Object Oriented First Steps

(TS 6139)

Michael Porter

I suppose a good place to start this session is with an admission that, in a sense, it is flying a false flag. After all, one could reasonably interpret the title of this session to indicate that it is going to be demonstrating how to start using objects in LabVIEW. That interpretation is incorrect due to that troublesome word "start". The fact of the matter is that you can't use LabVIEW without interacting with objects, and many parts of it (think: VI Server) are overtly object-oriented – even without an obvious class structure. The language is built on an objects oriented foundation and so, in a very real way, has been object-oriented since Version 1.

What I am going to be showing you is how to simplify your work by building your own classes. As I've often stated, a good starting point for this discussion is the recommendation given in NI's own object-oriented training class that you should make your first attempts at using explicit object-oriented technique small, easy to manage subsystems – or put more simply, we need to start with baby steps.

1 Object-oriented baby steps

OK, so this is the point in the presentation where most presenters hauls out some standard theory, and moth-eaten descriptions of objects and classes – often lifted wholesale from a book on C++ programming. The problem with this approach is, of course, that we aren't C++ programmers and the amount of useful information we can draw from an implementation of objects oriented programming that is based on a different language is minimal at best. It's sort of like trying to learn proper English grammar by reading a book on how to conjugate Latin verbs. The approach I intend to take instead focuses on key aspects of the technique that are of immediate, practical importance to someone who is working in LabVIEW and wants to take advantage of explicitly implementing object-oriented class structures.

1.1 A Quick Glossary

The first thing we need is a vocabulary that will let us talk about the topic at hand. Now be forewarned that some of these definitions may not exactly match what you may read elsewhere, but they are correct for the LabVIEW development environment.

- **Class** – An abstract datatype. If you think that sounds a lot like the definition of a cluster, you're right! Due to the way LabVIEW implements object orientation, a class is essentially a very fancy cluster. In fact, when you create a class the first item that LabVIEW inserts into it is a typedef consisting of an empty cluster. Although you don't have to put anything into the cluster, this structure provides a place where you can put data that is private to that class.
- **Object** – An instance of a class. The fundamental way LabVIEW represents data is not with named variables, but with wires. Consequently, in LabVIEW objects are wires. Moreover, a class wire is in most ways the same as any other wire in LabVIEW. We are still working in a dataflow environment.
- **Property** – A piece of data that tells you something about an object. This is why there is a cluster at the heart of the class. You want to put into that cluster such information as is necessary to describe the object is a way that is meaningful to your application. Because each instance of the

class is a separate wire that has its own memory space, the data contained in the cluster describes that particular object.

- **Method** – A function that is associated with a particular class and which makes it do something useful. So what do I mean by, "…something useful…"? Well that all depends on the class' purpose. If the class is responsible for creating a visual interface, "…something useful…" might mean methods that cause an object to draw something on the interface. Likewise a class that manages the interface to data storage would likely include methods to store or retrieve application data.

From this simple list of words we can begin to see the general shape of the arena in which we will be playing. To recap: A class is a kind of wire. An object is a particular wire. A property is data carried in the wire that describes it in a useful way, and methods are functions that use the object data (in concert with runtime data) to do something the application needs done.

1.2 Key Features

A good place to start our exploration of how to apply classes, objects, properties, and methods is with some of the unique aspects and concepts that govern our usage.

1.2.1 Dynamic Dispatch

First we need to talk about the mechanism that LabVIEW uses to call methods. Referred to as dynamic dispatch this feature it is often a source of confusion to developers getting started with object-oriented programming. A good way to come to grips with dynamic dispatch is to compare and contrast it to a feature of LabVIEW with which you may already be familiar: polymorphism.

Polymorphism (from the perspective of the developer using a polymorphic subVI) is the ability of a single functions to adapt to whatever datatype is wired to its inputs. For example, the low-level Add node in LabVIEW is polymorphic. Consequently, it can add scalar numeric of all types, as well as arrays of numerics of varied dimensions, clusters of numerics and even arrays of clusters of numerics.

Of course, from the perspective of the developer creating a polymorphic VI, the view is much different. This flexibility doesn't happen on its own. Rather, you have to create all the individual instance VIs that handle the various datatypes. Dynamic dispatch (which is actually a form of polymorphism) works much the same way, but with a couple significant differences.

