

GOOP Inheritance Toolkit

White Paper

Mattias Ericsson



Abstract

The new GOOP Inheritance Toolkit consists of:

- GOOP 2.0 – a new GOOP Kernel Implementation
- GOOP Wizard 3.0 – Tool for creating and editing classes. Improved debugging feature by the new tool *Objects In Memory*. Automatic icon generation.
- Development Distribution – Tool for making distributions and executables based on GOOP 2.0

GOOP 2.0 has all features of GOOP 1.0 but also supports *class attributes, inheritance* and *dynamic binding*. Attributes and methods may be inherited publicly. Methods may be declared *virtual* and be redefined in a sub class. All these new features make LabVIEW™ a *true object-oriented language*.

The GOOP Inheritance Toolkit is easy to use and a user of the “old” GOOP 1.0 will easily adapt to the new Toolkit and get going in no time. The GOOP Wizard 3.0 also work with “old” GOOP 1.0 and an application may very well consist of GOOP 2.0 classes mixed with “old” GOOP 1.0 classes.

System requirements are LabVIEW™ 6.1 (or later) Professional Development System or Full Development System and it runs on Windows, Mac, UNIX and Linux platforms.

Contents

1	Purpose.....	4
2	References.....	4
3	Introduction.....	4
3.1	Background.....	4
3.2	Object-orientated foundations.....	5
4	GOOP Inheritance Toolkit.....	5
4.1	GOOP Wizard 3 and Objects In Memory tool.....	6
5	Object creation and destruction	8
6	Attributes	9
7	Class attributes.....	10
8	Inheritance.....	10
8.1	Attributes.....	11
8.1.1	Class attributes	12
8.2	Methods.....	12
8.2.1	Virtual methods.....	13
8.2.2	Base class implementation in a sub class method.....	18
8.3	Constructor – the Create method	20
8.4	Destructor – the Destroy method	21
8.5	Abstract classes	23
9	Object status	24
10	Inspector	24
11	Error handling	26
12	Distribution.....	27
12.1	Development distribution.....	28
12.2	Building executable	29
13	Performance issues, common mistakes and pitfalls.....	30
13.1	Performance	30
13.2	Common mistakes and pitfalls.....	31
13.2.1	Deadlock situations	31
13.2.2	Virtual methods do not work	32
13.2.3	Other common mistakes	32
14	System requirements	33
15	Example	33
15.1	The challenge	33
15.2	The design solution	33
15.3	The implementation	34

1 Purpose

The purpose of this document is to present the new *GOOP Inheritance Toolkit*, what's in it, what's new and how to use it. It is assumed that the reader has basic knowledge of object-orientation.

If you are a GOOP user today, but do not know anything about inheritance, this document will give a good introduction to this, but will not explain it in detail. It is recommended to study some additional books in object-oriented design to fully benefit from the advantages of inheritance. The reference list in chapter 1 is a good start. A good idea is to take a look at the example in chapter 15 to get a glimpse of what it is all about before reading this paper from start.

2 References

- [1] LabVIEW™ System Design with GOOP. Course material. Available at www.endevo.se
- [2] Eriksson, Hans-Erik and Penker, Magnus: UML Toolkit. ISBN 0-471-19161-2
- [3] Booch and Rumbaugh: UML – The Unified Modeling Language User Guide. ISBN 0-201-57168-4
- [4] Jacobsen, Ivar: Object-Oriented Software Engineering, A Use Case Driven Approach. ISBN 0-201-54435-0
- [5] Gamma; Helm; Johnson; Vlissides: Design Patterns – Elements of Reusable Object-oriented software. ISBN 0-201-63361-2
- [6] Development Distribution – White Paper. Available at www.symbio.com
- [7] Icon Editor – White Paper. Available at www.symbio.com

3 Introduction

3.1 Background

The GOOP history started back in 1994 when one of the founders of Endevo (now known as Symbio), Jörgen Jehander, developed the first GOOP. This brought some of the object-orientation features into LabVIEW™ development. In 1997, the first GOOP course was held in Sweden, and ever since then has been teaching the power of object-orientation.

In 1999, the first GOOP Wizard 1.0 was launched and a cooperation with National Instruments was started that resulted in the "old" GOOP 1.0 kernel, which is part of the LabVIEW™ installation (LabVIEW™ 6). The GOOP Wizard 1.0 lacked a lot of editing functionality which led to the launch of the GOOP Wizard 2.0 in the spring of 2002 that was a tremendous improvement in class creation and editing, making it really easy to work with GOOP.

Object-orientated foundations

A true object-orientated language must support the following pillars:

- Encapsulation
 - Public methods
 - Private attributes
 - Encapsulation of data
- Inheritance
 - Methods and attributes can be reused in more specific subclasses
- Polymorphism¹ (dynamic binding)
 - Implementation of methods can be redefined in subclasses and dynamically binded at runtime

The “old” GOOP 1.0 as known today only supports the first pillar and do not support inheritance and polymorphism and could therefore be called an *object-based language* rather than object-orientated. The GOOP Wizard 2.0 was only a new tool for creating and editing classes based on GOOP 1.0 and not a new GOOP implementation itself.

As the leading developer of object-orientated support for LabVIEW™, Symbio is proud to present the new *GOOP Inheritance Toolkit*. This new toolkit supports all of the pillars mention above. The Symbio GOOP Inheritance Toolkit makes LabVIEW™ a true object-orientated language. The toolkit is easy to use and can be used by novice users and more advanced users alike. A GOOP 1.0 user should not have any problems learning how to use the new toolkit. Even for new users the new Toolkit provides improved debugging support.

4 GOOP Inheritance Toolkit

What’s in GOOP Inheritance Toolkit?

- GOOP 2.0 – a new GOOP Kernel implementation
 - All features of the “old” GOOP 1.0
 - Class attributes
 - Attributes shared by all objects of a class
 - Inheritance
 - Methods and attributes can be inherited publicly
 - Dynamic binding (polymorphism)
 - Methods can be redefined in a sub class by being declared *virtual* in the base class
 - Class description
 - A Possibility to add a description of the class and not only on each of the methods
 - New support method, *GetObjectStatus*, which returns information about the current object
 - Improved performance

¹ Polymorphism should not be confused with Polymorphic VIs in LabVIEW™. Polymorphism is usually implemented by a technique called *dynamic binding*.

- GOOP Wizard 3.0
 - Tool for creating and editing classes including automatic icon handling
 - Improved debugging tool, *Objects In Memory*
- Development Distribution 1.0
 - Tool for build distributions and executables based on GOOP 2.0
- GOOP 1.0 – the old “GOOP” Kernel implementation

These new features make LabVIEW™ *a true object-oriented language*.

The new GOOP 2.0 consists of two *class templates*², `_gw2bc.llb` (base class) and `_gw2sc.llb` (sub class) installed in the same folder as the Wizard. Every time a new class is created, one of these templates is copied. The class templates consist of method template VIs, private support VIs and a class unique repository VI called *GOOPKernel*. This unique class repository uses an *object repository*³, `_GOOP2.llb`, shared by all classes and is implemented in LabVIEW™ (no CIN, DLL or anything mysterious). This shared library must be installed in the `vi.lib\addons` folder, whenever an application using GOOP 2.0 runs. The `_GOOP2.llb` may be distributed freely.

Notice that the “old” GOOP 1.0 template, `_classtpt.llb`, is also distributed with the tool kit. The new GOOP Wizard 3 creates and edits classes based on the new GOOP 2.0 and also on GOOP 1.0 when necessary. A system may very well consist of new GOOP 2.0 classes mixed with GOOP 1.0 classes.

4.1 GOOP Wizard 3 and Objects In Memory tool

The new Wizard 3.0, shown in Figure 1, has a new built-in debugging tool, *Objects In Memory*, which shows all objects from all classes in one place. Figure 2 shows the front panel of the new tool. The tool can be seen as a “global” inspector. This way it is easy to get an overview of all objects that are created and destroyed and also to trace any memory leaks in the system.

² Compare with GOOP 1.0 template, `_classtpt.llb`.

³ The GOOP 1.0 uses the shared object repository, `_goopsub.llb` that is distributed with LabVIEW6 and is found in the `vi.lib\platform` folder and is therefore always installed for GOOP 1.0.

