

Object-Oriented Messaging, Command Pattern, and State Pattern in LabVIEW

Paul J. Lotz

Lowell Observatory, 1400 W Mars Hill Road, Flagstaff AZ 86001

ABSTRACT

After motivating the investigation by looking at past approaches, we present LabVIEW examples that demonstrate messaging using objects, an implementation of the Command Pattern, specifically for an XML configuration files utility, and an implementation of the State Pattern, including an example deployed on a real-time target. In addition we briefly mention the concept of Model/View/Controller. In an appendix we compare these solutions to selected popular approaches.

1. First, some history

I think it is useful to motivate the discussion by explaining a bit about how we have arrived where we are.

1.1. “LCOD”

I started to think seriously about these issues after reading *A Software Engineering Approach to LabVIEW*¹, which introduced me to “LabVIEW Component-Oriented Design” (LCOD). While I don’t remember everything in the book (and I no longer have access to it), I do remember that the authors stressed the concepts of *high cohesion* (although I don’t recall if the authors used this term for it or not) and *loose coupling*. In the broadest terms these mean 1) keep things together that logically belong together and 2) separate things that are logically separable.

I started from the examples and began an implementation of a component-oriented system using such an approach.

First, I understood each component should operate independently of its peers. So I created parallel loops with queue-based communication between them.

Then I focused on the message content. A particular queue can send a message of a single Type. One can send all the Boolean-typed messages, for instance, on one queue—but this isn’t a logically convenient grouping at all, of course. So I decided to explore (as have others) sending messages with queues configured to send Variant-typed messages. On this model the sender flattens the message data to a Variant on one end and the receiver unflattens it to the original type on the other. To do this the receiver must know the message type, so I defined a high-level typedef for the message that consisted of 1) a string with the message name and 2) a variant with the message content. The receiver looks up the message content type based on the message name.

¹ J Conway, J., & Watts, S. (2003). *A Software Engineering Approach to LabVIEW (National Instruments Virtual Instrumentation Series)* (1 ed.). Upper Saddle River: Prentice Hall PTR.

An extension I tried is to write the data to a functional global variable (FGV) or a set of FGVs. Now a functional global variable can have some behavior besides reading and writing its data, and those kinds of FGVs some developers call action engines. An FGV can perform some operations with or on its data, possibly changing state as a result.

Anyway, this was the first major step toward our current situation. I will attempt to explain how and why we moved from this to where we are, and explain why I think our current solution is a logical progression from this.

1.2. Object-Oriented analysis, design patterns, and Java messaging

The next step in my journey was reading *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*². The author talked a good deal about the principles of high cohesion and loose coupling (among others), but now in an Object-Oriented context, and included a discussion on design patterns. The concept of a software component plays a leading role in this book as well, but now it becomes easier to see how to implement such a thing.

In the world of objects we design for high cohesion and loose coupling by assigning responsibility for closely related behaviors (that is, operations) to an object. As far as is practicable the object owns the data it needs to perform its operations.

I read more about design patterns in *Design Patterns: Elements of Reusable Object-Oriented Software*³ (a brilliant book that started the design patterns field). It turns out that formal design patterns offer some advantages.

- One doesn't need to invent a design pattern—a design pattern by definition is something that others have used successfully—and carefully described the implementation.
- There is a catalog (actually catalogs) of design patterns, in which each has a name, guidance on how and when to use it, and an example.
 - Why is this important? Well, because:
 - Everyone in the field knows what we mean when we say State Pattern.
 - There is collective effort on refining the implementation of the patterns, instead of each developer trying to perfect his or her own things.

I think these are incredibly important benefits to using these Design Patterns.

Notes:

² Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)* (3 ed.). Upper Saddle River: Prentice Hall PTR.

³ Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2005). *Design Patterns*. Toronto: Addison Wesley.

- Design Patterns are *not* always “simple.” Some are highly stylized. They are, I think, generally *efficient* and *elegant*, and so far I have found that they are pretty simple to implement once I understand them.
- Design Patterns are not libraries one can call, but patterns one can follow. They are reusable in that they offer *mature* ways of doing things.
- Design Patterns are appropriate when they are applicable. We shouldn’t attempt to use them for everything!

A component is a replaceable module that encapsulates its contents and defines its behavior via an interface.⁴

Finally I read about Java messaging systems that allow applications to send objects as messages. Object messages can be arbitrarily complex and have very precise definitions in an inheritance structure (we will see why this is so important in a minute). Moreover, we can decouple the message content (objects) from the messaging system, which I think is a subtle but extremely important point.

I decided there was considerable power in all of these and decided to see if I could do the same sort of thing in LabVIEW, noting that the common denominators for creating scalable systems seemed to be to *encapsulate* data and behavior in *components* and to enable some sort of *messaging* between these.

2. Current examples

Now we present some examples of what we have learned to do with objects in LabVIEW.

2.1. Object-Oriented messaging in LabVIEW

After some false starts I figured out a way to send objects in LabVIEW by flattening them. The key points follow.

2.1.1. Send messages using networked shared variable communication (or other messaging system)

We decided to use networked shared variables to handle the messages. This isn’t strictly necessary to send objects, of course—one can use TCP/IP methods, queues, or just about anything, *because the messaging paradigm is independent of the content*, but we will discuss the design in this context.

We are implementing a component-oriented design. It’s nice if we can implement a component in its own loop, nicer if it resides in its own method, still better if we can define that method on a class defining the component, and even better if we can deploy it anywhere. To decouple our components as much as possible we want our components to be runnable anywhere (motivating networked communication) and communicate without knowing anything about the other components or implementing a server (motivating a publish-subscribe paradigm).

⁴See Larman, p. 654.

Briefly, we think shared variables offer the following advantages:

- Shared variables are the closest thing in LabVIEW to a ready-made implementation of something akin to the Object-Oriented Observer Pattern (publish-subscribe communication). [Briefly, the use of a publish-subscribe paradigm allows us to implement stand-alone components, each of which has its own state machine that responds to data received *asynchronously* on topics to which it subscribes, and it publishes data to which one or more components may subscribe. A message broker handles all subscriptions and message passing, so the components interface only in terms of data. One component need not know that another component even exists.]
- Shared variables are part of LabVIEW—we don't need to spend valuable application development time creating our own messaging system (which can be very expensive to do well!). Hopefully this also means National Instruments maintains the implementation, fixing bugs and adding features.
- The shared variable API easily lets us decouple the messaging system implementation from the application code.
- Networked shared variables work just the same from the code point of view whether we deploy the communicating applications on the same machine or at different locations on a network. This means my controller can respond to a message sent from a GUI on the same machine or a different machine, or from another software component running anywhere on the network. This turns out to be incredibly powerful (at least if security concerns don't prevent using this).⁵
- In our case we have the Datalogging and Supervisory Control (DSC) module, which gives us a built-in logging solution, among other things. (Again, writing as good a logger would be very difficult. I'm certainly glad we're not trying to maintain the Citadel code.)
- Finally, again with the DSC module, applications can handle shared variable value change events. [Aside: I certainly think this particular functionality should be part of the LabVIEW core, and I think it would be to NI's advantage to make it so. How can someone have access to shared variables but not shared variable value change events? If more NI customers successfully created component-based applications using shared variables, I suggest NI would sell more LabVIEW licenses.]

Now there are some disadvantages to shared variables, notably that the shared variable engine and the Citadel historical database still aren't the most robust things ever and in my opinion the various APIs are lacking in consistency and usability in some respects, so I want to stress this is a good (and improving) but not a perfect solution. On the other hand shared variable performance (speed and reliable message delivery) is really quite good now that they use TCP.

⁵ Note that we can point to the URL for a shared variable in a different project, so we can put our components in independent projects if we wish (and we often do!).

2.1.2. Flatten the object

Each shared variable has exactly one type (much like a queue), and while LabVIEW does allow developers to select a custom type, that type cannot be a LabVIEW class type. So...we flatten the LabVIEW object we want to publish—to XML or simply a string⁶. [Note: XML is human-readable but I recently encountered the issue that the Flatten To XML and Unflatten From XML functions don't work correctly if the class definitions reside in a LabVIEW library (.lvlib). Using the Flatten To String and Unflatten From String functions is currently more robust. Hopefully NI will improve the implementation of XML in this and other ways in the near future. This is actually at the top of my list of desired new features.]

2.1.3. Unflatten the object

Much like the queue with variants method, the receiver must unflatten the message to a specific type. So (as we'll show), the receiver must know:

- the type of data associated with the message
- the definition of that type.

2.1.3.1. Know the type

Now here is where things really start to get cool. The receiver only has to know the *generic* type of the message. In particular, we can generalize a parent message class and the recipient can cast the received message (one of the children) to the parent type. An example (see under the Command Pattern Example) will make this much clearer.

2.1.3.2. Share the definitions

We address the definition question by storing the definitions in a single location under version control. As long as we build the sender and receiver applications using the same version of the shared definitions (which is the natural thing to do) we know the two can understand one another.

OK, now let's see object messaging in action in a real example.