- **When the decision is made as to which instance VI is to be executed** – With conventional polymorphism, the decision of which instance VI to call happens as you wire in the subVI. In the case of my polymorphic subVI, as soon as I wire-up a U32 input, LabVIEW automatically selects the U32 version of the code. However, with dynamic dispatch, that decision gets put off until runtime with LabVIEW making the decision based on the datatype present on the wire as the subVI is called. Of course for that to work, you need a different kind of wire. Which brings us to the other point…

- **The criteria for choosing between VIs –** The wires that conventional polymorphism uses to select a VI all have one thing in common: they are all static datatypes. By that I mean that a wire is a U32, or a string or whatever and it can't change on the fly. By contrast, with dynamic dispatch, the basis for selection is a wire that is an instance of a class, but the exact datatype of an object can be dynamic. However this variability is not infinite. A given class wire can't hold just any object because class structure is also hierarchical.

Say you have a class named *Geometric Shapes to Draw*. You can define other classes (called sub-classes) like *Circle* or *Square* that are interpreted by LabVIEW as being more specific instances of *Geometric Shapes to Draw* objects. Due to this hierarchical relationship, a given wire can be typed as a *Geometric Shapes to Draw* but at runtime really be carrying a *Circle* or *Square*. As a result, a dynamic dispatch VI can – and often does – call different instances, called method VIs, every time it runs.

However, one big thing that conventional polymorphism does have in common with dynamic dispatch, is that the power doesn't come for free. You still have to write the method VIs for dynamic dispatch to call.

1.2.2 Inheritance

Remember a moment ago I referred to class datatypes as being hierarchical? The fancy computer science concept governing the use of hierarchical class structures is called inheritance. The point of this label is to drive home the idea that not only are subclasses logically related to the classes above them in the hierarchy, but these so-called child classes also have access to the properties and methods contained in their parent classes. In other words they can "inherit" or use the data and functional capabilities that belong to their parents.

Handled properly, inheritance can significantly reduced the amount of code that you have to write. Handled poorly, inheritance can turn an otherwise promising project into a veritable train wreck.

1.2.3 Proper Organization

Although organization isn't really a feature of object-oriented programming, it is never the less critical. The simple fact of the matter is that while a disorganized, undisciplined developer might be able to get by when working in conventional LabVIEW, introducing the explicit use of classes can result in utter chaos. When I consider all the real object-oriented failures that I have seen over the years, they all shared a lack of consistent organization.

So what sort of organizational things am I talking about? Well it's a lot of the same stuff that I have talked about before. For a more general discussion of the topic you can check out a post that I wrote very early on titled, Conventional Wisdom. What I want to do right now is highlight some of the points that are particularly important for object-oriented work.

The two main conventions (directory structure and file naming) go together because the point of one is to mirror the other. For example, I often start by creating a directory that is named for the class hierarchy that I will build inside it. So if the point of this class hierarchy is, for example, to update my pro-

gram's user interface, I would call the directory something obvious like *GUI Update*. Inside this directory I would then create the top-level class with the file name *GUI Update.lvclass*. At this time I will also create a couple subdirectories (*_subVIs* and *_typedefs*) that I know I will undoubtedly be needing. Finally, I have learned over the years that being able to tightly control access to VIs is very important, so I will also create at this time a project library named *GUI Update.lvlib* and put into it the top-level class and a subdirectory called *_subclasses*. Then to complete the process, I create virtual folders with the same names as the subdirectories.

So the parent class is set up, but what about the subclasses? I simply repeat the pattern. Let's say the *GUI Update* class has subclasses for three types of controls that it will need to update: Boolean, Digital and Cluster. I create subdirectories in the parent directory that are named for the subclass that will go into each, and hierarchically name the three subclasses *GUI Update_Boolean.lvclass*, *GUI Update_Digital.lvclass*, and *GUI Update_Cluster.lvclass*. I am also careful to remember to add the subclass files to the *_subclasses* virtual folder in the library, edit their icon overlays, and set their inheritance correctly – which is to say, identify their parent. Note that while the hierarchical naming structure doesn't automatically establish correct inheritance, this convention does make it easier to visualize class relationships in the project file.

So I go building each layer in my class hierarchy. With each new subclass I continue the same pattern so if I eventually want to find, say a subVI associated with the class

GUI Update_Digital_Unsigned Word.lvclass

I know I will find it in the directory:

../GUI Update/Digital/Unsigned Word/_subVIs

Having a pattern to which you stick relentlessly – even one as simple as this one – will save you immeasurable amounts of time.

