LabVOOP Design Patterns *(version 1.0)*

Stephen Mercer, LabVIEW R&D, National Instruments

Table of Contents

Introduction

When talking about computer programming, a *design pattern* is a standard correct way to organize your code. When trying to achieve some particular result, you look to the standard design patterns first to see if a solution already exists. This sounds a lot like an algorithm. An algorithm is a specific sequence of steps to take to calculate some result from a set of data. Generally algorithms can be written once in any given programming language and then reused over and over again. Design patterns are rewritten over and over again. For example, in house cleaning, consider this algorithm for vacuuming a carpet: "Start in one corner, walk horizontally across the floor, then move to the side and continue making adjacent parallel stripes until the whole floor is vacuumed." Compare this with the design pattern for cleaning a room: "Start cleaning at the ceiling and generally work your way down so that dirt and dust from above settle on the still unclean areas below." Design patterns are less specific than algorithms. You use the patterns as a starting point when writing the specific algorithm.

Every language develops its own design patterns. LabVIEW has patterns such as "Consumer/Producer Loops" or "Queued State Machine." These are not particular VIs. Many VIs are implementations of queued state machines. When a LabVIEW programmer describes a problem, another programmer may answer back, "Oh. You need a queued state machine to solve that problem." Starting in LabVIEW 7.0, LabVIEW has had VI Templates for many LabVIEW design patterns available through the **File>>New…** dialog. With the release of LabVIEW Object-Oriented Programming, we need to find the new patterns that arise when objects mix with dataflow.

The seminal text on design patterns appeared in 1995: Design Patterns by Gamma, Helm, Johnson and Vlissides. This text is colloquially known as "the Gang of Four book." This text focused on object-oriented design patterns, specifically those that could be implemented using C++. Many of those patterns are predicated on a by-reference model for objects. As such they apply very well to classes made with the GOOP Toolkit (or any of the several other GOOP implementations for LabVIEW). LabVIEW classes, added in LV8.2, use by-value syntax in order to be dataflow safe. This means that the known patterns of object-oriented programming must be adapted, and new patterns must be identified. This document describes the LabVOOP-specific patterns I have identified thus far.

This document is not on NI's DevZone as I do not wish to give an official imprimatur to these yet. The descriptions and methods of implementation may not be as optimal. I am posting these as a common LabVIEW user, not as a member of LabVIEW's R&D team.

– Stephen Mercer

Reference:

Design Patterns Gamma, Erich, et al. 1995.
Addison Wesley Longman, Inc.

Singleton Pattern

(adapted from Gang of Four's Singleton pattern)

Intent

Guarantee that a given class has only a single instance in memory, that no second instance can ever be created, and that all method calls refer to this single instance.

Motivation

When you create a class, sometimes it is advantageous to guarantee that a program always refers to the same global instance of the class. Perhaps the class represents a database. It would be unfortunate if some section of code accidentally instantiated its own database and thus failed to update the global database. Creating a global means all VIs can access the data. But it does not mean that all VIs *will* access the global. This pattern describes a framework of classes that guarantee that single instance of data.

Implementation

See
http://jabberwocky.outriangle.org/SingletonPattern.zip

(An example of this pattern shipped with LV8.2, but that example is flawed, a problem I did not realize before we released: <labview>\examples\lvoop\SingletonPattern\Design Pattern.lvproj)

In this implementation, we seek to guarantee a single instance of **Data.lvclass**. We achieve this by putting **Data.lvclass** into an owning library, **Singleton.lvlib**, and making the class private. Now no VI outside the class can ever drop the class as a control, constant or indicator, so we guarantee that all operations are limited to this library. Callers can use **Singleton.lvlib:Checkout.vi** to get the current value of the data, modify it using any of the operations defined by **Singleton.lvlib** and then set the new value with **Singleton.lvlib:Checkin.vi**. The public functions are the same functions that would normally be on the class itself. They are moved to the library so that the functionality is exposed without allowing the class itself to be dropped. **Checkin.vi** and **Checkout.vi** use a single-element queue to guarantee only one copy in memory at a time. While the element is checked out to be modified by some routine, no other operation can proceed, thus guaranteeing serial access to the data.

