to the operation. Otherwise, add the doc to the
// operation.
            if (doc.isVirtualDocument()) {
                IDfVirtualDocument vdoc = doc.asVirtualDocument("CURRENT", false);
                node = (IDfCancelCheckoutNode)cco.add(vdoc);
            }
            else
            {
                node = (IDfCancelCheckoutNode)cco.add(doc);
            }

// Check to see if the node is null - this will not throw an error.
            if (node==null) {return "Node is null";}

// Execute the operation and return the result.
            if (!cco.execute()){
                return "Operation failed";
            }
            return "Successfully cancelled checkout of file ID: " + docId;
        }
        catch (Exception e){
            e.printStackTrace();
            return "Exception thrown.";
        }
    }
}
```

Importing

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 local files (for example, through XML or OLE links), it imports those into the repository too.

Example 4-6. TutorialImport.java

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.IDfId;
import com.documentum.fc.common.IDfList;
import com.documentum.operations.IDfFile;
import com.documentum.operations.IDfImportNode;
import com.documentum.operations.IDfImportOperation;

public class TutorialImport {
  public TutorialImport() {
  }
  public String importExample (
    IDfSession mySession,
    String folderPath,
    String fileId
  ) throws DfException
  {
// Get a new client instance.
    IDfClientX clientx = new DfClientX();

// Create an ImportOperation instance, and set the session
// using the session you passed in.
    IDfImportOperation opi = clientx.getImportOperation();
    opi.setSession(mySession);

// Instantiate a folder object based on the path you passed in.
    IDfFolder folder = mySession.getFolderByPath(folderPath);

// Instantiate a file object based on the fileId you passed in.
    IDfFile file = clientx.getFile(fileId);

// If the file does not exist, return with a message.
    if (!file.exists()) return ("File does not exist.");

// Set your folder intance as the destination for the imported
// file.
    opi.setDestinationFolderId(folder.getObjectId());

// Create an import node instance and add the file.
    IDfImportNode node = (IDfImportNode)opi.add(file);
```

```
// If the node is null, return with a message.
    if (node==null) return ("Node is null.");

// Execute the operation and report success.
    if (opi.execute() ) {
      String resultString =  ("Item" + opi.getNewObjects().toString() +
      " imported successfully.");
      return resultString;
    }else{
      return ("Error during import operation.");
    }
  }
}
```

Special Considerations for Import Operations

Follow the steps in Steps for manipulating documents, page 53.

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.

XML processing

You can import XML files without doing XML processing. If nodeImport is an IDfImportNode object, you can turn off XML processing on the node and all its descendants by calling

```
nodeImport.setXMLApplicationName("Ignore");
```

Turning off this kind of processing can shorten the time it takes DFC to perform the operation.

Processing the imported documents

Executing an import operation results in the creation of new objects in the repository. If opImport is the IDfImportOperation object, you can obtain a complete list of the new objects by calling

```
IDfList list = opImport.getNewObjects();
```

The list contains the object IDs of the newly created SysObjects.

In addition, the IDfImportNode objects associated with the operation are still available after you execute the operation (see Working with nodes, page 56). You can use their methods to find out many other facts about the new SysObjects associated with those nodes. For example, you can find out object IDs, object names, version labels, file paths, and formats.

Exporting

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.

Example 4-7. TutorialExport.java

This example does not create registry information about the resulting file.

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.operations.IDfExportNode;
import com.documentum.operations.IDfExportOperation;

public class TutorialExport {
  public TutorialExport() {
  }
  public String exportExample (
    IDfSession mySession,
    String docId,
    String target_local_directory

  ) throws DfException {

// Create a new client instance.
    IDfClientX clientx = new DfClientX();

// Use the factory method to create an IDfExportOperation instance.
    IDfExportOperation eo = clientx.getExportOperation();

// Create a document object that represents the document being exported.
    IDfDocument doc = (IDfDocument) mySession.getObject(new DfId(docId));

// Create an export node, adding the document to the export operation object.
    IDfExportNode node = (IDfExportNode)eo.add(doc);


// Set the full file path on the local system.
    node.setFilePath(target_local_directory + doc.getObjectName());

// Execute and return results
    if (eo.execute()) {
      return "Export operation successful.";
    }
    else {
      return "Export operation failed.";
    }
  }
```

```
}
```

Special considerations for export operations

Follow the steps in Steps for manipulating documents, page 53.

By default, an export operation creates files on the local system and makes no provision for Documentum to manage them. You can tell DFC to create registry entries for the files by invoking the setRecordInRegistry method of an object of type either IDfExportOperation or IDfExportNode, using the parameters described in the Javadocs.

If any node corresponds to a checked out document, DFC copies the latest repository version to the local file system. DFC does not treat this as an error.

You can find out where on the file system the export operation creates files. Use the getDefaultDestinationDirectory and getDestinationDirectory methods of IDfExportOperation objects and the getFilePath method of IDfExportNode objects to do this.

Exporting the contents of a folder requires adding each document individually to the operation.

Copying

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. 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 4-8. TutorialCopy.java

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfCopyNode;
import com.documentum.operations.IDfCopyOperation;

public class TutorialCopy {
  public TutorialCopy() {
  }
  public String copyExample (
    IDfSession mySession,
    String docId,
    String destination

  ) throws DfException {

// Create a new client instance.
    IDfClientX clientx = new DfClientX();

// Use the factory method to create an IDfCopyOperation instance.
    IDfCopyOperation co = clientx.getCopyOperation();

// Create an instance for the destination directory.
    IDfFolder destinationDirectory = mySession.getFolderByPath(destination);

// Set the destination directory by ID.
    co.setDestinationFolderId(destinationDirectory.getObjectId());

// Create a document object that represents the document being copied.
    IDfDocument doc = (IDfDocument) mySession.getObject(new DfId(docId));
```

```
// Create a copy node, adding the document to the copy operation object.
    IDfCopyNode node = (IDfCopyNode)co.add(doc);


// Execute and return results
    if (co.execute()) {
      return "Copy operation successful.";
    }
    else {
      return "Copy operation failed.";
    }
  }
}
```

Special considerations for copy operations

Follow the steps in Steps for manipulating documents, page 53.

You must set the destination folder, either on the operation or on each of its nodes.

You can use the setDeepFolders method of the operation object (node objects do not have this method) to override the default behavior of recursively adding folder contents to the operation.

Certain settings of the attributes of dm_relation and dm_relation_type objects associated with an object may cause DFC to add related objects to the copy operation. Refer to the *Server Object Reference* manual for details.

Moving

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.

Example 4-9. TutorialMove.java

Move a document.

```
package dfctutorialenvironment;
import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfMoveNode;
import com.documentum.operations.IDfMoveOperation;

public class TutorialMove {
  public TutorialMove() {
  }
  public String moveExample (
    IDfSession mySession,
    String docId,
    String destination
  ) throws DfException {

// Create a new client instance.
    IDfClientX clientx = new DfClientX();

// Use the factory method to create an IDfCopyOperation instance.
    IDfMoveOperation mo = clientx.getMoveOperation();

// Create an instance for the destination directory.
    IDfFolder destinationDirectory = mySession.getFolderByPath(destination);

// Set the destination directory by ID.
    mo.setDestinationFolderId(destinationDirectory.getObjectId());

// Create a document object that represents the document being copied.
    IDfDocument doc = (IDfDocument) mySession.getObject(new DfId(docId));
```

```
// Create a move node, adding the document to the move operation object.
    IDfMoveNode node = (IDfMoveNode)mo.add(doc);

// Execute and return results
    if (mo.execute()) {
      return "Move operation successful.";
    }
    else {
      return "Move operation failed.";
    }
  }
}
```

Special considerations for move operations

Follow the steps in Steps for manipulating documents, page 53. Options for moving are essentially the same as for copying.

If the operation entails moving a checked out document, DFC leaves the document unmodified and reports an error.

Deleting

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.

Example 4-10. TutorialDelete.java

Delete a document.

```
package dfctutorialenvironment;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.operations.IDfDeleteNode;
import com.documentum.operations.IDfDeleteOperation;

public class TutorialDelete {
  public TutorialDelete() {
}
  public String deleteExample (
    IDfSession mySession,
    String docId,
    String currentVersionOnly
  ) throws DfException {

// Create a new client instance.
    IDfClientX clientx = new DfClientX();

// Use the factory method to create an IDfDeleteOperation instance.
    IDfDeleteOperation delo = clientx.getDeleteOperation();

// Create a document object that represents the document being copied.
    IDfDocument doc = (IDfDocument) mySession.getObject(new DfId(docId));

// Set the deletion policy. You must do this prior to adding nodes to
// the Delete operation.
    if (currentVersionOnly == "true") {
      delo.setVersionDeletionPolicy(IDfDeleteOperation.SELECTED_VERSIONS);
    }
    else
    {
      delo.setVersionDeletionPolicy(IDfDeleteOperation.ALL_VERSIONS);
    }

// Create a delete node using the factory method.
    IDfDeleteNode node = (IDfDeleteNode) delo.add(doc);
```

```
    if (node==null)return "Node is null.";

// Execute the delete operation and return results.
    if (delo.execute()) {
      return "Delete operation succeeded.";
    }
    else {
      return "Delete operation failed";
    }
  }
}
```

Special considerations for delete operations

Follow the steps in Steps for manipulating documents, page 53. If the operation entails deleting a checked out document, DFC leaves the document unmodified and reports an error.

Predictive caching

Predictive caching can help you to improve the user experience by sending system objects to Branch Office Caching Services servers before they are requested by users. For example, a company-wide report could be sent to all repository caches when it is added to the local repository rather that waiting for a user request on each server. Another use for this capability would be to cache an object in response to an advance in a workflow procedure, making the document readily available for the next user in the flow.

Example 4-11. Predictive caching operation for a single document

```
void preCache (IDfClientX clientx,IDfDocument doc, IDfList networkLocationIds) {
  IDfPredictiveCachingOperation pco =
    clientx.getPredictiveCachingOperation();
  // Add the document and cast the node to the appropriate type
  IDfPredictiveCachingNode node =
    (IDfPredictiveCachingNode)pco.add( doc );
  if( node == null ) { /* handle errors */ }

  // Set properties on the node
  node.setTimeToLive(IDfPredictiveCachingOperation.DAY);
  node.setNetworkLocationIds(networkLocationIds);
  node.setMinimumContentSize(1000);

  // Execute the operation
  if( !pco.execute() ) { /* handle errors */ }
}
```

Special considerations for predictive caching operations

Follow the steps in Steps for manipulating documents, page 53.

setTimeToLive sets a time limit, in milliseconds, for BOCS. BOCS will attempt to pre-cache the content until the specified delay after successful execution of the operation.

setNetworkLocationIds sets the list of the network location identifiers to be used for content pre-caching. All BOCS servers for the specified network locations will attempt to pre-cache the content.

setMinimumContentSize is used to ensure that documents that are cached will provide a performance improvement. Smaller documents are transferred quickly enough that there is no detectable improvement in performance. Use this method to set the smallest content size, in bytes, that will be cached. Documents smaller than the minimum size will be skipped.

Validating an XML document against a DTD or schema

DFC uses a modified version of the Xerces XML parser, which it includes in dfc.jar. See the DFC release notes for details about the specific version and the Documentum modifications.

The execute method of an IDfValidationOperation object runs the parser in validation mode to determine whether or not documents are well formed and conform to the DTD or schema. If you do not specify it explicitly, DFC determines the XML application automatically and looks in the application's folder in the repository for the DTD or schema.

If any argument or the DTD or schema is in the repository, the execute method makes a temporary copy on the file system and performs the validation there.

If the parser detects errors, the operation's execute method returns a value of false, and you can use the operation's getErrors method to obtain the error information that the parser returns.

Example 4-12. Validating an XML document

Validate an XML document, using C:\Temp for a working directory.

```
void validateXMLDoc( IDfClientX clientx, // Factory for operations
   IDfDocument doc )            // Document to validate
throws DfException
  {
   // Obtain validation operation
   IDfValidationOperation validate = clientx.getValidationOperation();
   validate.setDestinationDirectory( "C:/Temp" );

   // Convert the document to a node tree
   IDfVirtualDocument vDoc =  doc.asVirtualDocument( "CURRENT", false );

   // Add the document to the operation
   IDfValidationNode node = (IDfValidationNode)validate.add( vDoc );
   if( node == null ) { /* handle errors */ }

   // Execute the operation
   if( !validate.execute() ) { /* handle errors */ }
   }
```

Special considerations for validation operations

Follow the steps in Steps for manipulating documents, page 53.

You can use the operation's setDestinationDirectory method to specify the file system directory to which the operation exports the XML files and DTDs or schemas that it passes to the parser.

Performing an XSL transformation of an XML document

The execute method of the IDfXMLTransformOperation interface uses the Xalan transformation engine and the specified XSLT stylesheet to transform an XML file or document. It places the output into a new document or attaches it to the original document as a rendition. It can also output an object of type IDfFile or any java.io.Writer or java.io.OutputStream stream.

Note: Refer to the DFC release notes for details about the specific version of the Xalan transformation engine that DFC requires.

Example 4-13. Transform to an HTML file

Transform the file C:\Newsletter.xml into an HTML file C:\Newsletter.htm, using an XSLT stylesheet from the repository.

```
void transformXML2HTMLUsingStylesheetObject(
   IDfClientX clientx,            // Factory for operations
   IDfSession session,            // Repository session (required)
   IDfDocument docStylesheet )    // XSL stylesheet in repository
throws DfException, IOException
   {
   // Obtain transformation operation
   IDfXMLTransformOperation opTran = clientx.getXMLTransformOperation();

   // Set operation properties
   opTran.setSession( session );
   opTran.setTransformation( docStylesheet );
   FileOutputStream out = new FileOutputStream( "C:\\Newsletter.htm" );
   opTran.setDestination( out );

   // Add the XML file to the operation
   IDfXMLTransformNode node = (IDfXMLTransformNode)
                  opTran.add( "C:\\Newsletter.xml" );
   if( node == null ) { /* handle errors */ }

   // Set node properties
   node.setOutputFormat( "html" );

   // Execute the operation
   if( !opTran.execute() ) { /* handle errors */ }
   }
```

Example 4-14. Transform to an HTML file, import result into repository

Transform the file C:\Newsletter.xml into a new HTML document, using the XSLT stylesheet C:\Stylesheet.xsl, and import it into the repository.

```
void transformXML2HTMLUsingStylesheetFile(
   IDfClientX clientx,            // Factory for operations
   IDfSession session,            // Repository session (required)
   IDfId idDestFolder )           // Destination folder
throws DfException
   {
   // Obtain transformation operation
   IDfXMLTransformOperation opTran = clientx.getXMLTransformOperation();

   /// Set transformation operation properties
   opTran.setSession( session );
   opTran.setTransformation( "C:\\Stylesheet.xsl" );

   // Obtain import operation
   IDfImportOperation opImp = clientx.getImportOperation();

   // Set import operation properties
   opImp.setSession( session );
   opImp.setDestinationFolderId( idDestFolder );

   // Specify the import operation as the destination of the
   //  transformation operation. In effect, this adds the output
   //  of the transformation operation to the import operation,
   //  but it does not explicitly create an import node
   opTran.setDestination( opImp );

   // Add the XML file to the transform operation
   IDfXMLTransformNode nodeTran =
    (IDfXMLTransformNode)opTran.add( "C:\\Newsletter.xml" );
   if( nodeTran == null ) { /* handle errors */ }

   // Specify the output format
   //  (NOTE: on the transformation node. There is no import node)
   nodeTran.setOutputFormat( "html" );

   // Execute the operation
   if( !opTran.execute() ) { /* handle errors */ }
   }
```

Example 4-15. Transform an XML document into an HTML rendition

Transform an XML document into an HTML rendition, using an XSLT stylesheet from the repository.

```
void transformXML2HTMLRendition(
   IDfClientX clientx,            // Factory for operations
   IDfSession session,            // Repository session (required)
   IDfDocument docXml,            // Root of the XML document
   IDfDocument docStylesheet )        // XSL stylesheet in repository
throws DfException
   {
   // Obtain transformation operation
   IDfXMLTransformOperation opTran = clientx.getXMLTransformOperation();

   opTran.setSession( session );
   opTran.setTransformation( docStylesheet );

   // Add XML document to the transformation operation
   IDfXMLTransformNode node = (IDfXMLTransformNode) opTran.add( docXml );
   if( node == null ) { /* handle errors */ }

   //Set HTML file for the rendition

   // Set format for the rendition
   node.setOutputFormat( "html" );

   // Execute the operation
   if( !opTran.execute() ) { /* handle errors */ }
   }
```

DFC creates a rendition because the output format differs from the input format and you did not call optran.setDestination to specify an output directory.

Special considerations for XML transform operations

Follow the steps in Steps for manipulating documents, page 53.

You must use the operations setSession method to specify a repository session. This operation requires a session, even if all of the files it operates on are on the file system.

