



Inheritance in GOOP

Taking GOOP Design One Step Further

by Mattias Ericsson and Jan Klasson



This article builds on the concepts discussed in the “What Is GOOP?” article on page 12 of this LTR issue. While these concepts are fairly advanced for many users, they can be powerful tools to add to your process, especially when developing large object-oriented projects. Provided on this issue's LTR Resource CD is a link to download a limited functionality demo of the toolkit used to create the example presented in this article.

–Editor

This article builds on the basic principles of graphical object-oriented programming described in this issue's “What Is GOOP?” article by David Hoadley, PhD and shows a technique for making LabVIEW™ object oriented programs even more flexible.

With an understanding of the basic concepts of object oriented programming – objects, classes, methods, and attributes – established, the design possibilities can be taken one step further by using inheritance.

Object Oriented Programming and Inheritance

Inheritance describes the relationship between classes. The inheritance within a group of classes forms a hierarchical structure, with the more general class at the top and more specific classes inheriting from those classes. The methods of the top class define the common set of services provided within its inheritance hierarchy. All subclasses must have the methods of the top class; but with different implementations. This implies that they are variations of the same concept. For example if the top class is called Telephone and provides the method (VI) MakeCall, there could be two classes – GSM_Telephone and 3G_Telephone – which inherits from the Telephone class. The GSM and the 3G phone would both provide the method MakeCall, but with different implementations.

The Need for Flexibility

Consider the example of a system that controls an instrument, a spectrum analyzer, and periodically records the data in a spreadsheet. Over several revisions, the requirements for the system expand to accommodate additional models of spectrum analyzer and output to a database instead of a spreadsheet. For compatibility and other deployment-related issues, the system

must still support the earlier model of instrument and spreadsheet output.

Regardless of the model of instrument or type of output, there are two fundamental tasks in this system – spectrum analysis and measurement storage. The challenge is to implement these tasks such that the addition of a new instrument or storage type does not require significant redesign.

One solution is to add a Case structure each place voltage is measured or a measurement is stored, but this soon will clutter the code as more instruments, storage types, or other requirements are added. A better solution is to make two subVIs that perform the tasks of spectrum analysis and measurement storage. The drawback with this is that we still have to wire up the case structures in the diagram of these VIs for each new implementation added.

With an understanding of the basic concepts of object oriented programming – objects, classes, methods, and attributes – established, the design possibilities can be taken one step further by using inheritance.

An Interface-Driven Approach

The object oriented way of achieving the flexibility needed in this example is to introduce a class for each task.

Structuring the Functionality into Classes

The first step is to create the SpectrumAnalyzer class and specify its methods. The following list provides some example methods for this class:

- Create (creates a new object and initializes it)
- SetFrequencyStartAndStop (sets start and stop frequencies)
- Sweep (does a sweep)
- GetSpectrum (returns the spectrum)
- Destroy (destroys the object and performs any “cleaning” tasks necessary)



Tools and Techniques

The SpectrumAnalyzer class defines the interface (the method VIs) of any spectrum analyzer in the system. When new requirements are needed, new methods will be added to the class. This interface-driven approach means that the system is structured based on the interfaces needed rather than the variety of implementations there are for these interfaces.

The next step is to create a class for each spectrum analyzer that is used in the various deployed systems and let them inherit from the generic SpectrumAnalyzer. For example, the system could use a Rohde&Schwarz spectrum analyzer (though any spectrum analyzer could be used) or a simulated spectrum analyzer, which yields the R&SFSEM30 and SpectrumAnalyzerSimulated classes. These two classes contain the specific implementation for the respective spectrum analyzer model.

Because the R&SFSEM30 and SpectrumAnalyzerSimulated classes both inherit from the SpectrumAnalyzer class, they will have the same methods (VIs) as the SpectrumAnalyzer class. However, they can have their own implementations of these methods. For example the Sweep method on R&SFSEM30 will use GPIB to order the instrument to perform a sweep. The same method on the SpectrumAnalyzerSimulated class will simulate the sweep behavior, possibly by sending fake data or by returning no data at all.

The Configurable Generic Functions – Virtual Methods

The next step is to define a generic function (VI) and configure it to use the right implementation without any

wiring using *virtual methods*. A virtual method (VI) is a method that performs the following tasks:

1. Takes an object reference as input and in run-time.
2. Checks what class the object was actually created from.
3. Searches the inheritance hierarchy bottom up.
4. Executes the first implementation of this method that is found.