Editorial Comments

I do not like the implementation that shipped with LV8.2. In the time since LV8.2 released, I have found better ideas. This "post-release discovery" is one reason why more design patterns are not included with the shipping product. LabVOOP is too new, and many of the patterns identified during development may be found faulty when exposed to a wider audience (i.e. real customers). So I'm making this implementation available.

The only hole in the above solution that I can find is VI Server. VI Server could get a reference to the Front Panel controls of Singleton's member VIs and use the Value property to effectively manipulate a different instance of the class than the one in the queue. Would using Subroutine Priority prevent VI Server access? I think so, but that seems like a pretty odd way to fix these VIs. The fix may not be that necessary – I doubt that many people are going to attempt such a hack on the class. Anyone trying it is already aware that they're working against the design of the class and may break components that depend upon the singleton nature.

Factory Pattern

(adapted from Gang of Four's Factory pattern)

Intent

Provide a way to initialize the value on a parent wire with data from many different child classes based on some input value, such as a selector ring value, enum, or string input. This may include dynamically loading new child classes into memory. Ideally new child classes can be created and used without editing the framework.

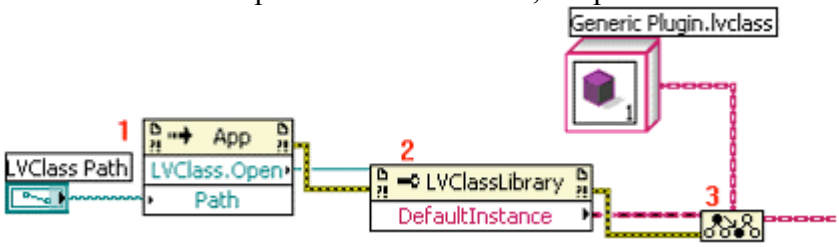
Motivation

Data comes from many sources – user interface, network connection, hardware – in a raw form. This data frequently comes in types, and the data is to be handled generically but each type has a specific behavior at the core of the handling. Such a problem cries out for the use of classes and dynamic dispatching. We want to write an algorithm around some common parent type and then let dynamic dispatching choose the correct subVI implementation for the parts of the algorithm that are type specific. The hard part is in initializing the right class from the input data.

Implementation

An example of this pattern was made available by Christina Rogers in her refactoring of the Getting Started Window in the Init From XML methods. Information about this example is available here: http://eyesonvis.blogspot.com/2006/08/object-oriented-getting-started-window.html

The general idea is this: Use your data to identify which class you are interested and get the name and/or path to that class' file. Then use the Application Properties to get access to that class. Suppose we name our parent class **Generic Plugin.lvclass**. All of our data will be wrapped by specific child types of **Generic Plugin**. After we have chosen the path to the child class, we proceed as follows:



1. Open reference to the .lvclass file on disk. This returns a LVClassLibrary refnum.
2. Get the default instance of the class.
3. Cast the LabVIEW Object that gets returned to your plug-in data type.

From there you would wire the chosen class instance to an Init method of the class which would take your data as an input. The advantage of this system is that the child class is only loaded into memory if someone actually wants to instantiate that child. If you have a great many possible types of data, this can save on memory and load time for your application. The name of this pattern comes from the fact that a single VI serves as a factory to produce all the myriad child types.

Editorial Comments

This implementation does not work in the run time engine because the LVClass refnum does not exist in the runtime engine. Without this, a much less elegant solution must be used. Instead of identifying the path to the class, use your data to pick an enum value, and wire that enum to a case structure. Each frame of the case structure should have a constant of the specific child class. This alternative implementation does not do dynamic loading and requires that all child classes be known and in memory when the VI loads. It further means editing the factory each time a new child class is created.

Hierarchy Composition Pattern

(adapted from Gang of Four's Composite pattern)

Intent

To represent a single object as a tree of smaller instances of that same object type. Useful for images (an image is made of a set of subimages), file systems (directory contains both files and other directories) and other systems where there is a common aspect to both the whole object and its parts.

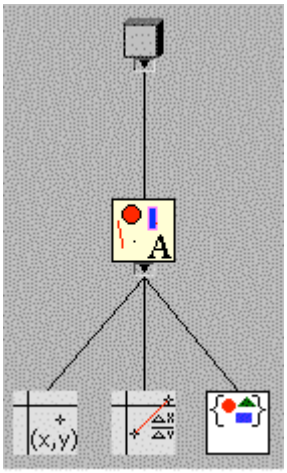
Motivation

"Composition" refers simply to using one class as member data of another class. One class is thus "composed" of another class. The Hierarchy Composition pattern is specifically for times when you are making a class that is going to be composed of smaller instances of itself.

