# Unit Test Framework

Innovation Release

Version 10.0

April 2017

WEBMETHODS

This document applies to webMethods Version 10.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.



**Document ID: GWM-WWST-100-20170419**

# Table of Contents

# About this Guide

This guide provides information on service testing options of Integration Server using webMethods Unit Test Framework. The scope of this document is to introduce the suite, design, and execute test cases.

It is assumed that the user is familiar with the standard build and test tools such as, Ant and JUnit.

## Document Conventions

| Convention | Description |
|---|---|
| **Bold** | Identifies elements on a screen. |
| Narrowfont | Identifies storage locations for services on webMethods Integration Server, using the convention *folder.subfolder:service* . |
| UPPERCASE | Identifies keyboard keys. Keys you must press simultaneously are joined with a plus sign (+). |
| *Italic* | Identifies variables for which you must supply values specific to your own situation or environment. Identifies new terms the first time they occur in the text. |
| `Monospace font` | Identifies text you must type or messages displayed by the system. |
| { } | Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols. |
| \| | Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the \| symbol. |
| [ ] | Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols. |
| ... | Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...). |

# Online Information

### Software  AG Documentation Website

You can find documentation on the Software AG Documentation website at "http://documentation.softwareag.com". The site requires Empower credentials. If you do not have Empower credentials, you must use the TECHcommunity website.

### Software AG Empower Product Support Website

You can find product information on the Software AG Empower Product Support website at "https://empower.softwareag.com".

To submit feature/enhancement requests, get information about product availability, and download products, go to "Products".

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the "Knowledge Center".

### Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at "http://techcommunity.softwareag.com". You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

# 1 Unit Test Framework

# Overview

Unit Test Framework is an Eclipse based testing tool that allows the developers to create unit tests for their development. These tests can be used to improve the overall development quality and helps to provide a mechanism to create automated tools for continuous integration and delivery.

Unit Test Framework provides the following functionalities:

- Provides service unit testing and regression testing tools that allows service developers to assemble unit tests without the need for additional development
- Enables integration with JUnit to leverage a standard unit testing framework, which already works well with Continuous Integration tools
- Enables ease of use for test development
- Provides a Java API for advanced users to create JUnit test cases
- Provides a user interface that is integrated in Software AG Designer. Software AG Designer ensures that users do not switch between tools for services development and corresponding test cases
- Provides a mechanism to repeatedly execute the service with same inputs and compare the results with an expected set of outputs
- Provides a framework for mocking service execution for steps that cannot be executed during the testing. For details, see "Using Mocks " on page 26

# Capabilities

Unit Test Framework has following capabilities:

- Unit testing
- Mock testing
- Regression testing

### Unit testing

Unit Test Framework is a unit-testing tool. You can design, build, and execute unit test cases using Eclipse User Interface. You can also execute the test cases externally using Ant scripts.

### Mock testing

Mocking is a feature that mimics the functionality of services that are dependent on external resources. When a test case encounters a service that is mocked, it executes the service.

### Regression testing

You can save the test cases, along with their inputs and outputs, in xml files. Run the reusable artifacts to ensure that the latest changes do not reintroduce the errors fixed in the earlier versions.

## Environment

| Option | Description |
|---|---|
| Hardware Requirements | No additional hardware is required other than the ones that are already in use for Integration Server and Software AG Designer. |
| Software Requirements | ■ Unit Test Framework Eclipse plug-in v 9.12<br>■ Unit Test Framework library extension for Integration Server<br>■ WmServiceMock package for Integration Server<br>■ Ant Build tool 1.7 (optional)<br>■ JUnit Tool (optional)<br>■ JDK 1.8 or later |
| Version Compatibilities | This suite depends on open source products like Ant and JUnit.<br>Following are the supported versions:<br>■ Ant: 1.7<br>■ JUnit: 3.8.2<br>Unit Test Framework works with all the currently supported General Availability (GA) versions of Integration Server and Software AG Designer. |

## What it is not

- Unit Test Framework is not an integration or system test platform. However, this suite is used to mock the application dependencies and simulate the integration or system test.
- Unit Test Framework is not a performance-testing tool. It cannot be used for performance, load, or volume testing.

# 2 Working with Unit Test Framework

# Getting Familiar

This section provides information on the terms and concepts used to understand Unit Test Framework.

## Services

The webMethods Integration Server hosts packages that contain services and related files. The server contains several packages.

For example, Packages that contain built-in services, which can be invoked from services or client applications and services that demonstrate features of the webMethods Integration Platform.