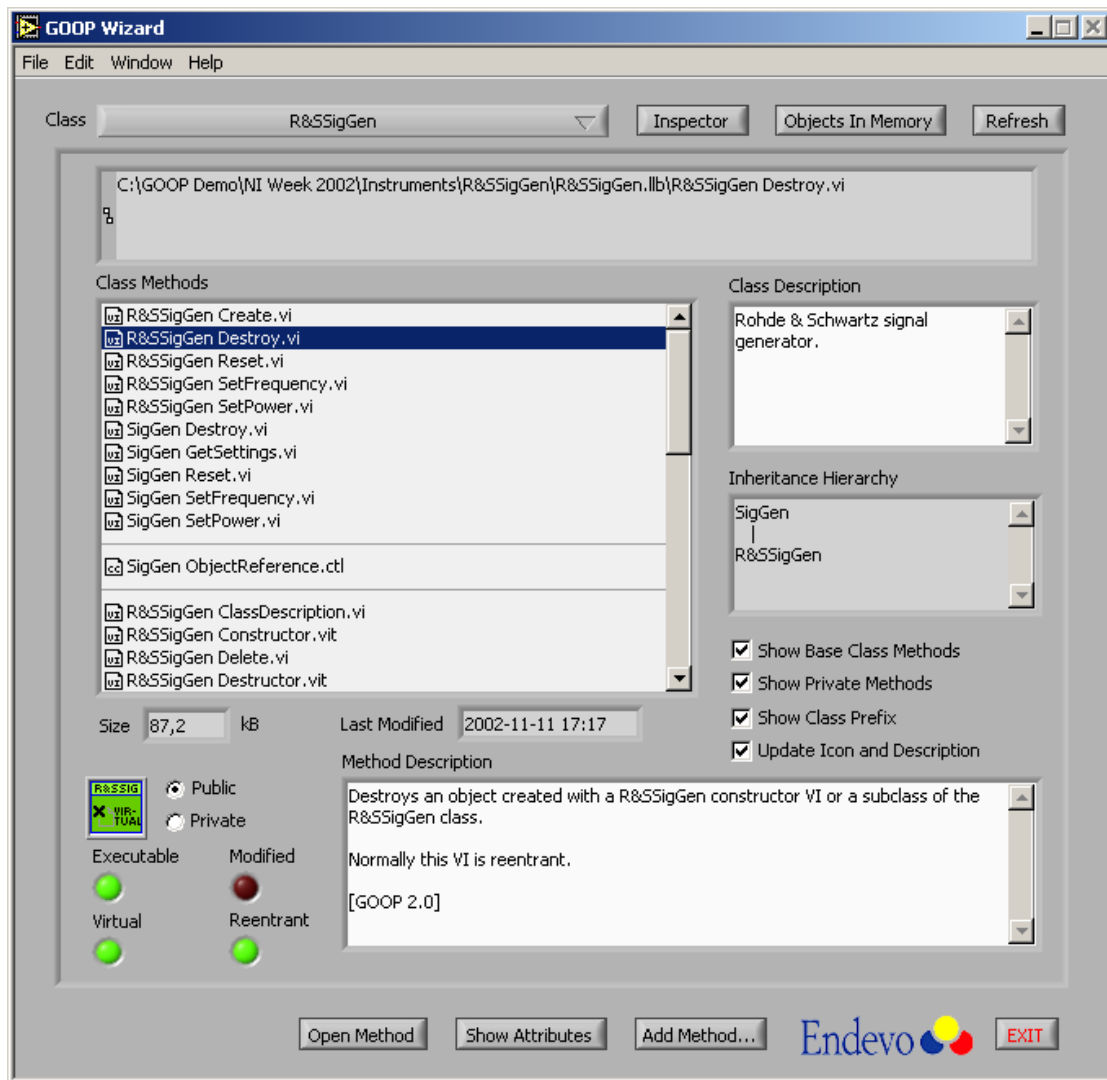


Figure 1. The front panel of the new GOOP Wizard 3.0. Notice the new “Class Description” and “Inheritance Hierarchy” to the right on the front panel. All public methods of the class *and its base class* are shown in the list. The example shows the *R&SSigGen* class that is inherited from the *SigGen* class.

The values of attributes are shown in a text format and it is also possible to kill single objects or all objects. If more detailed information is wanted, just *start the inspector for the specific class as usual*.

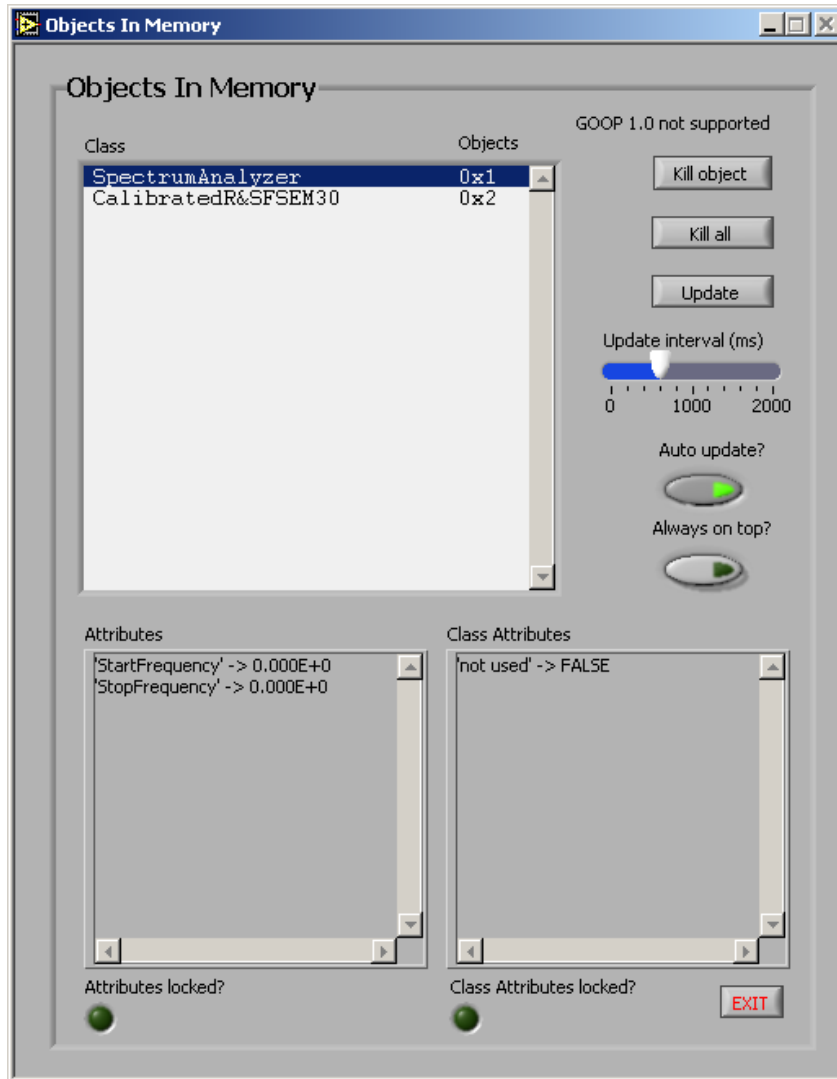


Figure 2. The new debugging tool, *Objects In Memory*, shows all objects in one place. This powerful support makes debugging easy. This is a kind of “global” inspector.

5 Object creation and destruction

In GOOP, objects are created dynamically by the constructor method, *Create* and deleted by the destructor method, *Destroy*. Figure 3 shows a very simple example of an object being created and destroyed.

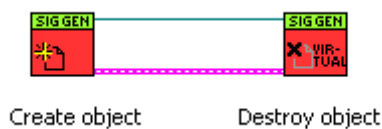


Figure 3. Very simple example where an object of the SigGen class is created and destroyed. This is performed exactly the same as in GOOP 1.0.

In GOOP 2.0 the object must be created first before the attributes may be set. This differs from GOOP 1.0, where the attributes were initialized by the private support method *New*. Notice that the number of objects instantiated of the class also is returned. Figure 4 illustrates this.

