

Unit Testing Documentum Foundation Classes (DFC) Code



Steve McMichael
Blue Fish Development Group

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

About Blue Fish

Focused on ECM

We help companies manage and extract value from their intellectual assets

Experience

In our 8th year of solving difficult content management problems

Expertise

Expertise across the Documentum platform; host the dm_developer community

Client Centric

"Be Smart, Be Nice, Get Things Done"



dm_developer

An Online Resource for Documentum Developers

Blue Fish Practices

Web Content Management

- Focus on empowering business users
- Attention to ease of use, standardization, and consistency



Custom ECM Solutions and Integrations

- Focus on unleashing clients' unique competitive advantages
- Attention to supportability and upgradeability



Content Migrations

- Focus on retaining value of intellectual assets
- Attention to migration of regulatory docs in validated environments



Information Access

- Focus on helping clients access and analyze their most critical information
- Attention to top-line business improvements



Blue Fish ECM Software

Web Content Management

- Focus on empowering business users
- Attention to ease of use, standardization, and consistency

Navigation Manager for Web Publisher

- Enables business users to manage website navigation

Custom ECM Solutions and Integrations

- Focus on unleashing clients' unique competitive advantages
- Attention to supportability and upgradeability

Bedrock

- Reusable code library
- Enables rapid, high-quality solutions

Content Migrations

- Focus on retaining value of intellectual assets
- Attention to migration of regulatory docs in validated environments

DIXI and Migration Workbench

- For full spectrum of migration challenges

Information Access

- Focus on helping clients access and analyze their most critical information
- Attention to top-line business improvements

Documentum Adapter for Endeca

- Enterprise and inter-application information access

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

Why Unit Test?

- Automated unit tests:
 - Are crucial for developing enterprise-quality code.
 - Give us confidence that the code we write is working according to the specified requirements.
 - Allow us to refactor our code and know that the code still works.
 - Remove the human variable from the testing equation, which opens the door to continuous integration.
 - Are a key ingredient to Agile and test-driven development methodologies.

Contents

- About Blue Fish
- Why Unit Test?
- **The Tools**
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

The Tools

- JUnit for creating tests
 - de facto industry standard for writing unit tests
 - Easy to use
 - Has a variety of “runners” for running your tests
- Ant for running tests
 - de facto industry standard for building Java programs
- CruiseControl for running nightly builds
 - A continuous build ensures execution of tests

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

The Pain Points

- Unit testing is frequently not estimated or accounted for.
- Project schedule pressures often interfere with developers writing effective unit tests.
- A DFC test harness must have some mechanism for managing user logins. Implementing such a mechanism is not difficult, but it is a sufficient barrier to entry for developers when it comes time to write tests, especially if the tests have been left to the end of the project.
- Setting up and tearing down the test state can be a very difficult task. JUnit provides general hooks that allow a test case to tap into the overall setup and teardown, but those hooks have no explicit knowledge of what is being tested.
- Automated tests require a process change, and change is not easy. If you are having a problem getting your unit tests written, it may be because you have never written automated unit tests before. As with most things, “practice makes perfect.”

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- **Managing Logins**
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

Managing Logins

- DFC unit tests need to authenticate multiple users against multiple repositories, especially when the code being tested has security requirements.
- Tips:
 - Do not hardcode login credentials in your tests. This is brittle, especially when you need to test against different repositories, such as “dev1”, “dev2”, or “integration”.
 - Prefer logical usernames, which are mapped to actual Documentum users. That way, you can map the same logical name to different users in different repositories.
 - Define logins in some external configuration that is used by the testing framework. This should contain the user credentials, the repository, and the logical name.
 - Be flexible in how the configuration is resolved. Consider using a System property that can point to a file, a co-located resource, or a URL. That allows testers to specify the logins to Ant using build properties.

