Automated Interface Test Framework

Service Oriented Integration (SOI) services

Sravana Krishna
Saran Sankar

April 14, 2010

Automated interface testing alleviates the inconsistencies introduced by manual testing. Comprehensive interface testing is very important to assure regression testing of the code, whenever a change is made to the code. Exhaustive testing of all the test cases is hard to achieve if this testing is done manually. This framework provides the foundation to build automated interface testing for Service Oriented Integration solution.

Introduction

This article introduces a proven automated interface test framework (ITF) for Service Oriented Integration (SOI) type of interfaces. The framework provides capabilities to manage test cases for unit testing of interfaces, ability to group them, and generate reports after test execution. This approach can be used on any Enterprise Service Bus (ESB) or middleware product.

Service Oriented Integration involves integration with applications or databases outside the middleware environment. The development of the solution is often done using middleware products using graphical editors or code snippets that are transformed by the product into executable code. It is hard to test the individual components of the interface comprehensively without an automated testing framework.

This framework provides a way to perform white box interface testing for Service Oriented Interfaces. White box testing requires knowledge of internal structures to identify the best ways to test the system thoroughly. This framework is a powerful tool that provides an environment for unit or component testing in which a component can be tested in isolation or with suitable stubs and drivers. This framework can be extended quickly to support regression testing and smoke testing of the Interfaces in various environments. It has the ability to:

- Test for expected results.
- Share common test data.
- Run selective or group of test cases.

It can also help in formalizing the requirements, clarifying the architecture, debugging, integrating, releasing, optimizing and testing the code.
Purpose of an Interface Test Framework

The act of writing test cases using the framework often uncovers design or implementation flaw. The unit tests serve as the first users of ESB system and will frequently identify design issues or functionality that is lacking. It takes the typical developer time and practice to become comfortable with ITF. Once a unit test suite is written, it serves as a form of documentation for the use of the ESB system. With ITF, it is easy to create data-driven unit tests, whereby a unit test is executed once per record in a data source, and has full access to that bound data.

Perhaps one of the most important benefits of ITF is that a well written test suite provides the original developer with the freedom to pass the code off to other developers for maintenance and further enhancement. Should those developers introduce a bug in the original functionality, there is a strong likelihood that those unit tests will detect that failure and help diagnose the issue.

ITF is an essential element of regression testing. Regression testing involves retesting a piece of software after new features has been added to make sure that errors or bugs are not introduced. Using ITF, time to Regression test is drastically reduced.

ITF features

The following are some of the key software testing features that are supported by the ITF.

Test-driven development

Test-driven development (TDD) is the practice of writing unit tests before writing the code that will be tested. Developers are encouraged to write test cases based on specifications before the system is designed and built. TDD encourages following a continuous cycle of development involving small and manageable steps. ITF can play a pivotal role in an enterprise with its support for TDD implementation.

Assertion

The most common way to determine success in unit test is to compare an expected result against an actual result. Assertion refers to the ability of ITF to facilitate developers to provide an expected result as the outcome of the test and force a comparison with the actual result post execution. Dynamic values within the expected results that are set during runtime are handled using regular expression based comparison. This feature is critical for test execution comparisons.

Code coverage

ITF features full support for code coverage. Code coverage automatically inserts tracking logic to monitor which activity within the process is executed during the execution of the tests. The most important result of this is the identification of regions of the code that was not reached with the tests. Often, one may have branching or exception-handling logic that isn't executed in common situations. It is critical to use code coverage to identify these areas. Code coverage is a useful tool, but it should not be relied upon as an exclusive indicator of unit test effectiveness. It cannot tell you the manner in which code was executed, possibly missing errors that would result with different data or timing. A suite of unit tests based on a variety of different inputs and execution orders will
help to ensure that the code is correct, complete, and resilient. Hence code coverage to helps identify code the tests are missing.

**Negative testing**

A robust code is expected to handle all exception scenarios is a clean manner without breaking. A developer will often want to verify that the code behaves correctly when it encounters an exception. This can be induced by providing erroneous or invalid input data. Normally, a unit test that throws an exception is considered to have failed. ITF supports negative test scenarios. Developers have option to specify if a test case is a positive test case or a negative test case.

