**Application Test Framework**

**Document Version: 1.0**

**Mar 8, 2016**

Table of Contents

1. Introduction 3

1.1 Benefits 3

1.2 Assertions 3

1.3 Design for Test 3

1.3.1 Notifications 4

1.3.2 Scripting Interface 4

1.3.3 Time functions 4

1.3.4 UI Hooks 4

1.4 Disclaimer 4

2. Test Preparation 5

2.1 Background 5

2.2 Installation 5

2.3 Tags 5

2.4 Database 5

2.5 Datasources 6

2.5.1 Test Script 6

2.5.2 Database 6

2.5.3 Database from Spreadsheet 6

2.6 SFCs 7

2.7 Initialization 7

3. Test Execution 8

3.1 Test Frame Module 8

3.2 Test Control Dialog 8

3.3 Client versus Designer Execution 9

4. Scripting Language 10

4.1 Time Specification 10

4.2 Comments 10

4.3 Referencing UI Elements 10

4.4 Data Sources 11

4.4.1 Script 11

4.4.2 Database 11

4.5 Assertions 11

4.6 Variables 11

4.7 Commands 12

4.7.1 Animate 12

4.7.2 Assert 12

4.7.3 Halt 13

4.7.4 Log 13

4.7.5 Notification 13

4.7.6 Poll 14

4.7.7 Run 14

4.7.8 Script 14

4.7.9 Select 15

4.7.10 Show 15

4.7.11 Step 16

4.7.12 Tag 16

4.7.13 TagData 16

4.7.14 TagProvider 16

4.7.15 TagSet 17

4.7.16 Test 17

4.7.17 TimeFactor 18

4.7.18 Timeout 18

4.7.19 Until 18

4.7.20 Wait 18

5. Built-in Variables 19

5.1 Project 19

5.2 User 19

5.3 Performance Variables 19

6. Sample Implementations 20

6.1 Diagnostic Toolkit Recommendations 20

6.1.1 Final Diagnosis 20

6.1.2 Quant Output 20

6.1.3 Recommendation 20

6.1.4 Example 21

7. Custom Tag Provider 23

8. Troubleshooting 24

8.1 Initial Tag Values 24

8.2 Using Mapped Drives on Windows 24

# Introduction

The Application Test Framework, the subject of this document, is an Ignition module designed to exercise and validate Ignition applications, in general. The framework is entirely separate from the application under test, driving its inputs and asserting correct values on its outputs. The application can be a fully functioning production version. Testing is completely scripted. The only manual intervention required is a launch of the application to be tested and the start of the test script.

## Benefits

Experience with past projects has demonstrated a huge advantage in the ability to run and re-run test suites that exercise targeted application features.

An important metric for development is the time interval between a code change and testing it. A primary objective of the test framework is to reduce that time by providing the ability to run the application at an accelerated rate. A 50-500 times execution rate improvement over real-time is targeted.

Some concrete benefits of the testing platform include:

* Validation – use the framework to demonstrate correctness of the application before deployment.
* Event handling – should a production application fail to handle an abnormal situation of some type, the test framework can be used to re-create the failure scenario and prove its correct handling by an updated application.
* Regression - regression tests (which can be the same as the rollout tests) are run whenever the application is modified. They can be used to prove that a code update does not break former correct results. Validation

## Assertions

The test execution script contains embedded assertion definitions. These represent checks for correct application behavior as the test executes. Assertions may be based on:

* Tag values
* Database query results
* Execution of a python script
* List contents or size
* Push notifications
* Text values in UI widgets

## Design for Test

A prime requirement for successful operation of the testing framework is that the application itself be designed-for-test. As a practical matter, this means adherence to the constraints listed below.

### Notifications

If the framework is testing an Ignition module that communicates to clients via push notifications and these notifications are to be used for assertions, then the framework must be configured to listen to and decipher these notifications.

### Scripting Interface

The test framework is able to execute Python scripts that take advantage of any Gateway scripting that might be built into a module or defined in the external Python area of the Ignition installation directory. For a Windows system, the default root directory for external Python is C:\Program Files\Ignition\user-lib\pylib.

