A Dataflow Approach to Object-Oriented Programming

Michael L Porter

NIWeek 2014

Abstract

Although object-oriented programming (OOP) can be a powerful tool for the LabVIEW developer, its usefulness and acceptance is often limited by the way it is taught. Much of the existing information is built on examples and approaches derived from programming languages such as C++ or Java. Consequently, the resulting approach runs (implicitly or explicitly) counter to LabVIEW's fundamental programming paradigm: dataflow. The author presents some techniques for applying OO techniques in a way that capitalizes on their the advantages, but without abandoning the very things that makes the LabVIEW approach to problem solving so valuable.

Introduction

Before starting on the main part of this material it will be helpful to do a tiny bit of self-assessment. For the purposes of this discussion, if you are curious about object-oriented concepts, but have no real experience with C++ or Java, consider yourself in "Group 1". If you have used C++ or Java and are looking for more information on how to use object-oriented programming in the context of a LabVIEW-based development environment, consider yourself in "Group 2". Now having made that basic distinction, let me assure you that I will do all within my power to catch those of you in Group 2 up to where the folks in Group 1 are. I put the matter in that way because, as I have learned while teaching LabVIEW itself, students often have an easier time learning about something new if they don't think they already know what it is that I'm trying to teach them.

While on the topic of setting expectations, I need to acknowledge that I am not going to be explaining the mechanics of creating classes or defining methods. For that level of instruction, I would recommend the latest version of the LVOOP training class that NI offers. Unlike an earlier iteration of the course, which was really pretty bad, the current class does a good job of getting you started. Unfortunately, it shares a limitation that is common to many LabVIEW courses: It teaches the mechanics very well, but spends far too little time showing how to program properly.

Due to this limitation, LabVIEW developers who want to gain in-depth knowledge about object-oriented programming must resort to trying to learn what they need from classes and books where the training agenda and content has largely been set by text-based languages such as C++ and Java.

To see why this situation is a problem, we need to consider a common aspect of language.

The Challenge of Language

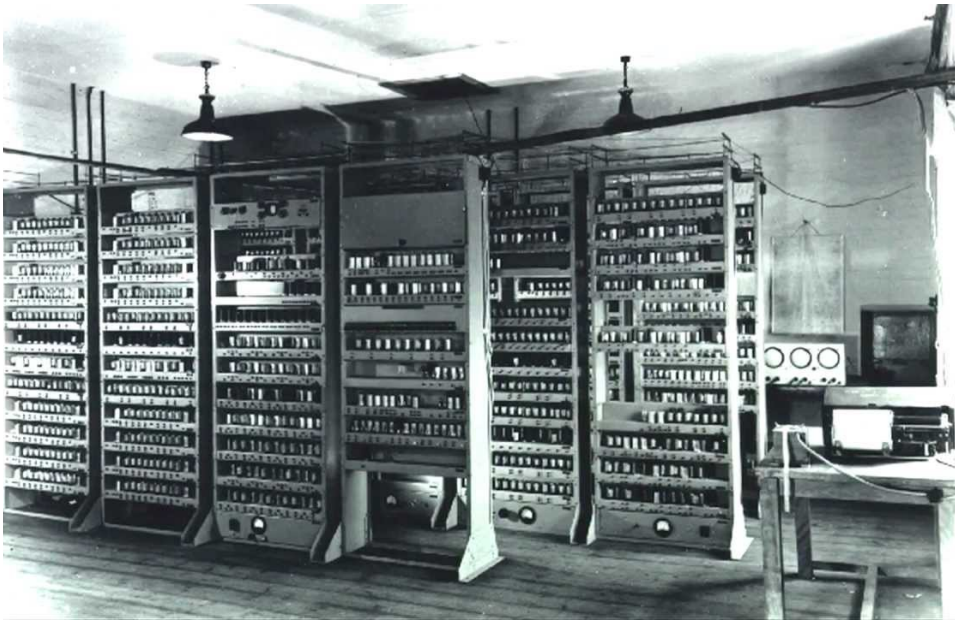
The most obvious purpose of any language (whether human or computer) is communications – which is to say: the expression of ideas and concepts from one entity to another. In human language the point is communications between people, with computer languages the communication is between a human operator or programmer and a computer – or perhaps even between two computers.

The thing to remember, however, is that this relationship between concepts and language is a two-way street. Without getting into the philosophical and linguistic nitty-gritty, language has been shown to affect not only *how* the entities using it express ideas and concepts, but also inform and limit what ideas and concepts they *can* express.

The Origins of Computer Languages

In order to apply that observation to computer languages we need to go back to the beginning, which is to say, the early 1940s. Back then the question of how to program a computer was a non-issue because, in the beginning, computers weren't programmed. Rather, they were custom-built machines designed to compute the answers to specific math problems, and to do so faster than the humans that were hired to do calculations with pencil and paper. In fact, that's why the original machines were called *electronic* computers – to distinguish them from the humans for whom "computer" was their job title.

However, the world changed forever May 6th 1949. That was the day that EDSAC, the world's first fully operational programmable computer, booted up at Cambridge University.



It's first job was to run a pair of programs which were, again, math problems. One generated a table of squares of numbers, the other calculated a list of prime numbers. Of course from that point forward, it

was game on, and the question of the day was: If you can create a machine that can in theory perform any calculation you want, how do you define the calculations that you want it to do? And the related question was naturally, How do you communicate with the beast? Then, as now, the two issues were inextricably linked because the possible answers for the first question are always constrained by the answer you give to the second. In other words, you can't implement a programming paradigm or methodology if you can't effectively communicate that methodology to the computer.

Over the years, advances in computer speed and capability allowed the available techniques for interacting with computers to evolve from toggle switches and push buttons, to perforated paper tape, punch-cards, magnetic tape and disks, and eventually keyboards and video displays based on various flat-screen technologies. Mirroring these advances, the logical content of what we were saying to computers, and how we were saying it, went through a similar evolution. It went from hardware-dependent binary look-up tables, to mnemonic-based assembler code, to "high-level" languages like COBOL, FORTRAN and BASIC.