The add method of an IDfXMLTransformOperation object accepts Java types as well as Documentum types. It allows you to specify the file to transform as an object of any of the following types:

- IDfDocument
- IDfFile
- String (for example C:/PhoneInfo.xml)
- InputStream
- Reader
- URL

Handling document manipulation errors

This section describes the ways that DFC reports errors that arise in the course of populating or executing operations.

The add Method Cannot Create a Node

The add method of any operation returns a null node if it cannot successfully add the document, file or folder that you pass it as an argument. Test for a null to detect and handle this failure. DFC does not report the reason for returning a null.

The execute Method Encounters Errors

The execute method of an operation throws DfException only in the case of a fatal error. Otherwise it creates a list of errors, which you can examine when the execute method returns or, if you use an operation monitor, as they occur.

Examining Errors After Execution

After you execute an operation, you can use its getErrors method to retrieve an IDfList object containing the errors. You must cast each to IDfOperationError to read its error message.

After detecting that the operation's execute method has returned errors, you can use the operation's abort method to undo as much of the operation as it can. You cannot undo XML validation or transform operations, nor can you restore deleted objects.

Example 4-16. Generate an Operation Exception

Generate a DfException to report the errors that occurred in the course of executing an operation.

```
public DfException generateOperationException(
   IDfOperation operation,
   IDfSession session,
   String msg )
throws DfException
   {
   String message = msg;
   DfException e;
   try {

    // Initialize variables
    String strNodeName = "";
```

```
    String strNodeId = "";
    String strSucceeded = "";
    IDfId idNodesObj = null;
    IDfOperationError error = null;
    IDfOperationNode node = null;

    // Get the list of errors
    IDfList errorList = operation.getErrors();

    // Iterate through errors and build the error messages
    for( int i = 0; i < errorList.getCount(); ++i ) {

     // Get next error
     error = (IDfOperationError)errorList.get( i );

     // Get the node at which the error happened
     node = error.getNode();

     // Use method described in another example
     idNodesObj = this.getObjectId( session, node );
     if( idNodesObj <> null ) {;
      strNodeId = idNodesObj.getId();
      strNodeName = session.apiGet( "get", strNodeId + ",object_name" );
      message += "Node: [" + strNodeId + "], " + strNodeName + ", "
         + error.getMessage() + ", " + error.getException().toString();
    } // end for
   }  // end try
   catch( Exception err )
    { message += err.toString(); }
   finally {
    // Create a DfException to report the errors
    e = new DfException();
    e.setMessage( message );
   } // end finally
   return e;
   }
```

Example 4-17. Obtain the Object ID of an Operation Node

Obtain the IDfId for the operation node.

```
IDfId getObjectId( IDfSession session, IDfOperationNode node ) {
try {
   return
   node instanceof IDfImportNode ? ((IDfImportNode)node).getObjectId()
   : node instanceof IDfExportNode ? ((IDfExportNode)node).getObjectId()
   : node instanceof IDfCheckoutNode ?
                  ((IDfCheckoutNode)node).getObjectId()
   : node instanceof IDfCheckinNode ?
                 ((IDfCheckinNode)node).getObjectId()
   : node instanceof IDfCancelCheckoutNode ?
               ((IDfCancelCheckoutNode)node).getObjectId()
   : node instanceof IDfDeleteNode ? ((IDfDeleteNode)node).getObjectId()
   : node instanceof IDfCopyNode ?   ((IDfCopyNode)node).getObjectId()
   : node instanceof IDfMoveNode ?   ((IDfMoveNode)node).getObjectId()
   : null;
```

```
   } catch( Exception e ) { return null; }
}
```

Using an Operation Monitor to Examine Errors

You can monitor an operation for progress and errors. Create a class that implements the IDfOperationMonitor interface and register it by calling the setOperationMonitor method of IDfOperation. The operation periodically notifies the operation monitor of its progress or of errors that it encounters.

During execution, DFC calls the methods of the installed operation monitor to report progress or errors. You can display this information to an end user. In each case DFC expects a response that tells it whether or not to continue. You can make this decision in the program or ask an end user to decide.

Your operation monitor class must implement the following methods:

- progressReport

  DFC supplies the percentage of completion of the operation and of its current step. DFC expects a response that tells it whether to continue or to abort the operation.
- reportError

  DFC passes an object of type IDfOperationError representing the error it has encountered. It expects a response that tells it whether to continue or to abort the operation.
- getYesNoAnswer

  This is the same as reportError, except that DFC gives you more choices. DFC passes an object of type IDfOperationError representing the error it has encountered. It expects a response of yes, no, or abort.

The Javadocs explain these methods and arguments in greater detail.

Operations and transactions

Operations do not use session transactions (see), 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. For information about what session transactions can and cannot do, refer to .

Chapter 5

Using the Business Object Framework (BOF)

This chapter introduces the Business Object Framework (BOF). It contains the following major sections:

Overview of BOF

BOF's main goals are to centralize and standardize the process of customizing Documentum functionality. BOF centralizes business logic within the framework. Using BOF, you can develop business logic that

- Always executes, regardless of the client program
- Can extend the implementation of core Documentum functionality
- Runs well in concert with an application server environment

In order to achieve this, the framework leaves customization of the user interface to the clients. BOF customizations embody business logic and are independent of considerations of presentation or format.

If you develop BOF customizations and want to access them from the .NET platform, you must take additional steps. We do not provide tools to assist you in this. You can, however, expose some custom functionality as web services. You can access web services from a variety of platforms (in particular, .NET). The *Web Services Framework Development Guide* provides information about deploying and using the web services.

BOF infrastructure

This section describes the infrastructure that supports the Business Object Framework.

Modules and registries

To understand BOF, first look at how it stores customizations in a repository. A *module* is a unit of executable business logic and its supporting material (for example, documentation, third party software, and so forth).

DFC uses a special type of repository folder (dmc_module) to contain a module. The *Content Server Object Reference Manual* describes the attributes of this type. The attributes identify the module type, its implementation class, the interfaces it implements, and the modules it depends on. Other attributes provide version information, a description of the module's functionality, and the developer's contact information.

Every repository has a System cabinet, which contains a top level folder called Modules. The folders in the Modules directory store the JAR files for the interface and implementation classes. Under Modules are the following folders, corresponding to the out-of-the-box module types:

- /System/Modules/SBO

  Contains service based objects (SBOs). An SBO is a module whose executable business logic is Java code that extends DfService. Refer to Service-based Business Objects (SBOs), page 92 for more information about SBOs.

- /System/Modules/TBO

  Contains type based objects (TBOs), that is, customizations of specific repository types. A TBO is a module in which the executable Java code extends a DFC repository type (normally, DfDocument) and implements the IDfBusinessObject interface. Refer to Type-based Business Objects (TBOs), page 100 for more information about TBOs.

- /System/Modules/Aspect

  Contains aspects, a type of module used to apply behaviors and properties to system objects dynamically.

You can create other subfolders of /System/Modules to represent other types of module. For example, if the repository uses Java-based evaluation of validation expressions (see Validation Expressions in Java, page 137), the associated modules appear under /System/Modules/Validation.

The bottom level folders under this hierarchy (except for aspects) are of type dmc_module. Each contains an individual module.

A module that is in other respects like an SBO but does not implement the IDfService interface is called a *simple module*. You can use simple modules to implement repository methods, such as those associated with workflows and document lifecycles. The implementation class of a simple module

should implement the marker interface IDfModule. Use the newModule method of IDfClient to access a simple module.

The hierarchy of folders under /System/Modules/ is the repository's module registry, or simply its *registry*.

Note: Earlier versions of DFC maintained a registry of TBOs and SBOs on each client machine. That registry is called the Documentum business object registry (DBOR). The DBOR form of registry is deprecated, but for now you can still use it, even in systems that contain repository based registries. Where both are present, DFC gives preference to the repository based registry.

Packaging support

Service-based Business Objects (SBOs), page 92 and Type-based Business Objects (TBOs), page 100, describe the mechanics of constructing the classes and interfaces that constitute the most common types of module. These details are not very different from the way they were in earlier versions. The key changes are in packaging. This section describes BOF features that help you package your business logic into modules.

Application Builder (DAB)

Application Builder (DAB) provides tools to package modules and install them in a repository's registry.

To prepare a module for packaging by DAB, you must first prepare a JAR file that contains only the module's implementation classes and another JAR file that contains only its (optional) interface classes. You must also have JAR files containing the interfaces of any modules your module depends on. Then prepare any Java libraries and documentation that you want to include in the module. DAB can package items, such as configuration files, that are not in JARs. You can access these from a module implementation by using the class's getResourceAsStream method.

Use DAB to package all of these into a module and place the module into a DocApp.

You can use the DocApp Installer (DAI) to install the module into the module registries of the target repositories. This requires administrator privileges on each repository.

Whether this is the first deployment or an update of your module, the process is the same. For the first deployment, you must also ensure that the module's interface JAR is installed on client machines. For updates, this is not required unless the interface changes. Refer to Deploying module interfaces, page 91 for more information. Be certain that you have properly configured the global registry for your DFC instance before attempting to access your custom modules. Refer to *Documentum Foundation Classes Installation Guide* for more information.

JAR files

DAB packages JAR files into repository objects of type dmc_jar. The *Object Reference Manual* describes the attributes of the dmc_jar type. Those attributes, which DAB sets using information that you supply, specify the minimum Java version that the classes in the JAR file require. They also specify whether the JAR contains implementations, interfaces, or both.

DAB links the interface and implementation JARs for your module directly to the module's top level folder; that is, to the dmc_module object that defines the module. It links the interface JARs of modules that your module depends on into the External Interfaces subfolder of the top level folder.

Libraries and sandboxing

DAB links JARs (in the form of dmc_jar objects) for supporting software into folders of type dmc_java_library, which are created in the /System/Java Libraries folder. It links the dmc_java_library folder to the top level folder of each module in which the Java library is included. The *Content Server Object Reference Manual* describes the attributes of the dmc_java_library type. The single attribute of this type specifies whether or not to sandbox the JAR files linked to that folder.

The verb *sandbox* refers to the practice of loading the given Java library into memory in such a way that other applications cannot access it. This can have a heavy cost in memory use, but it enables different applications to use different versions of the same library without conflicts. A module with a sandboxed Xerces library, for example, uses its own version, even if there is a different version on the classpath and a third version in use by a different module.

DFC achieves sandboxing by using a shared BOF class loader and separate class loaders for each module. These class loaders try to load classes first, before delegating to the usual hierarchy of Java class loaders.

Note: Java libraries can contain interfaces, implementations, or both. Do not include both interfaces and implementations in your own Java libraries. If the library is a third party software package, you may have to include both. In this case, do not use interfaces defined in that library in the method signatures of your classes.

If you prepare a separate JAR for your module's interfaces but fail to remove those interfaces from the implementation JAR, you will encounter a ClassCastException when you try to use your module.

You can sandbox libraries that contain only implementations. You can sandbox third party libraries. Never sandbox a library that contains an interface that is part of your module's method signature.

DFC automatically sandboxes the implementation JARs of modules.

DFC automatically sandboxes files that are not JARs. You can access them as resources of the associated class loader.

Deploying module interfaces

You must deploy the interface classes of your modules to each client machine, that is, to each machine running an instance of DFC. Typically, you install the interface classes with the application that uses them. They do not need to be on the global classpath.

A TBO that provides no methods of its own (for example, if it only overrides methods of DfDocument) does not need an interface. For a TBO that does not have an interface, there is nothing to install on the client machines.

In order to use hot deployment of revised implementation classes (see Dynamic delivery mechanism, page 91), you must not change the module's interface. You can extend module interfaces without breaking existing customizations.

Dynamic delivery mechanism

BOF delivers the implementation classes of TBOs, SBOs, and other modules dynamically from repository based registries to client machines. A TBO, an aspect, or a simple module is specific to its repository. An SBO is not. BOF can deliver SBO implementation classes to client machines from a single repository. Global registry, page 92 explains how DFC does this.

Delivering implementation classes dynamically from a repository means that you don't need to register those classes on client machines. It also means that all client machines use the same version of the implementation class. You deploy the class to one place, and DFC does the rest.

The delivery mechanism supports *hot deployment*, that is, deployment of new implementations without stopping the application server. This means that applications can pick up changes immediately and automatically. You deploy the module to the global registry, and all clients quickly become aware of the change and start using the new version. DFC works simultaneously with the new version and existing instantiations of the old version until the old version is completely phased out.

The delivery mechanism relies on local caching of modules on client machines (where the term client machine often means the machine running an application server and, usually, WDK). DFC does not load TBOs, aspects, or simple modules into the cache until an application tries to use them. Once DFC has downloaded a module, it only reloads parts of the module that change.

DFC checks for updates to the modules in its local cache whenever an application tries to use one or after an interval specified in seconds in dfc.bof.cacheconsistency.interval in the dfc.properties file. The default value is 60 seconds.

If DFC tries to access a module registry and fails, it tries again after a specified interval. The interval, in seconds, is the value of dfc.bof.registry.connect.attempt.interval in the dfc.properties file. The default value is 60 seconds.

DFC maintains its module cache on the file system of the client machine. You can specify the location by setting dfc.cache.dir in the dfc.properties file. The default value is the cache subdirectory of the

directory specified in dfc.data.dir. All applications that use the given DFC installation share the cache. You can even share the cache among more than one DFC installation.

Global registry

DFC delivers SBOs from a central repository. That repository's registry is called the *global registry*.

Global registry user

The global registry user, who has the user name of dm_bof_registry, is the repository user whose account is used by DFC clients to connect to the repository to access required service-based objects or network locations stored in the global registry. This user has Read access to objects in the /System/Modules, /System/BocsConfig, /dm_bof_registry, and /System/NetworkLocations only, and no other objects.

Accessing the global registry

The identity of the global registry is a property of the DFC installation. Different DFC installations can use different global registries, but a single DFC installation can have only one global registry.

In addition to efficiency, local caching provides backup if the global registry repository is unavailable. By default, DFC preloads all SBO implementation classes from the global registry to the local cache. That is, DFC downloads these classes, regardless of whether or not any application has tried to instantiate them. DFC does this only once. Thereafter, it downloads an implementation only if it changes presumably an infrequent event. Restarting DFC does not cause it to lose the contents of its local cache. This provides backup if the application loses its connection to the repository containing the global registry.

The dfc.properties file contains properties that relate to accessing the global registry. Refer to BOF and global registry settings, page 18 for information about using these properties.

Service-based Business Objects (SBOs)

This section contains the following main sections:

SBO introduction

A *service based object* (SBO) is a type of module designed to enable developers to access Documentum functionality by writing small amounts of relevant code. The underlying framework handles most of the details of connecting to Documentum repositories. SBOs are similar to session beans in an Enterprise JavaBean (EJB) environment.

SBOs can operate on multiple object types, retrieve objects unrelated to Documentum objects (for example, external email messages), and perform processing. You can use SBOs to implement functionality that applies to more than one repository type. For example, a Documentum Inbox object is an SBO. It retrieves items from a user's inbox and performs operations like removing and forwarding items.

You can use SBOs to implement utility functions to be called by multiple TBOs. A TBO has the references it needs to instantiate an SBO.

You can implement an SBO so that an application server component can call the SBO, and the SBO can obtain and release repository sessions dynamically as needed.

SBOs are the basis for the web services framework.

SBO architecture

An SBO associates an interface with an implementation class. Each folder under /System/Modules/SBO corresponds to an SBO. The name of the folder is the name of the SBO, which by convention is the name of the interface.

SBOs are not associated with a repository type, nor are they specific to the repository in which they reside. As a result, each DFC installation uses a global registry (see Global registry, page 92). The dfc.properties file contains the information necessary to enable DFC to fetch SBO implementation classes from the global registry.

You instantiate SBOs with the newService method of IDfClient, which requires you to pass it a session manager. The newService method searches the registry for the SBO and instantiates the associated Java class. Using its session manager, an SBO can access objects from more than one repository.

You can easily design an SBO to be stateless, except for the reference to its session manager.

Note: DFC does not enforce a naming convention for SBOs, but we recommend that you follow the naming convention explained in Follow the Naming Convention, page 99.

Implementing SBOs

This section explains how to implement an SBO.

An SBO is defined by its interface. Callers cannot instantiate an SBO's implementation class directly. The interface should refer only to the specific functionality that the SBO provides. A separate interface, IDfService, provides access to functionality common to all SBOs. The SBO's implementation class, however, should not extend IDfService. Instead, the SBO's implementation class must extend DfService, which implements IDfService. Extending DfService ensures that the SBO provides several methods for revealing information about itself to DFC and to applications that use the SBO.

To create an SBO, first specify its defining interface. Then create an implementation class that implements the defining interface and extends DfService. DfService is an abstract class that defines common methods for SBOs.

Override the following abstract methods of DfService to provide information about your SBO:

- getVersion returns the current version of the service as a string.

