Guidelines For Creating Steps For PI

1/20/2014

There are many ways to create steps. The old story of if 5 engineers were given the task to create a step there would be 5 different implementations is very true.

There isn’t a right way or a wrong way, but after much iteration, there is a recommended structure and set of language guidelines that seem to work for the Test Framework (TF).

The step definition is only the first of 4 layers or levels of abstraction.

The layers are steps, group(helpers), sections and programmatic interface (PI).

# PI Commands

The PI implementation has one responsibility, and that is to make sure that the “long” form of the PI command or query is correct.

## PI Set

The following is an example.

/// <summary>

/// Using OUTPut[n]:STOP:MARKer[n]:STATe set the output data position of a marker while the instrument is in the stop state

/// </summary>

/// <param name="channel">Which channel number</param>

/// <param name="marker">Which marker number</param>

/// <param name="value">Output data position of a waveform</param>

public void SetAwgOutputStopMarkerState(string channel, string marker, string value)

{

string commandLine = "OUTPut" + channel + ":SVALue:MARKer"+ marker + ":STATe " + value;

\_mAWGVisaSession.Write(commandLine);

}

## PI Get

The following is an example.

/// <summary>

/// Using OUTPut[n]:STOP:MARKer[n]:STATe? get the output data position<para>

/// of a marker while the instrument is in the stop state</para>

/// </summary>

/// <param name="channel">Which channel number</param>

/// <param name="marker">Which marker number</param>

/// <returns>Output data position of a waveform</returns>

public string GetAwgOutputStopMarkerState(string channel, string marker)

{

string response;

string commandLine = "OUTPut" + channel + ":SVALue:MARKer" + marker + ":STATe?";

\_mAWGVisaSession.Query(commandLine, out response);

return response;

}

## Organization

PI commands are to be grouped in their own module. This may mean adding to an existing one or creating a new module. Use the Programmer’s manual (PM) to help make that decision if necessary.

When adding the individual “set” and/or “get” for a command, place them into alphabetical order. This should be the same was what you would find in the PM.

Putting C# regions around each PI command helps organize them. ReSharper doesn’t like the use of regions but if it helps to keep the code organized and understandable, then why not.

Each method should have documentation describing the use of the method.

## Responsibilities

This layer of abstraction single responsibility is that the methods correctly configure the PI command or query. It is assuming that the caller knows what it wants sent. In the case of the query, it also formats the proper syntax but it needs to return the response as a “string”.

## Conventions

Start each PI with a verb such as “Set”, “Get”, “Save” and “Force” that represents the end result.

Parameter organization is “channel”, “marker” and then “value”. Do not use the actual PI requirements as a guide of ordering the method’s parameter list.

For the actual syntax, use the long form as found in the PM. This is including the use of the upper and lower case characters. Keeping the upper and lower case has proven to be very useful in debugging during development.

Then finally, update the interface for the PI commands, IPCmds.cs. Including some reference to the actual syntax for the PI(i.e. OUTput:OFF) in the header summary has proven useful.

# AWG Sections

The PI roles or functions have been organized as sections.

Each section is responsible to interface with the PI object which has all of the PI commands. The majority of methods are “set” and “get”. Depending on the nature of the PI there will be methods such as “save” and “restore”. No testing is done by the AWG. The AWG object does not do any formatting of the PI commands.

It is important to note that attributes (not properties) are the storage mechanism for PI query results. These are held for future use. The Gherkin language does not have a concept of moving results from one step to another.

### Set Section

The “sets” are normally just a straight call to the correct PI command as seen in the following example.

/// <summary>

/// Set the output data position of a marker when the instrument is in the stop state

/// </summary>

/// <param name="channel">Logical channel</param>

/// <param name="marker">Logical marker</param>

/// <param name="value">Output data position of a waveform</param>

public void SetOutputStopMarkerState(string channel, string marker, string value)

{

\_pi.SetAwgOutputStopMarkerState(channel, marker, value);

}

### Get Section

The “gets” are just slightly more exciting as they call to the correct PI query and then save the results in the proper attribute to be used later as seen in the following example.