2.2. *Command Pattern Example*

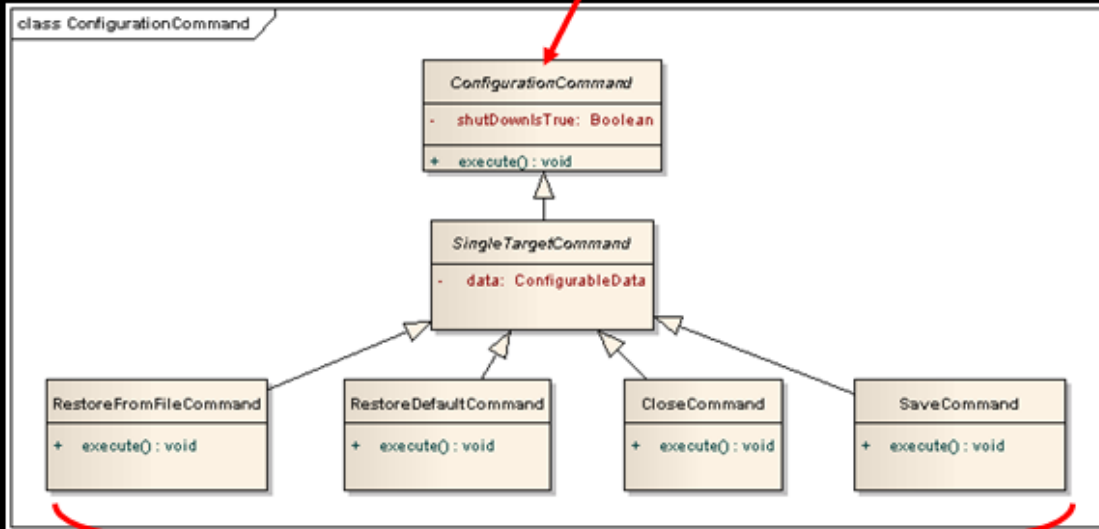
2.2.1. A command is an object

In the Command Pattern each command is an object. There is an abstract (never instantiated) class that represents the top-level command and then child classes that represent the actual commands that we can send (possibly with some layers in between). For an example I offer a portion of the command hierarchy for an XML-based configuration files editor I created:

⁶ This approach doesn't work with RT-FIFO-enabled shared variables since these don't support the String type.

Each command is an Object

Abstract top-level command

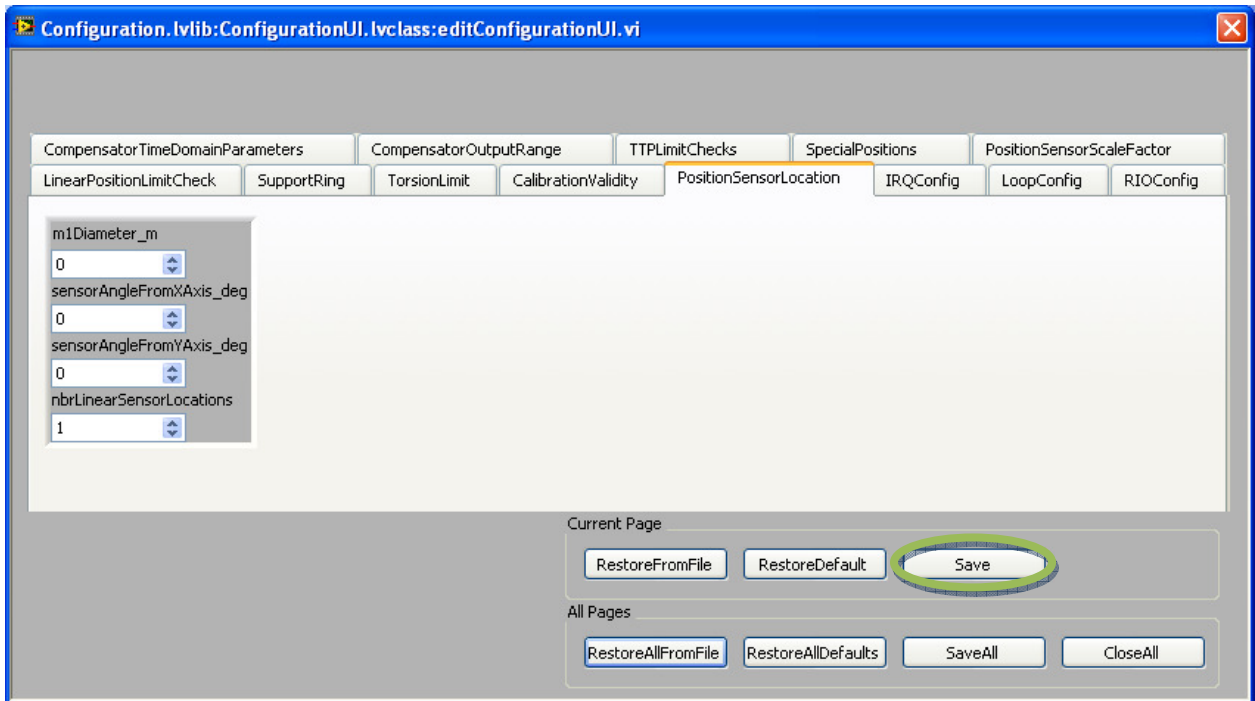


Concrete leaf commands

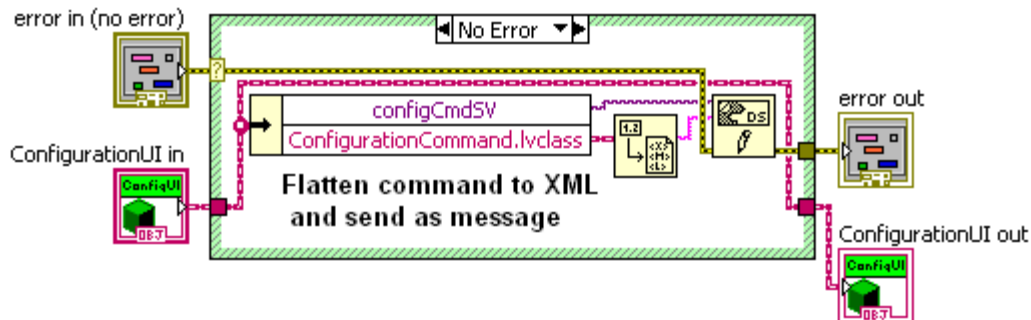
The leaf classes on the bottom row are the actual implemented commands. (Note that each command class has Command in the name.)

2.2.2. Sending the command

So, how does the application work? Let's say a user clicks the Save button on the user interface.



The user interface code handles that event and publishes the SaveCommand object to a shared variable (flattening it first—to the most general type for the topic).

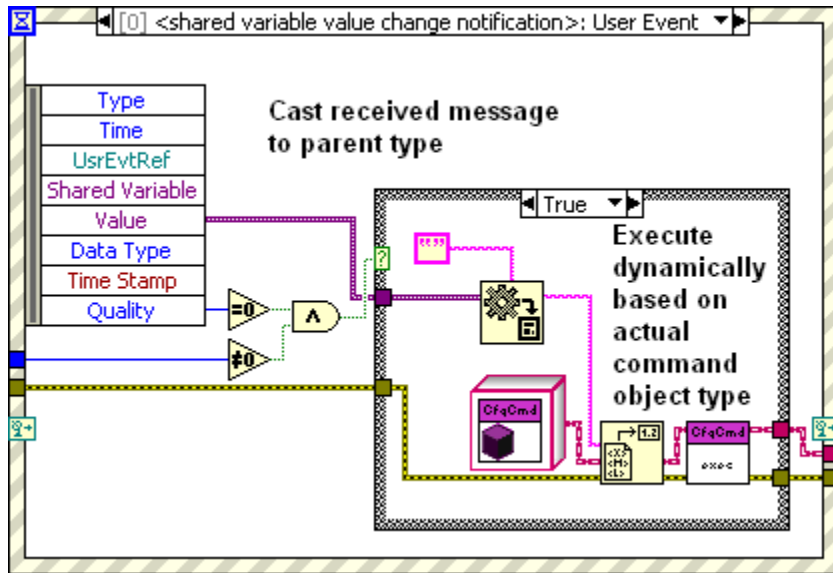
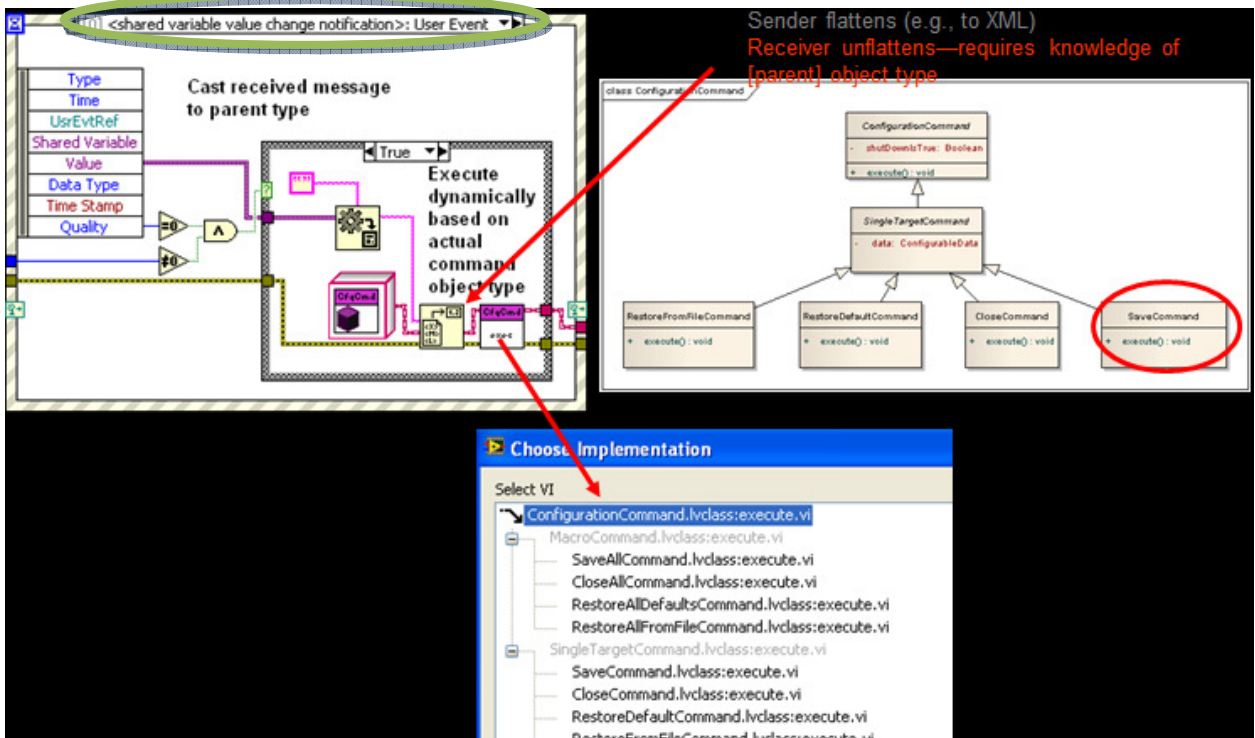