Implementation

An example of this pattern shipped with LV8.2. See
<labview>\examples\lvoop\Graphics\Graphics.lvproj

This example includes a class hierarchy for **Graphic.lvclass**, as shown in the image.



Graphic Hierarchy

A graphic is simply something that can be drawn. **Graphic** defines a method **Draw.vi**, which can be overridden by children. There are three children of **Graphic**. The first two are straightforward implementations: **Point.lvclass** and **Line.lvclass**. Each of these two have coordinate data as their member data, and implement **Draw.vi** accordingly. The third class, however, is **Collection.lvclass**. **Collection** has as its private data an array of **Graphics**. That is, **Collection**, which is itself a **Graphic**, contains other **Graphics**. It's implementation of **Draw** is to loop over that array and call **Draw** for each contained element. The demo shows how a **Collection** graphic can be built up from individual points and lines.

You could continue this, adding a **Collection** inside a **Collection**, nesting them to create more complex **Graphics**.

The difficulty arises when trying to call **Draw** for a **Collection** that contains another **Collection**. This is a recursive call to **Collection.lvclass:Draw.vi**. LabVIEW does not support recursion and will abort the VI when the recursion is attempted at runtime. There are ways around this barrier. Exploration of the various pros and cons of the workarounds has been proceeding on the LAVA Forums:
http://forums.lavag.org/index.php?showtopic=3830

Delegation Pattern

Intent

To have two independent classes share common functionality without putting that functionality into a common parent class.

Motivation

Good object-oriented design requires each class to focus on its assigned task. A class shouldn't have member VIs unrelated to its task. If you have two classes that have some shared functionality, the usual solution is to create a parent class that both of them inherit from. Sometimes, however, you already have a class hierarchy created and a new feature comes along. The new functionality needs to be added to two existing classes that either do not have a common ancestor or do have a common ancestor but that ancestor is several generations up and not every descendent of that ancestor should have the new functionality. In some languages you might try multiple inheritance. But even in languages that support multiple inheritance, delegation is generally a better solution.

This pattern applies best when you have a dynamic VI inherited from some ancestor and two descendent classes want to override that dynamic VI with exactly the same implementation. Delegation helps you avoid writing the implementation twice, once for each of the two classes.

Implementation

An example of this pattern was made available by Christina Rogers in her refactoring of the Getting Started Window in creation of her Project Wizard class. Information about this example is available here: http://eyesonvis.blogspot.com/2006/08/object-oriented-getting-started-window.html

The general idea is this: Create a third class that has the new functionality. The new class is frequently something like "Function Helper" or "Function Assistant," where "Function" is whatever functionality is to be delegated. Give that class all the data fields necessary to carry out the new functionality. Then add that third class as a data member to the two classes that need to share the functionality.

The two descendent classes override the ancestor's dynamic VI. But they only put in a bit of code necessary to call the exact same method on their data member of the helper class. The actual work is entirely in that method of the helper class. Now there is only a single instance of the code, which makes bug fixing easier. The two descendent classes have delegated the work to a third class. Thus the name of this pattern.

Editorial Comments

This pattern is not listed explicitly in the Gang of Four text. The idea occurs as an aspect of several other patterns. I think it is useful enough to warrant specific attention.

Aggregation Pattern

Intent

To treat an array of objects as a single object.

Motivation

An array nicely collects data together into an indexable list. But there are many primitives that operate on an array: **Build Array**, **Remove From Array**, etc. There are times when you want an array to guarantee certain properties, such as:

- an array that guarantees that no duplicates are ever inserted
- an array that is always sorted
- an array that doesn't allow removing elements

This pattern creates a class that lets you treat an array of another class as a single object.