To exemplify this let us assume the application GUI is using our SpectrumAnalyzer class. Since we are using real hardware the GUI has created an object of the class R&SFSEM30. Now the GUI wants to make a sweep, this is done by calling the virtual method Sweep on the SpectrumAnalyzer class. Using virtual methods, the application identifies the R&SFSEM30 object and executes the R&SFSEM30 class Sweep method. Because this is handled by the use of virtual methods, there is no Case structure.

The use of virtual methods ensures that the system can easily be extended with new spectrum analyzers without any modification of the virtual method calls. The *only* code change is the addition of new spectrum analyzer classes in the place in the application that creates new objects. To create a new object, the Create method (VI) of the actual class must be called, but once the object is created it can be used (and destroyed) using virtual methods without any dependency or knowledge about what class it was actually created from.

These concepts of inheritance and virtual methods are exactly the same as is found in many object oriented programming languages, such as C++, C#, and Java™.

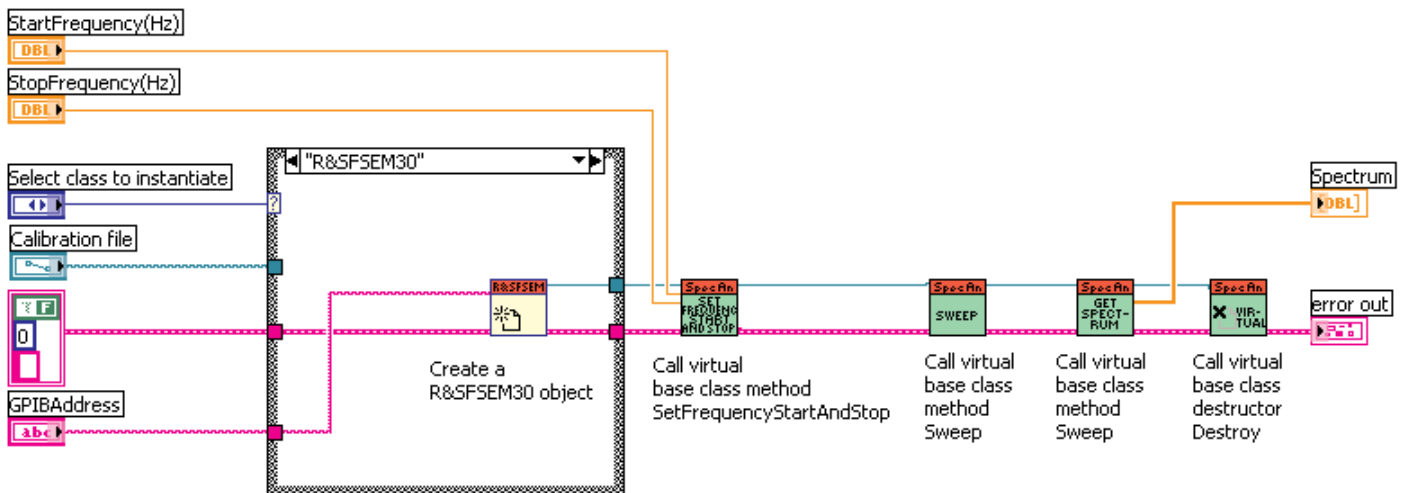


Figure 1: Spectrum Analyzer Application

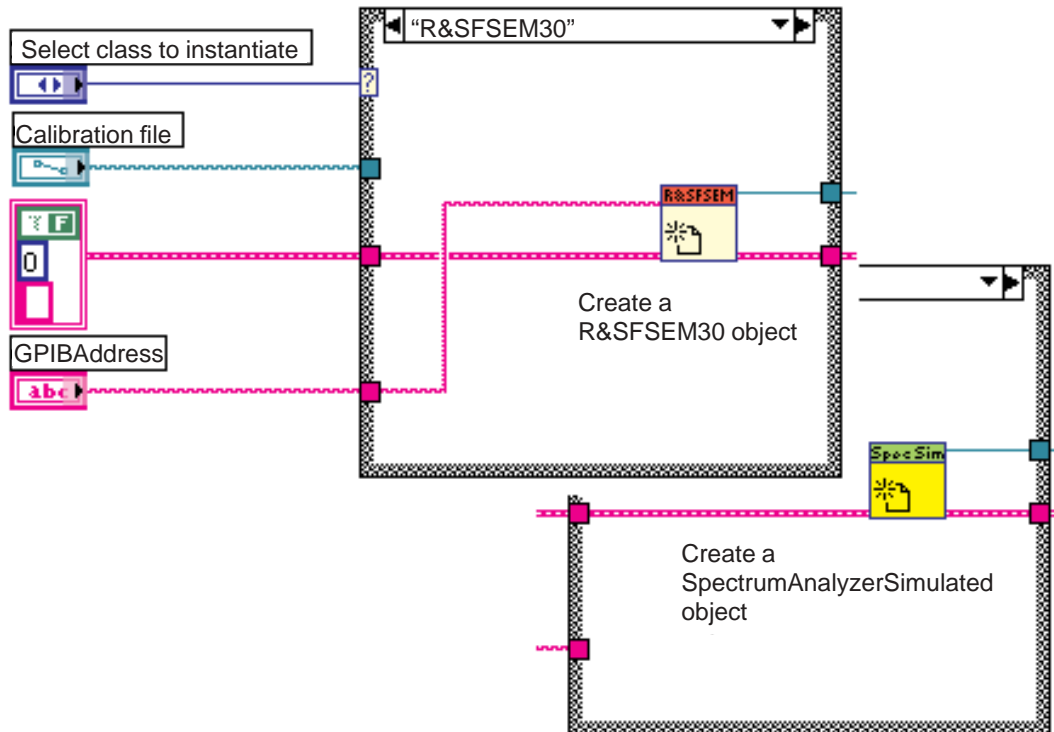


Figure 2: Case Structure Used to Create Objects

A Look at the LabVIEW Implementation

The example included on this issue's LTR Resource CD was created using a toolkit called the *GOOP Inheritance Toolkit*. This toolkit includes the *GOOP Wizard 3* which lets you generate new classes and lets you select if a new class inherits from an existing class.

