Object-Oriented Design and Programming in LabVIEW™ Course Manual

Course Software Version 2010
November 2010 Edition
Part Number 325616A-01

Worldwide Technical Support and Product Information
ni.com

National Instruments Corporate Headquarters
11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the Info Code feedback.

Contents

4

Object-Oriented Tools and Design Patterns

This lesson introduces several common tools and design patterns used with object-oriented programming. You will learn to develop an object-oriented application in G that uses one or more of these tools or design patterns.

Topics

A. Object References and Construction Guarantees
B. Front Panel Displays for Object Data
C. Design Patterns: Introduction
D. Channeling Pattern
E. Aggregation Pattern
F. Factory Pattern
G. Delegation Pattern
H. Visitor Pattern
I. Design Patterns: Conclusion

A. Object References and Construction Guarantees

LabVIEW users occasionally find a need for passing data by-reference instead of by-value or to guarantee construction of data.

For example, you may want to share a block of data between two different parts of a program. This can be achieved if you pass data by-reference. There are multiple possible solutions to this need, and choosing the right solution for a given application can be confusing.

When instantiating an object in G, using its default value will satisfy most use cases. However, when an object's private data includes a reference that needs to be initialized for the object to be valid, it is useful to have some way of guaranteeing that the reference has been constructed before any methods are called.

Multiple G features can be used to meet these needs, but when the data in question is an object, data value references are generally preferred. Before delving into the solutions to these use cases, it is important to first understand what a data value reference is and its advantages/disadvantages.

What are Data Value References?

Data value references are refnum data types capable of serving as a reference to any G data. This data can be a plain integer, an array, a cluster, or an object.

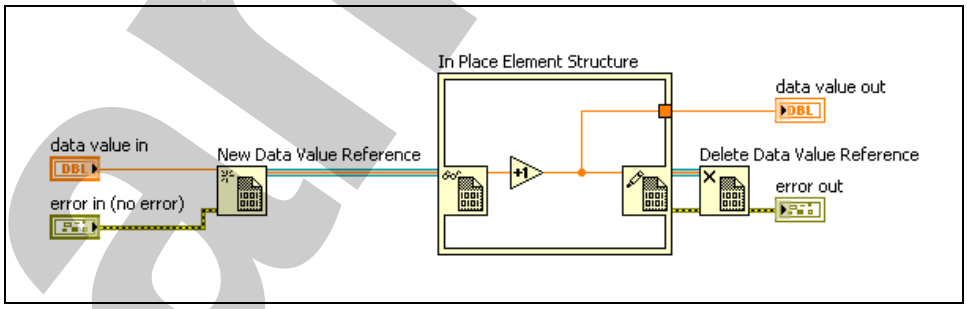

Figure 4-1. Data Value Reference Example

Data value references are created using the New Data Value Reference function. After creating a data value reference, you can pass it around instead of your data. Use the In Place Element Structure to access the data. If there is an error dereferencing the value (for example, if a refnum has been deleted), you get an error.

Use the following functions and structures to create and use data value references.

- New Data Value Reference (**Programming»Application Control» Memory Control**)—Creates a reference to data that you can use to transfer and access the data in a serialized way.

  Data value references contain the same data type of the data they point to. Do not type cast data value references. However, you can upcast or downcast data value references that contain LabVIEW classes.

- In Place Element Structure (**Programming»Structures**)—Controls how the LabVIEW compiler performs certain operations and, in some cases, increases memory and VI efficiency. Use this structure to operate on any data type that you want to maintain within the same data space in memory. Right-click the border of the structure and select the border node that matches the operation you want to perform.

  To work with a data value reference, add a Data Value Reference Read / Write Element Border Node to the structure. This node accepts a data value reference as an input, allows you to operate on the data within the In Place Element structure, and replaces that data in the original memory space. Right-click the border of an In Place Element structure and select **Add Data Value Reference Read / Write Element** from the shortcut menu to place this border node on the In Place Element structure. You cannot find these border nodes in the palette search or quick drop search.

- Delete Data Value Reference (**Programming»Application Control» Memory Control**)—Returns the data to which the reference points and then deletes the data value reference.

Advantages and Disadvantages of Data Value References

Advantages

- **Guarantee no unintended copies of data**—Because you are only passing a refnum, you are never in danger of accidentally creating a copy of your data.
- **Safe for parallel access for basic read-modify-write**—If you are using a global VI and you read the variable, modify the result and then write it back to the variable, it is possible that someone else has done a read-modify-write in parallel with you. Using data value reference gives you have an exclusive lock on the data for modifying the value.
- **No danger of deadlock specifically for out-of-order refnum access**—Single-element queues are good for basic data storage, but if you need to dequeue multiple queues at the same time, you can create deadlock if you do not dequeue from the refnums in the right order. The In Place Element Structure automatically sorts the data value references at the

border and accesses the refnums in order, thereby ensuring that this deadlock is impossible.