1.3 Getting Down to Specifics

To see how this theory all plays out in the real world, let's look at a typical sort of application that to this point saved all its configuration data in text files. Our goal is to migrate it onto a simple database. Now while this might sound like a major shift, it really is not because (as like I to say) the whole point of good design is to make changes and upgrades like this possible.

1.3.1 Creating the Blueprint

The next thing I do when creating a class hierarchy is look for how the rest of the application will interface with my new class. This is where the access scope we have been so careful to create comes into play. In the top-level class I always create a group of VIs that have their access scope set to public. These interface VIs form the totality of the external interface to the class hierarchy and so include the functions that define what the application as a whole needs the new class to do for it. The logical implications of this interface layer is why I sometimes call this step in the process, "Creating the Blueprint".

In addition to providing a very clean interface, another advantage of having this "blueprint" is that if you ever need to expand your stable of subclasses, these interface VIs will serve as a list of functions you need to support in the new subclass – or at least a list of functions that you should consider implementing in the new subclass. It is always important to remember that just because a parent method exists, you don't always have to override it.

So let's look at what this upgrade will need to do. Normally a large part of any upgrade projects is defining the requirements, but due to the design work that was put in originally, we already have a pretty good handle on what the new class structure has to do. In terms of surface functionality, we know we have to be able to handle all the same information as before – with, of course, the ability to add more when we want or need that ability.

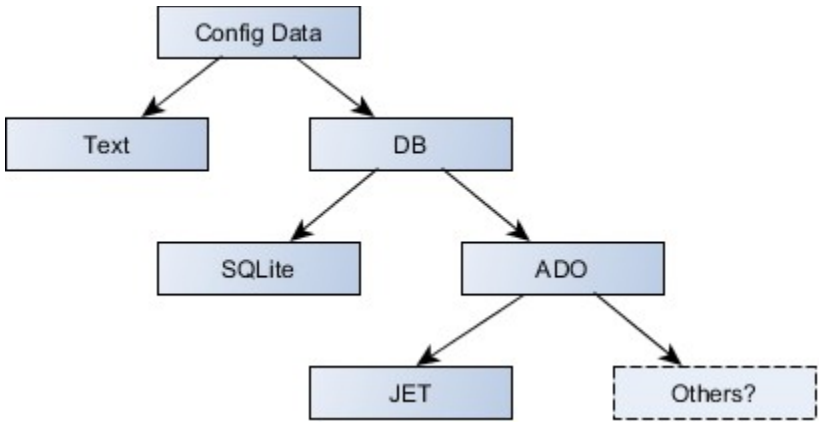
The result of all this analysis is a list of the VIs that the rest of the application will call. After going through our example application we see that there are really only 3 things that the rest of the application needs from the configuration data class:

1. *Default Sample Period*
2. *Error Handling Parameters*
3. *Processes to Launch*

Many of the others will still be used, but their presence will be hidden in subclasses. The actual creation of these VIs will wait for later, right now we just need to note them and what they do.

1.3.2 Designing the Structure

The trick is going to be sorting out what new functionality will be needed under the covers. At the most basic level we need to be able use either text files or databases to actually store the configuration data, so there we have two subclasses. But we need to consider whether each of those options needs to be broken down further.



On the "text file" side, the data might be coming from a standard INI file, or the code might be in using a custom text file format. Custom text configuration files are very common when some of the configuration data is tabular, since it is a pain to store tabular data in an INI file. However, since we right now

really aren't interested in expanding that capability right now, we won't be putting any subclasses under "text files".