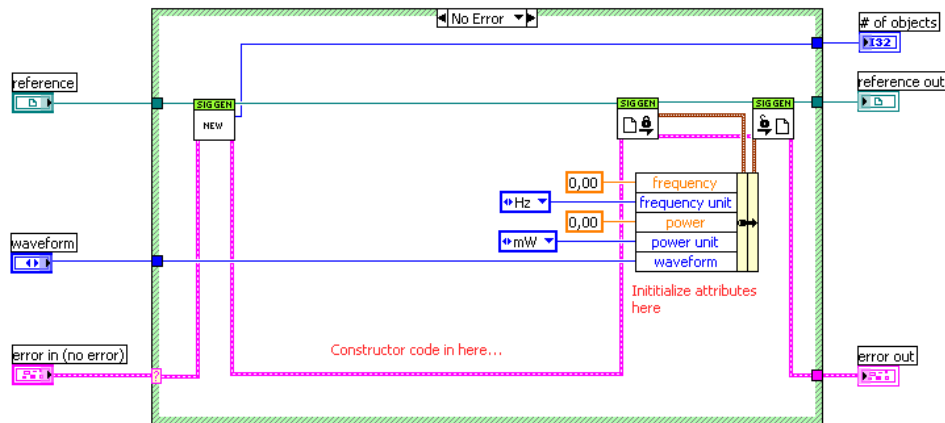


Figure 4. An example of the constructor method *Create* in the example class *SigGen*. Notice that the constructor has a reference input that must NOT be wired in a standard class. The reason why it exists and should not be wired is explained in chapter 8.3. The private support method *New* does not allow attributes to be initialized as in GOOP 1.0. This must be done by modifying the attributes afterwards.

The destructor method *Destroy* also looks different than in GOOP1.0. The private support method *Delete* does not return attributes, but these have to be read before executing the *Delete* method. The attribute must NOT be wired to an indicator in the *Destroy* method as in GOOP 1.0. The reason why is explained in detail in chapter 8.4.

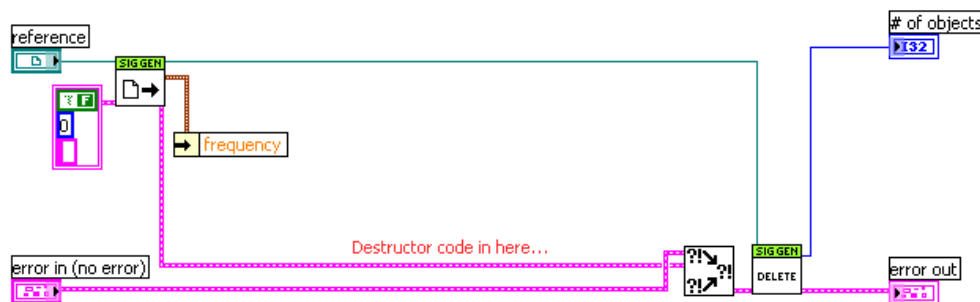


Figure 5. An example of a non-virtual destructor method, *Destroy*, in the example class *SigGen*. Notice that the support method *Delete* does not return attribute values. If attributes are needed one last time, use the *GetAttributes* method.

6 Attributes

The attributes of an object are accessed by the private support methods⁴, *GetAttributes*, *GetAttributesToModify* and *SetModifiedAttributes*.

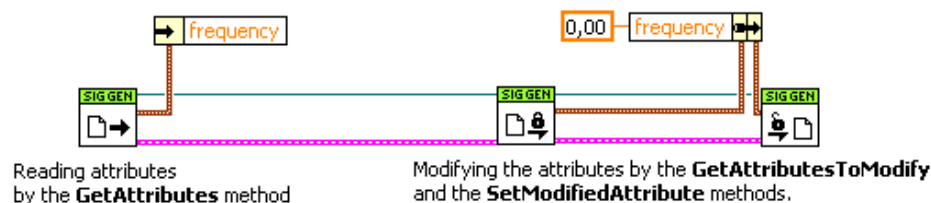


Figure 6. The attributes are accessed and modified by private support methods.

⁴ In GOOP 1.0 these are called *GetData*, *GetDataToModify* and *SetModifiedData*.

Before an attribute may be modified it must be locked using the *GetAttributesToModify* method. If the objects attributes are to be modified elsewhere in the execution, it will be forced to wait until the attributes are unlocked again by the *SetModifiedAttributes* method or a timeout occurs. The attributes are defined in the *ObjectAttribute.ctl* in the class.

7 Class attributes

The *class attributes*⁵ are attributes shared by all objects within a class. If one object updates an attribute, another object may read and modify them later. Using class attributes are as easy as using ordinary attributes. The attributes of an object are accessed by the private support method, *GetClassAttributes*, *GetClassAttributesToModify* and *SetModifiedClassAttributes*.

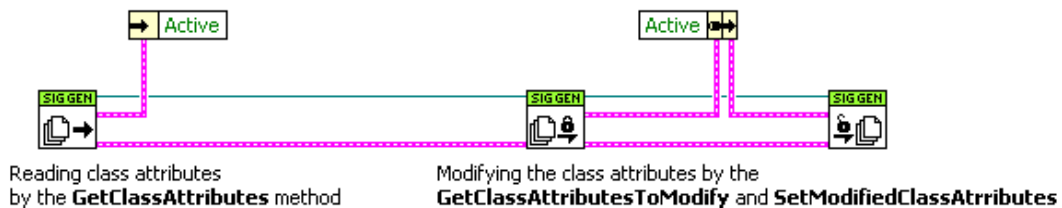


Figure 7. The class attributes are accessed and modified by private support methods.

Before an attribute may be modified it must be locked using the *GetClassAttributesToModify* method. If the class attributes are to be modified elsewhere in the execution, it will be forced to wait until the class attributes are unlocked again by the *SetModifiedClassAttributes* method or a timeout. The attributes are defined in the *ClassAttribute.ctl* in the class. It is also possible to create methods in the GOOP Wizard 3 that uses class attributes. These methods are called *Class methods*. When creating a method, select one of the *Class Method* options in the *Add Method* dialog. The initialization of class attributes requires a little bit more care than ordinary attributes. Normally the first instantiated object of the class initializes⁶ them.

Notice that it is possible not to wire the reference to the *GetClassAttributesToModify*, *SetModifiedClassAttributes* and *GetClassAttributes*. The class attributes of the class the methods belong to will be used as default. However, if the class method is inherited, there are things to consider. Read more in chapter 8.1.1

8 Inheritance

GOOP 2.0 supports *public inheritance*. Attributes and methods are inherited. Attributes may be seen as *protected*. This means that the attributes can only be used in subclasses, apart from the class in which they are declared, and seen as *protected* there as well. The *public* methods will be *public* in the subclass. Any class in GOOP 2.0 may be used as a base class and it is possible to inherit in as many generations as you like. Methods may also be declared *virtual* to be redefined in a subclass. If a virtual method from a base class is used with a sub class object, the sub class implementation will be used instead (if an implementation exists) and selected at runtime by dynamic binding. Notice that multiple inheritance (from two or more base classes) is not supported.

⁵ Compare with static attributes in Java and C++.

⁶ See chapter 5. Numbers of instantiated objects are returned when a new object is created. This way it is easy to figure out if this is the first object instantiated.

The new GOOP Wizard 3.0 makes it easy to create a sub class base on an existing base class. It is as simple as creating an ordinary class, all you have to do is to select which class to use as a base class and let the Wizard do the work. If you make any changes later in the base class this will affect all sub classes based on it, so it is a true inheritance. On the Wizard 3.0 front panel, there is an indicator showing if a method is declared virtual or not. Figure 1 shows the front panel on the Wizard 3.0 where the indicator is found in the bottom left. Notice that it is not possible to inherit from a class created in GOOP 1.0. All classes used as base classes must be created in GOOP 2.0.

8.1 Attributes

Both attributes and class attributes are inherited from the base class. The attributes of a class are defined in the *ObjectAttributes.ctl* and the class attributes are defined in the *ClassAttributes.ctl*. Figure 8 shows an example of a class and the attributes defined for that class.