- **Build complex reference-based data structures**—It is possible to use LabVIEW classes to develop nested tree-like data structures. Data structures built with references are much richer. They can include optimized database systems, circular graphs, flyweight patterns, and so on.

Disadvantages

- **Potential for race conditions and deadlock bugs**—Whenever you pass data without using a wire, you create the possibility that a race condition or deadlock bug will occur. These bugs may occur sporadically and you might not detect them until your code is used in a built executable, has debugging turned off, or is deployed to an end-user's machine. It is easy to have parallel writes to these data value references which can create problems.
- **Violating dataflow stops compiler optimizations**—Data value references are designed to minimize the number of copies of the data structure as a whole, but when you add references to your code, the LabVIEW compiler loses some of its ability to optimize access to that data, particularly repeated accesses to the same sub portion of the data.
- **Overhead of data access can cause performance drop if overused**—If you do not need references, do not use them. The overhead associated with locking and unlocking the data can compound if you use a reference type where a value type would have worked.
- **Code readability decreased**—The greatest danger to large applications is the loss of readability. With dataflow, you can follow the wire, drill down into subVIs, and find where the information came from and where it is going. With references, you have no such ability. The data could be modified in some VI that isn't in your current VI hierarchy.

On the course CD, navigate to `<Exercises>\...\Demonstrations\DVR Demonstration.lvproj`. Open and run `DVR w Classes Demo.vi` to learn more about the risks of using data value references with LabVIEW classes.

Use references when you need references, but make them a last resort, not a tool of first choice.

Data Value References and Classes

Data value references can be used to solve the object reference and construction guarantee use cases for LabVIEW classes.

Object References

Data value references are the recommended solution for creating references to LabVIEW classes because they are castable. Unlike single element queues or global variables, data value references may be cast using the To More Specific Class and To More Generic Class functions. This allows specific type references to be used in a generic type framework.

If a data value reference points to a class object, you cannot replace that class object with a different LabVIEW class object inside the In Place Element structure. However, you can replace any other object with an object of the same type.

Construction Guarantees

With a by-value type, it is impossible to enforce constructor/destructor mechanisms because an object exists from the moment you drop a front panel control or block diagram constant. It may not be deallocated until after the VI stops running. A reference, however, has a defined lifetime. Therefore, it can guarantee initialization and uninitialization of data.

Initialization will occur in a “constructor” method for your class. Within this method, you will set the object data values and use the New Data Value Reference function to create a reference to that object.

Uninitialization will occur in a “destructor” method for your class. Within this method, you will call the Delete Data Value Reference function to destroy the reference. This VI should include any additional cleanup work that needs to be done when the reference is deleted, such as removing it from any static lists.

The Inheritance page of the Class Properties dialog box, shown in Figure 4-2, allows you to set restrictions on data value references.

