

PegaRULES Process Commander Development

UnitTest RuleSet Documentation and Manual

An xUnit testing-framework for PegaRULES Process Commander

Paul Evans

07/22/2006

Revision Chart

The following chart lists the revisions made to this document. Use this to describe the edits each time this document is re-published (both in draft and final forms). The description should include as many details of the edits as possible, as well as identify the reviewers who requested the edits.

Date	Author	Description
02/01/2006	Paul Evans	Initial draft
07/22/2006	Paul Evans	Minor updates

Contents

Introduction.....	5
Overview.....	5
Motivations.....	5
Repeatability.....	5
Immediate Feedback of Test Results.....	5
Test-Driven Development (TDD).....	6
Regression Testing.....	6
Industry Standards-based (xUnit).....	6
Documentation.....	7
Rule Set Prerequisites.....	9
Description of Class Structure.....	10
UnitTest-.....	10
UnitTest-Assert.....	10
UnitTest-TestActivity.....	10
UnitTest-TestRunMetadata.....	10
Rule-Utility-Function-Test.....	10
Description of Supporting Rules.....	11
Activities.....	11
UnitTest-.RunAllUnitTests.....	11
UnitTest-Assert.AssertTrue.....	11
UnitTest-Assert.AssertFalse.....	11
UnitTest-Assert.AssertTrue_prop.....	11
UnitTest-Assert.AssertFalse_prop.....	11
UnitTest-Assert.AssertEquals_str.....	11
UnitTest-Assert.AssertEquals_int.....	11
UnitTest-Assert.AssertEquals_dbl.....	11
UnitTest-Assert.AssertEquals_DecisionTree.....	12
UnitTest-Assert.AssertEquals_DecisionTable.....	12
UnitTest-Assert.AssertEquals_MapValue.....	12
UnitTest-Assert.Fail.....	12
UnitTest-Assert.AssertGreaterThan_dbl.....	12
UnitTest-Assert.AssertGreaterThan_int.....	12
UnitTest-Assert.AssertGreaterThanEqualTo_dbl.....	12
UnitTest-Assert.AssertGreaterThanEqualTo_int.....	12
UnitTest-Assert.AssertLessThan_dbl.....	12
UnitTest-Assert.AssertLessThan_int.....	12
UnitTest-Assert.AssertLessThanEqualTo_dbl.....	13
UnitTest-Assert.AssertLessThanEqualTo_int.....	13
UnitTest-Assert.AssertPageExists.....	13
UnitTest-Assert.AssertPageNotExists.....	13
UnitTest-Assert.AssertHasMessages.....	13
UnitTest-Assert.AssertHasNoMessages.....	13
UnitTest-Assert.AssertFalse_prop.....	13
UnitTest-Assert.AssertTrue_prop.....	13
UnitTest-Assert.AssertNull_str.....	13

UnitTest-Assert.AssertNotNull_str.....	13
UnitTest-Assert.AssertNull_PgProp.....	13
UnitTest-Assert.AssertNotNull_PgProp.....	14
Connect SQL.....	14
UnitTest-TestActivity.GetTestActivities	14
Data-Gadget.RunUnitTests	15
UnitTest-.TestResults.....	16
Usage Overview	17
“Test” Rule sets.....	17
“Test” Classes	17
“UT-” Activities.....	17
Ideas for Enhancement.....	23
Automated Test Run	23
Initiate Test Run from External System.....	23
Test-Code Generation	23
Resources	24
Test-driven Development.....	24
xUnit Testing Framework Pattern.....	24

Introduction

Overview

In an effort to improve the quality of applications developed using PegaRULES Process Commander, I have developed a rule set for the explicit purpose of unit testing. This rule set contains the basic class structure and rules that system architects, process architects and administrators can use to create repeatable test activities to unit test various rule types within PRPC. This document describes the motivations behind the development of this framework, how to install and use the framework with screenshots included for clarity. The last section of this document describes possible enhancements that can be made to this framework.