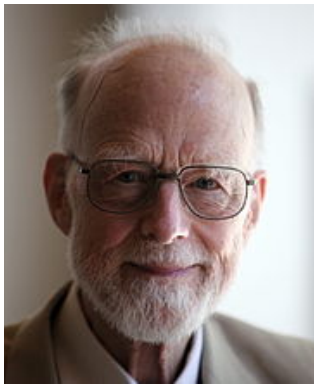
At each step in this process, increasing levels of abstraction allowed the human user to care less and less about how the underlying computing hardware worked. Thanks to abstraction, by the time you got to programming in any of the languages I just listed, you for the most part didn't care what hardware your program ran on. Your only worry was whether the computer spoke the same "dialect" of the language that you did. During this same time period certain languages came to specialize in various types of work. For example, COBOL was used primarily in business, FORTRAN was largely limited to science and engineering application, and BASIC was for beginners.

Eventually, people began to think about how to design universal programming languages to go with their universal computing machines. These efforts gave birth to a long list of so-called "general-purpose" languages including C, Pascal, and eventually C++ and Java. But regardless of the level of abstraction that was applied, the virtual machine that was being defined was for the most part still a *mathematical* machine. Consequently, the one thing that all these techniques had in common was the goal of developing a list of calculations that the programmer wanted the computer to perform. A situation, by the way, that suited mathematician just fine since most mathematicians I have known (and certainly the one to which I am married) have been very ducks-in-a-row, B-follows-A kind of folks.

Of course this tendency to see the world as one big "to do list" wouldn't be a problem if the primary use of computers was still to be a "computer". Unfortunately, most computer programmers today are not mathematicians, and most of the programs we write have surprisingly little to do with mathematics.

A History of Object-Oriented Techniques

In fact it didn't take long for other folks to start realize that while the discipline of mathematics might be helpful, working with computers was fundamentally different from mathematics. While it was true that computers are useful for mathematics, they are also useful for a lot of other things, as well. Pioneers like Tony Hoare (England), Edsger Dijkstra (Netherlands) and David Parnas (Canada) began a quest to uncover the fundamental principles that should govern good software development.



As far back as the late 1950s, people in the artificial intelligence community began experimenting with the idea of software that could represent or model the real world and not just perform simple calculations. We have them to thank for the concept of objects. Over the next several years, what we today call object-oriented programming grew as a logical extension of such foundational design techniques as structured programming and abstract data types. In fact, the fundamental distinction of OOP was of an object as an abstract data type.

The first programming language that was specifically designed to support and express object-oriented concepts came in 1967 when Simula 67 made its debut – the work of two Norwegians (Ole-Johan Dahl and Kristen Nygaard). It is notable that this language was designed to attack the very real-world problem of how to simulate the actions of physical systems, the first step on the road to process control

in the modern sense. Eventually, Smalltalk (1980), C++ (1983) and Java (1995) came along as well. Curiously, these very "advanced" languages still require programmers to express their thoughts as a typed list of words...

However in the early 1980s, while Bjarne Stroustrup was was working on *C with Classes* at Bell Labs, changes were afoot in Austin Texas at a small hardware company named National Instruments. At the time, their main product was a line of GPIB-interface boards so they were naturally trying to figure out how to sell more boards. It didn't take them long to realize that the main thing limiting their market was the difficulty their customers (who were mainly engineers) had in programming the boards to do what they wanted. So they asked their existing customer base a rather simple question: "Assuming anything was possible, how would you like to be able to define the operation of your systems?" And the answer they got was loud and clear: "Block Diagrams".

To make a long story short, to create this block-diagram programming environment, the decision was made to use the Apple Macintosh as a development platform (because it had a 32-bit processor and the requisite graphics capabilities) and implement the new graphical language using object oriented techniques. To appreciate the risk involved in this decision you need to remember that in 1985 there were few commercially-available object oriented programming environments available, and none for the Apple Macintosh. Instantly, the development team's list of deliverables went from:

1. Game-changing revolutionary graphical programming environment

To:

1. Object-oriented programming environment
2. Game-changing revolutionary graphical programming environment

Now since LabVIEW's earliest roots are in the object-oriented world, it would seem logical that it would be an ideal platform for implementing object-oriented concepts, and in fact it is. For one simple example, consider the basic definition of an object as an abstract data type – which is clearly a dataflow concept. A concept, I might add, that text-based list-oriented languages like C++ have a lot of difficulty representing accurately.

From this simple example we can see why, from the standpoint of LabVIEW developers, most of the instructional materials available today are problematic. They don't teach the principles of object-oriented development, but rather the C++ and Java expression of those principles, and just as you can't be really good in a human language until you get beyond translating and begin to think in it, so you can only be good in a computer language when you learn to think in that language. Of course the really good news for engineers is that they already think in terms of dataflow block diagrams. In addition, I would take this commonly-cited truism one step further and assert that engineers already think in object-oriented terms – remember Simula 67? Its use of object-oriented logic was successful because it reflected the way engineers already thought about real-world problems like hardware simulation. So what we really need as engineers is not a new way of thinking about problem solving, but rather to be shown how to apply *what we already know*.

In LabVIEW you have an excellent, perhaps perfect, environment for object-oriented development: a block-diagram based development interface that speaks the mother-tongue of engineering combined with a dataflow structure that can easily and clearly represent the concepts of object-oriented programming.

Dataflow Object-Oriented Programming

So with all the foundation for our discussion finally laid, we can start looking at object-orientation in LabVIEW, and how it can be used. The first thing to note is that there are really two complementary kinds of object-orientation present in LabVIEW.

Implicit OOP

This is what might be called "normal" LabVIEW, and has been largely in its present form since Version 1.0. Although it does not explicitly use classes, its object-oriented underpinnings are visible if you look for them – and if the code is any good. All the basic object concepts are there, albeit in an abbreviated or "lite" form. For example, encapsulation is accomplished through the creation of subVIs, and clusters provide a mechanism for implementing abstract datatypes.

Advantages:

- The block-diagram format is very straight-forward to read
- Its concepts are easy to understand
- Inherently parallel
- Introduces the idea of classes through object references

Disadvantages:

- Not a robust object-oriented implementation (e.g. no user classes and limited polymorphism)
- Poor style can result in a high degree of coupling between modules
- Errors in modularization can result in low cohesion.