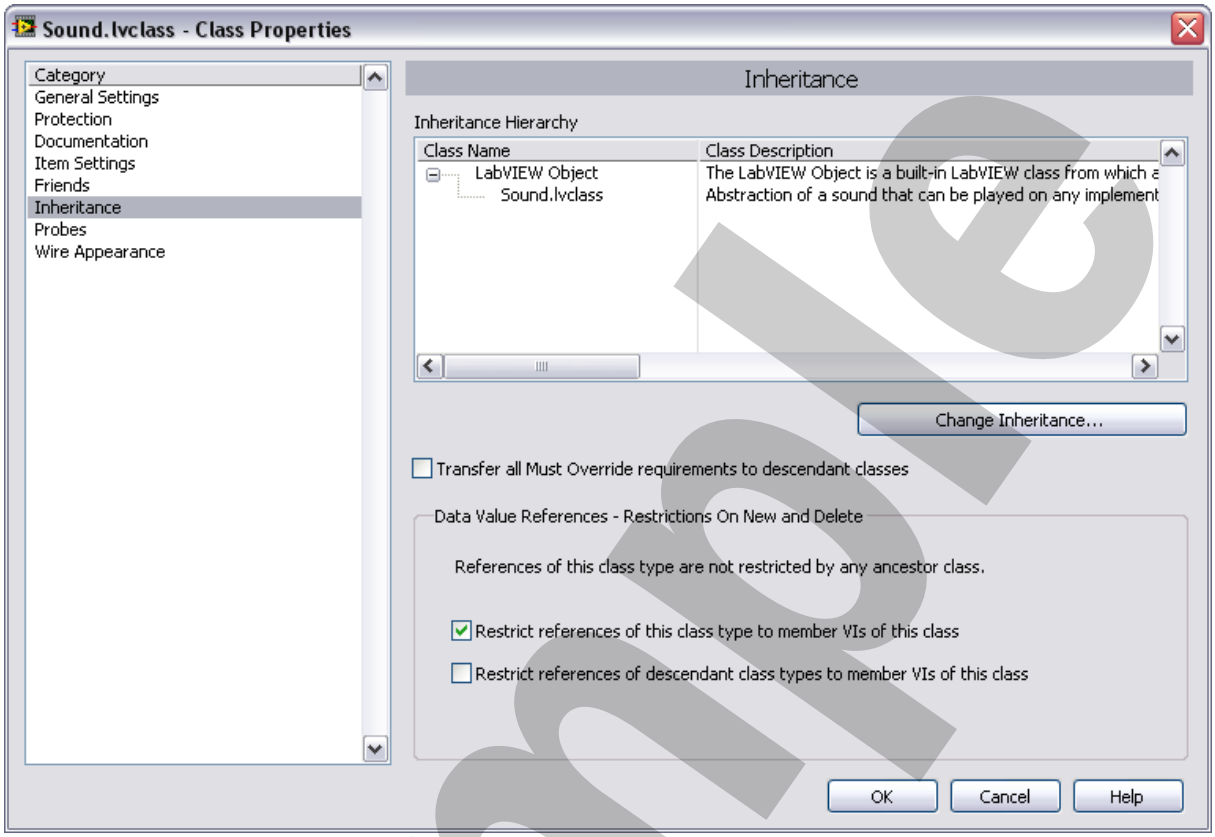

Figure 4-2. Class Properties Dialog Box—Inheritance Page

Use these options to set restrictions on data value references:

- **Restrict references of this class type to member VIs of this class**—Allows only member VIs of this class to create data value references to members of this class. If you create a new class, this option is enabled by default.
- **Restrict references of descendant class types to member VIs of this class**—Allows only member VIs of this class to create data value references to any descendants of this class. This option is disabled by default.

  This option allows you to create and enforce sophisticated factory patterns where the parent class produces all the descendant types based on an input.

  You will learn more about factory patterns in the *Factory Pattern* section of this lesson.

B. Front Panel Displays for Object Data

By default, if you place an object onto the front panel of a VI, the resulting control or indicator does not display any of the data stored within the object.

There may be times when you want to create a custom control so that your users can view and manipulate class data more directly. To meet this need, an XControl can be developed for your class.

What is an XControl?

An XControl is a control with customized behavior in addition to appearance. The `.xctl` file on disk is a specialized type of library. Like all libraries, it is a collection of VIs, but this library is dedicated to the specific purpose of defining a new front panel control.

Note XControl development is beyond the scope of this course. To learn more about creating and using XControls, refer to *LabVIEW Help* and the online help at `ni.com`. XControls are also discussed in greater depth in the Advanced Architectures in LabVIEW course.

An XControl aggregates the files that collectively define its appearance and behavior. An XControl is made up of four files (abilities) working in tandem:

- **Data.ctl**—The Data ability defines the data type of the control. This is the data type of the control's block diagram terminal. All reads and writes of the control value use this data type. The Data ability also defines the data in and data out terminals of the Facade. The Data ability can include complex data types such as clusters. However, you should store configuration information for the control in the State ability and access it through properties and methods rather than through the Data ability.
- **State.ctl**—The State ability defines any data, other than the actual terminal value, which the control needs to store persistently. This type definition defines the state information of the control. At any given time, the state of the control defines the behavior and appearance of the control. You should store the following data in the State:
  - Values altered by properties and methods
  - Configuration information
  - References
  - Local data storage
- **Init.vi**—The Init ability initializes the XControl. The Init ability is called when the XControl is added to a front panel window and when a VI that uses the XControl is loaded into memory. Use the Init ability to

allocate resources, open references, and initialize any state information that is not stored on disk with the VI. When an XControl saved with a VI is older than the current XControl version, the Init ability is also responsible for safely updating the state of the XControl to match the new version.

- **Facade.vi**—The Facade ability is the main engine behind an XControl. This VI defines the appearance and behavior of the XControl. The front panel window of the Facade VI defines the appearance of the XControl. The block diagram of the Facade VI defines the behavior of the XControl.