/// <summary>

/// Forces this awg to updates it's copy of the output Stop Value Marker State

/// </summary>

/// <param name="logicalChannel">Which channel number</param>

/// <param name="logicalMarker">Which marker number</param>

/// <returns>Output data position of a waveform</returns>

public void GetOutputStopMarkerState(string logicalChannel, string logicalMarker)

{

int channelNumber = Convert.ToInt32(logicalChannel);

int markerNumber = Convert.ToInt32(logicalMarker);

if ((channelNumber <= AwgMaxChannels) && (markerNumber <= AwgMaxMarkers))

{

\_outputChannelMarkerStopValues[channelNumber - 1, markerNumber - 1] =  
 \_pi.GetAwgOutputStopMarkerState(logicalChannel, logicalMarker);

}

}

## Organization

This has the same guidelines as for the PI commands above. The PI sections are to be grouped in their own module. Remember to use the PM for help for any question about ordering.

## Responsibilities

This layer of abstraction needs to hide any hardware related functions that will change for the different AWG family types. Examples of this would be number of channels, number of markers and number of clocks.

## Conventions

Start each method with a verb such as “Set” and “Get”. Just like the PI interface these verbs should represent the end action. Do not use “Set” or “Get” for a “Save Setup” end action an an example.

Parameter organization needs to be consistent throughout the TF. Use the same ordering as the PI which is is “channel”, “marker” and then “value”.

Attributes that are related to HW features that can vary such as channel count need to be accessed by a logical channel number. Do not create public attributes that have a channel number as part of the name such as “OutputChannel1StateValue” and “OutputChannel2StateValue”. Attributes such as these need to be return by a method that have the logical channel as one of the parameters.

If there are changes of function or additions because of new methods, be sure to update the IAWG and make useful comments for them.

# Group Helpers

The group helper(GH) layer does the work for one or more steps.

The work done can be very simple such as just passing along the “set” or “get” request.

Typically, the “should be” support does the most of the “work”. It normally does the testing and creates error messages and asserts.

Some GH methods do work that doesn’t really fit into the Gherkin language concept. As an example, some of the requirements for testing need to have timeouts changed to be longer for some PI operations. It would be in this layer where that would take place.

Some steps might require a list of actions to take place. An example would be “When I set the default for AWG 1”. Depending on the definition of what “default”, the work for the default could be a simple \*RST command or a number of set calls.

## Set Group Helper

The following is an example for the “set” GH.

public void SetOutputStopMarkerState(AWG awg, string channel, string marker,  
 OutputMarkerStopMode condition)

{

var setCondition = (condition == OutputMarkerStopMode.Off) ?  
 SyntaxForOutputStopValueMarkerOff :  
 SyntaxForOutputStopValueMarkerLow;

awg.SetOutputStopMarkerState(channel, marker, setCondition);

}

Parameter passing order is always in the order of “awg”, “channel”, “marker”, “x”. Where “x” is the value used for the PI command.

The “condition” in the parameter list is a “public” enum that is used between the step and the group helper methods. The goal is to abstract each layer so that any layer could be replaced without affecting the caller. It’s a goal. In this example the enum has defined an “Off” and a “Low”.

The actual syntax for the PI parameter is being defined by this group helper. It uses the condition parameter to determine which syntax is used.

It could be argued that really, the syntax should be part of the “AWG” class as it will implement some version of the AWG and there might very well be a change in the syntax in the future.

## Get Group Helper

The following is an example for the “get” GH.

/// <summary>

/// Have the AWG update its copy of the Output Stop state for a marker<para>

/// and for an AWG</para>

/// </summary>

/// <param name="awg">The AWG object</param>

/// <param name="channel">The logical channel</param>

/// <param name="marker">The logical marker</param>

public void GetOutputStopMarkerState(AWG awg, string channel, string marker)

{

awg.GetOutputStopMarkerState(channel, marker);

}

Parameter passing order is always in the order of “awg”, “channel”, “marker”.

The “get” method normally updates an attribute of the “awg” object where it can be retrieved later for testing.

## Should Be Group Helper

The following is an example for the “should be” GH.