**Logging and exception coverage**

Good coding practice calls for efficient logging and exception handling by the code. As part of unit testing, the developer needs to verify if the code is logging messages as expected and if the exception are handled correctly. It is common for enterprises to have global audit logging (GAL) and global exception handling (GEH) frameworks. ITF can integrate smoothly with the GAL and GEH frameworks. It provides capabilities for the developers to specify the number of expected Log and Exception message for a particular test case. This feature enables the developer to identify if the code is routing all log and exception messages correctly.

**Regression testing**

The intent of regression testing is to assure that a bug fix has been successfully corrected based on the error that was found, while providing a general assurance that no other errors were introduced in the process of fixing the original problem. Regression is commonly used to efficiently test bug fixes by systematically selecting the appropriate minimum test suite needed to adequately cover the affected software code/requirements change. Common methods of regression testing include rerunning previously run tests and checking whether previously fixed faults have re-emerged. The ability to rerun test cases from ITF very easily and quickly provides great capability to use ITF as a regression test platform.

**Performance testing**

Performance testing is performed to determine how fast some aspects of a system perform under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Performance testing strives to build performance into the design and architecture of a system, prior to the onset of actual coding effort. Test cases created in ITF can be run concurrently from multiple machines and can be iterated. This provides a good performance testing platform.

**Requirements for the Framework**

The following are some of the important requirements met by the ITF.

1. ITF should allow addition of new test cases.
2. ITF should have the ability to group related test cases into a test suite.
3. ITF should be capable of executing and summarizing Individual test case, or group of test-cases or all test-cases created in ITF.
4. ITF should be capable of creating various endpoints which can be reused across the test cases.

5. ITF test-cases should include:
   1. Unique Name
   2. Description
   3. Group(s) to which a test case belongs
   4. Input endpoint details
   5. Output endpoint details
   6. Timeout value

6. ITF endpoint should include:
   1. Endpoint type: Webservice (WS) or Java Message Service (JMS). This will determine if the service being tested is JMS-JMS, WS-WS, etc.
   2. Message type: Bytes, Text etc.
   3. Encoding type: This is needed for bytes messages. Default is UTF-8
   4. Compare Flag: This field indicates if framework needs to compare actual output with expected to assert the test.
   5. Header details
      1. destinationQueue: destination queue name
      2. replyToQueue: replyTo queue name, needed only for WS endpoint. Default is temporary queue.
      4. JMSDeliveryMode: JMS delivery mode, PERSISTENT or NON_PERSISTENT. Default is PERSISTENT.
      5. JMSCorrelationID
      6. requestTimeout: Message timeout value. Needed only for WS endpoint.
   6. Property details
      1. SOAPAction: Needed only for WS endpoint
   7. Body: Needed only for JMS end point
   8. Request: Needed only for WS end point. It should include the SOAP envelope for webservice
   9. Reply: Needed only for WS end point. It should include the SOAP envelope for webservice

7. ITF Test outcomes result should contain one of the following:
   1. Success: Everything worked fine
   2. Failure: The output was received but did not match the expected output
   3. Error: The process being tested generated an error and failed to generate any output for comparison

8. ITF should have the ability to run on any deployable environment.
9. ITF should have the ability to check the log entries in GAL and GEH.
10. ITF should have the ability to maintain versions of test case executions.

Architecture

ITF has a test engine component that has the utilities to add/create/modify test cases, execute the test cases, and generate execution results. ITF can be implemented with the same middleware
product that is used to build the interfaces that are being tested. This would facilitate seamless integration with the Integrated Development Environment (IDE).

ITF uses a set of tables on a back-end database to store all the Test Cases, Endpoints and Test Execution related information. The advantage of storing all the test case information on a database is that the developer can access the repository from any location and it is not tightly coupled to one particular workstation. This makes execution of the same test case from different locations feasible. ITF has complete access to all the components of the ESB as does the code. This is because ITF makes use of the same connection parameters and the client libraries as does the code. Also the configuration parameters can be modified on the fly. This facilitates ease of testing on various different environments. Various components of the ESB such as the Enterprise Messaging Services, Java Database Connectivity (JDBC) components, SOAP endpoints etc are readily accessed by ITF inbuilt connection settings.