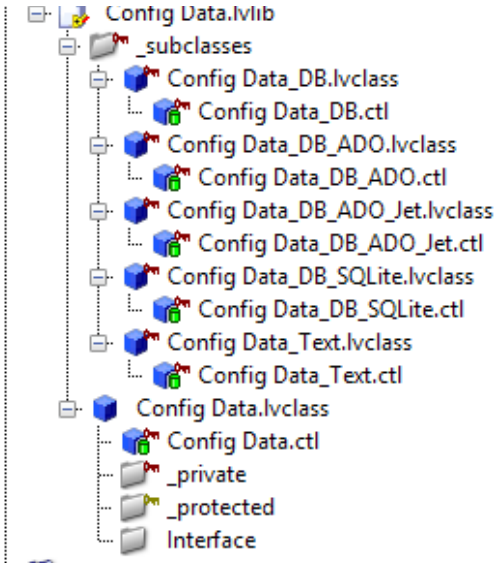
On the "database" side of things, however, the situation is very different. First of all, in terms of connectivity, you can access most databases through the standard ADO (ActiveX Data Objects, also sometimes called ole-db) interface. However, "most" is not the same as "all" and one common exception is SQLite. A popular, lightweight data management engine, SQLite can run on a variety of platforms – including some real-time systems. To keep its footprint small, SQLite utilizes a small custom DLL, rather than a large, but standardized, interface. So we need to make provisions for other types of connectivity by creating (for now) two subclasses below "database": "ado" and "sqlite".

Finally, what about "ado"? Can it be broken down further? Maybe. One of the advantages of ADO is that, for the most part, it does a pretty good job of hiding the differences between one database management system (or DBMS) and the next, but there are some variations it can't paper over. These differences often relate to the version, or dialect, of SQL the DBMS speaks. However sometimes differences arise because some DBMS fundamentally don't operate the same. For example, while most DBMS go to extraordinary lengths to hide exactly where and how the data is actually stored, Jet (the DBMS built into Windows) stores the data in a file you explicitly identify. Hence, while the connection to other DBMS might be defined in terms of network paths and logical names, with Jet you are connecting to a particular file.

To provide for these sorts of functional nuances, let's define a subclass below "ado" for "jet" – understanding there could be others in the future.

1.3.3 Doing the Rough Framing

When you are building a house the first tradesmen to show up on-site are the carpenters to do the so-called "rough framing". This process creates the skeletal form of the building with rough plywood shell that later workers will finish. Metaphorically speaking, that's what we have to do now for our configuration data class.



For a class hierarchy, the framing consists of the class files and the directory structure to hold them. As explained before, I create a directory structure in the class' base directory that mirrors this structure. Note that I have also created some virtual folders inside the *Config Data* class which represents actual subdirectories.

- **Interface** – The VIs in this folder will have public access scope. In fact they will be the only VIs in the library that are so scoped. Because these VIs are the only ones that outside callers will be able to call, they alone form the interface between the class hierarchy and the rest of the code.
- **_private** – As the folder title implies, the files that go in here will have private access scope. This assignment means that they will only be accessible from other VIs in the top-level class.
- **_protected** – This is another folder that specifies access scope for its contents. In the case of protected scope, the VIs in this folder will only be accessible from the top-level class, or any of its child classes.

1.3.4 Adding Infrastructure

Getting back to our housebuilding analogy: After the framework is completed and the outside skin is on, the next job is to start installing some of the needed infrastructure, because without electric, water, sewer and perhaps gas connections, our new home is not much better than the cave dwellings that our prehistoric ancestors inhabited – and in some ways is far worse. At least caves had some temperature control.

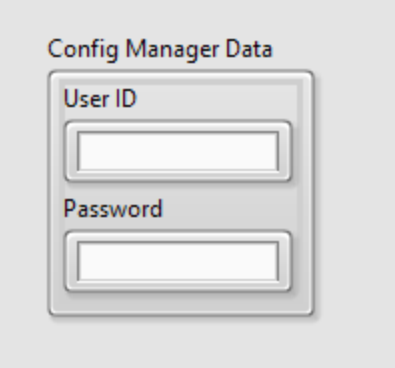
What we need to add to our nascent configuration management subsystem is some data handling, but not for our data. The data I'm talking is the private internal data that the subsystem needs to maintain in order to do its job.

Thinking about what our various bits of code need to do, we see first that there is certain data that will commonly be needed regardless of how you actually end up getting the data. What I'm thinking about here is the name of the operator and a password. Now, some subclasses might need both, while others might need only one or the other, and that's fine. The important point is that all subclasses could potentially need at least some of this data, so the data should be associated with the top-level class. However, this requirement for global availability causes a problem.