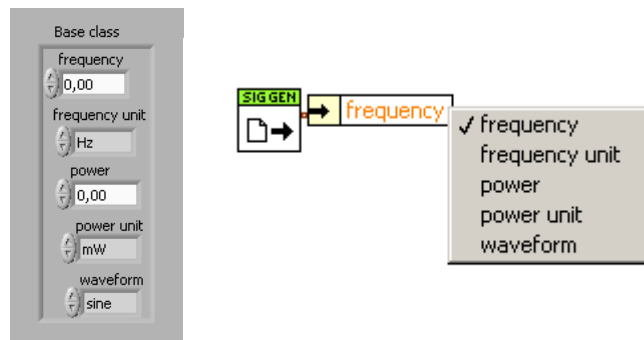


Figure 8. The attributes of the example class SigGen describing a standard signal generator. To the left the SigGen_ObjectAttribute.ctl that defines the attributes and to the right the private support method SigGen_GetAttributes.vi that is used for reading the attributes.

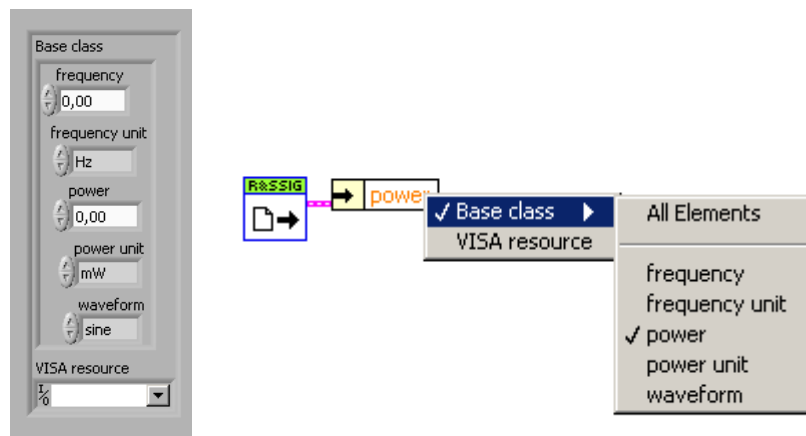


Figure 9. The attributes of the example class R&SSigGen that is inherited from the SigGen class. Notice that the SigGen_ObjectAttributes.ctl is placed inside the R&SsigGen_ObjectAttributes.ctl as an element called *Base class*. To the right, the private support method R&S_GetAttributes is used for reading the attributes. Notice how it is possible to access both attributes from the base class and from the subclass.

When a sub class is created, the base class *ObjectAttributes.ctl* will be a part of the sub class *ObjectAttributes.ctl* as an element called *Base class*. More attributes can of

course be added in the sub class *ObjectAttributes.ctl* to defined additional attributes to the attributes inherited from the base class. Classes can be inherited in as many generations as needed. Figure 10 shows an example of a class that has been inherited in two “generations”.

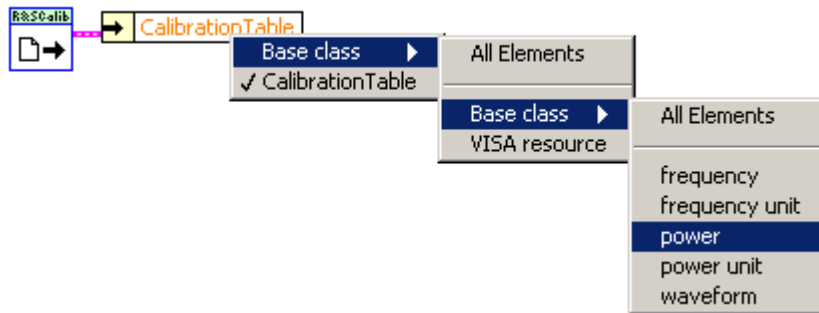


Figure 10. An example of a sub class *R&SSigGenCalibrated*, which is inherited from the *R&SSigGen* class that has been inherited from the root base class *SigGen*. For the *Unbundle By Name* to be more readable, it is recommended to activate the property *Hide Full Names*.

Methods from a base class use the private support methods *GetAttributes*, *GetAttributesToModify* and *SetModifiedAttributes* as described in chapter 6. If a base class method locks the attributes for modifying this will of course force any other method within that class to wait for the attribute to get unlocked, but also all sub class methods that tries to modify the attributes will also be forced to wait. This way base class methods and sub class methods work together while accessing attributes.

Notice that the base class *ObjectAttributes.ctl* **must be called *Base class*** when placed into a sub class *ObjectAttributes.ctl*. This is nothing that has to be performed manually; the GOOP Wizard 3.0 handles all of this when creating a sub class based on a base class.

8.1.1 Class attributes

Class attributes⁷ are inherited the exact same way as the object attributes described in the previous chapters. The *ClassAttributes.ctl* will behave the exact same way as the *ObjectAttributes.ctl*. Notice that the class attributes of a sub class is not the same as the base class, an object of a sub class does not share the class attributes with an object of the base class. However, if the reference to the *GetClassAttributesToModify*, *SetModifiedClassAttributes* and *GetClassAttributes* is not wired in a base class method, that is inherited, the class attributes of the base class will be used by default and not the sub class. This might not be the intention, but if the reference is wired⁸, the sub class' class attributes will be used instead. Therefore it is strongly recommended to wire the reference.

8.2 Methods

As mentioned in the previous chapters, methods can be inherited from a base class. To be able to use a method from a base class in a sub class, the *ObjectReference.ctl* must be the same for all methods that that are to be used together or else there will be a broken wire (just as when connecting two methods from different classes mistakenly). Letting the sub class use the same *ObjectReference.ctl* as the base class solves this

⁷ In old versions of the GOOP course material, the “normal” object attributes are denoted “Class attribute” mistakenly.

⁸ The reference is only used to determine the class of the object the reference is associated to.

problem. In fact all sub classes in a class hierarchy use the same *ObjectReference.ctl* as the root base class.

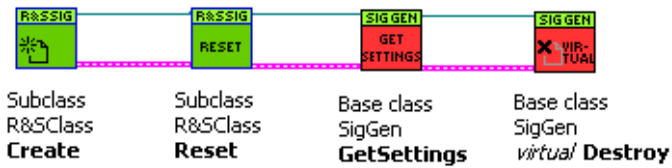


Figure 11. Methods from a base class can be connected together by letting the subclass use the same *ObjectReference.ctl* as the base class.

Is this a correct object-oriented way of treating references? Yes, it is. For example, the R&SSigGen is a sub class implementation of a specific R&S signal generator and is inherited from the SigGen class that is a generic signal generator class. The R&SSigGen class will then use the SigGen class *ObjectReference.ctl*. This is the correct way according to object orientation because the R&S signal generator *is a* signal generator. A sub class is always a more specific⁹ class than its base class. Figure 11 shows an example where the sub class R&SSigGen (that is inherited from the SigGen class) uses the sub class method “Reset” together with the inherited base class method “GetSettings”.

8.2.1 Virtual methods

In GOOP 2.0 methods may be declared *virtual* when they are created. This way it is possible to redefine base class methods in a sub class.

A reference, instantiated of a subclass, is passed to the *virtual declared base class method*. The method checks the reference and tests if there is an existing sub class implementation of the method and if there is, it uses this implementation instead. A very simplified way of describing this would be if a VI has one front panel and a set of different diagrams to choose from (one diagram for each sub class).

Look at Figure 12, at the top an object of the R&SSigGen class (inherited from the SigGen class) is created, the reference is passed to the virtual method *SetPower* belonging to the base class. The base class method checks the reference and finds out that it belongs to a sub class object and tries to locate an implementation of the method in the sub class. If an implementation exists, it will use this instead. In this example it exists so it will be used. The same thing happens with the virtual destructor *Destroy*.

If the method is not found in the sub class, a **search through the class hierarchy will be performed**. First the sub class will be searched, then its ancestor and upwards through the entire class hierarchy until it reaches the root base class. If no implementation is found *the default base class implementation of the calling virtual method will be used*.

⁹ Compare with C++ where a pointer to a sub class always may be type casted to a base class pointer (but not the other way around). A sub class always contains more (or equal) data as its base class. It is logical that something larger may be casted to something smaller.