ITF runs queries against the back-end tables to generate user defined customizable reports. These reports can be test case related or test execution related. ITF has options to provide summary or detailed reports.

**Figure 1. ITF Components**
Design

ITF objects

The four objects used by ITF are TestCase, InputEndpoint, OutputEndpoint and Group. TestCase object defines a Test Case. Table 1 shows all the attributes of the TestCase object. InputEndpoint object defines the type of message interaction between the ITF and the code to be tested on the entry side. Table 2 shows all the attributes of the InputEndpoint object. OutputEndpoint object defines the type of message interaction between the ITF and the code to be tested on the exit side. Table 3 shows all the attributes of the OutputEndpoint object. Group object defines the association of a test case to a test suite. Table 4 shows all the attributes of the Group object.

Table 1. Test Case Object

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>Y</td>
<td>The Name of the Test Case.</td>
</tr>
<tr>
<td>Description</td>
<td>string</td>
<td>Y</td>
<td>Description to describe what is being tested.</td>
</tr>
<tr>
<td>Outcome</td>
<td>string</td>
<td>Y</td>
<td>The Outcome of the Test. Success or Failure.</td>
</tr>
<tr>
<td>InputEndpoint</td>
<td>Object</td>
<td>Y</td>
<td>Input Endpoint described in Table 2.</td>
</tr>
<tr>
<td>OutputEndpoint</td>
<td>Object</td>
<td>Y</td>
<td>Output Endpoint described in Table 3.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Integer</td>
<td>Y</td>
<td>Timeout value for the Test Case.</td>
</tr>
<tr>
<td>ExpectedGALCount</td>
<td>Integer</td>
<td>Y</td>
<td>Expected number of entries in Audit Logging.</td>
</tr>
<tr>
<td>ExpectedGEHCount</td>
<td>Integer</td>
<td>Y</td>
<td>Expected number of entries in Exception Logging.</td>
</tr>
<tr>
<td>Group</td>
<td>Object</td>
<td>Y</td>
<td>The group the test case belongs to. Described in Table 4.</td>
</tr>
</tbody>
</table>

Table 2. InputEndpoint Object

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>Y</td>
<td>The Name of the Endpoint.</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>Integer</td>
<td>Y</td>
<td>The JMS Priority for the request.</td>
</tr>
<tr>
<td>Request</td>
<td>string</td>
<td>Y</td>
<td>The request message for the test case.</td>
</tr>
<tr>
<td>Reply</td>
<td>Object</td>
<td>Y</td>
<td>The reply message for the test case.</td>
</tr>
</tbody>
</table>
Table 3. OutputEndpoint Object

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>Y</td>
<td>The Name of the Endpoint.</td>
</tr>
<tr>
<td>Body</td>
<td>string</td>
<td>Y</td>
<td>The body of the message at the endpoint.</td>
</tr>
</tbody>
</table>

Table 4. Group Object

<table>
<thead>
<tr>
<th>Field name</th>
<th>Type</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>Y</td>
<td>The Name of the Test Case Group.</td>
</tr>
</tbody>
</table>

ITF process flow

Figures 2, 3 and 4 detail the steps involved in defining a Test Case/Group/Endpoint, executing a Test Case/Group and generating a Test execution report by ITF.

Figure 2. Create TestCase/Group/Endpoint

Input

1. Test Case Name
2. Test Case Description
3. Groups to which the test case belongs
4. Input message and output message. It consist of:
   1. Endpoint Reference
   2. Request and Reply message for WS Endpoint
   3. Body for JMS Endpoint
   4. Supersede any other Endpoint fields
5. Expected GAL/GEH count
6. Expected outcome – Success, Failure or Error
7. Timeout – time the framework waits for on output endpoint after sending message on input endpoint