Explicit OOP

Beginning with LabVIEW 8.2, classes have been publicly available – though to be honest it took a couple releases before they were really useful. The primary factor limiting developers learning about and using of this form of object-orientation is that NI considers it an advanced topic.

Advantages:

- Dynamic Dispatch (runtime polymorphism)
- Extends modularity in ways that inherently serves to reduce coupling
- LabVIEW classes (especially inside libraries) define a cohesive environment for implementing functionality

Disadvantages:

- Can cause code to balloon in size. Poorly done, classes can become the software equivalent of a Swiss Army knifes. Oh yeah, it can do a lot of stuff, but do you really want to carry it around in your pocket all day because you might need to saw off a tree limb at any moment?
- Done poorly (and it often is) an object-oriented program can become, and is often decried as, write-only code. You may get it running, but forget trying to go back later and figure out what you did by simply reading the code.
- The current implementation of LVOOP exhibits the “only-child” problem (i.e. you can’t really have true sibling classes that share parent data as well as parent functionality),

Basic Principles

Now let's look at some of the fundamental concepts that should guide our work. As we go you will notice that some of these ideas are specifically related to object-oriented development, and some are just good ideas in general and so apply to object-oriented work as well. It is important, however, to remember that I am not talking about design “rules”. Rather I am starting with the assumption that while you all want to learn to be good programmers, the simple fact of the matter is that rules don't make you good – at anything. Moral rules don’t make you a morally good person, traffic rules don’t make you a good driver, and programming rules certainly don’t make you a good programmer.

Rules may be useful for 3 or 4 things, but making you good is not one of them. What makes you "good" is assimilating and internalizing the underlying concepts. Or to put the point is a more philosophical way, “goodness” isn’t about doing, it’s about being. So you need to be the kind of developer that can...

Be Bold

For the most part, these principles are in no particular order. This one, however, is the exception, for without boldness, spending time on the rest of these ideas is just a waste of time. I won't try to define precisely what boldness is – there are, after all, whole books written on the topic. I will, instead, concentrate of the idea of boldness as it applies to our current inquiry.

To be successful in applying object-oriented concepts to LabVIEW we need to be bold enough to consider how these ideas apply to our work without the "overriding" baggage that comes with text-based languages. Just because C++ may have to do things in a particular way due to the language's limitations, that doesn't mean LabVIEW, which doesn't share those limitations, has to do it the same way.

Rather, we need to take the time to learn the principles of object-orientation, without burdening ourselves with a lot of foolhardy rules.

Let the Application Tell You How to Proceed

I have known many developers that have gotten themselves boxed into a corner because they only knew one program structure and they tried to force-fit every project into that structure. Sometimes they take this position due to a lack of experience, sometimes its a lack of understanding, and occasionally its a simple matter of ego.

Regardless of the reason, the better approach is to let each project’s requirements tell you what it wants and needs, while recognizing that requirements are not a homogenous mass. Requirements can, and typically do, vary from one section or module of an application to the next. For example, in one module that is handling a large volume of data, efficient memory management might be the most important consideration; whereas in another that is interfacing with hardware, speed might be of paramount importance.

Therefore, every program, done correctly, will be a mixture of techniques. Your goal as a programmer should be to implement what is needed, not just what you know how to do. Of course you might

complain that this sort of mixed approach will mean that you will need to spend a lot of time learning new things. To such a complaint I would reply, "Maybe, and maybe not…" – you'll see what I mean later.

Develop Tools Not Programs

In the military there is an old expression:

"Amateurs discuss tactics, professionals discuss logistics"

The idea here is that while tactics need to be good to win battles, wars are won and lost based on the organization's ability to get personnel and material where they are needed. In that spirit, I would like to propose a programming version of that statement:

"Amateurs write programs, professionals create toolboxes"

The point of this phrase is to insure that our focus is on the keeping the most important thing, the most important thing. No matter how important and valuable the programs you create for your organization may be, reusable and maintainable code is even more valuable – and the reason is easy to see. Maintainability reduces cost because nothing is static. Requirements change, platforms change, and environments change, so all code will need to be modified someday.

Reusability, in turn, maximizes return-on-investment by allowing you to leverage past work to complete new tasks. But even if your boss is a complete doofus that doesn't appreciate this point, there is still a very practical benefit to you personally in creating a toolbox.

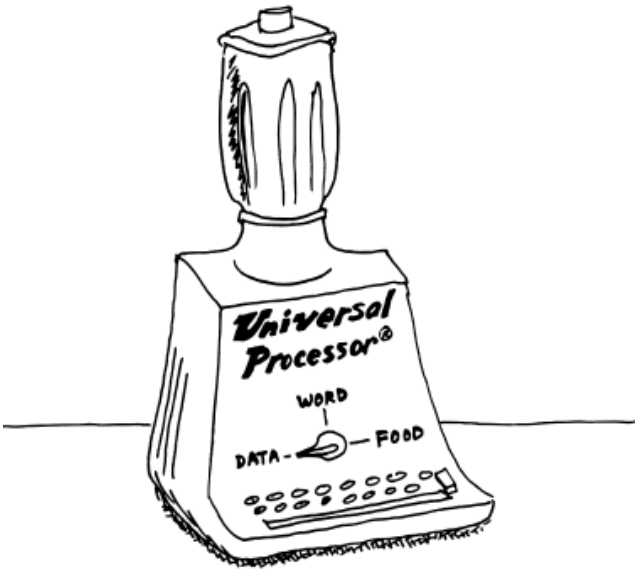
Say you build your first system and it takes you 6 months to complete it. When your boss comes to you a couple weeks later wanting you to design and build another new system I can assure you that he or she is not going to be happy camper when they hear that its going to take another 6 months – they'll probably be thinking more along the lines of 6 weeks. Your bosses may not use the term "return-on-investment" but, be assured, they still expect it.

Only Incorporate Useful Complexity

A common mistake that I have seen in a lot of code is incorporating complexity that doesn't make the

application better (no matter how you define that word), just more complex. There are many examples of what I am talking about, but in terms of object-oriented design, one common problem is seen in applications where everything gets turned into a class. This error results in programs that I guess are technically correct. Of course I could only guess because the applications as whole were almost totally incomprehensible. So we need to ask ourselves, just because something could be a formal class, is it going to be helpful to make it one?