Typical uses of XControls

You can use XControls to accomplish the following tasks:

- **Add functionality to existing controls**—You can use an XControl to expand the functionality of existing controls. Encapsulating a control in an XControl enables you to add properties and methods. You can also add shortcut menu options and/or configuration dialog boxes for the control without having to add code to the client application. XControls also allow you to alter the data type of the control and include inherent data analysis or processing in the control.

  Refer to `ni.com/info` and enter the Info Code `rdsxca` to view the *String XControl with Autocomplete Functionality* example on the NI Developer Zone for more information about using XControls to add functionality to existing controls.
- **Combine Existing Controls**—You can combine the functionality of two or more existing controls to simplify the application using the controls. If your user interface contains controls which affect only the display or analysis of data for another control, you can combine these controls into an XControl and assign a single data type to both controls.

  Refer to `ni.com/info` and enter the Info Code `rdcnfl` to view the *Creating New Front Panel Objects with LabVIEW XControls* tutorial on the NI Developer Zone for more information about using XControls to combine existing controls.
- **Abstract UI Components**—You can use XControls to abstract all or part of a user interface. Separate the UI code from the application code to create cleaner applications that have fewer restrictions for design patterns. For example, you can process events for controls embedded in an XControl without using an event-based design pattern in your client application.

- **Create New Controls**—You can also use XControls to create entirely new controls. For example:
  - Draw and handle custom shapes on the picture control.
  - Create custom graph controls.
  - Combine or use existing controls in unusual ways!

Advantages and Disadvantages of XControls

Advantages

XControls have the following advantages over standard controls and design patterns:

- **Encapsulate UI code**—You can use XControls to abstract code specific to the user interface from the main architecture of a project. UI code includes tasks such as setting display properties, formatting data for display, and responding to events. Because UI code often exists in the top-level VI, it can increase the complexity of the main VI.
- **Create reusable and distributable UI components**—An architect can create a library of XControls and distribute them to multiple developers for use throughout the project. XControls make this process as robust as possible because an XControl consists of a single library that contains all code related to the control. XControls also aid in continuous improvement. You can update an XControl after distribution and the developer can define the behavior of existing control instances to ensure that the integrity of the control remains intact and settings are not lost.

Disadvantages

- Because an XControl on a block diagram looks much like other control types, debugging code that utilizes XControls can be difficult if the functionality and use of the XControl is not well-documented.
- XControls can be time-consuming to develop and maintain versus the use of standard controls.

Using an XControl as the Front Panel Display for Object Data

You can use an XControl as the front panel display for object data by placing an instance of the class into the data.ctl file for the XControl. When the XControl is placed onto a front panel, the data passed out of the XControl will now be of your class type.

Modify the front panel of Facade.vi to define how the class data will be displayed in the XControl. Modify the block diagram of Facade.vi to use class methods as needed. By default, the XControl will have access to the public methods that you have defined for your class.

When possible, declare an XControl as a friend of the LabVIEW class whose data will be displayed. The class methods required to display and manipulate class data no longer have to be wide open and public. They can now be declared as community scope so that the XControl will have access to the VIs it needs to do its UI work without exposing all of the inner workings of the class to all VIs everywhere.

In the LabVIEW Example Finder, browse to **Fundamentals» Object-Oriented»FriendshipAndCommunityScope.lvproj**, shown in Figure 4-3, to explore a class and XControl which demonstrate the use of friendship and community scope.

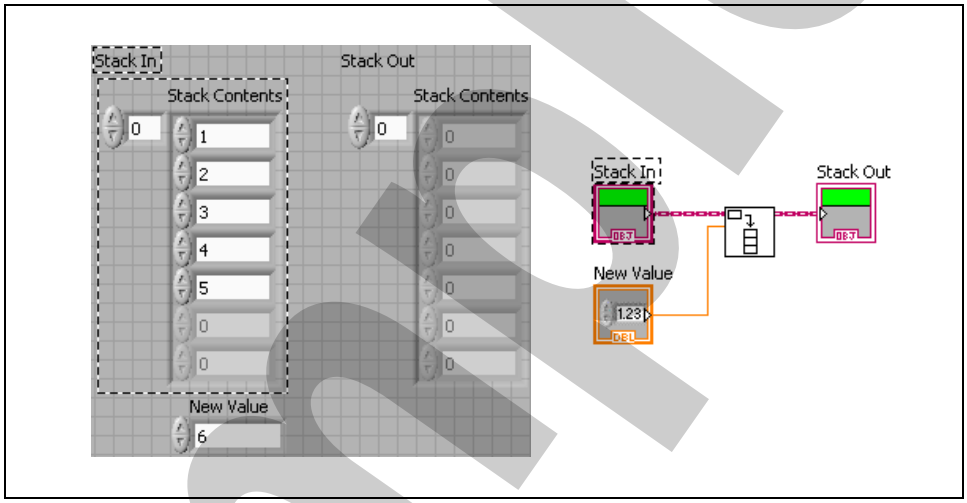

Figure 4-3. Friendship and Community Scope.lvproj:Example.vi

Caution Your class should not use the XControl in any of its member VIs. For best performance and to avoid complex load failures, the class should be written without any dependency on the XControl. The only reference a class or its member VIs should make to the XControl is in the list of friends of the class.