Figure 1 shows the diagram of the application.

The case structure selects the class from which to create an object. Object creation returns an object reference, which is wired into each of the virtual method calls made. After the object is created, the following virtual methods are called in order:

1. SetFrequencyStartAndStop
2. Sweep
3. GetSpectrum
4. Destroy

The virtual method calls are made by using the corresponding method VIs of the SpectrumAnalyzer class.

Object Creation

Objects are created by calling the Create VI (method) on the desired class. *Figure 2* illustrates two of the three cases of object creation. In this example, the third case, creating an object from the SpectrumAnalyzer class, does not make sense because that class exists only to define the interface of spectrum analyzers. Other scenarios exist in which creating objects from the top class in an inheritance hierarchy makes sense.

Regardless of which class the object is created from the reference is wired into the virtual method VIs of the SpectrumAnalyzer class.

Virtual Methods

What appears to be a call to a method VI on the top-level SpectrumAnalyzer class results in the corresponding method on the actual object class to be executed. For example, to make the GetSpectrum virtual method call, place the VI SpectrumAnalyzer_GetSpectrum.vi on the VI diagram (as shown in *Figure 1*) and wire the object reference to it. If the object is created from the R&SFSEM class, this virtual call results in the R&SFSEM30_GetSpectrum.vi to be executed.