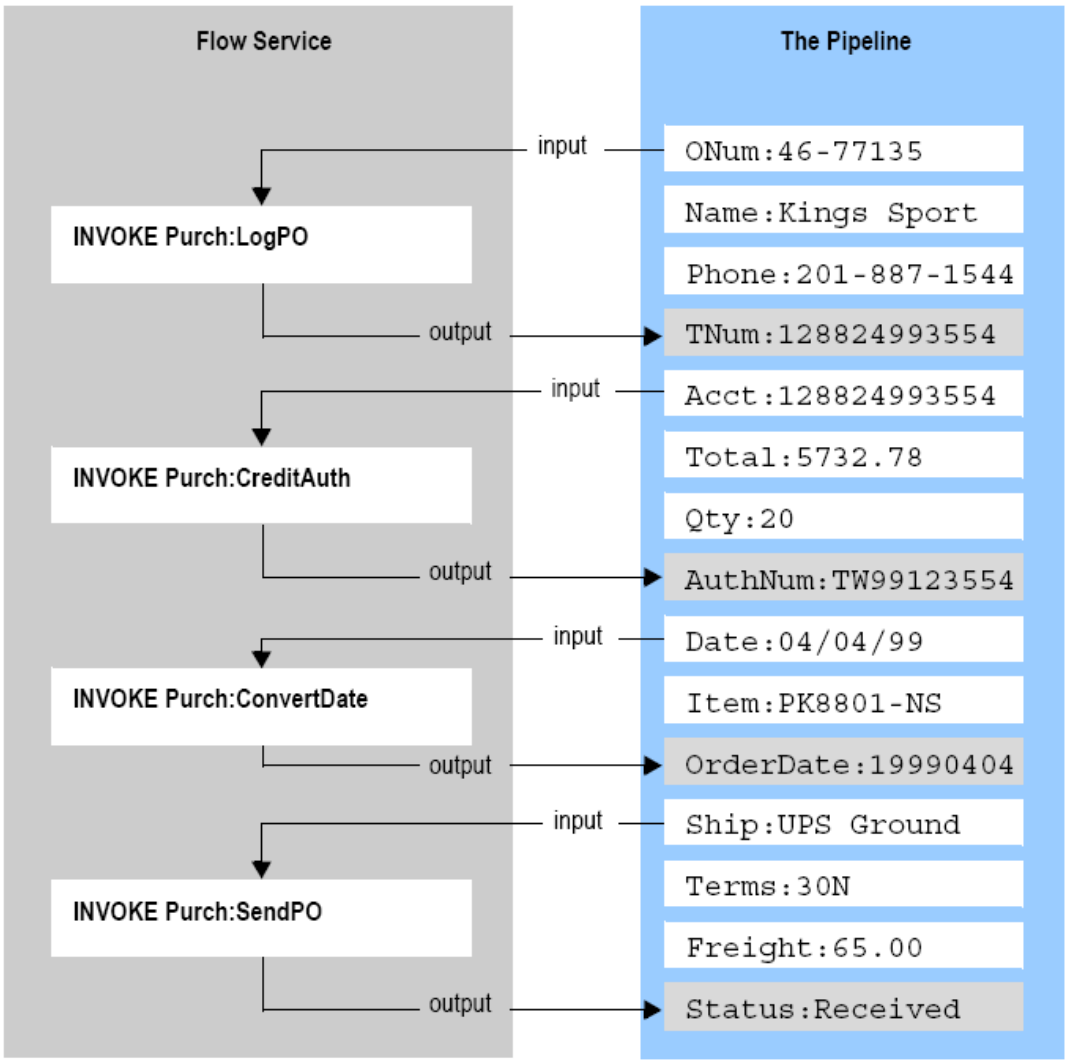
You can create additional packages to hold the services that your developers create. Developers can create services that perform functions, such as, integrating your business systems with those of your partners, retrieving data from legacy systems, and accessing and updating databases.

Integration Server provides an environment for the orderly, efficient, and secure execution of services. It decodes client requests, identifies the requested services, invokes the services, passes data to them in the expected format, encodes the output produced by the services, and returns output to the clients.

## Pipeline

Pipeline refers to the data structure in which input and output values are maintained for a flow service. It allows services in the flow to share data.

The pipeline holds the input and output for a flow service




Pipeline starts with the input to the flow service and collects inputs and outputs from subsequent services in the flow. The service in the flow has access to all data in the pipeline at that point.

## Unit Testing

Unit Test Framework uses the concepts of service execution, pipeline data, and the open source JUnit testing framework to provide unit testing functionality for Flow and Java services. Unit Test Framework provides the ability to create a suite of tests consisting of individual test cases. Each test case defines a service to be tested, the type of test to be performed, and provides a user interface to define input data to the test case through the pipeline. When the service execution is completed, the pipeline output is validated against the expected output defined in the test case.