Simple enough, right?

2.2.3. Dynamic dispatching on the command type

Upon receipt of the message the receiver unflattens the received message to the type of the *top-level command*, that is, to ConfigurationCommand. (The resulting *actual type* on the wire will be of type SaveCommand, however!) (In this example the reception is a shared variable value change event.)

The receiver then invokes the *ConfigurationCommand.execute* method. Since we have defined *execute* as an override method on each of the command classes, dynamic dispatching on the command object type determines which method actually executes (*SaveCommand.execute*, in this example).



Note that the override methods may inherit common behavior from a parent or (as in this case) do completely their own thing.

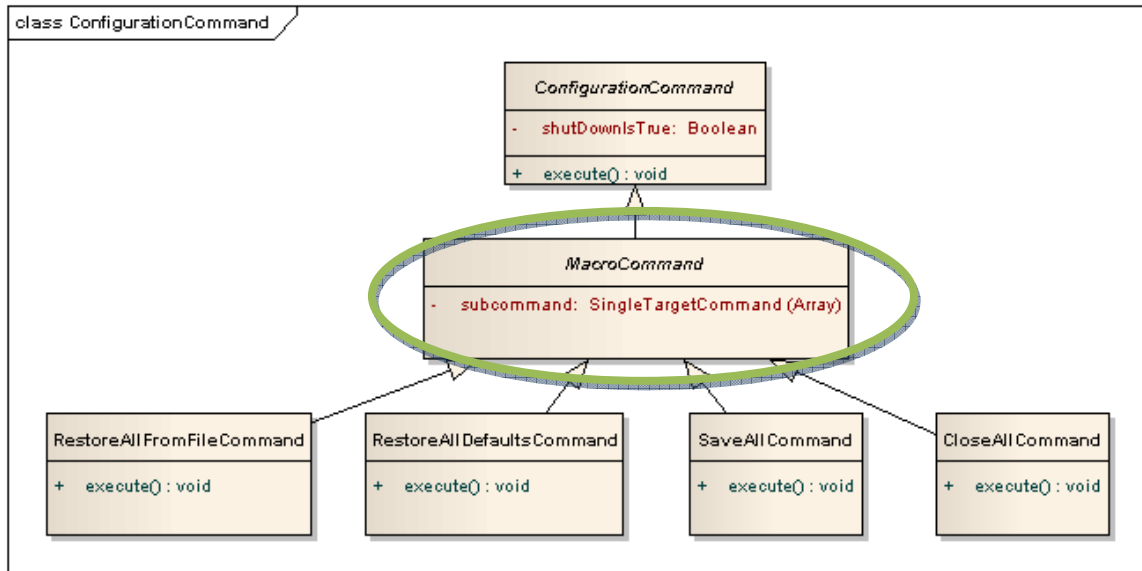
2.2.4. Add targets and parameters

Moreover, we can add data (e.g., targets or command parameters) to the commands very easily—and the *data type can be unique to each command class!*

For instance, let's say we have a controller that performs the same operation for a number of devices. Perhaps our controller can turn on DeviceA or DeviceB. We might indicate to turn on *DeviceA* by including a *DeviceA* object in an attribute in the *TurnOnCommand*.

In the example above we have included the `shutdownIsTrue` Boolean parameter in all commands, and the data attribute (of type `ConfigurableData`, which itself has a fairly complex definition) for all `SingleTargetCommands`.

The flexibility offered by the generalization (inheritance) relationship allows us to make this highly customizable. In a parallel part of the hierarchy `MacroCommand` has a subcommand parameter, where subcommand is an array of `SingleTargetCommand` command objects.

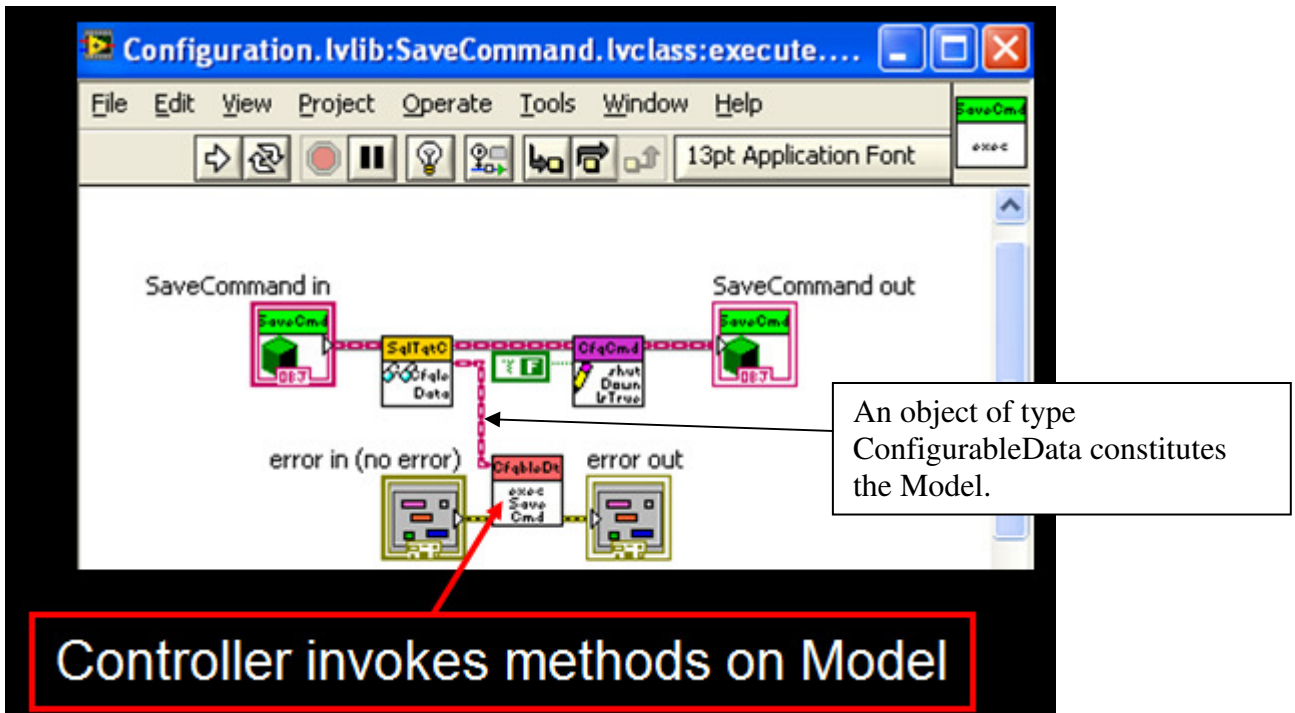


Note that in this case we are writing all types of commands with any type of data (which could include the device to which the command applies) to a *single shared variable* and that a *single controller* can handle all these messages appropriately.

The Command Pattern further allows the developer to implement undo functionality and so on, but I think this example is sufficient to show the essence of the pattern.