/// <summary>

/// Verify the expected state for the Output Stop state for a marker for an AWG

/// </summary>

/// <param name="awg"></param>

/// <param name="logicalChannel"></param>

/// <param name="logicalMarker"></param>

/// <param name="expectedState"></param>

public void OutputStopMarkerStateValueShouldBe(AWG awg, string logicalChannel,  
 string logicalMarker, OutputMarkerStopMode expectedState)

{

string expectedSyntax = (expectedState == OutputMarkerStopMode.Off) ?  
 SyntaxForOutputStopValueMarkerOff :  
 SyntaxForOutputStopValueMarkerLow;

string outputChannelMarkerStopState =  
 awg.OutputStopMarkerState(logicalChannel, logicalMarker);

string possibleErrorString = ErrorStringCheckingOutputStopMarker + logicalMarker +  
 ErrorStringForChannel + logicalChannel;

Assert.AreEqual(expectedSyntax, outputChannelMarkerStopState, possibleErrorString);

}

Here is where the work gets done. The *expectedState* determines what syntax to be used for checking.

The actual marker state needs to be retrieved from the *awg* object.

It is important to have some explanation of the error if it happens. Restating the obvious, that they aren’t equal in this example, is not helpful. At a minimum, indicate what is being checked. For the above example, there are two strings that have been defined.

public const string ErrorStringCheckingOutputStopMarker = "Checking the output stop value for marker ";

public const string ErrorStringForChannel = " for channel ";

At the time of writing these guidelines, there are numerous assert methods that have no error string created for them. When these asserts happen in a large set of test features, it is really helpful to have some hint/clue to where they are coming from.

## Organization

The name of the game is to be consistent. So just like the PI and Sections layers above, the GH methods are to be grouped in their own module. Group the “Set”, “Get” and “Should Be” methods together in alphabetical order of the related PI command, not the method name.

## Responsibilities

This layer is where most of the work is to be done. For the 99% of the methods, “work” is fairly straight forward. The “Set” and “Get” requirements are to pass parameters to the next layer. The “work” is normally in the “Should Be” method. This is where expected values are compared against the actual values.

The decision was made to keep “Get” as a separate method instead of part of the “Should Be” method. Without too much effort, you could envision a “Get” to be included with the “Should Be” work. It was separated out as there were numerous times where a “Get” was required but not a “Should Be” requirement. So instead of having sometimes “it is” and sometimes “it is not”, the “Get” is always separate. This does however put a burden on the implementer of the “feature” to always do a “Get” step before doing a “Should Be” step.

This layer may require othere “awg” functions/access methods. If there is this type of requirement, then this layer needs to instantiate the required class. Do NOT create a “common” that is shared.

## Conventions

When a requirement for any of the 3 are being implemented, implement all 3 of the methods at the same time. It has already been stated elsewhere that 95% will have all 3 methods, but not all. Do not make dummy methods that do not apply.

As previously stated earlier, the parameter organization needs to be consistent throughout the TF. This layer has an additional parameter, the “awg”. The ordering is now “awg”, “channel”, “marker” and “value”. These parameters are sometimes not used but when they are then use this ordering.

The GH layer works typically works on an “awg” object. This comes from the “Step” which is directing the GH to which “awg” and what “action” that needs to take place.

As of this writing, this layer has some limited knowledge of the PI command parameters. It is this layer that knows what the expected syntax is to turn on and off a function for example. The “Step” is not to know this. But in order for the “Step” to indicate that it wants to do just that, turn on and off, the GH needs to create a public enum to be used. The following are some examples.

public enum OutputStateMode { Off, On }

public enum OutputOffStateMode { Off, On }

public enum OutputAnalogStopMode { Off, Zero}

public enum OutputMarkerStopMode { Off, Low }

public enum OutputAnalogWaitMode { First, Zero }

public enum OutputMarkerWaitMode { First, Low, High }

Don’t be tempted to use a similar enum such as the first two examples, “Output State” and “Output Off State”. Keep them separate, unless of course they truly are the same.

Always think of parameters in their logical representation. This is especially true for channel, markers. Do not think of them in whatever manner they are being managed as. An example of this would be for keeping track of channels, a zero based indexed array would be straight forward for this purpose, so logical channel 1 would be 0. Do not think in terms of implementation, think in terms of logical.

Logical parameters should be used through out all the layers.