## Test Case

A test case is a unit of testing for a service that provides:

- service to be tested
- inputs to the service
- expected output from the service

A test case can also define expected output from a service as an exception or error. The service returns the defined errors when incorrect data is sent to it.

## Test Suite

A test suite is one or more test cases grouped together. Test suites are used to organize test cases into sets of related tests. For example, a service may provide a variety of capabilities based upon the inputs provided to it. A complete test suite should include test cases that provide inputs that fully test all of the possible outputs of the service, including errors or exceptions.

## Mock

Mocks provide a means of simulating interaction with resources that are unavailable or the data provided by these resources or systems is not consistent for test purposes. Mocks also have a lifetime that can either be limited to the test case in which they are defined or applied to all of the test cases that follow within a test suite from the point of the definition.

Mock intercept can control the session, user, and server based on the scope setting. If the scope is set to session, the test sessions will be affected by the mock. It is recommended to set the scope to session for most users. If the scope is set to user, all the sessions for the particular user will be affected by the mock. If the scope is set to server, all user sessions will be affected by the mock. Mocks can be enabled or disabled for test case or test suite execution.

## Service Mock

A service mock is used to replace the call to a service with a call to a different service. Any call to the mocked service is intercepted and the alternate service defined in the mock is called instead. The output of the mocked service is then returned to the calling service. This kind of mock is useful when the output of the mocked service needs to be dynamic based on some logic that can be created in the service.

## Exception Mock

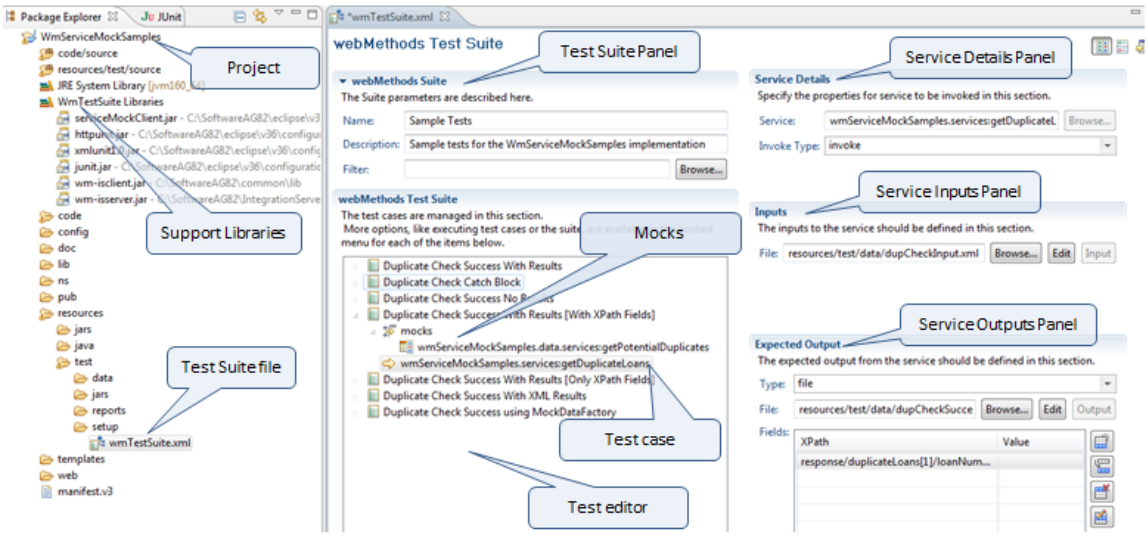
An exception mock is used to return an error or exception to the calling service and can be useful for testing error handling in a service. As with the other mocks described, any call to the service defined in the mock is intercepted and the exception defined in the mock is returned instead. This kind of mock is useful to simulate behavior that can cause exceptions in the normal flow.

## Factory Mock

A factory mock is used when a call to a service produce multiple different outputs based on the provided input. A factory mock is implemented as a Java class. Any call to the class defined in a factory mock is intercepted and the input is passed to the factory which evaluates the input and returns the appropriate results. While both the factory mock and the service mock options provide dynamic output simulation, the factory mock does not require an extra test service to be created on the server and relies on a lightweight java implementation.

# Layout

The figure below shows various components of with a test suite file open for editing.




Use the following icons on the toolbar to tailor the layout.

| Icon | Description |
|---|---|
|  | Allows you to place the Master and Details views next to each other, master on the right and details on the left. |

| Icon | Description |
|---|---|
|  | Allows you make the master appear at the top of the display, and the details view gets aligned underneath it. |
|  | Allows you to toggle the display between the master and details views, each occupying the entire display. Click again to return to the original layout . |

# Unit Test Framework Preferences

To display the preferences dialog box, select **Window>Preferences>webMethods Tests**.

