

OpenText™ Documentum™ Content
Management

**Foundation Java API Development
Guide**

Use sessions and the business object framework, and work with objects, document operations, and virtual documents.

EDCPKCL250400-DGD-EN-01

OpenText™ Documentum™ Content Management Foundation Java API Development Guide

EDCPKCL250400-DGD-EN-01

Rev.: 2025-Oct-18

This documentation has been created for OpenText™ Documentum™ Content Management CE 25.4.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product,
on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111

Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440

Fax: +1-519-888-0677

Support: <https://support.opentext.com>

For more information, visit <https://www.opentext.com>

© 2025 Open Text

Patents may cover this product, see <https://www.opentext.com/patents>.

Disclaimer

No Warranties and Limitation of Liability

Every effort has been made to ensure the accuracy of the features and techniques presented in this publication. However,
Open Text Corporation and its affiliates accept no responsibility and offer no warranty whether expressed or implied, for the
accuracy of this publication.

Table of Contents

| | | |
|----------|---|-----------|
| 1 | Getting started with OpenText Documentum Content Management (CM) Foundation Java API | 7 |
| 1.1 | Foundation Java API | 7 |
| 1.2 | Foundation Java API location | 9 |
| 1.3 | Foundation Java API programming resources | 9 |
| 1.4 | Using dfc.properties to configure Foundation Java API | 9 |
| 1.5 | Using Foundation Java API logging | 16 |
| 2 | Foundation Java API programming basics | 17 |
| 2.1 | Client/Server model | 17 |
| 2.2 | Packages | 17 |
| 2.3 | Interfaces | 18 |
| 2.4 | Processing a repository object | 19 |
| 2.5 | Failover enabled methods | 21 |
| 3 | Sessions and session managers | 23 |
| 3.1 | Sessions | 23 |
| 3.2 | Session managers | 23 |
| 3.3 | Getting session managers and sessions | 24 |
| 3.4 | Objects disconnected from sessions | 28 |
| 3.5 | Related sessions (subconnections) | 28 |
| 3.6 | Original vs. object sessions | 29 |
| 3.7 | Transactions | 29 |
| 3.8 | Configuring sessions using IDfSessionManagerConfig | 30 |
| 3.9 | Getting sessions using login tickets | 30 |
| 3.10 | User authentication | 32 |
| 3.11 | Maintaining state in a session manager | 35 |
| 4 | Creating a test application | 37 |
| 4.1 | The DfcTutorialFrame class | 37 |
| 4.2 | DfcTutorialApplication class | 42 |
| 4.3 | TutorialGetDirectory class | 43 |
| 4.4 | TutorialSessionManager class | 43 |
| 4.5 | Running the tutorial application | 45 |
| 5 | Working with objects | 47 |
| 5.1 | Understanding repository objects | 47 |
| 5.2 | The DFC Base Tutorial Frame | 47 |
| 5.3 | Creating a cabinet | 48 |
| 5.4 | Creating a folder | 50 |
| 5.5 | Creating a document object | 52 |
| 5.6 | Accessing attributes | 54 |

| | | |
|----------|---|------------|
| 5.7 | Setting attributes | 58 |
| 5.8 | Removing an attribute value | 67 |
| 5.9 | Getting object content | 69 |
| 5.10 | Destroying an object | 71 |
| 5.11 | Real-time content deletion | 72 |
| 6 | Working with document operations | 75 |
| 6.1 | Understanding documents | 75 |
| 6.2 | Understanding operations | 77 |
| 6.3 | Types of operation | 78 |
| 6.4 | Basic steps for manipulating documents | 79 |
| 6.5 | Operations for manipulating documents | 83 |
| 6.6 | Handling document manipulation errors | 109 |
| 6.7 | Operations and transactions | 112 |
| 7 | Using the Business Object Framework (BOF) | 113 |
| 7.1 | Overview of BOF | 113 |
| 7.2 | BOF infrastructure | 113 |
| 7.3 | Service-based Business Objects (SBOs) | 116 |
| 7.4 | Type-based Business Objects (TBOs) | 123 |
| 7.5 | Calling TBOs and SBOs | 138 |
| 7.6 | Sample SBO and TBO implementation | 139 |
| 7.7 | Calling Foundation SOAP API, JAXB API, or other components from a BOF module | 142 |
| 7.8 | Aspects | 142 |
| 8 | Working with virtual documents | 151 |
| 8.1 | Understanding virtual documents | 151 |
| 8.2 | Setting version labels | 152 |
| 8.3 | Getting version labels | 155 |
| 8.4 | Creating a virtual document | 156 |
| 8.5 | Traversing the virtual document structure | 160 |
| 8.6 | Binding to a version label | 162 |
| 8.7 | Clearing a version label binding | 165 |
| 8.8 | Removing a virtual document child | 166 |
| 9 | Support for other OpenText Documentum CM functionality | 169 |
| 9.1 | Security services | 169 |
| 9.2 | XML | 170 |
| 9.3 | Virtual documents | 170 |
| 9.4 | Workflows | 170 |
| 9.5 | Document lifecycles | 171 |
| 9.6 | Validation expressions in Java | 171 |

| | | |
|-----|-----------------------------------|-----|
| 9.7 | Unified Login | 172 |
| 9.8 | Search service | 172 |
| 9.9 | WORM support for filestores | 172 |

Chapter 1

Getting started with OpenText Documentum Content Management (CM) Foundation Java API

1.1 Foundation Java API

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

Foundation Java API is a key part of the OpenText Documentum Content Management (CM) software. While the main user of Foundation Java API is other OpenText Documentum CM software, you can use Foundation Java API in any of the following ways:

- Access OpenText Documentum CM functionality from within one of your company's enterprise applications.
For example, your corporate purchasing application can retrieve a contract from your OpenText Documentum CM system.
- Customize or extend products such as Documentum Webtop.
For example, you can modify Documentum Webtop functionality to implement one of your company's business rules.
- Write a method or procedure for OpenText™ Documentum™ Content Management 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 OpenText Documentum CM 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. |
| OpenText Documentum Content Management (CM) Server | Software that manages, protects, and imposes an object oriented structure on the information in repositories. It provides tools for managing the lifecycles of that information and automating processes for manipulating it. |

| | |
|-----------------|--|
| Client programs | Software that provides interfaces between Documentum CM Server and end users. The most common clients run on application servers (for example, Documentum Webtop). |
| End Users | People who control, contribute, or use your organization's information. They use a browser to access client programs running on application servers. |

In this view of OpenText Documentum CM functionality, Foundation Java API lies between Documentum CM Server and clients. OpenText™ Documentum™ Content Management Foundation SOAP API are the primary client interface to OpenText Documentum CM. Foundation Java API is used for server-side business logic and customization.

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

When application developers use Foundation Java API, it is usually within the customization model of OpenText Documentum CM clients, though you can also use Foundation Java API to develop the methods associated with Documentum CM Server functionality, such as document lifecycles.

In the Java application server environment, OpenText Documentum CM client software rests on the foundation provided by the Web Development Kit (WDK). This client 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 OpenText Documentum CM system is to use the Business Object Framework (BOF).

BOF enables you to implement business rules and patterns as reusable elements, called modules. The most important modules for application developers are type based objects (TBOs), service based objects (SBOs), and Aspects. 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.

BOF makes it possible to extend some of Foundation Java API's implementation classes. As a result, you can introduce new functionality in such a way that existing programs begin immediately to reflect changes you make to the underlying business logic.

For information about the conceptual explanation of the capabilities of Documentum CM Server, see *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)*. Foundation Java API provides a framework for accessing those capabilities. Using Foundation Java API and BOF makes your code much more likely to survive future architectural changes to the OpenText Documentum CM system.

1.2 Foundation Java API location

Foundation Java API runs on a Java virtual machine (JVM), which can be on:

- The machine that runs Documentum CM Server.

For example, to be called from a method as part of a workflow or document lifecycle, or to be executed on the Java Method Server on the Documentum CM Server host.

- A middle-tier system.

For example, on an application server to support WDK or to execute server methods.

For client machines, OpenText Documentum CM provides OpenText Documentum Content Management (CM) Foundation SOAP API as the primary support for applications communicating with OpenText Documentum CM.



Note: For information about the locations of files that Foundation Java API installs, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*. The config directory contains several files that are important to Foundation Java API's operation.

1.3 Foundation Java API programming resources

This section provides an overview of the resources available to application developers to help them use Foundation Java API.

1.3.1 Foundation Java API online reference documentation

The public API for Foundation Java API is documented in the Javadocs that ship with the product. You have the option of deploying the Javadocs during Foundation Java API installation, or you can download the Javadocs from OpenText My Support. Direct access to undocumented classes or interfaces is not supported.

1.4 Using dfc.properties to configure Foundation Java API

You can adjust some Foundation Java API 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 Foundation Java API 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. The `dfcfull.properties` file provides the complete list.

The Foundation Java API installer creates a simple `dfc.properties` file and places it in the classpath. Other installers for products that bundle Foundation Java API also create `dfc.properties` file in the appropriate classpath. This file is located in `APP-INF/classes` if you are deploying the EAR file, or in `WEB-INF/classes` if you deploying the WAR file.

If you prefer, you can use a `#include` statement to point to a properties file outside of the web application on the local file system. This can make access to some settings more convenient and allows you to modularize your configuration settings:

```
#include C:\Documentum\config\dfc.properties
```

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
```

The following sections provide some details on settings that your application may require:

1.4.1 BOF and global registry settings

All OpenText Documentum CM installations must designate a global registry to centralize information and functionality. [“Global registry” on page 116](#) provides information about global registries.

1.4.1.1 Connecting to the global registry

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

- `dfc.globalregistry.repository`

The name of the repository. The repository must project to a connection broker that Foundation Java API has access to.

- `dfc.globalregistry.username`

The user name part of the credentials that Foundation Java API uses to access the global registry. [“Global registry user” on page 116](#) provides information about how to create this user.

- `dfc.globalregistry.password`

The password part of the credentials that Foundation Java API uses to access the global registry. The Foundation Java API 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
```

You can call the `getGlobalRegistryPassword` method to get the global registry password. For example, the following code snippet prints this password:

```
System.out.println(DfPreferences.getInstance().getGlobalRegistryPassword());
```



Note: This method returns the global registry password as it is presented in the `dfc.properties` file. If the password is encrypted in the file, it is encrypted in the return.

The `dfc.properties` file also provides an optional property to resist attempts to obtain unauthorized access to the global registry. For example, the following entry sets the minimum interval between connection attempts to the default value of 60 seconds:

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

1.4.1.2 Performance tradeoffs

You can adjust the amount of time Foundation Java API relies on cached information before checking for consistency between the local cache and the global registry. For example, the following entry sets that interval to the default value of 60 seconds:

```
dfc.bof.cache.currency_check_interval=60
```

Because global registry information tends to be relatively static, you might be able to check less 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. For more information about the vstamp objects, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*. The name of this entry was formerly `dfc.bof.cacheconsistency.interval`.

1.4.2 Diagnostic settings

Foundation Java API provides a number of properties that facilitate diagnosing and solving problems.

1.4.2.1 Diagnostic mode

Foundation Java API can run in diagnostic mode. You can cause this to happen by including the following setting in `dfc.properties`:

```
dfc.diagnostics.resources.enable=true
```

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

Foundation Java API 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.

1.4.2.2 Configuring connection brokers

You must set the repeating property `dfc.docbroker.host`, one entry per connection broker. 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.

1.4.2.3 `dfc.data.dir`

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

1.4.2.4 Tracing options

Foundation Java API 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 property traces Foundation Java API entry calls, return values and parameters.

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

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

These properties include more details and RPCs sent to the server. Start with the simple trace because the verbose trace produces much more output.

1.4.3 XML processing options

Foundation Java API's XML processing is largely controlled by configuration files that define XML applications. Two properties provide additional options.



Note: You must use XDQL through Foundation Java API.

1.4.4 Search options

Foundation Java API supports the search capabilities of Documentum Federated Search Services with a set of properties. The Documentum Federated Search Services installer sets some of these. Most of the others specify diagnostic options or performance tradeoffs.

1.4.5 Storage policy options

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

1.4.6 Performance tradeoffs

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

Foundation Java API 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 Foundation Java API rereads that file. The `dfc.config.check_interval` property specifies the interval in which Foundation Java API rereads the Foundation Java API configuration file changes in seconds. The default value for this property is 30.

Foundation Java API 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](#)” on page 10 and “[Search options](#)” on page 13 sections also provide performance tradeoffs.

1.4.7 Registry emulation

Foundation Java API 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 Foundation Java API installer sets this property to `file` by default.

1.4.7.1 Microsoft Object Linking and Embedding (OLE)

When importing or exporting Microsoft OLE documents, Foundation Java API can automatically search for linked items and import them as individual system objects linked as nodes in a virtual document. On export, the document can be reassembled into a single document with OLE links. In order to enable this behavior, you must configure the registry emulation mode in `dfc.properties`.

By default, the setting is:

```
dfc.registry.mode = file
```

To enable OLE file handling, set the value to:

```
dfc.registry.mode = windows
```

1.4.8 Client file system locations

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

1.4.9 New dfc.properties key to turn off trusted login

By default, applications running on the Documentum CM Server host are allowed to make repository connections as the installation owner without presenting a password. This is called a trusted login.

If an application, such as Documentum Administrator, that has an explicit login dialog box is installed on a Documentum CM Server host, a user can log in as the installation owner without a password using a trusted login.

OpenText Documentum CM 6.7 and later uses a new `dfc.properties` key to turn off trusted logins if you do not want to allow trusted logins through such applications. The key is:

```
dfc.session.allow_trusted_login
```

Setting this key to false requires users always to provide a password, even when logging in as the installation owner.

1.4.10 Integrating Documentum Secret Integration Service with Foundation Java API

Foundation Java API is integrated with Documentum Secret Integration Service (DSIS). The DSIS daemon server must always be up and running during the Foundation Java API initialization.

The secret names for all the passwords related to Foundation Java API must be in line with the `dfc.properties` entries. For example, for `dfc.globalregistry.password`, the secret name is `DFC_GLOBALREGISTRY_PASSWORD` and the key name is the value of `dfc.globalregistry.repository`. Similarly, for `dfc.security.ssl.truststore_password`, the secret name is `DFC_SECURITY_SSL_TRUSTSTORE_PASSWORD` and the key name is the value of `dfc.docbroker.host[0]`.

Foundation Java API automatically processes the requests for the daemon server for global registry and trust store passwords and then retrieves secrets when instantiating the `DfClient` object. You must follow the `DFC_GLOBALREGISTRY_PASSWORD` (all uppercase with an underscore) format to add secrets and key information in Vault.

All OpenText Documentum CM client applications that rely on Foundation Java API can access the global registry (repository) and trust store (for SSL) passwords. Most

of the clients use the `dfc.globalregistry.password` and `dfc.security.ssl.truststore_password` passwords. The Foundation Java API automatically retrieves all these passwords from the DSIS daemon server, encrypt them, and stores in the Foundation Java API memory. For all the subsequent access, the passwords are retrieved from the Foundation Java API memory. If all the passwords are retrieved successfully and the application is accessible, then OpenText recommends you to shut down the DSIS deamon server.

All OpenText Documentum CM client applications must update the following entries in the `dfc.properties` file:

```
dfc.dsis.enabled=true
dfc.dsis.daemon.url=http://localhost:8200/dsis
dfc.dsis.daemon.token=<token generated for the DSIS daemon agent>
```

The default value of `dfc.dsis.daemon.url` is `http://localhost:8200/dsis`. OpenText recommends you to provide an appropriate port value if you want to use any port other than 8200.

The preceding list of entries except the `dfc.dsis.daemon.token` can be updated as environment variables as follows:

- Property `dfc.dsis.enabled` as environment variable `DFC_DSIS_ENABLED`
- Property `dfc.dsis.daemon.url` as environment variable `DFC_DSIS_DAEMON_URL`

! Important

The value updated as environment variables takes precedence over the entries in the `dfc.properties` file.

In cloud platforms, the preceding list of properties are defined as environmental variables. Token value is not mandatory in cloud platforms because, by default, the `dsis.dctm.tokenNeeded` property is set to `false` in the DSIS daemon server. However, it is mandatory for the on-premises environments.



Note: You can define the secret and key name in the `dfc.properties` file for both `dfc.globalregistry.password` and `dfc.security.ssl.truststore_password` in the `<secret_name>/<key_name>` format. If secret and key names are defined in the `dfc.properties` file, then Foundation Java API uses secret and key name to retrieve the passwords from Vault. If passwords mapping is not found in Vault, then Foundation Java API formulates the secret name and key name using uppercase and underscores. For example:

```
DFC_GLOBALREGISTRY_PASSWORD/<value of dfc.globalregistry.repository>
```

If password is not found with default mappings, then Foundation Java API returns the null value.

If all OpenText Documentum CM client applications that have application-specific passwords stored in Vault, then the passwords can be retrieved using public Foundation Java APIs.

You can find the public Foundation Java APIs for DSIS in Javadocs.

1.5 Using Foundation Java API logging

Foundation Java API provides diagnostic logging. The logging is controlled by a log4j configuration file, which is customarily named `log4j2.properties`. The dmcl installer as well as other 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 referring to the log4j documentation.

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

Chapter 2

Foundation Java API programming basics

This chapter describes some common patterns of Foundation Java API application code.

2.1 Client/Server model

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

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

An IDfSession object represents a connection with the Documentum CM Server and provides services related to that session. Foundation Java API programmers create new OpenText Documentum CM objects or obtain references to existing OpenText Documentum CM objects through the methods of IDfSession.

To get a session, first create an IDfSessionManager by calling IDfClient.newSessionManager(). Next, get the session from the session manager. For more information about sessions and session manager, see chapter “[Sessions and session managers](#)” on page 23.

2.2 Packages

Foundation Java API comprises a number of packages, that is, sets of related classes and interfaces.

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

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



Notes

- 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 Documentum CM Server XML functionality.

- The IDfClientX is the correct interface for accessing factory methods (all of its getXXX methods, except for those dealing with the Foundation Java API version or trace levels).

The Foundation Java API interfaces form a hierarchy; some derive methods and constants from others. Use the Tree link from the home page of the Foundation Java API online reference (see [“Foundation Java API online reference documentation” on page 9](#)) 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.

2.3 Interfaces

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



Tip: Foundation Java API provides factory methods to instantiate objects that implement specified Foundation Java API 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 Foundation Java API programming, the only implementation classes you should instantiate directly are DfClientX and the exception classes (DfException and its subclasses).

Foundation Java API 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). [“Using the Business Object Framework \(BOF\)” on page 113](#) provides information about BOF.



Caution

Foundation Java API 7.0 and later no longer support direct calls from the COM Interface.

2.3.1 IDfPersistentObject

An IDfPersistentObject corresponds to a persistent object in a repository. With Foundation Java API you usually do not 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 can 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.

