jettl

An Interface-Composition based
LabVIEW Actor Model

Nathan Davis

June 24, 2025

Contents

0.1 A Community Effort

This document is a community effort to create a LabVIEW Actor Model that is interface-composition based. The goal is to create a design pattern that is easy to use, understand, and extend. This document is a living document, and contributions are welcome.

Along with reading this document, the model will be best understood with examples that occur in projects. To show the power of the interface-composition based approach, in particular, examples that are difficult in DQMH, Actor Framework, and other frameworks will be of upmost importance. This document is not meant to be a complete guide to the jettl framework, but rather a starting point for understanding the design philosophy and implementation details.

0.2 Introduction

"The fundamental situation is, that we don't know much and some of it's wrong." -Carl Hewitt, creator of Actor Model

Strictly interface composition based asynchronous actor oriented design pattern for LabVIEW Applications. jettl also has the newer banners for vis. Easy adoption for the new age of LV developers. State pattern with decorators. It is interface composition, so stick with the same rule set for naming methods

0.3 Philosophy

0.3.1 Access Scope

Only public and private. Interfaces, classes, and methods have text in the icon/banner that are black for public and red for private.

0.3.2 Errors

Except the error that could potentially occur at the end of Queue Actor.vi and Event Actor.vi, no error goes unrecognized. Why are there are so many error case structures? All methods in Base jettl are assumed to run unconditionally when they are called. Also, no serialization of errors. Instead, it is serialization of the object wire since internally, the class on the iterface object in the private data has an error cluster. The merge errors method (which internally has the "Is Error.vi", dictating if Base jettl should enter the respective Error State.

0.4 Class Hierarchy

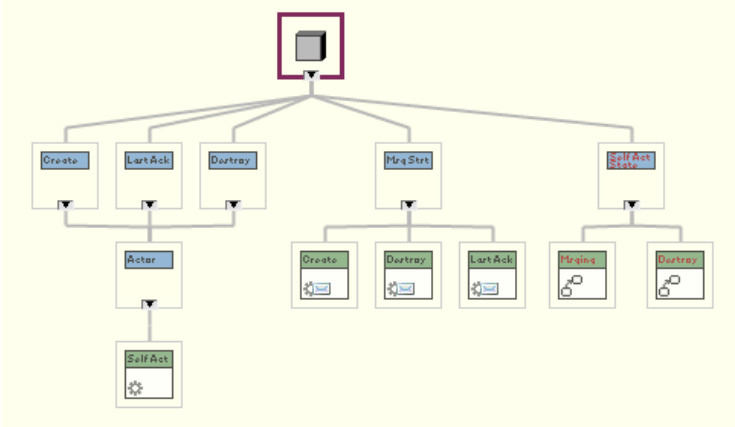

Figure 1: jettl Class Hierarchy.

Fig 1 shows the class hierarchy of jettl. In particular, there exist design patterns such as the Strategy Pattern, State Pattern, and Decorator Pattern.

0.5 Actors

0.5.1 Actor Benefits

jettl Queue Actor: Obtains and releases its own references. jettl Event Actor: Creates, Registers, Unregisters, and Destroys its own references.

0.6 LabVIEW Interfaces

The default implementation idea works so only as there is only one method implementation across all interfaces.

For example,
class implements interface 1 and interface 2

interface 1: method
interface 2: method
This cannot exist unless the class overrides the method. Otherwise, at runtime, LabVIEW does not know to execute interface 1: method or interface 2: method.

0.6.1 Descendants Must Override

Interface methods: check the 'descendants must override' checkbox and see which errors pop up. Attempt to fix them, and if you can't, then justify the reasoning.

For example, in the Self Actor State.lvclass, the DD methods are not required to be overridden. This is because the Self Actor State.lvclass has default functionality in the interface methods. Maybe this is bad practice, but it is a design choice showing only the methods that are required to be overridden with different functionality.

Another example, the DD methods in the Dev Actor.lvclass are not required to be overridden. This is because the methods here are all decorator methods that are not required to be overridden. Using the Actor.lvclass methods are sufficient for the Dev Actor.lvclass. These will be found in the palette.