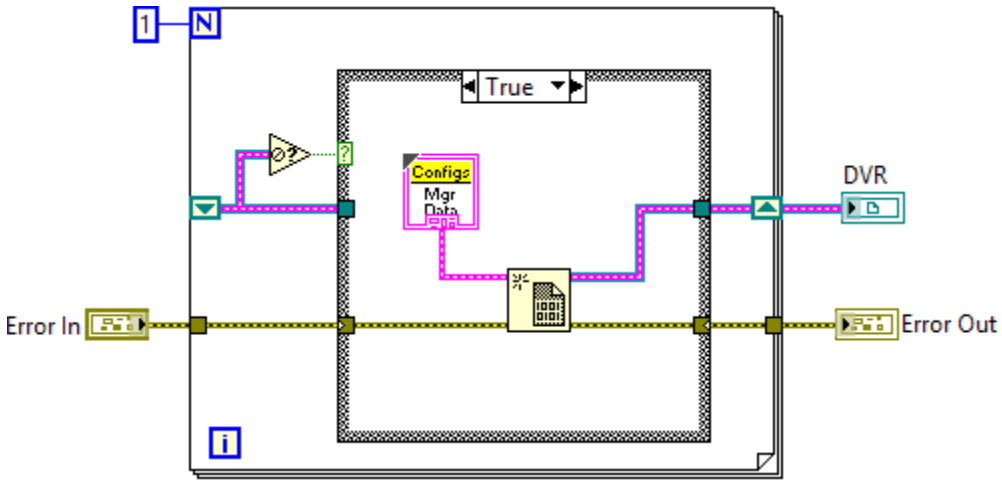
Remember how when we were defining terms earlier I said that in the LabVIEW world an object is a wire? LabVIEW wires, and data contained in them, are by definition not global. If you want data from a wire you need to be connected to it. So how can we create wires that carry separate distinct objects, but which still share at least some of the same data? Well, one very good way of doing it would be to create one central data store that all the wires can access. As it turns out LabVIEW incorporates a feature that is very efficient and so is perfect for such an implementation. I'm talking about the Data Value Reference, or DVR.

Our approach will be simple. We first create a DVR that is defined to hold the data we need and then put a reference to that DVR into the class data for our *Config Data* class. Then to access that data, we create a family of data access VIs to insert data into, or read data from, the DVR.

The first step on this process is to create another virtual folder in the top-level class named *_dvr* with private access scope. Next, I create in that folder a typedef control named *Config Mgr Data.ctl* that consists of a cluster containing two strings, one for user name and one for password.

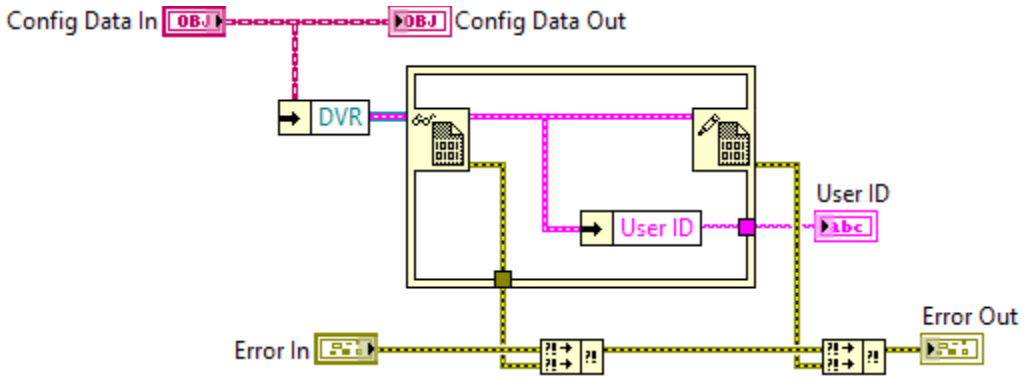


When saving this control I create a subdirectory (also called *_dvr*) to hold it. Next, I create a VI in the same directory (and virtual folder) called *Config Mgr DVR.vi* that contains this code:

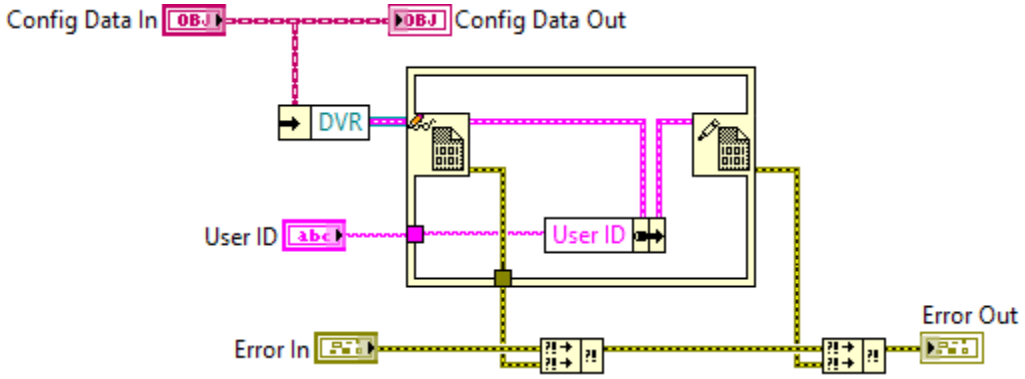


Three comments about this code. First, it has the basic form of a FGV where the variable value is the DVR reference. Because the case that generates the reference is only run once, this VI will always return a reference to the same DVR no matter how many times it is run. Second, the data for the DVR is the typedef I just created. This point is critical. As with UDEs, if you define a DVR reference using a typedef, you can later change the typedef and it won't break the reference. To make this DVR available and inheritable through the class, I copy and paste the reference indicator into the class data cluster. Third, if the data in the DVR needs to be initialized (and not just the reference) this is the place to put that logic as well.

At the same time I created the DVR, I also built a pair of protected scope VIs that I stored in a virtual folder and subdirectory both named *_data access*. Here is what the read VI for the user ID parameter looks like…



…and the write VI…

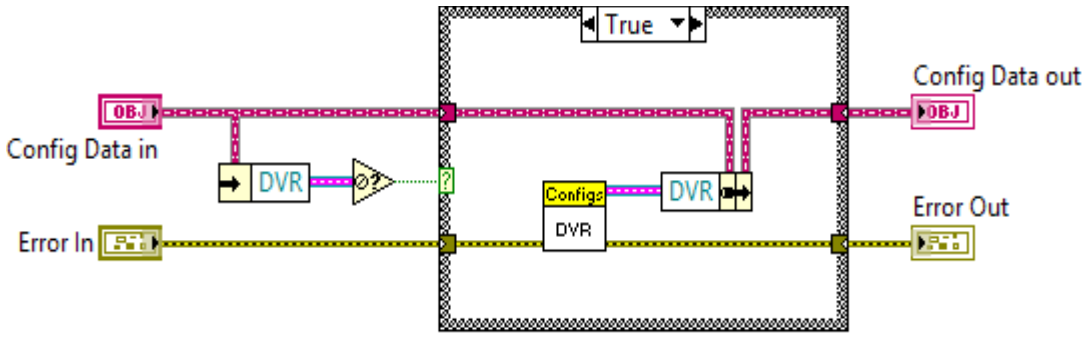


With the parent class handled, I next considered each of the subclasses to see whether they too have some data that will need to be held in common for their subclasses. For each subclass that has such data, I repeat the process I just completed.

1.3.5 Initializing the Class Data

Before moving on we are going to need a way to initialize all the logic we have created, and to do that we will take our first foray into the exciting – though sometimes confusing – world of creating dynamic dispatch VIs. The goal is to create a method called *Initialize New* that causes the DVR in a new instance of a class to automatically initialize itself.

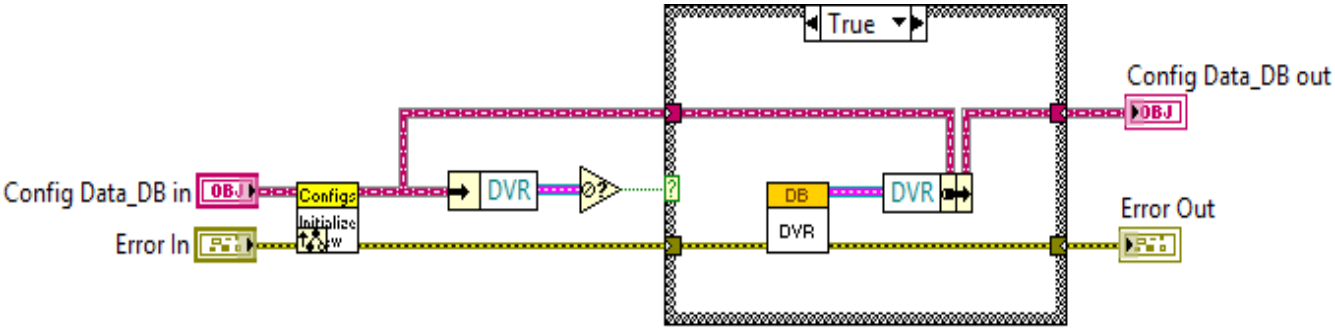
I start by right clicking on the *_protected* virtual folder in *Config Data.lvclass* and from the *New* submenu selecting the option to create a new VI using the *Dynamic Dispatch Template*. I leave the front panel of the resulting VI the way it is, but I change the connector pane, edit the icon and add this code to the block diagram.