### Time functions

The testing framework is able to scale any time intervals that it uses via the *timefactor* directive. Typically this appears in a setup script and indicates a speedup factor to be used during testing.

When testing time-aware applications, the hope is that the application will have an interface to scale the time. If not, then testing must be executed at a 1:1 time ratio.

### UI Hooks

One of the features of the test framework is the ability to execute commands on widgets, just as if a user had made a gesture in the user interface. In order to do this, a client window must be open and the test launched from a client dialog. The dialog is used to obtain the component tree which is used, in turn, to find UI components by path.

## Disclaimer

The application test framework and its documentation are works in-progress. At times, this document may describe features not-yet-implemented. In these cases the text will be annotated in *italics*.

# Test Preparation

For a number of reasons, the test environment must be completely separate from production servers. The most important of these reasons is a safety concern - the test must not be allowed to effect any actual control.

## Background

The following artifacts must be constructed prior to test execution:

* test script (or suite of test scripts). Each script should be designed to test specific features of a specific application (Note: script syntax is NOT Excel).
* database image. If the application interacts with a database, a separate database instance should be constructed specifically for testing. This instance should represent the initial state of the application. Use of a production instance for test is not advised.

## Installation

The application-under-test must be installed on a server allocated for testing. Ancillary components must be available in a test-appropriate mode. This may include UDT definitions, tags, tag providers, database instances, and/or modules. The test-framework module must be loaded into the Gateway

Note that, if any SFC-related tests are to be executed, the ILS-SFC module must be installed as it contains scripting interfaces useful for test.

## Tags

Commonly, tags in a production application are driven by OPC servers. In order for the framework to drive their values, these tags must be converted to simple memory tags, The “Tag Replication” dialog available under the “View” menu when the framework is loaded is provided for this purpose.

Tag path definitions within the test scripts do not include the provider. The provider used is either the default provider for the project that launched the test or a provider specified with the *tagprovider* directive.

## Database

There are no explicit commands for script interactions with databases. Use the Ignition *system.sql* functions inside python scripts or assertions for any database select, insert or update statements. The database connection to be used is either the current project default or must be available through a scripting interface. For ILS sequential control or block language testing, the database connection is available through the scripting interface and is dependent on the whether or not the application is in isolation mode.

## Datasources

The framework supports several sources of tag data:

* Test Script
* Database
* Database from spreadsheet

### Test Script

The default location for timed tag values is in the test scripts themselves. The tagset directive specifies the collection of tags to which each time-stamped line of data in the file applies.

Script-based tag input may be used along-side the other mechanisms for introducing tag values. Alternatively it my be turned off via the *DatabaseOnly* checkbox in the test control dialog.

### Database

An alternative method of supplying test values is a database table. This selection applies when there is an existing table or view listing tag values over time. The table must be structured with columns that are to be associated one-to-one with tag paths. An additional column contains a timestamp.

The application is configured for the tag value table in the test control dialog. Once configured, the configuration is saved in a table named “*ColumnTagMap*”. This table is created automatically by the framework and resides inside the database containing the tag values. However, the user is required to explicitly specify the tag paths associated with the table columns.

### Database from Spreadsheet

The data-source pull-down selection is labeled SPREADSHEET, but, in reality, applies to an alternate database structure designed for holding tag value information originally supplied in a spreadsheet. As with the previous data-source, test values are stored in a database table with columns that are associated one-to-one with tag paths, plus an additional timestamp column. Unlike the previous the tag-path-to-column mapping is not editable. It is derived from the original spreadsheet that contained the data,

Output tags are configured separately. A second table may be used to collect time-stamped output during a test run. The expectation is that writing is accomplished with a special tag writer supplied by the framework for this purpose. The output table and tag-to-column mappings may be configured directly from the test dialog.

## SFCs

As with databases, there are no explicit commands involving Sequential Function Charts. The ILS SFC module has been instrumented with Python-accessible features for full test control.