0.7 Future Scope

Creating an actor is NOT connected to the actor that creates it. Rather, this actor exists on its own in the "liquid" message transport. The overall "application" has access to the actor's reference (unique message address)

0.7.1 Pub-Sub Messaging

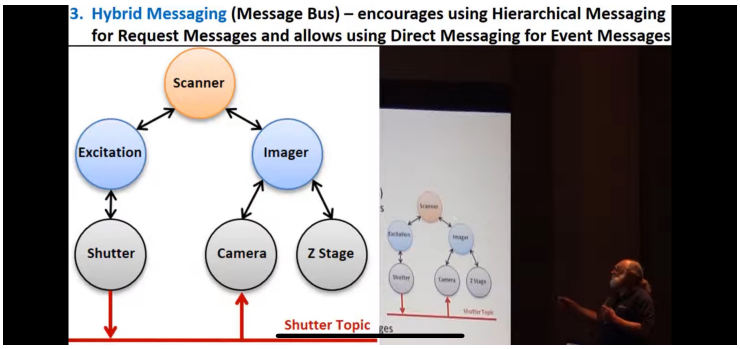

Figure 2: LabVIEW Actor Framework Message Transport.

As shown in Fig 2, it is as though the actors are just objects that "float" in this "liquid" messaging transport. So that way you can perform this pub-sub messaging. The objects are linked together by "who launches who".

Same as with the pub-sub pattern, everything is a publisher-subscriber. It's just that most relationships are just one way. Same with Git, the philosophy is that all branches are created equal

0.8 Unit Testing

This model is interface-composition based, so unit testing is built in. In effect, instead of wrapping the Queue Base with your developed class, you can wrap the Queue Base with a Queue Base Debug class, which is wrapped with your developed class. This occurs since these all inherit from the same Queue jettl interface.

This allows for arbitrary wrapping of classes, which is a powerful feature of the interface-composition based design pattern.

For the wrapping, it is as easy as have to wire in the object in the decorator method for debugging??

0.9 Debugging

0.9.1 Displaying Method Execution

What a log for which methods are executed, instead of the dialog popups

Two project conditional blocks:

1. Occurs in all methods

2. Occurs only in message specific methods

These conditionals occur in Self Actor.lvclass. If either conditional is true, debug panel that displays the names of actors (columns), along with timestamps (rows) of when methods (data) are executed. Think discrete time water fall display.

0.10 Git

This might be of interest for submodules in git for LabVIEW. Find here: Git Submodules: An Alternative Approach to Code Reuse - Greg Payne - GDevCon2.

In the repos, use the tag to have different "stable" versions of the repo such as v0.1.1 or v3.8.3 This allows others to easily look at the different versions of the repo without much thought. This could also help with submodules that are referenced in other repos. Check this video near the end for reference: Git and GitHub Tutorial for Beginners

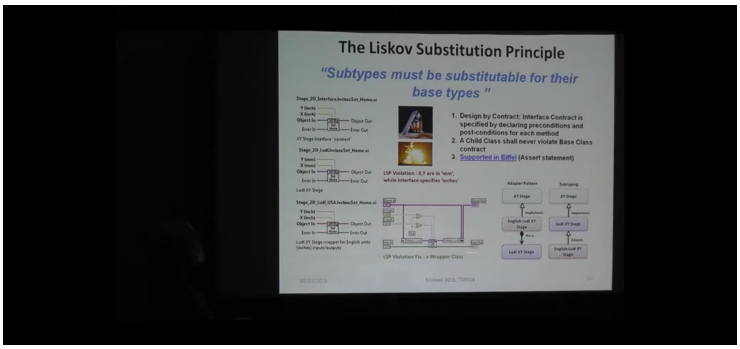

Figure 3: This image shows the naming convention for LabVIEW classes and methods.

0.11 Naming Conventions

Noting that Fig 3 shows the naming convention for LabVIEW classes and methods.

- Library-Name.lvlib
- Interface-Name.lvclass
- Class-Name.lvclass (control is capital by default!)
- method-Name.vi
- control-Name.ctl (this control, which is not tied to a class is lowercase!)

Note: Avoid underscores and spaces.

0.12 Examples

0.12.1 Hello World

Hello World Example: Don't need to send yourself Teardown for Hello World.. just have Teardown method itself at the end of the Hello World Msg.

0.12.2 Nested Actors

2018 NIWeek Allen C Smith Efficient Actor Framework Development 25:58 Teardown vs Orderly Teardown: Allen wanted a "Stop Nested Actors Msg", SLM against. There include two methods: Event Nested Teardown.vi and Queue Nested Teardown.vi. These two methods are used to teardown the respective nested actors.