Another common source of complexity is over-reaching. By that I mean that there is a point where a design can become too abstract, too general, and too inclusive. It might seem a laudable goal to try and develop a software framework that will serve for every possible application – and up to about the 80% coverage mark things don't get too bad. However, as you attempt to add support for an ever-greater numbers of special cases, you can find yourself implementing ever-more obscure bits of functionality. Unfortunately, once implemented, this obscure functionality has to be maintained and supported.



By the way, this is a good time to point out that complex behavior of a finished application or system never arises from complex functionality in the application's components. In fact, quite the opposite is true. Paradoxically, complex system behavior is always easiest to create when the individual components are very simple. In general, you need to be very skeptical of complexity.

Let Objects Represent Things

While this point might seem obvious, it apparently is not as I have on multiple occasions seen code incorporating things like an “add-two-numbers-together” object. One cause for such confusion is the curious modern tendency to turn nouns into verbs. So we run into questions like, is a “message” a thing or “messaging” something that objects do?

Another thing that can seem to complicate matters is that some things are concrete – in other words, they have height, depth, breadth and mass – while others are purely logical constructs, like a pressure waveforms or a database record. Just because something lacks a physical form that doesn’t mean that its not an object, logical form is just as important.

Finally, one last thing to remember about objects is that they need to be meaningfully distinct. For example, to represent my car you might define a class of object called “car” with a subclass “hyundai” and a further subclass “elantra”. The question is, would it be worthwhile to extend this hierarchy further to provide separate subclasses for red Elantras and blue Elantras? Well it depends, “meaningful” is a highly contextual term. For a typical auto mechanic the answer would be “no” because when you are fixing or tuning engines the color of the paint on a car isn't a meaningful distinction. If, however, the context for your application is a body and paint shop, the answer might be “yes”.

Reduce Redundancy

In the field of database design, folks talk about how to reduce or eliminate redundancy – a lot. The problem with redundancy (which is defined as the duplication of data) is twofold. First, the duplicate copies take up memory unnecessarily. Second, if a database has redundant data in it, the logic has to be able to deal with the situation where the various duplicate copies are not all the same.

Naturally the same concept applies when talking about code. You want to minimize duplicated code to save on memory footprint, but also to simplify the work involved when the duplicated code needs to change. Most people have no problem with this idea, but with object-oriented work it can be easy to mindlessly create multiple subclass methods that all do the exact same thing. The way to address this issue is to always give careful consideration as to where you put the logic that implements methods.

Realize That Organization Makes Life Good

A skill that the human species developed many, many millennia ago was the ability to identify or recognize patterns in the world. Now while pattern recognition can sometimes run amuck (as in the movie *A Beautiful Mind*) this ability is crucial in gaining understanding. Consequently, if our goal is avoid creating write-only code, it behooves us to organize our work in such a way as to make it easy to grasp – preferably without an instruction manual. So here are some organizational ideas that, over the years, I have found to be useful:

- Be consistent in naming. Don't give two logically distinct entities the same name, or use two different names to refer to the same logical entity. If you are part of a development team, it might be worth your time to create a naming lexicon for entities in the team's software. This simple convention can result in huge gains since you are no longer having to waste your creativity either thinking up names for things, or figuring out what a coworker meant when they use a particular label.
- Give classes hierarchical names. This kind of structure makes it easy to see the logical relationships between classes by improving readability. I use the underscore character to delimit levels in the class name, as in: `Instrument_DMM_HP34401`. It is easy to see how sorting a number of classes utilizing this structure by name would make it easy to visualize the class relationships.
- Place distinct class hierarchies in separate libraries. As we shall see in a moment, access control is important and putting all the files related to a given class hierarchy in a library enhances your ability to control access to subclasses.
- Create a directory structure that mirrors the class hierarchy structure. Again, because I have defined this convention I don't have to waste even a moment wondering about where to store VIs on disk. I have already decided and simple have to reuse that decision.

Automate Processes

I find no end to the irony that people like us who make our living automating things for other people, so rarely take the time to automate things for ourselves. Do you find yourself doing the same thing over and

over again? Are you getting any real benefit from doing these tasks manually? One very powerful technique that LabVIEW makes available is scripting. Basically, scripting (also sometimes called metaprogramming) is the process of writing LabVIEW code that, when run, writes LabVIEW code. The technique is not simple and is (surprise, surprise) poorly documented, but it offers significant potential for streamlining the mundane, repetitive tasks that always seem to be jumping in the way of your creativity.

A related automation technology is VI server, especially the parts that allow you to directly manipulate the structure and contents of projects. Finally, don't forget that there are no rules against writing utilities to automatically do things like creating directory structures like the one we just discussed for keeping classes organized, building applications or programmatically maintaining the contents of enumerations.

Hide Stuff

What I'm talking about here is the computer science concept of *Information Hiding* and in the object-oriented design approach it has some important practical implications:

- The purpose of the class data is to hold information that is private to that class and its subclasses. While there is often the need to uniquely identify particular objects or populate a class with runtime data, this information can often be read from the database storing setup information, and not passed into it as parameters – this is especially true with the higher-level classes of a class hierarchy. This idea implies yet another type of hierarchy in your application. In addition to a structural hierarchy and a functional hierarchy, there is also a data hierarchy.
- The top-most class in a given user-defined class hierarchy should form the interface that exposes the hierarchy's functionality. Specifically, the application calling the class should never call a subclass directly. In fact, in a well-designed application, such direct access should never be necessary. Packaging the class hierarchy in a library is helpful in enforcing this constraint because it provides an additional layer of access control.
- Subclasses should be loaded dynamically. One of the things that can cause code size to balloon is having classes loaded into memory that are not being used. Dynamic loading is easy to implement and can reduce the application's runtime memory footprint. A further simplification

can be realized from utilizing the hierarchical naming convention that was mentioned earlier.