  The version is a string and must consist of an integer followed by up to three instances of dot integers (for example, 1.0 or 2.1.1.36). The version number is used to determine installation options.
- getVendorString returns the vendor's copyright statement (for example, "Copyright 1994-2005 EMC Corporation. All rights reserved.") as a string.
- isCompatible checks whether the class is compatible with a specified service version

  This allows you to upgrade service implementations without breaking existing code. Java does not support multiple versions of interfaces.
- supportsFeature checks whether the string passed as an argument matches a feature that the SBO supports.

The getVersion and isCompatible methods are important tools for managing SBOs in an open environment. The getVendorString method provides a convenient way for you to include your copyright information. The supportsFeature method can be useful if you develop conventions for naming and describing features.

SBO programming differs little from programming for other environments. The following sections address the principal additional considerations.

Stateful and stateless SBOs

SBOs can maintain state between calls, but they are easier to deploy to multithreaded and other environments if they do not do so. For example, a checkin service needs parameters like retainLock and versionLabels. A stateful interface for such a service provides get and set methods for such parameters. A stateless interface makes you pass the state as calling arguments.

Managing Sessions for SBOs

This section presents session manager related considerations for implementing SBOs.

Overview

When implementing an SBO, you normally use the getSession and releaseSession methods of DfService to obtain a DFC session and return it when finished. Once you have a session, use the methods of IDfSession and other DFC interfaces to implement the SBO's functionality.

If you need to access the session manager directly, however, you can do so from any method of a service, because the session manager object is a member of the DfService class. The getSessionManager method returns this object. To request a new session, for example, use the session manager's newSession method.

Structuring Methods to Use Sessions

Each SBO method that obtains a repository session must release the session when it is finished accessing the repository. The following example shows how to structure a method to ensure that it releases its session, even if exceptions occur:

```
public void doSomething( String strRepository, . . . ) {
   IDfSession session = getSession ( strRepository );
   try { /* do something */ }
   catch( Exception e ) { /* handle error */ }
   finally { releaseSession( session ); }
}
```

Managing repository names

To obtain a session, an SBO needs a repository name. To provide the repository name, you can design your code in any of the following ways:

- Pass the repository name to every service method.

  This allows a stateless operation. Use this approach whenever possible.
- Store the repository name in an instance variable of the SBO, and provide a method to set it (for example, setRepository (strRepository)).

  This makes the repository available from all of the SBO's methods.
- Extract the repository name from an object ID.

  A method that takes an object ID as an argument can extract the repository name from the object ID (use the getDocbaseNameFromId method of IDfClient).

Maintaining State Beyond the Life of the SBO

The EMC | Documentum architecture enables SBOs to return persistent objects to the calling program. Persistent objects normally maintain their state in the associated session object. But an SBO must release the sessions it uses before returning to the calling program. At any time thereafter, the session manager might disconnect the session, making the state of the returned objects invalid.

The calling program must ensure that the session manager does not disconnect the session until the calling program no longer needs the returned objects.

Another reason for preserving state between SBO calls occurs when a program performs a query or accesses an object. It must obtain a session and apply that session to any subsequent calls requiring authentication and Content Server operations. For application servers, this means maintaining the session information between HTTP requests.

The main means of preserving state information are setSessionManager and transactions. Maintaining state in a session manager, page 37 describes the setSessionManager mechanism and its cost in resources. Using Transactions With SBOs, page 96 provides details about using transactions with SBOs.

You can also use the DfCollectionEx class to return a collection of typed objects from a service. DfCollectionEx locks the session until you call its close method.

Obtaining Session Manager State Information

For testing or performance tuning you can examine such session manager state as reference counters, the number of sessions, and repositories currently connected. Use the getStatistics method of IDfSessionManager to retrieve an IDfSessionManagerStatistics object that contains the state information. The statistics object provides a snapshot of the session manager's internal data as of the time you call getStatistics. DFC does not update this object if the session manager's state subsequently changes.

The DFC Javadocs describe the available state information.

Using Transactions With SBOs

DFC supports two transaction processing mechanisms: session based and session manager based. describes the differences between the two transaction mechanisms. You cannot use session based transactions within an SBO method. DFC throws an exception if you try to do so.

Use the following guidelines for transactions within an SBO:

- Never begin a transaction if one is already active.

  The isTransactionActive method returns true if the session manager has a transaction active.

- If the SBO does not begin the transaction, do not use commitTransaction or abortTransaction within the SBO's methods.

  If you need to abort a transaction from within an SBO method, use the session manager's setTransactionRollbackOnly method instead, as described in the next paragraph.

When you need the flow of a program to continue when transaction errors occur, use the session manager's setTransactionRollbackOnly. Thereafter, DFC silently ignores attempts to commit the transaction. The owner of the transaction does not know that one of its method calls aborted the transaction unless it calls the getTransactionRollbackOnly method, which returns true if some part of the program ever called setTransactionRollbackOnly. Note that setTransactionRollbackOnly does not throw an exception, so the program continues as if the batch process were valid.

The following program illustrates this.

```
void serviceMethodThatRollsBack( String strRepository, IDfId idDoc )
   throws DfNoTransactionAvailableException, DfException {

  IDfSessionManager sMgr = getSessionManager();
  IDfSession = getSession( strRepository );
  if( ! sMgr.isTransactionActive() ) {
  throw new DfNoTransactionAvailableException();
  }

  try {
  IDfPersistentObject obj = session.getObject( idDoc );
  obj.checkout()
  modifyObject( obj );
  obj.save();
  }

  catch( Exception e ) {
  setTransactionRollbackOnly();
  throw new DfException();
  }
}
```

When more than one thread is involved in session manager transactions, calling beginTransaction from a second thread causes the session manager to create a new session for the new thread.

The session manager supports transaction handling across multiple services. It does not disconnect or release sessions while transactions are pending.

For example, suppose one service creates folders and a second service stores documents in these folders. To make sure that you remove the folders if the document creation fails, place the two service calls into a transaction. The DFC session transaction is bound to one DFC session, so it is important to use the same DFC session across the two services calls. Each service performs its own atomic operation. At the start of each operation, they request a DFC session and at the end they release this session back to the session pool. The session manager holds on to the session as long as the transaction remains open.

Use the beginTransaction method to start a new transaction. Use the commitTransaction or abortTransaction method to end it. You must call getSession after you call beginTransaction, or the session object cannot participate in the transaction.

Use the isTransactionActive method to ask whether the session manager has a transaction active that you can join. DFC does not allow nested transactions.

The transaction mechanism handles the following issues:

- With multiple threads, transaction handling operates on the current thread only.

  For example, if there is an existing session for one thread, DFC creates a new session for the second thread automatically. This also means that you cannot begin a transaction in one thread and commit it in a second thread.
- The session manager provides a separate session for each thread that calls beginTransaction.

  For threads that already have a session before the transaction begins, DFC creates a new session.
- When a client starts a transaction using the beginTransaction method, the session manager does not allow any other DFC-based transactions to occur.

The following example illustrates a client application calling two services that must be inside a transaction, in which case both calls must succeed, or nothing changes:

```
IDfClient client = DfClientX.getLocalClient();
IDfSessionManager sMgr = client.newSessionManager();

sMgr.setIdentity(repo, loginInfo);

IMyService1 s1 = (IMyService1)
    client.newService(IMyService1.class.getName(), sMgr);

IMyService2 s2 = (IMyService2)
    client.newService(IMyService2.class.getName(), sMgr);

s1.setRepository( strRepository1 );
s2.setRepository( strRepository2 ) ;

sMgr.beginTransaction();

try {
s1.doRepositoryUpdate();
s2.doRepositoryUpdate();
sMgr.commitTransaction();
}

catch (Exception e) {
sMgr.abortTransaction();
}
```

If either of these service methods throws an exception, the program bypasses commit and executes abort.

Each of the doRepositoryUpdate methods calls the session manager's getSession method.

Note that the two services in the example are updating different repositories. Committing or aborting the managed transaction causes the session manager to commit or abort transactions with each repository.

Session manager transactions involving more than one repository have an inherent weakness that arises from their reliance on the separate transaction mechanisms of the databases underlying the repositories. Refer to for information about what session manager transactions can and cannot do.

SBO Error Handling

The factory method that instantiates SBOs throws a variety of exceptions. For example, it throws DfServiceInstantiationException if DFC finds the requested service but is unable to instantiate the specified Java class. This can happen if the Java class is not in the classpath or is an invalid data class. Security for Java classes on an application server can also cause this exception.

SBO Best Practices

This section describes best practices for using SBOs.

Follow the Naming Convention

DFC does not enforce a naming convention for SBOs, but we recommend that you give an SBO the same name as the fully qualified name of the interface it implements. For example, if you produce an SBO that implements an interface called IContentValidator, you might name it com.myFirm.services.IContentValidator. If you do this, the call to instantiate an SBO becomes simple. For example, to instantiate an instance of the SBO that implements the IContentValidator interface, simply write

```
IContentValidator cv = (IContentValidator)client.newService(
      IContentValidator.class.getName(), sMgr);
```

The only constraint DFC imposes on SBO names is that names must be unique within a registry.

Don’t Reuse SBOs

Instantiate a new SBO each time you need one, rather than reusing one. Refer to Calling SBOs, page 116 for details.

Make SBOs Stateless

Make SBOs as close to stateless as possible. Refer to Stateful and stateless SBOs, page 94 for details.

Rely on DFC to Cache Repository Data

DFC caches persistent repository data. There is no convenient way to keep a private cache synchronized with the DFC cache, so rely on the DFC cache, rather than implementing a separate cache as part of your service's implementation.

Type-based Business Objects (TBOs)

Use of Type-based Business Objects

Type-based Business Objects are used for modifying and extending the behavior of persistent repository object types, including core DFC types (such as documents and users) and other TBOs. TBOs extend DFC object types that inherit from IDfPersistentObject, and which map to persistent repository objects, such as dm_document or dm_user. The TBOs themselves map to custom repository object types.

For example, suppose you want to add or modify behaviors exhibited by a custom repository type derived from dm_document, which we will call mycompany_sop. Typically, you might want to extend behavior that occurs whenever a document of type mycompany_sop is checked in, by starting a workflow, validating or setting XML attributes, applying a lifecycle, or creating a rendition. Or you may want to add new behaviors to the object type that are called from a client application or from an SBO.

A TBO provides a client-independent, component-based means of implementing this type of business logic, using either or both of the following techniques:

- Override the methods of the parent class from which the TBO class is derived. This approach is typically used to add pre- or postprocessing to the parent method that will be invoked during normal operations such as checkin, checkout, or link.
- Add new methods to the TBO class that can be called by an SBO or by a DFC client application.

Creating a TBO

The following sections describe how to create a TBO. Here is a summary of the steps required:

1. Create a custom repository type.
2. Create the TBO interface.
3. Create the TBO class.
4. Implement the methods of IDfBusinessObject.
5. Code your business logic by adding new methods and overriding methods of the parent class.

The following sections provide more detailed instructions for building TBOs. The sample code provided is works from the assumption that the TBO is derived from the DfDocument class, and that its purpose is to extend the behavior of the custom object on checkin and save.

Create a custom repository type

Using Application Builder, create and configure your custom type. For an example implementation, seeDeploying the SBO and TBO, page 122.

Create the TBO interface

Creating an interface for the TBO is generally recommended, but optional if you do not intend to extend the parent class of the TBO by adding new methods. If you only intend to override methods inherited from the parent class, there is no strict need for a TBO interface, but use of such an interface may make your code more self-documenting, and make it easier to add new methods to the TBO should you have a need to add them in the future.

The design of the TBO interface should be determined by which methods you want to expose to client applications and SBOs. If your TBO needs to expose new public methods, declare their signatures in the TBO interface. Two other questions to consider are (1) whether to extend the interface of the TBO superclass (e.g. IDfDocument), and (2) whether to extend IDfBusinessObject.

While the TBO *class* will need to extend the base DFC class (for example DfDocument), you may want to make the TBO *interface* more restricted by redeclaring only those methods of the base class that your business logic requires you to expose to clients. This avoids polluting the custom interface with unnecessary methods from higher-level DFC interfaces. On the other hand, if your TBO needs to expose a large number of methods from the base DFC class, it may be more natural to have the TBO interface extend the interface of the superclass. This is a matter of design preference.

Although not a functional requirement of the BOF framework, it is generally accepted practice for the TBO interface to extend IDfBusinessObject, merging into the TBO's contract its concerns as a business object with its concerns as a persistent object subtype. This enables you to get an instance of the TBO class and call IDfBusinessObject methods without the complication of a cast to IDfBusinessObject:

```
IMySop mySop = (IMySop) session.getObject(id);
if (mySop.supportsFeature("some_feature"))
```

```
{
  mySop.mySopMethod();
}
```

The following sample TBO interface extends IDfBusinessObject and redeclares a few required methods of the TBO superclass (rather than extending the interface of the superclass):

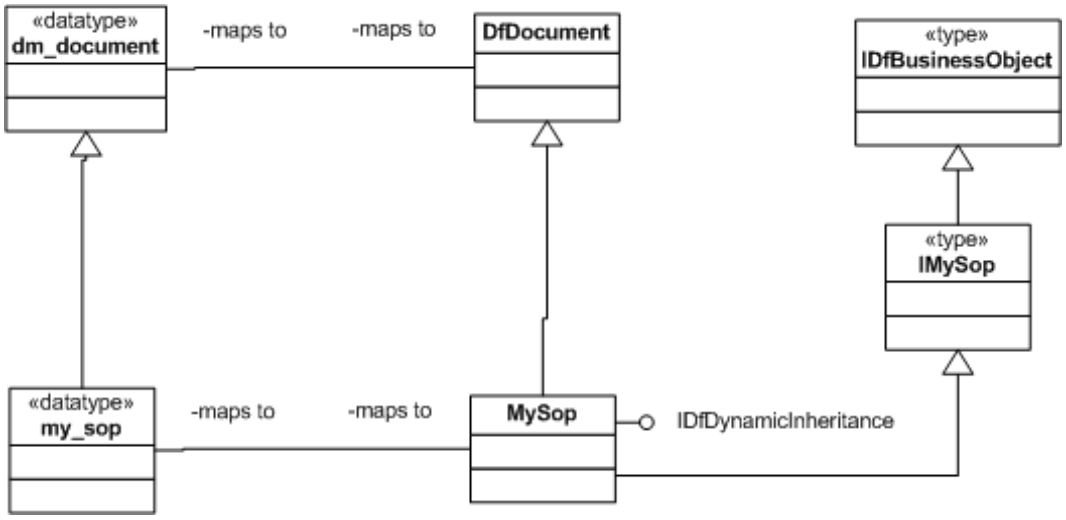
```
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.IDfId;
import com.documentum.fc.client.IDfBusinessObject;

/**
 * TBO interface intended to override checkout and save behaviors of
 * IDfDocument. IDfDocument is not extended because only a few of its
 * methods are required IDfBusinessObject is extended to permit calling
 * its methods without casting the TBO instance to IDfBusinessObject
 */
public interface IMySop extends IDfBusinessObject
{
  public boolean isCheckedOut() throws DfException;
  public void checkout() throws DfException;
  public IDfId checkin(boolean fRetainLock, String versionLabels)
    throws DfException;
  public void save() throws DfException;
}
```

Define the TBO implementation class

The main class for your TBO is the class that will be associated with a custom repository object type when deploying the TBO. This class will normally extend the DFC type class associated with the repository type from which your custom repository type is derived. For example, if your custom repository type my_sop extends dm_document, extend the DfDocument class. In this case the TBO class must implement IDfBusinessObject (either directly or by implementing your custom TBO interface that extends IDfBusinessObject) and it must implement IDfDynamicInheritance.

```
public class MySop extends DfDocument implements IMySop,
  IDfDynamicInheritance
```

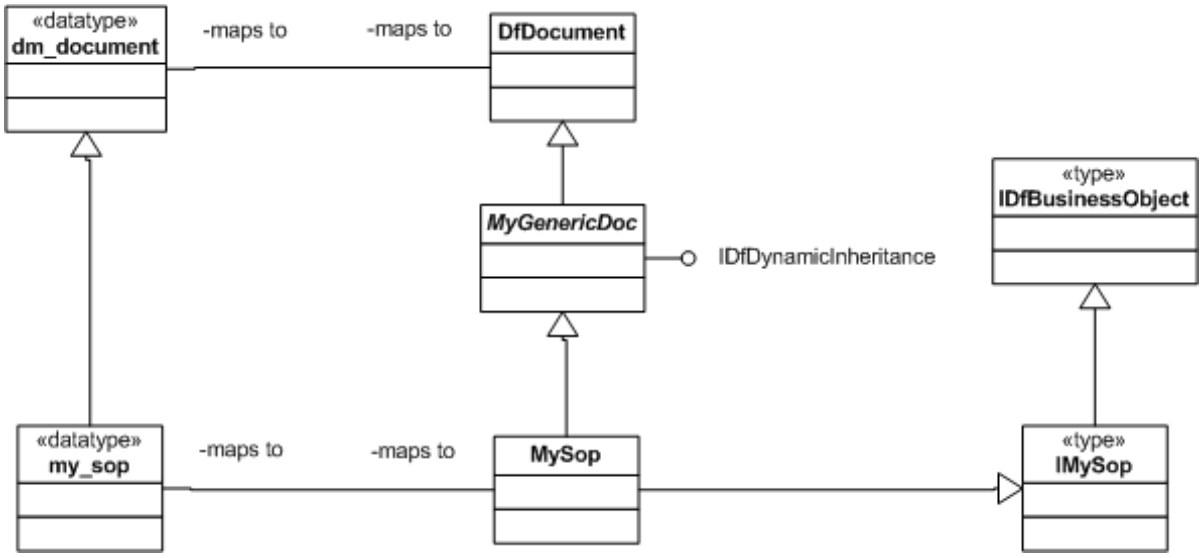
Figure 7. Basic TBO design



It is also an option to create more hierarchical levels between your main TBO class and the DFC superclass. For example, you may want to place generic methods used in multiple TBOs in an abstract class. In this case the higher class will extend the DFC superclass and implement IDfDynamicInheritance, and the main TBO class would extend the abstract class and implement the TBO interface. This will result in the correct runtime behavior for dynamic inheritance.

```
public abstract class MyGenericDoc extends DfDocument
  implements IDfDynamicInheritance
public class MySop extends MyGenericDoc implements IMySop
```

Figure 8. TBO design with extended intervening class



Note that in this situation you would need to package both MyGenericDoc and MySop into the TBO class jar file, and specify MySop as the main TBO class when deploying the TBO in the repository. For an example of packaging and deploying business objects see Deploying the SBO and TBO, page 122.