2.4 Processing a repository object

The following general procedure may help to clarify the Foundation Java API approach. This example is not meant to give complete details, but to provide an overview of a typical transaction. More detail on the process is provided in subsequent chapters.

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();
```

“Sharable and Private Sessions” on page 23 provides information about the difference between the getSession and newSession methods of IDfSessionManager.

5. If you do not have a reference to the OpenText Documentum CM 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 on. “[Working with document operations](#)” on page 75 provides the examples that use operations.
7. Release the session.

The following example does not use the operations package:

 **Example 2-1: Processing a repository object**

The following fragment from a Java program that uses Foundation Java API 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.
- 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 Foundation Java API 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 Foundation Java API methods report errors by throwing a DfException object. Java code mentioned in the preceding example normally appears within a try/catch/finally block, with an error handler in the catch block.



Tip: When writing code that calls Foundation Java API, it is a best practice to include a finally block to ensure that you release storage and sessions.

2.5 Failover enabled methods

Documentum CM Server installations can be set up and configured for high availability. One part of high availability is that the Java Method Server (JMS) is set up to failover to another JMS. If you are writing custom methods, and you wish them to be failover enabled, here are some minimum requirements:

- Logic that modifies repository objects should be done in a transaction.
 - If failover occurs, failed or incomplete methods will be rerun, from the beginning, on the backup JVM.
 - Use a transaction to enable roll back of any object changes in the event of a failure.
- Do not write code that assumes file, directory paths, or other resources exist if they only exist on one host.
- Test failover of custom method code by configuring multiple Documentum CM Server/JMS and shutting down one while the method is running.

Chapter 3

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, then for more information on session management techniques and methods specific to WDK, see *OpenText Documentum Web Development Kit - Development Guide (EDCPKCLWT160709-PGD)*.

3.1 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 establishing the connection. 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 Javadocs for a complete reference).

3.1.1 Sharable and Private Sessions

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

3.2 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, the application should release the session using the `IDfSessionManager.release()` method in a `finally` clause. The session initially remains available to be reclaimed by the session manager instance that released it, and simultaneously, the connection associated with that session will be placed to the 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 Documentum CM Server connections, which are an extremely

expensive resource, in a web programming environment where a large number of sessions are required.

3.3 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:

3.3.1 Instantiating a session manager

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

Example 3-1: 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.

3.3.2 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 is not 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, such as 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.

3.3.3 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 actively being used 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 preceding method. In those cases, you should call `IDfSession.disconnect()`.

3.3.3.1 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("myRepository");
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.

3.3.3.2 Sessions can be released only once

After 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("myRepository");
sessionManager.release(session);
sessionManager.release(session); // throws runtime exception
```

3.3.3.3 Sessions cannot be used after it is released

After 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("myRepository");
sessionManager.release(session);
session.getObject(objectId); // throws runtime exception
```

3.4 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. Foundation Java API 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, Foundation Java API will automatically reopen a session for the object. You can get a reference to this session by calling object.getSession.

Users of versions of Foundation Java API earlier than release 6, 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.

3.5 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 new session is a peer of the original session, and your code is responsible for releasing 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 Foundation Java API should note that using setDocbaseScope for creating subconnections is no longer recommended. Use one of the preceding techniques instead.

3.6 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 Foundation Java API 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().

3.7 Transactions

Foundation Java API 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 Foundation Java API 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.beginTransaction()) and occur before it completes (see IDfSession.commitTrans() and IDfSession.abortTrans()). Previous versions of Foundation Java API did not support calling beginTrans() on a session obtained from a session manager. This restriction has been removed. The following code 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().beginTransaction();
            txStartedHere = true;
        }
        try
        {
            doSomething();           // Do something that requires transactions
            if ( txStartedHere )
                getObjectSession().commitTrans();
        }
        finally
        {
            if ( txStartedHere && getObjectSession().isTransactionActive() )
```

```
        getObjectSession().abortTrans();
    }
}
```

3.8 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("myRepository");
```

3.9 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 OpenText Documentum CM 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 more information about getting sessions using login tickets, see *OpenText Documentum Web Development Kit - Development Guide (EDCPKCLWT160709-PGD)*.

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 Documentum CM Server. For more information, see *OpenText Documentum Web Development Kit - Development Guide*

(*EDCPKCLWT160709-PGD*). “Principal authentication support” on page 33 provides information on building support for principal authentication in custom (non-WDK) web applications.

For more information about the login ticket behavior and server configuration dependencies, see *OpenText Documentum Content Management - Server Fundamentals Guide* (*EDCCS250400-GGD*). For more information about the server configuration settings, see *OpenText Documentum Content Management - Server Administration and Configuration Guide* (*EDCCS250400-AGD*).

3.9.1 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.

3.9.2 Generating a login ticket using a superuser account

The `IDfSession.getLoginTicketEx` method allows 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.

“Getting session managers and sessions” on page 24 provides information on instantiating session managers and setting identities.

```
/*
 * 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 and return it

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

3.10 User authentication

You can use either OTDS SSO or Java Enterprise Edition (JEE) principal authentication for user authentication.

3.10.1 OTDS SSO

To use OTDS, you must configure OTDS on Documentum CM Server. For more information, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)* and OTDS documentation.

3.10.2 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, after 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 *OpenText Documentum Web Development Kit - Development Guide (EDCPKCLWT160709-PGD)*.

3.10.2.1 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 Foundation Java API 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 session = sessMgr.getSession(repository);  
  
    try  
    {  
        System.out.println("Got session: " + session.getSessionId());  
        System.out.println("Username: " + session.getLoginInfo().getUser());  
    }  
    // Release the session in a finally clause.  
    finally  
    {  
        sessMgr.release(session);  
    }  
}
```

3.10.2.2 Default classes for demonstrating principal support implementation

Foundation Java API 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.

3.11 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 4

Creating a test application

The primary use of Foundation Java API is to add business logic to your applications. The presentation layer is built using the Web Development Kit, most often via customization of the Documentum Webtop interface. Building a custom UI on top of Foundation Java API 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.

To the extent possible, the sample classes have been created as self-contained entities. However, a best practice is to instantiate a Session Manager when you first open an application and continue to use it until the application stops. Therefore, the logic for creating a Session Manager has been incorporated as a method of the DfcTutorialFrame class.

4.1 The DfcTutorialFrame class

DfcTutorialFrame displays the controls used to set arguments, execute commands, and view results. In this example, it displays buttons used to navigate a directory in the repository.

Example 4-1: DfcBaseTutorialFrame.java

```
package com.emc.tutorial;

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

import com.documentum.fc.common.IDfId;

import java.awt.Dimension;
import java.awt.List;
import java.awt.Rectangle;

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

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

```

import javax.swing.ActionMap;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

public class DfcTutorialFrame extends JFrame {
    private JTextField jTextField_repositoryName = new JTextField();
    private JLabel jLabel_repositoryName = new JLabel();
    private JLabel jLabel_userName = new JLabel();
    private JTextField jTextField_userName = new JTextField();
    private JTextField jTextField_password = new JTextField();
    private JLabel jLabel_password = new JLabel();
    private JTextField jTextField_cwd = new JTextField();
    private JLabel jLabel_cwd = new JLabel();
    private JButton jButton_getDirectory = new JButton();
    private Vector m_fileIDs = new Vector();
    private List list_id = new List();
    private JLabel jLabel_messages = new JLabel();
    private JScrollPane jScrollPane_results = new JScrollPane();
    private JTextArea jTextArea_results = new JTextArea();
    private JLabel jLabel_results = new JLabel();
    private JLabel jLabel_directoryContents = new JLabel();
    private Vector m_breadcrumbs = new Vector();
    private JButton jButton_upDirectory = new JButton();

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

    private void jbInit() throws Exception {
        this.getContentPane().setLayout( null );
        this.setSize(new Dimension(679, 479));
        this.setTitle( "DFC Tutorial Frame" );
        jTextField_repositoryName.setBounds(new Rectangle(15, 20, 100, 25));
        jTextField_repositoryName.setText("MyRepository");
        jLabel_repositoryName.setText("Repository Name");
        jLabel_repositoryName.setBounds(new Rectangle(15, 5, 100, 15));
        jLabel_repositoryName.setHorizontalAlignment(SwingConstants.CENTER);
        jLabel_userName.setText("User Name");
        jLabel_userName.setBounds(new Rectangle(135, 5, 95, 15));
        jLabel_userName.setHorizontalAlignment(SwingConstants.CENTER);
        jTextField_userName.setBounds(new Rectangle(135, 20, 95, 25));
        jTextField_userName.setText("MyLogin");
        jTextField_password.setBounds(new Rectangle(245, 20, 100, 25));
        jTextField_password.setText("MyPassword");
        jLabel_password.setText("Password");
        jLabel_password.setBounds(new Rectangle(245, 5, 100, 15));
        jLabel_password.setHorizontalAlignment(SwingConstants.CENTER);
        jTextField_cwd.setBounds(new Rectangle(15, 70, 330, 25));
        jTextField_cwd.setText("/MyLogin");
        jTextField_cwd.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                jTextField_cwd_actionPerformed(e);
            }
        });
        jLabel_cwd.setText("Current Working Directory");
        jLabel_cwd.setBounds(new Rectangle(15, 50, 130, 20));
        jLabel_cwd.setHorizontalAlignment(SwingConstants.CENTER);
        jButton_getDirectory.setText("Get Directory");
        jButton_getDirectory.setBounds(new Rectangle(15, 230, 100, 20));
    }
}

```

```

jButton_getDirectory.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        jButton_getDirectory_actionPerformed(e);
    }
});
jLabel_messages.setText("Messages appear here.");
jLabel_messages.setBounds(new Rectangle(5, 430, 495, 15));
jScrollPane_results.setBounds(new Rectangle(15, 275, 330, 140));
jLabel_results.setText("Results:");
jLabel_results.setBounds(new Rectangle(15, 260, 135, 15));
jLabel_directoryContents.setText("Directory contents:");
jLabel_directoryContents.setBounds(new Rectangle(15, 100, 140, 15));
jButton_upDirectory.setText("Up One Level");
jButton_upDirectory.setBounds(new Rectangle(245, 230, 100, 20));
jButton_upDirectory.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        jButton_upDirectory_actionPerformed(e);
    }
});
list_id.setBounds(new Rectangle(15, 115, 330, 110));
list_id.setSize(new Dimension(330, 110));
list_id.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        list_id_actionPerformed(e);
    }
});
jScrollPane_results.setViewportView(jTextArea_results, null);
this.getContentPane().add(jButton_upDirectory, null);
this.getContentPane().add(jLabel_directoryContents, null);
this.getContentPane().add(jLabel_results, null);
this.getContentPane().add(jScrollPane_results, null);
this.getContentPane().add(list_id, null);
this.getContentPane().add(jLabel_messages, null);
this.getContentPane().add(jButton_getDirectory, null);
this.getContentPane().add(jLabel_cwd, null);
this.getContentPane().add(jTextField_cwd, null);
this.getContentPane().add(jLabel_password, null);
this.getContentPane().add(jTextField_password, null);
this.getContentPane().add(jTextField_userName, null);
this.getContentPane().add(jLabel_userName, null);
this.getContentPane().add(jLabel_repositoryName, null);
this.getContentPane().add(jTextField_repositoryName, null);
}

private void getDirectory() {
    // Get the arguments and assign variable values.

    String repository = jTextField_repositoryName.getText();
    String userName = jTextField_userName.getText();
    String password = jTextField_password.getText();
    String directory = jTextField_cwd.getText();
    TutorialGetDirectory tgd = new TutorialGetDirectory();

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

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

    IDfCollection folderList = tgd.getDirectory(
        repository,
        userName,
        password,
        directory
    );
    try {
        // Cycle through the collection getting the object ID and adding it to the
        // m_fileIDs Vector. Get the object name and add it to the file list control.
        while (folderList.next())
    }
}

```

```

        IDfTypedObject doc = folderList.getTypedObject();
        list_id.add(doc.getString("object_name"));
        m_fileIDs.addElement(doc.getString("r_object_id"));
    }

    // 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) {
        jLabel_messages.setText("Directory query complete.");
    }
    else {
        jLabel_messages.setText("Directory query complete. No items found.");
    }
}

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

private void jButton_getDirectoryActionPerformed(ActionEvent e) {
    String repository = jTextField_repositoryName.getText();
    String userName = jTextField_userName.getText();
    String password = jTextField_password.getText();
    String directory = jTextField_cwd.getText();

    IDfSession mySession = null;
    TutorialSessionManager mySessMgr = null;
    try {
        mySessMgr = new TutorialSessionManager
            (repository, userName, password);
        mySession = mySessMgr.getSession();

        if(list_id.getSelectedItem()!=null) {

            String objectIdString =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
            IDfId idObj = mySession.getIdByQualification(
                "dm_sysobject where r_object_id='" + objectIdString + "'");
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            if(sysObj.getTypeName().equals("dm_folder")) {
                jTextField_cwd.setText(directory + "/" + list_id.getSelectedItem());
                getDirectory();
                StringTokenizer st = new
StringTokenizer(jTextField_cwd.getText(),"/");
                m_breadcrumbs.removeAllElements();
                while(st.hasMoreTokens()) m_breadcrumbs.add(st.nextToken());
                StringBuffer newDirectory = new StringBuffer("");
                for(int i=0; i<m_breadcrumbs.size(); i++) {newDirectory.append("/")
                    + m_breadcrumbs.elementAt(i));
                }
                jTextField_cwd.setText(newDirectory.toString());
            }
            else {
                getDirectory();
            }
        }
        else {
            getDirectory();
        }
    }
    catch (Exception ex) {
        jLabel_messages.setText("Exception has been thrown: " + ex);
        ex.printStackTrace();
    }
}

```

```

        finally {
            mySessMgr.releaseSession(mySession);
        }
    }

    private void changeDirectory() {
        String repository = jTextField_repositoryName.getText();
        String userName = jTextField_userName.getText();
        String password = jTextField_password.getText();
        String directory = jTextField_cwd.getText();

        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;
        try {
            mySessMgr = new TutorialSessionManager
                (repository, userName, password);
            mySession = mySessMgr.getSession();
            String objectIdString =
                m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

            IDfId idObj = mySession.getIdByQualification(
                "dm_sysobject where r_object_id=" + objectIdString + "");

            // Instantiate an object from the ID.

            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            if(sysObj.getTypeName().equals("dm_folder")) {

                jTextField_cwd.setText(directory + "/" + list_id.getSelectedItem());
                getDirectory();
                StringTokenizer st = new StringTokenizer(jTextField_cwd.getText(),"/");
                m_breadcrumbs.removeAllElements();
                while(st.hasMoreTokens()) m_breadcrumbs.add(st.nextToken());
                StringBuffer newDirectory = new StringBuffer("");
                for (int i=0; i<m_breadcrumbs.size(); i++) {
                    newDirectory.append("/" + m_breadcrumbs.elementAt(i));
                }
                jTextField_cwd.setText(newDirectory.toString());
            }
        }
        catch (Exception ex) {
            jLabel_messages.setText("Exception has been thrown: " + ex);
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }
    }

    private void jButton_changeDirectoryActionPerformed(ActionEvent e) {
        changeDirectory();
    }

    private void jTextField_cwdActionPerformed(ActionEvent e) {
        getDirectory();
    }

    private void list_idActionPerformed(ActionEvent e) {
        changeDirectory();
    }

    private void jButton_upDirectoryActionPerformed(ActionEvent e) {
        StringBuffer newDirectory = new StringBuffer("");
        newDirectory.append("/") + m_breadcrumbs.elementAt(0));
        if (m_breadcrumbs.size(>1){
            m_breadcrumbs.removeElementAt(m_breadcrumbs.size()-1);
            for (int i=1; i<m_breadcrumbs.size(); i++) {
                newDirectory.append("/" + m_breadcrumbs.elementAt(i));
            }
        jTextField_cwd.setText(newDirectory.toString());
        getDirectory();
    }
}

```

```
    }  
}
```



4.2 DfcTutorialApplication class

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

Example 4-2: DfcTutorialApplication.java

```
package com.emc.tutorial;  
  
import java.awt.Dimension;  
import java.awt.Toolkit;  
  
import javax.swing.JFrame;  
import javax.swing.UIManager;  
  
public class DfcTutorialApplication {  
    public DfcTutorialApplication() {  
        JFrame frame = new DfcTutorialFrame();  
        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.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  
        frame.setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        try {  
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        new DfcTutorialApplication();  
    }  
}
```



4.3 TutorialGetDirectory class

This class returns the directory contents.

Example 4-3: TutorialGetDirectory.java

```
package com.emc.tutorial;

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

public class TutorialGetDirectory {
    public TutorialGetDirectory() {
    }
    public IDfCollection getDirectory(
        String repository,
        String userName,
        String password,
        String directory
    ) {
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;
        IDfCollection fileList = null;

        try {
            mySessMgr = new TutorialSessionManager(repository, userName, password);
            mySession = mySessMgr.getSession();
            IDfFolder myFolder = mySession.getFolderByPath(directory);
            fileList = myFolder.getContents(null);
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
        finally {
            mySessMgr.releaseSession(mySession);
        }
        return fileList;
    }
}
```



4.4 TutorialSessionManager class

This class creates the session manager to the repository.

Example 4-4: TutorialSessionManager.java

```
package com.emc.tutorial;

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 encapsulate session and user information.
private IDfSessionManager m_sessionMgr;
private String m_repository;
private String m_userName;
private String m_password;

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

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

        // Call the local 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);
}
}
```



4.5 Running the tutorial application

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

To list the contents of a directory, enter values in the **Repository Name**, **User Name**, **Password**, and **Current Working Directory** fields. Click **Get Directory** to update the file listing. For convenience, you can modify the DfcBaseTutorialFrame class to launch the application with these fields already populated with your own settings.