## Initialization

Unless scripts do this ...

# Test Execution

## Test Frame Module

Load test framework module into the Gateway. Once this is done, application control selects appear in the “View” menu in either Designer or Client scopes. Essential configuration includes selection of the script files used to control test execution.

## Test Control Dialog

When the test-frame module is loaded, a new selection appears at the bottom of the *View* menu in the Designer. A similar selection appears in a Client menu bar.

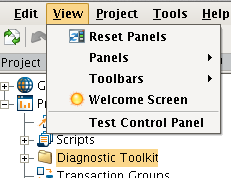


Figure 1 - Launch Menu

Use this selection to launch the control panel. Note that once a test is running, the control panel can be dismissed at will and be brought back as desired without affecting the running test.

The top section of the dialog contains links to file selection dialogs that allow configuration of the various files and directories incorporated by the framework. Once defined, the system will "remember" the location and there is no need to revisit these setting on subsequent executions of the same test.

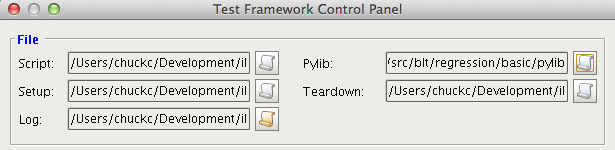


Figure 2 - File Selections

The selection categories are as follows:

* Script - a file path to the test script that is to be run
* Pylib - a directory path to the root location for Python test extensions. The script aliases used to identify the individual scripts reference from this loction.
* Setup - a file path to a test script that will be run prior to the main script. While this script does not differ syntactically from the main test script, it is expected that a setup containing alias and tag-set definitions may be common to a large number of individual scripts.
* Teardown - file path to a test script that will be run after the main script. It is the intent that this script return the application to its default state, undoing any changes (e.g. time scaling) that may have been performed in the setup.
* Log - a file path to a location for recording a detailed log of the test execution. If the file does not pre-exist it will be created. Any existing logs will be cleared. The control dialog does provide access to a test result summary, but the log file contains more detail.

## Client versus Designer Execution

There is no difference between launching a test suite from a Designer window or launching it from a Client. The actual test execution takes place in the Gateway and the two launch mechanisms are equivalent.

Beyond launching, there are several execution differences between Designer and Client scope as some of the test directives are scope-specific.

The following directives have an affect only in the Designer:

* Display a workspace by selection in the Designer navigation tree.

These directives require an open Client:

* Assert presence of a Vision widget
* Selection of a Vision button
* Data entry into Vision text box

# Scripting Language

The files that drive the application testing are simple text scripts, editable in any text editor. Sections of a script file that are used purely to drive tag values can be pasted directly from Excel. The first column of each row in the script is a timestamp.

The script syntax is formally described in an ANTLR text file listed in the appendix.

## Time Specification

For those test directives that must execute at a specific time or after a time delay, there are two mechanisms for specifying that time interval. Both varieties appear at the beginning of the line that hold the time-specific directive.

The simplest form specifies a delay in seconds or minutes (test-time). For example:

wait: 2.3 seconds

Various spellings of time units are recognized. For example seconds may be: seconds, second, secs, or sec. In addition, the parser recognizes all capital or title-case versions.

The second form places a timestamp in the first column. The timestamp consists of an optional date followed by a time. The date is of the form: yyyy/MM/dd. If the date is missing the most recently specified day (or today) is assumed.

Time is in the form HH:mm:ss format (the seconds are optional). It is perfectly acceptable for this time to correspond to a recorded historical time. For the purposes of operating the test framework, only the elapsed time between entries is important.

Date and time are separated by whitespace. If the date is missing and a subsequent time is less than the previous, the test framework will assume rollover to the next day.

Timestamps are used to time data changes, properly schedule standins for operator interactions, and delay assertions so that they occur at the time intended. If a statement’s timestamp is prior to a preceding timestamp, the statement will be executed with no delay. No error will be reported.