Implement methods of IDfBusinessObject

To fulfill its contract as a class of type IDfBusinessObject, the TBO class must implement the following methods:

- getVersion
- getVendorString
- isCompatible
- supportsFeature

The version support features getVersion and isCompatible must have functioning implementations (these are required and used by the Business Object Framework) and it is important to keep the TBO version data up-to-date. Functional implementation of the supportsFeature method is optional: you can provide a dummy implementation that just returns a Boolean value.

For further information see IDfBusinessObject in the Javadocs.

getVersion method

The getVersion method must return a string representing the version of the business object, using the format <major version>.<minor version> (for example 1.10), which can be extended to include as many as four total integers, separated by periods (for example 1.10.2.12). Application Builder returns an error if you try to deploy a TBO that returns an invalid version string.

getVendorString method

The getVendorString method returns a string containing information about the business object vendor, generally a copyright string.

isCompatible method

The isCompatible method takes a String argument in the format <major version>.<minor version> (for example 1.10), which can be extended to include as many as four total integers, separated by periods (for example 1.10.2.12). The isCompatible method, which is intended to be used in conjunction with getVersion, must return true if the TBO is compatible with the version and false if it is not.

supportsFeature method

The supportsFeature method is passed a string representing an application or service feature, and returns true if this feature is supported and false otherwise. Its intention is to allow your application to store lists of features supported by the TBO, perhaps allowing the calling application to switch off features that are not supported.

Support for features is an optional adjunct to mandatory version compatibility support. Features are a convenient way of advertising functionality that avoids imposing complicated version checking on the client. If you choose not to use this method, your TBO can provide a minimal implementation of supportsFeature that just returns a boolean value.

Code the TBO business logic

You can implement business logic in your TBO by adding new methods, or by adding overriding methods of the class that your TBO class extends. When overriding methods, you will most likely want to add custom behavior as pre- or postprocessing before or after a call to super.<methodName>. The following sample shows an override of the IDfSysObject.doCheckin method that writes an entry to the log.

```
protected IDfId doCheckin(boolean fRetainLock,
            String versionLabels,
            String oldCompoundArchValue,
            String oldSpecialAppValue,
            String newCompoundArchValue,
            String newSpecialAppValue,
            Object[] extendedArgs) throws DfException
{
Date now = new Date();
DfLogger.warn(this, now + " doCheckin() called", null, null);
// your preprocessing logic here
return super.doCheckin(fRetainLock,
         versionLabels,
         oldCompoundArchValue,
         oldSpecialAppValue,
         newCompoundArchValue,
         newSpecialAppValue,
         extendedArgs);
// your postprocessing logic here
}
```

Override only methods beginning with do (doSave, doCheckin, doCheckout, and similar). The signatures for these methods are documented in Appendix .

Using a TBO from a client application

A TBO is an extension of a core DFC typed object, so instantiating a TBO is no different from instantiating a core DFC typed object. To instantiate a TBO from a client application, use a method of IDfSession that fetches an object, such as getObject, newObject, or getObjectByQualification. The session object used to generate the TBO must be a managed session, that is, a session that was instantiated using a IDfSessionManager.getSession or newSession factory method.

The following test method exercises a TBO by getting a known object of the TBO type from the repository using getObjectByQualification and checking it in.

```
private void testCheckinOverride(String userName,
                 String docName,
                 String password,
                 String docbaseName,
                 String typeName) throws Exception
{
IDfSessionManager sessionManager = null;
IDfSession docbaseSession = null;
IMyCompanySop mySop = null;
try
{
  // get a managed session
  IDfClient localClient = DfClient.getLocalClient();
  sessionManager = localClient.newSessionManager();
  IDfLoginInfo loginInfo = new DfLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(password);
  sessionManager.setIdentity(docbaseName, loginInfo);
  docbaseSession = sessionManager.getSession(docbaseName);

   // get the test document
  // the query string must uniquely identify it
  StringBuffer bufQual = new StringBuffer(32);
  bufQual.append(typeName)
      .append(" where object_name like '")
      .append(docName).append("'");
   mySop = (IMyCompanySop)
    docbaseSession.getObjectByQualification(bufQual.toString());
  if (mySop == null)
  {
   fail("Unable to locate object with name " + docName);
  }

    // check in document to see whether anything gets
    //written to the log
  if (!mySop.isCheckedOut())
  {
   mySop.checkout();
  }
  if (mySop.isCheckedOut())
  {
   mySop.checkin(false, "MOD_TEST_FILE");
  }
}
```

```
catch (Throwable e)
{
  fail("Failed with exception " + e);
}
finally
{
  if ((sessionManager != null) && (docbaseSession != null))
  {
sessionManager.release(docbaseSession);
  }
}
}
```

Using TBOs from SBOs

If you are instantiating a TBO from an SBO, use the IDfService.getSession method, which returns a session managed by the session manager associated with the SBO. The following sample SBO method gets and returns a TBO:

```
public IDfDocument getDoc( String strRepository, IDfId idDoc )
{
  IDfSession session = null;
  IDfDocument doc = null;
  try
  {
  // calls IDfService.getSession
  session = getSession ( strRepository );
  doc = (IDfDocument)session.getObject( idDoc );
  // note no call to setSessionManager
  // this is not needed in Documentum 6
  }
  finally
  {
  releaseSession( session );
  }
  return doc;
}
```

Note the absence of any call to setSessionManager in the preceding listing. In Documentum 5 a call to setSessionManager was required to *disconnect* the object from the session and place it in the state of the session manager. This allowed the SBO to release the session and return the TBO instance without the object becoming stale (that is, disassociated from its session context). In Documentum 6 there is transparent handling of the association of objects and sessions: if you return an object and release its session, DFC will create a new session for the object when it is required. Legacy calls to setSessionManager will continue to work, but are no longer required for this purpose.

Getting sessions inside TBOs

You can obtain a reference to the session that was originally used to fetch an object using the IDfTypedObject.getSession method. However, when you obtain a session in this way you cannot release it (if you attempt this an exception will be thrown), and you cannot change its repository scope (that is, the name of the repository to which the session maintains a connection). This is because no ownership of the session is implied when you get an existing session using IDfTypedObject.getSession. You can only release sessions that were obtained using a factory method of a session manager. This restriction prevents a misuse of sessions that could lead to abstruse bugs.

If you need to get a session independent of any session associated with the TBO object, you can use IDfTypedObject.getSessionManager to return the session manager associated with the TBO object. You can then get sessions using the factory methods of this session manager (which allows you to use any identities defined in the session manager) and release them (in a finally block) after you have finished using the session.

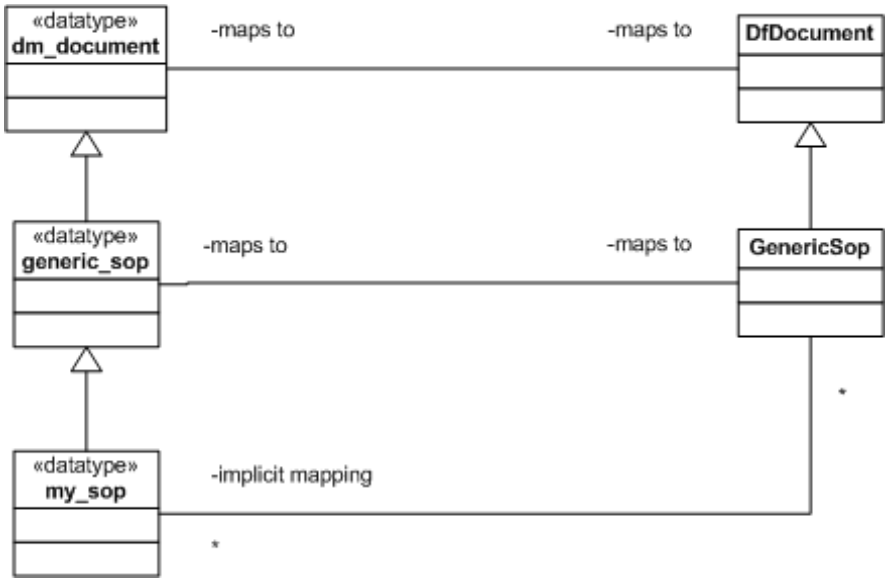
The IDfTypedObject.getSession method returns the session on which the object was originally fetched, which in most situations will be the session maintaining a connection to the repository that holds the object. However, in a distributed environment where DFC is doing work in multiple repositories, the session on which the object was originally fetched and the session to the object's repository may not be the same. In this case you will generally want to obtain the session maintaining a connection to the object's repository. To do this you can use the IDfTypedObject.getObjectSession method instead of getSession. If you specifically want to get the session that was originally used to fetch the object, you can use the getOriginalSession method. The IDfTypedObject.getSession method is a synonym for getOriginalSession.

Inheritance of TBO methods by repository subtypes without TBOs

If you create a repository object type B that does not have a TBO but which inherits from a repository type A that does have a TBO, object B will inherit the behaviors defined by A's TBO. This means that you do not have to create a TBO for each repository subtype unless you need to extend or override the behaviors associated with the parent type. This inheritance mechanism allows administrators to create repository subtypes that behave in a natural way, inheriting the behaviors of the parent type, without the developer having to write another TBO.

For example, suppose you have a repository type generic_sop, which has custom behaviors on checkin defined in a class GenericSop. If an administrator then created a new type my_sop, the new type would inherit the custom checkin behaviors of generic_sop automatically, without the developer needing to implement and test a new TBO for my_sop. Similarly, if a process fetches an object of type my_sop using IDfSession.getObject or a similar method, the getObject method will return an instance of GenericSop.

Figure 9. Inheritance by object subtype without associated TBO

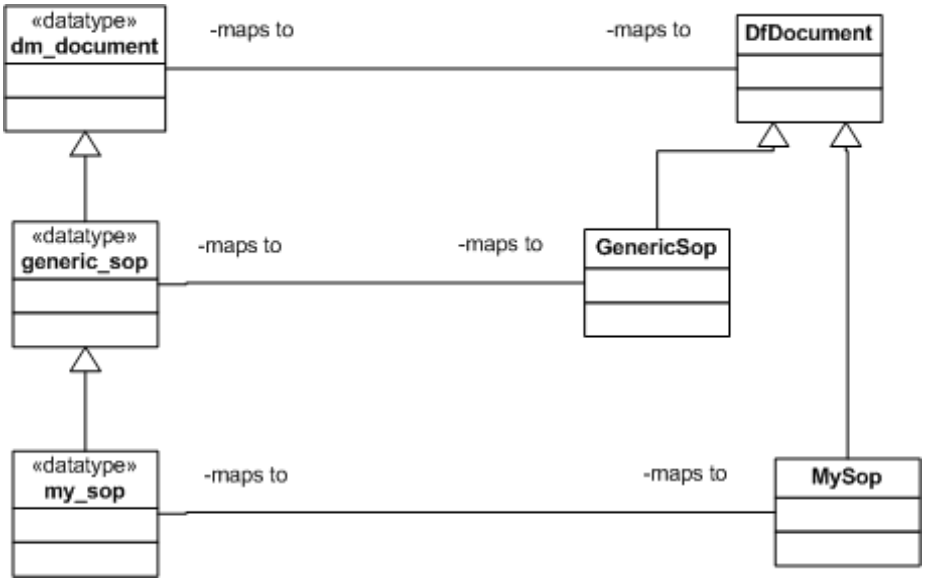


Dynamic inheritance

Dynamic inheritance is a BOF mechanism that modifies the class inheritance of a TBO dynamically at runtime, driven by the hierarchical relationship of associated repository objects. This mechanism enforces consistency between the repository object hierarchy and the associated class hierarchy. It also allows you to design polymorphic TBOs that inherit from different superclasses depending on runtime dynamic resolution of the class hierarchy.

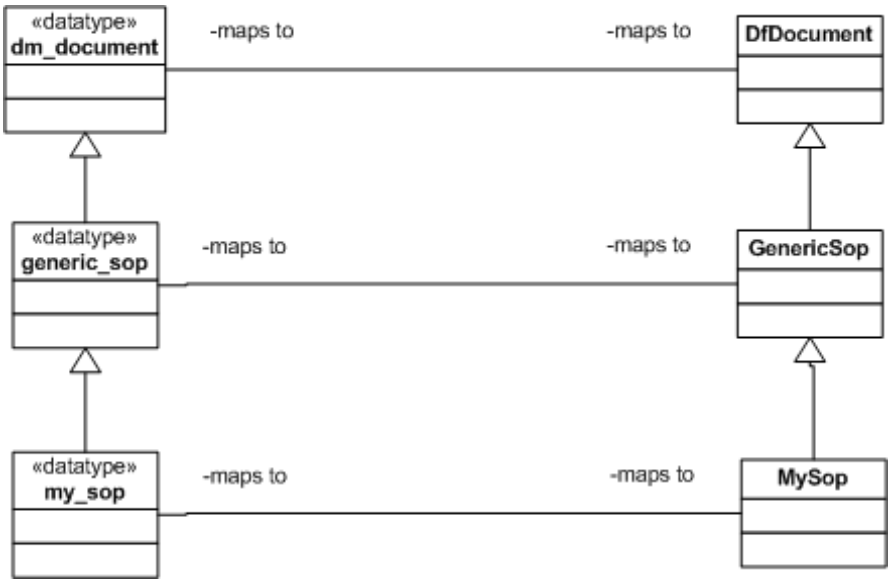
For example, suppose you have the following TBO design, in which repository objects are related hierarchically, but in which the associated TBO classes each inherit from DFDocument:

Figure 10. Design-time dynamic inheritance hierarchies



If dynamic inheritance is enabled, at runtime the class hierarchy is resolved dynamically to correspond to the repository object hierarchy, so that the MySop class inherits from GenericSop:

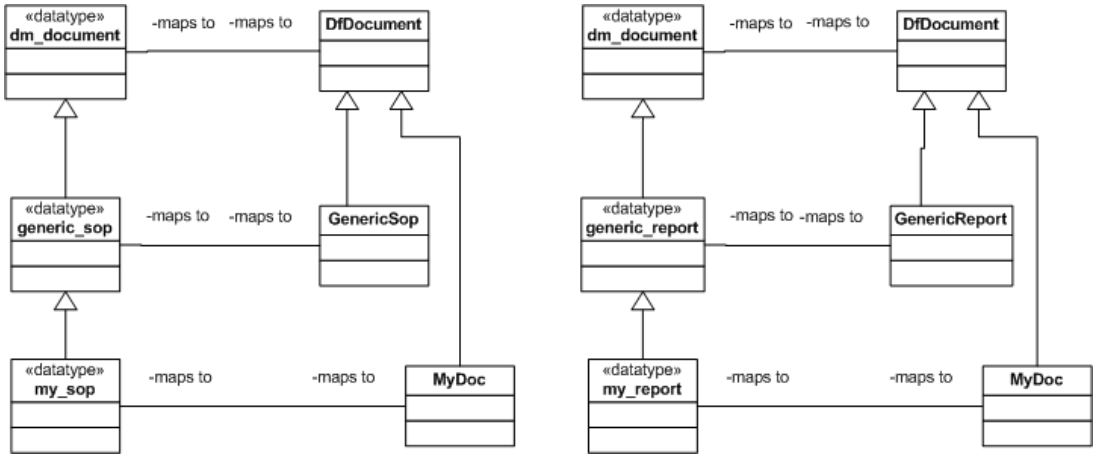
Figure 11. Runtime dynamic inheritance hierarchies



Exploiting dynamic inheritance with TBO reuse

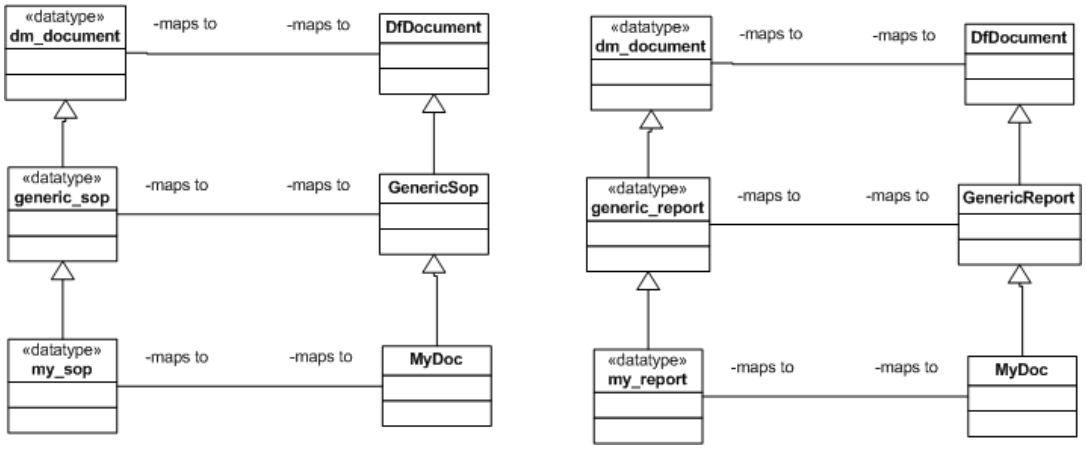
The dynamic inheritance mechanism allows you to design reusable components that exhibit different behaviors at runtime inherited from their dynamically determined superclass. For example, in the following design-time configuration, the MyDoc class is packaged in two TBOs: one in which it is associated with type my_sop, and one in which it is associated with type my_report:

Figure 12. Design-time dynamic inheritance with TBO reuse



At runtime, MyDoc will inherit from GenericSop where it is associated with the my_sop repository object type, and from GenericReport where it is associated with the my_report repository object type.