2.2.5. Delegation

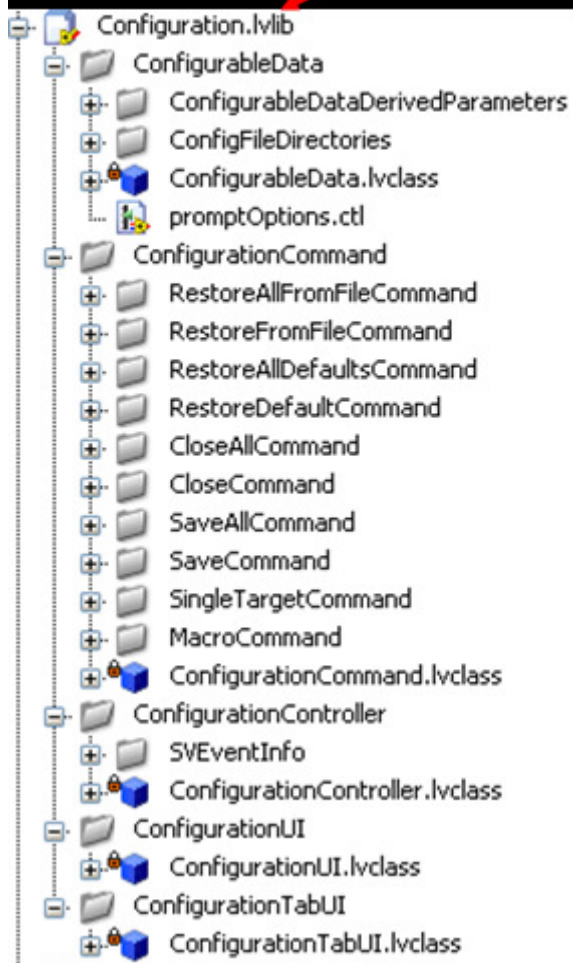
One last point on the Command Pattern is that the Controller determines what to do, but then delegates all the actual work to the Model (in this case `ConfigurableData`).



2.3. XML configuration files application

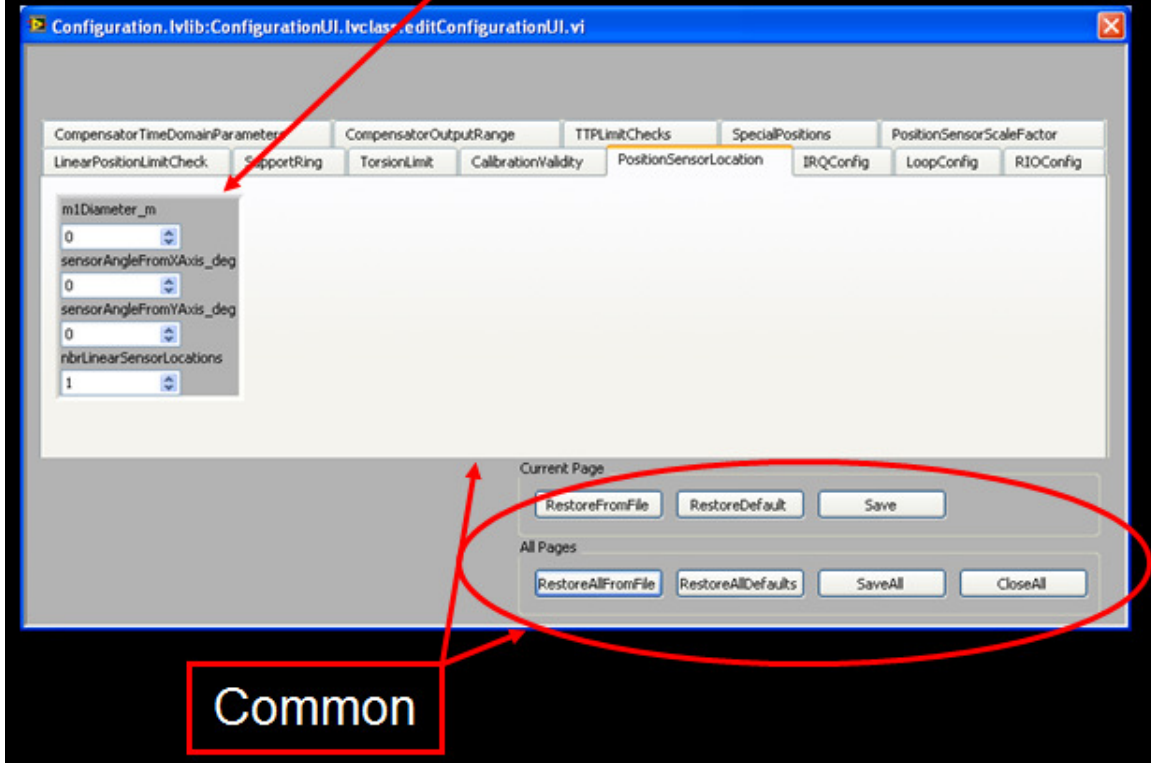
In the end we have a code library we can reuse in various projects.

Reusable library

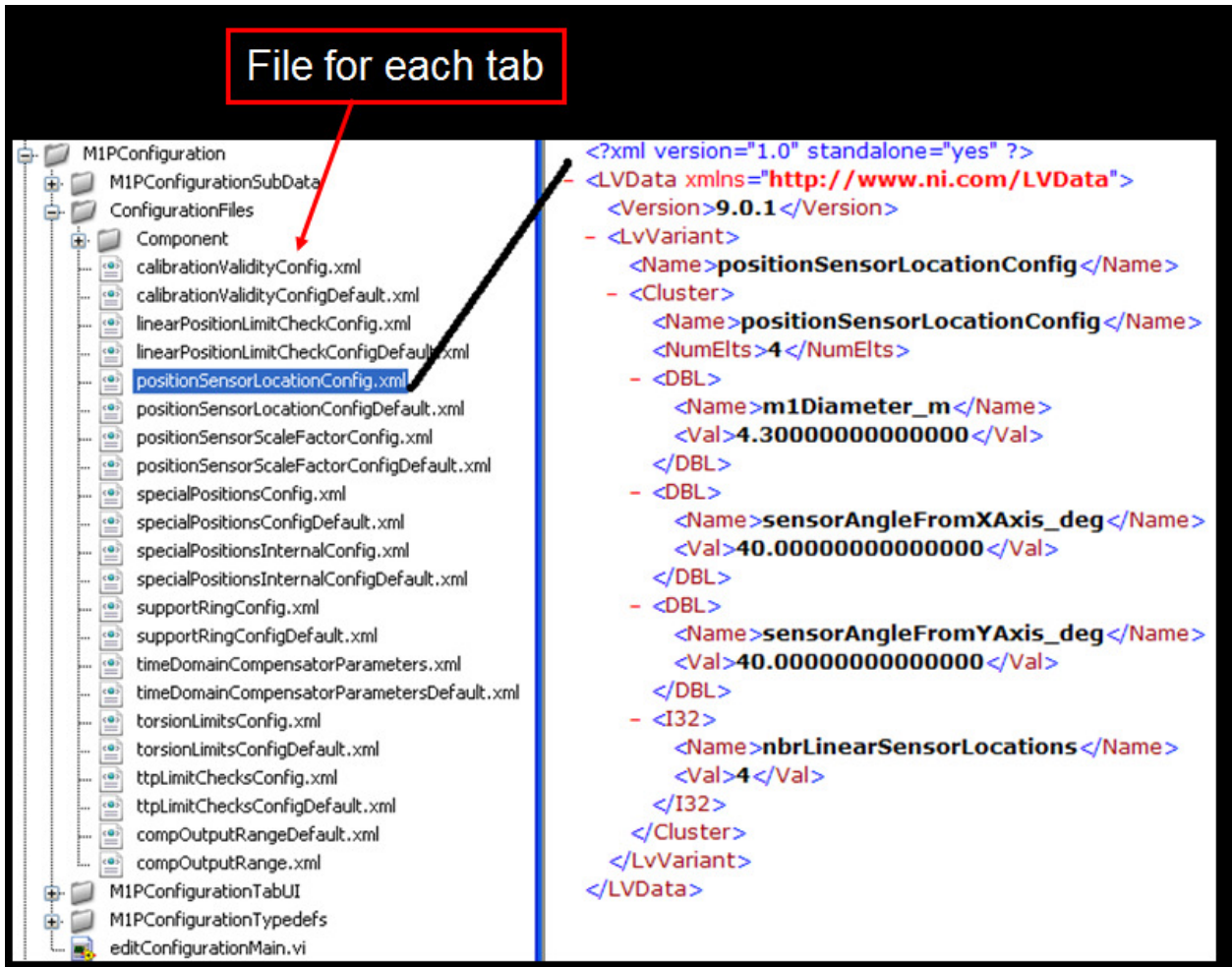


When we create a new component we define the parameter definitions and create a new shared variable. The rest of the configuration files application code is common.

Each component defines parameters



The application writes an XML file (currently using LabVIEW's native XML tools) for the parameters on each tab. (Note that we are using XML here just for reading and writing files, not for communication.)



