Using Fluke MET/CAL® to Implement a Flexible Measurement Driver Model with Expanded Measurement Uncertainties and Error Checking

Speaker/Author: Michael L. Schwartz
Cal Lab Solutions
PO Box 111113, Aurora CO 80042 US
T: 303-317-6670 | F: 303-317-5295
Email: MSchwartz@CalLabSolutions.com

Learning Objectives: Programmers, and especially MET/CAL® programmers, will gain valuable knowledge and insight on how to structure their procedures to provide easier interchangeability of standards and incorporate measurement uncertainties. This driver based model incorporates the principals of object oriented programming, and facilitates faster procedure development, greater flexibility in standards, with lower support overhead.

Abstract: Writing one or two automated procedures is easy; however, automating an entire calibration lab is not an easy undertaking. Even with today's software tools built specifically for metrology, a significant effort must to be put into designing an architecture that will foster the reuse of code, flexibility of standards and incorporate expanded measurement uncertainties. Companies and/or developers who skip this crucial step are quickly overwhelmed with rework that ultimately hinders long-term productivity.

After more than 15 years of MET/CAL® procedure development trial and error, we believe we have found the balance between productivity and architecture. This paper outlines the structure and development methodologies we use to write more robust code, with a greater emphasis on quality and testing with less refactoring of our procedures.

At the heart of our development principles is an interchangeable flexible driver model complete with expanded measurement uncertainties. Measurement based drivers provide the bases for interchangeability and reuse of code; it is what allows our developers to take full advantage of the principals of Rapid-Application-Development, while simultaneously providing higher quality and shorter development cycles with minimal rework.

Software developers who read this paper, understand and incorporate the architecture into their own best practices will see shorter development times and greater reuse of code, as well as decreased support requirements and procedure rework.

1. Introduction

In today's competitive business environment, the name of the game is "Better, Cheaper, Faster." Calibration labs are no different, but they are faced with the additional challenge of calculating measurement uncertainties on every standard and every UUT at every test point. If you have implemented measurement uncertainties to this level, you know it is better, but it is also a time consuming and costly endeavor.

Since we are a software company, we naturally looked to solve these problems in software. There are several software packages out there that will help you calculate measurement uncertainties on the fly--MET/CAL® is one such solution. But creating a calibration procedure for each set of standards a customer might have is still labor intensive.

After many years of aggravation, we decided to incorporate the principles of OOP (Object Oriented Programming) into our MET/CAL® procedure development. Though MET/CAL® is not natively an OOP Language, we found by applying some specific design patterns we could drastically decrease our support burden. Now we are able to support and maintain a library of procedures covering 853 of the more complex instruments, with a relatively small programming staff (i.e. "Better, Cheaper, Faster!").

2. Background

In short, Object Oriented Programming (OOP) is a programing paradigm; it changes how we think of software and its individual elements. The idea is to think of every element in your software as "Objects" – data structures consisting of data fields and methods together with their interactions. Then allow those objects to interact with each other based on a defined set of rules, thus allowing for advanced programming techniques and features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance.

OOP will allow the programmer to create a simple abstract top level layer of code that interfaces with the lower levels, (i.e. the objects) who become more specific and sort out the exact details. If you think of an Object as a wrapper containing data and functionality, you allow objects to do some of the work for you. The top level programmer does not need to know the details of how the Object is implemented--only the interface by which his is allowed to use the Object.

Object Oriented Programming has several design patterns, but in this example we are only going to focus on the Abstract Class Pattern, the intent of which is to provide an interface for creating families of related or dependent objects without specifying their concrete class or implementation (as defined by the Gang of Four) [1]. This design pattern has all the basic elements required to design software with interchangeable modules. Its design provides a proven methodology for writing interchangeable sections of code. Imbedded in the pattern is the ability to pick and choose the specific code to be used at a later point in time. The developer writing the top level code must conform to a standardized interface, while the lower levels of code (i.e. interchangeable objects) are tasked with the specific implementations as required by the object's interface.

MET/CAL®, as you know, is not an Object Oriented Programming language, instead it is high level scripting language written specifically for metrology and the controlling of instrumentation. By design, MET/CAL® simplifies the process of creating an automated test script for a novice programmer. But just because the MET/CAL® complier is not able to natively implement OOP, that does not mean, we programmers can structure programs in a fashion to take advantage of OOP.

3. Example MET/CAL® Procedure

In order to keep things simple in the provided examples, we will focus more on the concepts of the architecture instead of the specific implementations, by demonstrating our OOP implementation of Source.Volt.DC. In this example, we will demonstrate how to implement an Abstract Class Factory pattern using MET/CAL® and how we changed our programming methodologies from a standard specific programming model to an Object programming model using the Source.Volts.DC interface. Subsequently, we will also show how we moved the specific knowledge of the exact standards being used to a sub procedure, containing error checking, command and programming calls and associated measurement uncertainty calculations for the standards being used.