Figure 13. Runtime dynamic inheritance with TBO reuse



Signatures of Methods to Override

This section contains a list of methods of DfSysObject, DfPersistentObject, and DfTypedObject that are designed to be overridden by implementors of TBOs. All of these methods are protected. You should make your overrides of these methods protected as well.

Table 2. Methods to override when implementing TBOs

Methods of DfSysObject
IDfId doAddESignature (String userName, String password, String signatureJustification, String formatToSign, String hashAlgorithm, String preSignatureHash, String signatureMethodName, String applicationProperties, String passThroughArgument1, String passThroughArgument2, Object[] extendedArgs) throws DfException
IDfId doAddReference (IDfId folderId, String bindingCondition, String bindingLabel, Object[] extendedArgs) throws DfException
void doAddRendition (String fileName, String formatName, int pageNumber, String pageModifier, String storageName, boolean atomic, boolean keep, boolean batch, String otherFileName, Object[] extendedArgs) throws DfException
void doAppendFile (String fileName, String otherFileName, Object[] extendedArgs) throws DfException
IDfCollection doAssemble (IDfId virtualDocumentId, int interruptFrequency, String qualification, String nodesortList, Object[] extendedArgs) throws DfException
IDfVirtualDocument doAsVirtualDocument (String lateBindingValue, boolean followRootAssembly, Object[] extendedArgs) throws DfException
void doAttachPolicy (IDfId policyId, String state, String scope, Object[] extendedArgs) throws DfException
void doBindFile (int pageNumber, IDfId srcId, int srcPageNumber, Object[] extendedArgs) throws DfException
IDfId doBranch (String versionLabel, Object[] extendedArgs) throws DfException
void doCancelScheduledDemote (IDfTime scheduleDate, Object[] extendedArgs) throws DfException
void doCancelScheduledPromote (IDfTime scheduleDate, Object[] extendedArgs) throws DfException
void doCancelScheduledResume (IDfTime schedule, Object[] extendedArgs) throws DfException
void doCancelScheduledSuspend (IDfTime scheduleDate, Object[] extendedArgs) throws DfException

IDfId doCheckin (boolean fRetainLock, String versionLabels, String oldCompoundArchValue, String oldSpecialAppValue, String newCompoundArchValue, String newSpecialAppValue, Object[] extendedArgs)
IDfId doCheckout (String versionLabel, String compoundArchValue, String specialAppValue, Object[] extendedArgs)
void doDemote (String state, boolean toBase, Object[] extendedArgs) throws DfException
void doDestroyAllVersions (Object[] extendedArgs) throws DfException
void doDetachPolicy (Object[] extendedArgs) throws DfException
void doDisassemble (Object[] extendedArgs) throws DfException
boolean doFetch (String currencyCheckValue, boolean usePersistentCache, boolean useSharedCache, Object[] extendedArgs) throws DfException
void doFreeze (boolean freezeComponents, Object[] extendedArgs) throws DfException
void doInsertFile (String fileName, int pageNumber, String otherFileName, Object[] extendedArgs) throws DfException
void doGrant (String accessorName, int accessorPermit, String extendedPermission, Object[] extendedArgs) throws DfException
void doGrantPermit (IDfPermit permit, Object[] extendedArgs) throws DfException
void doLink (String folderSpec, Object[] extendedArgs) throws DfException
void doLock (Object[] extendedArgs) throws DfException
void doMark (String versionLabels, Object[] extendedArgs) throws DfException
void doPromote (String state, boolean override, boolean fTestOnly, Object[] extendedArgs) throws DfException
void doPrune (boolean keepSLabel, Object[] extendedArgs) throws DfException
IDfId doQueue (String queueOwner, String event, int priority, boolean sendMail, IDfTime dueDate, String message, Object[] extendedArgs) throws DfException
void doRefreshReference (Object[] extendedArgs) throws DfException
void doRegisterEvent (String message, String event, int priority, boolean sendMail, Object[] extendedArgs) throws DfException
void doRemovePart (IDfId containmentId, double orderNo, boolean orderNoFlag, Object[] extendedArgs) throws DfException
void doRemoveRendition (String formatName, int pageNumber, String pageModifier, boolean atomic, Object[] extendedArgs) throws DfException
String doResolveAlias (String scopeAlias, Object[] extendedArgs) throws DfException
void doResume (String state, boolean toBase, boolean override, boolean fTestOnly, Object[] extendedArgs) throws DfException

void doRevert (boolean aclOnly, Object[] extendedArgs) throws DfException
void doRevoke (String accessorName, String extendedPermission, Object[] extendedArgs) throws DfException
void doRevokePermit (IDfPermit permit, Object[] extendedArgs) throws DfException
void doSave (boolean saveLock, String versionLabel, Object[] extendedArgs) throws DfException
IDfId doSaveAsNew (boolean shareContent, boolean copyRelations, Object[] extendedArgs) throws DfException
void doScheduleDemote (String state, IDfTime scheduleDate, Object[] extendedArgs) throws DfException
void doSchedulePromote (String state, IDfTime scheduleDate, boolean override, Object[] extendedArgs) throws DfException
void doScheduleResume (String state, IDfTime scheduleDate, boolean toBase, boolean override, Object[] extendedArgs) throws DfException
void doScheduleSuspend (String state, IDfTime scheduleDate, boolean override, Object[] extendedArgs) throws DfException
void doSetACL (IDfACL acl, Object[] extendedArgs) throws DfException
void doSetFile (String fileName, String formatName, int pageNumber, String otherFile, Object[] extendedArgs) throws DfException
void doSetIsVirtualDocument (boolean treatAsVirtual, Object[] extendedArgs) throws DfException
void doSetPath (String fileName, String formatName, int pageNumber, String otherFile, Object[] extendedArgs) throws DfException
void doSuspend (String state, boolean override, boolean fTestOnly, Object[] extendedArgs) throws DfException
void doUnfreeze (boolean thawComponents, Object[] extendedArgs) throws DfException
void doUnlink (String folderSpec, Object[] extendedArgs) throws DfException
void doUnmark (String versionLabels, Object[] extendedArgs) throws DfException
void doUnRegisterEvent (String event, Object[] extendedArgs) throws DfException
void doUpdatePart (IDfId containmentId, String versionLabel, double orderNumber, boolean useNodeVerLabel, boolean followAssembly, int copyChild, String containType, String containDesc, Object[] extendedArgs) throws DfException
void doUseACL (String aclType, Object[] extendedArgs) throws DfException
void doVerifyESignature (Object[] extendedArgs) throws DfException
Methods of DfPersistentObject
IDfRelation doAddChildRelative (String relationTypeName, IDfId childId, String childLabel, boolean isPermanent, String description, Object[] extendedArgs) throws DfException

IDfRelation doAddParentRelative (String relationTypeName, IDfId parentId, String childLabel, boolean isPermanent, String description, Object[] extendedArgs) throws DfException
void doDestroy (boolean force, Object[] extendedArgs) throws DfException
void doRemoveChildRelative (String relationTypeName, IDfId childId, String childLabel, Object[] extendedArgs) throws DfException
void doRemoveParentRelative (String relationTypeName, IDfId parentId, String childLabel, Object[] extendedArgs) throws DfException
void doRevert (boolean aclOnly, Object[] extendedArgs) throws DfException
void doSave (boolean saveLock, String versionLabel, Object[] extendedArgs) throws DfException
void doSignoff (String user, String password, String reason, Object[] extendedArgs) throws DfException
Methods of DfTypedObject
void doAppendString (String attrName, String value, Object[] extendedArgs) throws DfException
String doGetString (String attrName, int valueIndex, Object[] extendedArgs) throws DfException
void doInsertString (String attrName, int valueIndex, String value, Object[] extendedArgs) throws DfException
doSetString (String attrName, int valueIndex, String value, Object[] extendedArgs) throws DfException
void doRemove (String attrName, int beginIndex, int endIndex, Object[] extendedArgs) throws DfException
Methods of DfGroup
boolean doAddGroup (String groupName, Object[] extendedArgs) throws DfException
boolean doAddUser (String userName, Object[] extendedArgs) throws DfException
void doRemoveAllGroups (Object[] extendedArgs) throws DfException
void doRemoveAllUsers (Object[] extendedArgs) throws DfException
boolean doRemoveGroup (String groupName, Object[] extendedArgs) throws DfException
boolean doRemoveUser (String userName, Object[] extendedArgs) throws DfException
void doRenameGroup (String groupName, boolean isImmediate, boolean unlockObjects, boolean reportOnly, Object[] extendedArgs) throws DfException
Methods of DfUser
void doChangeHomeDocbase (String homeDocbase, boolean isImmediate, Object[] extendedArgs) throws DfException
void doRenameUser (String userName, boolean isImmediate, boolean unlockObjects, boolean reportOnly, Object[] extendedArgs) throws DfException

void doSetAliasSet (String aliasSetName, Object[] extendedArgs) throws DfException
void doSetClientCapability (int clientCapability, Object[] extendedArgs) throws DfException
void doSetDefaultACL (String aclName, Object[] extendedArgs) throws DfException
void doSetDefaultFolder (String folderPath, boolean isPrivate, Object[] extendedArgs) throws DfException
void doSetHomeDocbase (String docbaseName, Object[] extendedArgs) throws DfException
void doSetUserOSName (String accountName, String domainName, Object[] extendedArgs) throws DfException
void doSetUserState (int userState, boolean unlockObjects, Object[] extendedArgs) throws DfException

Calling TBOs and SBOs

This section describes special considerations for using TBOs and SBOs.

The BOF deployment mechanism requires you to take steps to ensure that your applications have access to the interfaces of your modules. Refer to Deploying module interfaces, page 91 for more information.

Calling SBOs

This section provides rules and guidelines for instantiating SBOs and calling their methods.

The client application should instantiate a new SBO each time it needs one, rather than reusing one. For example, to call a service during an HTTP request in a web application, instantiate the service, execute the appropriate methods, then abandon the service object.

This approach is thread safe, and it is efficient, because it requires little resource overhead. The required steps to instantiate a service are:

1. Prepare an IDfLoginInfo object containing the necessary login information.
2. Instantiate a session manager object.
3. Call the service factory method.

The following code illustrates these steps:

```
IDfClient client = DfClient.getLocalClient();
IDfLoginInfo loginInfo = new DfLoginInfo();
loginInfo.setUser( strUser );
loginInfo.setPassword( strPassword );
```

```
if( strDomain != null )
   loginInfo.setDomain( strDomain );

IDfSessionManager sMgr = client.newSessionManager();
sMgr.setIdentity( strRepository, loginInfo );
IAutoNumber autonumber = (IAutoNumber)
   client.newService( IAutoNumber.class.getName(), sMgr);
```

An SBO client application uses the newService factory method of IDfClient to instantiate a service:

```
public IDfService newService ( String name, IDfSessionManager sMgr )
    throws DfServiceException;
```

The method takes the service name and a session manager as parameters, and returns the service interface, which you must cast to the specific service interface. The newService method uses the service name to look up the Java implementation class in the registry. It stores the session manager as a member of the service, so that the service implementation can access the session manager when it needs a DFC session.

Returning a TBO from an SBO

The following example shows how to return a TBO, or any repository object, from within an SBO method.

```
public IDfDocument getDoc( String strRepository, IDfId idDoc ) {
  IDfSession session = null;
  IDfDocument doc = null;
  try {
    session = getSession ( strRepository );
    doc = (IDfDocument)session.getObject( idDoc );
    doc.setSessionManager (getSessionManager());
  }
  finally { releaseSession( session ); }
  return doc;
}
```

Because getDoc is a method of an SBO, which must extend DfService, it has access to the session manager associated with the service. The methods getSession, getSessionManager, and releaseSession provide this access.

Refer to Maintaining state in a session manager, page 37 for information about the substantial costs of using the setSessionManager method.

Calling TBOs

Client applications and methods of SBOs can use TBOs. Use a factory method of IDfSession to instantiate a TBO's class. Release the session when you finish with the object.

Within a method of an SBO, use getSession to obtain a session from the session manager. DFC releases the session when the service method is finished, making the session object invalid.

Use the setSessionManager method to transfer a TBO to the control of the session manager when you want to:

- Release the DFC session but keep an instance of the TBO.
- Store the TBO in the SBO state.

Refer to Maintaining state in a session manager, page 37 for information about the substantial costs of using the setSessionManager method.

Sample SBO and TBO implementation

This section presents a straightforward example of a SBO and TBO for you to use as reference for creating your own business objects. The example is trivial — the TBO and SBO work together to set an arbitrary value (flavor) when a document is saved or checked in to the repository. The setFlavor() method represents the location where you can add any business logic required for your application.

ITutorialSBO

Create an interface for the service-based object. This interface provides the empty setFlavorSBO method, to be overridden in the implementation class. All SBOs must extend the IDfService interface.

Example 5-1. ITutorialSBO.java

```
package com.documentum.tutorial

import com.documentum.fc.client.IDfService;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfException;

public interface ITutorialSBO extends IDfService
{
// This is our empty setFlavor method.
    public void setFlavorSBO (IDfSysObject myObj, String flavor)
        throws DfException;
}
```

TutorialSBO

The TutorialSBO class extends the DfService class, which provides fields and methods to provide common functionality for all services.

Example 5-2. TutorialSBO.java

```
package com.documentum.tutorial

import com.documentum.fc.client.DfService;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfException;

public class TutorialSBO extends DfService implements ITutorialSBO {

// Overrides to standard methods.

  public String getVendorString() {
    return "Copyright (c) Documentum, Inc., 2007";
  }

  public static final String strVersion = "1.0";

  public String getVersion() {
    return strVersion;
  }

  public boolean isCompatible(String str) {
    int i = str.compareTo( getVersion() );
    if(i <= 0 )
      return true;
    else
      return false;
  }

// Custom method. This method sets a string value on the system object.
// You can set any number of values of any type (for example, int, double,
// boolean) using similar methods.

  public void setFlavorSBO(IDfSysObject myObj, String myFlavor)
    throws DfException
  {
    myObj.setString("flavor",myFlavor);
  }
}
```

ITutorialTBO

The interface for the TBO is trivial — its only function is to extend the IDfBusinessObject interface, a requirement for all TBOs.

Example 5-3. ITutorialTBO.java

```
package com.documentum.tutorial

import com.documentum.fc.client.IDfBusinessObject;

public interface ITutorialTBO extends IDfBusinessObject
{
    /**
     * No code required (just extends IDfBusinessObject)
     */
}
```

TutorialTBO

The TutorialTBO is the class that pulls the entire example together. This class overrides the doSave(), doSaveEx() and doCheckin() methods of DfSystemObject and uses the setFlavorSBO() method of TutorialSBO to add a string value to objects of our custom type.