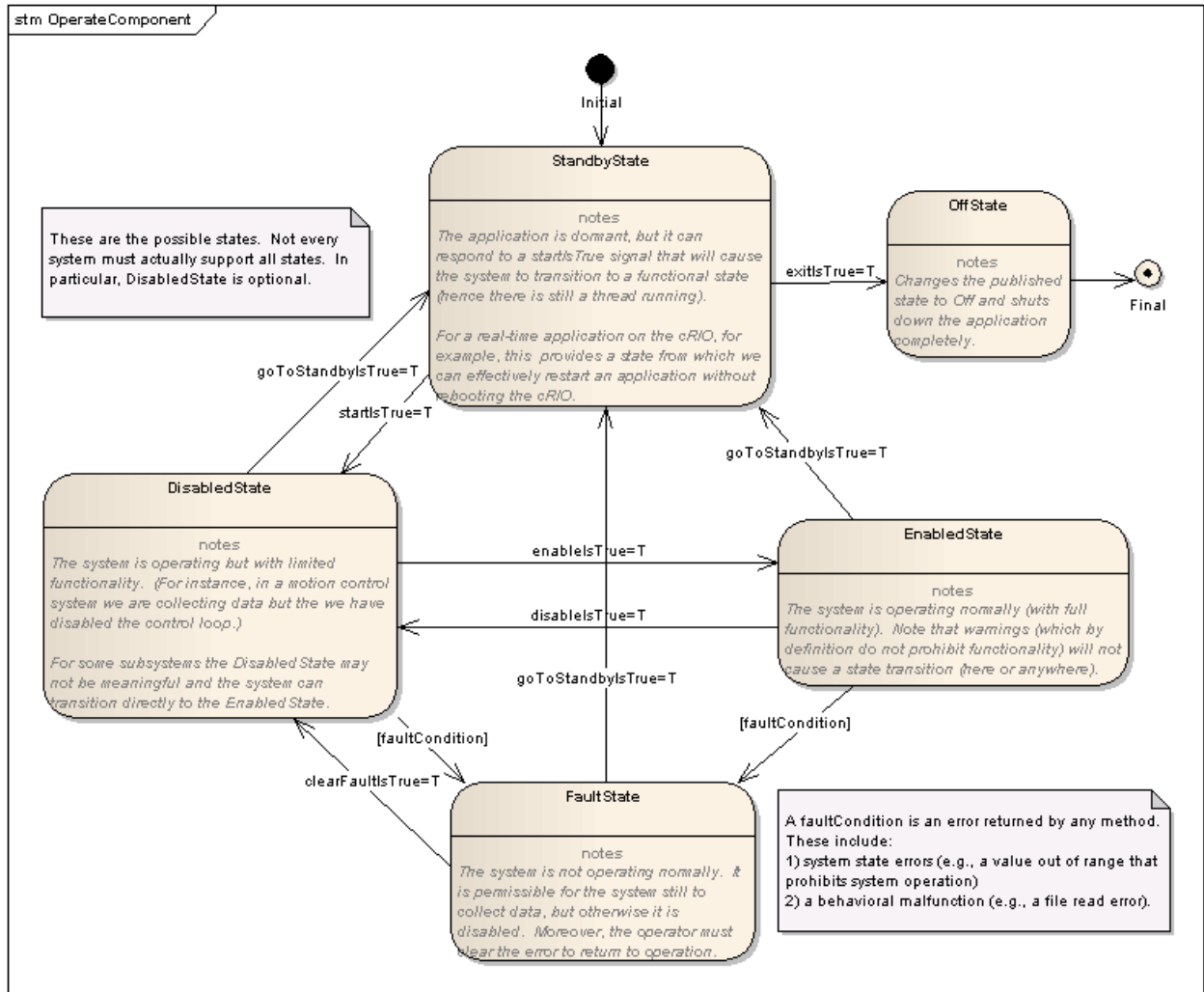
2.4. State Pattern Example

We'll next present an example of implementing a statemachine using the State Pattern.

2.4.1. Component statemachine

In this example we will consider a statemachine for the high-level status of each component. The statemachine diagram below shows the possible states and transitions.⁷ (In our system each component also has a detailed statemachine with states peculiar to it.)

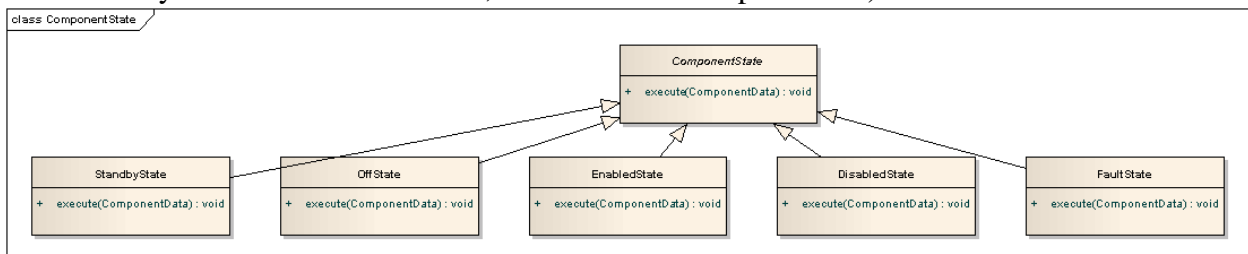
⁷ Maybe we should have included a StartupState between StandbyState and DisabledState. I'm on the fence about that.



2.4.2. Each state is a class

Once we have defined the states, we implement a class for each state. Each class has State in the name. Each state implements an execute method with a single parameter. That parameter (of type ComponentData in the example) is the Model.

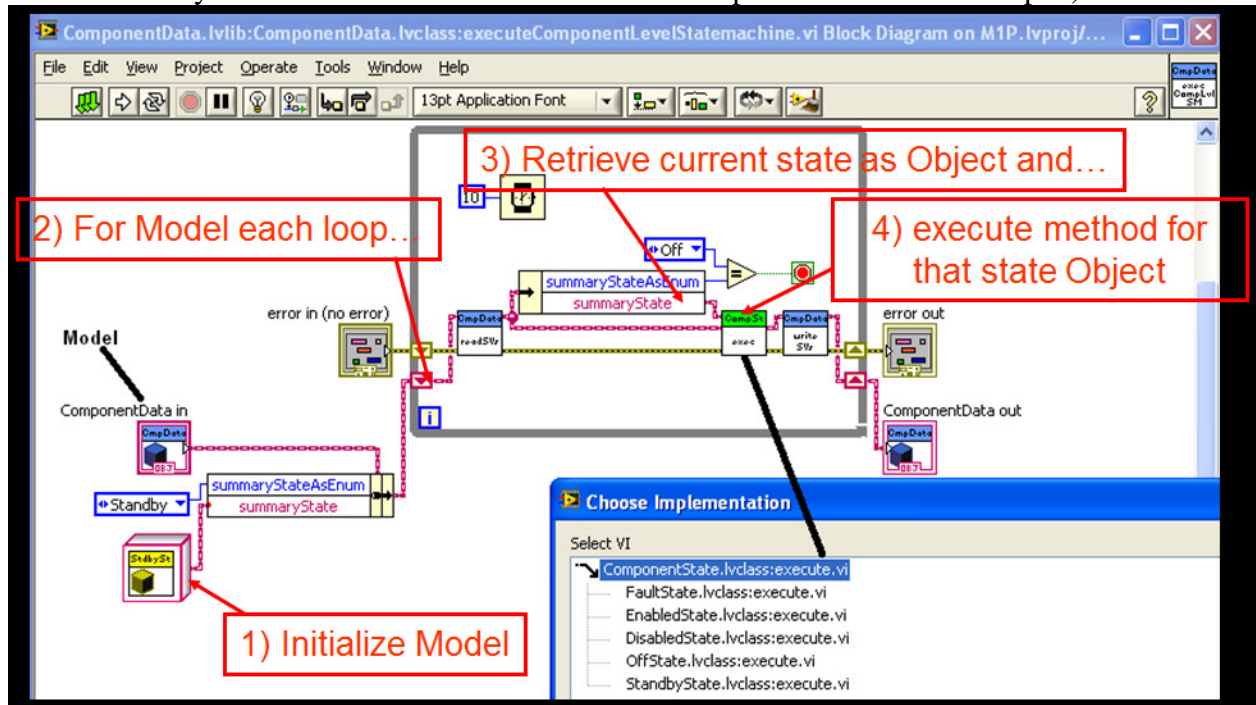
Note that child classes extend the execute method for reuse or override. This means we can define common functionality only once on the parent, but we can customize behavior in any child without affecting any of its peers or ancestors in any way. (In this example we have only one level of inheritance; more we use multiple levels.)