If the DVR in the class data is not valid, I call the DVR VI to get a valid reference and then use that reference to populate the class data. If the DVR reference is valid, the false case does nothing but pass on the class data unmodified. When I save this VI, I put it in a subdirectory named *_protected*.

To create the subclass versions of this method, I right-click on the subclass name and from the *New* submenu select the item to create a new *VI for Override* – which is the technical term for what we are doing. We are overriding the parent functionality with different functionality in the child.

However before we get a new VI, LabVIEW opens a dialog to ask us which parent method we want to override. After double clicking on *Initialize New* we get our new VI, also called *Initialize New*. After saving this new VI, (the default location LabVIEW picks is perfect) I modify the code to look like this:



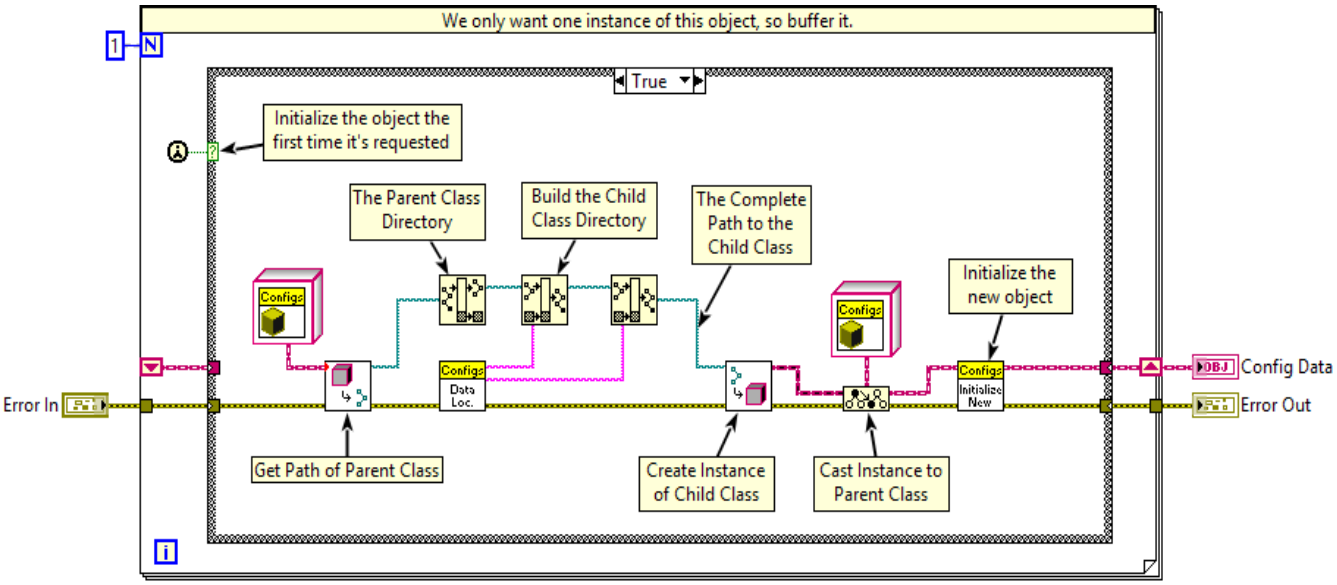
Most of this logic should be familiar because it is the same as we did for the parent. If the child DVR reference is not valid, we initialize it. Otherwise, we do nothing. But what is that funny looking VI in front of the initialization logic?

Object-oriented methodology recognizes that there are going to be times when a method in a child class will need to do what the parent method does, but perhaps a bit more or maybe do it a bit differently. One solution to this situation would be to simply duplicate all the parent code in the child. However that approach would be wasteful. What object-oriented logic does instead is it allows a child to directly call its parent's version of the method. So here, the code first calls the parent's version of the initialization VI and then executes the logic to initialize itself. In the end, both the parent and the child will get initialized.