To start, let's take a typical test point from a MET/CAL® procedure testing 1 Volt DC on a DMM:

```
#======  Sample Test Point 1 ============
  4.001  5520         1.0000V                                 S  2W
  4.002  TARGET       -m
  4.003  IEEE         Read?[i]
  4.004  MEMCX        V              0.0001U
```

Example 1. Sample Test Point 1.

Most MET/CAL® programmers who wanted to change the standards on this test point would do so by simply creating a whole new procedure by making changes and then saving the procedure (one each for each standard they wanted to support). If they wanted to test this UUT with the following standards (5500, 5520, 5700 or 5720) they would have 4 copies of the exact same procedure. And in the end, they would now have several procedures they needed to support.

As programmers, we have to limit the amount of code we write, since every line of code we write is a line of code we have to debug and support. Less code means less work, and more importantly, less re-work. If we have four or more procedures we have to maintain and need to make a change to the procedure--such as a test specification change--we have to change it in all four procedures. Over time, this adds up, so soon we are spending more time supporting procedures than we are writing them.

Our objective for Example 2 was to make any standard capable of generating 1 Volts DC a drop-in replacement for the Fluke 5520. To do this, we had to create a programming standard defining the variables we passed into a Source.Volts.DC driver and the parameters we received back, then replace the 4.001 line using the 5520 FSC with something more abstract. In this case, our only requirement is that the standard must be able to produce 1 Volt DC two wire.

```
# #===== Sample Test Point 2 ============
  4.001  MATH         S[30]="Source.Volts.DC Volts= 1"
  4.002  CALL         My Config Sub
  4.003  MATH         L[1]=Fld(S[31],2,"VoltsUnc=")
  4.003  MATH         MEM=Fld(S[31],2,"Volts=")
  4.003  TSET         UUT_Res= 0.0001
  4.004  ACC          V        L1U
  4.005  TARGET       -m
  4.006  IEEE         Read?[i]
  4.007  MEMCX        V              0.0001U
```

Example 2. Sample Test Point 2.

By removing the specificity of the Fluke 5520 call and replacing it with Source.Volts.DC, we are now able to use any standard that supports a Source.Volts.DC interface. This change to the procedure design allows us to have a single test point procedure we have to support, independent of the specific standards used. If the manufacturer were to change the test points or test limits, we only have one procedure we need to update.

In the Test Point 2 sample code, we have removed the 5520 FSC and inserted an abstract command in S[30] that specifies we want to Source.Volts.DC with Volts = 1 V. At this point, we have not specified a particular standard, only the parameters we require. Then we call a sub procedure “My Config Sub” that implements an Abstract Class Factory pattern and is responsible for choosing the specific Source.Volts.DC driver sub.

Another thing to note is when we removed the specific FSC call to the Fluke 5520, we also introduced some additional unknowns. In the Sample Test Point 1, we knew the Fluke 5520 would output 1 Volt DC with uncertainties based on the interval set in the MET/TRACK® database, though now, because we requested 1V DC to the Source.Volts.DC driver, the exact value and associated uncertainties are unknown and we have to accommodate that unknown in the procedure.

Next, in the “My Config Sub Sample Code” (Example 3), we are able to take the commands found in S[30] and select the a specific Driver. In this example, we are still using a Fluke 5520A, but now the procedure can be quickly changed to any other standard that supports the Source.Volts.DC interface. The programmer only needs to change the connection message on line 4.007 and the driver call on line 4.011, thus making it easier to support multiple standards without a large support burden.

```
#====== My Config Sub Sample Code =========================================
  4.001  LABEL        VoltsDC
  4.002  JMPL         VoltsDC_Conn      Find(S[30],"Connect",1)>0
  4.003  JMPL         VoltsDC_Source    Find(S[30],"Source.Volts.DC",1)>0
  4.004  DISP         Error Calling Sub
  4.005  END
#=========================================
  4.006  LABEL        VoltsDC_Conn
  4.007  DISP         Connect the Fluke 5520 to the UUT as Follows;
  4.007  DISP         [32]  NORMAL HI <-----> V
  4.007  DISP         [32]  NORMAL LO <-----> COM
  4.008  END
#=========================================
  4.009  LABEL        VoltsDC_Source
  4.010  CALL         CLSD-Source.Volts.DC                     (5520A Normal)
  4.011  END
```

Example 3. My Config Sub Sample Code.