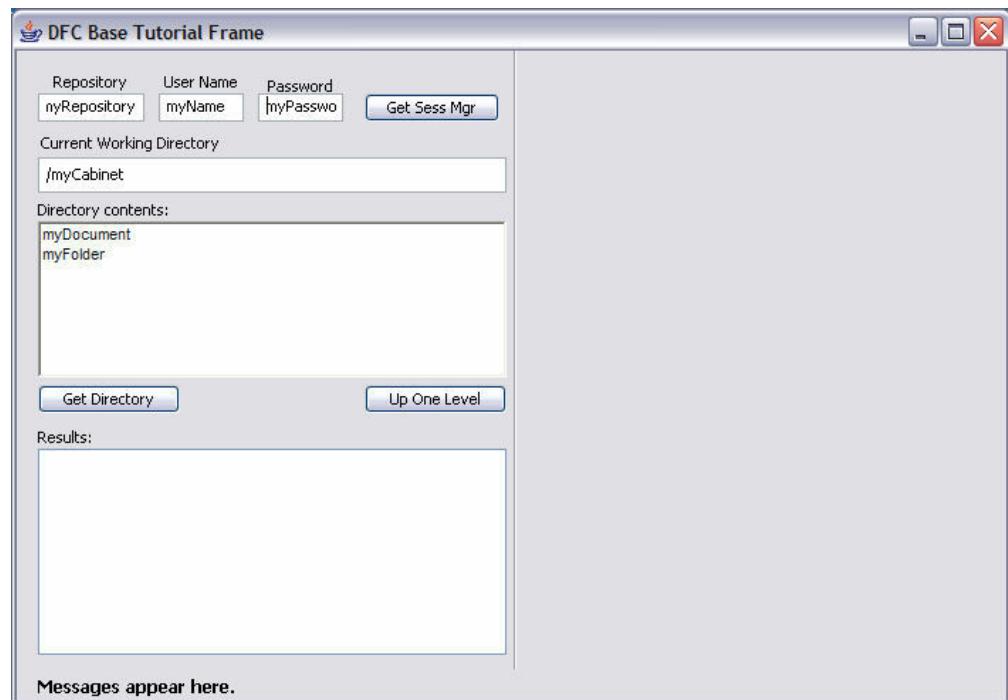


Figure 4-1: DFC Base Tutorial Frame

This example demonstrated how you can create an application that connects to the Documentum CM Server and retrieves information. Many of the remaining examples in this guide provide the code for a button handler that you can add to this test application to call classes that demonstrate implementation of individual behaviors using Foundation Java API.

Chapter 5

Working with objects

This chapter describes how to use Foundation Java API calls to work with objects stored in the OpenText Documentum CM repository.

When you are working with the metadata associated with a repository object, using Foundation Java API methods directly can be more efficient and quicker to deploy. When you are working with document content, a better choice is to work with document operations, covered in [Working with Document Operations](#).

5.1 Understanding repository objects

A repository object has two basic components: metadata and content. Metadata describe the object (format, type, and so on) and store other pertinent information such as the object name, owner, creation date, and modification date. As a developer, you are primarily concerned with manipulating document metadata. Metadata are stored in database tables and associated with content objects stored in a local file system.

Object content is the information of most interest to end users. If the object is a text document, the content is the text. If the object is a graphic, the content is the binary information used to create the picture.

Using Foundation Java API methods, developers can examine and manipulate the metadata associated with repository objects.

5.2 The DFC Base Tutorial Frame

The examples in this chapter start with the **DFC Base Tutorial Frame** and add buttons and fields to demonstrate object manipulation techniques. You can implement any or all of these self-contained examples.

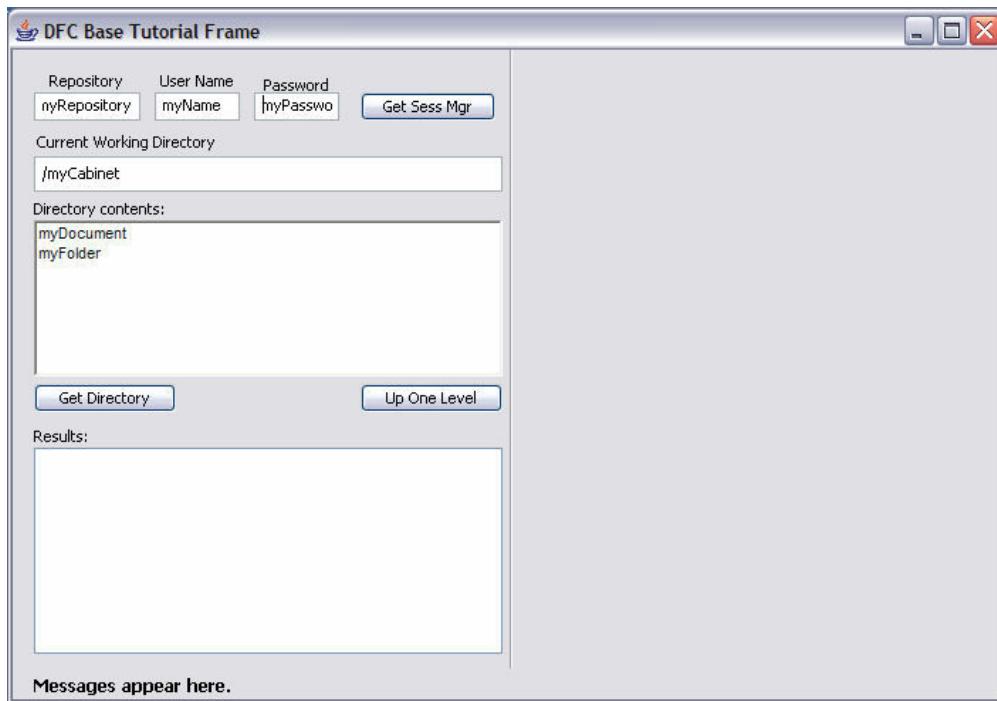


Figure 5-1: DFC Base Tutorial Frame

5.3 Creating a cabinet

Before you begin manipulating repository objects, you might want to create your own cabinet to work in. A cabinet is a top level container object used to hold folders. The cabinet is, in fact, a type of folder, and is created using the interface `IDfFolder`. Setting its object type to "dm_cabinet" gives it additional features, including the ability to exist as a top-level object.

You can create a test button for creating a cabinet in **DFC Base Tutorial Frame**.

To add the Create Cabinet button to the DFC Base Tutorial Frame:

1. Create a JTextField control named `jTextField_cabinetName`.
2. Create a JButton control named `jButton_makeCabinet`.
3. Add a button handler method for **Make Cabinet**.
4. Create the class **TutorialMakeCabinet**.

The source code for the button handler and `TutorialMakeCabinet` class follow.

➤ Example 5-1: Make Cabinet button handler method

```
private void jButton_makeCabinetActionPerformed(ActionEvent e)
{
```

```

String repositoryName = jTextField_repositoryName.getText();
String cabinetName = jTextField_cabinetName.getText();

if (!cabinetName.equals(""))
{
    TutorialMakeCabinet tmc = new TutorialMakeCabinet();
    if (tmc.makeCabinet(
        m_sessionManager,
        repositoryName,
        cabinetName)
    )
    {
        jTextField_cwd.setText("//" + cabinetName);
        getDirectory();
        jLabel_messages.setText(
            "Created cabinet " + cabinetName + "."
        );
    }
    else
    {
        jLabel_messages.setText("Cabinet creation failed.");
    }
}
else
{
    jLabel_messages.setText("Enter a unique cabinet name " +
                           "to create a new cabinet.");
}
}
}

```



Example 5-2: The TutorialMakeCabinet class

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;

public class TutorialMakeCabinet
{
    public TutorialMakeCabinet()
    {
    }

    public Boolean makeCabinet(
        IDfSessionManager sessionManager,
        String repositoryName,
        String cabinetName
    )
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // check to see if the cabinet already exists
            IDfFolder myCabinet = mySession.getFolderByPath("//" + cabinetName);
            if (myCabinet == null)
            {
                IDfSysObject newCabinet =
                    (IDfFolder) mySession.newObject(DM_CABINET);
                newCabinet.setObjectName(cabinetName);
                newCabinet.save();
                return true;
            }
        }
    }
}

```

```
        }
    }
}
catch (Exception ex)
{
    ex.printStackTrace();
    return false;
}
finally
{
    sessionManager.release(mySession);
}
public static final String DM_CABINET = "dm_cabinet";
}
```



5.4 Creating a folder

Creating a folder is similar to creating a cabinet. The essential differences are that you will create a `dm_folder` object and identify the parent cabinet or folder in which you will create it.

To add a Make Folder button to the DFC Base Tutorial Frame:

1. Create a JTextField control named `jTextField_folderName`.
2. Create a JButton control named `jButton_makeFolder`.
3. Add a handler method for the **Make Folder** button.
4. Create the class `TutorialMakeFolder`.

The code for the **Make Folder** button handler and the `TutorialMakeFolder` class follow.

Example 5-3: Make Folder button handler method

```
private void jButton_makeFolderActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String folderName = jTextField_newFolder.getText();
    String parentName = jTextField_cwd.getText();

    if (!folderName.equals("") & !parentName.equals(""))
    {
        TutorialMakeFolder tmf = new TutorialMakeFolder();
        if (
            tmf.makeFolder(
                m_sessionManager,
                repositoryName,
                folderName,
                parentName)
        )
            getDirectory();
            jLabel_messages.setText("Created folder " + folderName +

```

```

        ".");
    }
    else
    {
        jLabel_messages.setText("Folder creation failed.");
    }
}
else
{
    jLabel_messages.setText("Enter a folder name and current working" +
                           " directory to create a new folder.");
}
}

```



➤ Example 5-4: The TutorialMakeFolder class

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;

public class TutorialMakeFolder
{
    public TutorialMakeFolder()
    {
    }

    public Boolean makeFolder(
        IDfSessionManager sessionManager,
        String repositoryName,
        String folderName,
        String parentName
    )
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfSysObject newFolder =
                (IDfFolder) mySession.newObject(DM_FOLDER);
            IDfFolder aFolder =
                mySession.getFolderByPath(parentName + "/" + folderName);
            if (aFolder == null)
            {
                newFolder.setObjectName(folderName);
                newFolder.link(parentName);
                newFolder.save();
                return true;
            }
            else
            {
                return false;
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return false;
        }
        finally
        {
            sessionManager.release(mySession);
        }
    }
}

```

```
    }
    public static final String DM_FOLDER = "dm_folder";
}
```



5.5 Creating a document object

A document object represents both the content of a document and the metadata that describe the document. In most cases, you create a document by importing an existing document from a local source to the repository.

In the example [TutorialMakeDocument](#), we create a document setting just the minimal required information: the name of the document, the document type, the source of the content, and the parent folder in which to create the new document object.

To add a Make Document button to the DFC Base Tutorial Frame:

1. Create a JTextField control named jTextField_documentName.
2. Create a JTextField control named jTextField_srcDocPath.
3. Create a JTextField control named jTextField_docType.
4. Create a JButton control named jButton_makeDocument.
5. Add a [button handler method for Make Document](#).
6. Create the class [TutorialMakeDocument](#).

The following is an example of the Make Document button handler:

Example 5-5: The Make Document button handler

```
private void jButton_makeDocumentActionPerformed(ActionEvent e)
{
    String repository = jTextField_repositoryName.getText();
    String documentName = jTextField_documentName.getText();
    String folderName = jTextField_cwd.getText();
    String srcPath = jTextField_sourceDocumentPath.getText();
    String docType = jTextField_documentType.getText();

    TutorialMakeDocument tmd = new TutorialMakeDocument();
    if (tmd.makeDocument(
        m_sessionManager,
        repository,
        documentName,
        docType,
        srcPath,
        folderName
    ))
    {
        jLabel_messages.setText("Created document " + documentName + ".");
        getDirectory();
    }
    else
    {
```

```
        jLabel_messages.setText("Document creation failed.");
    }
}
```



The following example creates a document object in the repository:

► **Example 5-6: The TutorialMakeDocument class**

```
package com.emc.tutorial;

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

public class TutorialMakeDocument
{
    public TutorialMakeDocument()
    {
    }

    public Boolean makeDocument(
        IDfSessionManager sessionManager,
        String repositoryName,
        String documentName,
        String documentType,
        String sourcePath,
        String parentName)
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfSysObject newDoc =
                (IDfSysObject) mySession.newObject(DM_DOCUMENT);
            newDoc.setObjectName(documentName);
            newDoc.setContentType(documentType);
            newDoc.setFile(sourcePath);
            newDoc.link(parentName);
            newDoc.save();
            return true;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return false;
        }
        finally
        {
            sessionManager.release(mySession);
        }
    }
    public static final String DM_DOCUMENT = "dm_document";
}
```



5.6 Accessing attributes

Having created a document or folder, you can examine its attributes by performing a “dump” of all attributes or by specifying a particular attribute you want to examine.

5.6.1 Dumping Attributes

The most convenient way to begin working with attributes is to “dump” all of the object attributes to get a list of their names and values.

To add a Dump Attributes button to the DFC Base Tutorial Frame:

1. Create a JButton control named jButton_dumpAttributes.
2. Add a **button handler** method for Dump Attributes.
3. Create and deploy the class **TutorialDumpAttributes**.

 **Example 5-7: The Dump Attributes button handler**

```
private void jButton_dumpAttributesActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();

    if (list_id.getSelectedIndex() == -1)
    {
        jLabel_messages.setText(
            "Select an item in the file list to dump attributes."
        );
    }
    else
    {
        String objectIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        TutorialDumpAttributes tda = new TutorialDumpAttributes();
        jTextArea_results.setText(
            tda.dumpAttributes(
                m_sessionManager,
                repositoryName,
                objectIdString)
        );
        jLabel_messages.setText("Query complete.");
    }
}
```



 **Example 5-8: TutorialDumpAttributes class**

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfContainment;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;

public class TutorialDumpAttributes
{
    public TutorialDumpAttributes()
```

```

    }

    public String dumpAttributes
    (
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString
    )
    {
        IDfSession mySession = null;
        StringBuffer attributes = new StringBuffer("");
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            IDfId idObj = mySession.getIdByQualification(
                "dm_sysobject where r_object_id='" + objectIdString + "'"
            );
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            attributes.append(sysObj.dump());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            attributes.append("\nException: " + ex.toString());
        }
        finally
        {
            sessionManager.release(mySession);
        }
        return attributes.toString();
    }
}

```



5.6.2 Getting a single attribute by name

If you know the name of the attribute you want to access (or you just looked it up by dumping all of the attributes), you can request the specific attribute directly.

To add a Get Attribute By Name button to the DFC Base Tutorial Frame:

1. Create a JTextField control named jTextField_attributeName.
2. Create a JButton control named jButton_getAttributeByName.
3. Add a handler method for the **Get Attribute By Name** button.
4. Create and deploy the class **TutorialGetAttributeByName**.



Example 5-9: Handler for the Get Attribute By Name button

```

private void jButton_getAttributeByNameActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();

    if (list_id.getSelectedIndex() == -1 |
        jTextField_attributeName.getText().equals(""))
    {
        jLabel_messages.setText(
            "Select an item and enter an attribute name to get" +

```

```

        " attribute information."
    );
}
else
{
    String objectIdString =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    String attributeName = jTextField_attributeName.getText();
    TutorialGetAttributeByName tgabn =
        new TutorialGetAttributeByName();
    jTextArea_results.setText(
        tgabn.getAttribute(
            m_sessionManager,
            repositoryName,
            objectIdString,
            attributeName
        )
    );
    jLabel_messages.setText("Query complete.");
}
}
}

```



Example 5-10: The TutorialGetAttributeByName class

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.IDfId;

public class TutorialGetAttributeByName
{
    public TutorialGetAttributeByName()
    {
    }

    public String getAttribute(
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString,
        String attributeName
    )
    {
        IDfSession mySession = null;
        StringBuffer attribute = new StringBuffer("");
        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + objectIdString + "'"
                );
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            attribute.append(attributeName);
            attribute.append(": ");
            attribute.append(sysObj.getValue(attributeName).toString());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            attribute = new StringBuffer("");
            attribute.append(ex.toString());
        }
    // Always, always, always release the session when finished using it.
    finally
    {
    }
}

```

```

        {
            sessionManager.release(mySession);
        }
        return attribute.toString();
    }
}

```



5.6.3 Getting a single attribute by number

There may be times where you capture the attribute's index number rather than its name. The `IDfSystemObject.getAttr()` method requires the attribute name as the argument. You can use the attribute's index number to get the attribute name, then continue on to get the attribute.

To add a Get Attribute By Number button to the DFC Base Tutorial Frame:

1. Create a `JTextField` control named `jTextField_attributeNumber`.
2. Create a `JButton` control named `jButton_getAttributeByNumber`.
3. Add a handler method for the **Get Attribute By Number** button.
4. Create and deploy the class `TutorialGetAttributeByNumber`.



Example 5-11: Handler for the Get Attribute By Number button

```

private void jButton_getAttributeByNumberActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();

    if (list_id.getSelectedIndex() == -1 |
        jTextField_attributeNum.getText().equals(""))
    {
        jLabel_messages.setText(
            "Select an item and enter an attribute number to get" +
            "attribute information.");
    }
    else
    {
        String objectIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        int theIndex = Integer.parseInt(jTextField_attributeNum.getText());

        TutorialGetAttributeByNumber tga =
            new TutorialGetAttributeByNumber();

        jTextArea_results.setText(
            tga.getAttribute(
                m_sessionManager,
                repositoryName,
                objectIdString,
                theIndex
            )
        );
        jLabel_messages.setText("Query complete.");
    }
}

```



 **Example 5-12: The TutorialGetAttributeByNumber class**

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.IDfId;

public class TutorialGetAttributeByNumber {
    public TutorialGetAttributeByNumber() {
    }
    public String getAttribute (
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString,
        int theIndex
    ) {
        IDfSession mySession = null;
        StringBuffer attribute = new StringBuffer("");
        try {
            mySession = sessionManager.getSession(repositoryName);
            IDfId idObj = mySession.getIdByQualification(
                "dm_sysobject where r_object_id='" + objectIdString + "'");
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            attribute.append(sysObj.getAttr(theIndex).getName());
            attribute.append(": ");
            attribute.append(sysObj.getValueAt(theIndex).toString());
        }
        catch (Exception ex) {
            ex.printStackTrace();
            attribute = new StringBuffer("");
            attribute.append(ex.toString());
        }
        finally {
            sessionManager.release(mySession);
        }
        return attribute.toString();
    }
}

```



5.7 Setting attributes

Attributes are set according to their datatype. For example, you set a String value by calling the following method:

```
IDfTypedObject.setString(String attributeName, String value)
```

The maximum length of the string input field (*String value*) is calculated based on the number of bytes after the UNICODE conversion. Therefore, for the multibyte characters (for example, non-English characters), you must ensure to provide the value for *String value* based on the number of bytes.

The available datatypes are Foundation Java API versions of Boolean, integer, String, time, DfID, and undefined. In practice, you will not work with the DfID because all object IDs are set internally and are not mutable by custom applications.

Time values (including date values) are stored as numeric values, and are entered or displayed using a pattern mask. For example, in these examples, we use the pattern

IDfTime.DF_TIME_PATTERN_2, mm/dd/yyyy. The complete list of time formats can be found in the Javadoc entry for IDfTime.

5.7.1 Supported date and time formats

The following table lists the Foundation Java API supported date formats:

| | | | |
|------------|-------------|-----------|----------|
| yyyy/MM/dd | yyyy/MM/d | yyyy/M/dd | yyyy/M/d |
| yy/MM/dd | yy/MM/d | yy/M/dd | yy/M/d |
| yyyy/dd/MM | yyyy/dd/M | yyyy/d/MM | yyyy/d/M |
| yy/dd/MM | yy/dd/M | yy/d/MM | yy/d/M |
| MM/dd/yyyy | MM/d/yyyy | M/dd/yyyy | M/d/yyyy |
| MM/dd/yy | MM/d/yy | M/dd/yy | M/d/yy |
| dd/MM/yyyy | d/MM/yyyy | dd/M/yyyy | d/M/yyyy |
| dd/MM/yy | d/MM/yy | dd/M/yy | d/M/yy |
| dd/MMM/yy | dd/MMM/yyyy | | |

The following table lists the Foundation Java API supported time formats for the preceding list of date formats:

| | | | |
|----------|---------|-------|------|
| hh:mm:ss | h:mm:ss | hh:mm | h:mm |
| HH:mm:ss | H:mm:ss | HH:mm | H:mm |

5.7.2 Setting a single attribute

You can set attributes directly by type. Most often, you will have a specific control that will set a specific data type. Alternatively, this example queries for the data type of the attribute name the user supplies, then uses a switch statement to set the value accordingly.

To add a Set Attribute By Name button to the DFC Base Tutorial Frame:

1. If you have not done so already, create a JTextField control named jTextField_attributeName.
2. Create a JTextField control named jTextFieldAttributeValue.
3. Create a JButton control named jButton_setAttributeName.
4. Add a **button handler method for Set Attribute By Name**.
5. Create and deploy the class **TutorialSetAttributeByName**.

► **Example 5-13: Handler for the Set Attribute By Name button**

```
private void jButton_setAttributeNameActionPerformed(ActionEvent e)
{
```

```

String repositoryName = jTextField_repositoryName.getText();
String attributeName = jTextField_attributeName.getText();
String attributeValue = jTextField_attributeValue.getText();

if (list_id.getSelectedIndex() == -1 |
    jTextField_attributeName.getText().equals(""))
{
    jLabel_messages.setText(
        "Select an item, enter an attribute name and value to set" +
        " attribute information.");
}
else
{
    String objectIdString =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    TutorialSetAttributeByName tsabn =
        new TutorialSetAttributeByName();
    jTextArea_results.setText(
    tsabn.setAttributeByName(
        m_sessionManager,
        repositoryName,
        objectIdString,
        attributeName,
        attributeValue)
    );
    jLabel_messages.setText("Set attribute complete.");
}
}

```



Example 5-14: The TutorialSetAttributeByName class

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfType;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.DfTime;
import com.documentum.fc.common.IDfId;
import com.documentum.fc.common.IDfTime;

public class TutorialSetAttributeByName
{
    public TutorialSetAttributeByName()
    {
    }

    public String setAttributeByName(
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString,
        String attributeName,
        String attributeValue)
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='"
                    + objectIdString + "'");
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
        }
    }
}

```

```

int attributeDatatype = sysObj.getAttrDataType(attributeName);
StringBuffer results = new StringBuffer("");
results.append("Previous value: " +
              sysObj.getValue(attributeName).toString());
switch (attributeDatatype)
{
    case IDfType.DF_BOOLEAN:
        if (attributeValue.equals("F") |
            attributeValue.equals("f") |
            attributeValue.equals("0") |
            attributeValue.equals("false") |
            attributeValue.equals("FALSE"))
            sysObj.setBoolean(attributeName, false);
        if (attributeValue.equals("T") |
            attributeValue.equals("t") |
            attributeValue.equals("1") |
            attributeValue.equals("true") |
            attributeValue.equals("TRUE"))
            sysObj.setBoolean(attributeName, true);
        break;

    case IDfType.DF_INTEGER:
        sysObj.setInt(attributeName,
                      Integer.parseInt(attributeValue));
        break;

    case IDfType.DF_STRING:
        sysObj.setString(attributeName, attributeValue);
        break;

        // This case should not arise - no user-settable IDs
    case IDfType.DF_ID:
        IDfId newId = new DfId(attributeValue);
        sysObj.setId(attributeName, newId);
        break;

    case IDfType.DF_TIME:
        DfTime newTime =
            new DfTime(attributeValue, IDfTime.DF_TIME_PATTERN2);
        sysObj.setTime(attributeName, newTime);
        break;

    case IDfType.DF_UNDEFINED:
        sysObj.setString(attributeName, attributeValue);
        break;
}

if (sysObj.fetch(null))
{
    results = new StringBuffer("Object is no longer current.");
}
else
{
    sysObj.save();
    results.append("\nNew value: " + attributeValue);
}
return results.toString();
}
catch (Exception ex)
{
    ex.printStackTrace();
    return "Set attribute command failed.";
}
finally
{
    sessionManager.release(mySession);
}
}
}

```



5.7.3 Setting an attribute by number

The IDfSysObject.setAttr() methods require an attribute name. If your application captures the attribute's index number, you can use it to get the name of the attribute, then continue on to set the value.

To add a Set Attribute By Number button to the DFC Base Tutorial Frame:

1. If you have not done so already, create a JTextField control named jTextField_attributeNumber.
2. If you have not done so already, create a JTextField control named jTextField_attributeValue.
3. Create a JButton control named jButton_setAttributeByNumber.
4. Add a **button handler method for Set Attribute By Number**.
5. Create and deploy the class **TutorialSetAttributeByNumber**.



Example 5-15: Handler for the Set Attribute By Number button

```
private void jButton_setAttributeByNumberActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String attributeNumber = jTextField_attributeNum.getText();
    String attributeValue = jTextField_attributeValue.getText();

    if (list_id.getSelectedIndex() == -1 | jTextField_attributeNum.getText().equals(""))
    {
        jLabel_messages.setText(
            "Select an item, enter an attribute name and value to set" +
            " attribute information."
        );
    }
    else
    {
        String objectIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        TutorialSetAttributeByNumber tsabn =
            new TutorialSetAttributeByNumber();
        jTextArea_results.setText(
            tsabn.setAttributeByNumber(
                m_sessionManager,
                repositoryName,
                objectIdString,
                attributeNumber,
                attributeValue
            )
        );
        jLabel_messages.setText("Set attribute complete.");
    }
}
```



▶ **Example 5-16: The TutorialSetAttributeByNumber class**

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfType;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.DfTime;
import com.documentum.fc.common.IDfId;
import com.documentum.fc.common.IDfTime;

public class TutorialSetAttributeByNumber
{
    public TutorialSetAttributeByNumber()
    {
    }

    public String setAttributeByNumber
    (
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString,
        String attributeNumber,
        String attributeValue
    )
    {
        IDfSession mySession = null;

        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='"
                    + objectIdString + "'"
                );
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            int attributeInt = Integer.parseInt(attributeNumber);
            String attributeName = sysObj.getAttr(attributeInt).getName();
            int attributeDatatype = sysObj.getAttrDataType(attributeName);

            StringBuffer results = new StringBuffer("");
            results.append("Previous value: " +
                sysObj.getValue(attributeName).toString());
            switch (attributeDatatype)
            {
                case IDfType.DF_BOOLEAN:
                    if (attributeValue.equals("F") |
                        attributeValue.equals("f") |
                        attributeValue.equals("0") |
                        attributeValue.equals("false") |
                        attributeValue.equals("FALSE"))
                        sysObj.setBoolean(attributeName, false);
                    if (attributeValue.equals("T") |
                        attributeValue.equals("t") |
                        attributeValue.equals("1") |
                        attributeValue.equals("true") |
                        attributeValue.equals("TRUE"))
                        sysObj.setBoolean(attributeName, true);
                    break;

                case IDfType.DF_INTEGER:
                    sysObj.setInt(attributeName,
                        Integer.parseInt(attributeValue));
                    break;
            }
        }
    }
}

```

```

        case IDfType.DF_STRING:
            sysObj.setString(attributeName, attributeName);
            break;

        // This case should not arise - no user-settable IDs
        case IDfType.DF_ID:
            IDfId newId = new DfId(attributeValue);
            sysObj.setId(attributeName, newId);
            break;

        case IDfType.DF_TIME:
            DfTime newTime =
                new DfTime(attributeValue, IDfTime.DF_TIME_PATTERN2);
            sysObj.setTime(attributeName, newTime);
            break;

        case IDfType.DF_UNDEFINED:
            sysObj.setString(attributeName, attributeName);
            break;
        }

        if (sysObj.fetch(null))
        {
            results = new StringBuffer("Object is no longer current.");
        }
        else
        {
            sysObj.save();
            results.append("\nNew value: " + attributeName);
        }
        return results.toString();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        return "Set attribute command failed.";
    }
    finally
    {
        sessionManager.release(mySession);
    }
}
}

```



5.7.4 Appending a repeating attribute

If you use the regular set[Datatype] method for setting a repeating attribute, the first value (at the zero index) will be set with the value you provide. You can use an append[Datatype] method to add a value to a repeating attribute.

To add an Append Repeating Attribute button to the DFC Base Tutorial Frame:

1. If you have not already done so, create a JTextField control named jTextField_attributeName.
2. If you have not already done so, create a JTextField control named jTextField_attributeValue.
3. Create a JButton control named jButton_appendRepeatingAttribute.
4. Add a **button handler** method for Append Repeating Attribute.

5. Create and deploy the class **TutorialAppendRepeatingAttribute**.

 **Example 5-17: Handler for the Append Repeating Attribute button**

```
private void jButton_appendRepeatingAttributeActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String attributeName = jTextField_attributeName.getText();
    String attributeValue = jTextField_attributeValue.getText();

    if (list_id.getSelectedIndex() == -1 ||
        jTextField_attributeName.getText().equals(""))
    {
        jLabel_messages.setText(
            "Select an item, enter an attribute name, index, and value " +
            "to set repeating attribute information.");
    }
    else
    {
        String objectIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        TutorialAppendRepeatingAttribute tara =
            new TutorialAppendRepeatingAttribute();
        tara.appendRepeatingAttribute(
            m_sessionManager,
            repositoryName,
            objectIdString,
            attributeName,
            attributeValue
        );
        jLabel_messages.setText("Set attribute complete.");
    }
}
```



 **Example 5-18: The TutorialAppendRepeatingAttribute class**

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfType;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.Dftime;
import com.documentum.fc.common.IDfId;
import com.documentum.fc.common.IDftime;

public class TutorialAppendRepeatingAttribute
{
    public TutorialAppendRepeatingAttribute()
    {

    }

    public String appendRepeatingAttribute
    (
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString,
        String attributeName,
        String attributeValue)
    {
        IDfSession mySession = null;
        try
        {
```

```
mySession = sessionManager.getSession(repositoryName);
IDfId idObj =
    mySession.getIdByQualification(
        "dm_sysobject where r_object_id='"
        + objectIdString + "'"
    );

IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

int attributeDatatype = sysObj.getAttrDataType(attributeName);
StringBuffer results = new StringBuffer("");
results.append("Previous value: " +
    sysObj.getValue(attributeName).toString());

switch (attributeDatatype)
{
    case IDfType.DF_BOOLEAN:
        if (attributeValue.equals("F") |
            attributeValue.equals("f") |
            attributeValue.equals("0") |
            attributeValue.equals("false") |
            attributeValue.equals("FALSE"))
            sysObj.setBoolean(attributeName, false);
        if (attributeValue.equals("T") |
            attributeValue.equals("t") |
            attributeValue.equals("1") |
            attributeValue.equals("true") |
            attributeValue.equals("TRUE"))
            sysObj.appendBoolean(attributeName, true);
        results.append("\nNew value: " + attributeValue);
        break;

    case IDfType.DF_INTEGER:
        sysObj.appendInt(attributeName,
            Integer.parseInt(attributeValue));
        break;

    case IDfType.DF_STRING:
        sysObj.appendString(attributeName, attributeValue);
        break;

    // This case should not arise - no user-settable IDs
    case IDfType.DF_ID:
        IDfId newId = new DfId(attributeValue);
        sysObj.appendId(attributeName, newId);
        break;

    case IDfType.DF_TIME:
        Dftime newTime =
            new Dftime(attributeValue, IDfTime.DF_TIME_PATTERN2);
        if (newTime.isValid())
            sysObj.appendTime(attributeName, newTime);
        else
            results = new StringBuffer("");
        break;

    case IDfType.DF_UNDEFINED:
        sysObj.appendString(attributeName, attributeValue);
        break;
    }
    sysObj.save();
    return results.toString();
}
catch (Exception ex)
{
    ex.printStackTrace();
    return "Set attribute command failed.";
}
finally
{
    sessionManager.release(mySession);
}
```

```
    }
}
```



5.8 Removing an attribute value

You can also remove attribute values from system objects. If the attribute value is a single value, you can set the value of the variable to `<null>`. For a repeating attribute, you can either delete an individual value by providing the index of the value, or delete all values for the attribute.

To add a Remove Attribute button to the DFC Base Tutorial Frame:

1. If you have not already done so, create a JTextField control named `jTextField_attributeName`.
2. Create a JTextField control named `jTextField_attributeIndex`.
3. Create a JButton control named `jButton_removeAttribute`.
4. Add a **button handler method for Remove Attribute**.
5. Create and deploy the class `TutorialRemoveAttribute`.

The following example uses simple tests to decide which of these options to choose your own application will most likely need to use more robust decision-making logic, but the API calls will be the same after you have determined which attributes are to be removed:

Example 5-19: Handler for the Remove Attribute button

```
private void jButton_removeAttributeActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String attributeName = jTextField_attributeName.getText();
    String objectIdString =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    String attributeIndex = jTextField_attributeIndex.getText();
    TutorialRemoveAttribute tra = new TutorialRemoveAttribute();
    if (
        tra.removeAttribute(
            m_sessionManager,
            repositoryName,
            objectIdString,
            attributeName,
            attributeIndex
        )
    )
    {
        jLabel_messages.setText("Attribute removed.");
    }
    else
    {
        jLabel_messages.setText("Attribute removal operation failed.");
    }
}
```



Example 5-20: The TutorialRemoveAttribute class

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.IDfId;

public class TutorialRemoveAttribute
{
    public TutorialRemoveAttribute()
    {
    }

    public Boolean removeAttribute
    (
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString,
        String attributeName,
        String attributeIndex
    )
    {
        IDfSession mySession = null;
        TutorialSessionManager mySessMgr = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + objectIdString + "'"
                );
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            int indexInt = -1;
            try
            {
                indexInt = Integer.parseInt(attributeIndex);
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
            if (sysObj.isAttrRepeating(attributeName))
            {
                if (indexInt > -1)
                {
                    sysObj.remove(attributeName, indexInt);
                    sysObj.save();
                }
                else
                {
                    sysObj.removeAll(attributeName);
                    sysObj.save();
                }
            }
            else
            {
                TutorialSetAttributeByName tsabn =
                    new TutorialSetAttributeByName();
                tsabn.setAttributeByName(
                    sessionManager,
                    repositoryName,
                    objectIdString,
                    attributeName,
                    attributeIndex
                );
                if (indexInt > -1)
                {
                    tsabn.remove(attributeName, indexInt);
                    tsabn.save();
                }
                else
                {
                    tsabn.removeAll(attributeName);
                    tsabn.save();
                }
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

```

        null
    );
}

return true;
}
catch (Exception ex)
{
    ex.printStackTrace();
    return false;
}
finally
{
    sessionManager.release(mySession);
}
}
}

```



5.9 Getting object content

The `IDfSysObject.getContent()` command lets you get the contents of a document as a `ByteArrayInputStream`. If you are working with the content of a file, particularly if you intend to save changes, you should use the operation interfaces, which wrap the behavior required to safely check out, update, and check in document content. There may be times, though, that you want to read the content of a file without manipulating it. In those cases, you can use the `getContent()` method.

To add a Get Content button to the DFC Base Tutorial Frame:

1. Create a JButton control named `jButton_getDocumentContent`.
2. Add a **button handler** method for **Get Document Content**.
3. Create and deploy the class **TutorialGetTextContent**.



Example 5-21: Handler for the Get Document Content button

```

private void jButton_getDocumentContentActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String objectIdString =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

    TutorialGetTextContent tgtc = new TutorialGetTextContent();

    jTextArea_results.setText(
        tgtc.getContent(
            m_sessionManager,
            repositoryName,
            objectIdString
        )
    );
}

```



 **Example 5-22: The TutorialGetTextContent class**

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.IDfId;

import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.InputStreamReader;

public class TutorialGetTextContent
{
    public TutorialGetTextContent()
    {
    }

    public String getContent(
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString
    )
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");

        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + objectIdString + "'"
                );
        }

        // Instantiate an object from the ID.

        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
        if (sysObj.getContentType().equals("crtext") |
            sysObj.getContentType().equals("text"))
        {
            ByteArrayInputStream buf = sysObj.getContent();
            int i = 0;
            InputStreamReader readInput =
                new InputStreamReader(buf, "UTF-8");
            BufferedReader br = new BufferedReader(readInput);
            while (br.ready())
            {
                sb.append(br.readLine());
                sb.append("\n");
            }
            return sb.toString();
        }
        else
        {
            return "Use getContent to view text documents.";
        }
    }

    // Handle any exceptions.
    catch (Exception ex)
    {
        ex.printStackTrace();
        return "Exception has been thrown: " + ex;
    }
    // Always, always, release the session in the "finally" clause.
}
```

```

        finally
        {
            sessionManager.release(mySession);
        }
    }
}

```



5.10 Destroying an object

You can use the `IDfSystemObject.destroyAllVersions()` method to permanently remove an object from the database. If you use the `IDfPersistentObject.destroy()` method, you will destroy only the specific system object corresponding to the `r_object_id` you provide. In this example, we use the `destroyAllVersions()` method, which destroys not only the system object with the corresponding ID but all iterations of the object.

If you attempt to destroy a directory that has children, the method will return an error.

To add a Destroy button to the DFC Base Tutorial Frame:

1. Create a JButton control named `jButton_destroy`.
2. Add a handler method for the **Destroy** button.
3. Create and deploy the class `TutorialDestroyObject`.



Example 5-23: Handler for the Destroy button

```

private void jButton_destroyActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String objectIdString =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

    TutorialDestroyObject tdo = new TutorialDestroyObject();
    if (tdo.destroyObject(
        m_sessionManager,
        repositoryName,
        objectIdString
    ))
    {
        getDirectory();
        jLabel_messages.setText("Destroyed object '" + objectIdString +
                               "'.");
    }
    else
    {
        jLabel_messages.setText("Destroy command failed.");
    }
}

```



 **Example 5-24: The TutorialDestroyObject class**

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.IDfId;

public class TutorialDestroyObject
{
    public TutorialDestroyObject()
    {
    }

    public Boolean destroyObject(
        IDfSessionManager sessionManager,
        String repositoryName,
        String objectIdString
    )
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='"
                    + objectIdString + "'"
                );
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
            sysObj.destroyAllVersions();
            return true;
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return false;
        }
        finally
        {
            sessionManager.release(mySession);
        }
    }
}

```



5.11 Real-time content deletion

For a conventional object deletion, when you delete an object, the contents of the object are deleted asynchronously. A real-time content deletion removes the contents immediately with the object.