## Advanced

The Advanced preferences control other behaviors of Unit Test Framework.

| Preference | Description |
|---|---|
| **Use Relative Paths in Filenames** | Makes file name references relative to the Eclipse project containing the test suite rather than using the full path. Using relative names helps in executing tests when the test suite and data files are moved between environments and systems. This option is enabled by default. |
| **Allow custom comparator** | Enables the option for the test creator to specify a custom comparator for expected output other than provided with the plug-in. The default setting is appropriate for most test case scenarios but may be needed for the advanced user. This setting only controls the display of the field that allows the selection of the comparator and does not allow/disallow this during execution. |
| **Allow client side mock factory objects** | Enables the option of creating client side factory object that do not need to be deployed on the server before executing test cases. This option controls the display of the field that controls whether the mock factory objects are needed on the server or can be pushed during the execution of the tests. |
| **Display all fields in the expression editor** | Enables additional fields in the Expected Output section of a test case. This option is useful if you want to check several output conditions together. |

| Preference | Description |
|---|---|
| **Allow editing of XML (Effective after reopen)** | Enables editing of the XML source for the test suite. This option requires that the open test suite file is closed and reopened before its behavior reflects in the editor. |
| **Allow scope selection for mocks** | Enables scope selection for the mock. The default setting of unchecked is appropriate for most test case scenarios. |
| **Confirm delete of single objects** | Enables the display of a confirmation dialog box for every mock service or data entry delete. |
| **Confirm delete of multiple objects** | Enables the display of a confirmation dialog box when you delete multiple mock service or data entries. |
| **Confirm service paste** | Enables the display of a confirmation dialog box when you paste a test services. |

## License

The License preferences provide the license related information of Unit Test Framework.

| Preference | Description |
|---|---|
| **License file** | Allows you to select the license file for Unit Test Framework. |
| **Check License** | Allows you to check the validity of the license file. |

## Server

The Server preferences describe the connection to for Unit Test Framework.

The editor does not always require an active connection to the for test development. But some of the introspection features which allow for service lookup and service signature are not available if a connection is not available and the user would have to enter these manually.

| Preference | Description |
|---|---|
| **Connect to Server** | When selected, executes tests against a specific . Additionally, provide the **Server**, **Port**, **Username**, and **Password** and click **Test Connection** to ensure connectivity to the selected . |
| **Package Filter** | Allows you to optionally enter a comma-separated list of packages for Unit Test Framework to load to the Service Browser. When a large number of packages exist on the , this feature loads only those services for which test cases are developed and thereby conserves memory in Unit Test Framework. |

## Validation

The Validation preferences describe when and how Unit Test Framework should validate the information that you enter.

| Preference | Description |
|---|---|
| **Validate on Save** | Allows Unit Test Framework to validate the test suite prior to saving it to the file system. This option is enabled by default. |
| **Validate before switching mode** | Allows Unit Test Framework to perform validation of the test suite prior to switching from the XML editor mode to the test suite editor. |
| **Validate against schema** | Allows Unit Test Framework to validate the test suite against the XML schema. This is useful if you use the XML Source tab to enter details of one or more test cases in the suite. This option is enabled by default. |

# 3 Creating a Test Suite

## Before You Begin

Before creating a test suite, ensure that you organize the required test cases and data files in a test folder.

If the test cases are packaged in the Integration Server package, it provides a common source location for all related assets. You can import the package on eclipse workspace.

Example: Consider that the test cases are created in the resources test folder of the Integration Server package.

While any organizational structure that conforms to Integration Server package structure can be used, the following example provides a useful approach for organizing test cases in your environment. For ease of organization, follow the steps below:

1. Right click the **Resources** directory and create a subdirectory under it called **test**.
2. Right-click the **test** directory and create two additional folders **data** and **setup**.
3. Add the test suites to the **setup** directory and organize the **data** directory using subdirectories for each test suite to contain the input data files for the test cases comprising a test suite.

## To Create a Test Suite

Follow the below steps to create a test suite:

1. Navigate to **resources**> **test**> **setup folder**.
2. Select **File**>**New**>**Software AG**
3. Select **webMethods TestSuite** and click **Next**.
4. On the **webMethods TestSuite** screen, enter the folder path that contains the test suite setup files, provide a file name, and a suite name.
5. Click **Finish**.

   You can now start creating the test cases.

# 4 Adding Test Cases

# Adding a Test Case

Adding a test case involves the following steps:

1. Adding required test details.
2. Adding the required service details.
3. Defining inputs to services.
4. Defining the expected output.