2.4.3. Dynamic dispatch on the state type

To implement the Controller statemachine we do the following:

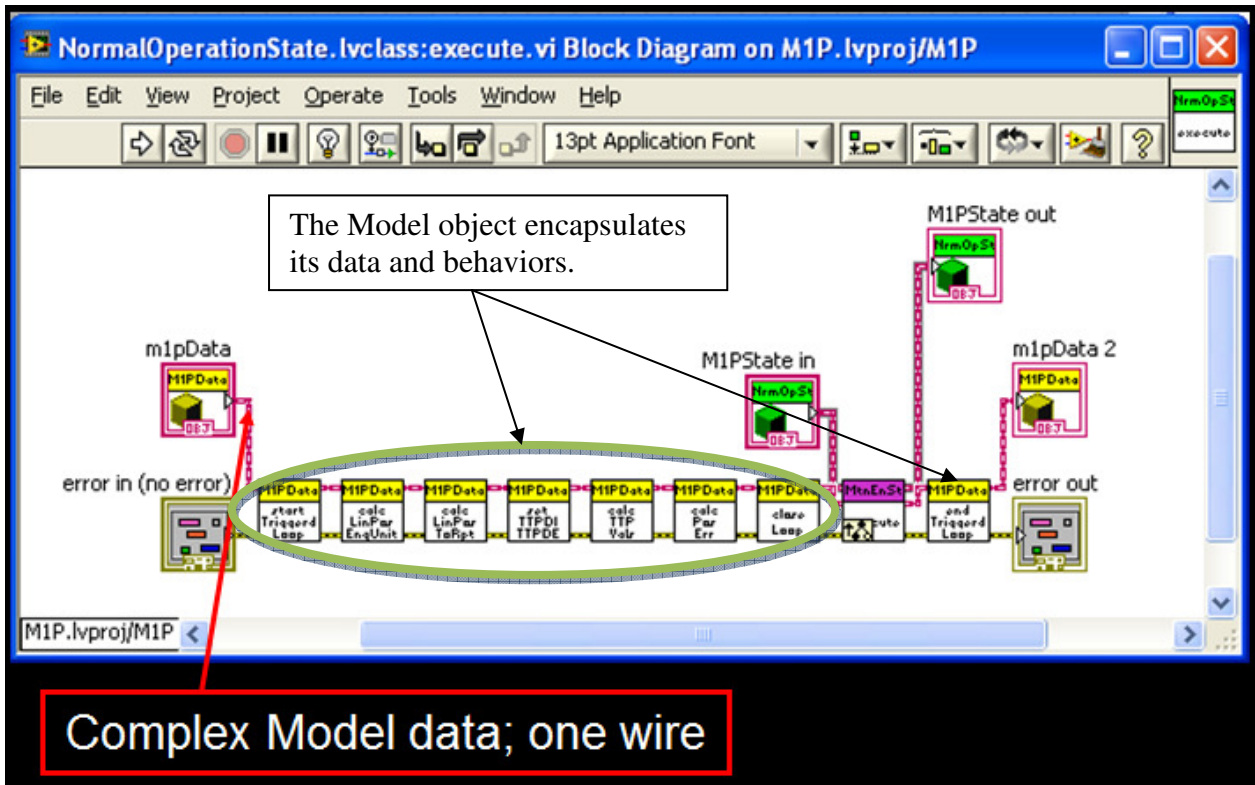
1. Initialize the Model with the initial state. In the example the initial state is StandbyState, so we write the corresponding object to the Model.
2. On each loop iteration, the Controller uses the current value of the Model (hence the current state of the system).
3. The Controller reads from the Model the current state (as an object).
4. The controller invokes the execute method on the top-level state. The method that executes is that for the actual state object type on the wire. (So StandbyState:execute will execute in the first loop iteration in our example.)



Note that inside the execute method we set the value of the next state by updating the state object in the Model as necessary.

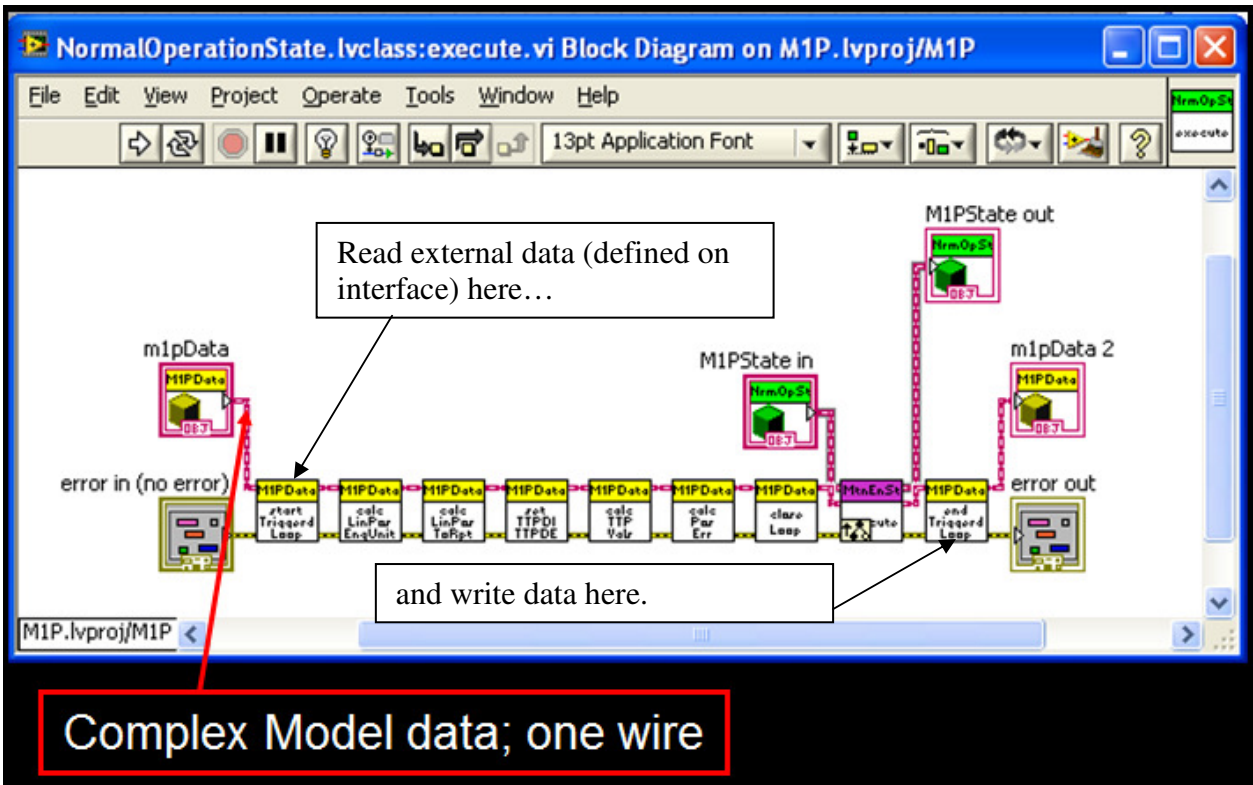
2.4.4. Delegate

The controller passes the current value of Model (the current state of the system) to the execute method. The execute method decides what to do but it delegates all the actual work to the Model, which has the information required to do the work and can change and track its own state as necessary.



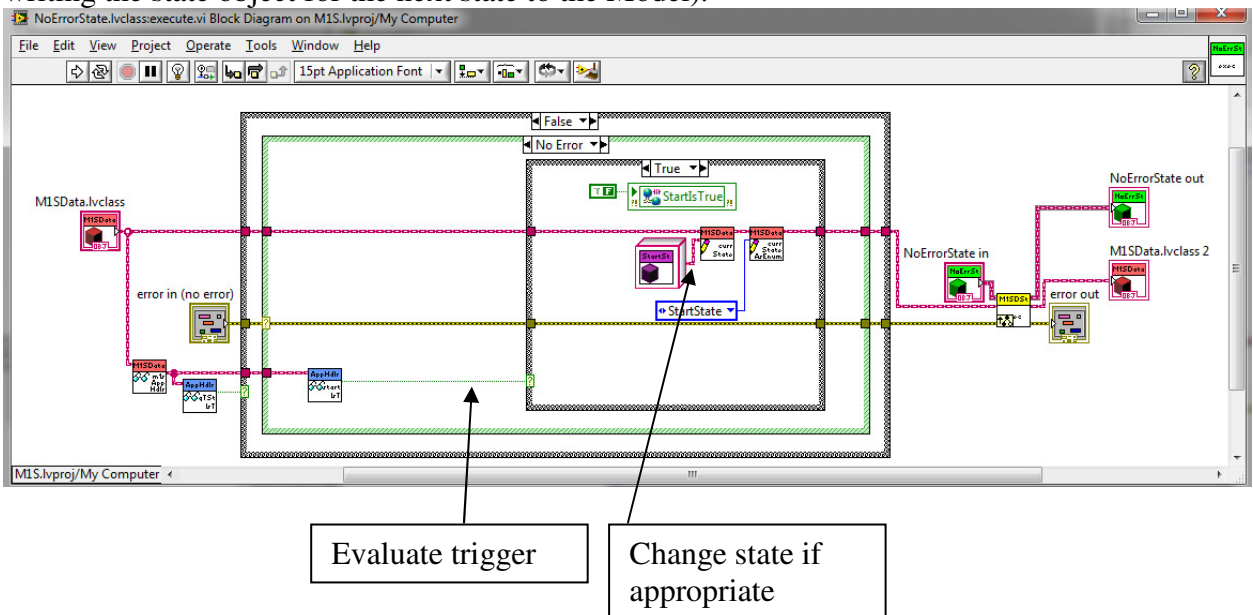
2.4.5. Exchange data via interfaces

This code executes in one loop iteration. (The execute method is called inside a loop. In this example the code is running on the RT processor of a compactRIO and the loop starts upon the receipt of an interrupt from the FPGA.) The Model (state) enters *on a wire* on the left. (It comes from a shift register on the loop outside.) We read the shared variable values (we have defined the shared variables to which this component subscribes in an interface definition) and data from the FPGA at the beginning of the loop (or wherever we want, really—the point is we explicitly read them once each loop), act on the system appropriately, then write values to another set of shared variables to which this component publishes data and to the FPGA. The Model state can change in the process, but we can look at the code and understand where this happens and probe the Model if we want to see the current state. Of course the shared variables themselves—which together form the external interface—connect to external systems without dataflow, but interactions between the shared variables and the model are always via wires.



2.4.6. Handle triggers

Within the controller we can evaluate any triggers, changing state if appropriate (by writing the state object for the next state to the Model).