C. Design Patterns: 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. Design patterns are less specific than algorithms. You use the patterns as a starting point when writing the specific algorithm.

You may already be familiar with design patterns that ship with LabVIEW. Additionally, through reading books on LabVIEW, networking with other developers, or participating in the LabVIEW community, you may have become familiar with many of the design patterns, some of which are based on those that ship with LabVIEW.

With the introduction of object-oriented features in G, many of these traditional design patterns and new ones are now being solved with object-oriented solutions.

Every language develops its own design patterns. G has patterns such as "Producer/Consumer Design Pattern" or "Queued State Machine." These are not particular VIs. Many VIs are implementations of queued state machines. With the release of LabVIEW object-oriented programming, new patterns arise when objects mix with dataflow. These patterns become even more important as the size and complexity of object-oriented applications increase.

Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson & Vlissides is a classic text used by software engineers. Many of the design patterns covered in the following sections are adapted from this book.

D. Channeling Pattern

The channeling pattern was designed to guarantee the execution of pre-processing/post-processing code around some dynamic central functionality. The name of this pattern refers to the control of the flow of data. Data is channeled through a static VI to get to dynamic VIs.

Use the channeling pattern if you have a class hierarchy and you want to implement an algorithm on the parent class. A core step of the algorithm must be dynamic dispatch so that each child class can provide its own behavior. However, it may be unsafe to call that step directly—due to pre-conditions that must be checked or post-conditions you want to enforce. So you need an interface that guarantees the core is only called from a particular entry point.

Implementation

To make this pattern work, you give the parent class a static dispatch VI that implements the algorithm. That way the child classes cannot override the algorithm itself and thereby avoid calling the pre-processing/post-processing. Make the core step of the algorithm a dynamic dispatch VI that the child classes can override. Let the dynamic dispatch VI be protected scope 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 core VIs directly.

One common task that could benefit from the use of this pattern is the execution of file input/output.

- Pre-processing—Locate the existing file or create a new file, open a reference to the file.

- Dynamic dispatch—Different code may execute depending on the type of file that you are writing (binary, ASCII, etc).
- Post-processing—Close the file reference and handle any errors that may have occurred.

The sound player demo project utilizes this pattern in the Wait For Current Play to Finish method of Sound Player.lvclass, shown in Figure 4-4.

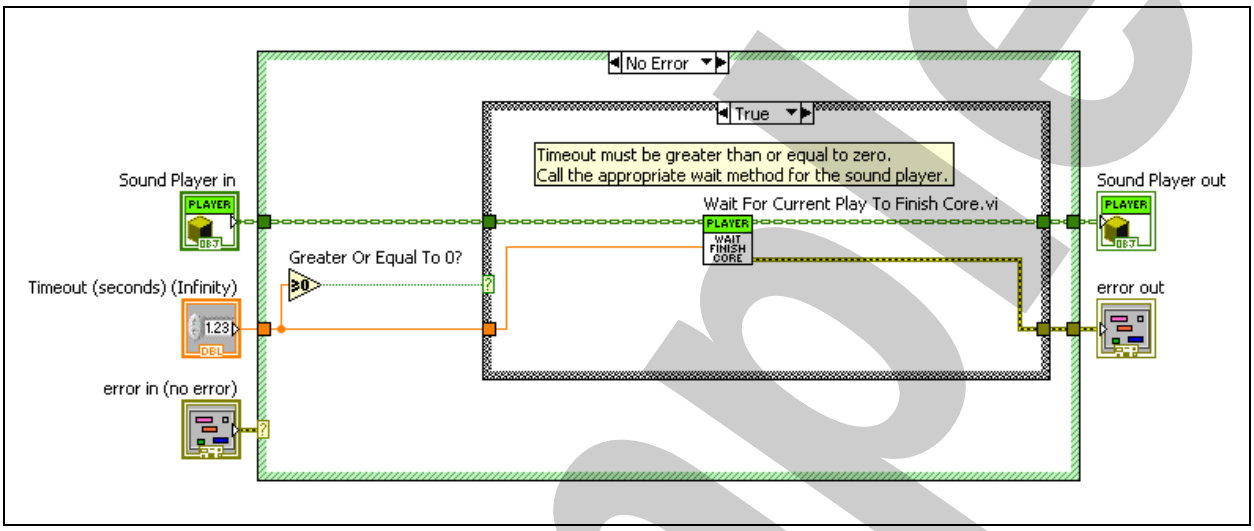

Figure 4-4. Channeling Pattern—Sound Player.lvclass:Wait For Current Play to Finish.vi