# Test Details

To add test cases:

- Right click in the **webMethods Tests** section of the display.

  The **Test Details** section appears in the details area to the right side of the display. Fill in the name of the test case and a description of the test case.

# Service Details

After creating the test details, you can now select the service to be tested.

1. Click **Browse** in the Service Details section to enable searching for the service within the Integration Server.

   **Note:** You can narrow the search by entering a package name or service name and the service browser will restrict the display to only those packages or services matching the search criteria entered.

2. Click ▼ to refresh or set additional options for the service browser.

# Inputs

looks at the *service signature* for services in the . When defining the inputs to the service selected in "Service Details" on page 22:

| Options | Description |
|---|---|
| **Input** | Displays the structure of the inputs defined in the service signature. Provide values for each of the input parameters |

| Options | Description |
|---|---|
| | defined in the service, including complex structures such as, document and nested documents. |
| **Edit** | Uses the same interface for modifying previously saved input data files in the same way. |
| **Save** | Saves the file in the location designated for input data to the test suite after defining the input. |
| **Browse** | Selects the saved file and adds it as the input to the service. |

# Expected Output

The final step in defining a test case is to complete the Expected Output section. Expected Output can be in the form of data returned from the service or an exception returned from the service.

Use the class browser to define an exception or error output from the service. Click **Browse** to display the classes and select the proper exception class to use.

To define the data output from the service, use the following methods:

- Use XPath expressions to define which data elements in the output data should be evaluated.

  1. Copy the XPath expressions from Software AG Designer. To copy the XPath expression, right click on the variable in the **Results** area when executing the service within and select **Copy**.
  2. Click  ,  , or  and paste the copied value into the XPath field.

     An entry for the variable selected is created. However, it in most cases it is not required to edit them as the webMethods paths are 0 index based whereas XPath expressions are 1 index based.
  3. Enter the expected output for the field in the **Value** field.

     You can select AND, OR, parentheses, and operators from the drop-down lists to create complex evaluations of the output.

- Use regular expressions to evaluate the output returned from the service. The regular expression should be placed in the Value field preceded and followed by "/" character.

# 5 Mocks

# Using Mocks

Mocks are used when resources that a service may require to properly execute may not be available when a test case or test suite is developed or executed. Mocks provide a means of simulating interaction with resources that are unavailable.

**To create a mock**

1. Expand the test case for which you want to define the mock, right click on **Mocks** and select **Add**.
2. Click **Browse** and use the **Service Browser** to select the service to be mocked.
3. Enter the first few characters of the service name to reduce the list. Click to refresh the list and access preferences for the **Service Browser.**
4. Select the required **Lifetime**.

   Valid selections are **test** (mock is effective only for the selected test case) and **suite** (mock will be effective for all of the test cases that follow in the test suite).
5. Select the required **Type**.

   Following are the valid selections:

| Type | Function |
|---|---|
| **pipeline** | Intercepts the service and returns the specified pipeline (for details on creating or editing pipeline data see "Inputs " on page 22). |
| **service** | Intercepts the service and substitutes the selected service with a call (for details on working with the Service Browser, see "Service Details" on page 22). |
| **exception** | Intercepts the service and returns an exception (for details on selecting exception classes, see "Expected Output" on page 23). |
| **factory** | Intercepts a call to the mocked service and returns the data based on the input (for details on creating a mock factory, see "Sample Mock Factory" on page 27). |

6. Select the scope.

   It is recommended to use the **session** scope for most purposes. To allow the scope selection, select the corresponding preference.