Example 5-4. TutorialTBO.java

```
package com.documentum.tutorial

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;

import com.documentum.fc.client.DfDocument;
import com.documentum.fc.client.IDfClient;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;

import com.documentum.fc.common.DfException;
import com.documentum.fc.common.IDfId;

/**
 * simple TBO that overrides the behavior of save(), saveLock() and checkinEx()
 */

public class TutorialTBO extends DfDocument implements ITutorialTBO {

  private static final String strCOPYRIGHT =
    "Copyright (c) Documentum, Inc., 2007";

  private static final String strVERSION = "1.0";

// Instantiate a null client.
  IDfClientX clientx = new DfClientX();
  IDfClient client = null;

// Override standard methods.
  public String getVersion() {
    return strVERSION;
  }
```

```
  public String getVendorString() {
    return strCOPYRIGHT;
  }

  public boolean isCompatible( String s ) {
    return s.equals("1.0");
  }

  public boolean supportsFeature( String s ) {
    String strFeatures = "createhtmlfile";

    if( strFeatures.indexOf( s ) == -1 )
      return false;

    return true;
  }

/*
 * Overridden IDfSysObject methods. These methods intercept the save(),
 * saveLock(), and checkinEx() methods, use the local setFlavor method
 * to attach a String value to the current system object, then pass
 * control to the parent class to complete the operation.
 */

  public void doSave() throws DfException {
    setFlavor("Pistachio");
    super.save();
  }//end save()


  public void doSaveEx() throws DfException
  {
    setFlavor("Banana Fudge");
    super.saveLock();
  }

  public IDfId doCheckin(boolean b, String s, String s1,
    String s2, String s3, String s4) throws DfException
  {
    setFlavor("Strawberry Ripple");
    return super.checkinEx(b, s, s1, s2, s3, s4);
  }

// The setFlavor gets a session, session manager, and local client
// instance, then uses them to access an instance of the custom
// service-based object ITutorialSBO. Once instantiated, we can use
// the setFlavorSBO method to attach the value to the current system
// object.

  private void setFlavor(String myFlavor) throws DfException {
    IDfSession session = getSession();
    IDfSessionManager sMgr = session.getSessionManager();
    client = clientx.getLocalClient();
    ITutorialSBO tutSBOObj =
      (ITutorialSBO)client.newService(ITutorialSBO.class.getName(),sMgr);
    tutSBOObj.setFlavorSBO(this, myFlavor);
```

```
  }
} //end class
```

Deploying the SBO and TBO

You use Documentum Application Builder (DAB) to package your BOF modules. The following procedures walk you through the steps of deploying your Java classes and defining your custom modules in DAB.

1. Compile your SBO and TBO Java classes.
2. Create a separate JAR file for interface and implementation for each TBO and SBO (in this example, four JAR files in total).
3. Create a new type to be used with your TBO.
   a. Open Documentum Application Builder and create a new DocApp.
   b. Choose Insert>Object Type
   c. Double-click the new type to bring up the editor.
   d. Name the type *tutorial_flavor*.
   e. For the label, enter *Tutorial Flavor*.
   f. Set the SuperType to dm_document.
   g. Close the type editor.
   h. Choose Insert>Attribute.
   i. Double click the new attribute to bring up the editor.
   j. Change the Attribute name to *flavor.*
   k. Set the label to *Flavor.*
   l. Set the Data type to String.
   m. Close the attribute editor.
   n. Check in the new Object Type by right-clicking on it and selecting Check in selected object(s).
   o. Click OK.
4. Insert a new module for ITutorialSBO business object.
   a. Choose Insert > Module.
   b. Double-click Module1 to display the module editor.
   c. Change the name of the module to *com.documentum.tutorial.ITutorialSBO*.

d. Choose SBO from the Module type drop-down box. Leave the Check in... field as Minor Version.

e. Click Add, next to Interface JAR(s).

f. Navigate to your ITutorialSBO.jar file and add it.

g. Click Add, next to Implementation JAR(s) and add TutorialSBO.jar.

h. From the Class Name drop-down, choose *com.documentum.tutorial.TutorialSBO*.

i. Check in the module by right-clicking ITutorialSBO under the modules forlder and selecting Check in selected object(s). Then click OK.

5. Insert a new module for ITutorialTBO

a. Choose Insert>Module.

b. Double-click the new module to edit it.

c. Name the module *tutorial_flavor*.

d. Select TBO as the module type.

e. Click Add, next to Interface JAR(s).

f. Navigate to your ITutorialTBO.jar file and add it.

g. Click Add, next to Implementation JAR(s).

h. Navigate to your TutorialsTBO.jar file and add it.

i. From the Class Name drop-down, choose *com.documentum.tutorial.TutorialTBO*.

j. Click the Dependencies tab.

k. Click the Add button below Required Modules.

l. Enter *com.documentum.tutorial.IMySBO* as the name.

m. Click Add and select Copy from Docbase.

n. Navigate into the /System/Modules/SBO folder and double-click ITutorialSBO.

o. Select ITutorialSBO.jar and click Insert.

p. Click OK.

6. Check in the DocApp and close Application Builder.

To see your modules in action, use Webtop to create an object of the tutorial_flavor type. Check the item out and save it, then look at the complete document properties to see the flavor property update. Check in the file to update the value again.

Aspects

Aspects are a mechanism for adding behavior and/or attributes to a Documentum object *instance* without changing its type definition. They are similar to TBOs, but they are not associated with any one document type. Aspects also are late-bound rather than early-bound objects, so they can be added to an object or removed as needed.

Aspects are a BOF type (dmc_aspect_type). Like other BOF types, they have these characteristics:

- Aspects are installed into a repository.
- Aspects are downloaded on demand and cached on the local file system.
- When the code changes in the repository, aspects are automatically detected and new code is "hot deployed" to the DFC runtime.

Examples of usage

One use for aspects would be to attach behavior and attributes to objects at a particular time in their lifecycle. For example, you might have objects that represent customer contact records. When a contact becomes a customer, you could attach an aspect that encapsulates the additional information required to provide customer support. This way, the system won't be burdened with maintenance of empty fields for the much larger set of prospective customers.

If you defined levels of support, you might have an additional level of support for "gold" customers. You could define another aspect reflecting the additional behavior and fields for the higher level of support, and attach them as needed.

Another scenario might center around document retention. For example, your company might have requirements for retaining certain legal documents (contracts, invoices, schematics) for a specific period of time. You can attach an aspect that will record the date the document was created and the length of time the document will have to be retained. This way, you are able to attach the retention aspect to documents regardless of object type, and only to those documents that have retention requirements.

You will want to use aspects any time you are introducing cross-type functionality. You can use them when you are creating elements of a common application infrastructure. You can use them when upgrading an existing data model and you want to avoid performing a database upgrade. You can use them any time you are introducing functionality on a per-instance basis.

General characteristics of aspects

Applications attach aspects to an object instance. They are not a "per application" customization — an aspect attached by one application will customize the instance across all applications. They have an affinity for the object instance, not for any particular application.

A persistent object instance may have multiple uniquely named aspects with or without attributes. Different object instances of the same persistent type may have different sets of aspects attached.

Aspects define attributes and set custom values that can be attached to a persistent object. There are no restrictions on the attributes that an aspect can define: they can be single or repeating, and of any supported data type. An aspect with an attribute definition can be attached to objects of any type — they provide a natural extension to the base type. Aspect attributes should be fully qualified as *aspect_name.attribute_name* in all setters and getters.

Attributes defined by one aspect can be accessed by other aspects. All methods work on aspect attributes transparently: fetching an object retrieves both the basic object attributes and any aspect attributes; destroying an object deletes any attached aspect attributes.

Creating an aspect

Aspects are created in a similar fashion to other BOF modules.

1. Decide what your aspect will provide: behavior, attributes, or both.
2. Create the interface and implementation classes. Write any new behavior, override existing behavior, and provide getters and setters to your aspect attributes.
3. Deploy the aspect module. For details, see the *Documentum Foundation Classes Release Notes Version 6*.

As an example, we'll walk through the steps of implementing a simple aspect. Our aspect is designed to be attached to a document that stores contact information. The aspect identifies the contact as a customer and indicates the level of service (three possible values — customer, silver, gold). It will also track the expiration date of the customer's subscription.

Creating the aspect interface

Define the new behavior for your aspect in an interface. In this case, we'll add getters and setters for two attributes: *service_level* and *expiration_date*.

Example 5-5. ICustomerServiceAspect.java

```
package dfctestenvironment;
```

```
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.IDfTime;

public interface ICustomerServiceAspect {

// Behavior for extending the expiration date by <i>n</i> months.
public String extendExpirationDate(int months)
  throws DfException;

// Getters and setters for custom attributes.
  public abstract IDfTime getExpirationDate()
    throws DfException;
  public abstract String getServiceLevel()
    throws DfException;
  public abstract void setExpirationDate(IDfTime expirationDate)
    throws DfException;
  public abstract void setServiceLevel(String level)
    throws DfException;
}
```

Creating the aspect class

Now that we have our interface, we can implement it with a custom class.

Example 5-6. CustomerServiceAspect.java

```
package dfctutorialenvironment;

import com.documentum.fc.client.DfDocument;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfTime;
import com.documentum.fc.common.IDfTime;
import dfctestenvironment.ICustomerServiceAspect;

import java.util.GregorianCalendar;

public class CustomerServiceAspect
    extends DfDocument
    implements ICustomerServiceAspect
{
    public String extendExpirationDate(int months) {
        try {

// Get the current expiration date.
            IDfTime startDate = getExpirationDate();

// Convert the expiration date to a calendar object.
            GregorianCalendar cal = new GregorianCalendar(
                startDate.getYear(),
                startDate.getMonth(),
                startDate.getDay()
            );
```

```
// Add the number of months -1 (months start counting from 0).
            cal.add(GregorianCalendar.MONTH,months-1);

// Convert the recalculated date to a DfTime object.
            IDfTime endDate = new DfTime (cal.getTime());

// Set the expiration date and return results.
            setExpirationDate(endDate);
             return "New expiration date is " + endDate.toString();
         }
         catch (Exception ex) {
             ex.printStackTrace();
             return "Exception thrown: "  + ex.toString();
         }
   }
// Getters and setters for the expiration_date and service_level custom attributes.
   public IDfTime getExpirationDate() throws DfException
   {
      return getTime("customer_service_aspect.expiration_date");
   }
   public String getServiceLevel() throws DfException
   {
      return getString("customer_service_aspect.service_level");
   }
   public void setExpirationDate(IDfTime expirationDate) throws DfException {
      setTime("customer_service_aspect.expiration_date", expirationDate);
   }
   public void setServiceLevel (String serviceLevel) throws DfException {
      setString("customer_service_aspect.service_level", serviceLevel);
   }
}
```

Deploy the customer service aspect

For details on deploying aspect modules, please see the *Documentum Foundation Classes Release Notes Version 6*

TestCustomerServiceAspect

Once you have compiled and deployed your aspect classes and defined the aspect on the server, you can use the class to set and get values in the custom aspect, and to test the behavior for adjusting the expiration date by month. This example is compatible with the sample environment described in Chapter Chapter 3, Creating a Test Application.

Example 5-7. TestCustomerServiceAspect.java

```
package dfctestenvironment;
```

```
  import com.documentum.com.DfClientX;
  import com.documentum.com.IDfClientX;

  import com.documentum.fc.client.IDfDocument;
  import com.documentum.fc.client.IDfSession;
  import com.documentum.fc.client.aspect.IDfAspects;

  import com.documentum.fc.common.DfId;
  import com.documentum.fc.common.DfTime;
  import com.documentum.fc.common.IDfTime;

public class TestCustomerServletAspect {
  public TestCustomerServletAspect() {
  }

  public String attachCustomerServiceAspect(
    IDfSession mySession,
    String docId,
    String serviceLevel,
    String expirationDate
  )
  {
// Instantiate a client.
    IDfClientX clientx = new DfClientX();

    try {
      String result = "";

// Get the document instance using the document ID.
      IDfDocument doc =
        (IDfDocument) mySession.getObject(new DfId(docId));

// Convert the expirationDate string to an IDfTime object.
// DF_TIME_PATTERN1 is "mm/dd/yy"
      IDfTime ed =
        clientx.getTime(expirationDate,DfTime.DF_TIME_PATTERN1);

// Attach the aspect.
      ((IDfAspects)doc).attachAspect("customer_service_aspect",null);

// Save the document.
      doc.save();

// Get the document with its newly attached aspect.
      doc = (IDfDocument)mySession.getObject(doc.getObjectId());

// Set the aspect values.
      doc.setString("customer_service_aspect.service_level",
        serviceLevel);
      doc.setTime("customer_service_aspect.expiration_date", ed);

// Save the document.
      doc.save();

      result = "Document " + doc.getObjectName() +
      " set to service level " +
      doc.getString("customer_service_aspect.service_level") +
```

```
      " and will expire " +
      doc.getTime("customer_service_aspect.expiration_date").toString()
      +".";
      return result;
    }
    catch (Exception ex) {
      ex.printStackTrace();
      return "Exception thrown:" + ex.toString();
    }
  }

  public String extendExpirationDate(
    IDfSession mySession,
    String docId,
    int months)
  {
// Instantiate a client.
    IDfClientX clientx = new DfClientX();
    try {
      String result = "";

// Get the document instance using the document ID.
      IDfDocument doc =
        (IDfDocument) mySession.getObject(new DfId(docId));

// Call the extendExpirationDate method
      result = ((ICustomerServiceAspect)doc).extendExpirationDate(months);

// Save the document.
      doc.save();

      return result;
    }
    catch (Exception ex) {
      ex.printStackTrace();
      return "Exception thrown.";
    }
  }
  public String setExpirationDate(
    IDfSession mySession,
    String docId,
    String expDate)
  {
  // Instantiate a client.
    IDfClientX clientx = new DfClientX();

    try {
      String result = "";
      IDfTime expirationDate = clientx.getTime(
        expDate,
        IDfTime.DF_TIME_PATTERN1
      );
// Get the document instance using the document ID.
      IDfDocument doc =
        (IDfDocument) mySession.getObject(new DfId(docId));

// Set the date using the time object.
```

```
      doc.setTime("customer_service_aspect.expiration_date",
        expirationDate);

// Save the document and return the result.
      doc.save();
      result = "Expiration date set to " +
      doc.getTime("customer_service_aspect.expiration_date").toString();
      return  result;

    }
    catch (Exception ex) {
      ex.printStackTrace();
      return "Exception thrown.";
    }
  }

  public String setServiceLevel(
    IDfSession mySession,
    String docId,
    String serviceLevel
  )
  {
    String result = "";
// Instantiate a client.
    IDfClientX clientx = new DfClientX();

    try {

// Get the document instance using the document ID.
  IDfDocument doc =
        (IDfDocument) mySession.getObject(new DfId(docId));

// Set the date using the time object.
      doc.setString("customer_service_aspect.service_level",
        serviceLevel);

// Save the document.
      doc.save();
      result = "Service Level set to " +
        doc.getString("customer_service_aspect.service_level")
        + ".";
      return result;
    }
    catch (Exception ex) {
      ex.printStackTrace();
      return "Exception thrown.";
    }
  }
}
```

Using aspects in a TBO

Attaching and detaching aspects from within a TBO or aspect requires the use of callbacks to execute aspect methods or access an aspect's attributes. The attach/detach is encapsulated within the object instance. Aspect behavior and attributes are not available until the object has been fully initialized.

To use aspects within a TBO or aspect, you must

- Create a class that implements IDfAttachAspectCallback or IDfDetachAspectCallback.
- Implement the doPostAttach() or doPostDetach() method of the callback interface.

In the following examples, the attachAspect() method is called inside the overridden version of the doSave() method, but only if it is not already attached. Attempting to attach an aspect more than once will result in an exception. The MyAttachCallback callback implementation sets the attributes *retained_since* and *years_to_retain* in the new aspect.

Example 5-8. Code snippet from MySopTBO.java

```
...
//Override doSave()
protected synchronized void doSave(
  boolean saveLock,
  String v,
  Object[] args)
{
  if (this.getAspects().findString("my_retention_aspect") < 0) {
    MyAttachCallback myCallback = new MyAttachCallback();
    this.attachAspect("my_retention_aspect", myCallback);
  }
  super.doSave(saveLock, v, args);
}
```

Example 5-9. MyAttachCallback.java

```
public class MyAttachCallback implements IDfAttachAspectCallback
{
  public void doPostAttach(IDfPersistentObject obj) throws Exception
  {
    obj.setTime("my retention_aspect.retained_since", Datevalue);
    obj.setInt("my_retention_aspect.years_to_retain", Years);
    obj.save();
  }
}
```

Using DQL with aspects

Once an aspect has been defined in the repository, you can use DQL (Documentum Query Language) instructions to add, modify, or drop attributes. Aspects can be modified using the following commands:

```
ALTER ASPECT aspect_name_ ADD attribute_def[,attribute_def][OPTIMIZEFETCH|
NO OPTIMIZEFETCH] [PUBLISH]

ALTER ASPECT aspect_name MODIFY attribute_modifier_clause[, attribute_modifier_clause]
[PUBLISH]

ALTER ASPECT aspect_name DROP attribute_name[, attribute_name] [PUBLISH]

ALTER ASPECT aspect_name DROP ALL [PUBLISH]
```

The syntax for the *attribute_def* and *attribute_modifier_clause* is the same as the syntax for the ALTER TYPE statement.