Implementation

No examples of this pattern are publicly available yet. Here is a summary of the pattern:

Suppose you have **Data.lvclass**. Now you create **ArrayOfData.lvclass**. In **ArrayOfData**'s private data cluster, you add an array of **Data**. The only operations that can be performed on this array are those that you add to **ArrayOfData**. To guarantee no duplicates, write **Insert Element.vi** such that it checks the array for existing values before doing the insert. If you need to make sure the data stays sorted, you can write **Insert Element.vi** such that it inserts data at the correct position. Whatever functionality you supply – or leave out, as in the case of an array that does not allow removing elements – defines what can happen to the array.

The new class represents an aggregation of many instances of another class. Thus the name of this pattern.

Editorial Comment

This is an easy pattern to understand, and it is easy to think of the implementation on your own. Not all the patterns are complex systems. Sometimes pointing out the obvious is useful. When planning a new application, you may look at a list of patterns to consider what the best design would be. Having these so-called "obvious" patterns in the list helps remind you that you ought to use them. Because as easy as this pattern is, it is easier just to have a raw array running around on your diagram (you don't have to write a bunch of accessor VIs to duplicate the functionality already found on certain primitives). There are times when you'll choose to do that. But as your code gets more complicated over time, you may find yourself wishing that you had wrapped the array up so that you could guarantee certain characteristics.

Specification Pattern

Intent

To have different functionality for a class depending upon some value of the class without updating case structures throughout your VI hierarchy when you add another value.

Motivation

Your data has some field – perhaps an enum, perhaps a string, perhaps something more complex – and you have a case structure that cases out on that field to decide which of several actions to perform. The data in this field is constant (once you have initialized the class this field never changes). When you find yourself creating a lot of these case structures, or, worse, updating a lot of these structures to add a new case, you should remember this pattern.

Implementation

Examples of this pattern occur in just about every shipping example.

If the data is constant from the time the class is initialized onward, then you should strongly consider creating a set of child classes. The data in that field is unnecessary data – stop hauling it around the diagram. A class knows its data type. If you need that value, you can get it from some static VI on the class that just returns a constant. Further, those case structures can now be replaced with dynamic subVI calls. For every distinct case structure, create a new dynamic VI on the parent class and put the code that used to be in each frame of the case structure in one of the children's override VIs. Now when you add a new case, there are no case structures to update. So instead of one monolith class with an enum type, you now have many more specific classes. Thus the name of this pattern.

This pattern is frequently used in conjunction with the Factory pattern.

Editorial Comment

Like the Aggregation pattern, this is an "obvious" pattern. But it is easy to overlook the fact that a piece of data never changes. If data never changes, that data is effectively part of the data type. Clusters that carry around the name of their data or a string representing the source of the data cry out for application of this pattern. This pattern is basically the central use case for class inheritance and dynamic dispatching. It simplifies many very complex VIs and clusters into manageable chunks by giving an easy way to break up the functionality of the case structure across multiple VIs. This pattern is the first one that a new user of object-orientation should learn, and generally the first one to consider when refactoring existing applications.

Channeling Pattern

Intent

To provide a guaranteed pre-processing/post-processing around some dynamic central functionality.

Motivation

You have a class hierarchy. And you have some algorithm that you want to implement on the parent class. There is a core step of the algorithm that needs to be dynamic so that each child class can provide its own behavior. But it is unsafe to call that step directly – you want a class interface that guarantees the step is only called from the algorithm VI.

Implementation

No examples of this pattern are publicly available yet. Here is a summary of the pattern:

The parent class has the VI that implements the algorithm. Make the algorithm VI a static VI (no dynamic inputs). That way the child classes cannot override the algorithm itself and thereby avoid calling the pre-processing/post-processing. Make the dynamic step of the algorithm a dynamic VI that the child classes can override and *make the dynamic VI be protected in the parent class.* This forces all the children to use protected scope for their override VIs. No VI outside the class hierarchy will be able to call the dynamic VIs directly.

This pattern does not prevent child classes from calling the step on themselves or other child classes without going through the algorithm. It does make sure that all the VIs outside the class hierarchy always go through the algorithm VI.

The name of this pattern refers to the control of the flow of data – the data is channeled through the static VI to get to the dynamic VIs.