Sample Input

```xml
  <TestCase>
    <def:Name>Sample</def:Name>
    <def:Description>Sample TestCase for illustration</def:Description>
    <def:Outcome>S</def:Outcome>
    <def:InputEndpoint>
      <def:Name>SampleWSEndpoint</def:Name>
    </def:InputEndpoint>
  </TestCase>
</Input>
```
<def:JMSPriority>7</def:JMSPriority>
<def:Request><![CDATA[<?xml version="1.0" encoding="UTF-8"?>
<def:Reply>Some reply with UNIQUE_TRANSACTIONID_REPLACEMENT</def:Reply>
</def:InputEndpoint>
<def:OutputEndpoint>
<def:Name>SampleJMSEndpoint</def:Name>
<def:Body>Some body with UNIQUE_TRANSACTIONID_REPLACEMENT</def:Body>
</def:OutputEndpoint>
<def:Timeout>10</def:Timeout>
<def:ExpectedGALCount>1</def:ExpectedGALCount>
<def:ExpectedGEHCount>0</def:ExpectedGEHCount>
<def:Group>
<def:Name>Sample</def:Name>
</def:Group>
</def:TestCase>
</def:Input>

Steps:

1. Get input.
2. Check input against existing definitions in database.
3. Create endpoints as needed.
4. Create groups as needed.
5. Create test cases.
6. Create relationship between group and the test cases.
7. Return a summary showing successful and failed endpoint, group and test-case.

Output: Test Case/Group/Endpoint Created

Figure 3. Execute TestCase/Group

Input:

1. Test Case Name
2. Group Name

Sample Input

<?xml version = "1.0" encoding = "UTF-8"?>
<Input>
<TestCases>
  <TestCase xmlns = "http://www.ibm.com/ams/common/unittestframework/execution">
  <Name xmlns = "http://www.ibm.com/ams/common/unittestframework/definition">
    testcase1</Name>
  </TestCase>
</TestCases>
</Input>
Main Steps:

2. Get all the test cases from database based on input.
3. Run Process Execute for each test case.
4. Collect result for each test case.
5. Update the summary in the database.
6. Returns summary and detail back.

Sub Steps:

1. Check the type of message, input and output endpoint.
2. Start listener of output endpoint.
3. Send message on input endpoint.
4. If input endpoint is WS, wait for response.
5. If output endpoint is WS, send back response when message is received.
6. Compare message on output endpoint with expected message if compare flag is true. Please refer to validation component below for more details on how ITF does the comparison.
7. If applicable and compare flag is true, compare message on input response.
8. Get the count from GAL and GEH database and compare with the expected count.
9. Return the results back.

Output: Test Case/Group executed and the execution details have been saved to the database.

Validation Component:

Expected results could potentially contain dynamic values such as timestamp, correlation ID etc. In such cases the dynamic part of the result is replaced with an "exclusion" pattern. This forces ITF to do a regular expression comparison of the actual output and expected output. Without this feature, the comparisons would always result in a mismatch. For cases where there are no dynamic components in the output result, ITF does a regular string comparison.

Figure 4. Generate Test Execution Report

Input:

1. Test Case Name
2. Group Name
3. Version Number
Sample Input

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<Input>
  <Report>
    <TestCase xmlns = "http://www.ibm.com/ams/common/unittestframework/report">
      <Name xmlns = "http://www.ibm.com/ams/common/unittestframework/definition">testcase1</Name>
    </TestCase>
  </Report>
</Input>
```

Steps:

2. Query the ITF database to obtain the test execution details.
3. Generate a file with the execution summary.

Output: Execution summary report is generated.

**ITF test cycle**

Figure 5 shows the ITF Test Cycle. The developer/tester identifies the test cases, test groups and endpoints and defines them using the ITF specified input templates. ITF creates Test Cases, Test Groups and Endpoints and stores them to the database. During Test Case execution, the code being tested (CBT) receives input from ITF through one of the pre defined input endpoints. After execution, the CBT sends the response back to ITF on one of the pre defined output endpoints. Optionally, for every input from ITF, the CBT might send back an input response and for every output to ITF, the code might expect an output response from ITF.
Based on the SOI integration pattern, four different types of message flows between the CBT and ITF are chosen for illustration.

1. JMS to JMS
2. Web service to web service
3. JMS to web service
4. Web service to JMS

**JMS to JMS**

The input and the output endpoints of the CBT are JMS queues. Hence none of the optional responses are required. An input message is sent on the input queue of the CBT and the output is sent to the output queue. ITF compares the message read from the output queue to the expected output message associated with the test case being executed. Based on the comparison, ITF determines if the test case was a success or a failure.
Figure 7. JMS to JMS Interaction

Web service to web service
The input and the output endpoints of the CBT are Web Services. Hence both the optional responses are required. An input message is sent on the input queue of the CBT and ITF waits for a response back from the code on the replyTo queue of the request. The CBT generates an Input response to the request and this is captured by ITF and compared with the expected response. ITF mimics the service the code is invoking by sending back a pre-defined response to the output replyTo queue that is set by the CBT. The response is eventually sent back to the input replyTo queue by the CBT. This response is captured by ITF. ITF compares the input response with the expected response. Based on the comparison, ITF determines if the test case was a success or a failure.

Figure 8. Web service to web service interaction

JMS to web service
The CBT does not generate an input response. It expects only an output response. ITF sends the input message to the input queue. The CBT processes the input, and sends the request on the output queue. This is captured by the ITF and compared with expected output. The ITF mimics the service the code is invoking by sending back pre-defined output response on output replyTo queue set by CBT.

Figure 9. JMS to web service Interaction

Web Service to JMS
The CBT generates a response to the input request and does not expect a response for the output message. The Input message is sent on the input queue, the sender waits for the response back on the input replyTo queue. Once the CBT processes the request, it sends the message on the
output queue, which is captured by the ITF and compared with expected output. The CBT is also expected to generate response back on input replyTo queue. This response is captured by the ITF. The ITF compares the input response with the expected response. The CBT generates a response to the input request and does not expect a response for the output message. The Input message is sent on the input queue, the sender waits for the response back on the input replyTo queue. Once the CBT processes the request, it sends the message on the output queue, which is captured by the ITF and compared with expected output. The CBT is also expected to generate response back on input replyTo queue. This response is captured by the ITF. The ITF compares the input response with the expected response.

**Figure 10. Web Service to JMS Interaction**

![Web Service to JMS Interaction Diagram]

**Database Design**

1. ITF uses 6 tables to stores all the necessary information.
2. Table UT_DEF_TEST_CASE contains all the test case related information.
3. Table UT_DEF_GROUP contains all the test case group related information.