Starting with Foundation Java API 7.0, you can use the `setRealtimeContentDel(bool enable)` method of `DfSession` to control whether or not to enable the real-time content deletion feature for a Foundation Java API session. This method throws `DfException` with the `DFC_REALTIME_CONT_DEL_UNSUPORT` error if Documentum CM Server (an earlier version, such as 6.7) does not support this feature.

You can use the following queries to search for the contents that are not deleted successfully:

- For real-time content deletion failures:

```
Select r_object_id from dmr_content where content_state = 2
```

- For conventional (asynchronous) content deletion failures:

```
Select r_object_id from dmr_content where content_state = 1
```

- For all deletion failures:

```
Select r_object_id from dmr_content where content_state <> 0
```


Chapter 6

Working with document operations

This chapter describes the way to use Foundation Java API 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.

6.1 Understanding documents

The *OpenText Documentum Content Management - Server Fundamentals Guide* (EDCCS250400-GGD) explains OpenText Documentum CM facilities for managing documents. This section provides a concise summary of what you need to know to understand the remainder of this chapter.

OpenText Documentum CM 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.

OpenText Documentum CM 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.

6.1.1 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. OpenText Documentum CM imposes no limit on the depth of nesting in a virtual document.

OpenText Documentum CM 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).

OpenText Documentum CM 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, OpenText Documentum CM creates a new containment object corresponding to that component.



Note: Foundation Java API 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 Foundation Java API to check out doc, it does not check out any of the descendants. If you want Foundation Java API to check out the descendants along with the root document, you must first execute an instruction as follows:

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

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

OpenText Documentum CM 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.

6.1.2 XML documents

OpenText Documentum CM's XML support has many features. For more information about XML support, see *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)*.

Using XML support requires you to provide a controlling XML application. When you import an XML document, Foundation Java API 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, Foundation Java API 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.

6.2 Understanding operations

Operations are used to manipulate documents in OpenText Documentum CM. Operations provide interfaces and a processing environment to ensure that OpenText Documentum CM 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, perform the following steps:

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

Foundation Java API carries out the behind-the-scenes tasks associated with checking out a document. For a virtual document, for example, Foundation Java API 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 Foundation Java API does when you check out a document. Because of the number and complexity of the underlying tasks, Foundation Java API wraps seemingly elementary document-manipulation tasks in operations.



Note: A `SysObject` in the operations framework is processed only once. If you add the same object twice to an operation, it only performs the operation steps

once per object. For example, if you add a document object multiple times in order to retrieve multiple content pages with one operation, you will only retrieve one page of content. In order to retrieve multiple content pages (for example, renditions), you must use a separate operation for each operation.

An IDfClientX object provides factory methods for creating operations. After you have an IDfClientX object (say cX) and a SysObject (say doc) representing the document, the code for the checkout is as follows:

```
// 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.

6.3 Types of operation

Foundation Java API 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 lists the Foundation Java API operation types and nodes:

Table 6-1: Foundation Java API 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 |
| 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 |

| Task | Operation type | Operation node type |
|---|-------------------------------|--------------------------|
| Pre-cache objects in a OpenText™ Documentum™ Content Management Branch Office Caching Services repository | IDfPredictiveCachingOperation | IDfPredictiveCachingNode |

6.4 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.

6.4.1 Steps for manipulating documents

This section describes the basic steps common to document manipulation operations. “[Details of manipulating documents](#)” on page 80 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.

“[Handling document manipulation errors](#)” on page 109 provides detailed information.

- 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, Foundation Java API may add the linked documents to the operation.

The execute method returns a boolean value to indicate its success (true) or failure (false). [“Handling document manipulation errors” on page 109](#) provides the detailed information.

5. Process the results.

- a. Handle errors.

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

[“Processing the results” on page 82](#) provides details of how to process errors.

- 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.

6.4.2 Details of manipulating documents

This section discusses some issues and background for the steps of the general procedure in [“Steps for manipulating documents” on page 79](#).

6.4.2.1 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.

6.4.2.2 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.

6.4.2.3 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” on page 82).



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.

Foundation Java API might include additional nodes in the operation. For example, if you add a repository folder, Foundation Java API 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” on page 82). You might want to set parameters on individual nodes differently from the way you set them on the operation.

6.4.2.4 Executing the Operation

The operations package processes the objects in an operation as a group, possibly invoking many Foundation Java API calls for each object. Operations encapsulate OpenText Documentum CM client conventions for registering, naming, and managing local content files.

Foundation Java API 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.

After Foundation Java API 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 Foundation Java API proceed through the execution steps. Foundation Java API provides a limited ability for you to execute an operation in steps, so that you can perform special processing between steps. OpenText Documentum CM does not recommend this approach, because the number and identity of steps in an operation may change with future versions of Foundation Java API. If you have a programming hurdle that you cannot get over without using steps, work with OpenText Global Technical Services to design a solution.

6.4.2.5 Processing the results

If Foundation Java API 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. Foundation Java API 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.

After you have checked the errors you may want to examine and further process the results of the operation. The next section, “[Working with nodes](#)” on page 82, shows how to access the objects and results associated with the nodes of the operation.

6.4.2.6 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.

6.5 Operations for manipulating documents

This section provides sample code and discusses the different 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 Foundation Java API objects of type `IDfDocument` and `IDfFolder`.

These examples assume the use of a simple UI, similar to the one described in “[Creating a test application](#)” on page 37, which enables the user to pass a Session Manager and additional information such as a document ID or directory path to give the essential information required by the operation.

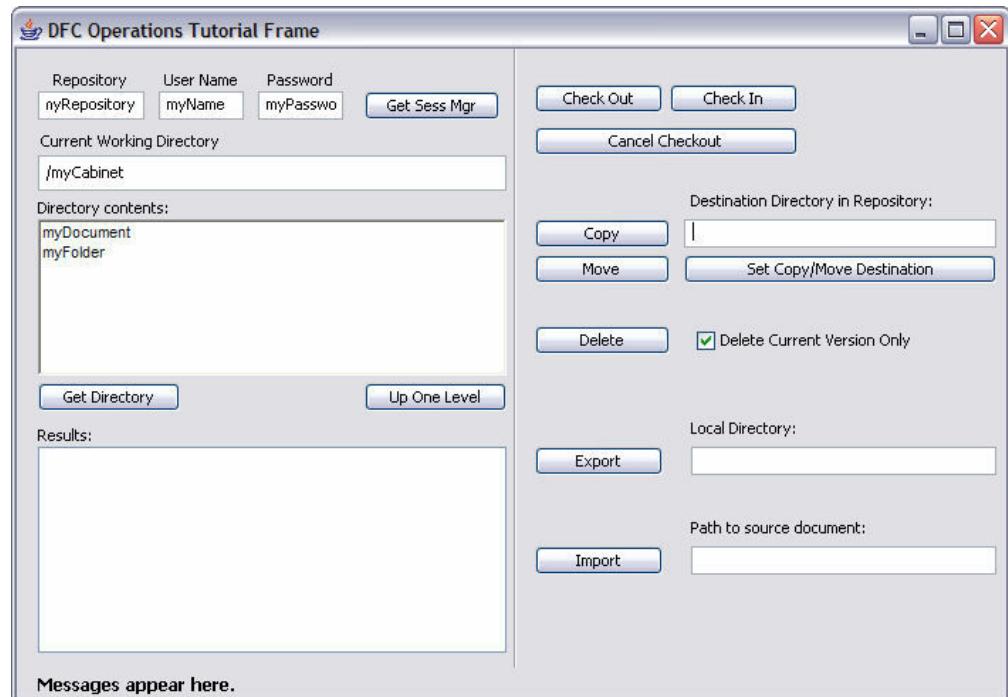


Figure 6-1: DFC Operations Tutorial Frame

6.5.1 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 Foundation Java API to manage the files it creates on the file system.

➤ Example 6-1: Handler for the Check Out button

```
private void jButton_checkOutActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    TutorialCheckOut tco = new TutorialCheckOut();
    jLabel_messages.setText(
        tco.checkoutExample(
            m_sessionManager,
            repositoryName,
            docId
        )
    );
    initDirectory();
    getDirectory();
}
```



➤ Example 6-2: TutorialCheckout.java

```
package com.emc.tutorial;

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.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfid;
import com.documentum.operations.IDfCheckoutNode;
import com.documentum.operations.IDfCheckoutOperation;

public class TutorialCheckOut
{
    public TutorialCheckOut()
    {
    }

    public String checkoutExample
    (
        IDfSessionManager sessionManager,
        String repositoryName,
        String docId
    )
    {
        StringBuffer result = new StringBuffer("");
        IDfSession mySession = null;

        try
```

```

{
    mySession = sessionManager.getSession(repositoryName);

    // Get the object ID based on the object ID string.
    IDfId idObj =
        mySession.getIdByQualification(
            "dm_sysobject where r_object_id='" + docId + "'");

    // Instantiate an object from the ID.
    IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

    // 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.append("coNode is null");
    }

    // Execute the checkout operation. Return the result.
    if (coOp.execute())
    {
        result.append("Successfully checked out file ID: " + docId);
    }
    else
    {
        result.append("Checkout failed.");
    }
    return result.toString();
}
catch (Exception ex)
{
    ex.printStackTrace();
    return "Exception has been thrown: " + ex;
}
finally
{
    sessionManager.release(mySession);
}
}
}

```



6.5.1.1 Special considerations for checkout operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

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

Foundation Java API 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.

6.5.1.2 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 6-3: 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(
        IDfSessionManager sessionManager,
        String repositoryName
        String docId)
    {
        try {
            String result = "";

            // Instantiate a session.
            IDfSession mySession = sessionManager.getSession (repositoryName);
            // 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 has been thrown: " + ex;
}
finally {
    sessionManager.release(mySession);
}
}
}

```



6.5.2 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 6-4: Handler for the Check In button

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).

```

private void jButton_checkInActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    TutorialCheckIn tci = new TutorialCheckIn();
    jLabel_messages.setText(
        tci.checkinExample(
            m_sessionManager,
            repositoryName,
            docId
        )
    )
}

```

```
    );
    initDirectory();
    getDirectory();
}
```



Example 6-5: The TutorialCheckIn class

```
package com.emc.tutorial;

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.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
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(
        IDfSessionManager sessionManager,
        String repositoryName,
        String docId
    )
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + docId + "'"
                );
        }

        // Instantiate an object from the ID.

        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

        // Instantiate a client.
        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
        // ( version + 1 )
        cio.setCheckinVersion(IDfCheckinOperation.NEXT_MAJOR);

        // When updating to the next major version, you need to explicitly
        // set the version label for the new object to "CURRENT".
        cio.setVersionLabels("CURRENT");

        // 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.";
}
finally
{
    sessionManager.release(mySession);
}
}
}

```



6.5.2.1 Special considerations for checkin operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

6.5.2.1.1 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, Foundation Java API records the path and filename of the local file that represents the content of an object. If you move or rename the file, Foundation Java API loses track of it and reports an error when you try to check it in.

Setting the content file, as in IDfCheckinNode.setFilePath, overrides Foundation Java API's saved information.

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

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

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

6.5.2.1.2 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](#)” on page 82). You can use their methods to find out many other facts about the new SysObjects associated with those nodes.

6.5.2.1.3 Behavior Change after Failed Save or Checkin

In OpenText Documentum CM 6.5 and later, if a save or checkin of an object fails, Foundation Java API reverts the object automatically, before returning the object to the application issuing the save or checkin. Reverting the object removes any changes made to the object before the attempted save or checkin. The application must then reapply any changes made before the save or checkin before reattempting the save or checkin operation. This behavior is different from how earlier DMCL releases handled failed saves or checkins. The DMCL in prior releases simply marked the object with an error state and returned it to the application.

6.5.3 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 6-6: Handler for the Cancel Checkout button

```
private void jButton_cancelCheckOutActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    TutorialCancelCheckOut tcco = new TutorialCancelCheckOut();
    jLabel_messages.setText(
        tcco.cancelCheckOutExample(
            m_sessionManager,
            repositoryName,
            docId
        )
    );
}
```



Example 6-7: TutorialCancelCheckout.java

Cancel checkout of a document.

```
package com.emc.tutorial;

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.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfException;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfCancelCheckoutNode;
import com.documentum.operations.IDfCancelCheckoutOperation;

public class TutorialCancelCheckOut
{
    public TutorialCancelCheckOut()
    {
    }

    public String cancelCheckOutExample(
        IDfSessionManager sessionManager,
        String repositoryName,
        String docId
    )
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + docId + "'"
                );
        }
        catch (DfException e)
        {
            System.out.println("An error occurred: " + e.getMessage());
        }

        // Instantiate an object from the ID.
        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

        // 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;

        // Populate the cancel checkout node and add it to the cancel checkout
        // operation.
        node = (IDfCancelCheckoutNode) cco.add(doc);

        // Check to see if the node is null - this will not throw an error.
        if (node == null)
        {
            System.out.println("The node is 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;
    }
    // Handle any exceptions.
    catch (Exception ex)
    {
        ex.printStackTrace();
        return "Exception has been thrown: " + ex;
    }

    // Always, always, release the session in the "finally" clause.
    finally
    {
        sessionManager.release(mySession);
    }
}
```



6.5.3.1 Special considerations for cancel checkout operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

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

6.5.3.2 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 6-8: 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 virtual 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.";
}
}
}

```



6.5.4 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 6-9: Handler for the Import button

```
private void jButton_importActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String sourceFilePath = jTextField_importFilePath.getText();
    String destinationDirectory = jTextField cwd.getText();
```

```
TutorialImport ti = new TutorialImport();
jLabel_messages.setText(
    ti.importExample(
        m_sessionManager,
        repositoryName,
        destinationDirectory,
        sourceFilePath
    )
);
initDirectory();
getDirectory();
}
```



➤ Example 6-10: TutorialImport.java

```
package com.emc.tutorial;

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.IDfSessionManager;
import com.documentum.fc.common.DfException;
import com.documentum.operations.IDfFile;
import com.documentum.operations.IDfImportNode;
import com.documentum.operations.IDfImportOperation;

public class TutorialImport
{
    public TutorialImport()
    {
    }

    public String importExample(
        IDfSessionManager sessionManager,
        String repositoryName,
        String destinationDirectory,
        String sourceFilePath
    )
    {
        IDfSession mySession = null;

        try
        {
            mySession = sessionManager.getSession(repositoryName);

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

            // Use the factory method to create an IDfImportOperation instance.
            IDfImportOperation opi = clientx.getImportOperation();

            // You must explicitly set the session for an import operation.
            opi.setSession(mySession);

            // Create an instance of the target folder.
            IDfFolder folder = mySession.getFolderByPath(destinationDirectory);

            // Create a file instance for the local file.
            IDfFile file = clientx.getFile(sourceFilePath);
            if (!file.exists())
                return ("File does not exist.");

            // Set the destination folder.
            opi.setDestinationFolderId(folder.getObjectId());
        }
    }
}
```

```

// Create the import node, adding the file to the import operation.
IDfImportNode node = (IDfImportNode) opi.add(file);
if (node == null)
    return ("Node is null.");

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

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

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

```



6.5.4.1 Special Considerations for Import Operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

6.5.4.1.1 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, Foundation Java API 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, Foundation Java API 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.

If you are importing a document with OLE links, all of the linked files can be imported for you automatically and stored as nodes in a virtual document.

[“Microsoft Object Linking and Embedding \(OLE\)” on page 13](#) provides information on configuring your system to accommodate OLE links.

6.5.4.1.2 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 Foundation Java API to perform the operation.

6.5.4.1.3 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” on page 82](#)). 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.

6.5.5 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 6-11: Handler for the Export button

```
private void jButton_exportActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    String targetLocalDirectory = jTextField_localDirectory.getText();
    TutorialExport te = new TutorialExport();
    jLabel_messages.setText(
        te.exportExample(
            m_sessionManager,
            repositoryName,
```

```

        docId,
        targetLocalDirectory)
    }
}

```



➤ Example 6-12: TutorialExport.java

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

```

package com.emc.tutorial;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfFormat;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfExportNode;
import com.documentum.operations.IDfExportOperation;

public class TutorialExport
{
    public TutorialExport()
    {
    }

    public String exportExample(
        IDfSessionManager sessionManager,
        String repositoryName,
        String docId,
        String targetLocalDirectory
    )
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");

        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='"
                    + docId + "'"
                );
        }

        // Instantiate an object from the ID.

        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

        // 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);
    }
}

```

```

// Get the document's format.
IDfFormat format = doc.getFormat();

// If necessary, append a path separator to the targetLocalDirectory value.
if (targetLocalDirectory.lastIndexOf("/") != targetLocalDirectory.length() - 1 &&
    targetLocalDirectory.lastIndexOf("\\\\") != targetLocalDirectory.length() - 1)
{
    targetLocalDirectory += "/";
}

// Set the full file path on the local system.
node.setFilePath(targetLocalDirectory + doc.getObjectName() + "." +
                  format.getDOSExtension());

// Execute and return results
if (eo.execute())
{
    return "Export operation successful." + "\n" + sb.toString();
}
else
{
    return "Export operation failed.";
}
}

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

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

```



6.5.5.1 Special considerations for export operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

By default, an export operation creates files on the local system and makes no provision for OpenText Documentum CM to manage them. You can tell Foundation Java API 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, Foundation Java API copies the latest repository version to the local file system. Foundation Java API 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.

If you are exporting a virtual document with OLE links, the document can be reassembled for you automatically. ["Microsoft Object Linking and Embedding \(OLE\)" on page 13](#) provides information on configuring your system to accommodate OLE links.

6.5.6 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 6-13: Handler for the Copy button

```
private void jButton_copyActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

    // Add a field to the application that allows you to enter a path
    // for the location of copy of the document.
    String destinationDirectory = jTextField_destinationDirectory.getText();
    TutorialCopy tc = new TutorialCopy();
    jLabel_messages.setText(
        tc.copyExample(
            m_sessionManager,
            repositoryName,
            docId,
            destinationDirectory
        )
    );
}
```



Example 6-14: TutorialCopy.java

```
package com.emc.tutorial;

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.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
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(
        IDfSessionManager sessionManager,
        String repositoryName,
        String docId,
        String destination
    )
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");

        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj = mySession.getIdByQualification(
                "dm_sysobject where r_object_id='" + docId + "'"
            );

            // Instantiate an object from the ID.
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

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

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

            // Remove the path separator if it exists.
            if (destination.lastIndexOf("/") == destination.length()-1 ||
                destination.lastIndexOf("\\" ) == destination.length()-1)
            {
                destination = destination.substring(0,destination.length()-1);
            }

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

            // Set the destination directory by ID.
            // To copy from one repository to another repository, add the folder ID of a folder from
            // the target repository
            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.";
    }
}
// Handle any exceptions.
catch (Exception ex)
{
    ex.printStackTrace();
    return "Exception has been thrown: " + ex;
}

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

```



6.5.6.1 Special considerations for copy operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

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 Foundation Java API to add related objects to the copy operation. For more information about the dm_relation and dm_relation_type objects, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

6.5.7 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 6-15: 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.";
        }
    }
}
```



6.5.7.1 Special considerations for move operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79. Options for moving are essentially the same as for copying.