Managing Logins - IAuthenticator

- Pluggable interface
- Manufactures *IDfSessionManager* objects, which can be used to perform DFC actions

```
public interface IAuthenticator {  
    public IDfSessionManager getSessionManager(String logicalName)  
        throws DfException;  
  
    public IDfLoginInfo getLoginInfo(String logicalName)  
        throws DfException;  
  
    public String getDocbase(String logicalName)  
        throws DfException;  
}
```

Managing Logins - XmlAuthenticator

The XmlAuthenticator is an IAuthenticator that digests XML. Here's a sample:

```
<logins>
  <login logicalName="admin"
    userOSName="dadmin"
    password="password1"
    repository="test_53" />
  <login logicalName="read-only"
    userOSName="readonly"
    password="password2"
    repository="test_53" />
  <login logicalName="read-write"
    userOSName="readwrite"
    password="password3"
    repository="test_53" />
</logins>
```

Managing Logins – Integration Into TestCase

- Add a *getAuthenticator()* method to your *TestCase*
 - `public IAuthenticator getAuthenticator() { ... }`
 - Subclasses may override to return different implementation
- Add a *getSession()* method to your *TestCase*
 - `public IDfSession getSession(String logicalUser) { ... }`
 - Provides support for “application security”
 - One of the most commonly called test methods

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- **Setting Up State**
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

Setting Up State

- Use transactions when setting up state, which makes tear down easy.
 - Provides isolation, allowing tests to run concurrently.
 - The tear down for the test simply rolls back the transaction.
 - This is not always possible. For example, tests that use multiple session managers cannot share the same transaction.
 - Transactions are controlled via IDfSessionManagers, so this is handled by your TestCase's getSessionManager().
- Use random names where possible.
 - Hardcoding folder and cabinet names will preclude multiple instances of the test from running against the same docbase.
- Use top-level test cabinets to facilitate easy tear down.
 - All IDfSysObjects that your tests create need to be linked somewhere.
 - Link all the objects for a test under a single test cabinet.
 - Tear down for the test is easy because the test cabinet can be deep-deleted, meaning the cabinet and all of its sub-folders and objects are deleted.
- Provide convenience methods to create test objects

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

Tearing Down State

- You don't want automated unit tests populating your repositories with test objects that are useless after the execution of the tests.
 - Automated tests are run continuously.
 - Test data needs to be cleaned up.
- Define an interface that manages the cleanup.
 - IDeleter
- Invoke delete actions in JUnit's *tearDown()*.
 - Guarantees that all test data is cleaned up.
 - Needs to be invoked as a superuser.
 - Consider adding an abstract method *getTearDownUser()* to your *TestCase* base that defines logical name of the tear-down user.

Tearing Down State - IDeleter

- Pluggable interface that is responsible for cleaning up repository state.
 - Allows generic objects to be queued for cleanup.
 - Provides hook for cleaning up all the queued objects.

```
public interface IDeleter {  
    public void queueObjectForDeletion(Object o)  
        throws DfException;  
  
    public void deleteQueuedObjects(IDfSession session)  
        throws DfException;  
}
```

Tearing Down State - Tips

- The order of deletion does matter.
 - There are special cases where you need to delay deleting an object until some other objects have been deleted.
- Inspect the type of the queued object to determine the action to take for a delete.
 - Use an IDfDeleteOperation for IDfSysObject, IDfFolder, IDfVirtualDocument, and IDfVirtualDocumentNode objects.
 - Other DFC objects, like IDfACL and IDfType, are handled on a case-by-case basis.
- Be sure to process all queued objects, even if an error is encountered while deleting one in the middle of the queue.
 - Use try/catch to prevent errors from “unrolling” the processing.

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- **Writing Your Tests**
- Running Your Tests
- Resources
- Questions

Writing Your Tests

- Set up a test repository.
- Decide on how you want to represent your login configuration data (XML, INI properties, database, etc.) and provide an implementation of *IAuthenticator*.
 - Can be reused across projects within your organization
- Provide an implementation of *IDeleter* based on the types of objects that your test code is creating.
 - Can be reused across projects within your organization
- Provide a subclass of `junit.framework.TestCase`
 - Can be reused across projects within your organization
 - Implements `getAuthenticator()` and `getDeleter()`
 - `getSessionManager()` calls to delegate and then adds transaction support
 - Has a `getSession()` method that handles application security
 - Has a `getTearDownUser()` that returns the logical user used for `tearDown()`
- Define your login configuration.
 - Determine how many logical users you will need in your tests
- Write your test methods.