## Comments

Commentary is permitted at any point in the file. Comments start with a ‘#’ and are terminated by a new-line character. Comments are ignored by the script processor.

## Referencing UI Elements

blah, blah.

## Data Sources

Tag values may either be included in-line in the test script, read from a database, *csv file or directly from a tag historian (HDA)*. No matter which data source is referenced, the test file is the master control. The database table is read in parallel using only those rows that occur within the test-time specified in the control file. Data source configuration is provided on the main control screen.

### Script

Tag values may be supplied in-line in the same file as assertions and other commands. This option is always available even if external sources are used. The TagSet command may be used to define a set of tags for which values will be supplied. The tags may be either a tag path (without provider) or an alias to a pre-defined path. Follow this directive with time-stamped data lines, either space or comma-delimited. Values will be written to corresponding memory tags.

### Database

If a database is used, the table for tag values must contain a column for each tag, plus one for a timestamp. Column names must correspond to tag aliases. Any column name-to-alias mismatches will be ignored.

A database tag source is specified in the test control dialog. Using this dialog, specify a database connection and table for tag values. Tag values created from this database are assigned the test time specified in the column holding the timestamp. When using a tag value database, timestamps both in the database and control file must contain date information.

## Assertions

The ability to control the user interface requires that the application be constructed with special “hooks” that indicate when components, especially windows and tabbed panes, become visible. Within the test framework a variety of *assertion* statements are provided for this purpose. Pass/fail conclusions may be made on the basis of tag values, notifications (Gateway to client/designer communications) and python script results.

## Variables

Variables may be used to simplify a script and make it more readable. Variables are indicated by a name starting in ‘$’. Once defined, a variable can be use anywhere in the current script or subsequent scripts. A variable retains its value as of the time it was defined. Variable names must start with an alpha character followed by an arbitrary number of alphanumeric characters, dashes or underscores.

diagramPath = “some\_path\_to\_a\_diagram”

notification diagramNotification $notificationKey($diagramPath)

NOTE: In the example above, notificationKey is the name of a presumed script variable.

Variables can also be referenced in log messages using the normal ‘$’ syntax.

log: [DATE] [TIME] testing of $diagramPath complete.

## Commands

The list below describes the syntax of individual directives. The command names are case-insensitive.

### Animate

Animate an object in the UI. Currently this command is only effective for SFC charts. The argument is the chart ID.

Animate: chart *id*

### Assert

Perform a check for correctness. The assertion may refer to a tag value, a python script result, a UI component or the most recent value from a push notification. Database queries may be tested against via the Python scripting mechanism. The assert statement is preceded by a timestamp which schedules its application. The assertion may express an exact equality or inequality, or it may express a valid range. Any quoted comment will appear in the log.

Using an alias as an example:

10:01 assert alias = True “test of ...”

10:02:30 assert 0.0 < alias < 6.2 “tag between zero and 6.2”

10:03:01 assert not 100 > alias > 5

10:03:01 assert alias >= 12.

after 1 minute assert alias!= WAITING

where “alias” refers to any of a number of methods for specifying the target value. The alias must have been previously defined with a TAG or NOTIFICATION statement. In the case of a TAG alias, the current tag value is tested, unless the tag value is null. In this case the test is against the quality name.

The assertion may test for the existence of a UI component.

after 1 second assert exists window “window name” “window is open”

after 2 seconds assert exists component “/RootContainer/Panel/TextField” “text box exists”

Alternatively the assertion may test the result of a script execution:

10:03:05 assert script(args) = true “script returns true”

where “script” refers a script alias.

If the alias or script returns a list (of Strings), an assertion may also test membership in that list. E.g.:

10:03:06 assert script(args) contains “ERROR” “script returns true”

where, as before, “script” refers to a script alias. The test succeeds if one of the elements that the script returns is “ERROR”, exactly.

Additionally, if the alias or script returns a list, an assertion may test its length. E.g.:

10:03:07 assert count of script(args) = 4 “script returns 4 elements”

where, as before, “script” refers to an alias for a script that returns a list.