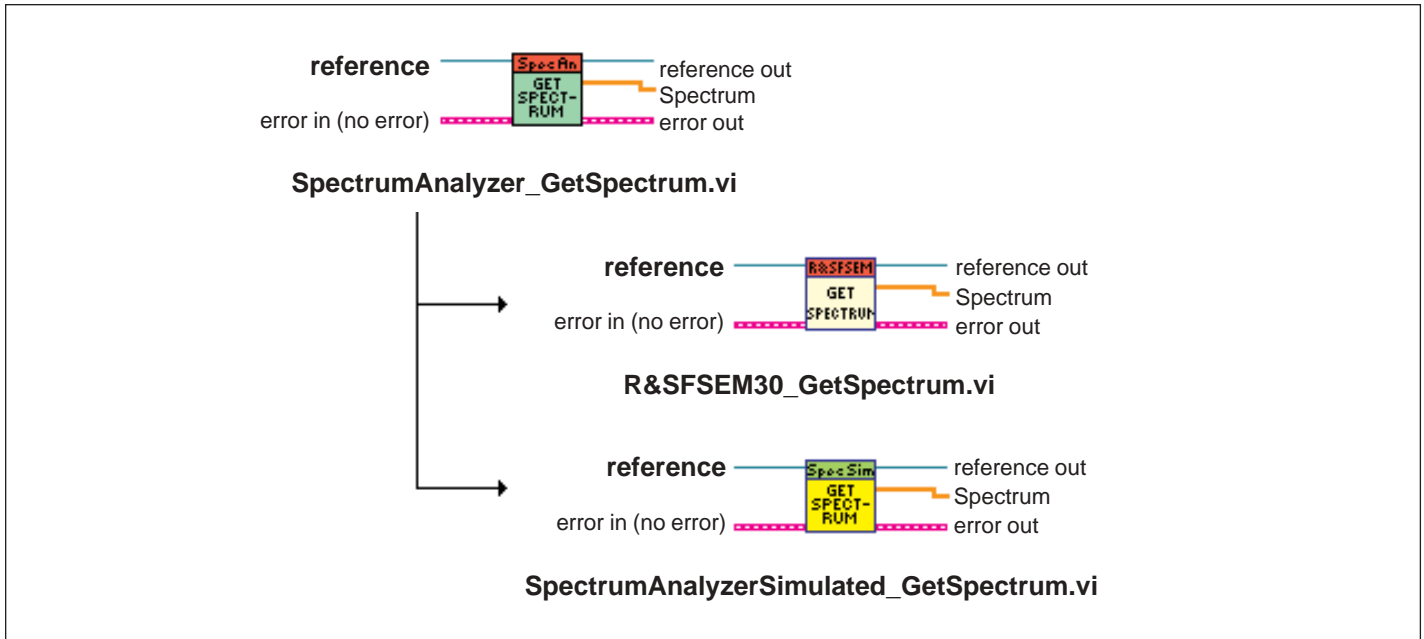


Figure 3: VIs of the SpectrumAnalyzer Class

The VIs for the GetSpectrum method have the same controls and indicators and connector pane, as shown in *Figure 3*. What differs is the class name prefix in the VI name.

Figure 4 illustrates the diagram of the method GetSpectrum of the SpectrumAnalyzer class. All code in this diagram is generated by the GOOP Wizard 3 tool. The VI R&SFSEM30_GetSpectrum.vi is run using the Call By Reference node.

Adding a New Spectrum Analyzer

Using inheritance, you can add support for a new spectrum analyzer by following these steps:

1. Implement the new spectrum analyzer class and verify that it works by itself.
2. Update the case structure in the top-level VI to create objects of the new spectrum analyzer.

Because adding a virtual method for the new instrument does not modify the block diagram of the top-level VI, there is much less need for testing the entire application.

To add new capabilities to this application we extend the code rather than modifying existing code. Avoiding modifications is good because if we modify existing code, we also need to retest that functionality.

General Guidelines for the Use of Inheritance

To determine if inheritance is appropriate for an application, first determine what things will vary in the system. When you have identified some things which will vary, the next step is to find the concepts (like signal generator, result manager, and so on) that are used by these variations. The challenge is to find a good way to structure these concepts into classes and, looking for commonalities, utilize inheritance wherever it can be effective. When deciding if inheritance is the proper design to follow, consider the following questions:

- Does inheritance add needed flexibility? If you know the application needs the flexibility, then inheritance could enhance the design. If you try to anticipate future cause for flexibility, you want to reconsider. While it can prove very useful, inheritance does introduce additional complexity, such as the need for at least two classes in an inheritance hierarchy when just one class might be appropriate.
- Do the classes in the same inheritance hierarchy represent the same concept? Here's a simple test: For each class, say the class name and then "is a" followed by the name of the class which is inherited from. If this does not make sense they do not represent the same concept.