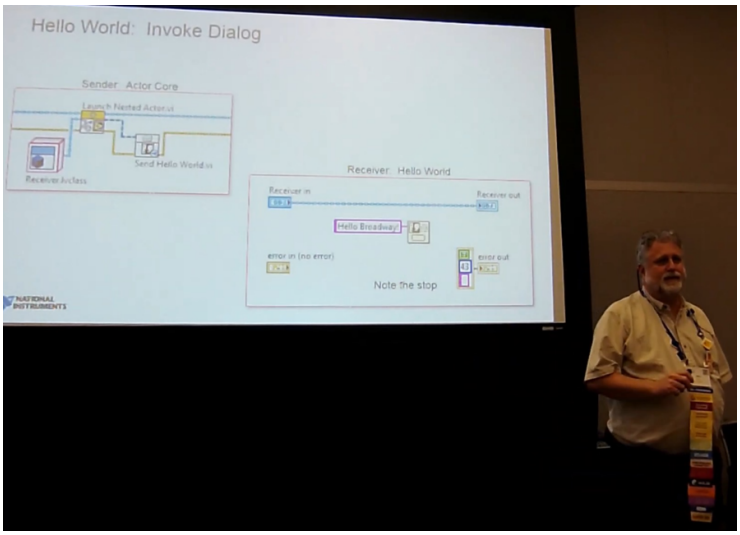

Figure 4: This image shows the hello world example.

0.12.3 Sending Event Message

Button press sends message to Self (queue actor). Queue actor internally sends a string indicator Event which occurs in event loop.

0.13 Design Patterns

0.13.1 State Pattern

Context:

Method.vi (just a wrapper for Method.vi "State").[1]

State.lvclass (interface):

Concrete State.lvclass

Method.vi "State"

State Pattern Privacy

Context classes should be private, since only interface objects should be composed into a class. Dependency Inversion Principle. Since the context class is private to the library it is in (and the State Interface with its concrete state classes), public static dispatch methods can be used in the context class AND concrete state classes without worrying that they'll be used outside the library since the context class is private.

[1] Best Practice: use this Method.vi in other methods, rather than the Method.vi "State" itself

0.13.2 Memento Pattern

When saving the state of an actor, the Memento Pattern is used. Instead of sending the entire actor state in Actor Last Ack Msg, there should be another dedicated message that sends the actors state to the calling actor. This has not been done, but it is a good idea to implement this in the future.

Memento, actors shouldn't know about each other, so if the state of a nested is to be saved, then there is a separate message for this since last ack shouldn't know about this. Rather, this specialty message couples with last ack if the developer wants this functionality.

0.13.3 Decorator Pattern

0.13.4 Strategy Pattern

0.13.5 Observer Pattern

Actors references cannot be passed between actors or helper loops. They are intentionally abstracted away from the developer. Now, because helper loops are async and own their own references. Their references can be passed around the actor tree. This provides a modular way of sharing async references, following the observer pattern.

0.14 Event Actors

This is to declutter the actor by offloading certain tasks such as panel display or subprocesses that do not require an additional actor to be created. Since the Event Actors are not dependent on the Queue Actor, we can unit test an Event Actor.

0.14.1 Event Actor Best Practices

References should not change after they are created.

0.14.2 Event Actor Benefits

Event are designed to be reusable and modular. Event Actors are not tied to the Queue Actor that creates them. Furthermore, Queue Actors should not depend on Event Actors.

These can be orphan processes i.e. they are not tied to the lifetime of the actor that creates them.

0.14.3 Window and Subpanel Event Actors

Split the events as controls and indicators as shown in Fig 5.

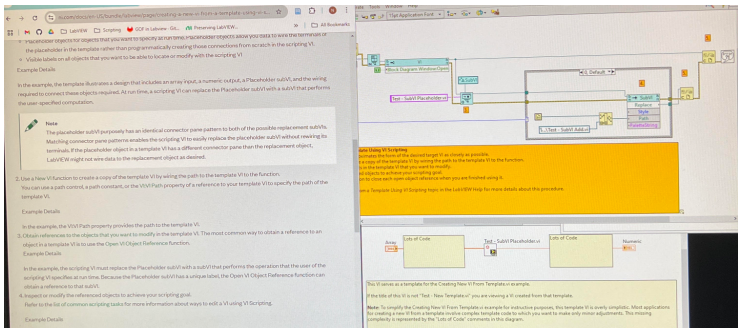

Figure 5: This image shows the UI events in LabVIEW.