### Halt

Abort a script immediately. This is a debugging aid that allows one to test an early section of script without having to execute the script in its entirety. Importantly, it leaves the code under test in a state that can be inspected. The teardown script does not run after a *halt*.

halt:

### Log

Create a text message to be sent to the log file and logging area in the control dialog. The message may contain markup fields that are filled in by the framework. The recognized markup fields are:

* [AVECPU] – average CPU usage
* [AVEMEM] – average RAM usage ~ mb
* [CPU] – most recent sample of CPU utilization
* [DATE] – today’s date
* [MEM] – most recent sample of memory usage ~ mb
* [TIME] – the current time
* [TDATE] – the current date in “test time”
* [TTIME] – the current time of day in “test time”

log: This is a time stamped message at [DATE] [TIME]

CPU and memory values are read from the Ignition performance tags. Averages are computed during the period of test execution.

### Notification

Define an alias for a push notification for use in an assertion. The syntax of the notification is completely dependent on the Ignition module that posts the notification. (The syntax for ILS modules is described in Appendix B). The command fields are: alias, module ID and push notification key.

notification: output blt P:123-456-er:propname

The notification argument may optionally be specified as the result of a script execution as for example.

notification: output blt getNotificationKey(“diagram”,”block”,”port”)

### Poll

Set the default poll interval for subsequent Until commands. The time unit may be either minutes or seconds

poll: 13.2 secs

### Run

Execute a test in-line. The included test inherits the current environment (such as database, tag provider), but starts off with a new time scope. The file path argument is either a full path name to the script file, or is relative to the directory holding the current script.

run: test-case.txt

### Script

Define an alias for a Python script. The script can access any of the scripting interfaces exposed in the Gateway by modules under test. It can also access any Python in the external python area of the Ignition installation. The second argument is the name of a python module and entry point relative to the *pylib* directory defined in the Test . The module argument must also contain a parentheses-enclosed, comma-separated list of module arguments.

script: resetDiagram diagramProxy.reset(name)

Note: In the above example, diagramProxy is the presumed name of a python module in the pylib directory that is configured in the execution dialog.

When this python is executed in the test script, the command would be:

resetDiagram(name)

or, optionally, with a preceeding timestamp:

09:58:00 resetDiagram(name)

It is also permissible to use a script-path directly in a statement. This is functionally equivalent, but less efficient. The script command compiles the script and saves the compilation. This step then does not have to be repeated if the alias is re-used.

The test framework will inject a “common” dictionary as part of the module’s local scope. This dictionary will appear as the first argument in the call list (in addition to the arguments specified in the *script* statement).

This dictionary is intended for two purposes:

1. The same dictionary object is passed with each call. This allows the Python module to store state information that will persist between invocations.
2. For scripts that are in situations requiring return values (e.g. an assertion), the script result is passed back to the test framework via a “result” member of the dictionary. This value is then used as the return value by the framework after the script completes.

With the exception of the “result” member, the framework does nothing with the dictionary.

For example if an inline script specification (using a notification as an example) is:

notification: someNote notificationKey.construct(diagram,block,port)

The actual code, in notificationKey.py, might be:

def construct(common,diagram,block,port):

notification = ...

common[“result”] = notification

### Select

Perform the equivalent of a component selection by a user. Specifically this method calls the “onClick” method of the specified component.

select: <option> path

where the option may contain one of the following values:

* + *BUTTON – click on a component. The component is identified by its component path. Use “/” to delimit containers.*
  + *MENU – make a selection from the main menu shown in the client. Use a “/” to delimit menu items.*

### Show

Display a specified window on the visible screen. This action may be a necessary precursor to commands that operate on UI widgets displayed on a screen.

show: <option> path

where the option may contain one of the following values:

* + DIAGRAM – open a frame in the Designer as if one had clicked on the specified path in the navigation tree panel. Path elements are separated by “/”.
  + *CHART – open a chart edit frame in the Designer displaying the SFC chart corresponding to the specified path*
  + *WINDOW – open a window in the Client.*

### Start:

Start a secondary (database) test data source. The source itself must be defined through the scripting interface. If preceded by a timestamp, the data player will start playing at that point. If a database player is already running, it will be stopped and restarted. Starting a database player resets test time to the start of the database time column range. This is the only command where it makes sense to go backward in time.