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

6.5.8 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 6-16: Handler for the Delete button

```
private void jButton_deleteActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

    // Create a checkbox control that lets you set a Boolean value
    // for whether or not the application should destroy all versions
    // or just the current version (current version only = true).
    Boolean currentVersionOnly = jCheckBox_deleteCurrentOnly.isSelected();
    TutorialDelete td = new TutorialDelete();
    jLabel_messages.setText(
        td.deleteExample(
            m_sessionManager,
            repositoryName,
            docId,
            currentVersionOnly
        )
    );
    initDirectory();
    getDirectory();
}
```



Example 6-17: TutorialDelete.java

Delete a document.

```
package com.emc.tutorial;

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.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.client.IDfVirtualDocumentNode;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfDeleteNode;
import com.documentum.operations.IDfDeleteOperation;

public class TutorialDelete
{
    public TutorialDelete()
    {

    }

    public String deleteExample(
```

```
    IDfSessionManager sessionManager,
    String repositoryName,
    String docId,
    Boolean currentVersionOnly
)
{
    IDfSession mySession = null;

    try
    {
        mySession = sessionManager.getSession(repositoryName);

        // Get the object ID based on the object ID string.
        IDfId idObj = mySession.getIdByQualification(
            "dm_sysobject where r_object_id='" + docId + "'");

        // Instantiate an object from the ID.
        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

        // 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)
        {
            // Default value is SELECTED VERSIONS
            // 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";
        }
    }
    // Handle any exceptions.
    catch (Exception ex)
    {
        ex.printStackTrace();
        return "Exception has been thrown: " + ex;
    }

    // Always, always, release the session in the "finally" clause.
}
```

```

        finally
        {
            sessionManager.release(mySession);
        }
    }
}

```



6.5.8.1 Special considerations for delete operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79. If the operation entails deleting a checked out document, Foundation Java API leaves the document unmodified and reports an error.

6.5.9 Predictive caching

Predictive caching can help you to improve the user experience by sending system objects to OpenText Documentum Content Management (CM) 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 than 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 6-18: 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 */ }
}

```



6.5.9.1 Special considerations for predictive caching operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

`setTimeToLive` sets a time limit, in milliseconds, for Branch Office Caching Services. Branch Office Caching Services 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 Branch Office Caching Services 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.

6.5.10 Validating an XML document against a DTD or schema

Foundation Java API uses a modified version of the Xerces XML parser, which is in `dfc.jar`.

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, Foundation Java API 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 6-19: 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 */ }
```



6.5.10.1 Special considerations for validation operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

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.

6.5.11 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.

Example 6-20: 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 6-21: 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 6-22: 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 */ }
}

```

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



6.5.11.1 Special considerations for XML transform operations

Follow the steps in “[Steps for manipulating documents](#)” on page 79.

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 OpenText Documentum CM 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)
- DOM (org.w3c.dom.Document)
- java.io.Reader
- URL

6.6 Handling document manipulation errors

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

6.6.1 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. Foundation Java API does not report the reason for returning a null.

6.6.2 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.

6.6.2.1 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 6-23: 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 6-24: 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; }
}

```



6.6.2.2 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, Foundation Java API 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 Foundation Java API 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`

Foundation Java API supplies the percentage of completion of the operation and of its current step. Foundation Java API expects a response that tells it whether to continue or to abort the operation.

- `reportError`

Foundation Java API 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 Foundation Java API gives you more choices. Foundation Java API 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.

6.7 Operations and transactions

Operations do not use session transactions (see “[Transactions](#)” on page 29), 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. “[Transactions](#)” on page 29 provides information about what session transactions can and cannot do.

Chapter 7

Using the Business Object Framework (BOF)

This chapter introduces the Business Object Framework (BOF).

7.1 Overview of BOF

BOF's main goals are to centralize and standardize the process of customizing OpenText Documentum CM 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 OpenText Documentum CM 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).

7.2 BOF infrastructure

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

7.2.1 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).

Foundation Java API uses a special type of repository folder (`dmc_module`) to contain a module. For more information about the attributes of the `dmc_module` type, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*. 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. [“Service-based Business Objects \(SBOs\)” on page 116](#) provides 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 Foundation Java API repository type (normally, DfDocument) and implements the IDfBusinessObject interface. [“Type-based Business Objects \(TBOs\)” on page 123](#) provides 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” on page 171](#)), 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 such as 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 Foundation Java API 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, Foundation Java API gives preference to the repository based registry.

7.2.2 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](#)” on page 116 explains how Foundation Java API does this.

Delivering implementation classes dynamically from a repository means that you do not 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 Foundation Java API 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. Foundation Java API 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). Foundation Java API does not load TBOs, aspects, or simple modules into the cache until an application tries to use them. After Foundation Java API has downloaded a module, it only reloads parts of the module that change.

Foundation Java API 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.cache.currency_check_interval` in the `dfc.properties` file. The default value is 60 seconds.

If Foundation Java API 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.

Foundation Java API 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 Foundation Java API installation share the cache. Do not share the cache among more than one Foundation Java API installation. If you have more than one Foundation Java API installation, ensure that the value of `dfc.cache.dir` in the `dfc.properties` file does not point to the same directory.

7.2.3 Global registry

Foundation Java API delivers SBOs from a central repository. That repository's registry is called the global registry.

7.2.3.1 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 Foundation Java API 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.

7.2.3.2 Accessing the global registry

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

In addition to efficiency, local caching provides backup if the global registry is unavailable. Only when the client application tries to instantiate an SBO for the first time, the corresponding implementation and interface jar files are downloaded from the global registry to the local cache. Thereafter, Foundation Java API reloads only those parts of the module that change. For that purpose, Foundation Java API checks for updates related to the cached modules whenever an application tries to use them or after the interval specified in seconds for the Foundation Java API property `dfc.bof.cache.currency_check_interval` in the `dfc.properties` file. Its default value is 60 seconds.

The `dfc.properties` file contains properties that relate to accessing the global registry. “[BOF and global registry settings](#)” on page 10 provides information about using these properties.

7.3 Service-based Business Objects (SBOs)

7.3.1 SBO introduction

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

SBOs can operate on multiple object types, retrieve objects unrelated to OpenText Documentum CM 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 OpenText Documentum CM Inbox object is an SBO. It retrieves items from a user's inbox and performs operations such as 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.

7.3.2 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 Foundation Java API installation uses a global registry (see “[Global registry](#)” on page 116). The `dfc.properties` file contains the information necessary to enable Foundation Java API 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: Foundation Java API does not enforce a naming convention for SBOs, but we recommend that you follow the naming convention explained in “[Follow the Naming Convention](#)” on page 122.

7.3.3 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 Foundation Java API 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 XYZ 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:

7.3.3.1 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 such as 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.

7.3.3.2 Managing Sessions for SBOs

This section presents session manager related considerations for implementing SBOs.

7.3.3.2.1 Overview

When implementing an SBO, you normally use the getSession and releaseSession methods of DfService to obtain a Foundation Java API session and return it when finished. After you get a session, use the methods of IDfSession and other Foundation Java API 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.

7.3.3.2.2 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 ); } }
```

7.3.3.2.3 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`).

7.3.3.2.4 Maintaining State Beyond the Life of the SBO

The OpenText Documentum CM 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 Documentum CM 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](#)” on page 35 describes the `setSessionManager` mechanism and its cost in resources. “[Using Transactions With SBOs](#)” on page 120 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.

7.3.3.2.5 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`. Foundation Java API does not update this object if the session manager's state subsequently changes.

The Foundation Java API Javadocs describe the available state information.

7.3.3.3 Using Transactions With SBOs

Foundation Java API supports two transaction processing mechanisms: session based and session manager based. “[Transactions](#)” on page 29 describes the differences between the two transaction mechanisms. You cannot use session based transactions within an SBO method. Foundation Java API 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, Foundation Java API 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 ensure that you remove the folders if the document creation fails, place the two service calls into a transaction. The Foundation Java API session transaction is bound to one Foundation Java API session, so it is important to use the same Foundation Java API session across the two services calls. Each service performs its own atomic operation. At the start of each operation, they request a Foundation Java API 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. Foundation Java API 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, Foundation Java API 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, Foundation Java API creates a new session.

- When a client starts a transaction using the beginTransaction method, the session manager does not allow any other Foundation Java API-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. “[Transactions](#)” on page 29 provides information about what session manager transactions can and cannot do.

7.3.4 SBO Error Handling

The factory method that instantiates SBOs throws a variety of exceptions. For example, it throws DfServiceInstantiationException if Foundation Java API 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.

7.3.5 SBO Best Practices

This section describes best practices for using SBOs.

7.3.5.1 Follow the Naming Convention

Foundation Java API 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, write:

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

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

7.3.5.2 Do not reuse SBOs

Instantiate a new SBO each time you need one, rather than reusing one. “[Calling SBOs](#)” on page 138 provides the details.

7.3.5.3 Make SBOs Stateless

Make SBOs as close to stateless as possible. “[Stateful and stateless SBOs](#)” on page 118 provides the details.

7.3.5.4 Rely on Foundation Java API to Cache Repository Data

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

7.4 Type-based Business Objects (TBOs)

7.4.1 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 Foundation Java API types (such as documents and users) and other TBOs. TBOs extend Foundation Java API 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 Foundation Java API client application.

7.4.2 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.

7.4.2.1 Create a custom repository type

Using Documentum Composer, create and configure your custom type. [“Deploying the SBO and TBO” on page 141](#) provides an example implementation.

7.4.2.2 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 (for example, IDfDocument), and (2) whether to extend IDfBusinessObject.

While the TBO class will need to extend the base Foundation Java API 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 Foundation Java API interfaces. On the other hand, if your TBO needs to expose a large number of methods from the base Foundation Java API 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;
}
```

7.4.2.3 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 Foundation Java API 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
```

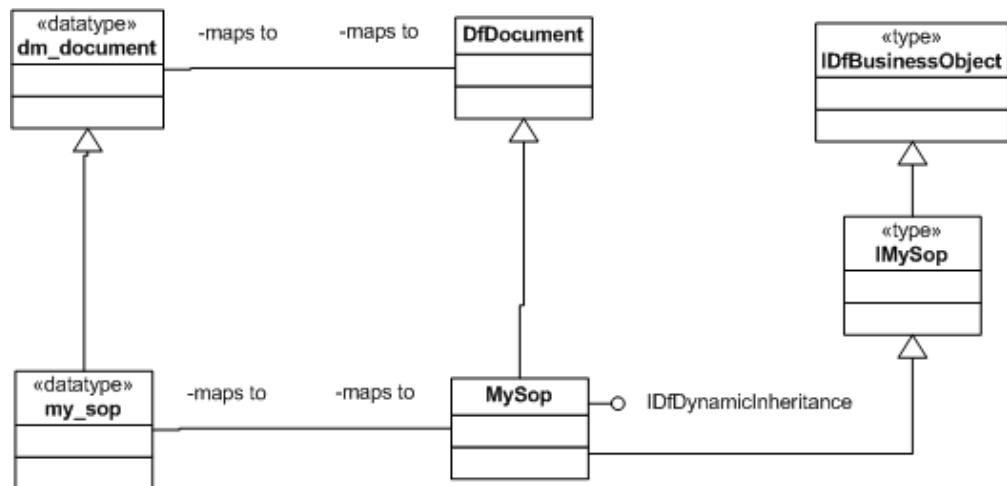


Figure 7-1: Basic TBO design

It is also an option to create more hierarchical levels between your main TBO class and the Foundation Java API 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 Foundation Java API 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
```

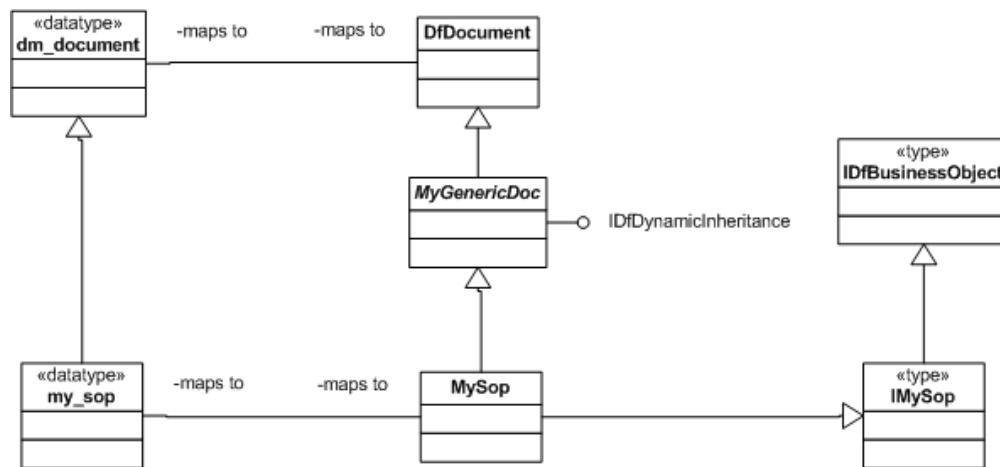


Figure 7-2: TBO design with extended intervening class

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. [“Deploying the SBO and TBO” on page 141](#) provides an example of packaging and deploying business objects.

7.4.2.4 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.

The IDfBusinessObject in the Javadocs provides the detailed information.

7.4.2.4.1 **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).

7.4.2.4.2 **getVendorString** method

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

7.4.2.4.3 **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.

7.4.2.4.4 **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.

7.4.2.5 **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 “[Signatures of Methods to Override](#)” on page 133.

7.4.3 Using a TBO from a client application

A TBO is an extension of a core Foundation Java API typed object, so instantiating a TBO is no different from instantiating a core Foundation Java API 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 docbaseName,
                               String password,
                               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);
} } }
```

7.4.4 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 OpenText Documentum CM 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 OpenText Documentum CM 6, there is transparent handling of the association of objects and sessions: if you return an object and release its session, Foundation Java API 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.

7.4.5 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 Foundation Java API 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`.

7.4.6 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.

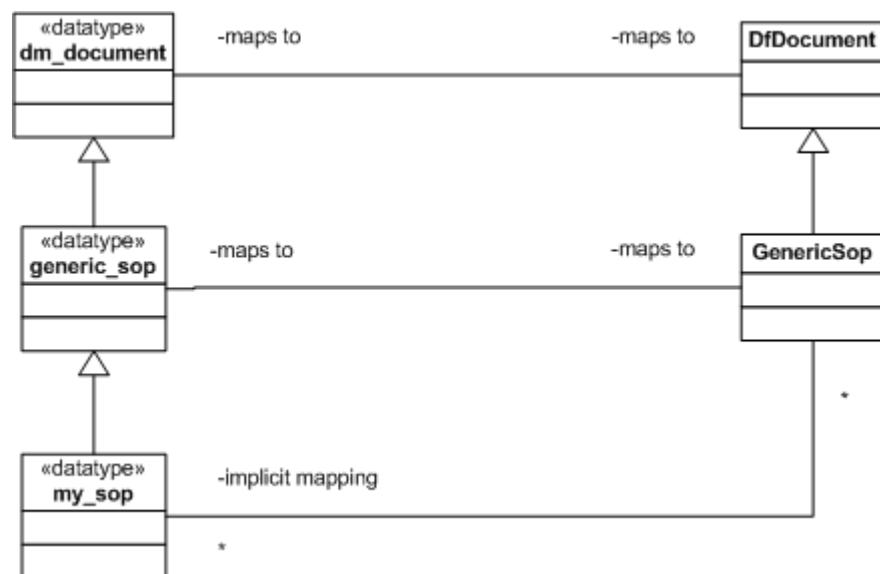


Figure 7-3: Inheritance by object subtype without associated TBO

7.4.7 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:

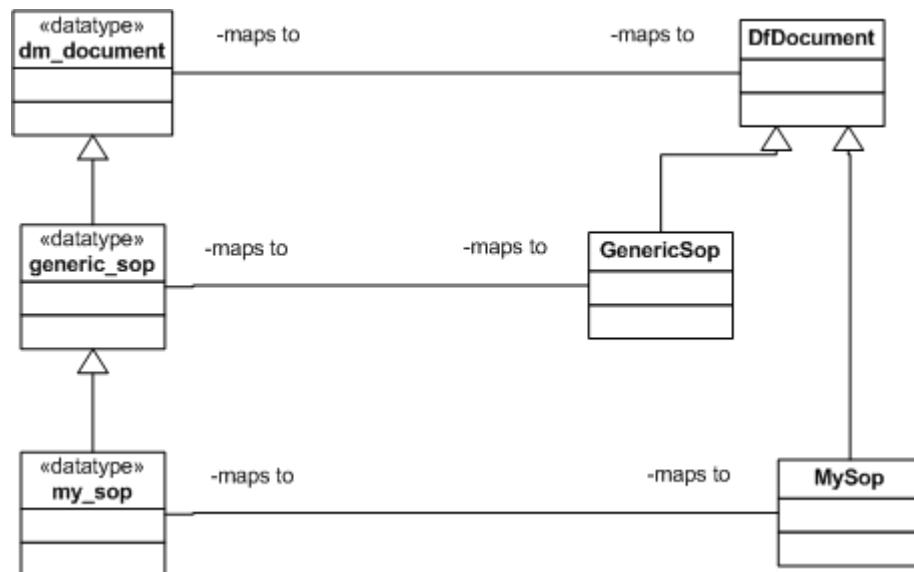


Figure 7-4: Design-time dynamic inheritance hierarchies

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:

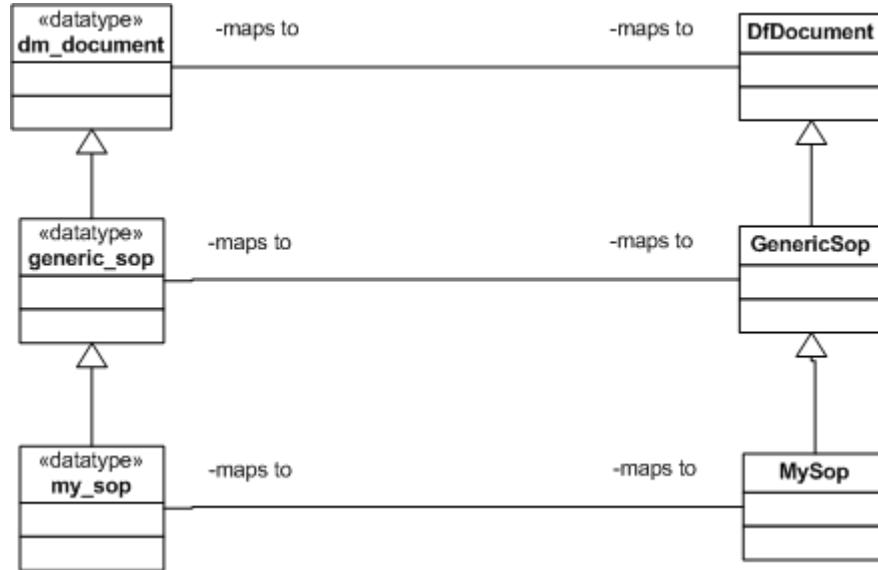


Figure 7-5: Runtime dynamic inheritance hierarchies

7.4.7.1 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:

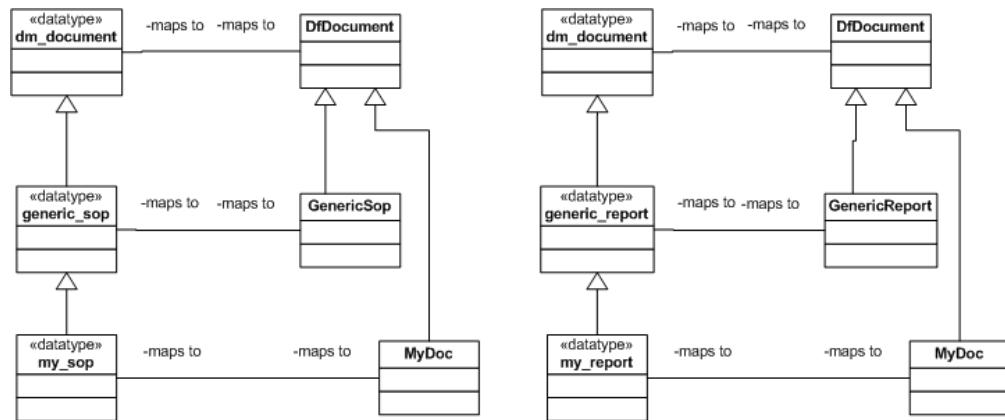


Figure 7-6: Design-time dynamic inheritance with TBO reuse

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.

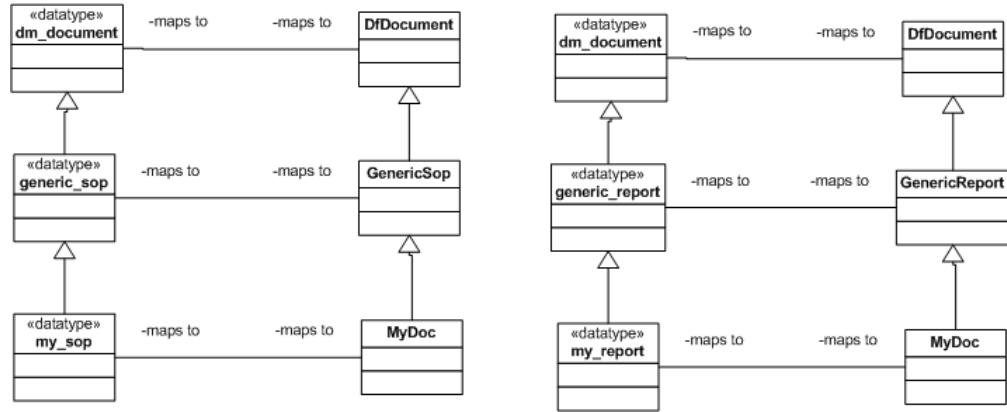


Figure 7-7: Runtime dynamic inheritance with TBO reuse

7.4.8 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. The following are the methods to override when implementing TBOs:

7.4.8.1 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 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



Note: If you are using OpenText™ Documentum™ Content Management Accelerated Content Services or Branch Office Caching Services, the doSetFile method is not used for content transfer. If your TBO relies on this method, the content transfer does not happen via OpenText Documentum Content Management (CM) Accelerated Content Services or Branch Office Caching Services, but the content will be brought to the application server, and doSetFile will be executed.

- 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

7.4.8.2 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

7.4.8.3 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

7.4.8.4 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

7.4.8.5 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

7.5 Calling TBOs and SBOs

This section describes special considerations for using TBOs and SBOs.

7.5.1 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 Foundation Java API session.

7.5.2 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.

[“Maintaining state in a session manager” on page 35](#) provides information about the substantial costs of using the `setSessionManager` method.

7.5.3 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. Foundation Java API 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 Foundation Java API session but keep an instance of the TBO.
- Store the TBO in the SBO state.

[“Maintaining state in a session manager” on page 35](#) provides information about the substantial costs of using the `setSessionManager` method.

7.6 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.

7.6.1 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 7-1: 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; }
```