Getting Down to Particulars

Finally, let's consider how these principles can fit together. By the way, before you ask: Yes, we are going to gloss over a lot… For instance, we won't consider in detail the design or structure of the non-class portion of applications. For issues related to overall program architecture, I do however recommend a 1999 paper by Lynn Stein. She approaches the issue of computer architecture from the perspective of robotics and I believe that much more attention needs to be paid to some of the ideas that she discusses – perhaps a NIWeek session for next year.

While talking about glossing over things, one more thing upon which I will not elucidate is iteration. By that I mean that I will walk us straight through the design and development processes. The thing to remember is that if you see the various steps we will cover as metaphorical pearls, the "Golden Braid" that is holding them together will often double-back on itself and loop around. To get the most out of any process, you must be willing to go back and rework earlier representations and logic to incorporate insights that you gain while working on the project.

Where's the Objects?

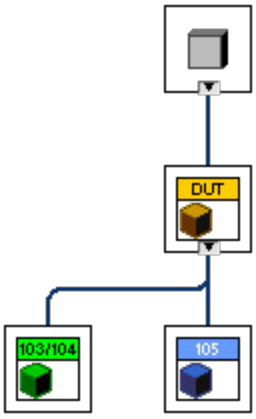
Given that one of the basic principle that we discussed before was that an application will typically incorporate a mixture of techniques, a major consideration has to be, "Where do we look in our applications to identify objects that we can turn into useful classes?" There are many places that experience will reveal to you, but let's consider three of the most significant to get us started.

Real-World Objects

The first, and most obvious, place to look for objects in your application is with what the application itself is doing. For example, I once worked on an application that involved the testing and calibration of meters that measure the flow of a gas. The calibration portion of the application was particularly important because the metered flow was what provided revenue for the meters' eventual owners.

The meter manufacturer makes several different models of flow meters that all have different flow ranges and accuracies, but use the same system for testing and calibration. Moreover the basic test operations remain constant because industry standards specify how the meters are to be tested. The only variability in how the system operates from one model of meter to the next lies in the commands that the test system used to interact with the meter under test (MUT).

In the real world, the test and calibration process consists of passing physical objects of different types (the meters), through a process that does essentially the same thing each time it runs. It can make sense, therefore, to model that structure in software by creating an application where the core application logic is constant but it operates on different software objects that flow through it.

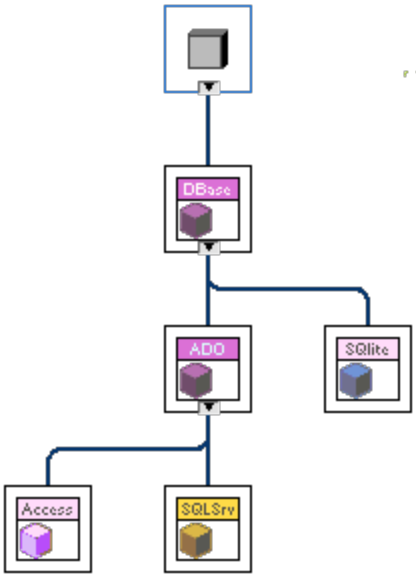


Functional Objects

A second place that you can look for objects in your application is with things in the implementation that you choose to embody the application's logic. For example, in the preceding meter test and calibration system, there is nothing inherent in the test process that would require a database, but let's say that being a smart developer you decide to include one to store program setups, operating parameters, object definitions and test results. Moreover, let's say the customer is enthusiastic about the idea because there are already plans to centralize their configuration management and test results in a network database.

In this situation (or even if the customer isn't enthusiastic about the idea) it can be valuable to abstract

the database connection as an object that provides methods implementing the database interactions that are needed. This sort of structure can be valuable because database locations, database management systems, database structure all can – and do – change over time. Encapsulating the database in a class structure can effectively hide the details that could cause the calling application to break by defining a clear and consistent API. In addition, when changes do need to be made later, the class structure clearly identifies where the changes need to be made.

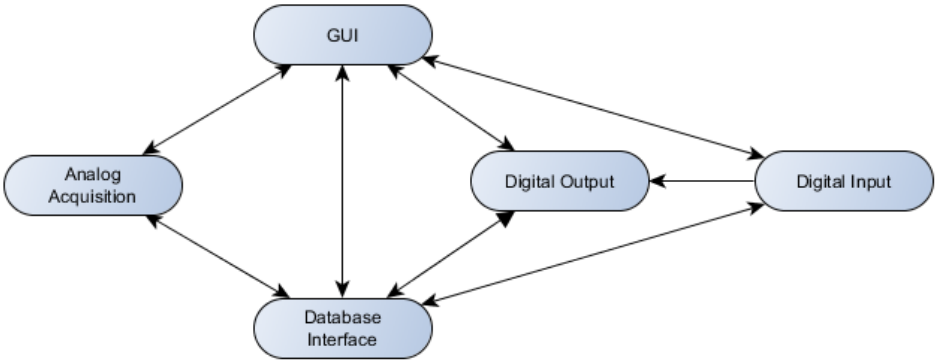


Structural Objects

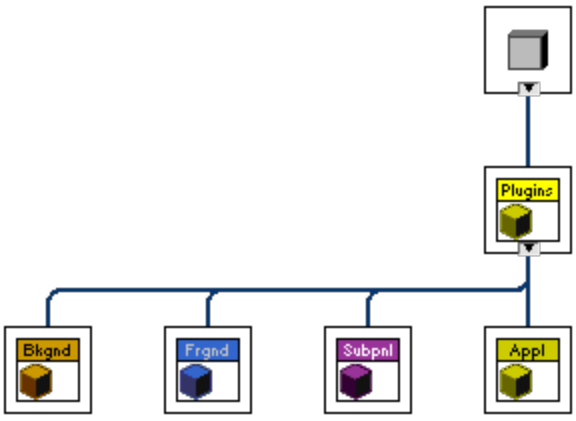
A third place to look for objects can be in the overall structure of the application itself. We have already asserted that smart developers make use of databases to manage the data in their systems. Smart developers also break their applications up into distinct pieces that run in parallel either inside the same instance of LabVIEW, or as separate executables that communicate using a standard protocol such as TCP/IP. With such a high-level architecture, these parallel bits can also be seen as objects: structural objects.