10:03:07 start:

### Step

Annotate logs for the current step within a test. Pass-fail statistics are kept on the basis of test steps.

step: “test step name”

step: test step name

### Stop

Terminate a secondary (database) test data source. The source itself must have been previously started for this to have an effect.

2016/01/01 10:03:07 stop:

### Tag

Define an alias for a tag for use in a assertions or tag set definitions. The tag path does not include a provider specification.

Tag: output MemoryTags/DiagramOutput

### TagData

The command name is not used. Tag data consists of a timestamp followed by comma or space delimited data values. The data values correspond to tags specified by the current tag set.

TagSet: regression

2015/01/05 10:00 0.34 0.45 True

10:01 0.35, 0.46, False # Date optional

10:02:20 0.36,,True # Skip value with consecutive commas

If the target tag is of type date-time, then its value must be set with a quoted string of the form: yyyy/MM/dd hh:mm:ss.

### TagProvider

Define a tag provider to be used for all subsequent tag references. Note that, for ILS applications that are run in isolation mode, the specified provider should be the provider that corresponds to isolation (as opposed to the production provider).

Tagprovider: TestTags

The command may include optional name, value arguments. E.g.:

Tagprovider: TestTags primary=true,mode=test

Available arguments are:

* primary – If true, this provider is to be used to write to tags that are referenced in the test scripts. Default is true
* mode – Defines how the provider sets the timestamp when tags are written. Options are:
  + current – use the current real-time.
  + test – the time is derived from the test file.
  + history – use the time derived from the test file and also write to a custom historical record. This may be used to support plotting of test results. If history is specified, a database and table name are also required.

The default value is “current”.

* db – database connection name to be used for “history” mode providers.
* table – database table name to be used to store history results.

### TagSet

Define (or select a previously defined) set of tags. This definition serves to provide associated tag paths for subsequent tag data directives. If the command is given with no paths specified, then the set of paths from the previous invocation of the set name is used. This behavior allows several sets to be defined in a setup script, for example, and then be simply referenced in the execution script. Command syntax is:

tagset: setname [tagpath ...]

Note that the tagpath may be expressed either as a string or as a constant variable. The tag path list may be either comma- or whitespace-delimited. For example:

Tag: T423 Tags/Control/T423

tagset: myset $T423 Tags/Control/T424

Note that a single tag may be driven directly. The alias name must be used. For example:

T423 = 16.5

### Test

Define the name of the test. In order for test summary statistics to be meaningful, the test name should be specified before any of its steps. This directive is logged.

test: “test name”

test: test name

### TimeFactor

Set the time speedup factor for test execution. The timescale is the ratio of actual time to test time. For example, a timescale value of 4 sets the test to run 4 times faster than in production.

timefactor: 60

### Timeout

Set the default timeout interval for subsequent Until commands. The time unit may be either minutes or seconds. The time is interpreted as test-time.

Timeout: 30 minutes

### Until

Wait until an asserted expression becomes true, checking at a configured poll rate, timing out after a preset interval. This command is treated as a normal assertion, with a timeout condition being handled as an error.

For example, to wait until the status of some chart becomes “complete”:

Until: getStatus($CHART) = “complete” “Status becomes complete” poll: 5 secs timeout: 2 minutes

Poll and timeout clauses are optional. If not present, then values previously set with stand-alone poll or timeout commands are used.

### Wait

Insert a delay into the test. The delay is expressed in either minutes or seconds of test time. If not specified, seconds are assumed.

wait: 22 seconds

# Built-in Variables

The variables described below are provided automatically by the test framework and may be used in test scripts anywhere variables are allowed. Note that variable names are case-sensitive.

## Project

The project from which the test frame is launched is stored in the PROJECT variable.