The OPTIMIZEFETCH keyword in the ALTER ASPECT statement causes the aspect attribute values to be stored alongside the object to which the aspect is attached. This results in reduced database queries and can improve performance. The aspect attributes are duplicated into the object's property bag (a new Documentum 6 feature). The trade off is the increased storage cost of maintaining more than one instance of the attribute value.

There are some limitations when using the DQL SELECT statement. If you are selecting a repeating aspect attribute, r_object_id should be included in the selected list. Repeating aspect attributes cannot be in the select list of a sub_query. Repeating aspect attributes from different aspects cannot be referenced in an expression. If the select list contains an aggregate function on a repeating aspect attribute, then the 'GROUP BY' clause, if any, must be on r_object_id.

Data Dictionary-specific clauses are available in the ALTER ASPECT statement, but the semantics (validations, display configuration, etc.) are not supported in the Documentum 6 release.

Enabling aspects on object types

By default, dm_sysobject and its sub-types are enabled for aspects. This includes any custom object sub-types. Any non-sysobject application type can be enabled for use with aspects using the following syntax.

```
ALTER TYPE type_name ALLOW ASPECTS
```

Default aspects

Type definitions can include a default set of aspects. This allows you to modify the data model and behavior for future instances. It also ensures that specific aspects are attached to all selected object instances, no matter which application creates the object. The syntax is

```
ALTER TYPE type_name [SET | ADD | REMOVE] DEFAULT ASPECTS aspect_list
```

the aspect_list value is a comma-separated list of dmc_aspect_type object_name values. No quotes are necessary, but if you choose to use quotes they must be single quotes and surround the entire list. For example, aspect_list could be a single value such as *my_retention_aspect*, or it could be multiple values specified as *'my_aspect_name1, my_aspect_name2'* or *my_aspect_name1, my_aspect_name2*.

Referencing aspect attributes from DQL

All aspect attributes in a DQL statement must be fully qualified as *aspect_name.attribute_name.* For example:

```
SELECT r_object_id, my_retention_aspect.retained_since
FROM my_sop WHERE my_retention_aspect.years_to_retain = 10
```

If more than one type is specified in the FROM clause of a DQL statement, aspect attributes should be further qualified as *type_name.aspect_name.attribute_name* OR *alias_name.aspect_name.attribute_name*.

Aspect attributes specified in a DQL statement appear in a DQL statement like a normal attribute, wherever legally permitted by the syntax.

Full-text index

By default, full-text indexing is turned off for aspects. You can control which aspect attributes have full-text indexing using the following DQL syntax.

```
ALTER ASPECT aspect_name FULLTEXT SUPPORT ADD | DROP a1, a2,...

ALTER ASPECT aspect_name FULLTEXT SUPPORT ADD | DROP ALL
```

Object replication

Aspect attributes can be replicated in a second repository just as normal attributes are replicated ("dump and load" procedures). However, the referenced aspects must be available on the target repository.

Chapter 6

Support for Other Documentum Functionality

Because DFC is the principal low level interface to all Documentum functionality, there are many DFC interfaces that this manual covers only superficially. They provide access to features that other documentation covers in more detail. For example, the *Server Fundamentals* manual describes virtual documents and access control. The DFC Javadocs provide the additional information necessary to enable you to take advantage of those features. Similarly, the Enterprise Content Integration (ECI) Services product includes extensive capabilities for searching Documentum repositories. The DFC Javadocs provide information about how to use DFC to access that functionality.

This chapter introduces some of the Documentum functionality that you can use DFC to access. It contains the following major sections:

Security Services

Content Server provides a variety of security features. From the DFC standpoint, they fall into the following categories:

- User authentication

  Refer to for more information.

- Object permissions

  Refer to the *Server Fundamentals* manual and the DFC Javadocs for IDfACL, IDfPermit, and other interfaces for more information.

DFC also provides a feature related to folder permissions. Users may have permission to view an object but not have permission to view all of the folders to which it is linked. The IDfObjectPath interface and the getObjectPaths method of IDfSession provide a powerful and flexible mechanism for finding paths for which the given user has the necessary permissions. Refer to the Javadocs for more details.

XML

Chapter 4, Working with Document Operations provides some information about working with XML. DFC provides substantial support for the Documentum XML capabilities. Refer to *XML Application Development Guide* for details of how to use these capabilities.

Virtual Documents

Chapter 4, Working with Document Operations provides some information about working with virtual documents. Refer to *Server Fundamentals* and the DFC Javadocs for the IDfVirtualDocument interface for much more detail.

Workflows

The *Server Fundamentals* manual provides a thorough treatment of the concepts underlying workflows. DFC provides interfaces to support the construction and use of workflows, but there is almost no reason to use those interfaces directly. The workflow manager and business process manager software packages handle all of those details.

Individual workflow tasks can have methods associated with them. You can program these methods in Java and call DFC from them. These methods run on the method server, an application server that resides on the Content Server machine and is dedicated to running Content Server methods. The code for these methods resides in the repository's registry as modules. Modules and registries, page 88 provides more information about registries and modules.

The com.documentum.fc.lifecycle package provides the following interfaces for use by modules that implement lifecycle actions:

- IDfLifecycleUserEntryCriteria to implement userEntryCriteria.
- IDfLifecycleUserAction to implement userAction.
- IDfLifecycleUserPostProcessing to implement userPostProcessing

There is no need to extend DfService, but you can do so. You need only implement IDfModule, because lifecycles are modules, not SBOs.

Document Lifecycles

The *Server Fundamentals* manual provides information about lifecycles. There are no DFC interfaces for constructing document lifecycles. Application Builder (DAB) includes a lifecycle editor for that purpose. You can define actions to take place at various stages of a document's lifecycle. You can code these in Java to run on the Content Server's method server. Such Java methods must implement the appropriate interfaces from the following:

- IDfLifecycleAction.java
- IDfLifecycleUserAction.java
- IDfLifecycleUserEntryCriteria.java
- IDfLifecycleUserPostProcessing.java
- IDfLifecycleValidate.java

The code for these methods resides in the repository's registry (see Modules and registries, page 88) as modules.

Validation Expressions in Java

When you create a repository type, you can associate constraints with it. Some of these constraints are expressions involving either a single attribute or combinations of attributes of the type. These reside in the repository's data dictionary, where client programs can access them to enforce the constraints. Content Server does not enforce constraints defined in the data dictionary.

In earlier versions of DFC these constraints were exclusively Docbasic expressions. Since DFC 5.3, you can provide Java translations of the constraint expressions. If a Java version of a constraint exists, DFC uses it in preference to the Docbasic version of the same constraint. This usually results in a substantial performance improvement.

The key points about this feature are the following:

- You must take steps to use it.

  If you do nothing, DFC continues to use the existing Docbasic expressions. A server script enables the feature by creating the necessary types and the methods for creating Java translations.
- The migration is incremental and non-destructive.

  You can migrate all, none, or any portion in between of the Docbasic expression evaluation in a Content Server repository to Java. The Docbasic versions remain in place. You can disable any specific Java translations and revert to using Docbasic for that function or that object type.

- Translations are available for all Docbasic functions that you are likely to use in validation expressions.

  We do not provide Java translations of operating system calls, file system access, COM and DDE functions, print or user interface functions, and other similar functions. We do not provide Java translations of financial functions.

Search Service

The DFC search service replaces prior mechanisms for building and running queries. You can use the IDfQuery interface, which is not part of the search service, for simple queries. The search service provides the ability to run searches across multiple Documentum repositories and, in conjunction with the Enterprise Content Integration (ECI) Services product, external repositories as well.

The Javadocs for the com.documentum.fc.client.search package provide a description of how to use this capability.

Appendix A

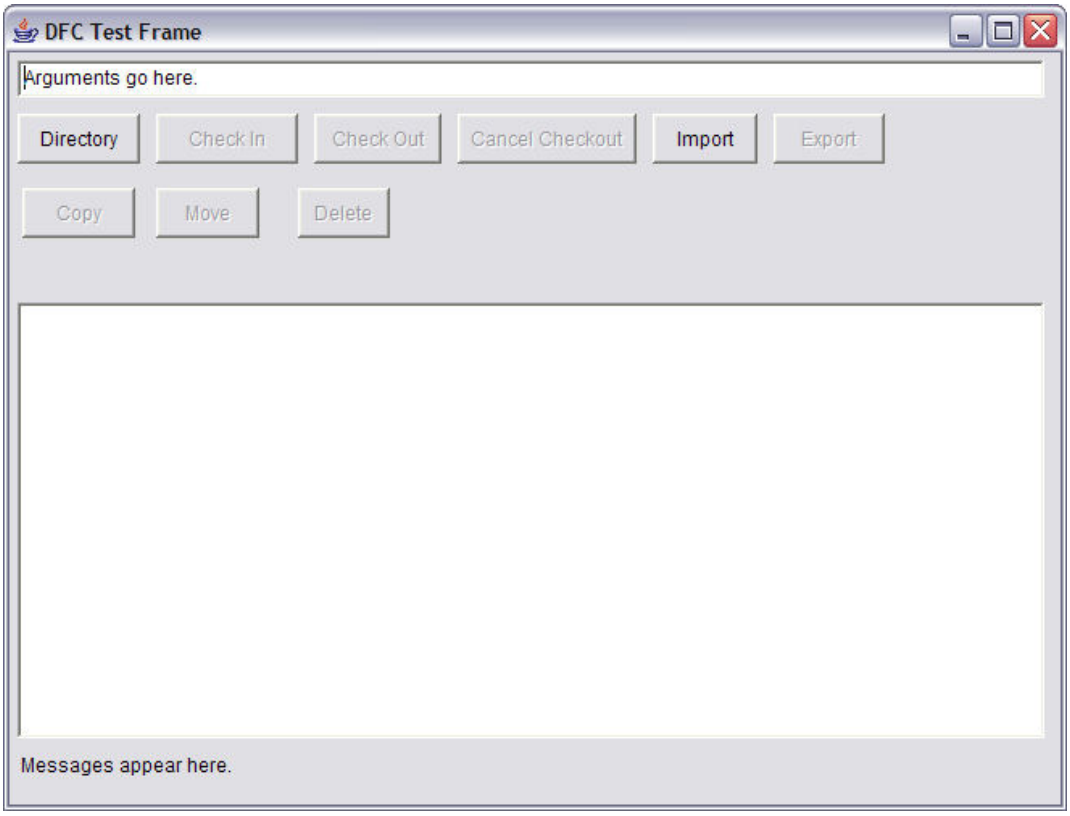
Sample Test Interface

This is a very simple Java application you can use to try out the code samples in Chapter 4, Working with Document Operations. It was created using AWT controls for ease of use. The IDE used to create this application was Oracle JDeveloper, and so it has some of that program's idiosyncrasies. You may want to create a similar sample application using an IDE with which you're familiar, and reference the button handling routines to create your own.

To run the application, write and compile the enhanced DfcTestFrame.java listing in this appendix along with the sample classes from these sections of this manual:

- The TutorialSessionManager class, page 39
- The DfcTutorialApplication class, page 44
- Cancelling checkout, page 64
- Checking in, page 61
- Checking out, page 58
- Copying, page 72
- Deleting, page 76
- Exporting, page 70
- Importing, page 67

Figure 14. Sample interface with buttons for typical document manipulation commands

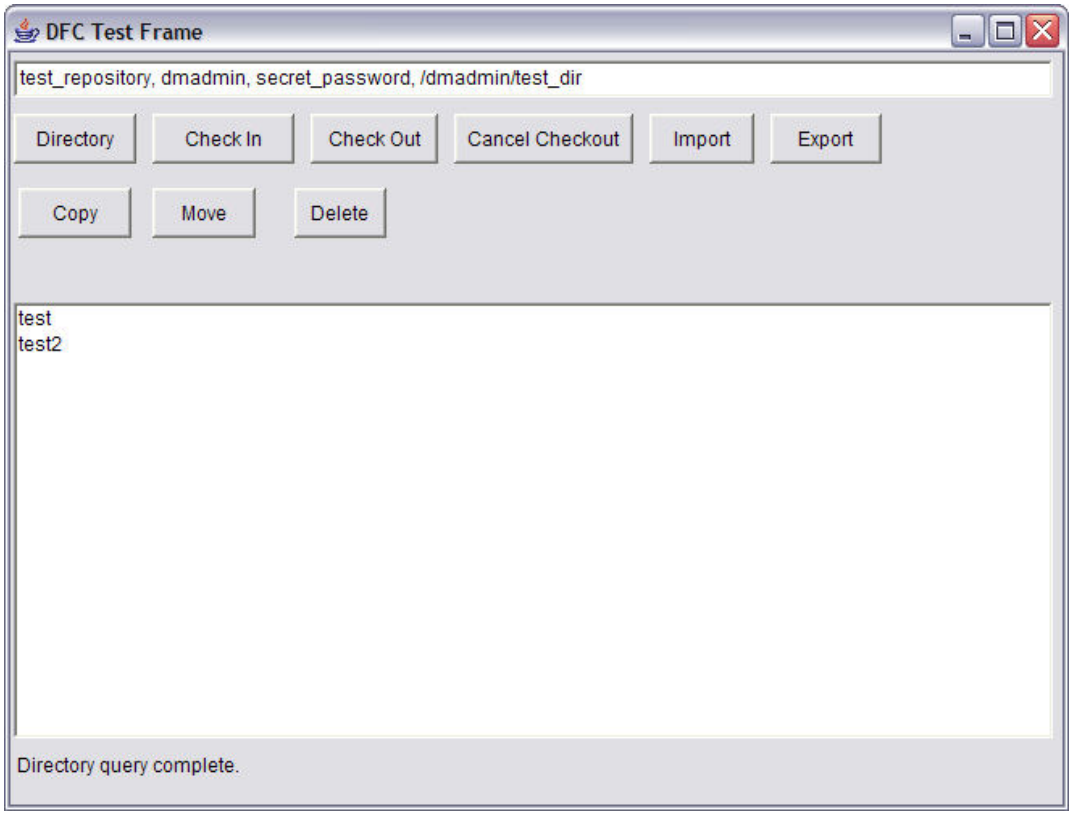


The rationale behind this test interface is that while it is reasonable to expect that you might have to enter file paths to try out the commands, you shouldn't have to enter internal document IDs. This interface primarily performs the lookup of the document IDs so that you don't have to find them yourself and enter them.

This is not in any way intended to be a sample of an interface you might create yourself for your users. The perfect "sample" UI application is Webtop, which demonstrates how Documentum engineers feel our content management features should be implemented.

The Directory and Import buttons are enabled when you first open the application. Once you have a directory listing, the other buttons are enabled.

Figure 15. DFC Test Frame with File Listing



Enter the arguments in the field at the top of the window, then click the button for the operation you want to test. The arguments are as follows.

Table 3. Arguments for sample commands used with the DFC test frame

Command	Arguments
Directory	repository, userName, password, directoryPath
Check In	repository, userName, password, directoryPath. Select an item from the file list that has already been checked out and is ready to check in.
Check Out	repository, userName, password, directoryPath. Select an item from the file list you want to check out.
Cancel Checkout	repository, userName, password, directoryPath. Select an item from the file list that has already been checked out.

Import	repository, userName, password, desitnationDirectoryPath, fullLocalFilePath.
Export	repository, userName, password, directoryPath, targetLocalDirectory. Select an item from the file list to export.
Copy	repository, userName, password, sourceDirectory, destinationDirectory. Select a document from the list you want to copy.
Move	repository, userName, password, sourceDirectory, and destinationDirectory. Select a document from the list that you want to move.
Delete	repository, userName, password, directory. Select a document from the list that you want to delete.