Creating such a class structure, and appropriate methods to go along with it, can turn the potentially onerous tasks of managing the operation of, and interactions between, the various parts of an application into child's-play. If you think about it, there are only so many different kinds of modules

possible, and there are only so many things that you can do from the management level to any these modules.



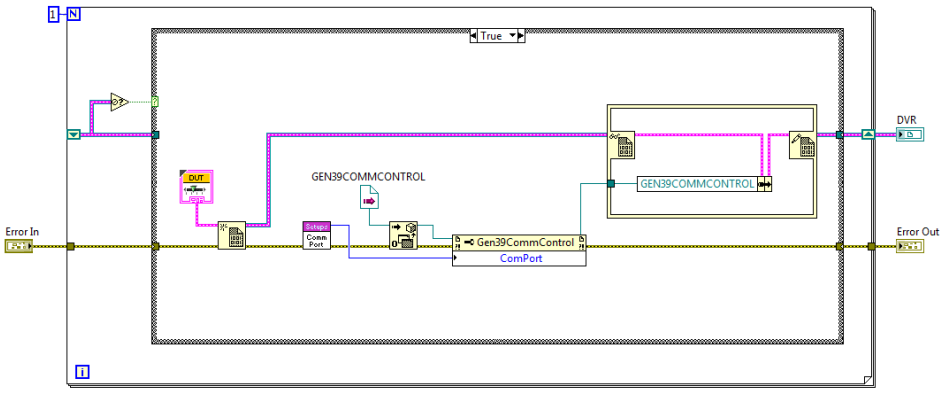
Simplifying these high-level type operations can offer other advantages as well. For example, a problem that often comes up with architectures utilizing multiple parallel processes is how to pass data from the acquisition process to the GUI process. Proper handling of the structural objects in the application can simply make the problem go away by removing the need to pass data altogether. All you have to do is define the front panel of the acquisition process to go into a subpanel in the GUI process. Now you have no communications problems because there is no communications going on. Although the GUI looks like a single monolithic interface, the user is really looking at the front panels of the processes that are actually acquiring the data.



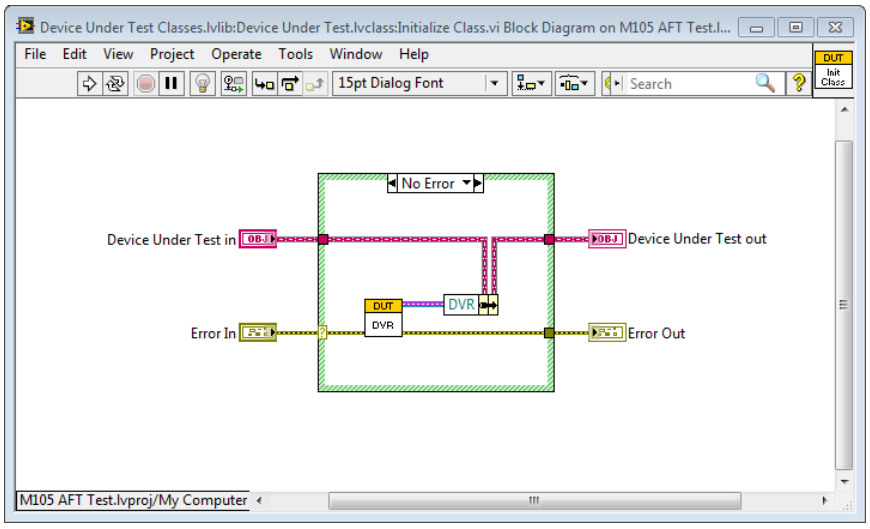
Implementing the Object Classes

So having defined the principles that should guide our work, and the places that we can look to find useful objects, we can begin to consider how to implement the object classes to get the most from them. The first thing we need to consider is the number of objects that we will be instantiating at one time. In the gas meter example given earlier, the system is designed to test just one meter at a time so we will only have one meter object instantiated. While this situation will often be the case, there can be situations where we might need to have more than one instance of a given type of object.

For example, the meter manufacturer could decide to fundamentally redesign the entire system such that it would test multiple MUTs simultaneously, and so would have multiple meter objects to manage simultaneously. To create sibling objects that are capable of sharing parental data, as well as logic, all you have to do is store the needed parent data in a DVR, and make the parent class data a reference to that DVR.



In the data access VIs for the parent objects, you then use an in-place node to access the DVR's contents.



Because the parent data is stored outside the parent objects, all parent objects that have access to a given DVR reference appear as though they are the same parent.

Object-Oriented Functional Decomposition

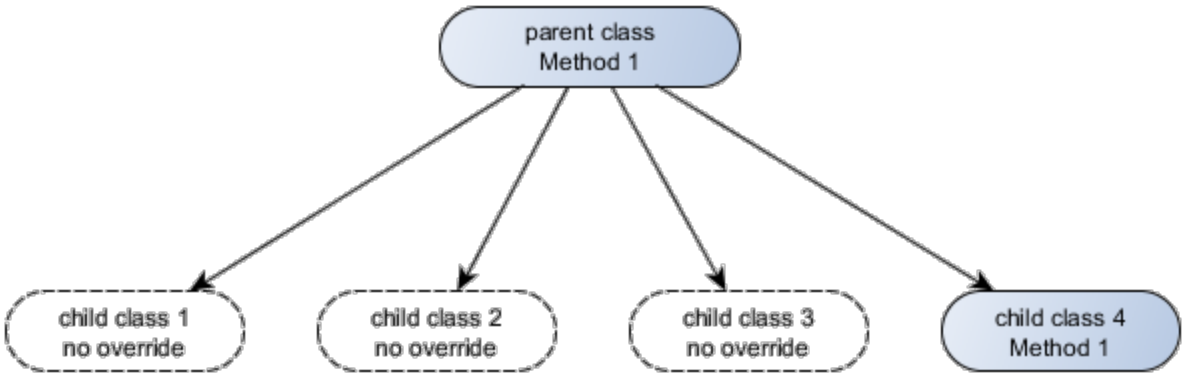
Now that we have our top-most object classes defined, we need to start taking them apart. Continuing with our meter example, we need to look for the natural divisions occurring within that broad category. Or to put it another way, what useful distinctions exist within the meter class that we could turn into subclasses? Considering the client's product line we immediately see two broad types of meters: those that meter gas in metric units and those that use imperial units. Moreover this basic selection has significant impact in terms of the way the test results are setup and the results evaluated. So we now have two subclasses: *metric* and *imperial*.