4. Table UT_DEF_ENDPOINT contains all the end point related information.
5. Table UT_DEF_CASE_GROUP_REL contains all the test case – group relation information.
6. Table UT_EXE_SUMMARY contains all the test case execution summary related information.
7. Table UT_EXE_DETAILS contains all the test case execution details related information.

**Figure 11. ER Model**

![ER Model Diagram]
### Table 5. UT_DEF_TEST_CASE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>varchar2</td>
<td>256</td>
<td>GUID Primary Key(PK)</td>
</tr>
<tr>
<td>name</td>
<td>varchar2</td>
<td>1024</td>
<td>Unique name given to each test case</td>
</tr>
<tr>
<td>description</td>
<td>varchar2</td>
<td>4000</td>
<td>Description of the test case</td>
</tr>
<tr>
<td>created_by</td>
<td>varchar2</td>
<td>1024</td>
<td>User who created the test case</td>
</tr>
<tr>
<td>create_date</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp at creation</td>
</tr>
<tr>
<td>expected_outcome</td>
<td>char</td>
<td>1</td>
<td>Flag to indicate expected outcome of the test. S means success, F means failure and E means error. Success means everything worked, output matches expected output, number of messages in GAL and GEH matches expected count. Failure indicates test case was executed but expected result didn’t match the actual result. Error means test was either timed out or some other error was encountered. If this indicator is F, the framework will skip matching output with expected output. If this value is E the framework is expected to timeout. In all cases framework expected the number of messages for GAL and GEH to match.</td>
</tr>
<tr>
<td>input_message</td>
<td>clob</td>
<td>-</td>
<td>Input message to endpoint which needs to be tested. In the input message UNIQUE_TRANSACTIONID_REPLACEMENT word will be replaced with transaction id.</td>
</tr>
</tbody>
</table>
output_message | clob | - | Expected output message to be compared with the actual output message. In the expected output message UNIQUE_TRANSACTIONID_REPLACEMENT will be replaced with transaction id. | - |

| input_endpoint | varchar2 | 256 | ID referring to ut_def_endpoint table | Foreign Key (FK) |
| output_endpoint | varchar2 | 256 | ID referring to ut_def_endpoint table | Foreign Key (FK) |
| timeout_interval | number | - | Interval in millisec. If within this interval output is not received, the testcase will be marked as failure. | - |
| expected_gal_count | number | - | Number of entries expected in the GAL with the transaction id created. | - |
| expected_geh_count | number | - | Number of entries expected in the GEH with the transaction id created. | - |

**Table 6. UT_DEF_GROUP**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
<th>DBA info</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>varchar2</td>
<td>256</td>
<td>GUID</td>
<td>Primary Key(PK)</td>
</tr>
<tr>
<td>name</td>
<td>varchar2</td>
<td>1024</td>
<td>Unique name of the group</td>
<td>Unique</td>
</tr>
<tr>
<td>description</td>
<td>varchar2</td>
<td>4000</td>
<td>Description of the test group</td>
<td>-</td>
</tr>
<tr>
<td>created_by</td>
<td>varchar2</td>
<td>1024</td>
<td>User who created the group</td>
<td>-</td>
</tr>
<tr>
<td>create_date</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp at creation</td>
<td>default system timestamp</td>
</tr>
</tbody>
</table>

**Table 7. UT_DEF_ENDPOINT**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
<th>DBA info</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>varchar2</td>
<td>256</td>
<td>GUID</td>
<td>Primary Key(PK)</td>
</tr>
<tr>
<td>name</td>
<td>varchar2</td>
<td>1024</td>
<td>Unique name of the endpoint</td>
<td>Unique</td>
</tr>
<tr>
<td>description</td>
<td>varchar2</td>
<td>4000</td>
<td>Description of the endpoint</td>
<td>-</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Size</td>
<td>Description</td>
<td>DBA info</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------</td>
<td>------</td>
<td>--------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>created_by</td>
<td>varchar2</td>
<td>1024</td>
<td>User who created the endpoint</td>
<td>-</td>
</tr>
<tr>
<td>create_date</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp at creation</td>
<td>default system timestamp</td>
</tr>
<tr>
<td>type</td>
<td>varchar2</td>
<td>256</td>
<td>JMS or WS</td>
<td>-</td>
</tr>
<tr>
<td>message_type</td>
<td>varchar2</td>
<td>256</td>
<td>Bytes or text</td>
<td>-</td>
</tr>
<tr>
<td>encoding</td>
<td>varchar2</td>
<td>256</td>
<td>Encoding of the byte messages</td>
<td>Default UTF8</td>
</tr>
<tr>
<td>compare</td>
<td>char</td>
<td>1</td>
<td>Y or N</td>
<td>-</td>
</tr>
<tr>