Motivations

The motivation for creating this unit testing framework is many-fold:

Repeatability

PRPC provides out-of-the-box functionality for unit testing various rule types. However such unit tests are not persisted to the rule base and therefore can not easily be repeated on demand. In addition, unit tests can only be created and executed by developers intimately familiar with the rules being tested. This greatly deters the ability to regression test code units within a PRPC-based application. Since the out-of-the-box pattern for running unit tests are not repeatable (they typically consist of hitting the “play” button which launches a run-dialog with the ability to set up clipboard pages necessary for the rule in question to execute properly), it cannot easily be executed in a repeatable fashion. Unit tests created using the UnitTest rule set are implemented as activities, and therefore persisted to the rule base. At any time a user, with the appropriate access group, can execute a test activity and know immediately if the test succeeded or failed. In addition a developer unfamiliar with the rules (perhaps a new addition to the development team for the project) can check to see if a rule passes its unit tests by simply running the corresponding unit test activity. Even non-technical members of a team can check if a rule is working properly by running the unit test activities. Unit test activities developed using this framework can simply be executed; no clipboard pages need to be manually created and initialized with data; this task is done by the unit test activity.

Immediate Feedback of Test Results

Unit tests developed using this framework provide the user running the test immediate feedback if the test failed or succeeded along with a reason-message in the event of a failure. In contrast, the out-of-the-box mechanism for running unit tests forces the user to manually analyze values displayed on the run-dialog or on the clipboard that result from running the unit test. This is time-consuming and error prone; especially for users running the tests who are not the original author of the rule being tested. In some cases,

given the complexity of the rule being tested, it may be exceptionally difficult for new developers to successfully unit test a rule using the out-of-the-box mechanism.

Test-Driven Development (TDD)

Unit testing is a building block of test-driven development (TDD). This is important since experience has shown many PRPC projects typically live in rapid-development or iterative-development environments. It has also been reported Pegasystems' sales team touts the ability of PRPC to be utilized in an iterative, or agile environment. Test-driven development is a technique heavily emphasized in agile development circles in order to improve the quality of the code as well as aid in the actual design of the application. Restated, since unit testing (more specifically, unit tests that are repeatable) is central to an agile development approach, this package greatly enhances the probability for success. For more information on TDD, please refer to the resources section at the bottom of this document.

Regression Testing

In the event of code refactoring, which is commonplace in an agile development model, it is of great value to be able to re-run the unit tests to ensure the refactoring has not "broken" some other part of the system. This unit testing framework ships with a gadget that is included in the standard developer portals (*SysArchDD*, *SysAdminDD*, etc) on both the "Dashboard" and "Manage Rules" tabs that contains a button when clicked will search the rule base for all unit test activities accessible by the current operator, and runs them. A report is then presented showing the user which unit test activities failed or succeeded providing the developer with immediate feedback about whether edits to a rule have broken some other rule within the system.

Industry Standards-based (xUnit)

The design and usage of the rules defined within this framework are based on the testing framework developed by Kent Beck. The design of the UnitTest rule set follows the pattern of the xUnit family of testing frameworks. For more information about xUnit, please refer to the resources-section at the bottom of this document. The following present similarities between this framework and JUnit (<http://www.junit.org>):

- Developers create a "Test" class for each production-class to be tested (in JUnit, this class extends `junit.framework.TestCase`; in this framework, the Test class should direct-inherit from the class it is testing)
- Developers create a "Test" function/activity for each corresponding production function/activity (or any other unit-testable-rule type) to test
- The "Test" function/activity contains "Assert" calls verifying expected values versus actual values (in JUnit, the "Assert" methods are inherited from the `TestCase` class; in this framework, activity-calls are explicitly made to the appropriate `UnitTest-Test.AssertXXX` activity)
- Both frameworks have the ability to batch-run all of the found "Test" functions/activities providing a report of all the failures and successes; JUnit does this using a `TestSuite`; in this framework this is accomplished by using the "Run Unit Tests" portal-gadget

Given the myriad similarities between this framework and JUnit, PegaRULES developers with a Java background with experience with JUnit should experience a relatively small learning curve.

Documentation

In addition to providing the foundation for creating robust and reliable rules, unit test activities written using this framework double as a source of documentation for the developer on how the rule being tested is meant to be used. Because the unit test activity simulates the runtime-state of the clipboard as it would be in the production system, it provides a level of documentation and clarity about the rule being tested that is extremely valuable.

Installation Instructions

To install the UnitTest rule set:

- Log in to PRPC with an account with administrator-access
- Upload the **UnitTest-XX-XX-XX.zip** file located in the **dist/** folder to the application server from the “Administrator” portal workspace
- Load the rules from the **UnitTest-XX-XX-XX.zip** file from the “Administrator” portal workspace
- Modify the appropriate access groups adding the **UnitTest** rule set to the list of available rule sets

Once the developer logs into PRPC with the updated access group, immediately the “Run Unit Tests” portal-gadget will be visible on both the Dashboard and Manage Rules workspaces. The “UnitTest-“ class will also appear in the class explorer.