0.14.4 Synchronous Event Actor

Since Queue Actors should always be ready to dequeue messages, it should not wait on a response. Rather, the Queue Actor should create the Synchronous Event Actor. When the Event Actor is waiting for response, that's okay since when the event actor is waiting, it is also ready to teardown whenever since it is always waiting for an event.

0.15 Messaging

All messages come from an interface and follow ISP where one message belongs to one interface. If there is a naming issue, that is a good thing. It means in the dependencies, you should be packaging modules so you don't run into naming issues.

All messages are interface messages. Note, they do not need to be implemented.

Actor messages error on generation of message. Error when creating a message and wiring in the interface object as an input, the message scripting doesn't know how to differentiate the class input and the parameter input.

0.15.1 Private Messages

Private messages that aren't exposed to external callers: Library access scope. To conform to this, the Last Ack messages should be private messages to the Queue Base class. This is because the Last Ack messages are not intended to be used by external callers. Further, this should also be done for the Update Queue Nested message. This should not be used by external callers, but rather only by the Event Base class.

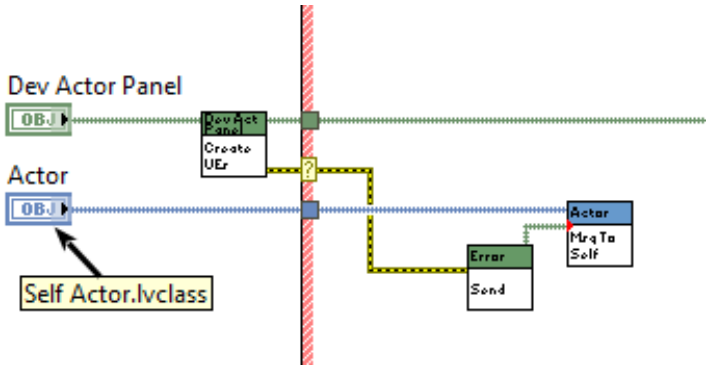

Figure 6: This is the Error message in jettl. This is a custom error message that occurs when an error occurs asynchronously.

0.15.2 Messages Up and Down the Tree

When messages are sent up the queue tree, followed by being sent down the tree, the developer could consider using event actors to message across the tree. This is a direct use case for using an Event Actor. Sharing the event actor reference to the necessary Queue Actors or Event Actors for direct communication. This is the power of Event Actors. Gives rise to the Observer (Pub-Sub) Pattern, sending NOT Msgs, but Events across the tree.

0.15.3 Messages with Type Definitions

Type definitions as inputs SHOULD be in the library, not the class they're implemented in. This is fundamentally an exercise in dependency inversion.

Cluster message with type def inside of the library. Good practice to put the type def here and NOT in the class that implements the message. This takes care of otherwise awful circular dependencies.

0.16 Tools

0.16.1 Scripting

Tips: Go through and replace all the Opens and Traverse with the hidden gem. User groups for the scripting (quick drop, right click, etc.).

Right Click Message Creation

0.17 TODO

In the DD methods, because they are not required to be overridden, have functionality within them that calls the Self Actor method (some kind of checking

mechanism?).

Actor interface methods (all implemented with default functionality) to have the *new* (Actor Interface) Read Actor DD method that reads from the Dev Actor the Self Actor class and performs that interface function, then bundles back in with the Setup method.

Do this for all for the default behavior.

Note this is breaking the contract for interface DD methods not needing to be overridden.

Instead of the Setup method, create a new Actor Write method which ONLY Writes to the Actor. This can also be put inside the Setup method as a first step, the setup code follows after

State Enter Core and State Exit Core are NOT check marked. That way the developer does not need to override, just to have no functionality anyway. Read State and Write State do because they'll have functionality.

0.18 Errors

In frameworks, errors are fundamental to the program's operation. They are not just incidental issues but rather integral to the design and flow of the application. jettl has an error object in the private data of the jettl object. At the end of the method, the error object is unbundled and checked for errors before teardown.

Error philosophy: Errors occur ONLY from unexpected events. For example, the error case in Msg.vi: in the case structure, a custom error saying that a message could not be executed occurs when 'message name' occurs for 'actor name'. This is unexpected behavior since this SHOULD be known at edit time. This is a legit error.