Writing Your Tests - Sample getSessionManager()

```

/** Collection of session managers that have been transacted and need to be rolled back in tearDown() */
private Collection _transactedSessionManagers = new HashSet();

protected IDfSessionManager getSessionManager(String logicalName, boolean transacted)
    throws DfException
{
    // Get the session manager (guaranteed non-null if successful)
    IDfSessionManager sessMgr = getAuthenticator().getSessionManager(logicalName);

    // If the "cleanup via transaction" state is set, then make sure a transaction is started on the session manager
    if (transacted && !sessMgr.isTransactionActive()) {
        sessMgr.beginTransaction();

        // Keep track of all the transacted session managers, so we know which ones to rollback in tearDown()
        _transactedSessionManagers.add(sessMgr);
    }

    return sessMgr;
}

```

Writing Your Tests - Sample getSession()

```
protected IDfSession getSession(String logicalName, boolean transacted) throws DfException {
    // Grab a session manager and session
    IDfSessionManager mgr = getSessionManager(logicalName, transacted);
    IDfSession session = mgr.getSession(getAuthenticator().getDocbase(logicalName));

    // Apply application security, if it is defined
    if (getApplicationSecurityCode() != null) {
        try {
            // Recover the session config
            IDfTypedObject sessionConfig = session.getSessionConfig();

            // Set the application security on it
            sessionConfig.setRepeatingString("application_code", 0, getApplicationSecurityCode());
        } catch (DfException e) {
            // If error, be sure to release session before rethrowing
            release(session);
            throw e;
        }
    }

    return session;
}
```

Writing Your Tests - Sample tearDown()

```
protected void tearDown() throws Exception {
    // Keep track of last exception
    Exception exception = null;

    // Rollback transacted session managers
    try {
        for (Iterator iter=_transactedSessionManagers.iterator(); iter.hasNext(); ) {
            IDfSessionManager sessMgr = (IDfSessionManager) iter.next();

            try {
                sessMgr.setTransactionRollbackOnly();
                sessMgr.commitTransaction();
            } catch (Exception e) {
                exception = e;
            }
        }
    } finally {
        // Be sure to clear the list
        _transactedSessionManagers.clear();
    }
}
```

Writing Your Tests - Sample tearDown() (continued)

```
IDfSession session = null;
try {
    // Acquire session for teardown, but not in a transaction
    session = getSession(getTearDownUser(), false);

    // Delete everything which has been queued
    deleteQueuedObjects(session);
} finally {
    release(session);
}

// Throw any exceptions
if (exception != null) {
    throw exception;
}
}
```

Writing Your Tests - Sample Unit Test

```

public void testObjectCreation() throws Exception {
    IDfSession session = null;
    try {
        // Acquire a session as the logical account manager user
        session = getSession("account-manager");

        // Create a test cabinet. This will be randomly named and will be automatically queued for deletion.
        IDfFolder testCabinet = createTestCabinet(session);

        // Create a test object. This will be randomly named and will be linked under the test cabinet.
        // Because the cabinet will be cleaned up, the test object in the cabinet will be deleted as well.
        IDfSysObject testObject = createTestObject(session, "dm_document", testCabinet);

        // Set some data on the object
        String[] testValues = { "some", "sample", "data", "" };
        DfcUtils.setStringValues(testObject, "keywords", testValues);
        testObject.save();

        // Get the values back out and compare them
        String[] actualValues = DfcUtils.getStringValues(testObject, "keywords");
        assertTrue("keywords are incorrect", ArrayUtils.equals(testValues, actualValues));
    } finally {
        release(session);
    }
}

```

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- Resources
- Questions

Running Your Tests

- Use Ant to execute your tests.
 - Use a <junit> target to execute tests.
 - Use <sysproperty> sub-elements to pass System property definitions to the tests.
 - This is how you get your login definitions for your *IAuthenticator* passed to your test.
 - Be sure to include one for “java.library.path”, which must point to the directory containing the DFC dynamic libraries (i.e., C:\Documentum\Shared).
 - **If you do not have this, your DFC tests will not run!**
- Consider using these build properties:
 - For locating DFC JARs and dynamic libraries
 - documentum.config.dir
 - documentum.jars.dir
 - documentum.library.dir
 - For locating the configuration for the IAuthenticator
 - dfc.logins.resource (logins are in a co-located resource)
 - dfc.logins.filename (logins are in a local file)
 - dfc.logins.url (logins are accessible via a URL)