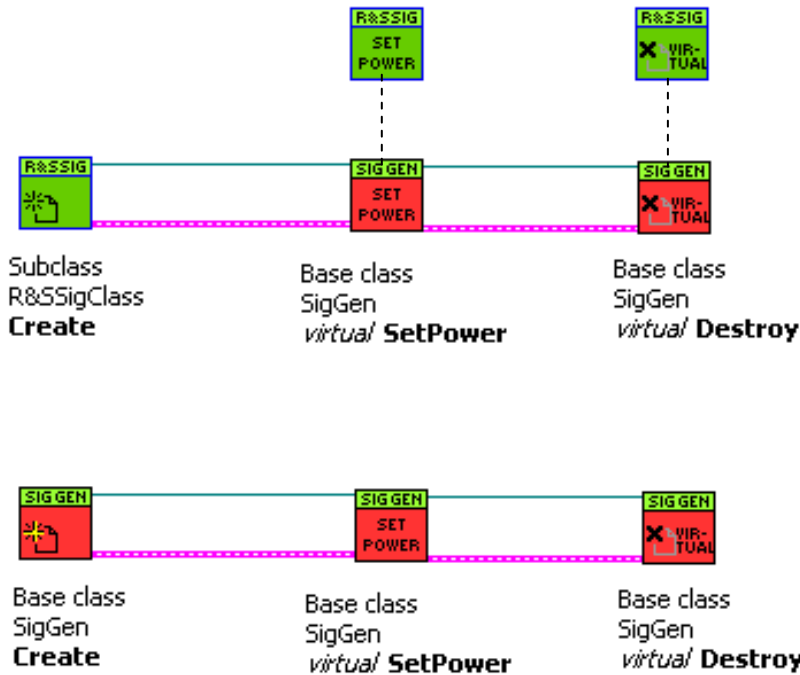


Figure 12. An example of the use of virtual declared methods. At the top an object of the R&SSigGen class is created. The reference is passed to the virtual declared base class method SetPower and through to the virtual destructor Destroy. At the bottom an example where an object of the base class SigGen is created instead.

At the bottom of the figure, there is another object created from the base class SigGen, the virtual SetPower method will check the reference and find out that it belongs to the base class (the same class as itself) and will not try to find any sub class implementation, but use the default base class implementation instead..

How is all this possible? Virtual methods are implemented by using a *strictly typed VI server*. Each virtual method has one extra case statement with two cases, one case called "Class implementation" which is the base class default implementation and one case called "Call dynamic method". Figure 13 and Figure 14 shows an example of a virtual method with the two cases. A private support method called *FindDynamicMethod* always executes first and checks the reference and if it belongs to a sub class tries to find a sub class implementation of the method. If it exists, the *FindDynamicMethod* selects the "Call dynamic method" case and calls the sub class implementation by the *Call By Reference* node. If not found, it selects the "Class implementation" case and executes the class implementation code there instead.

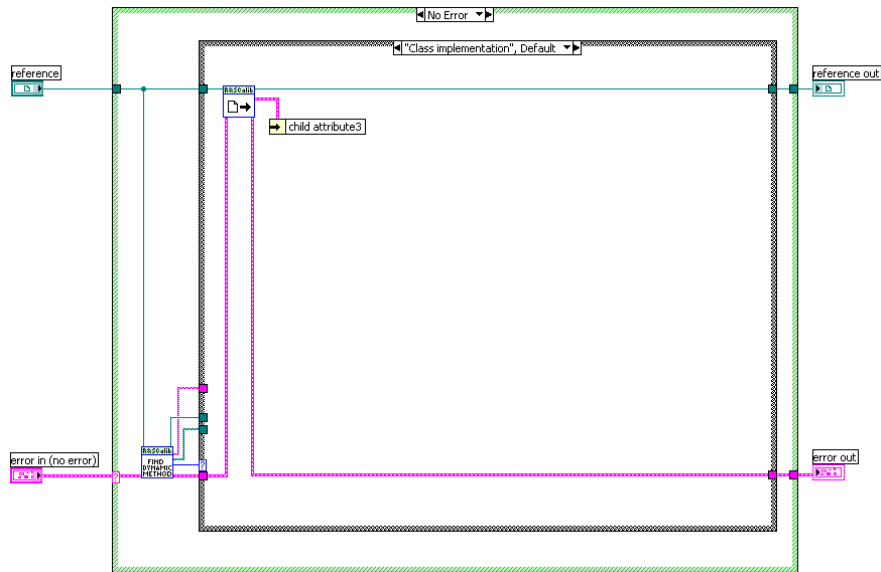


Figure 13. A virtual declared method will use a different template than a “standard” method. This example shows the “Class implementation” case. Inside this case it looks like a regular GOOP method.

To declare a method virtual when creating it, use the new option *Virtual* in the *Add Method* dialog in GOOP Wizard 3.0. A different method template will then be used for the new virtual template. Figure 13 and Figure 14 shows one of these templates.

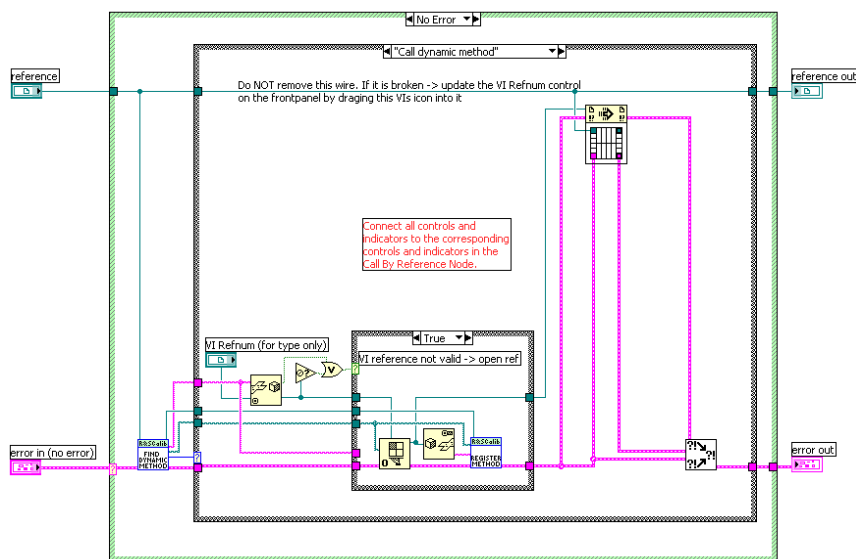


Figure 14. The same virtual method as in Figure 13, but with the “Call dynamic method” case selected. This case looks the same for all virtual methods except for the controls and indicators that must be wired to the Call by reference node.

For a strictly typed VI server to work properly, the connector pane of the VI that is to be called and executed must be known. **Forcing the sub class implementation of a method to have exactly the same connector pane as the virtual base class method solves this¹⁰**. The connector pane pattern, controls and indicators must be identical.

¹⁰ Compare with C++ where all virtual methods and redefined sub class methods must have the exact same input and output parameters.

Even controls that are required in the base class method must be required in a sub class implementation. **The name of the method must also be identical.** Figure 15 shows an example of a base class method and a sub class implementation of the same method.

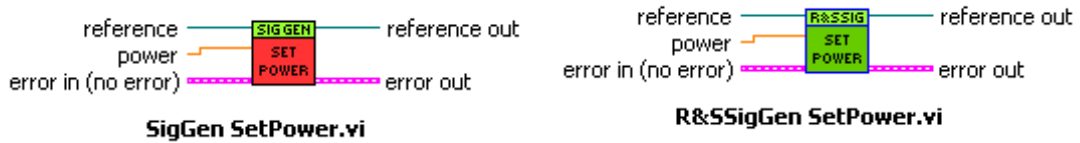


Figure 15. The base class virtual method SetPower is shown to the left and the sub class implementation of the same method to the right. The connector pane (pattern, controls and indicators) must be exactly the same. Even controls that are required must also be required in a sub class method. The name of the method must also be identical.

On the front panel of a virtual method there is a strictly typed VI Refnum control that is NOT wired to the connector pane. After all controls and indicators has been placed on the front panel and wired to the connector pane, the VI Refnum control has to be updated. This VI reference control selects the connector pane on the Call By Reference Node and this must be set to match the connector panes of the methods to be called. That is easy because one of the rules was that all dynamically called methods must have the same connector pane as the calling virtual method. In other words the *VI Server Class*¹¹ of the reference is always set to the same as the method VI it belongs to!