In this example, we determine whether the value of Timeout is positive or negative as a pre-processing step.

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 go through the algorithm VI.

This pattern is one of several reasons not to make every VI dynamic dispatch. If the algorithm itself were dynamic dispatch, child classes might decide to override the algorithm and skip the pre/post processing.

The Channeling 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.

Benefits

- You can protect the algorithm of a method by defining it statically. Child classes will not be able to modify the algorithm or otherwise change the order of execution of processing steps.
- The main functionality of the algorithm is abstracted as a dynamic dispatch method. This provides improved scalability since expanding the pattern to work with new class types will only require creating an appropriate override method as part of that class.

E. Aggregation Pattern

The Aggregation pattern enables you to treat an array of objects as a single object and define special behavior for that particular type of array.

The name of this pattern comes from the fact that you are creating a data aggregator—that is, this new class is a data type that represents some aggregation of many instances of another data type. The concept may expand far beyond just a simple array to become any number of collections of data such as lists, sets, maps, and so on.

Here are several examples of array properties that you may want to guarantee:

- No duplicate entries
- Array is always sorted
- Array elements cannot be removed

Implementation

To begin, identify the invariant properties that you want the array to maintain. Create a class with a name that reflects that invariant. If this is an array that should always be sorted, you might create a class named `Ordered Array.lvclass`. Perhaps it is an array that has no duplicate elements. Then you might create a class named `Unique Elements Array.lvclass`. Regardless of the name you choose, you will give the class an array of some data type as its private data.

Define methods on the array class that maintain the invariant. So instead of a raw array that allows arbitrary modification using the array primitives, you now have an encapsulated array that only allows the specific actions that you define. For example, if the array is to be always ordered, you would create Insert Element.vi such that it inserts the new element in the correct sorted position in the array. 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 sound player demo project utilizes this pattern in the Write Element of Sounds method of Aggregate Sound.lvclass, shown in Figure 4-5.

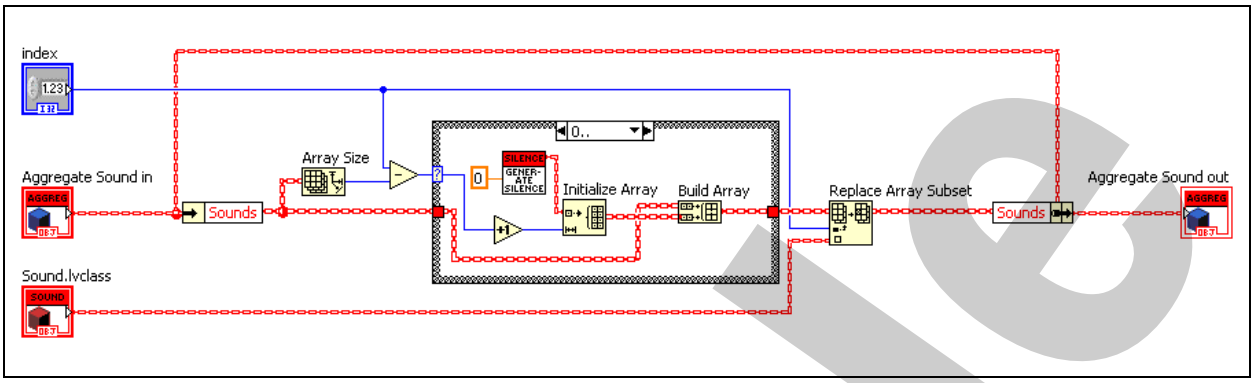

Figure 4-5. Aggregation Pattern—Aggregate Sound.lvclass:Write Element of Sounds.vi

In this example, if the index input provides a value that is larger than the size of the array, a buffer of Silence objects will be added to the array as a buffer between the last array element and the index that was specified. Thus, we are guaranteeing the behavior caused by using an index larger than the size of the array.