7.6.2 TutorialSBO

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

Example 7-2: 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);
} }
```



7.6.3 ITutorialTBO

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

Example 7-3: ITutorialTBO.java

```
package com.documentum.tutorial import com.documentum.fc.client.IDfBusinessObject;
public interface ITutorialTBO extends IDfBusinessObject {    /**
 * No code required (just extends IDfBusinessObject)    */ }
```



7.6.4 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 7-4: 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 tutSBObj =
(ITutorialSBO)client.newService(ITutorialSBO.class.getName(),sMgr);
tutSBObj.setFlavorSBO(this, myFlavor); } } //end class

```



7.6.5 Deploying the SBO and TBO

You use Documentum Composer to package your BOF modules. The *OpenText Documentum Content Management - Composer BOF Development Tutorial (EDCPC250400-PTL)* contains information on creating and deploying a simple BOF module.

7.7 Calling Foundation SOAP API, JAXB API, or other components from a BOF module

To call Foundation SOAP API, JAXB API, or any third-party component which relies on the ThreadContext ClassLoader to load classes from within a BOF module, you will need to set the module class loader to the ThreadContext class loader. The following code saves the current ThreadContext ClassLoader, sets up ThreadContext ClassLoader as the current class ClassLoader in a try block, calls Foundation SOAP API (or other component), and uses the finally block to reset the ThreadContext ClassLoader to the original ThreadContext ClassLoader:

```
ClassLoader oldContextClassLoader = Thread.currentThread().getContextClassLoader();
try { ClassLoader classClassLoader = this.getClass().getClassLoader();
    Thread.currentThread().setContextClassLoader(classClassLoader);
    // call dfs service here } catch (ServiceException e) { // catch
block } finally {
    Thread.currentThread().setContextClassLoader(oldContextClassLoader);
}
```

7.8 Aspects

Aspects are a mechanism for adding behavior and/or attributes to a OpenText Documentum CM 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). Similar to 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 Foundation Java API runtime.

7.8.1 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.

7.8.2 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.

If you attach an aspect that appends a new attribute when creating an object, you must have the Write permission or higher. By contrast, the creation of an object that does not have an aspect attached only requires the Version permission.

7.8.3 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 using the Documentum Composer tool. For more information about Documentum Composer, see *OpenText Documentum Content Management - Composer User Guide (EDCPCL250400-UGD)*.

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.

7.8.3.1 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 7-5: 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;
}
```



7.8.3.2 Creating the aspect class

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

Example 7-6: CustomerServiceAspect.java

```
package dfctutorialenvironment; import com.documentum.fc.client.DfDocument;
import com.documentum.fc.common.DfException; import com.documentum.fc.common.IDfTime;
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
months); cal.add(Calendar.MONTH, months);
setExpirationDate(cal.getTime()); }
catch (DfException e) { e.printStackTrace(); }
}
}
```

```

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); }
}

```



7.8.3.3 Deploy the customer service aspect

The *Documentum Foundation Classes 6.0 Release Notes* provides details on deploying aspect modules.

7.8.3.4 TestCustomerServiceAspect

After you have compiled and deployed your aspect classes and defined the aspect on the server, you can use the class [Example 7-7, “TestCustomerServiceAspect.java” on page 145](#) 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 [“Creating a test application” on page 37](#).

Example 7-7: 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.";
} } }

```



7.8.4 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 7-8: 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 7-9: 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(); } }
```



7.8.5 Using DQL with aspects

After an aspect has been defined in the repository, you can use Documentum Query Language (DQL) 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 (OpenText Documentum CM 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, and so on) are not supported in the OpenText Documentum CM 6 release.

7.8.5.1 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
```

7.8.5.2 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.

When you attach a default aspect to a type, all the subtypes of that type will also have that aspect attached. If you attach a default aspect to dm_sysobject, that aspect will be inherited by all the dm_sysobject subtypes. However, some of the dm_sysobject subtypes are implemented as final types and will throw a runtime error, "java.lang.VerifyError: Cannot inherit from final class". Do not attach default aspects to dm_sysobject. Instead, attach it to the specific types that will use the aspect. Alternatively, create a subtype of dm_sysobject, attach the default aspect to that, and derive your other subtypes from that subtype of dm_sysobject, rather than dm_sysobject itself.

7.8.5.3 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 similar to a normal attribute, wherever legally permitted by the syntax.

7.8.6 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
```

7.8.7 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 8

Working with virtual documents

This chapter covers some of the basics of working programmatically with virtual documents.

8.1 Understanding virtual documents

A virtual document is a data-driven construction that allows you to treat more than one system object as a single document. The components, or nodes, of the document remain their individual identities, but are related to one another using dmr_containment objects to link a parent node to child nodes in a hierarchical structure.

Complexity is composed of simple things. While the structure of a virtual document can be quite complex taken as a whole, the relationship of any one node to its parent is straightforward and easy to follow.

Virtual documents are commonly created in three ways. When you import a document with Microsoft OLE links, it can be added as a parent, with the linked objects imported as child nodes. XML documents can be automatically converted to virtual documents on import using an XML application. When working with documents created automatically from OLE or XML documents, you are able to check out and edit individual nodes of a virtual document. When you check out the parent document, the document is reassembled into one cohesive unit.

The third way is to manually convert a document to a virtual document (programmatically or through an interface) and add arbitrary repository objects as children. The virtual document imposes a hierarchical structure that can help to manage the components as a single unit (for example, you can check out a parent document and all descendants in a single operation).

Virtual documents can be linked to what is always the CURRENT version of a child node, or they can be linked to a specific version by its version label (either the system-supplied version number or a user-supplied symbolic version label). For example, you could label a version of each document in a publication August and create a virtual document linking to the versions of the nodes that have that label. This way, regardless of the updates made to the document for September, the August virtual document would point to the nodes used at the time of its publication.

8.2 Setting version labels

To link to a particular version of a document, you first need to set a version label. Setting a version label requires that you check out the document, then check it in again with the version label you want to use. You can set more than one label, using a single, comma-delimited String value. If you want the document to also be the CURRENT version, you must explicitly include that label, as well.

To add a Set Version Labels button to the DFC Base Tutorial Frame

1. Create a JTextField control named jTextField_versionLabel.
2. Create a JButton control named jButton_setVersionLabel.
3. Update the **TutorialCheckIn** class to set the version label on check in.
4. Create the class **TutorialSetVersion**.
5. Add a button handler method for **Set Version Label**.

Example 8-1: Updated TutorialCheckIn class

```
package com.emc.tutorial;

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.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
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 checkinLabelExample(
        IDfSessionManager sessionManager,
        String repositoryName,
        String docId,
        // Add a variable for incoming version labels, entered as
        // a single comma-delimited string.
        String versionLabels
    )
    {
        IDfSession mySession = null;
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='"
                    + docId + "'";
            );

            // Instantiate an object from the ID.

            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
        }
    }
}
```

```

// Instantiate a client.
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
// ( version + 1 )
cio.setCheckinVersion(IDfCheckinOperation.NEXT_MAJOR);

// Set one or more comma-delimited labels for this version.
// When updating to the next major version, you need to explicitly
// set the version label for the new object to "CURRENT".
cio.setVersionLabels(versionLabels);

// 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.";
}
finally
{
    sessionManager.release(mySession);
}
}
}

```



Example 8-2: The TutorialSetVersion class

```

package com.emc.tutorial;

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

public class TutorialSetVersion
{
    public TutorialSetVersion()
    {

        public String setVersion(
            IDfSessionManager sessionManager,
            String repositoryName,
            String documentIdString,
            String versionLabels

```

```

        }
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Check out the document.
            TutorialCheckOut tco = new TutorialCheckOut();
            tco.checkoutExample(
                sessionManager,
                repositoryName,
                documentIdString
            );

            // Check in the document, specifying the version labels.
            TutorialCheckIn tci = new TutorialCheckIn();
            sb.append( "\n" +
                tci.checkinLabelExample(
                    sessionManager,
                    repositoryName,
                    documentIdString,
                    versionLabels
                )
            );
            return "Success!\n*****\n\n" + sb.toString();
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return ("Failure!\n*****\n\n" + sb.toString());
        }
        finally
        {
            sessionManager.release(mySession);
        }
    }
}

```



Example 8-3: Handler for the Set Version Label button

```

private void jButton_setVersionLabelActionPerformed(ActionEvent e)
{
    if (_parentId == null)
    {
        jLabel_messages.setText("Please reset the virtual document parent.");
    }
    else
    {
        String repositoryName = jTextField_repositoryName.getText();
        String childIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        String versionLabel = jTextField_versionLabel.getText();
        TutorialSetVersion tsv =
            new TutorialSetVersion();
        jTextArea_results.setText(
            tsv.setVersion(
                m_sessionManager,
                repositoryName,
                childIdString,
                versionLabel
            )
        );
        initDirectory();
    }
}

```

```
    getDirectory();
}
```



8.3 Getting version labels

You can query to get the labels from a document in the system in preparation for linking to a specific version of the document. There is, in fact, an `IDfVersionLabels` interface specially designed to let you query and manipulate the labels on a system object.

To add a Get Version Labels button to the DFC Base Tutorial Frame

1. If you have not done so already, create a `JTextField` control named `jTextField_versionLabel`.
2. Create a `JButton` control named `jButton_getVersionLabel`.
3. Create the class `TutorialGetVersion`.
4. Add a button handler method for `Get Version Label`.



Example 8-4: The `TutorialGetVersion` class

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfVersionLabels;
import com.documentum.fc.common.IDfId;

public class TutorialGetVersion
{
    public TutorialGetVersion()
    {
    }

    public String getVersion(
        IDfSessionManager sessionManager,
        String repositoryName,
        String documentIdString
    )
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");
        try
        {
            mySession = sessionManager.getSession(repositoryName);

// Get the object ID based on the object ID string.
            IDfId idObj =
                mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + documentIdString + "'"
                );
        }

// Instantiate an object from the ID.
        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
```

```

        IDfVersionLabels vl = sysObj.getVersionLabels();
        for (int i=0; i<vl.getVersionLabelCount();i++)
        {
            if (i==0) {
                sb.append(vl.getVersionLabel(i));
            }
            else
            {
                sb.append ("," + vl.getVersionLabel(i));
            }
        }
        return sb.toString();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        return ("Failed to get version labels for selected object.");
    }
    finally
    {
        sessionManager.release(mySession);
    }
}
}

```



Example 8-5: Handler for the Get Version Label button

```

private void jButton_getVersionLabelActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String documentIdString =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    TutorialGetVersion tgv = new TutorialGetVersion();
    jTextField_versionLabel.setText(
        tgv.getVersion(
            m_sessionManager,
            repositoryName,
            documentIdString
        )
    );
    initDirectory();
    getDirectory();
}

```



8.4 Creating a virtual document

Creating a virtual document is a fairly straightforward exercise. If you select any system object (with the exception of folders and cabinets), you can get use the DfClient.asVirtualDocument() method to indicate that you want to convert the object into a virtual document. It is not persisted as a virtual document in the repository, though, until you add one or more child nodes and save the document. You can, and should, set the r_is_virtual_document property on the file to TRUE as a signal to other applications that the document should be treated as a virtual document, then set the value to FALSE if the children are subsequently removed.

In this example, we will add a field for the parent name, and a button that populates the field and captures the ID of the selected item in a global variable. You can then select another document to add it as a child node.

By default, the child node is added as the first node of the virtual document. If you add another child node it becomes the first node, and the existing node moves down to become the second child node. If you want to control the placement of the node in the virtual document hierarchy, you have the option of passing a sibling node as an argument to the `addNode()` method. The new node is placed after the sibling node in the virtual document hierarchy. We will add a field to capture the name of an optional virtual document sibling.

The only new class created for this example is the `TutorialAddVirtualDocumentNode` class. The remaining button handlers (`Set Virtual Document Parent`, `Set Preceding Sibling`, `Clear Preceding Sibling`) capture supporting information and set member variables for use by the `Add Virtual Document Node` button handler.

To add an Add Virtual Document Node button to the DFC Base Tutorial Frame:

1. Create a `JTextField` control named `jTextField_virtualDocumentParent`.
2. Create a `JButton` control named `jButton_setVirtualDocumentParent`.
3. Create a `JTextField` control named `jTextField_precedingSibling`.
4. Create a `JButton` control named `jButton_setPrecedingSibling`.
5. Create a `JButton` control named `jButton_clearSibling`.
6. Create the class `TutorialAddVirtualDocumentNode`.
7. Add a button handler method for `Set Virtual Document Parent`.
8. Add a button handler method for `Set Preceding Sibling`.
9. Add a button handler method for `Clear Sibling`.
10. Add a button handler method for `Add Virtual Document Node`.