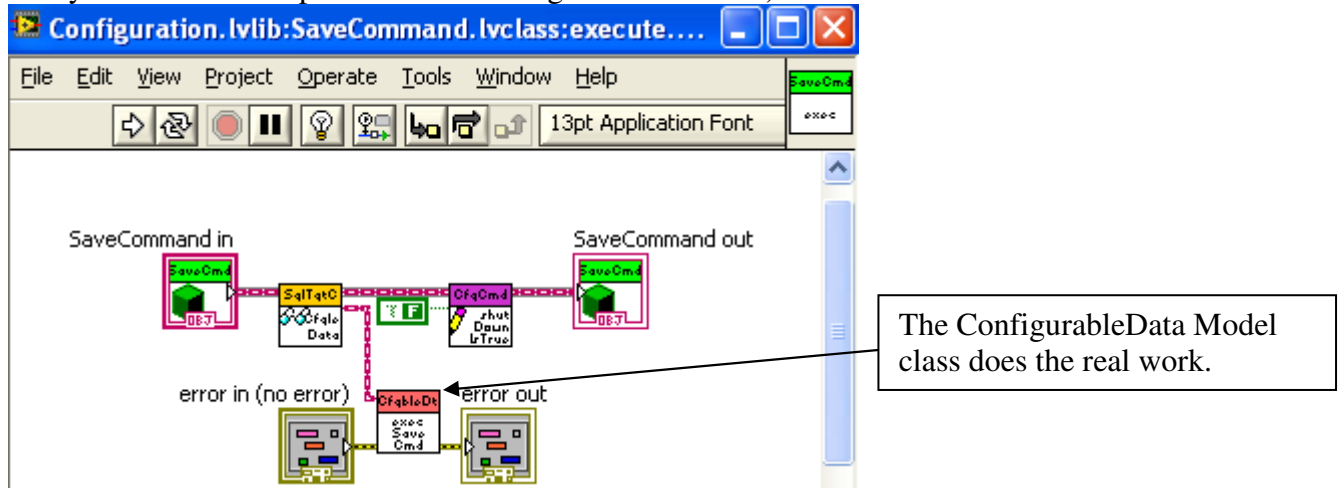


2.5. Model/View/Controller

Since we mentioned these terms they deserve a brief note. We implement the View (e.g., the user interface) as a separate application from the Controller. This allows us to create

an alternative view or even an external component that can implement the Controller interface.

The Controller invokes methods on the Model. (The Model data represents the state of the system and it has operations to do things with that data.)

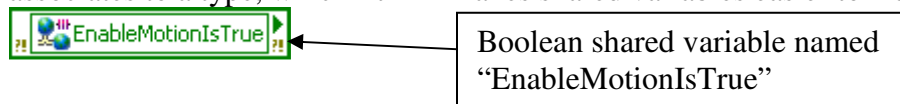


2.6. Appendix: Comparison with queue and functional global variable implementations

How does this approach compare to some other popular solutions? This section speculates on some possible comparisons. The intent is to stimulate discussion.

2.6.1. Queues

I actually think queues offer a reasonable messaging option, but are just not as flexible as shared variables. One reason I prefer shared variables is because the topic already associates to a type, which I think makes shared variables easier to manage.



Moreover, I think shared variables offer all features queues do except for the ability to reorder messages in the queue. In the applications we write this feature just isn't necessary since our Controllers always return in one loop time to handle the next message. (This is essential for our Controllers to operate effectively asynchronously. So if an axis move starts the axis controller state might be MovingState, but the controller keeps looping so it can still receive a stop command; it doesn't wait for the axis to finish the move before going to the next loop!)

Effectively our statemachine controllers implement the preemptive mechanism. For instance, if an axis is moving and the controller receives a stop command, the controller stops the axis; it doesn't wait for the move to finish.

Next, shared variables can work across a network (queues don't) and I think this is a major advantage.

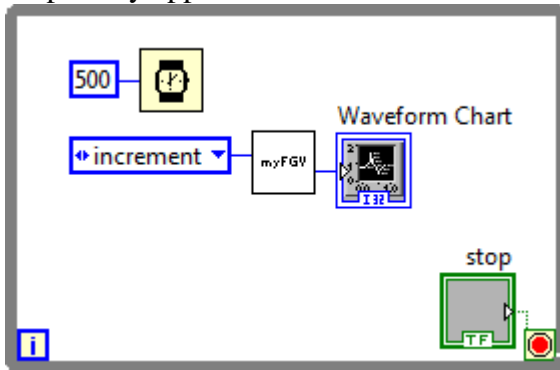
Finally, using a publish-subscribe protocol allows further decoupling of the components from one another and from the messaging system. The shared variable engine handles all the connections.

2.6.2. Functional global variables and “action engines”

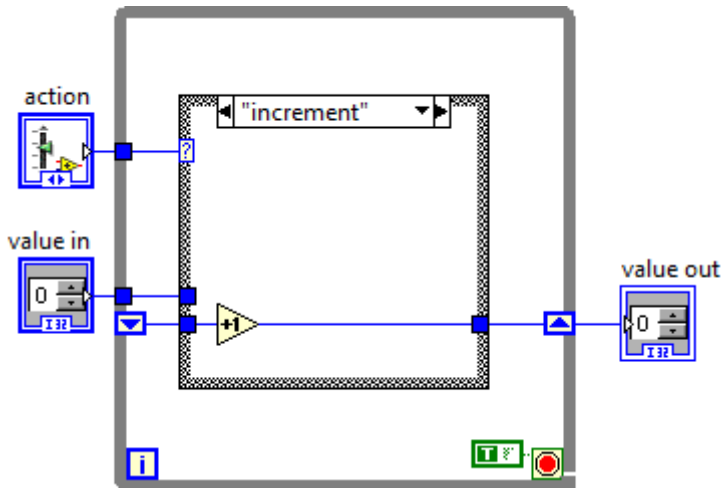
I first saw a functional global variables (FGV) in code I inherited about a decade ago. At first FGVs confused me. Then I figured them out and I was still confused when I tried to understand how they worked in a particular application. Writing this document helped to clarify why.

I already knew that I wasn't especially fond of the API for functional global variables (they look like regular VIs on the outside, although of course one can customize the icon; one often has to specify both an operation and the parameters via at least two inputs, which just takes a bit longer to comprehend; it is a lot of work to make the inputs polymorphic and clumsy to write a single value in a cluster as an alternative; it takes a somewhat sophisticated developer just to understand the point of using uninitialized shift registers with a single-iteration loop). No one of these API concerns, though, constitutes my biggest confusion point.

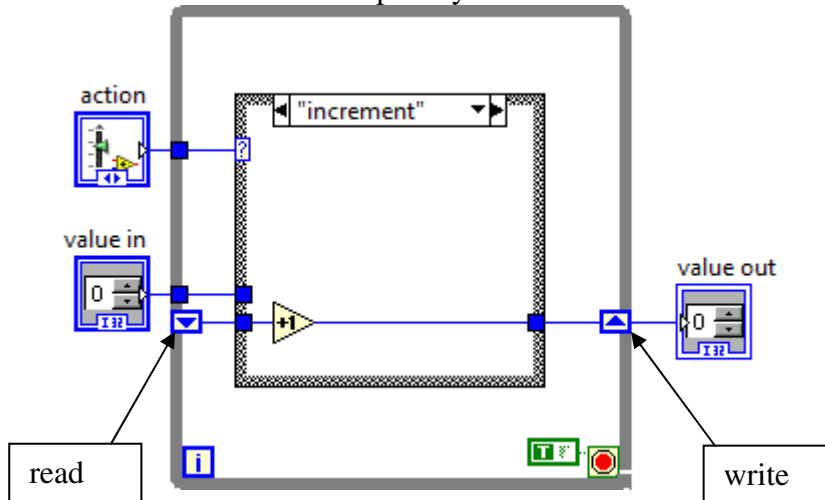
Let's illustrate what I think is the biggest issue by putting a functional global variable in a loop in myApplication.



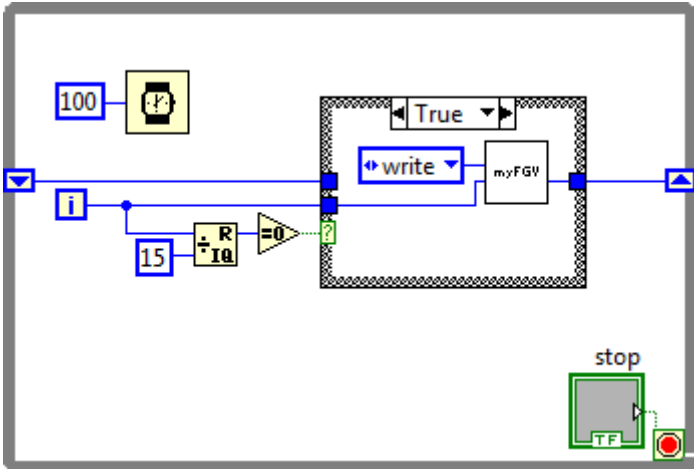
Suppose someone gave us this code and the code for myFGV. (The “actions” are write, read, increment, and decrement.)