The sign that you might want to apply this pattern is when you start referring to an array of the type by some specific name. For example, you might have a class called Athlete and you have an array of Athletes that you're passing around and calling "the team", and you keep checking to make sure that no Athlete is on the same team twice. This is a good sign that you should create a Team class instead of using the array directly.

Benefits

- You can guarantee characteristics of your class array.
- Limiting array variations can lead to optimizations. By limiting the variations of your collection, you may be able to optimize some cases. For example, if all the functions on the class that can insert elements guarantee that the array stays sorted then you can use faster algorithms when searching the array. A sorted array can use a binary search, which is a much faster way of searching an array.
- Your array can have its own custom wire appearance. You might be tempted to give the new class a wire appearance that looks like an array of your element data type. That is generally confusing because the class wire will break if it is wired directly to any of the array primitives.

The major drawback to this pattern is the amount of effort required. It can be a lot of work to re-implement all of the common array operations for your special array type. Fortunately, most of the time when these invariants are

required, not all of the array operations are commonly needed. 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) and there are times when you might choose to do that.

However, 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.

F. Factory Pattern

The factory pattern is designed to 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 an 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.

The name of this pattern comes from the fact that a single VI serves as a factory to produce all the myriad child types.

Data may come from many sources – user interface, network connections, 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.

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

Any application using LabVIEW classes generally has a framework that is written using the parent class, and then at run time, child classes travel on the wires through the framework. This pattern provides a mechanism for dynamically selecting the child classes, including dynamically loading them into memory after the program starts running, if necessary.

Complete the following steps to implement the factory pattern:

1. Use some bit of data to identify the class you are interested in passing through the framework.
2. Place a default instance of that class on the parent class wire that is used to pass the class data generically through the framework. This can be done using a case structure with different class constants wired to the parent class output tunnel, depending on which case executes.

3. After you have used the factory to get a particular class of data, you are free to use it as you would any other class. Commonly, you would wire the chosen class instance to an Init method of the class which would take your data as an input.

The sound player demo project utilizes this pattern in the Setup method of Sound Player.lvclass, shown in Figure 4-6.

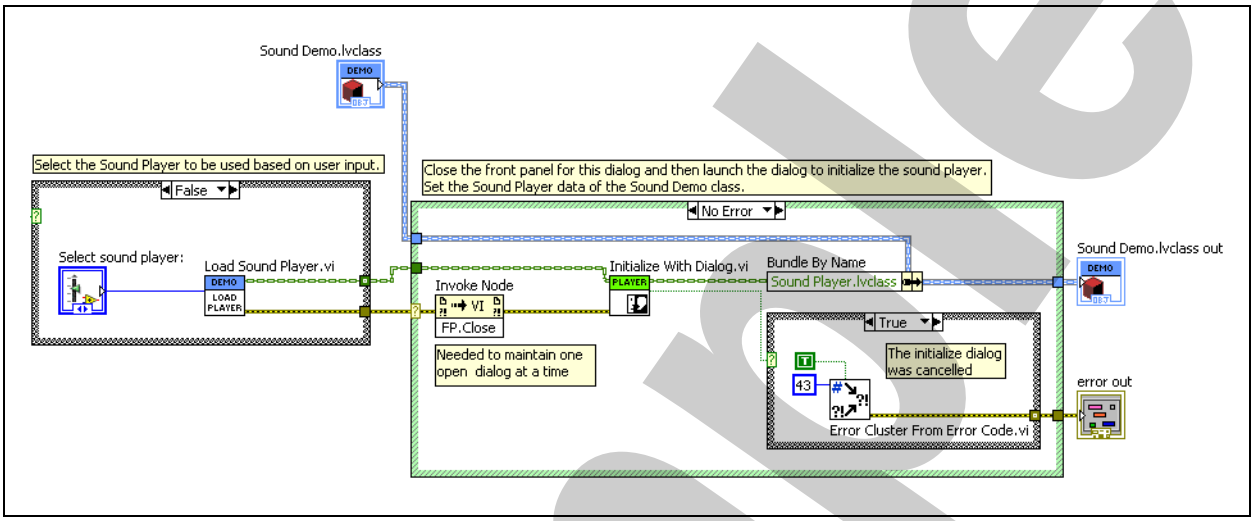

Figure 4-6. Sound Player.lvclass:Setup.vi as and Example of the Factory Pattern

Note This code is implemented as part of Exercise 4-2.

In this example, the Load Sound Player method dynamically load the sound player class that will be passed to the Initialize With Dialog dynamic dispatch method.

Benefits

- This pattern leverages dynamic dispatch to remove the need for additional code to make execution decisions based on the type of the data that is passed along. This results in a much cleaner block diagram.
- When the factory pattern is used to dynamically load plug-in classes, the child classes are only loaded into memory if they are instantiated. If you have many possible types of data, this can save on memory and load time for your application.