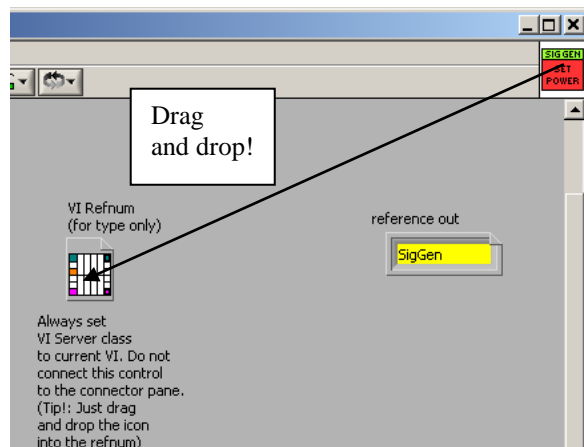


Figure 16. Each method declared virtual must update the VI refnum control on the front panel to the same VI Server Class as the virtual method itself. This is easiest performed by just dragging and dropping the icon of the same method onto the VI ref control.

The easiest way to perform this is by just dragging and dropping the icon of the method onto the VI Refnum control!

¹¹ The VI Server Class only selects the connector pane layout from a template VI. Right clicking on the reference control on the front panel and select “Browse” to a select the template VI performs this. However, a better way is to just drag and drop the current VIs icon into the VI Refnum control and it will adapt to the current VI connector pane.

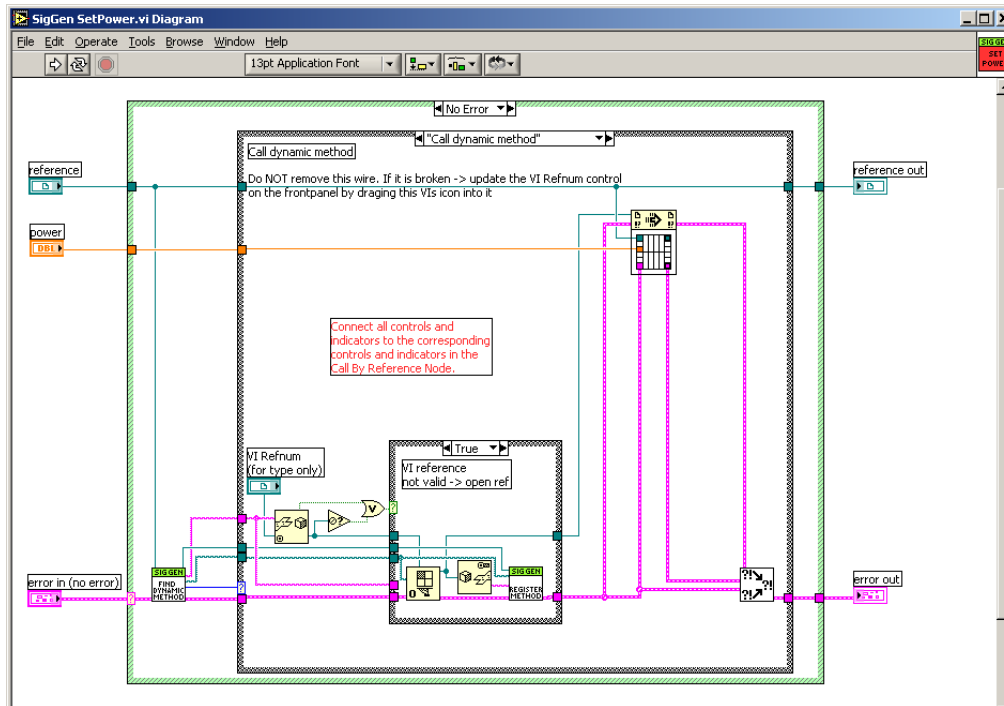


Figure 17. When all controls and indicator of the virtual method have been added and connected to the connector pane, the VI Refnum control must be updated as described in Figure 16. The Call By Reference Node will then adapt to exactly the same connector pane as the virtual method. All controls and indicators must then be connected to the Call By Reference Node.

In the “Call dynamic method” case, the Call By Reference Node will now adapt to the exact same connector pane as the virtual method itself. Now connect all controls and indicators to the corresponding input and output on the Call By Reference Node, this way the parameter values are passed to and from the sub class implementation.

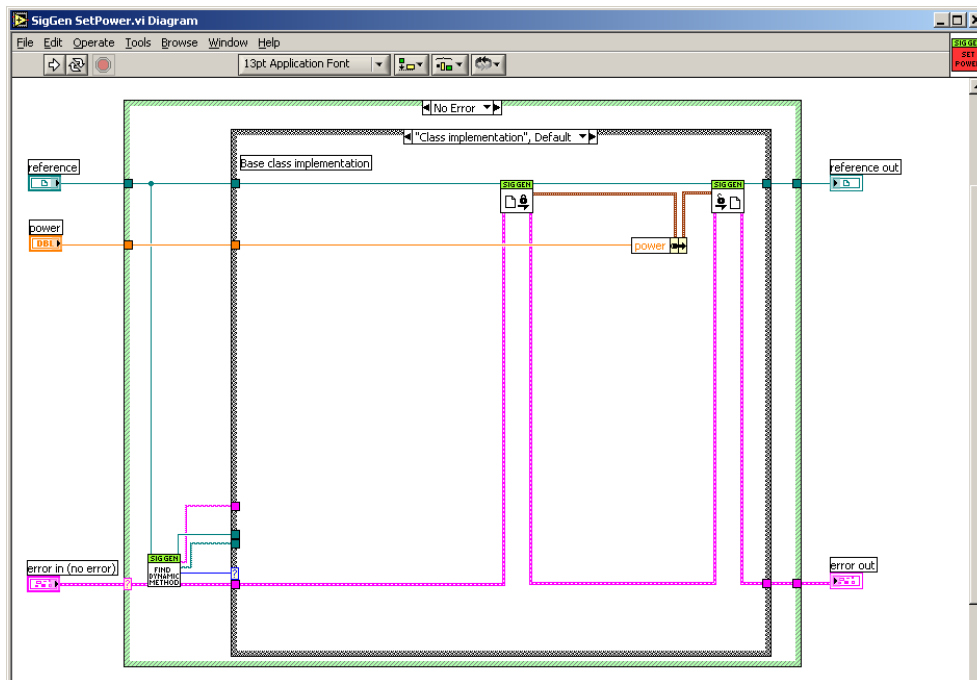


Figure 18. The “Class implementation” case of the same virtual method as in Figure 17. This case contains the base class implementation of the method. In this simple example, the attribute “power” is updated with a new value.

Notice that a method in a sub class that is called dynamically may or may not also be virtual. Normally only the base class method is virtual and all sub class implementations of the method is non-virtual¹². non-virtual method is always faster to execute because a virtual method always has to perform a check¹³ by the private support *FindDynamicMethod* to try to locate a subclass implementation of the method. If the subclass method also is declared virtual, the check in this method will always result in using the class implementation (why call it otherwise?) and that is why it is normally no use in declaring this method virtual as well.

When a virtual method has found a sub class implementation, a VI reference will be opened to that method and registered in the GOOP 2 kernel. The next time this method is called, the reference will be reused (this improves performance). The VI references will be closed by the kernel when the object is deleted, so there is no risk of “leaking” VI references. The sub class methods that are redefined in a subclass and called dynamically by a *virtual method must NOT be a LabVIEW™ polymorphic VI*. Strictly typed VI server cannot call polymorphic LabVIEW™ VIs¹⁴.

8.2.2 Base class implementation in a sub class method

When redefining a method in a sub class that is declared virtual in a base class, you might not want to rewrite the code for the whole method. Actually you only want to add some additional code, but still execute the base class implementation as well.

For example, assume that the virtual SetPower method in the SigGen class, shown in Figure 17 and Figure 18, is redefined in the sub class R&SSigGen as shown in Figure 15. Let's say that we want to implement some sub class specific code but in addition use the implementation in the base class without having to copy and paste this code into the sub class. This is performed by **using the base class method VI as a sub VI in the subclass implementation of the method**¹⁵. Figure 19 shows an example what this may look like.