Running Your Tests – Sample Ant Target

```

<!-- Run an individual DFC test (specified via run.dfc.test.class property) -->
<target name="run-dfc-test" depends="set-test-classpath">
  <!-- Make sure the VM args and login locators are set to something -->
  <property name="run.dfc.test.vargs" value="" />
  <property name="dfc.logins.filename" value="" />
  <property name="dfc.logins.resource" value="" />
  <property name="dfc.logins.url" value="" />

  <junit printsummary="yes" haltonfailure="yes" filtertrace="{junit.filtertrace}" fork="yes">
    <!-- Pass the VM args, which are useful for debugging the test -->
    <jvmarg line="{run.dfc.test.vargs}" />
    <classpath refid="test-classpath"/>

    <!-- We need the java.library.path set to pickup the DMCL DLLs -->
    <sysproperty key="java.library.path" value="{documentum.library.dir}" />

    <!-- Set the properties used to resolve the logins -->
    <sysproperty key="dfc.logins.filename" value="{dfc.logins.filename}"/>
    <sysproperty key="dfc.logins.resource" value="{dfc.logins.resource}"/>
    <sysproperty key="dfc.logins.url" value="{dfc.logins.url}"/>

    <formatter type="plain" usefile="false"/>
    <test name="{run.dfc.test.class}" />
  </junit>
</target>

```

Running Your Tests – Sample Classpath

```

<target name="set-test-classpath" depends="set-classpath" description="Setup the
  classpath for the module tests">
  <path id="test-classpath">
    <!-- Add in the Documentum jars -->
    <fileset dir="${documentum.jars.dir}">
      <include name="*.jar"/>
    </fileset>

    <!-- Add in the Documentum config dir -->
    <pathelement location="${documentum.config.dir}"/>

    <!-- Add everything else -->
    <path refid="classpath"/>
  </path>
</target>

```

Running Your Tests – Sample Properties

```
# Specify how to resolve the login configuration.
# You have three options here, depending on where your configuration is stored (as a file, as a co-located resource
# to the build, and as a URL). Uncomment the type that makes sense for you and edit its value.
dfc.logins.filename=c:/some_project/logins.xml
#dfc.logins.resource=logins.xml
#dfc.logins.url=http://myserver:myport/somepath/logins.xml

# For running tests and cmdline programs
documentum.dir=c:/Documentum
documentum.config.dir=${documentum.dir}/config
documentum.jars.dir=${documentum.dir}/Shared
documentum.library.dir=${documentum.dir}/Shared

# If you have Documentum installed to Program Files, these lines will probably look like:
# documentum.dir=c:/Program Files/Documentum
# documentum.config.dir=C:/Documentum/config
# documentum.jars.dir=${documentum.dir}/Shared
# documentum.library.dir=${documentum.dir}/Shared

# Specify the single JUnit test class to run via the run-test target
run.dfc.test.class=com.bluefishgroup.bedrock.dctm.dfc.sample.SampleDfcTest

# Enable the following line if you need to debug your test. Be sure to set the port in your IDE and use the socket connection method.
#run.dfc.test.vmargs=-Xdebug -Xrunjdwp:transport=dt_socket,address=5000,server=y,suspend=y
```

Contents

- About Blue Fish
- Why Unit Test?
- The Tools
- The Pain Points
- Managing Logins
- Setting Up State
- Tearing Down State
- Writing Your Tests
- Running Your Tests
- [Resources](#)
- Questions

Resources

- Presentation material (coming this week to dm_developer)
 - <http://www.dmdeveloper.com/articles/programming/unitTestingDFC.html>
- JUnit
 - <http://www.junit.org>
- JUnit FAQ
 - <http://junit.sourceforge.net/doc/faq/faq.htm>
- *Test Driven Development using JUnit*
 - http://www.junit.org/news/article/test_first/index.htm
- *Mike Clarke's primer for JUnit*
 - <http://www.clarkware.com/articles/JUnitPrimer.html>
- Ant
 - <http://ant.apache.org>
- Ant FAQ
 - <http://ant.apache.org/faq.html>
- CruiseControl
 - <http://cruisecontrol.sourceforge.net>

EMC²[®]

where information lives[®]