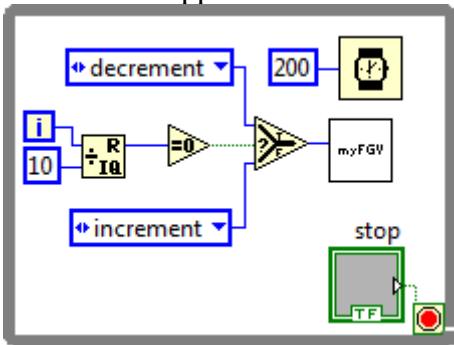
Then this person asked us what the expected behavior is. We might answer that the value on the chart on each iteration will be one integer value larger than on the last. This will be correct, in fact, if myApplication is the only application running. On the other hand, it is entirely possible some application somewhere else is changing the state of the system. The increment action implicitly includes read and write actions.



We can implement this “action engine” simultaneously in other applications, and each of these can change the state. As examples, I created otherApp and otherApp2, either of which can change the state. The implementation of otherApp is

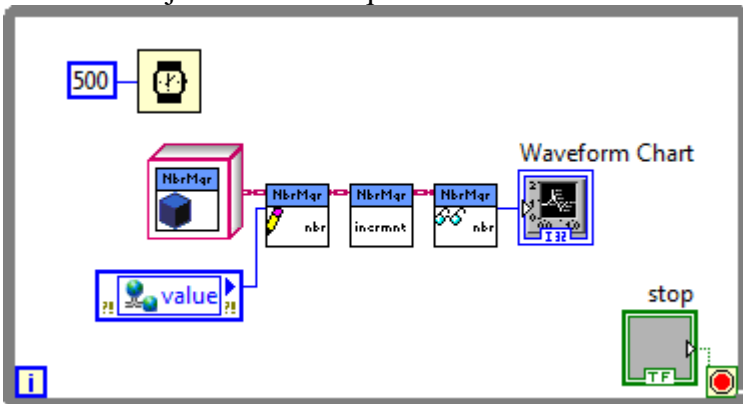


and of otherApp2 is



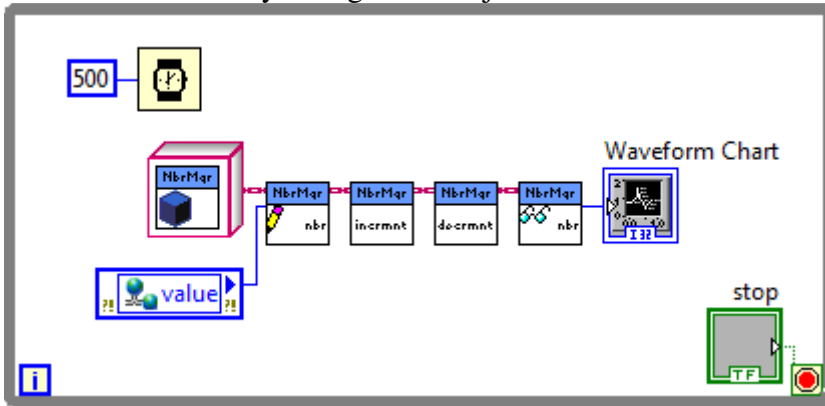
Well, in order to know what the output will be I have to know that otherApp and otherApp2 exist, whether or not they are running, and the details of what they do—not just the read and write actions but also the increment and decrement actions and any other arbitrarily complex actions we add, as well as when they do these actions.

This in itself isn't necessarily bad, actually. I might say that myApplication is just reading a value on an interface every 500 ms and incrementing that value. We could make the Object-Oriented equivalent.

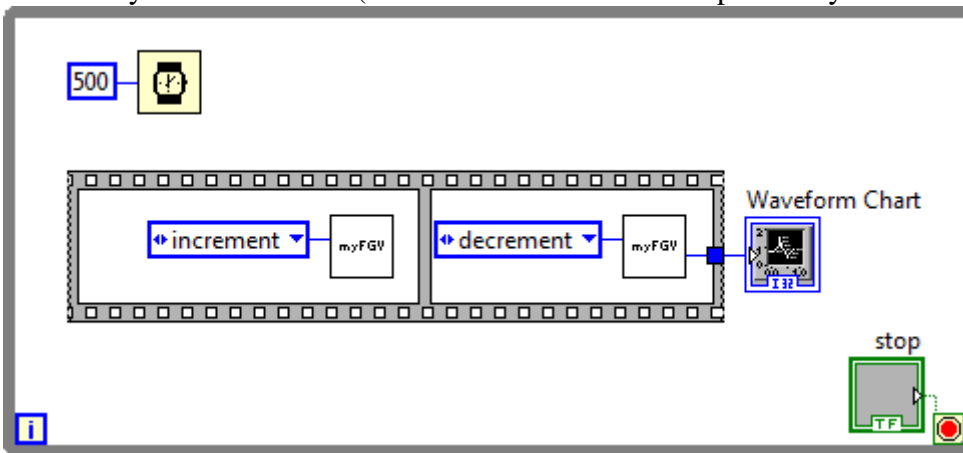


On the other hand, let's say we wanted to do two operations on the data.

We can do this easily enough with objects.

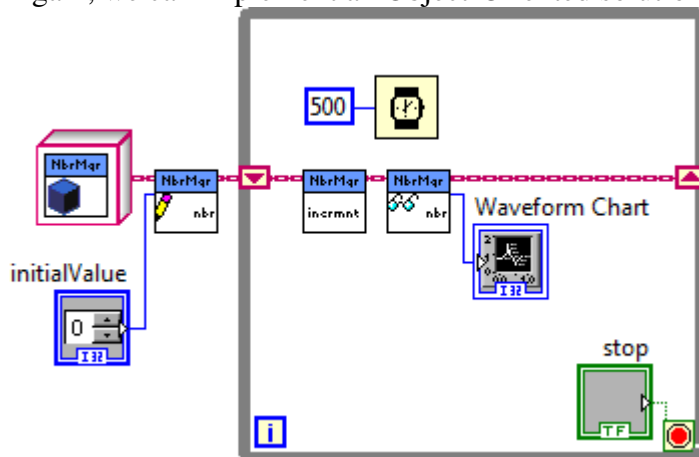


We can try this with FGVs (but in a moment we will explain why this is not suitable.)

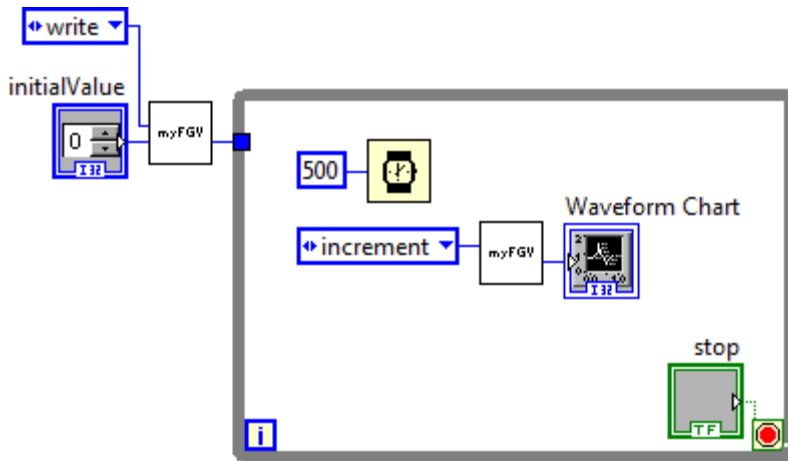


Or maybe we want to use this functionality without an external interface.

Again, we can implement an Object-Oriented solution.



We can again attempt a solution with FGVs.



These applications are simple to implement with by-value objects but the functional global variable approach is not appropriate for either of these needs.

Strictly speaking, it is possible to code something with a functional global variable (as shown) that can work in these instances but since functional global variables do not use dataflow

- 1) One has to use sequence structures, error wires, or some similar mechanism to ensure things happen in the proper order (inconvenient but certainly not a deal breaker). More importantly,
- 2) The implicit read and write functions *in every call to the FGV* mean that the state of the model can change at any time via a call to the FGV *anywhere in scope*.

The consequence is that just when we thought we had encapsulated the data and behavior we found we don't have encapsulation at all.

It doesn't take much before this gets out of hand and we don't know what the application does unless we have achieved a certain level of omniscience with respect to the implementation of all the applications. Now omniscience may be a good thing, but it is not something I possess or want another developer to expect of me, so I try not to expect it of anyone who might read my code.

I think that the problem with "action engine" FGVs is that we have mixed communication and operations. We can't meaningfully create an interface for myApplication or its kin and as a consequence we can't turn it into a proper component.

In practice, I think FGVs are OK (although still not the best option because of the limitations of the API) as long as we just use them as global variables (their original purpose), that is, for communication. Once we mix other state-changing actions in them we obfuscate the application code, precisely because we violate the principles of tight cohesion and loose coupling in a very bad way—we have to know what every other application in scope is capable of doing!

I think LabVIEW by-value objects offer a much better approach to *encapsulation*, and I think the LabVIEW object API makes these a lot easier to develop and maintain.

Moreover, LabVIEW objects support *inheritance*, which offers a great deal more power and flexibility. They also have the advantage of being a leading programming paradigm.

Since we have begun using LabVIEW objects with a messaging system here, we no longer use FGVs anywhere in our applications.