Example A-1. DFCTestFrame.java (enhanced)

```
package dfctutorialenvironment;

import com.documentum.fc.client.IDfCollection;
import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfTypedObject;

import java.awt.Button;
import java.awt.Frame;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.Label;
import java.awt.List;
import java.awt.Rectangle;
import java.awt.SystemColor;
import java.awt.TextField;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.util.StringTokenizer;
import java.util.Vector;


public class DfcTestFrame extends Frame {

// Generate UI elements

  private Label label_results = new Label();
  private TextField textField_arguments = new TextField();
  private List list_id = new List();
```

```
  private Button button_directory = new Button();

// Definitions of operation buttons.
  private Button button_checkOut = new Button();
  private Button button_cancelCheckout = new Button();
  private Button button_checkIn = new Button();
  private Button button_import = new Button();
  private Button button_export = new Button();
  private Button button_copy = new Button();
  private Button button_move = new Button();
  private Button button_delete = new Button();


// This Vector is used to store the list of internal document IDs
// returned by the directory query.
  private Vector m_fileIDs = new Vector();

  private GridBagLayout gridBagLayout1 = new GridBagLayout();

  public DfcTestFrame() {
    try {
      jbInit();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

// Initialize UI components

  private void jbInit() throws Exception {
    this.setLayout(gridBagLayout1);
    this.setTitle( "DFC Test Frame" );
    this.setBackground( SystemColor.control );
    this.setBounds(new Rectangle(10, 10, 660, 500));

    textField_arguments.setText("Arguments go here.");
    label_results.setText("Messages appear here.");
    button_directory.setLabel("Directory");
    button_directory.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            button_directory_actionPerformed(e);
          }
        });
    this.add(button_delete,
         new GridBagConstraints(
           3, 2, 1, 1, 0.0, 0.0,
           GridBagConstraints.CENTER,
           GridBagConstraints.NONE,
           new Insets(15, 0, 0, 0), 7, 8)
         );
    this.add(button_move,
         new GridBagConstraints(
           1, 2, 1, 1, 0.0, 0.0,
           GridBagConstraints.CENTER,
           GridBagConstraints.NONE,
           new Insets(15, 10, 0, 0), 22, 8)
         );
```

```
    this.add(button_copy,
         new GridBagConstraints(
           0, 2, 1, 1, 0.0, 0.0,
           GridBagConstraints.CENTER,
           GridBagConstraints.NONE,
           new Insets(15, 10, 0, 0), 28, 8)
         );
    this.add(textField_arguments,
         new GridBagConstraints(
           0, 0, 10, 1, 1.0, 0.0,
           GridBagConstraints.WEST,
           GridBagConstraints.HORIZONTAL,
           new Insets(5, 10, 0, 15), 527, 0)
         );
    this.add(list_id,
         new GridBagConstraints(
           0, 3, 10, 1, 1.0, 1.0,
           GridBagConstraints.CENTER,
           GridBagConstraints.BOTH,
           new Insets(40, 10, 0, 15), 372, 160)
         );
    this.add(button_directory,
         new GridBagConstraints(
           0, 1, 1, 1, 0.0, 0.0,
           GridBagConstraints.CENTER,
           GridBagConstraints.NONE,
           new Insets(10, 10, 0, 0), 14, 8)
         );
    this.add(label_results,
         new GridBagConstraints(
           0, 4, 9, 1, 0.0, 0.0,
           GridBagConstraints.WEST,
           GridBagConstraints.NONE,
           new Insets(0, 10, 7, 0), 507, 11)
         );
    button_checkOut.setLabel("Check Out");
    button_checkOut.setEnabled(false);
    button_checkOut.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            button_checkOut_actionPerformed(e);
          }
        });
    button_cancelCheckout.setLabel("Cancel Checkout");
    button_cancelCheckout.setEnabled(false);
    button_cancelCheckout.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            button_cancelCheckout_actionPerformed(e);
          }
        });
    button_checkIn.setLabel("Check In");
    button_checkIn.setEnabled(false);
    button_checkIn.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            button_checkIn_actionPerformed(e);
          }
        });
    button_import.setLabel("Import");
```

```
button_import.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button_import_actionPerformed(e);
      }
    });
button_export.setLabel("Export");
button_export.setEnabled(false);
button_export.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button_export_actionPerformed(e);
      }
    });
button_copy.setLabel("Copy");
button_copy.setEnabled(false);
button_copy.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button_copy_actionPerformed(e);
      }
    });
button_move.setLabel("Move");
button_move.setEnabled(false);
button_move.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button_move_actionPerformed(e);
      }
    });
button_delete.setLabel("Delete");
button_delete.setEnabled(false);
button_delete.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        button_delete_actionPerformed(e);
      }
    });
this.add(button_checkOut,
     new GridBagConstraints(
       3, 1, 2, 1, 0.0, 0.0,
       GridBagConstraints.CENTER,
       GridBagConstraints.NONE,
       new Insets(10, 10, 0, 0), 8, 8)
     );
this.add(button_export,
     new GridBagConstraints(
       7, 1, 1, 1, 0.0, 0.0,
       GridBagConstraints.CENTER,
       GridBagConstraints.NONE,
       new Insets(10, 10, 0, 0), 21, 8)
     );
this.add(button_import,
     new GridBagConstraints(
       6, 1, 1, 1, 0.0, 0.0,
       GridBagConstraints.CENTER,
       GridBagConstraints.NONE,
       new Insets(10, 10, 0, 0), 16, 8)
     );
this.add(button_checkIn,
     new GridBagConstraints(
       1, 1, 2, 1, 0.0, 0.0,
```

```
            GridBagConstraints.CENTER,
            GridBagConstraints.NONE,
            new Insets(10, 10, 0, 0), 26, 8)
          );
    this.add(button_cancelCheckout,
          new GridBagConstraints(
            5, 1, 1, 1, 0.0, 0.0,
            GridBagConstraints.CENTER,
            GridBagConstraints.NONE,
            new Insets(10, 10, 0, 0), 3, 8)
          );
  }

// Handler for the Directory button. To use the button, enter the arguments
// repository_name, user_name, password, directory_path in the arguments field.

 private void button_directory_actionPerformed(ActionEvent e) {

 // Get the arguments and assign variable values.

     String temp = textField_arguments.getText();
     StringTokenizer st = new StringTokenizer(temp, ",");
     String repository = st.nextToken().trim();
     String userName = st.nextToken().trim();
     String password = st.nextToken().trim();
     String directory = st.nextToken().trim();

// Create an empty session.
     IDfSession mySession = null;

// Create a TutorSessionManager object.
     TutorialSessionManager mySessMgr = null;

     String docId = "";
     String docName = "";

     try {

         label_results.setText("Accessing directory information....");

// Populate the custom session manager object.

         mySessMgr = new TutorialSessionManager(repository, userName, password);

// Get a session and assign it to the mySession variable.
         mySession = mySessMgr.getSession();

// Create a folder object based on the directory_path argument.
         IDfFolder myFolder = mySession.getFolderByPath(directory);

// Get the contests of the folder as a collection.
         IDfCollection folderList = myFolder.getContents(null);

// Empty the file IDs member variable.
         m_fileIDs.clear();

// Empty the file list display.
```

```
        list_id.removeAll();

// Cycle through the collection getting the object ID and adding it to the
// m_listIDs Vector. Get the object name and add it to the file list control.
        while (folderList.next())
        {
            IDfTypedObject doc = folderList.getTypedObject();
            docId = doc.getString("r_object_id");
            docName = doc.getString("object_name");
            list_id.add(docName);
            m_fileIDs.addElement(docId);
        }

// Display a success message for 1 or more objects, or a different message if
// the command succeeds but there are no files in the directory. Enable the
// document manipulation buttons if there are items available in the list box.
        if (list_id.getItemCount()>0) {
            label_results.setText("Directory query complete.");
            button_checkOut.setEnabled(true);
            button_checkIn.setEnabled(true);
            button_cancelCheckout.setEnabled(true);
            button_export.setEnabled(true);
            button_copy.setEnabled(true);
            button_move.setEnabled(true);
            button_delete.setEnabled(true);
        }
        else {
            label_results.setText("Directory query complete. No items found.");
        }
    }

// Handle any exceptions.
    catch (Exception ex) {
        label_results.setText("Exception has been thrown: " + ex);
        ex.printStackTrace();
    }

// Always, always, release the session in the "finally" clause.
    finally {
        mySessMgr.releaseSession(mySession);
    }
 }


/*
 * Handler for the Checkout button. To use the button, enter valid values for
 * respository, userName, password, and directory path in the arguments
 * field and click the Directory button.
 * Choose one of the document names displayed in the list, and click the
 * Checkout button.
 */

 private void button_checkOut_actionPerformed(ActionEvent e) {

 // Get the arguments and assign variable values.
    String temp = textField_arguments.getText();
    StringTokenizer st = new StringTokenizer(temp, ",");
```

```
    String repository = st.nextToken().trim();
    String userName = st.nextToken().trim();
    String password = st.nextToken().trim();


    label_results.setText("Attempting to check out....");

// Create an empty session and session manager.
    IDfSession mySession = null;
    TutorialSessionManager mySessMgr = null;

    try {

// Populate the session manager with user and repository info.
        mySessMgr = new TutorialSessionManager(repository, userName, password);

// Get a session based on the valid credentials.
        mySession = mySessMgr.getSession();

// Get the internal document ID from the m_fileIDs member variable, based on
// the selected item's position in the list control.
        String docId =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

// Create a new instance of the TutorialCheckout object.
        TutorialCheckout tco = new TutorialCheckout();

// Populate the TutorialCheckout object with session info and the internal
// ID of the document to be checked out, perform the checkout operation,
// and display the results.
        label_results.setText(
            tco.checkoutExample(
                 mySession,
                 docId
             )
        );
     }
     catch (Exception ex) {
         System.out.println("Exception hs been thrown: " + ex);
         ex.printStackTrace();
    }
    finally {

// Always, always release the session in the "finally" clause.
         mySessMgr.releaseSession(mySession);
        }
    }

/*
 * Handler for the cancel checkout action. To use this command, enter the
 * repository name, user name, password, and directory path in the argurments
 * field, then click the Directory button. From the file list, choose an object
 * that you have checked out, then click the Cancel Checkout button.
 */
    private void button_cancelCheckout_actionPerformed(ActionEvent e) {

// Capture the arguments as strings.
```

```
        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
        String password = st.nextToken().trim();

        label_results.setText("Attempting to cancel checkout....");

// Create an empty seesion and session manager.
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;

        try {
// Populate the session manager with user and repository info.
            mySessMgr =
                 new TutorialSessionManager(repository, userName, password);

// Get a session based on the valid credentials.
            mySession = mySessMgr.getSession();

// Get the document ID of the selected item.
            String docId =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

// Instantiate the TutorialCancelCheckout class.
            TutorialCancelCheckout tcco = new TutorialCancelCheckout();

// Run the checkout example and display the result.
            label_results.setText(
                tcco.cancelCheckoutExample(mySession,docId)
            );
        }
        catch (Exception ex) {
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
// Always, always release the session in the finally clause.
            mySessMgr.releaseSession(mySession);
        }
    }

 /*
  * Handler for the Check In button. To use the button, enter valid values for
  * respository, userName, password, and directory path in the arguments
  * field and click the Directory button.
  * Choose one of the document names displayed in the list that you have
  * checked out, and click the Check In button.
  */
    private void button_checkIn_actionPerformed(ActionEvent e) {

// Get the arguments and set the as string variables.

        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
```

```
        String password = st.nextToken().trim();

        label_results.setText("Attempting to check in....");

// Create empty session and session manager objects. Create a new
// TutorialCheckinObject
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;
        TutorialCheckIn tci = new TutorialCheckIn();
        try {

// Populate the session manager object with valid credentials.
        mySessMgr =
            new TutorialSessionManager(repository, userName, password);

// Get a session based on the valid credentials.
        mySession = mySessMgr.getSession();

// Get the internal ID of the selected document.
        String docId =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

// Run the check in example and display the results.
        label_results.setText(tci.checkinExample(mySession,docId));
        }
        catch (Exception ex) {
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
// Always, always release the session in the finally clause.
            mySessMgr.releaseSession(mySession);
        }

// Refresh the directory listing.
        button_directory_actionPerformed(e);
    }
/*
 * Handler for the Import button. To use the button, enter valid values for
 * respository, userName, password, destination directory path, and path to the
 * file you want to import in the arguments field, then click the Import
 * button. This method refreshes the directory list after the item has been
 * imported by calling the button handler for the Directory button.
 */
   private void button_import_actionPerformed(ActionEvent e) {

// Parse arguments and assign to variables.
        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
        String password = st.nextToken().trim();
        String directory = st.nextToken().trim();
        String inputFilePath = st.nextToken().trim();

        label_results.setText("Attempting to import....");
```

```
// Instantiate an empty session and session manager.
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;

// Instantiate the custom TutorialImport class.
        TutorialImport ti = new TutorialImport();

        try {

// Populate the session manager with credentials and get a session.
            mySessMgr =
                new TutorialSessionManager(repository, userName, password);
            mySession = mySessMgr.getSession();

// Run the import example and display the result.
            label_results.setText(
                ti.importExample(mySession,directory, inputFilePath)
            );
        }
        catch (Exception ex) {
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }

// Refresh the directory listing.
        button_directory_actionPerformed(e);
    }

    /*
     * Handler for the Export button. To use the button, enter valid values for
     * respository, userName, password, directory path, and the target local
     * directory to which you want to export in the arguments field, then click
     * the Export button.
     */

    private void button_export_actionPerformed(ActionEvent e) {

// Capture the arguments as variables.
        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
        String password = st.nextToken().trim();
        String directory = st.nextToken().trim(); // For the Directory button.
        String targetLocalDirectory = st.nextToken().trim();

        label_results.setText("Attempting to export....");

// Instantiate an empty session and session manager.
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;

        try {
```

```
// Populate the session manager with credentials and get a session.
            mySessMgr =
                new TutorialSessionManager(repository, userName, password);
            mySession = mySessMgr.getSession();

// Get the ID of the selected document.
            String docId =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

//Instantiate the custom TutorialExport class.
            TutorialExport te = new TutorialExport();

// Run the export example and display the results.
            label_results.setText(
                te.exportExample(
                    mySession,
                    docId,
                    targetLocalDirectory
                )
            );

        }
        catch (Exception ex) {
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }   }

/*
 * Handler for the Copy button. To use this button, enter valid values for the
 * repository, userName, password, sourceDirectory, and destinationDirectory,
 * then click the Directory button. Choose a document from the list, then click
 * the Copy button.
 */
    private void button_copy_actionPerformed(ActionEvent e) {

// Capture the arguments as variables.
        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
        String password = st.nextToken().trim();
        String directory = st.nextToken().trim();
        String destinationDirectory = st.nextToken().trim();

        label_results.setText("Attempting to copy....");

// Create an empty session and session manager.
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;

        try {

//Populate the session manager with valid credentials.
            mySessMgr =
```

```
                new TutorialSessionManager(repository, userName, password);

// Instantiate a session using the valid credentials.
            mySession = mySessMgr.getSession();

// Get the docId from the selected item.
            String docId =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

// Instantiate the custom TutorialCopy class.
            TutorialCopy tc = new TutorialCopy();

// Run the copy example method and display the results.
            label_results.setText(
                tc.copyExample(
                    mySession,
                    docId,
                    destinationDirectory
                    )
                );
        }
        catch (Exception ex){
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }
    }

    /*
     * Handler for the Move button. To use this button, enter valid values for
     * the repository, userName, password, sourceDirectory, and
     * destinationDirectory, then click the Directory button. Choose a document
     * from the list, then click the Move button.
     */

    private void button_move_actionPerformed(ActionEvent e) {

// Capture the arguments and assign to variables.
        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
        String password = st.nextToken().trim();
        String sourceDirectory = st.nextToken().trim();
        String destinationDirectory = st.nextToken().trim();

        label_results.setText("Attempting to move....");

// Instantiate an empty session and session manager.
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;
        try {

// Populate the session manager with vaild credentials.
            mySessMgr =
```

```
                new TutorialSessionManager(repository, userName, password);

// Create a session based on valid credentials.
            mySession = mySessMgr.getSession();

// Get the ID of the selected document.
            String docId =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

// Instantiate the custom TutorialMove class.
            TutorialMove tm = new TutorialMove();

// Run the move example and display the result.
            label_results.setText(
                tm.moveExample(
                    mySession,
                    docId,
                    sourceDirectory,
                    destinationDirectory
                    )
                );
        }
        catch (Exception ex){
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }
// Refresh the directory listing.
        button_directory_actionPerformed(e);
    }

 /*
  * Handler for the Delete button. To use this button, enter valid
  * comma-separated values for the repository, userName, password, directory;
  * as the fifth argument, enter true to delete all versions or false to
  * delete only the current version of the document.
  * Click the Directory button. Choose a document from the list, then
  * click the Delete button.
  */
    private void button_delete_actionPerformed(ActionEvent e) {

 // Capture the arguments and assign to variables.
        String temp = textField_arguments.getText();
        StringTokenizer st = new StringTokenizer(temp, ",");
        String repository = st.nextToken().trim();
        String userName = st.nextToken().trim();
        String password = st.nextToken().trim();
        String directory = st.nextToken().trim();
        String currentVersionOnly = st.nextToken().trim();

        label_results.setText("Attempting to delete...");

// Create a blank session and session manager.
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;
```

```
         try {

// Populate the session manager with valid credentials.
            mySessMgr =
                new TutorialSessionManager(repository, userName, password);

// Get a session based on the valid credentials.
            mySession = mySessMgr.getSession();

// Get the ID of the selected document.
            String docId =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

// Instantiate the custom TutorialDelete class.
            TutorialDelete td = new TutorialDelete();

// Execute the delete example and display the result.
            label_results.setText(
                td.deleteExample(
                    mySession,
                    docId,
                    currentVersionOnly
                    )
                );
        }
        catch (Exception ex) {
            System.out.println("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }

// Refresh the directory listing.
        button_directory_actionPerformed(e);
    }
}
```

Index

E

F

G

H

I

T

U

V

X