¹² Non-virtual methods may sometimes be referred as *leaf* methods.

¹³ This happens in all object-oriented languages where virtual methods are used. The compilers have to create tables of possible dynamical method calls and each this method is called a check in the table has to be performed to find the correct implementation. This always takes time, even if smart caching algorithms are used to improve performance.

¹⁴ This will cause the LabVIEW™ error code 1035 (Operation is invalid for this type of VI).

¹⁵ In C++ this would be the same as typing *SigGen::SetPower* in the sub class implementation of the SetPower method to enforce using the base class implementation.

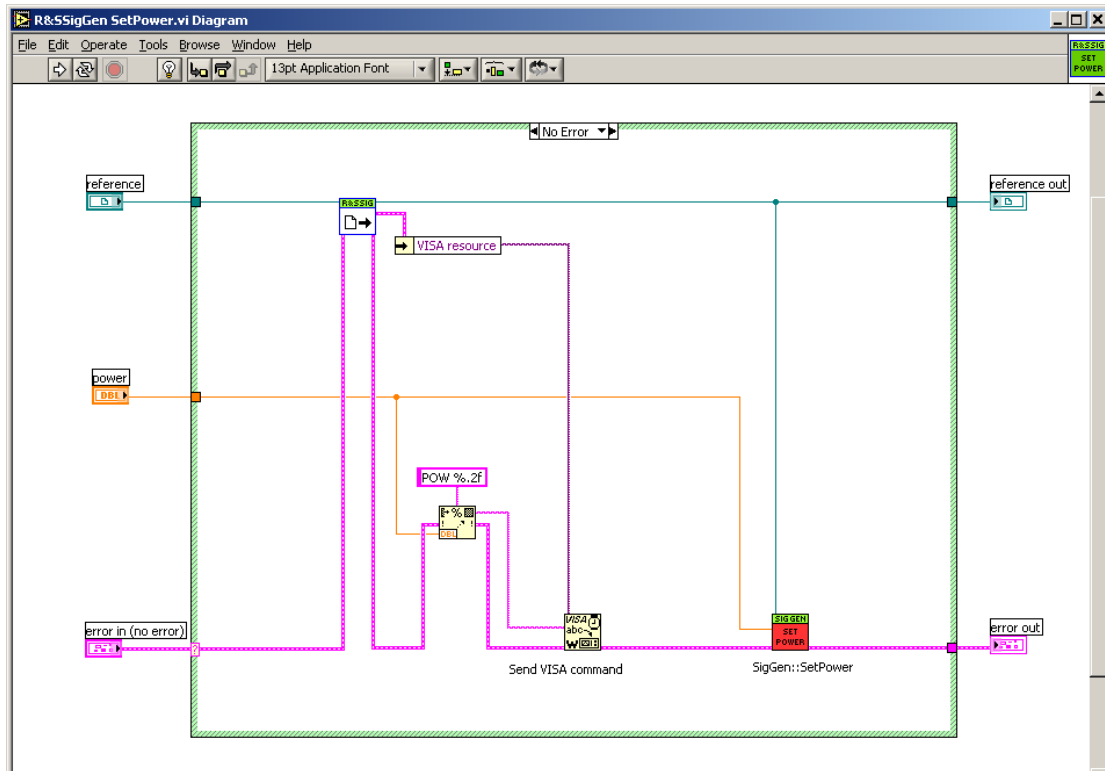


Figure 19. The implementation of the SetPower method in the sub class R&SSigGen. This will be dynamically called by the virtual base class method SetPower shown in Figure 17 and Figure 18. Notice that the base class method is used as a sub VI to use the base class implementation of the method.

Now, assume that the code at the top in Figure 12 is executed. Will this really work? If we look at the call chain for the virtual method SetPower, it will look like SigGen SetPower.vi -> R&SSigGen SetPower.vi -> SigGen SetPower.vi. The base class method will actually have itself in the call chain! Is this allowed? Yes, it is. The private support method *FindDynamicMethod* will check if the virtual method has itself in the call chain and thereby enforce using the “Class implementation” case of the method the second time it executes and this is when it is used in the sub class implementation. This way it will work and not get stuck in an endless “recursive” loop¹⁶.

However, there is one condition, the **virtual base class implementation of the method must be reentrant if it going to be used in a sub class implementation**. If not, the virtual base class method is actually still executing when it calls the sub class implementation of the method (the Call By Reference Node will not be finished until the sub class method is finished). When trying to execute the base class implementation in the sub class, this method must be reentrant; otherwise it will be a deadlock situation, since it is actually waiting for itself to finish, which will never happen if it is not reentrant!

¹⁶ This recursive loop limitation will actually remove the possibility to aggregate objects of the same class and perform recursive operations on them. For example the *Composite* pattern in ref [5] will not work.

8.3 Constructor – the Create method

In GOOP2 the *constructor* method, *Create*, differs from GOOP 1.0¹⁷. This is because it must work with inheritance. A standard class constructor method may look like the example in Figure 4. Notice that the constructor has a reference input that must not be wired in a standard class. Why is this reference needed and why does it exist? The reason for this is that it is needed when creating a sub class from the class. Look at the example in Figure 20, this shows the constructor method, *Create*, of the subclass R&SSigGen that is inherited from the SigGen class.

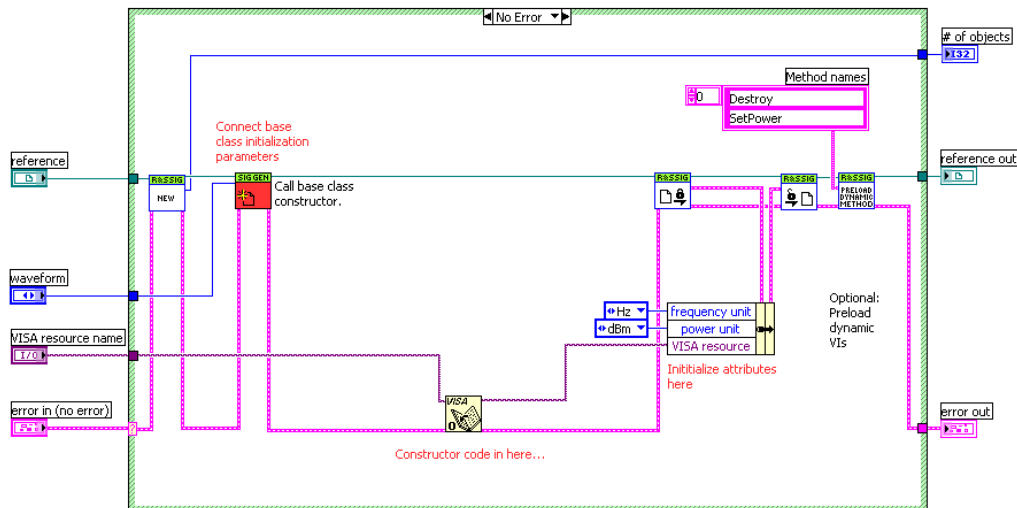


Figure 20. The constructor method, *Create*, of the sub class R&SSigGen of the base class SigGen. Compare with Figure 4. Notice two differences, the base class constructor is used as a sub VI and there is a new private support method called *PreloadDynamicMethods* executing at the end.

Notice that the base class constructor, *Create*, is used as a sub VI and that the reference is passed to the VI. When an object is created by the private support method *New*, the base class constructor is called and the code in this VI is executed. This way all ancestors (all the way to the root base class) constructors will be called *before*¹⁸ the actual constructor code is executed in the sub class. If a base class constructor initializes an attribute, it will be possible to reinitialize this in a sub class (if wanted).

By executing the *New* method (and thereby creating an object) and passing the reference to the base class constructor, the *New* method in the base class constructor will check the incoming reference and notice that it is a valid reference and understand that it is executing as a sub VI to a subclass constructor and thereby NOT create a new object.

All input parameters that are required by the base class constructor must be wired to the sub class constructors' front panel. In the example in Figure 20, the “waveform” parameter is required by the constructor and is therefore wired to the front panel.