Create string constants for syntax strings such as the following.

public const string SyntaxForOutputStateOn = "ON";

public const string SyntaxForOutputStateOff = "OFF";

public const string SyntaxForReturnedOutputStateOn = "1";

public const string SyntaxForReturnedOutputStateOff = "0";

Note that the above examples are for the same command. These are referred to as boolean type parameters. The sending for this command happens to be using “ON” and “OFF”. It could have been “1” and “O” and it would still work. The returned syntax will always be “1” and “O”. For this example it was decided to send one type and check a different return type. It’s additional testing.

# Steps

The step definition is written in Gherkin and normally has 3 forms, a “set”, a “get” and a “should be”. There are exceptions such as delete and save related PI functions.

There may not always be 3 forms because of the end target such as the PI command may only have a “set” or a “get”. An example of a “set” only would be a PI command that forces a trigger. An example of the “get” only would be to get the state of the AWG.

There are two basic types of steps: functional and syntax.

Each PI command should have a syntax feature and one or more functional features.

## Syntax Feature

The syntax feature uses either the  
 “When I send the “<command>” command for AWG 1”  
or  
 “When I send the “<query>” query for AWG 1”.

The syntax feature should only be testing the valid forms as stated in the DevSpec. Testing for incorrect syntax is not necessary. The reason that incorrect syntax testing is not required is that there is only one parser that tries to match a PI command or query to a method to implement that command or query. We have enough confidence that the parser will catch the incorrect syntax.

In addition to the syntax checking, it is not necessary to do both the command and/or query forms. Choose the best fitting for the task at hand. The end goal is the throw a command or query in all of its forms to check the syntax.

The following is an example feature for checking the syntax for the PI command “CLOCk[1]:PHASe[:ADJust]”.

Feature: CLOCk[1]:PHASe[:ADJust] Sets the clock phase using long and short syntax

Background:

Given a reset AWG

And the AWG is of the *70*k family

Scenario Outline: Verify CLOCk[1]:PHASe[:ADJust] command syntax is valid

When I send the "*<command>*" command to the AWG

And I wait for the AWG to complete

Then there should be no error from AWG *1*

Examples:

| *command* |

| CLOCk1:PHASe:ADJust 0 |

| CLOCk1:PHASe:ADJ 0 |

| CLOCk1:PHASe 0 |

| CLOCk1:PHAS 0 |

| CLOCk:PHAS 0 |

| CLOC:PHAS 0 |

| CLOC:PHAS MAX |

| CLOC:PHAS MIN |

Testing should be done one keyword at a time as in the above example. The keyword “Adjust” has a long and short form and is optional. The 3 variations of keyword “Adjust” does not need to be iterated through each possibility of the other keywords. Each keyword represents a unique node in a tree and the parser either finds the node, (i.e. CLOCK) or not and it has nothing do with the following keywords (i.e. PHASe).

## Functional Feature

Functional features are used in various ways. The first is to use the steps to exercise the PI in such a manner that the PI effectively changes the AWG. The simplest exercise is to do a “set”, a “get” and a “should be” step in that order.

The following are some possible tests that might be considered

* To verify that change occurs it might be as simple as:

1. Use the “set” step to send command to the AWG
2. Use the “get” step to get the AWG setting
3. Use the “should” step to verify that expected and actual settings match.

* Can the MIN or MAX be sent and does reading back the value give a valid response?
* If a reset is applied, does reading back the value give a valid response?

The second way is to use the features to duplicate an end user’s process to perform a task using the AWG.

Additionally using the set of features could be used to duplicate a sequence that causes errors for debugging and testing (once the defects are fixed).