1.4 Creating Objects

The time is now upon us to start tying this infrastructure together into an organized system. The first thing to sort out is how to create an object of a given type. One way is very similar to what you would

do with a conventional datatype. If you wanted to create, say an I32 value, you would drop down a constant and start using it. In the same way, you can also drop down a class constant and wire to it, thus creating an object. The problem with this approach is that when LabVIEW instantiates a class it also loads into memory all the VIs associated with the class. What you can end up with is a situation where all the VIs for all the classes are loaded into memory, even if you will never use some of the classes. The way to get around that problem is to load classes dynamically as you need them. This is the code I use to perform that operation:



You will notice that the VI has no inputs, save the requisite error cluster. This is because the basic pieces of information that specifies the specific class to be created will be loaded from the application INI file. But why the INI file? Isn't the point of this exercise to get rid or configuration data in that file? Well yes, but there is a bit of a paradox at work here. Simply put, an application can't go to a database it doesn't know it has to find if it should look in the database to get its setup data. That basic piece of information has to be stored somewhere that will always be there, and on the Windows platform you have exactly two choices: the INI file and the Windows registry. Of the two, the INI file is much safer – you at least don't have to worry about a user who is trying to reconfigure things going wild and trashing their whole computer.

We are initially only going to support two options for managing configuration data, so we only need two settings (Text and Jet). The VI that communicates with the INI file reads one of these values from the file returns two values based on what it finds there: A relative path to the location of a class file, and the name of the class file. Thanks to the naming convention we use, these two values are very closely related.

The remaining code loads the target class into memory and initializes it. (Note the call to our *Initialize* New method.) In addition, the resulting object is buffered as in a FGV and output from this VI is an indicator of the *Config Data* class datatype.

1.4.1 Building Out the Remaining Methods

All that's left now is to implement the code that does what our example application needs done. In creating these methods, we have as our guiding principle minimizing the amount of code we have to create by reusing as much code as we can. In other words we need to really spend some time thinking about how to structure our code such that it combines maximum reuse with maximum flexibility.

One good way of attaining that ideal is to allow for multiple levels of dynamic dispatch within the same method. Let's say for the sake of argument that you have a method that will need to be accessible from 10 different subclasses. Furthermore, let's say that 4 of those subclasses all need to do the same thing, an additional 4 are mostly the same but with a few differences, and the last 2 subclasses require fundamentally different logic to perform the same task. You could implement the logic that is common to the largest group of subclasses (the first 4) in the parent and let the remaining 6 override the parent to define their own solutions. This solution would work, but could potentially result in a lot of duplicated code. It depends on how similar the second group of 4 subclasses are to the first group of 4.

In creating the *Initialize New* method we see a far better way to optimize the code: For the 4 subclasses that are similar, we could call the parent method in the child (thus taking advantage of that existing code) and then add a little logic to customize the functionality. This solution will work well in many situations, but one area where it will not is in scenarios where the common parts of the code need to pass data to the unique parts.

To address those situations, a very useful solution can be to write the parent method such that it has a subVI encapsulating the similar, but unique bits, that is itself a dynamic dispatch VI. By providing multiple levels of dynamic dispatch, you create a situation that is easy to understand and minimizes duplication of code. In our example, the 4 subclasses would use the parent implementations of both the method VI and the dynamic dispatch subVI. The 4 subclasses that are similar, but a bit different would use the parent method, but override the dynamic dispatch subVI, and the two that are fundamentally different would override the parent method VI itself.

1.4.2 Finishing the Detail Work

So all we need to do to finish our conversion is create the VIs that will fetch the three pieces of information that we identified earlier – which means we are about done. Why do I say that? Simple, the process we will use for them is the same as the one we have already gone over for the *Initialize New* method we have already created, and here, in review are the steps:

- Create the parent method.
- Update the parent's icon, connector pane and front panel IO to incorporate what it needs.
- On the block diagram, add whatever default behavior you want the method to exhibit if it is not overridden.
- Store the finished parent in the *Interfaces* subdirectory and virtual folder.

- For each subclass that needs to override the method, create an override VI and save it to the default location LabVIEW selects.
- On the block diagram build the code that does what is needed for this particular subclass. Feel free to leverage code from any parent class(es).

So that's about it, if you would like to see a more detailed discussion of this process, please visit *NotATameLion.com*.

Any questions?