Repeating the process, we now look at each of these subclasses and see if we can identify any useful distinctions within them. At this point, we find that the only useful distinction left is the one between individual models.

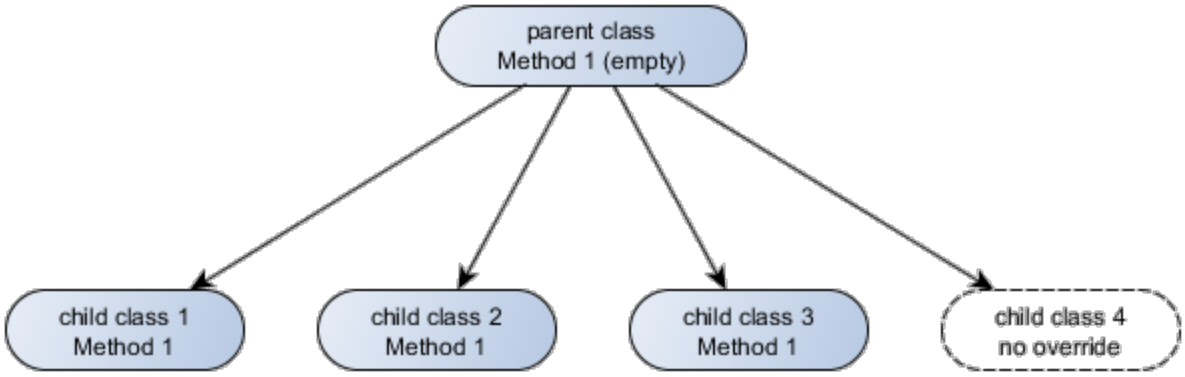
Obviously, to do this work effectively, we will have needed to complete our analysis of the application requirements and have the non-class part of the project pretty well mapped out. The information we will be needing specifically are the operations that the class interface will need to support (this will give us our methods) and the data that will be needed to perform those operations. An important thing to

remember when creating the list of methods that we will be needing, is that we are looking for a superset of actions that includes *all* the possible subclasses. If a given method is not needed for a particular subclass, you have two options:

In the case where the required logic for a particular method is usually the same, you can put the logic in the parent class and then override the parent method with one in the subclass. This override method can either do nothing, or perhaps does something different that this particular subclass needs.

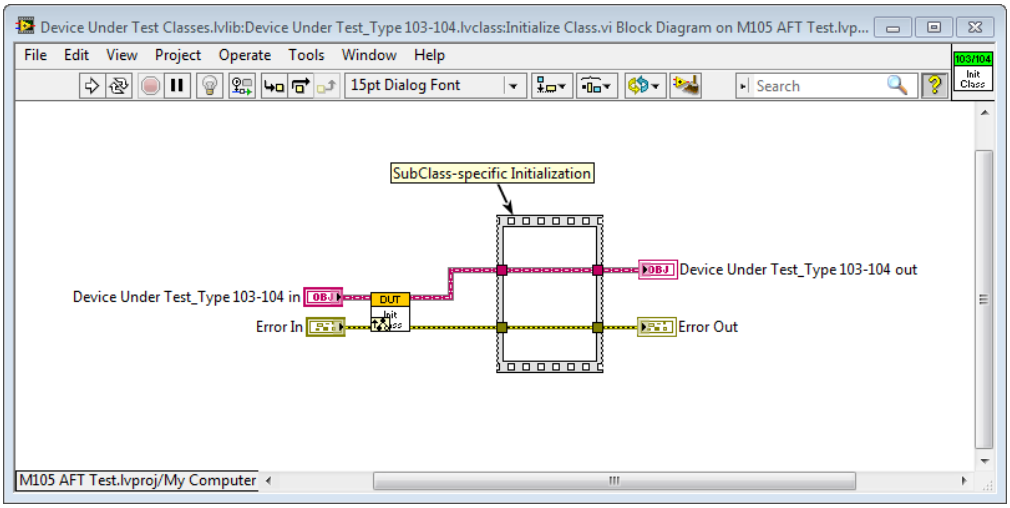


Alternately if the parent method does nothing and all the "magic" happens in the subclasses, you can choose to *not* override the parent method in the subclass and let the empty parent method handle the call. The big point here is that within the context of a single application, you need a consistent criteria for deciding how you are going to handle these situations.



And while you're at it don't forget that you can call a parent method from within a subclass. This tactic

can be particularly effective if a subclass needs to basically the same as the parent, but with a few added operations before of after the parent's functionality. For example, earlier we talked about using a DVR to hold the data associated with a parent class. You can extend this idea to subclasses like this:



Here, the override VI for a subclass calls the parent method to initialize itself and then performs whatever logic is needed to initialize its own private data. Extending this logic through multiple levels of a class hierarchy creates a very detailed, well-formed data structure that is essentially self-initializing.

One last critical step before going on is to grab a cup of your beverage of choice and sit back and think about what you have done so far. Remember what we covered earlier about language informing and limiting what can be expressed? Well, the same is true of the logical "vocabulary" that was just defined. So before going on, take a few minutes to consider the application as a whole and think about the aspects of it that you aren't sure about. If some of the assumptions that you made about the system turn out to not be valid will you be able to modify the design without redoing it all? Likewise, from your conversations with the customer, what are the things that are likely to change over time? It is certain that out meter manufacturer will add new types of meters, but what about other possible changes? Can the design incorporate scope changes gracefully? Is it clear where in the design the modifications for each potential scope change should go?

And if somebody comes by and asks what you are doing drinking a cup of coffee and staring at your computer screen, just tell them you are saving time.