## Set

The basic “set” form is  
 “When I set <noun> <verbal phrase > [<for marker 1>] [<for channel 1>] [<for AWG 1>]”

An example of a noun would be “the marker output stop condition” and the verbal phrase would be “to off”. The created step would be something like  
 “When I set the marker output stop condition to off for marker 1 for channel 1 for AWG 1”

In Gherkin the step would look like the following

[When(@"I set the marker output stop condition to zero volts for marker ([1-4]) for channel ([1-4]) for AWG ([1-4])")]

## Get

The basic “get” form is similar to the “set” form but it is has the word “get” instead and contains no verbal phrase. The form for a “get” would be  
 “When I get <noun> [<for marker 1>] [<for channel 1>] [<for AWG 1>]”

Using the same examples, the created step would be something like  
 “When I get the marker output stop condition for marker 1 for channel 1 for AWG 1”

In Gherkin the step would look like the following

[When(@"I get the marker output stop condition for marker ([1-4]) for channel ([1-4]) for AWG ([1-4])")]

## Get Responses

When a “get” step is executed, it causes the AWG to update an attribute related to the noun of the step. In order to ensure success in the “should be” step, a “get” step must be executed for an update to occur. Saving the response in the AWG is essential because the Gherkin language really doesn’t support the concept of variables passing from one step to another.

## Should Be

The basic “should be” form is  
 “Then <noun> should be <verbal phrase > [<for marker 1>] [<for channel 1>] [<for AWG 1>]”

An example of a noun is the same as for a “set” or “get” such as “the marker output stop condition” but the verbal phrase is now “off”. Using the examples, the step would be  
 “Then the marker output stop condition should be off for marker 1 for channel 1 for AWG 1”

In Gherkin the step would look like the following

[Then(@"the marker output stop condition should be off for marker ([1-4]) for channel ([1-4]) for AWG ([1-4])")]

## Organization

I don’t think it can be stated enough, the name of the game is to be consistent. So just like the layers above, the steps are to be grouped in their own module. Group the “Set”, “Get” and “Should Be” methods together in alphabetical order of the related PI command, not the method name.

## Responsibilities

The main responsibility of the step is to ensure that all the parameters that are passed to the next layer, the GH, are valid.

The following is an example of an implementation of a “get” step.

[When(@"I get the marker output stop condition for marker ([1-4]) for channel ([1-4]) for AWG ([1-4])")]

public void GetMarkerOutputStopConditionChannelMarker(string marker, string channel, string awgNumber)

{

AWG awg = AwgSetupSteps.GetAWG(awgNumber);

\_utilitiesGroup.VerifyChannelMarkerClockParameters(awg, channel, marker);

\_awgOutputGroup.GetOutputStopMarkerState(awg, channel, marker);

}

It is important to note that the Gherkin language will make sure that the numbers for marker, channel and AWG will be between 1 and 4 and will error if not. What Gherkin won’t do is dynamically change those ranges depending on the AWG type. It is necessary to verify any potential parameter to be passed. The method VerfiyChannelMarkerClockParameters(), which can be found in the utilities group, will need to be called. This method will cause an assert if the marker and channel numbers do not fall within the range for the given AWG.

## Conventions

In general, Steps are not to know any details about PI command and/or query it is supporting. It will know about functionality in general such as “on” and “off” or setting the control to maximum.

The basic step structures are as follows:

* “When I set <noun> <verbal phrase > [<for marker 1>] [<for channel 1>] [<for AWG 1>]”
* “When I get <noun> [<for marker 1>] [<for channel 1>] [<for AWG 1>]”
* “Then <noun> should be <verbal phrase > [<for marker 1>] [<for channel 1>] [<for AWG 1>]”

References to markers, clocks, channels and AWGs have a logical number from a range of 1-4. There is a requried space before the logical number. Do not use the word “the” before any of these.

It is OK to make a reference to “low” and “high” in the step even though these words could actually be used in the PI command. But when using them, they need to be lowercase with no quotes.

Triggers references use A and/or B. These also require a proceeding space.

Generally steps are viewed as a means to an end, and that is to test the functionality of the AWG.

It is important to understand that a step is not a syntax checker but there are a few exceptions.

There are specific steps that support just sending syntax. Two of these steps are the “send a command” and “send a query”. These are used for syntax testing for new and backward compatible commands and queries.

Steps are to exercise the latest PI commands and queries. If there are backward compatible commands and queries they need to be reviewed for what is really being exercised. If there is a new command that is replacing the them and both call the same underlying code, then only do a syntax check on the backward compatible command. That really should be the only case.

References to waveform and files should be in quotes such as the following example.

And I resample the waveform "*join\_sine*" from the waveform list to size *4800* for AWG *1*