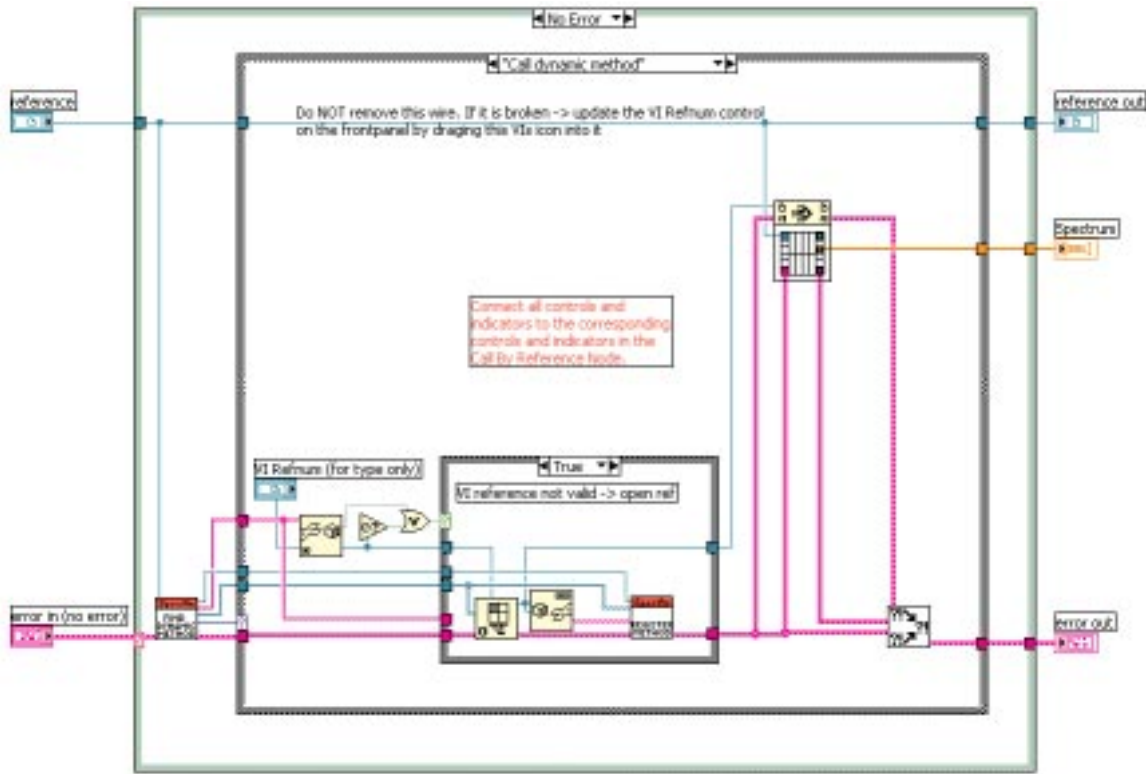


Figure 4: Call By Reference Implementation of a Virtual Method

- Do all methods of a class have a natural meaning on the classes inheriting from this class? This is sometimes called *subtyping*. For example if the top class is Bird and there is a Fly method, the class Ostrich will not work as they cannot fly. Therefore letting Ostrich inherit from Bird is probably not a good idea even if an ostrich is a bird.

Inheritance hierarchies are normally shallow. Do not put all your classes into the same inheritance hierarchy and avoid many levels of inheritance.

Misuse of Inheritance

Inheritance is not always the best solution. The following examples demonstrate scenarios where improvements could be made in the structuring of classes and use of inheritance.

- Case structures that are used to check the actual class of the object – OO programming and inheritance normally remove Case structures. This could indicate you are not using subtyping.
- All classes are a part of an inheritance hierarchy – It is unlikely you will find this much use for inheritance.
- Many levels of classes in the inheritance hierarchy – There are no strict numbers but typical inheritance hierarchies have less than five levels. Many levels of inheritance can make the code (within the inheritance hierarchy) complicated to maintain and understand.

Conclusion

We described the need for flexibility in systems and looked at some ways to achieve this; thereafter we proposed inheritance and virtual methods as a good solution and showed benefits with this approach. We have also shown that although OO and inheritance are not built-in features of LabVIEW™, they can successfully be applied by using tools like Endevo's *GOOP Inheritance Toolkit*.



About the authors:

Jan Klasson is Vice President of R&D measurement systems at Endevo – a Swedish Alliance Member - and he is also teaching the LabVIEW System Design with GOOP and Advanced Application Development courses in Sweden and Europe. He has been working with object oriented development for 13 years, using C++ and LabVIEW. For the last 7 years, Jan has been working as a consultant using LabVIEW in different measurement projects. He can be reached at jan.klasson@endevo.se.

Mattias Ericsson is system architect at Endevo and he is also teaching the LabVIEW System Design with GOOP course in Sweden and Europe. He is the main developer of the GOOP Wizard 3.0 and GOOP 2.0. He can be reached at mattias.ericsson@endevo.se.