► Example 8-6: Handler for the Set Virtual Document Parent button

```
private void setVirtualDocumentParent()
{
    // Store the parent ID in a global variable for use by other operations.
    m_parentId = m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    jTextField_virtualDocumentParent.setText(
        jTextField_cwd.getText() + "/" + list_id.getSelectedItem());
    jTextArea_results.setText(m_parentId + "\n" + list_id.getSelectedItem());
}
```



► Example 8-7: Handler for the Set Preceding Sibling button

```
private void jButton_setPrecedingSiblingActionPerformed(ActionEvent e)
{
    m_siblingId = m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
    jTextField_precedingSibling.setText(
        list_id.getSelectedItem());
}
```

```

        jTextArea_results.setText(m_siblingId + "\n" + list_id.getSelectedItem());
    }
}

```



Example 8-8: Handler for the Clear Sibling button

```

private void jButton_clearSiblingActionPerformed(ActionEvent e)
{
    m_siblingId = "";
    jTextField_precedingSibling.setText("");
}

```



Example 8-9: Handler for the Add Virtual Document Node button

```

private void jButton_addVirtualDocumentNodeActionPerformed(ActionEvent e)
{
    if (m_parentId == null)
    {
        jLabel_messages.setText("Please reset the virtual document parent.");
    }
    else
    {
        String repositoryName = jTextField_repositoryName.getText();
        String childIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        String bindingLabel = jTextField_versionLabel.getText();
        TutorialAddVirtualDocumentNode tavdn =
            new TutorialAddVirtualDocumentNode();
        jTextArea_results.setText(
            tavdn.addNode(
                m_sessionManager,
                repositoryName,
                m_parentId,
                childIdString,
                m_siblingId,
                bindingLabel
            )
        );
        initDirectory();
        getDirectory();
        m_parentId = null;
    }
}

```



Example 8-10: The TutorialAddVirtualDocumentNode class

```

package com.emc.tutorial;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.client.IDfVirtualDocumentNode;
import com.documentum.fc.common.IDfId;

public class TutorialAddVirtualDocumentNode
{
    public TutorialAddVirtualDocumentNode()
    {

```

```

    {
}
public String addNode(
    IDfSessionManager sessionManager,
    String repositoryName,
    String parentIdString,
    String childIdString,
    String siblingIdString,
    String versionLabel
)
{
    IDfSession mySession = null;
    StringBuffer sb = new StringBuffer("");
    try
    {
        mySession = sessionManager.getSession(repositoryName);

        // Check out the parent object
        TutorialCheckOut tco = new TutorialCheckOut();
        tco.checkoutExample(
            sessionManager,
            repositoryName,
            parentIdString
        );

        IDfClientX clientx = new DfClientX();

        //Instantiate the parent object.
        IDfVirtualDocument vDoc = null;
        IDfId parentIdObject = clientx.getId(parentIdString);
        IDfSysObject sysObj =
            (IDfSysObject) mySession.getObject(parentIdObject);
        if (versionLabel.equals("")) versionLabel = null;
        if (sysObj != null)
        {

            // Instantiate the parent as a virtual document.
            vDoc = sysObj.asVirtualDocument(null, false);

            // Instantiate the root node of the virtual document.
            IDfVirtualDocumentNode root = vDoc.getRootNode();

            // Create an ID object for the child node.
            IDfId childId = mySession.getIdByQualification(
                "dm_sysobject where r_object_id='" + childIdString + "'"
            );

            //Instantiate the child as a sysobject.
            IDfSysObject childObj =
                (IDfSysObject) mySession.getObject(childId);

            // Instantiate the sibling object (if not null) as a virtual document node.
            IDfVirtualDocumentNode siblingNode = null;
            if (!siblingIdString.equals(null) & !siblingIdString.equals("")){
                IDfId siblingId = mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + siblingIdString + "'"
                );
                IDfSysObject siblingObj =
                    (IDfSysObject) mySession.getObject(siblingId);
                String siblingChronicleId = siblingObj.getChronicleId().toString();
                siblingNode = vDoc.find(root, siblingChronicleId, "i_chronicle_id", 0);
            }

            // Add the child to the virtual document.
            IDfVirtualDocumentNode childVDNode = vDoc.addNode(
                root,
                siblingNode,
                childObj.getChronicleId(),
                versionLabel,
                false,
            );
        }
    }
}

```

```

        false
    );
}

// Check in the parent document.
TutorialCheckIn tci = new TutorialCheckIn();
sb.append( "\n" +
    tci.checkinExample(
        sessionManager,
        repositoryName,
        parentIdString
    )
);
return "Success!\n*****\n\n" + sb.toString();
}
catch (Exception ex)
{
    ex.printStackTrace();
    return ("Failure!\n*****\n\n" + sb.toString());
}
finally
{
    sessionManager.release(mySession);
}
}
}

```



8.5 Traversing the virtual document structure

After you have created a virtual document, you can use a recursive routine to traverse the hierarchical structure and display information about each node.

One side note: in the past, there have been other methods requiring several statements in order to get the underlying system object represented by a virtual document node. Going forward, you should use the method `IDfVirtualDocumentChild.getSelectedObject()` to get the object represented by a virtual document node.

To add a Traverse Virtual Document button to the DFC Base Tutorial Frame:

1. Create a jButton control named `jButton_traverseVirtualDocument`.
2. Create the `TutorialTraverseVirtualDocument` class.
3. Create the handler for the `Traverse Virtual Document` button.

Example 8-11: The `TutorialTraverseVirtualDocument` class

```

package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.client.IDfVirtualDocumentNode;
import com.documentum.fc.common.IDfId;

public class TutorialTraverseVirtualDocument
{

```

```

public TutorialTraverseVirtualDocument()
{
}
public String traverseVirtualDocument(
    IDfSessionManager sessionManager,
    String repositoryName,
    String docId
)
{
    IDfSession mySession = null;
    StringBuffer results = new StringBuffer("");
    try
    {
        mySession = sessionManager.getSession(repositoryName);

        // Get the object ID based on the object ID string.
        IDfId idObj =
            mySession.getIdByQualification(
                "dm_sysobject where r_object_id='"
                + docId + "'");

        // Instantiate an object from the ID.
        IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);
        IDfVirtualDocument parent = sysObj.asVirtualDocument(null, false);
        IDfVirtualDocumentNode node = parent.getRootNode();

        // Capture useful information about the root node.
        results.append( "\nNode Name: " + sysObj.getObjectName() +
            "\nVersion: " + sysObj.getVersionLabel(0) );

        int childCount = node.getChildCount();
        IDfVirtualDocumentNode child = null;
        for( int i = 0; i < childCount; ++i )
        {
            child = node.getChild( i );

            // If the child is also a virtual document, make a nested
            // call to this method.
            if( child.getSelectedObject().isVirtualDocument() )
            {
                sysObj = child.getSelectedObject();
                results.append(
                    traverseVirtualDocument(
                        sessionManager,
                        repositoryName,
                        sysObj.getObjectId().toString()
                    )
                );
            }
            else
            {
                // Otherwise, capture interesting information about the child node.
                sysObj = child.getSelectedObject();
                results.append( "\nNode Name: " + sysObj.getObjectName() +
                    "\n Version: " + sysObj.getVersionLabel(0) +
                    "\n VDM Number: " + child.getVDMNumber() +
                    "\n Binding Label: " + child.getBinding());
            }
        }
        return results.toString();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
        return ("Traversal failed.");
    }
    finally
}

```

```

        {
            sessionManager.release(mySession);
        }
    }
}

```



Example 8-12: Handler for the Traverse Virtual Document button

```

private void jButton_traverseActionPerformed(ActionEvent e)
{
    String repositoryName = jTextField_repositoryName.getText();
    String docId =
        m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();

    TutorialTraverseVirtualDocument ttvd =
        new TutorialTraverseVirtualDocument();
    jTextArea_results.setText(
        ttvd.traverseVirtualDocument(
            m_sessionManager,
            repositoryName,
            docId
        )
    );
}

```



8.6 Binding to a version label

In practical use, you may want to bind to a particular version label after the virtual document has been created. To do this, you check out the parent node, instantiate the child node based on its chronicle ID, use the `IDfVirtualDocumentNode.setBinding(String bindingLabel)` method to assign the binding value, then check in the parent node.

To add a Set Binding button to the DFC Base Tutorial Frame:

1. Create a JButton control named `jButton_setBinding`.
2. Create the `TutorialSetBinding` class.
3. Create the handler for the `Set Binding` button.

Example 8-13: The TutorialSetBinding class

```

package com.emc.tutorial;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.client.IDfVirtualDocumentNode;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfCheckinNode;

public class TutorialSetBinding

```

```

{
    public TutorialSetBinding()
    {
    }
    public String setBinding (
        IDfSessionManager sessionManager,
        String repositoryName,
        String parentIdString,
        String childIdString,
        String bindingLabel // An existing label already assigned to the child node.
    )
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");
        try
        {

            // Start by checking out the parent document.
            mySession = sessionManager.getSession(repositoryName);
            TutorialCheckOut tco = new TutorialCheckOut();
            tco.checkoutExample(
                sessionManager,
                repositoryName,
                parentIdString
            );
            IDfClientX clientx = new DfClientX();
            IDfVirtualDocument vDoc = null;

            // Use the parent ID to instantiate a sysobject.
            IDfId parentIdObject = clientx.getId(parentIdString);
            IDfSysObject sysObj =
                (IDfSysObject) mySession.getObject(parentIdObject);

            if (sysObj != null)
            {

                // Get a virtual document instance of the parent.
                vDoc = sysObj.asVirtualDocument(null, false);
                IDfVirtualDocumentNode root = vDoc.getRootNode();

                // Get an instance of the child document.
                IDfId childId = mySession.getIdByQualification(
                    "dm_sysobject where r_object_id='" + childIdString + "'"
                );
                IDfSysObject childObj =
                    (IDfSysObject) mySession.getObject(childId);

                // Use the current instance to get the chronicle ID of the child.
                String childChronId = childObj.getChronicleId().toString();

                // Instantiate a virtual document node based on the child's chronicle ID.
                IDfVirtualDocumentNode childNode =
                    vDoc.find(root, childChronId, "i_chronicle_id", 0);

                // Set the child's binding to the label provided.
                childNode.setBinding(bindingLabel);
            }

            // Check in the parent document.
            TutorialCheckIn tci = new TutorialCheckIn();
            sb.append( "\n" +
                tci.checkinExample(
                    sessionManager,
                    repositoryName,
                    parentIdString
                )
            );
        }
        return "Success!\n*****\n\n" + sb.toString();
    }
}

```

```

        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            return ("Failure!\n*****\n\n" + sb.toString());
        }
        finally
        {
            sessionManager.release(mySession);
        }
    }
}

```



Example 8-14: Handler for the Set Binding button

```

private void jButton_setBindingActionPerformed(ActionEvent e)
{
    if (_parentId == null)
    {
        jLabel_messages.setText("Please reset the virtual document parent.");
    }
    else
    {
        String repositoryName = jTextField_repositoryName.getText();
        String childIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        String bindingLabel = jTextField_versionLabel.getText();
        TutorialSetBinding tsb = new TutorialSetBinding();
        jTextArea_results.setText(
            tsb.setBinding(
                m_sessionManager,
                repositoryName,
                _parentId,
                childIdString,
                bindingLabel
            )
        );
        initDirectory();
        getDirectory();
    }
}

```



8.7 Clearing a version label binding

To clear a binding based on version label, you can set the binding to the value CURRENT, which will restore the default behavior of always including the CURRENT version of the child object as a virtual document node.

To add a Set Binding button to the DFC Base Tutorial Frame:

1. Create a jButton control named jButton_clearBinding.
2. If you have not done so already, create the `TutorialSetBinding` class.
3. Create the handler for the `Clear Binding` button.

➡ Example 8-15: Handler for the Clear Binding button

```
private void jButton_clearBindingActionPerformed(ActionEvent e)
{
    if (_parentId == null)
    {
        jLabel_messages.setText("Please reset the virtual document parent.");
    }
    else
    {
        String repositoryName = jTextField_repositoryName.getText();
        String childIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        String bindingLabel = "CURRENT";
        TutorialSetBinding tsb = new TutorialSetBinding();
        jTextArea_results.setText(
            tsb.setBinding(
                _sessionManager,
                repositoryName,
                _parentId,
                childIdString,
                bindingLabel
            )
        );
        initDirectory();
        getDirectory();
    }
    // Nullify the parent ID global variable, as a new version of the
    // parent document has been saved, and the existing value is no
    // longer current.
    _parentId = null;
}
```



8.8 Removing a virtual document child

Removing a virtual document child is another operation that takes a little extra wangling, because you need to remove the child node based on its chronicle ID. You code needs to check out the parent document, instantiate it as a virtual document, locate the child node, get its ID, use that to find the child node's chronicle ID, instantiate the child virtual document node using the chronicle ID, pass the node as an argument to the `IDfVirtualDocument.remove(IDfVirtualDocumentNode childNode)` method, then check in the parent document.

You should also test to see if you are removing the last node from the virtual document (if the child count is now 0). If so, use the method `IDfSysObject.setIsVirtualDocument()` to set the property to FALSE.

To add a Remove Virtual Document Child button to the DFC Base Tutorial Frame:

1. Create a jButton control named `jButton_removeVirtualDocumentChild`.
2. Create the `TutorialRemoveVirtualDocumentNode` class.
3. Create the handler for the Remove Virtual Document Child button.

➡ **Example 8-16: The TutorialRemoveVirtualDocumentNode class**

```
package com.emc.tutorial;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfVirtualDocument;
import com.documentum.fc.client.IDfVirtualDocumentNode;
import com.documentum.fc.common.IDfId;

public class TutorialRemoveVirtualDocumentNode
{
    public TutorialRemoveVirtualDocumentNode()
    {
    }

    public String removeNode(
        IDfSessionManager sessionManager,
        String repositoryName,
        String parentIdString,
        String childIdString
    )
    {
        IDfSession mySession = null;
        StringBuffer sb = new StringBuffer("");
        try
        {
            mySession = sessionManager.getSession(repositoryName);

            // Check out the parent document.
            TutorialCheckOut tco = new TutorialCheckOut();
            tco.checkoutExample(
                sessionManager,
                repositoryName,
                parentIdString
            );
        }
    }
}
```

```

IDfClientX clientx = new DfClientX();

// Create an empty virtual document object.
IDfVirtualDocument vDoc = null;

// Get the parent object
IDfId parentIdObject = clientx.getId(parentIdString);
IDfSysObject sysObj =
    (IDfSysObject) mySession.getObject(parentIdObject);

if (sysObj != null)
{

// Get the parent document as a virtual document.
vDoc = sysObj.asVirtualDocument(null, false);

// Get the root node of the virtual document.
IDfVirtualDocumentNode root = vDoc.getRootNode();

// Create an ID object based on the child ID string.
IDfId childId = mySession.getIdByQualification(
    "dm_sysobject where r_object_id='" + childIdString + "'";
);
// Instantiate the child object.
IDfSysObject childObj =
    (IDfSysObject) mySession.getObject(childId);

// Get the child object's chronicle ID
String childChronId = childObj.getChronicleId().toString();

// Get the child object as a virtual document node.
IDfVirtualDocumentNode childNode =
    vDoc.find(root, childChronId, "i_chronicle_id", 0);
sb.append("Child ID: " + childId.toString());
sb.append("\nChild Chron ID: " + childChronId);
sb.append("\nChild Node: " + childNode.toString());

// And here is the actual command that removes the node.
vDoc.removeNode(childNode);

// If you have removed the last virtual document node,
// set the r_is_virtual_document property to FALSE.
if (root.getChildCount() == 0)
{
    sysObj.setIsVirtualDocument(false);
}
}

// Check in the parent virtual document to save the change.
TutorialCheckIn tci = new TutorialCheckIn();
sb.append( "\n" +
tci.checkinExample(
    sessionManager,
    repositoryName,
    parentIdString
)
);
return "Success!\n*****\n" + sb.toString();
}
catch (Exception ex)
{
    ex.printStackTrace();
    return ("Failure!\n*****\n" + sb.toString());
}
finally
{
    sessionManager.release(mySession);
}

```

```
    }
```



► Example 8-17: Handler for the Remove Virtual Document Child button

```
private void jButton_removeVirtualDocumentChildActionPerformed(ActionEvent e)
{
    if (m_parentId == null)
    {
        jLabel_messages.setText("Please reset the parent field.");
    }
    else
    {
        String repositoryName = jTextField_repositoryName.getText();
        String childIdString =
            m_fileIDs.elementAt(list_id.getSelectedIndex()).toString();
        TutorialRemoveVirtualDocumentNode trvdn =
            new TutorialRemoveVirtualDocumentNode();
        jTextArea_results.setText(
            trvdn.removeNode(
                m_sessionManager,
                repositoryName,
                m_parentId,
                childIdString
            )
        );
        initDirectory();
        getDirectory();
    }
}
```



Chapter 9

Support for other OpenText Documentum CM functionality

Foundation Java API is the principal low level interface to all OpenText Documentum CM functionality and there are many Foundation Java API interfaces. They provide access to features that other documentation covers in more detail. For example, the *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)* describes virtual documents and access control. The Foundation Java API Javadocs provide the additional information necessary to enable you to take advantage of those features. Similarly, the Documentum Federated Search Services product includes extensive capabilities for searching repositories. The Foundation Java API Javadocs provide information about how to use Foundation Java API to access that functionality.

9.1 Security services

Documentum CM Server provides a variety of security features. From the Foundation Java API standpoint, they fall into the following categories:

- User authentication

For more information about user authentication, see “[Sessions and session managers](#)” on page 23.

- Object permissions

For more information about the Foundation Java API Javadocs for IDfACL, IDfPermit, and other interfaces, see *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)*.

Foundation Java API 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. The Javadocs provide more details.

9.2 XML

“Working with document operations” on page 75 provides some information about working with XML. Foundation Java API provides substantial support for the OpenText Documentum CM XML capabilities.

9.3 Virtual documents

“Working with document operations” on page 75 provides some information about working with virtual documents. For more information about the IDfVirtualDocument interface, see *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)* and Foundation Java API Javadocs.

9.4 Workflows

For information about the concepts underlying workflows, see *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)*. Foundation Java API 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 Foundation Java API from them. These methods run on the method server, an application server that resides on the Documentum CM Server machine and is dedicated to running Documentum CM Server methods. The code for these methods resides in the repository's registry as modules. “Modules and registries” on page 113 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.

9.5 Document lifecycles

For information about lifecycles, see *OpenText Documentum Content Management - Server Fundamentals Guide (EDCCS250400-GGD)*. There are no Foundation Java API interfaces for constructing document lifecycles. 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 Documentum CM 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](#)” on page 113) as modules.

9.6 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. Documentum CM Server does not enforce constraints defined in the data dictionary.

In earlier versions of Foundation Java API these constraints were exclusively Docbasic expressions. Since Foundation Java API 5.3, you can provide Java translations of the constraint expressions. If a Java version of a constraint exists, Foundation Java API 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, Foundation Java API 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 Documentum CM 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.

9.7 Unified Login

If you are using Unified Login with your Foundation Java API client there is an important requirement for authentication to work. Unified Login relies on the Windows SID authentication mechanism. Foundation Java API forms an SID based on the login info from the client machine and sends it to the server. The server authenticates it using the Windows Domain Controller (DC). If the DC does not recognize the client SID, then authentication will fail. The client SID should be recognized by the DC and the user should be registered using the same domain as the repository. So the requirement is that client and repository machine should be in the same workgroup or domain.

9.8 Search service

The Foundation Java API 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 repositories and, in conjunction with the Documentum Federated Search 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.

9.9 WORM support for filestores

WORM support is available for filestores.

Run the following command to enable or disable WORM for a filestore:

```
EXECUTE set_storage_state with store='<store>' ,worm=[true | false]
```

Any new content that is created on a file store with the WORM feature enabled is saved with the read-only flag on the filesystem. When the read-only file attribute is enabled, the filesystem that is WORM-enabled must keep this file under infinite retention. Any edit operation on the content at the filesystem level is denied.

Objects with content stored on the WORM-enabled filestore cannot be deleted or destroyed.

You can perform the MIGRATE_CONTENT operation on worm content. However, the remove_original parameter is ignored and is considered as false.