## User

If the test is launched from a Client screen, the name of the logged-in user will be stored in the USER variable.

## Performance Variables

Instantaneous CPU and memory usage values are available in the CPU and MEM variables. Average usage figures for the entire test sequence are available in AVECPU and AVEMEM. Samples are taken after the execution of each command. Averages are time-weighted. Memory is expressed in megabytes.

# Sample Implementations

The following sections describe specific implementations of the testing framework for various testing purposes.

## Diagnostic Toolkit Recommendations

This section describes a specific implementation of the testing framework for testing the final diagnosis recommendation logic within the ILS Diagnostic Toolkit platform. The extensions take the form of Python in scripts that query the XOM database. There are three entities that need to be interrogated during a test: the final diagnosis, a quant output, and a recommendation.

### Final Diagnosis

The dynamic properties of a final diagnosis can be interrogated using the getFinalDiagnosisProperty script. The properties that can be accessed are: TextRecommendation, Active, and Explanation.

script: getFinalDiagnosisProperty finalDiagnosisProxy.getFinalDiagnosisProperty (Application, QuantOutput, Property, db)

10:03:05 assert getFinalDiagnosisProperty(“CSTR”, “MLR”, “Active”, “XOM”) = true “MLR is Active”

### Quant Output

The dynamic properties of a quant output can be interrogated using the getQuantOutputProperty script. The properties that can be accessed are: FeedbackOutput, FeedbackOutputManual, FeedbackOutputConditioned, OutputLimitedStatus, OutputLimited, OutputPercent, ManualOverride, Active, CurrentSetpoint, FinalSetpoint, DisplayedRecommendation. This can be used to determine the total effect of several final diagnosis on a quant output for an application.

script: getQuantOutputProperty finalDiagnosisProxy.getFinalDiagnosisProperty (Application, QuantOutput, Property, db)

10:03:05 assert getQuantOutputProperty(“CSTR”, “VCC160\_TARGET”, “FeedbackOutput”, “XOM”) = 12.5 “Feedback output =12.5”

### Recommendation

The dynamic properties of a recommendation can be interrogated using the getRecommendationProperty script. The properties that can be accessed are: Recommendation, AutoRecommendation, ManualRecommendation, AutoOrManual. This can be used to determine the specific effect of a single final diagnosis on a quant output.

script: getRecommendationProperty finalDiagnosisProxy.getRecommendationProperty (Application, FinalDiagnosis, QuantOutput, Property, db)

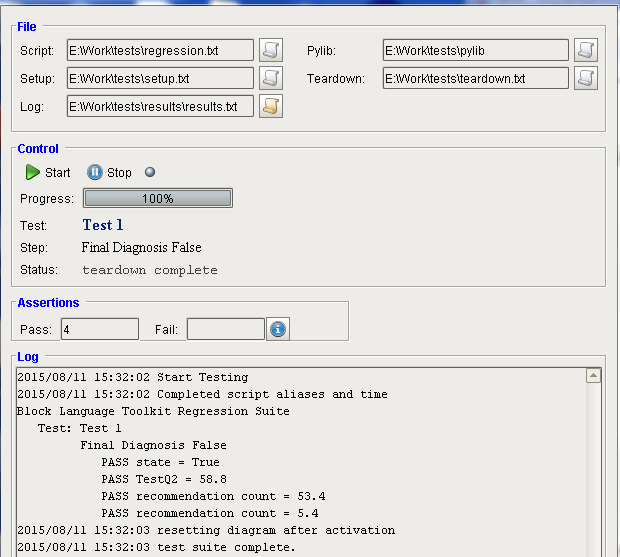
10:03:05 assert getRecommendationProperty(“CSTR”, “MLR”, “VCC160\_TARGET”, “Recommendation”, “XOM”) = 4.3 “Recommendation = 4.3”

Note: The property names are not case sensitive. Refer to the XOM database specification for a description of properties.