Rule Set Prerequisites

As of this writing the UnitTest rule set has the following prerequisites:

Pega-ProCom:04-02-33

Pega-RULES:04-02-61

The Pega-RULES is a prerequisite rule set because of the Rule-Utility-Function-Test class that comes with the UnitTest rule set. This class direct inherits from Rule-Utility-Function, which lives in the Pega-RULES rule set; therefore in order for the Rule-Utility-Function-Test class to exist in the UnitTest rule set, the Pega-RULES prerequisite-entry is required.

Description of Class Structure

UnitTest-

Base class of the UnitTest framework.

UnitTest-Assert

Contains the assert-activities used in test activities.

UnitTest-TestActivity

Models a test activity. Contains an activity and connect SQL rule for looking up all of the test activities in the rule base. Each test activity found is stored in an instance of this class within the Code-Pega-List created from the RDB-List method.

UnitTest-TestRunMetadata

Contains properties that store metadata about the full test run of all the test activities.

Rule-Utility-Function-Test

This test class comes with the UnitTest rule set to be used as a container for test activities for testing utility-function rules.

Description of Supporting Rules

Activities

UnitTest-.RunAllUnitTests

Searches the rule base for test activities defined in test classes defined in test rule sets visible to the current operator. Each test activity found is executed and the result of each test run is tabulated and reported to the user in a popup screen. The report contains the following information:

- Number of test activities found and executed
- Number of failures
- Number of successes
- Duration of entire test-run
- The test status (success or failure), test-activity name, test class, ruleset and message of each executed test activity

UnitTest-Assert.AssertTrue

Throws a `java.lang.RuntimeException` if the inputted named when rule evaluates to “False.”

UnitTest-Assert.AssertFalse

Throws a `java.lang.RuntimeException` if the inputted named when rule evaluates to “True.”

UnitTest-Assert.AssertTrue_prop

Throws a `java.lang.RuntimeException` if the inputted Boolean parameter is false.

UnitTest-Assert.AssertFalse_prop

Throws a `java.lang.RuntimeException` if the inputted Boolean parameter is true.

UnitTest-Assert.AssertEquals_str

Throws a `java.lang.RuntimeException` if the two inputted strings (`param.aExpected`, `param.aActual`) are not equal.

UnitTest-Assert.AssertEquals_int

Throws a `java.lang.RuntimeException` if the two inputted integers (`param.aExpected`, `param.aActual`) are not equal.

UnitTest-Assert.AssertEquals_dbl

Throws a `java.lang.RuntimeException` if the two inputted doubles (`param.aExpected`, `param.aActual`) are not equal.

UnitTest-Assert.AssertEquals_DecisionTree

Throws a `java.lang.RuntimeException` if the return value from the inputted named `Rule-Declare-DecisionTree` does not match `param.aExpected`.

UnitTest-Assert.AssertEquals_DecisionTable

Throws a `java.lang.RuntimeException` if the return value from the inputted named `Rule-Declare-DecisionTable` does not match `param.aExpected`.

UnitTest-Assert.AssertEquals_MapValue

Throws a `java.lang.RuntimeException` if the return value from the inputted named `Rule-Obj-MapValue` does not match `param.aExpected`.

UnitTest-Assert.Fail

Throws a `java.lang.RuntimeException` with the inputted message (`param.aMessage`). This is useful if an activity is meant to throw an exception given a set of inputs at a certain step; the step after this would invoke this activity causing a failure; the reason being, the `UnitTest-Assert.Fail` invocation shouldn't be reached if the preceding step threw the exception it was supposed to.

UnitTest-Assert.AssertGreaterThan_dbl

Throws a `java.lang.RuntimeException` if the inputted double value, `param.aValue` is greater than `param.aGreaterThanValue`.

UnitTest-Assert.AssertGreaterThan_int

Throws a `java.lang.RuntimeException` if the inputted integer value, `param.aValue` is greater than `param.aGreaterThanValue`.

UnitTest-Assert.AssertGreaterThanEqualTo_dbl

Throws a `java.lang.RuntimeException` if the inputted double value, `param.aValue` is greater than or equal to `param.aGreaterThanValue`.