## Sample Mock Factory

The following code snippet illustrates the minimum requirements for creating a mock factory. The factory class and any other classes should be designed to evaluate the input data to the factory and return data relevant to that input in an IData format. The example below returns static data.

```
package com.wm.ps.serviceMock.samples;
import com.wm.app.b2b.server.BaseService;
import com.wm.app.b2b.server.invoke.ServiceStatus;
import com.wm.data.IData;
import com.wm.data.IDataFactory;
import com.wm.ps.serviceMock.MockDataFactory;
public class SampleMockDataFactory implements MockDataFactory
{
  private static final long serialVersionUID = 2L;
  public IData createData(BaseService baseService, IData pipeline, ServiceStatus serviceStatus)
  {
    IData[] results = new IData[]{IDataFactory.create(new Object[][]{
        {"originationSource","W"},
          {"bizType","RT"},
          {"lockExpirationDate","20050427"},
          {"floatLoanIndicator","Y"},
          {"uwFinalDecisionCode","0"},
          {"uwDecisionExpiryDate","20050427"},
          {"canDate","20050427"},
          {"loanCloseStatusType","T"},
          {"fileReceivedAtRocDate","20050221"},
          {"loanReadyToFundIndicator","P"},
          {"regisDate","20051221"},
          {"loanSubmitToUwDate","20050427"},
          {"loanNumber","0000000001"},
          {"branch","TOTAL ADVANTEDGE LLC          "},
          {"underwritingDecisionCode","0"},
          {"underwritingDecisionExpirationDate","20050427"},
          {"lockDate","20051220"},
          {"lockIndicator","Y"},
          {"tmoLoanStageCode","3"},
          {"tmoLoanStageDate","20050427"},
          {"product","C30       "},
          {"borrowerFirstName",".           "},
          {"borrowerLastName","XX              "},
          {"propertyAddress","937 S MEYER                   "},
          {"propertyCity","TUCSON                    "},
          {"propertyState","AZ"},
          {"propertyZip","85701"}
      })};
    IData output = IDataFactory.create(new Object[][]{{"results", results}});
    return IDataFactory.create(new Object[][]{{"getPotentialDuplicatesOutput", output}});
  }
}
```

## Mocks beyond Unit Testing

Although Unit Test Framework added the ability to mock service calls in Integration Server for unit testing, the feature is so powerful that its use cannot be limited to unit

testing alone. One common case is to use the mocking capability to provide flow service instrumentation.

Using the wm.ps.serviceMock:loadMock service in the WmServiceMock package, any service can be mocked with an alternate service or class. The new service or Java class code can invoke any operations and then invoke the original mocked service. The mocking framework is intelligent enough to detect recursion and, as such, provides an instrumentation capability.

Mocks can be used to design test cases. A service being tested can also itself be mocked with other code. In such a scenario, the mocked test service can be replaced with other code that can execute pre and post-test operations. This can provide some basic functional testing capabilities for Unit Test Framework.

# 6 Advanced Options

## Pipeline Filter

The execution of test cases is initiated as a client to the Integration Server hosting the services to be tested. For this reason, the inputs supplied to the service during execution and the expected outputs need to be serialized over the client-server interaction. If the input or output pipeline contains non-serializable objects, these objects are either lost or seen incorrectly during test execution. In addition, the service input needs to be more dynamic in nature than the static pipeline setup in the test case. Pipeline Filter helps to resolve these issues.

The Pipeline Filter is set once for the entire suite and provides a callback mechanism for the test developer to inject code that can modify various pipeline objects during execution. The Pipeline Filter is a class that implements the com.wm.ps.test.PipelineFilter interface and enables a user to add, remove, or change variables in the pipeline that are created from files, as pipelines created from files may not be able to persist custom java objects. The output pipeline from a service can also be filtered using the appropriate method. Only one such instance of the implementing class is created for the test suite and the name of test case is passed as a parameter.

The pipeline filter can be setup for the test suite in the main panel for the suite parameters.

## Custom Comparator

Custom comparator provides an extension that can be used to extend, enhance, or replace the standard comparison of expected output.

Each test suite can have its comparator that can be specified from the user interface.

Custom comparators are Java classes that implement the com.wm.ps.test.ResultsValidator interface.

Custom comparators also provide a mechanism to execute operations pre and post service execution. Using custom comparators you can build some basic functional testing capability. For example, if a service writes to a database table, a custom comparator can compare the results from the service and check the destination table to confirm the operation executed successfully.

## Client Mock Factory

The benefit of using a mock factory object as opposed to a service is that it provides a light-weight alternative that does not require the creation of a new service on the server.

Unit Test Framework provides an option to dynamically push the classes needed for supporting the mock factory on the server during test execution. This option can be used to avoid the need for frequent restart when Java objects are changing. Once stabilized,

it would be helpful to deploy the code to the server as this feature is experimental in nature and will only work if the dependency tree is not too complex. Using the user interface, you can set the option to dynamically push the objects to the server.

If the option is disabled, enable the corresponding preference as discussed in the Advanced section.

| webMethods expression | JXPath equivalent |
|---|---|
| PosRequest/ns:Log/ns:Transaction[0]/@Flag | PosRequest[@name='ns:Log'] [@name='ns:Transaction'][1] [@name='@Flag'] |
| PosRequest/ns:Log/ns:Transaction[0]/ ns:BUnit[0]/ns:ID/*body | PosRequest[@name='ns:Log'] [@name='ns:Transaction'] [1][@name='ns:BUnit'][1] [@name='ns:ID'][@name='*body'] |

# XPath Expressions

The XPath expressions used in the expected output panel are different from the usual webMethods path expressions. As mentioned in the “Expected Output” on page 23 section, the indices start at 1 instead of the 0 based webMethods indices.