'That was handled, or I wouldn't have been called' - SLM (The Errors of our Ways — Stephen Loftus-Mercer GDevCon N.A. 2021: 52:08)

Wouldn't this make the API more beautiful and easy to understand? Having just the object wire come out of the method, and ONLY the object wire coming out of the method? I suppose, adopting the OO paradigm. Instead of having the error cluster inside the objects class data. Instead, it's a dedicated "error cluster" shared for every single object in use. Encourages data flow since unbundling of errors will always occur. It's a step in the right direction having no "error input" for methods. Now it's time to get rid of the error out. It's almost like branching an objects wire, in a way. There should only be one thing coming out of a method.

0.19 Example

0.19.1 Dev Actor

Merge Error and Override Error are decorators used. Otherwise, the other DD methods are trivial and only used once.

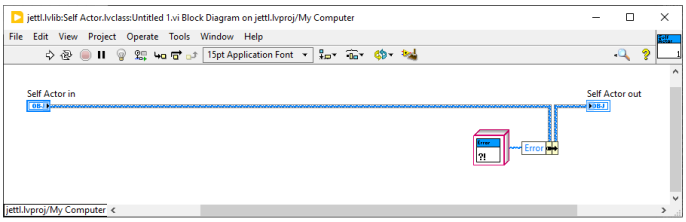

Figure 7: Method template. This has no error terminals, and the error cluster is handled internally. This is a different approach to error handling in LabVIEW methods.

0.20 Creation

Queue Actor

- Has One Caller Queue Actor
- Is a Nested Queue Actor of Queue Actor
- Is a Self Queue Actor

Event Actor

- Has One Caller Queue Actor
- Is a Nested Event Actor of Queue Actor
- Is a Self Event Actor

Framework constraint: A Queue Actor can create both Queue Actors and Event Actors whereas an Event Actor cannot create either. The Event Actor can queue itself either Queue Create or Event Create messages, but this goes to the Queue Actor to handle these. The handling does not occur in the Event Actor since the Event Actor cannot create.

A future question: Should Launch.vi exist outside the Queue Base class? If it does, then Queue Base can be marked as a private class, and the Launch.vi can be public.

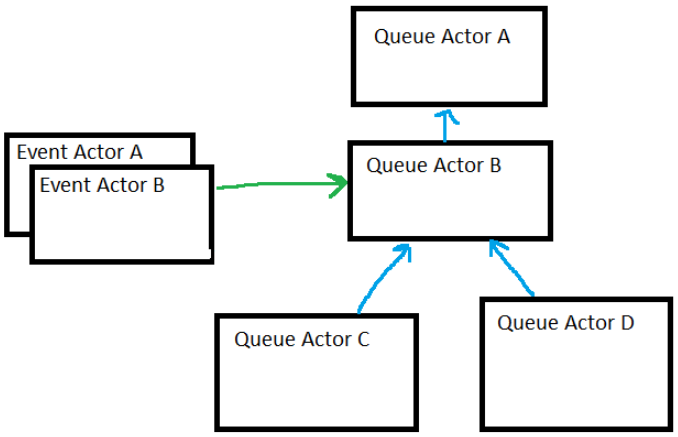

Figure 8: This image shows the creation of Base Actors.

Only Queue Actors can create Queue Actors and Event Actors. Event actors cannot create either. Further, because of these rules, only queue actors can receive the Last Ack from Nesteds. This is respectively the Queue Last Ack and Event Last Ack. Event Actors cannot receive either Last Ack. These Last Ack messages are not used by the developer. These occur in the framework itself, though functionality can be wrapped in the developed queue actor.

0.21 PPL Support

PPLs is not currently supported in jettl. This otherwise should be a simple task to implement. How to change a class to use the PPL version of jettl is to change the implemented interface to the PPL one and compose in the PPL Interface.

For more information on PPLs, Darren Nattinger and Derrick Bommarito have excellent content on this:

- Debugging Symptoms - Packed Project Library PPL Dependencies - Searching for Dependencies Dialog When Running Executable
- PPL Namespaced Dependencies - Strategy/Design Discussion - Development Issues
- LUDICROUS ways to Fix Broken LabVIEW Code with Darren Nattinger — GDevConNA 2022

Further points include:

- Ensure xnodes are not used in jettl.
- Get rid of all malleable vis (.vim extension)

0.22 Opinionated Design Choices

0.22.1 Color Scheme

"Look down at the green grass, look up to the blue sky, and look further to the purple galaxy."

- Purple Library: RGB (166,153,182)
- Blue Interface: RGB (104,136,190)
- Green Class: RGB (110,149,108)