UnitTest-Assert.AssertGreaterThanEqualTo_int

Throws a `java.lang.RuntimeException` if the inputted integer value, `param.aValue` is greater than or equal to `param.aGreaterThanValue`.

UnitTest-Assert.AssertLessThan_dbl

Throws a `java.lang.RuntimeException` if the inputted double value, `param.aValue` is less than `param.aLessThanValue`.

UnitTest-Assert.AssertLessThan_int

Throws a `java.lang.RuntimeException` if the inputted integer value, `param.aValue` is less than `param.aLessThanValue`.

UnitTest-Assert.AssertLessThanEqualTo_dbl

Throws a `java.lang.RuntimeException` if the inputted double value, `param.aValue` is less than or equal to `param.aLessThanValue`.

UnitTest-Assert.AssertLessThanEqualTo_int

Throws a `java.lang.RuntimeException` if the inputted integer value, `param.aValue` is less than or equal to `param.aLessThanValue`.

UnitTest-Assert.AssertPageExists

Throws a `java.lang.RuntimeException` if inputted named page does not exist.

UnitTest-Assert.AssertPageNotExists

Throws a `java.lang.RuntimeException` if inputted named page does exist.

UnitTest-Assert.AssertHasMessages

Throws a `java.lang.RuntimeException` if the primary page has no messages on it. Useful for testing validation rules.

UnitTest-Assert.AssertHasNoMessages

Throws a `java.lang.RuntimeException` if the primary page has messages on it. Useful for testing validation rules.

UnitTest-Assert.AssertFalse_prop

Throws a `java.lang.RuntimeException` if the inputted Boolean parameter is true.

UnitTest-Assert.AssertTrue_prop

Throws a `java.lang.RuntimeException` if the inputted Boolean parameter is false.

UnitTest-Assert.AssertNull_str

Throws a `java.lang.RuntimeException` if the inputted string parameter is not null.

UnitTest-Assert.AssertNotNull_str

Throws a `java.lang.RuntimeException` if the inputted string parameter is null.

UnitTest-Assert.AssertNull_PgProp

Throws a `java.lang.RuntimeException` if the named embedded page property is not null.

UnitTest-Assert.AssertNotNull_PgProp

Throws a `java.lang.RuntimeException` if the named embedded page property is null.

Connect SQL

UnitTest-TestActivity.GetTestActivities

The “Browse” tab is populated with a query to retrieve all the activities stored in the rule base that begin with the string: “UT-” in a rule set starting with the string: “Test.” The decision to use a Rule-Connect-SQL instead of an Obj-List invocation was due to the need of tightly controlling the query issued to the rule base in order to pull back the test activities. It is not possible using an Obj-List to specify that a “like” operand be used in the generated SQL.

The screenshot shows the configuration for the 'Connect SQL' activity. The 'Applies To' field is set to 'UnitTest-TestActivity', the 'Package Name' is 'JDBC', and the 'Request Type' is 'GetTestActivities'. The 'Short Description' is 'Retrieves the set of test activities from the rulebase'. The 'Browse' tab is selected, showing the following SQL query:

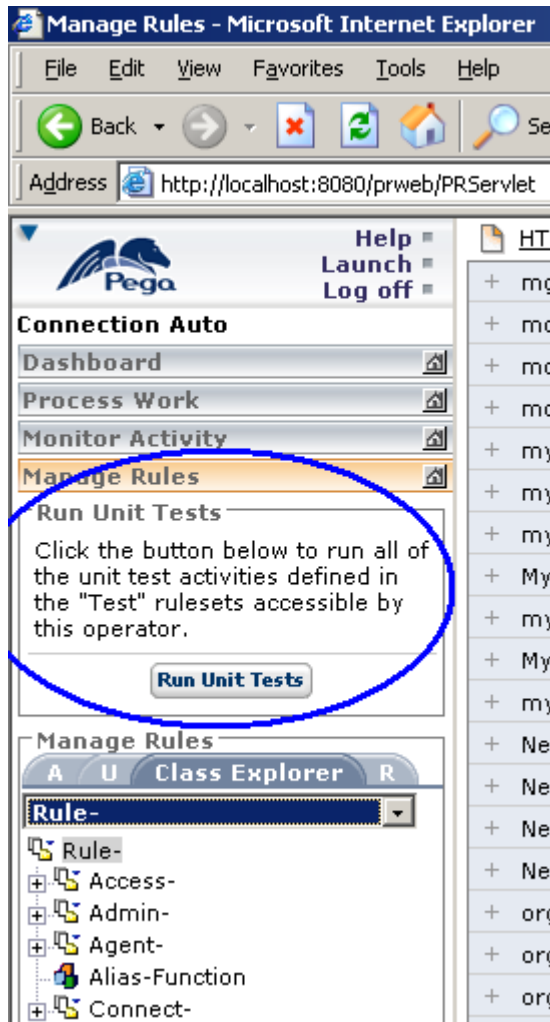
```
select
  pyRuleset as ".Ruleset", pyRulesetVersion as ".RulesetVersion",
  pyActivityName as ".Name", pyClassName as ".Class",
  pyLabel as ".ShortDescription", pzInsKey as ".InstanceKey",
  pxInsName as ".InstanceName", pzInsKey as ".pxInsHandle",
  pxCreateOpName as ".CreateOpName", pxUpdateOpName as ".LastUpdateOpName"
from pr4_rule
where
  pyActivityName like 'UT-%' and
  pyRuleset like 'Test%'
```