Unit Test Framework uses JXPath for evaluating XPath expressions. For details on JXPath expressions, visit “http://commons.apache.org/jxpath/”. Special characters such as '@' and ':' in the name have special meaning in JXPath expressions hence, you should use special syntax variant to use these characters in variable name.

# 7 Test Suite Internals

# Test Suite Internals

In Software AG Designer, test suite editor provides a user interface to graphically and quickly develop test cases for Integration Server services. The test suite and the test cases are saved in an XML file. The Software AG Designer editor allows editing the XML source directly, provided that the user is aware of the format and the associated schema.

It is not recommended to edit the XML file as it is error-prone. It provides the option to automate the creation of test cases automatically by using code to generate the XML file directly. One such use case is the scenario where service inputs and outputs have been captured in an environment and test cases have to be generated to use these files for regression testing.

A sample XML test suite file in its simplified form can be as shown in the figure below:

```
<?xml version="1.0" encoding="UTF-8"?>
<webMethodsTestSuite description=
"Sample tests for the WmServiceMockSamples implementation" name="Sample Tests">
    <webMethodsTestCase description=
"Duplicate Check Success with IData results" name="Duplicate Check Success With Results">
        <mock folder="wmServiceMockSamples.data.services" name="getPotentialDuplicates">
            <pipeline filename="resources/test/data/mockDupCheckOutputResults.xml"/>
        </mock>
        <service folder="wmServiceMockSamples.services" name="getDuplicateLoans">
            <input>
                <file filename="resources/test/data/dupCheckInput.xml"/>
            </input>
            <expected>
                <file filename="resources/test/data/dupCheckSuccessWithResults.xml"/>
            </expected>
        </service>
    </webMethodsTestCase>
    <webMethodsTestCase description=
"Duplicate Check Failure handled by the catch block" name="Duplicate Check Catch Block">
        <mock folder="wmServiceMockSamples.data.services" name="getPotentialDuplicates">
            <exception class="java.lang.IllegalArgumentException" message="Bad argument"/>
        </mock>
        <service folder="wmServiceMockSamples.services" name="getDuplicateLoans">
            <input>
                <file filename="resources/test/data/dupCheckInput.xml"/>
            </input>
            <expected>
                <exception class="java.lang.IllegalArgumentException" message="Bad argument"/>
            </expected>
        </service>
    </webMethodsTestCase>
</webMethodsTestSuite>
```

# 8 Java Unit Tests

# Java Unit Tests

Unit Test Framework Java API allows you to create pure JUnit test cases that can provide the same features that a user interface driven codeless test cases do.

The change when creating a Unit Test Framework JUnit test case from the traditional test case is that the implementing class extends com.wm.ps.test.WmTestCase instead of junit.framework.TestCase. com.wm.ps.test.WmTestCase does extend the junit.framework.TestCase. The two important methods that are needed for creating test cases using the java API are:

- invokeService – The method to invoke a service on the server
- mockService – There are various variants of this method that allow the user to setup a mock for a service on the server.

A sample JUnit test case is provided here:

```
package com.wm.ps.serviceMock.samples;
import com.wm.data.*;
import com.wm.ps.test.*;
public class DuplicateCheckTest extends WmTestCase
{
  public void testDupCheckCatchBlock() throws Exception
  {
    IData input = IDataFactory.create(new Object[][]{
        {"lienType", "1"},
        {"borrowerSSN", "111-11-1111"},
        {"propertyAddress", "937 S Meyer"},
        {"propertyZip", "85701"}
        });
    String exceptionText = "Bad argument";
    mockService("wmServiceMockSamples.data.services", "getPotentialDuplicates", new
IllegalArgumentException(exceptionText));
    try
    {
      invokeService("wmServiceMockSamples.services", "getDuplicateLoans", input);
      assertFalse(true); //Control getting here means failure
    }
    catch (Exception e)
    {
      assertTrue(e.getMessage().endsWith(exceptionText));
    }
  }
  public void testDupCheckSucessWithResults() throws Exception
  {
    IData input = IDataFactory.create(new Object[][]{
    {"lienType", "1"},
    {"borrowerSSN", "111-11-1111"},
    {"propertyAddress", "937 S Meyer"},
    {"propertyZip", "85701"}
    });
    IData mockOutput =
WmTestSuiteUtils.getIDataFromFile("resources/test/data/mockDupCheckOutputResults.xml");
    mockService("wmServiceMockSamples.data.services", "getPotentialDuplicates",
    mockOutput);
    IData output = invokeService("wmServiceMockSamples.services", "getDuplicateLoans",
     input);
```

```
    IDataCursor outCursor = output.getCursor();
    IData response = IDataUtil.getIData(outCursor, "response");
    IDataCursor responseCursor = response.getCursor();
    String creationTime = IDataUtil.getString(responseCursor, "@creationTime");
    assertNotNull(creationTime);
    assertEquals(28, creationTime.length());
    IData[] duplicateLoans = IDataUtil.getIDataArray(responseCursor, "duplicateLoans");
    assertEquals(duplicateLoans.length, 1);
  }
}
```