Building the Basic Structure

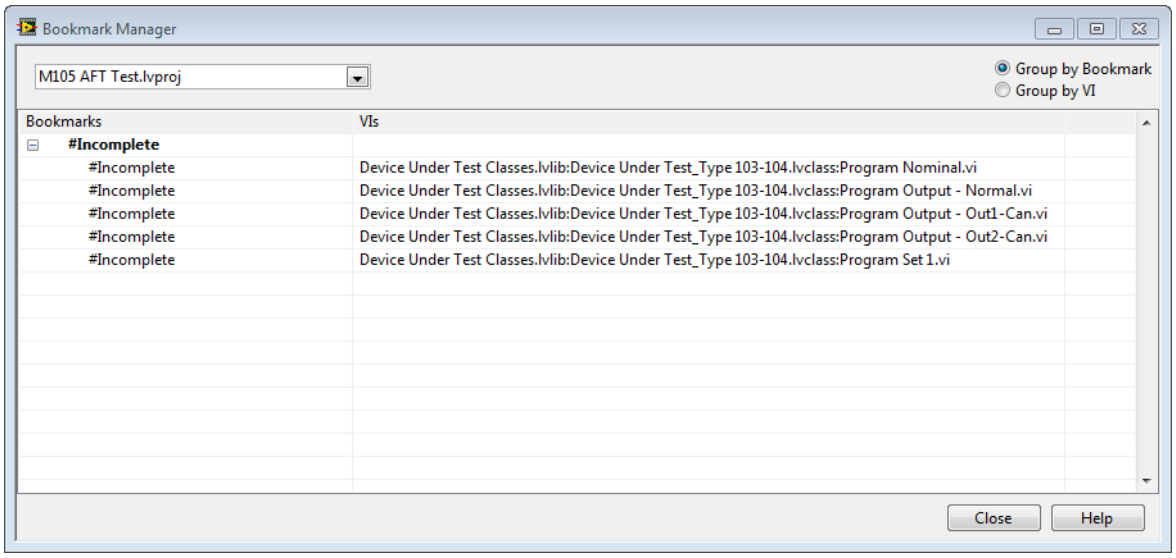
Obviously, the first step here is to create the required classes, edit the class icon banners, assign the inheritances and finally define the data for the classes. If you are going to need virtual sibling objects, now is a good time to define the parent data DVR and implement the mechanism that you will be using to make the DVR reference available to the parents that will use it. If you do create a DVR to hold the parent data *always* make the DVR datatype a type definition.

Next, in the top-most class of each class hierarchy, create a VI to instantiate required objects based on program inputs and, using the dynamic dispatch template, create methods to implement the various operations that the application will need to perform. As you create these top-most parent methods – many of which will probably only serve as a call entry-point and so functionally do nothing – it can be a good idea to leave notes on the block diagram concerning what the method does. To save time later, also be sure to edit the icon for the method now. The icon that you give the method now will be inherited later when you create override VIs in the subclasses.

Creating the override VIs comes next, however, when creating these I like to go through and create all the override VIs without worrying about implementing the functionality that they will encapsulate. Rather I simply place a free label on the block diagram containing the text:

```
#Incomplete
```

The result is that when I have finished creating all the empty override VI's I can go to the LabVIEW bookmark manager and it will give me a nice todo list of all the things I still have left to implement.



In addition, the empty override methods can provide a quick and easy mechanism for determining whether the correct subclasses are being called.

In Closing…

With this work done, you have the basic object-oriented skeleton in place. All you have to do now is fill in the blanks with the required functionality and that, as they say, is all there is to it, but before stopping I want to leave you with a couple final thought about programming approaches.

First, after all the talking and thinking we did about how to do a dataflow object-oriented design, the actual implementation may have seemed rather – well – anticlimactic. However, I would like to assert that in a sense that is as it should be. One thing that I have learned in doing this work for the better part of 30 years, is that the hardest part of any job is learning how to think about it properly. Once you have the correct mental vision of the finished system, the implementation often just falls out on the floor.

Second, I want to leave you with some very sage advice from Dr Parnas that he gave in a 1999 interview:

"*I would advise students to pay more attention to the fundamental ideas rather than the latest technology. The technology will be out-of-date before they graduate. Fundamental ideas never get out of date. However, what worries me about what I just said is that some people would think of Turing machines and Goedel's theorem as fundamentals. I think those things are fundamental*

but they are also nearly irrelevant. I think there are fundamental design principles, for example structured programming principles, the good ideas in "Object Oriented" programming, etc."

David Lorge Parnas

Dr Parnas' basic point here is simple: Programming fads come and go, but good ideas are forever. As professionals we need to learn the good ideas and hang onto them, regardless of how the next generation tries to dress them up in new clothes. This is why to "keep current" you don't necessarily have to learn a lot of new things. There are, after all, only so many fundamental ideas.

A second point is that we shouldn't become too attached to a particular favorite buzzword because sooner or later it will fall out of fashion. For example, we are beginning to see that object-oriented this-and-that is already on its way out and another set of principles are on their way in. Think that is an overstatement? Consider this:

"Object-oriented programming is eliminated entirely from the introductory curriculum, because it is both anti-modular and anti-parallel by its very nature, and hence unsuitable for a modern CS curriculum."

Robert Harper - Professor Carnegie Mellon University

That, gentle listeners, is what is known as the "handwriting on the wall". Currently there are a couple contenders to be the OO-replacement. One is based on a branch of mathematics called lambda-calculus that was first postulated in the 1930s. The other draws inspiration from the 1950s, and specifically the languages FORTRAN and Lisp. One potentially positive note for the years ahead is that unlike the situation when object-orientation burst on the scene as a new kind of language, most people so far seem to be seeing these "new" paradigms as design approaches that any competent language should be able to support.

For example, LabVIEW can already supports both of these techniques – and has since 1986.

Bibliography

Software Fundamentals: Collected Papers by David L. Parnas, 2001, ed. Daniel M. Hoffman, David M. Weiss, Addison-Wesley Professional, ISBN-10: 0201703696

Gödel, Escher, Bach: An Eternal Golden Braid, 1979, Douglas R. Hofstadter, Vintage, ISBN-10: 0394745027

Challenging the Computational Metaphor: Implications of how we think. L.A.Stein, Cybernetics and Systems, 30(6), 1999.

Thinking Forth, 1984, Leo Brodie, Punchy Publishing, ISBN-10: 0976458705