Below the query, there is an 'Error Handler Flow' field with a 'Test Connectivity' button.

HTML

Data-Gadget.RunUnitTests

Portal interface for executing all of the unit test activities accessible to the current operator.



UnitTest-TestResults

Rule-Obj-HTML rule displaying the results of the executed test activities. Each row of the test results is click-able; clicking a row opens the test activity.

Example with 1 error in the unit test results:

Unit Test Results:

Number of test activities:	130
Total time to run all test activities (in seconds):	68.365
Number of successes:	129
Number of failures:	1

Test-runs Detail:

Activity Name	Activity Class	Activity Description	Create Operator	Last Update Operator	Success?	Message
UT-CalculatePremiumTestCase3	Allmerica-PL-SQP-Quote-ConAuto-Data-Policy-Sts-NI-Test	Test activity for CalculateAvgDriverFactor activity-rule	Richard Coakley	Richard Coakley	false	Expected value [1431.62] does not match the actual value [1343.1]
UT-DoesNothing	Allmerica-PL-SQP-Quote-ConAuto-Work-Test	Test activity for DoesNothing activity-rule	Paul Evans	Paul Evans	true	
UT-PrepareAddDriver	Allmerica-PL-SQP-Quote-ConAuto-Work-PolSrc-	Test activity for PrepareAddDriver activity-rule	Paul Evans	Paul Evans	true	

Usage Overview

“Test” Rule sets

For each application rule set, create a corresponding “Test” rule set. For example, for an application rule set: “Acme,” create a test rule set: “TestAcme.” The “Test” rule set should have as its prerequisite the target rule set along with the “UnitTest” rule set. For example, the “TestAcme” rule set should list as its prerequisite, “Acme:01-01.” Once created, edit the TestAcme rule set version adding to its required-list the UnitTest:01-01 rule set version. This is required so that test activities defined within the TestAcme rule set can invoke the UnitTest-Assert.AssertXXX activities defined in the UnitTest rule set. Be sure to list all “Test” rule sets in the appropriate access groups.

By storing all test-related rules in “Test” rule sets, they can easily be omitted from builds bound for QA and production environments. In addition access groups can be created if desired that do not include the “Test” rule sets in order to give a “clean” view of the rule base without also viewing the unit test class structures.

“Test” Classes

For each class that contains unit-testable rule(s), create a corresponding “-Test” class that both pattern and direct-inherits from the class it is a test-fixture for. For example, given a class: Acme-Data-Vehicle,” create a test class: “Acme-Data-Vehicle-Test” in the “TestAcme” rule set that pattern and direct-inherits from Acme-Data-Vehicle. This is not a requirement, but a recommended convention to logically segment test activities and production activities. If the size of the class name containing the rules to be tested is more than 52 characters in length, either create a “-Tst” class or, put the test activities in the same class as the production rules.