# 9 Executing Tests

## Executing Test Cases

To execute test cases created with Unit Test Framework, right click in the webMethods Tests section and select one of the following options:

- **Run Tests** - executes the selected test case, using the mocks that are defined for the test case.
- **Run Test with Mocks Disabled** - executes the selected test case by disabling the mocks defined for the test case.
- **Disable Test** - marks test case as disabled. Tests are not executed until it is enabled again.
- **Disable Mocking in Tests** - marks mocks defined for selected test as disabled. Mocks are not executed until they are enabled again.
- **Add** - adds another test case to the suite.
- **Insert** - inserts another test case to the suite after the selected test case.
- **Remove** - removes the selected test case from the suite.

## Executing Test Suites

To execute test suites created using Unit Test Framework, right click in the webMethods Tests section and select one of the following options:

- **Run Suite** - executes the selected test suite, using the mocks that are defined for the test suite.
- **Run Suite with Mocks Disabled**- executes the selected test suite by disabling the mocks de-fined for the test suite.
- **Disable Suite** - marks test suite as disabled. The test suite is not executed until it is enabled again.
- **Disable Mocking in Suite** - marks mocks defined for this suite as disabled. Mocks are not executed until they are enabled again.
- **Shift Up or Shift Down** - changes the order of test cases in the test suite by shifting the selected test case up or down in the test suite.

## Debugging Java Code

Various components in Unit Test Framework rely on Java code.

Example: Java mock factory and pipeline filter classes.

Use the **Debug** menu to debug a Java code. The support library jar files also have source code associated with them. Debugging into the source can be helpful to understand the internals of the test execution or to enhance capabilities new features like custom comparators.

# 10 Service Usage

## WmServiceMock Services

The services in the WmServiceMock package deal with the mocking aspects of testing. These services provide a way to enable or disable mocks for individual services or server-wide.

| wm.ps.serviceMock:loadMock | | | |
|---|---|---|---|
| Sets up mocking for a service. | | | |
| Inputs | scope | session, user, or server. The default is session. | |
| | service | The name of the service to be mocked. No validation is performed on the name of the server or its existence. | |
| | mockObject | The mockObject is an object type. The behavior of the mock is controlled by the type of the actual object. | |
| Inputs | | java.lang.String | The name of the alternate service to mock the mocked service with. |
| | | java.lang.Exception | The exception to be thrown for the mocked service. |
| | | com.wm.data.IData | The fixed pipeline to return for the mocked service |
| | | com.wm.ps.serviceMock.MockDataFactory | The implementation of the factory objects that creates the dynamic pipeline for |

| wm.ps.serviceMock:loadMock | | |
|---|---|---|
| | | the mocked service. |
| | parms | Optional document containing all extra parameters for the alternate service. This parameter is only needed when mocking a service with an alternate service and the alternate service needs additional inputs. |
| Outputs | None | |

| wm.ps.serviceMock:clearMock | | |
|---|---|---|
| Removes mocking for a service. | | |
| Inputs | scope | session, user or server. The default is session. |
| | service | The name of the service to be mocked. No validation is performed on the name of the server or its existence. |
| Outputs | None | |

| wm.ps.serviceMock:clearAllMocks | |
|---|---|
| Removes mocking for all services. | |
| Inputs | None |
| Outputs | None |

| wm.ps.serviceMock:suspendMocks | |
|---|---|
| Suspends mocking for all services. | |
| Inputs | None |
| Outputs | None |

| wm.ps.serviceMock:resumeMocks | | |
|---|---|---|
| Resumes mocking for all services, for which it was suspended. | | |
| Inputs | None | |
| Outputs | None | |

| wm.ps.serviceMock:getMockedServices | | |
|---|---|---|
| Retrieves the list of services that are currently mocked in all the scopes. | | |
| Inputs | None | |
| Outputs | mockedServices | The list of services that are currently mocked in the session, user, or server scope. |

# References

- Javadoc API Reference - The javadoc reference for the Unit Test Framework java API. This is useful for advanced Unit Test Framework features as well as creating pure java unit tests for Integration Server services.
- "JXPath"– Documentation for JXPath API