At first it may appear all we have accomplished is taking 4 lines of code and spreading it across 3 sub procedures, only to increase the procedures complexity. But as you study the code, what we have accomplished is the transformation of 5520 specific procedure to a configuration management solution. Instead of supporting several individual procedures each with different standards, now we are supporting only one procedure with an unlimited number of possible configurations, all utilizing a set of interchangeable drivers. All a user needs is the correct "My Config" file for his standards.

4. CLSD-Driver Model

The core principles of these OOP/interchangeable drivers have served us well, allowing huge flexibility in standards, adapting to customers' specific requirements, and the ability support complex standards before FSC are available.

Over the years, as our programming standards have evolved, we have learned a few valuable lessons. The most important of them is that you have to have a well-documented programming standard. It is imperative that all the developers in our company be on the same sheet of music. We must all be able to generate the same quality of code, all of our drivers must conform to a standard and all of our procedures be writing with interchangeability in mind.

We started by creating an I/O command architecture that all UUT procedures and drivers must implement. This becomes the contract between the calling procedure and the driver, so it must be implemented exactly the same in every instance.

Under the driver contract, every driver must support the following commands:

Name – Returns the Name of the STD and Connection Point
Setup – Performs any required Setup / Configuration Tasks
Reset – Resets the Standard(s)

OutputOff – Turns the Output Off (Implemented on Sources Only)
<Metrology Method> – Source.Volts.DC in this example

Every driver will support one or more metrology methods. It is the driver's responsibility to carry out the commands given, or present an error to the operator detailing why it is unable to carry out the task. It is the driver's responsibility to know its specifics, and adhere to the command interface/contract with the calling procedure.

Error handling is one of the most important tasks of the driver sub. Once the input parameters have been parsed out of the command string in S[30], they have to be error and range checked to insure the standard is able to perform the task. Then, if possible, the driver needs to be able to error check the standards to insure no additional errors have accorded, such as overvoltage or unleveled instrumentation error.

Once the driver has performed its entire list of tasks, it then calculates the measurement uncertainty and reports them back to the calling procedure. How it calculates its uncertainties is specific to its implementation and the standards being use, as long as the driver adheres to its defined interface/contract with the calling procedure.

Driver	Standard
CLSD-Source.Volts.DC (5500A Normal)	Fluke 5500 Volts Connection Post
CLSD-Source.Volts.DC (5570A Normal)	Fluke 5520 Volts Connection Post
CLSD-Source.Volts.DC (5700A Normal)	Fluke 5700 Volts Connection Post
CLSD-Source.Volts.DC (5720A Normal)	Fluke 5720 Volts Connection Post

Table 1. Examples of interchangeable drivers with this paper.

5. Polymorphism

Now we have a problem. We are satisfied with all but one of the standards we have listed in Table 1; the Fluke 5500A is not accurate enough to source 1V DC for a test point with +/- 100uV test limit, but we need to be able to use the 5500A on this UUT in the field. We know we can monitor the output with a more accurate meter, but we don't want to completely re-write this procedure. Welcome Polymorphism!

Polymorphism is a word you certainly don't hear every day, but it holds the answer to our problem. In OOP we can have vastly different implementations of Source.Volts.DC and the calling procedure does not have to worry about specifics. By completely passing the responsibility to the sub procedure (i.e. the object), we also pass the responsibility of the specifics. This allows us to create and implement Source.Volts.DC in any manner we can fathom. The possibilities are limit less.

So now we need to create 1 Volt DC with a 5500 and monitor it with an Agilent/HP 3458A (Table 2). In doing so, we are going to completely change our test methodology. We will source the voltage from the calibrator, measure it with the 3458A, slightly correct the output so we are close to 1.0000V DC and report back the uncertainties to the calling procedure. Best of all, this all can be handled in the driver sub, without having to update the calling procedure.

Driver	Standard
CLSD-Source.Volts.DC (5500 Normal & 3458A)	Fluke 5500 Volts Connection Post monitored and corrected with an HP/Agilent 3458A.

Table 2. Examples demonstrating Polymorphism drivers with this paper.

6. Measurement Uncertainties

This paper is an expansion on the paper "Implementing A2LA's new Budget Requirements for Electrical and RF Uncertainties in Fluke MET/CAL® Procedures," presented last year at NCSLI [2]. So the specifics of how we are mapping the U values and calculating out the expanded measurements uncertainties are explained in more detail in that paper.