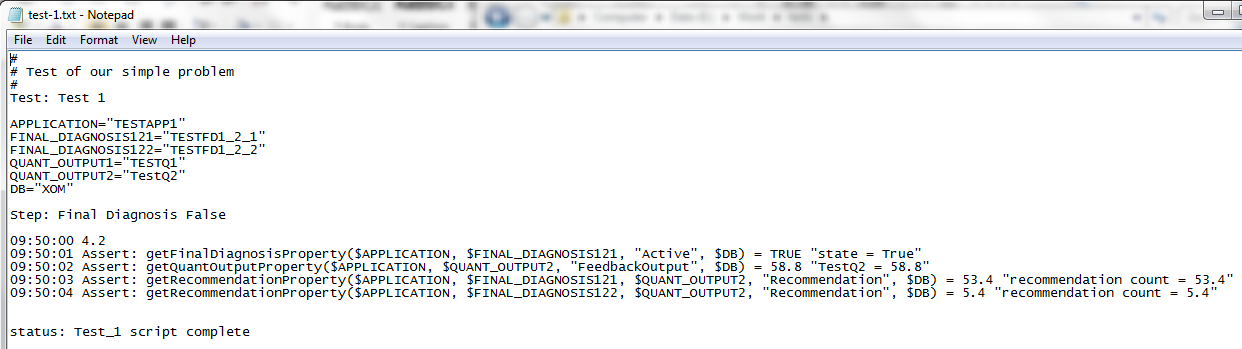
### Example

The following screen shots show the scripts, output and problem under test for a simple scenario. (In this scenario, there was not an actual diagram, but the application, family, final diagnosis and quant outputs were configured in the database.)

The test setup and output is shown below:



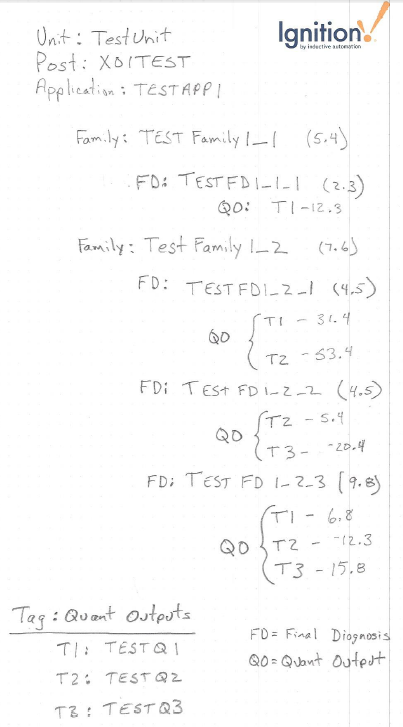
The test script is:



The setup script should contain:



The architecture of the application / family / final diagnosis and quant outputs under test is shown below:



# Custom Tag Provider

The tag replication dialog under the *View* menu provides for the creation of custom tag providers. These are based on the Ignition *SimpleTagProvider* class. Unlike tags under standard providers, tags owned by these providers have the ability to acquire time stamps with past values. This feature may be essential for certain tests that rely on playback of historical information.

The custom providers have limitations however:

* Tags do not persist over a Gateway restart
* Tags may not be added through the Designer browser

# Troubleshooting

The sections below address issues that uses may encounter setting up test scenarios.

## Initial Tag Values

Test scripts are often run repeatedly. If the ending value of a tag is not different from the value set at the start of a script, then when that script is re-run, this tag will not fire an event as it hasn’t changed. A simple workaround is to always start a test with a set of “throw-away” values, then call any initialization code, then start the test.

## Using Mapped Drives on Windows

If the application fails to read script files that reside on a Windows system mapped drive, it may be because the drive was mapped as a local user rather than system. The test framework runs (and reads the files) from within the Ignition Gateway. The Gateway service is probably not running with permissions of the local user and may not recognize the mapping.

A discussion regarding ways to correct the problem may be found at <http://stackoverflow.com/questions/182750/map-a-network-drive-to-be-used-by-a-service>. It may be easier just to move the testing scripts to a directory on the C: drive.