“UT-” Activities

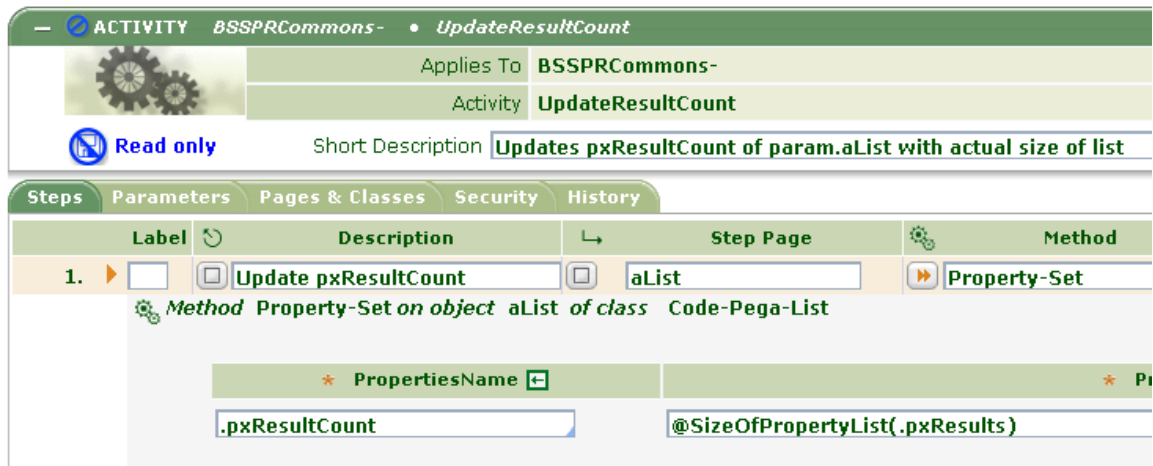
For each unit-testable rule, create a test activity of the form: “UT-RULETESTED” where *RULETESTED* is the name of the rule being tested. For example, for a Rule-Obj-When rule named: IsApproved defined in the Acme-Work-CalculatePremium class, you would create a unit test activity to exercise the paths through the when rules’ boolean expression:

```
Acme-Work-CalculatePremium-Test.UT-IsApproved
```

This activity should be saved in the “TestAcme” rule set.

Examples:

Below is a very simple activity that takes in a Code-Pega-List as input and updates its pxResultCount property to reflect the actual length of its pxResults pagelist. Many times an activity needs to manually filter the pxResults pagelist of a Code-Pega-List page using a Java step or Page-Remove methods; the pxResultCount is not automatically updated as elements are added or removed to the pxResults pagelist property. This activity ensures the param.aList.pxResultCount property reflects the actual size of param.aList.pxResults:



The following is a unit test activity to test the above activity using the `UnitTest` framework. Note that the above activity being tested is defined in the `BSSPRCommons`-class; although the ruleset is not displayed, it is saved in the `BSSPRCommons` rule set. The following unit test activity is defined in the `BSSPRCommons-Test` class within the `TestBSSPRCommons` rule set. Both the `BSSPRCommons-Test` class and unit test activity are saved in the `TestBSSPRCommons` rule set.

The screenshot displays the configuration for a unit test activity named 'UT-UpdateResultCount' within the 'BSSPRCommons-Test' rule set. The interface includes a header with the activity name and a 'Read only' status. Below this, there are tabs for 'Steps', 'Parameters', 'Pages & Classes', 'Security', and 'History'. The 'Steps' tab is active, showing a list of 14 steps with their descriptions, associated pages, and methods.

Label	Description	Step Page	Method
1.	Create a list	TmpListPg	Page-New
2.	Set pxResultCount to 0	TmpListPg	Property-Set
3.	Assert pxResultCount is 0	TmpListPg	Call UnitTest-Assert.A
4.	Call UpdateResultCount		Call UpdateResultCour
5.	Assert pxResultCount is still 0	TmpListPg	Call UnitTest-Assert.A
6.	Create a tmp page	TmpPg	Page-New
7.	Copy TmpPg into TmpListPg	TmpPg	Page-Copy
8.	Call UpdateResultCount		Call UpdateResultCour
9.	Assert pxResultCount is 1	TmpListPg	Call UnitTest-Assert.A
10.	Add 5 more elements to the list	TmpPg	Page-Copy
11.	Assert pxResultCount is still 1	TmpListPg	Call UnitTest-Assert.A
12.	Call UpdateResultCount		Call UpdateResultCour
13.	Assert pxResultCount is 6	TmpListPg	Call UnitTest-Assert.A
14.	Cleanup		Page-Remove

At the bottom of the interface, there are two 'All' buttons and a 'Describe this Rule' button.