<td>destination_queue</td>
<td>varchar2</td>
<td>1024</td>
<td>Destination queue for the request</td>
<td>-</td>
</tr>
<tr>
<td>reply_to_queue</td>
<td>varchar2</td>
<td>1024</td>
<td>Reply to Queue for the reply</td>
<td>-</td>
</tr>
<tr>
<td>priority</td>
<td>number</td>
<td>-</td>
<td>Priority for the request</td>
<td>-</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>varchar2</td>
<td>1024</td>
<td>The delivery mode for the message</td>
<td>-</td>
</tr>
<tr>
<td>correlation_id</td>
<td>varchar2</td>
<td>1024</td>
<td>Correlation ID for the message</td>
<td>-</td>
</tr>
<tr>
<td>timeout</td>
<td>number</td>
<td>-</td>
<td>Timeout value</td>
<td>-</td>
</tr>
<tr>
<td>soap_action</td>
<td>varchar2</td>
<td>1024</td>
<td>Soap Action for webservice</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 8. UT_DEF_CASE_GROUP_REL**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
<th>DBA info</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>varchar2</td>
<td>256</td>
<td>Unique ID. GUID will be used.</td>
<td>PK</td>
</tr>
<tr>
<td>created_by</td>
<td>varchar2</td>
<td>1024</td>
<td>User who created the relationship</td>
<td>-</td>
</tr>
<tr>
<td>create_date</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp at creation</td>
<td>Default system timestamp</td>
</tr>
<tr>
<td>def_group_id</td>
<td>varchar2</td>
<td>256</td>
<td>ID referring to ut_def_group table</td>
<td>FK</td>
</tr>
<tr>
<td>def_test_case_id</td>
<td>varchar2</td>
<td>256</td>
<td>ID referring to ut_def_test_case table</td>
<td>FK</td>
</tr>
</tbody>
</table>

**Table 9. UT_EXE_SUMMARY**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
<th>DBA info</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>varchar2</td>
<td>256</td>
<td>GUID</td>
<td>PK</td>
</tr>
<tr>
<td>user_name</td>
<td>varchar2</td>
<td>1024</td>
<td>User who executed the test case</td>
<td>-</td>
</tr>
<tr>
<td>description</td>
<td>varchar2</td>
<td>4000</td>
<td>Description of the test cases executed</td>
<td>-</td>
</tr>
<tr>
<td>Column Name</td>
<td>Type</td>
<td>Size</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>external_reference</td>
<td>varchar2</td>
<td>1024</td>
<td>Can have details of defect number etc.</td>
<td></td>
</tr>
<tr>
<td>test_request</td>
<td>clob</td>
<td>-</td>
<td>Entire request for testing</td>
<td></td>
</tr>
<tr>
<td>start_time</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp when execution started</td>
<td></td>
</tr>
<tr>
<td>end_time</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp when execution ended</td>
<td></td>
</tr>
<tr>
<td>success_count</td>
<td>number</td>
<td>-</td>
<td>Number of test cases whose outcome was success on execution.</td>
<td></td>
</tr>
<tr>
<td>failure_count</td>
<td>number</td>
<td>-</td>
<td>Number of test cases whose outcome was failure on execution.</td>
<td></td>
</tr>
<tr>
<td>error_count</td>
<td>number</td>
<td>-</td>
<td>Number of test cases whose outcome was error on execution.</td>
<td></td>
</tr>
<tr>
<td>total_count</td>
<td>number</td>
<td>-</td>
<td>Total number of test cases.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 10. UT_EXE_DETAILS**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>varchar2</td>
<td>256</td>
<td>GUID</td>
</tr>
<tr>
<td>exe_summary_id</td>
<td>varchar2</td>
<td>256</td>
<td>ID referring to ut_exe_summary table</td>
</tr>
<tr>
<td>def_test_case_id</td>
<td>varchar2</td>
<td>256</td>
<td>ID referring to ut_def_test_case table</td>
</tr>
<tr>
<td>actual_outcome</td>
<td>char</td>
<td>1</td>
<td>S, F or E. This is actual outcome of the test. Refer to description of ut_def_test_case.outcome</td>
</tr>
<tr>
<td>test_outcome</td>
<td>char</td>
<td>1</td>
<td>S, F or E. Actual outcome is matched with expected outcome and this is final result of test case outcome. Refer to description of ut_def_test_case.outcome</td>
</tr>
<tr>
<td>test_input</td>
<td>clob</td>
<td>-</td>
<td>Actual input</td>
</tr>
<tr>
<td>test_output</td>
<td>clob</td>
<td>-</td>
<td>Actual output</td>
</tr>
<tr>
<td>start_time</td>
<td>timestamp</td>
<td>-</td>
<td>Timestamp when test case started</td>
</tr>
</tbody>
</table>
### Conclusion

ITF is an efficient tool to perform various critical tasks in the testing phase of an SOI implementation. It can be quickly built with and seamlessly integrated into most middleware integration products. Large scale cost benefits are observed when ITF is embraced by the development/testing team. Teams using ITF consistently generated lower number of defects in comparison to those that did not.
Related topic

• Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)