Editorial Comment

A customer asked me why all member VIs in classes aren't dynamic. The performance overhead of dynamic VIs is pretty small (and constant no matter how deep the class hierarchy or how many dynamic VIs there are on the class) such that it made sense to the customer to make all the VIs dynamic and only make static VIs if a performance problem actually arose. That would give maximum flexibility to the class. This pattern is one of several reasons not to make everything dynamic. If the algorithm itself were dynamic, child classes might decide to override the algorithm and skip the pre/post processing. This pattern is particularly useful for large programming teams where you might not know all the developers, or where child classes will be implemented by third parties. It lets the original author put compiler-enforceable guidelines into the code so that future programmers do not unwittingly write VIs that will create issues for the architecture.

Visitor Pattern

(adapted from Gang of Four's Visitor pattern)

Intent

To write a traversal algorithm of a set of data such that the traversal can be reused for many different operations.

Motivation

Suppose I have an array of integers. I need a function that will calculate the sum of those integers *(ignore, for the sake of argument, the fact that there's a primitive to do this)*. I drop a **For** loop and iterate over every value, adding each element to a running total in an uninitialized shift register. The value in the shift register when the loop finishes is my sum.

Now suppose I want to find the product of those integers. I write the same **For** loop and shift register, only instead of the **Add** primitive, I use **Multiply**. Everything about these two VIs is exactly the same except for the core action. How can I avoid duplicating so much code?

Dynamic dispatching does not immediately help in this case – I do not have any child types on any of the wires that I can dispatch on. I could make my traversal VI have a VI Reference input. Then, instead of calling a primitive, I could have a **Call By Reference** node. The VI that I pass in would have either the **Add** or **Multiply** primitive.

This is helpful, but only works if my two VIs have exactly the same connector pane. What if I need other information, other parameters, to do the job in that **Call By Reference** node?

Implementation

An implementation of this pattern is available here:
http://jabberwocky.outriangle.org/LV2OO_Style_Global_v1.0.zip

Instead of taking in a VI Reference, the traverse VI takes in an object of class **Action**. This is a class that you define. Create one child class of **Action** for each specific action you want to do during the traversal. Then instead of a **Call By Reference** node, call the **Do Action.vi** method. This will dynamically dispatch to the correct implementation. The **Action** object can have all sorts of data inside it that can augment the operation at hand. You can implement as many new **Action** children as you want without ever having to change your original traversal framework.

The traversal VI can get very complicated, far beyond the simple **For** loop over an array that I mention here. As the traversal walks over a data set, the **Action** object "visits" each piece of data and updates itself. Thus the name of this pattern. Your visitor can collect a summary of information about the pieces of data (such as the sum and product calculations), or it might search for a particular value, or it might even modify the data as it visits (divide each value in the array by 2). The traversal works just as well in all of these cases.

Editorial Comment

I am tired of writing LV2-style globals. These globals are great – very dataflow efficient, very easy to understand. But you have to duplicate the VI every time you want to support a different data type. I figured there had to be a better way. I've been thinking about this implementation for a while. It may not be optimal, but I think it is an excellent starting point. Imagine… the possibility of never writing another LV2-style global VI ever again, because you could just reuse one and give it the particular action that you want. A whole new flavor of über-geek nirvana!

Conclusion

Those are the design patterns thus far. I hope to update the list as time goes by, both with more patterns and with better examples for the existing patterns. Pay attention to the code you write – when you find yourself following the same routine over and over, try giving a name to the pattern. See if you can clearly identify when it is useful and how best to do it.

Exploration of these patterns helps us design better code. Using the patterns is using the learning of previous developers who have already found good solutions. In studying the patterns, we can learn what good code "feels" like, and we are then more aware when we create an unmaintainable hack in our own programs.

When we follow the standard designs it helps others looking at our code understand what they're looking at. If a programmer knows the general pattern being used, he or she can ignore the framework and focus on the non-standard pieces. Those are the points are where the program is doing its real work. It is akin to looking at a painting in a museum – the painting is the interesting part. The picture frame and the wall are just there to hold up the art. If the frame or wall are too decorative, they pull the focus.

I do not talk about any anti-patterns in this document. Anti-patterns are common attempts that look like good ideas but have some generally subtle problem that will arise late in development. Some anti-patterns in other languages are sufficiently detectable that a compiler could even be taught to recognize them and report an error. Perhaps these exist in LabVIEW. We should be cataloguing these as well.

For those who chose object designs,
we want the world of wire and node
to morph naturally
into a world of class and method.
– LabVOOP