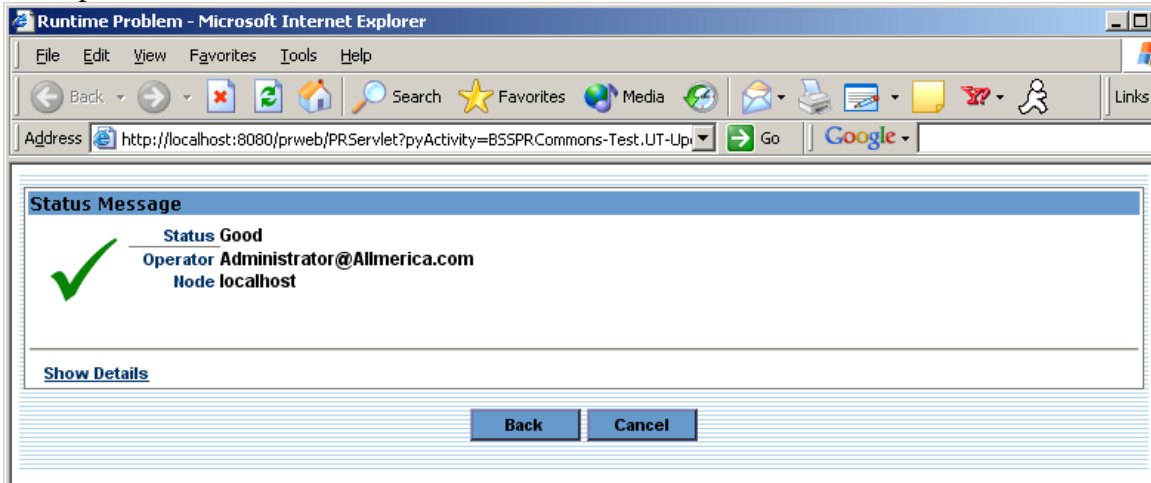
Another screenshot with a few of the steps expanded:

The screenshot displays the 'Steps' tab of the 'UT-UpdateResultCount' activity. The table below summarizes the visible steps and their expanded details.