I want to specifically point out one of the distinct advantages of using an OOP driver based procedure design model is the ability to encapsulate measurement uncertainties and process inside each driver. The process of encapsulation is another important part of OOP, allowing us to not only interchange standards, but also associate the measurement uncertainties with each specific standard. Since the CLSD-Source.Volts.DC (5500A Normal) sub is shared, when we update its measurement uncertainties, we do so for all procedures calling it—again, minimizing the code base needed to support.

Because we are able to store process in the drivers underlying code, this allows us to tightly couple the process with the measurement uncertainties. We can now handle huge variances in the process, such as monitoring the output and auto leveling the output based on the DMM's reading, with seamless integration with the calling procedure.

By comparing Examples 4 and 5 below, we are not only able to change standards, but we can also change the procedure methodology and associated measurement uncertainties. In Example 4, a Source-to-Measure test methodology is used, where the 5500 is the calibrated source and the UUT was reading its value. Then in example 5, we changed to a 5500 monitored by a 3458A so the test methodology becomes a Measure-to-Measure. This process must report back the uncertainties of the applied value compared to nominal (Value= and Unc=), as well as the value measured by the Agilent/HP 3458A and its associated measurement uncertainties.

```
#====== Source.Volts.DC 5520 Normal =====================================
# Calculate the Uncertainties
  1.028  MATH         L[11]=ACCV("Fluke 5520A","Volts", MEM)
  1.029  MATH         S[31]=       " Value= " & MEM
  1.030  MATH         S[31]=S[31]& " Unc= " & L[11]
  1.031  MATH         S[31]=S[31]& " Volts= " & MEM
  1.032  MATH         S[31]=S[31]& " VoltsUnc= " &L[11]
# Standard Resolution
  1.033  IF           L[1]<330e-3
  1.034  MATH         L[31] = .1e-6
  1.035  ELSEIF       L[1]<3.30
  1.036  MATH         L[31] = 1e-6
  1.037  ELSEIF       L[1]<33.0
  1.038  MATH         L[31] = 10e-6
  1.039  ELSEIF       L[1]<330
  1.040  MATH         L[31] = 100e-6
  1.041  ELSE
  1.042  MATH         L[31] = 1000e-6
  1.043  ENDIF
  1.044  MATH         L[31]=L[31]/2/Sqrt(3)
  1.045  TSET         U7 = [L31]
# Standard Traceability (Assuming 4 to 1 or Better)
  1.046  MATH         L[31]=L[11]*.25
  1.047  TSET         U7 = [L31]
```

Example 4. Source.Volts.DC 5520 Normal.

```
#====== Source.Volts.DC 3458A 5520A Normal ==============================
  1.045  MATH         L[11]=ACCV("HP 3458A","Volts E", MEM)
  1.046  MATH         S[31]=       " Value= " & MEM
  1.047  MATH         S[31]=S[31]& " Unc= " & L[11]
  1.048  MATH         S[31]=S[31]& " Volts= " & L[1]
  1.049  MATH         S[31]=S[31]& " VoltsUnc= " & (L[11]+(MEM-L[1]))
#-----------------------------------
# Standard Resolution
  1.050  IF           MEM<1.2
  1.051  MATH         L[31] = 10e-9
  1.052  ELSEIF       MEM<12
  1.053  MATH         L[31] = 100e-9
  1.054  ELSEIF       MEM<120
  1.055  MATH         L[31] = 1e-6
  1.056  ELSE
  1.057  MATH         L[31] = 10e-6
  1.058  ENDIF
  1.059  MATH         L[31]=L[31]/2/Sqrt(3)
  1.060  VSET         U7 = [L31]
# Standard Traceability (Assuming 4 to 1 or Better)
  1.061  MATH         L[31]=L[1]*.25
  1.062  VSET         U5 = [L31]
```

Example 5. Source.Volts.DC 3458A 5520A Normal.

7. Conclusion

Though MET/CAL® is not by design an Object Oriented Programming Language like Microsoft .Net and Java, we can still take full advantage of the architectural principles, design patterns and other features of OOP to write more robust, innovative and fault tolerant procedures. By learning and implementing OOP techniques, it has allowed us, as a company, to support more procedures with a smaller staff and at the same time produce more robust procedures. Besides the examples given here, a packet of samples can be found online at http://www.callabsolutions.com.

8. References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, p. 87, 1995.
2. M. Schwartz and K. Haynes, "Implementing A2LA's new Budget Requirements for Electrical and RF Uncertainties in Fluke MET/CAL® Procedures," NCSL International 2011 Workshop and Symposium.

Other recommended resources for further information:

Design Patterns Explained, by Alan Shalloway and James R. Trott (2005).
Lean Software Development: An Agile Toolkit, by Mary and Tom Poppendieck (2003).
Head First Design Patterns, by Eric Freemand and Elisabeth Freemand (2004).