G. Delegation Pattern

The delegation pattern allows you to have two independent classes share common functionality without putting that functionality into a common parent class. The name of this pattern is derived from two descendent classes delegating the work to a third class.

Good object-oriented design requires each class to focus on its assigned task. A class should not 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.

Implementation

To implement the delegation pattern, 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.

This pattern is most applicable 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.

Benefits

- Delegation helps you avoid writing the implementation multiple times, once for each of the classes that needs it.
- Delegation helps you avoid placing shared functionality into a parent class when it is possible that other child classes could be implemented that should d not have access to that functionality.
- You can switch out the delegate object at run-time.

H. Visitor Pattern

This pattern is designed to write a traversal algorithm of a set of data such that the traversal can be reused for many different operations.

For example, if you have an array of integers and you need a function to calculate the sum of those integers. You can use 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 the sum.

If you want to find the product of those integers, you can use the same For Loop and shift register, only instead of using Add, you use Multiply. Everything about these two VIs is exactly the same except for the core action. How can you avoid duplicating so much code?

Dynamic dispatching does not immediately help in this case—you do not have any child types on any of the wires that you can dispatch on. You could make your traversal VI have a VI Reference input. Then, instead of using a primitive, you could use a Call By Reference node. The VI that you pass in would have either the Add or Multiply primitive.

This is helpful, but only works if your two VIs have exactly the same connector pane. What if you need other information, other parameters, to do the job in that Call By Reference node? What if there is more than one running total that you are keeping track of? The Visitor Pattern helps answer these questions.

Implementation

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 dynamically dispatches 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 new Action children without changing the original traversal framework.

The traversal VI can get very complicated, far beyond the simple For Loop over an array that is discussed here. As the traversal walks over a data set, the Action object “visits” each piece of data and updates itself, which explains 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.

Benefits

- This pattern provides many of the same benefits as functional global variables (discussed in greater detail in LabVIEW Core 1, 2, and 3). However, this pattern helps you to avoid duplication of the functional global variable framework (While Loop with a shift register) for each separate piece of data that you wish to store.
- You can create this framework once and expand/reuse it for many additional actions.

I. Design Patterns: Conclusion

Exploring the design patterns covered in this lesson helps you design better code. Using the patterns is using the learning of previous developers who have already found good solutions. In studying the patterns, you can learn what good code feels like and you are more aware when you create an unmaintanable hack in your own programs.

When you follow the standard designs, it helps others looking at your code understand what they are 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.

Table 4-1. Design Pattern Comparison

Design Pattern	Usage
Channeling	Ensure that all pre/post-processing steps will always be executed
Aggregation	Guarantee certain array characteristics, for example, always sorted, no duplicate elements
Factory	Initialize the value on a parent wire with data from many different child classes based on some input value.
Delegation	Develop two independent classes that share common functionality without putting that functionality into a common parent class.
Visitor	Develop a transversal algorithm of a set of data such that the traversal can be reused for many different operations.

Refer to the Developer Zone document, *Moving Common OO Design Patterns from Other Languages into LabVIEW* at `ni.com/zone` for more information about design patterns and additional examples.

Summary: Quiz

1. Which design pattern should be considered if you want your code to always create a log before a dynamically dispatched processing step and then always close that file afterward?
   a. Channeling Pattern
   b. Aggregation Pattern
   c. Factory Pattern

2. Which design pattern should be considered if you want to generically handle data for a DUT in your top-level application and then dynamically initialize the object and call into more specific DUT types for processing?
   a. Channeling Pattern
   b. Aggregation Pattern
   c. Factory Pattern

3. Which design pattern should be considered if you want to define a two dimensional array where the number of rows will always equal the number of columns?
   a. Channeling Pattern
   b. Aggregation Pattern
   c. Factory Pattern

Summary: Quiz Answers

1. Which design pattern should be considered if you want your code to always create a log before a dynamically dispatched processing step and then always close that file afterward?

   a. Channeling Pattern

   b. Aggregation Pattern

   c. Factory Pattern

2. Which design pattern should be considered if you want to generically handle data for a DUT in your top-level application and then dynamically initialize the object and call into more specific DUT types for processing?

   a. Channeling Pattern

   b. Aggregation Pattern

   c. Factory Pattern

3. Which design pattern should be considered if you want to define a two dimensional array where the number of rows will always equal the number of columns?

   a. Channeling Pattern

   b. Aggregation Pattern

   c. Factory Pattern

Notes