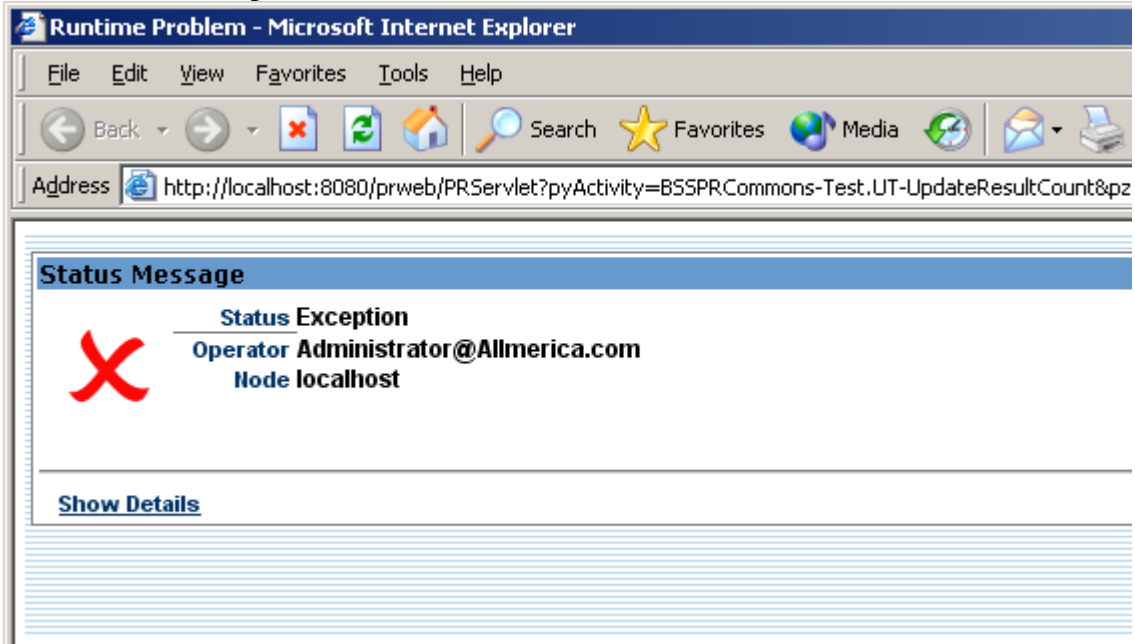
Step #	Description	Step Page	Method						
1.	Create a list	TmpListPg	Page-New						
2.	Set pxResultCount to 0	TmpListPg	Property-Set						
3.	Assert pxResultCount is 0	TmpListPg	Call UnitTest-Assert.A						
Activity AssertEquals_int on object TmpListPg of class UnitTest-Assert Pass current parameter page? <input type="checkbox"/> <table border="1"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>* aExpected</td> <td>0</td> </tr> <tr> <td>* aActual</td> <td>.pxResultCount</td> </tr> </tbody> </table>				Parameter	Value	* aExpected	0	* aActual	.pxResultCount
Parameter	Value								
* aExpected	0								
* aActual	.pxResultCount								
4.	Call UpdateResultCount		Call UpdateResultCour						
5.	Assert pxResultCount is still 0	TmpListPg	Call UnitTest-Assert.A						
6.	Create a tmp page	TmpPg	Page-New						
7.	Copy TmpPg into TmpListPg	TmpPg	Page-Copy						
8.	Call UpdateResultCount		Call UpdateResultCour						
Activity UpdateResultCount on object of class BSSPRCommons-Test Pass current parameter page? <input type="checkbox"/> <table border="1"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>* aList</td> <td>TmpListPg</td> </tr> </tbody> </table>				Parameter	Value	* aList	TmpListPg		
Parameter	Value								
* aList	TmpListPg								
9.	Assert pxResultCount is 1	TmpListPg	Call UnitTest-Assert.A						
10.	Add 5 more elements to the list	TmpPg	Page-Copy						
11.	Assert pxResultCount is still 1	TmpListPg	Call UnitTest-Assert.A						
12.	Call UpdateResultCount		Call UpdateResultCour						
13.	Assert pxResultCount is 6	TmpListPg	Call UnitTest-Assert.A						
Activity AssertEquals_int on object TmpListPg of class UnitTest-Assert Pass current parameter page? <input type="checkbox"/> <table border="1"> <thead> <tr> <th>Parameter</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>* aExpected</td> <td>6</td> </tr> <tr> <td>* aActual</td> <td>.pxResultCount</td> </tr> </tbody> </table>				Parameter	Value	* aExpected	6	* aActual	.pxResultCount
Parameter	Value								
* aExpected	6								
* aActual	.pxResultCount								
14.	Cleanup		Page-Remove						

Notice that in this test activity, the UpdateResultCount activity is fairly well tested. If any of the steps that calls an Assert-activity (step #s: 3, 5, 8, 9, 11, 13) throws an exception, the test fails. Note the UnitTest-Assert.AssertXXX activities follow the same pattern as the JUnit assert-methods; each Assert-activity receives as input an expected-value along with an actual value (or some derivative thereof); if the 2 inputs don't match, a runtime exception is thrown with an appropriate message effectively failing the test. Individual test activities such as this one can be executed directly by clicking the ">" (play) button on the activity rule form, or indirectly by clicking the "Run All Tests" button from the "Run Unit Tests" portal-gadget. The following screenshots illustrate this activity being invoked directly; showing a success-run and a failure-run:

Example of the succeeded unit test:



Example of the failed unit test after intentionally introducing a bug in the activity being tested (details collapsed):



Ideas for Enhancement

The following are ideas regarding how this framework might be enhanced:

Automated Test Run

It would be useful to have the unit tests executed in an automated fashion on a recurring cycle using an agent-queue rule. This would provide an element of continuing integration to the PRPC product-line; in addition workflow could be put in place to allow for the notification of the results of a test run to interested parties via email.

Initiate Test Run from External System

It may provide worthwhile to create service rules to allow the unit tests to be executed by some external system on demand and allow for the reporting of the test results.

Test-Code Generation

Similar to the accelerators, a rule-generation wizard that would generate the test classes and test activities given a target class. For each testable rule present in the target class, a corresponding UT-XXX activity would be created in the –Test class within the specified Test rule set.

Resources

Test-driven Development

<http://www.testdriven.com/>

Site dedicated to TDD

http://en.wikipedia.org/wiki/Test_driven_development

Wikipedia article about TDD

xUnit Testing Framework Pattern

<http://en.wikipedia.org/wiki/XUnit>

Wikipedia article about xUnit