¹⁷ In GOOP 1.0 the constructor name may be selected arbitrary (like *Open*), but in GOOP 2.0 the constructor must be called *Create*.

¹⁸ Compare with C++ where all code in the ancestors constructors are executed before the subclass constructor.

A constructor may never be declared virtual. This is logical, because by executing the Create method of a class, you actually decide which sub class to use (how should this otherwise be known?). All the constructor methods in the ancestor base classes will be executed anyway (they are sub VIs to the sub class constructor). The constructor method of a sub class, *Create*, is actually the only VI that has to be part of an application if all method VIs that are used, is declared virtual in the base class and only these virtual base class method VIs are used in the application¹⁹. Why declare the base class methods virtual if the sub class method VIs are used in the application?

In the constructor method of a sub class there is also the possibility to *preload dynamic methods implemented in the class*. Normally when only using virtual base class methods in the application, the sub class implementation of the methods is not in memory then the applications starts executing, but are called and dynamically loaded the *first time* the virtual base class method executes.

However, if the VI is large, the file access speed is low or high performance is required. **A good rule is to preload all dynamic VIs implemented in the sub class.** Using the optional private support method *PreloadDynamicMethods* performs this. This method executes at the end of the constructor and loads the listed dynamic methods to memory. Notice that this is optional and not required for the dynamic binding to work, it only increases the performance the first time a virtual method is called. If the method in the list is not found, it will cause an error, avoiding misspelling of the name. Therefore only enter the redefined methods that are implemented. If the method list is left unwired or is empty, all dynamic methods of a class will be preloaded.

8.4 Destructor – the Destroy method

In GOOP2 the *destructor* method, *Destroy*, differs from GOOP 1.0²⁰. This is because it must work with inheritance. A standard class destructor method may look like the example in Figure 5, but more common is that the **destructor is declared virtual**. If so, the base class virtual destructor method, *Destroy*, may *delete any object belonging to a base class or any sub class*. This is very useful and recommended. Figure 21 shows an example of a virtual base class destructor method. If the destructor is not virtual the correct destructor must be used to kill the object.

¹⁹ Only using *virtual* base class methods (except for the constructor) are a good example of *generic* code.

²⁰ In GOOP 1.0 the destructor name may be selected arbitrary (like *Close*), but in GOOP 2.0 the destructor must be called *Destroy*.

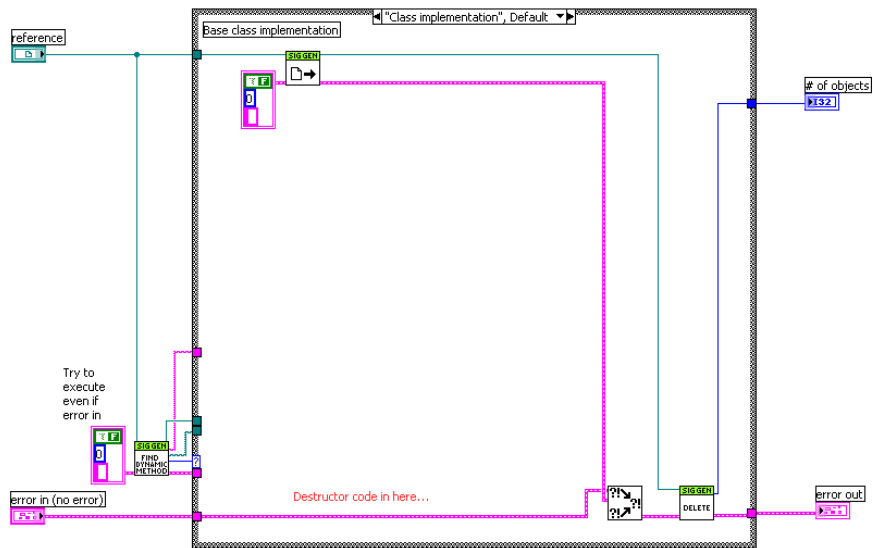


Figure 21. An example of a virtual destructor method, *Destroy*, in the root base class *SigGen*. It looks like an ordinary virtual method, but it does not have any error case. This is because it must execute even if an error in occurs. Notice that the attributes have to be accessed before the object is deleted by the private support method, *Delete*.

Two major things differ from a GOOP 1.0 destructor. First, the *Delete* support method is executed last in the destructor. It actually executes the destructor code before if deletes the object. Therefore the attributes must be read using the private support method to access the attributes if needed.

Second, the attributes are not returned via an indicator out from the *Destroy* method. The reason for this is that if the destructor is declared virtual, all controls and indicators must be identical in all sub class destructors as well. However the attributes in the *ObjectAttributes.ctl* is not the same in the base class as in the sub class²¹. Therefore, the rule of identical connector panes for virtual methods is violated and therefore the attributes must not be passed out via an indicator²². **A destructor must not have any additional controls and indicators.**

Figure 22 shows an example of a sub class destructor. In this example, it is declared virtual, but a sub class destructor may very well be non-virtual as well. A non-virtual destructor would only have the code inside the “Class implementation” case.

Notice that the sub class destructor uses the base class destructor as a sub-VI, called just before the private support method *Delete*. This way all ancestors (all the way up to the root base class) destructors will be called *after*²³ the actual destructor code is executed in the sub class. Compare with the constructor where the base class constructor is called before the sub class constructor code executes.

²¹ See chapter 8.1 where attribute inheritance is discussed.

²² Compare with C++ where the destructor must not have any input and output parameters.

²³ Compare with C++ where all code in the ancestors destructors are executed *after* the subclass destructor.

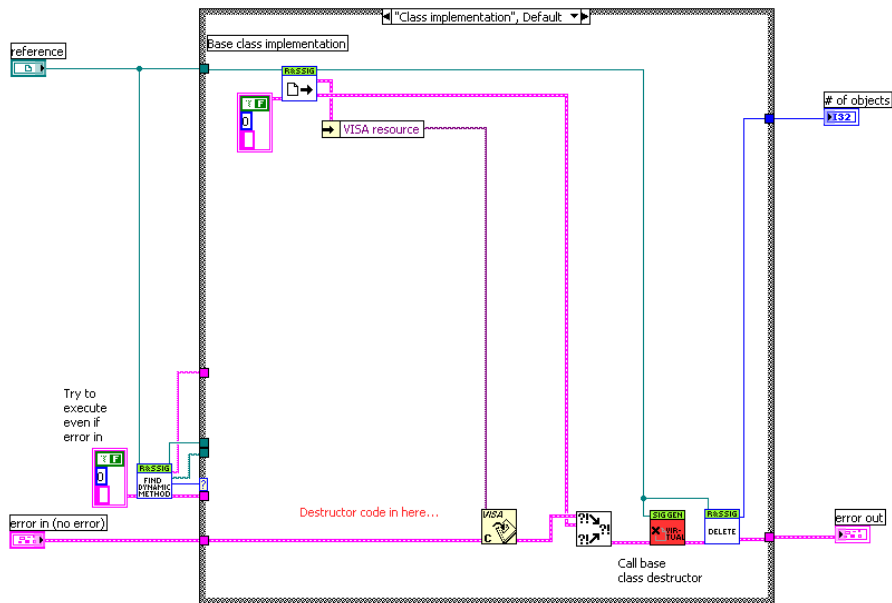


Figure 22. An example of a virtual destructor method, Destroy, in the sub class R&SSigGen inherited from SigGen. This method may very well be non-virtual, but would look like the code inside the “Class implementation” case. Notice that the base class destructor is called before the Delete method. This always happens in a sub class destructor (vital or non-virtual).

When the *Delete* method is executed in a destructor, it will check if the reference belongs to the class, if not (reference must then belong to a sub class), the object will be deleted or else the delete request is ignore (the sub class will delete the object).

If a destructor is virtual, all sub classes of the class will use the base class destructor as a sub-VI as explained above. This is actually the same situation as described in chapter 8.2.2. Therefore a **virtual destructor must always be reentrant**. This is automatically set by the GOOP Wizard and is nothing that has to be performed manually.

8.5 Abstract classes

The GOOP 2.0 does *not support abstract classes*. An abstract class is a class that has to be sub classed and may not be instantiated itself. All virtual methods in the abstract class must be redefined in a sub class.

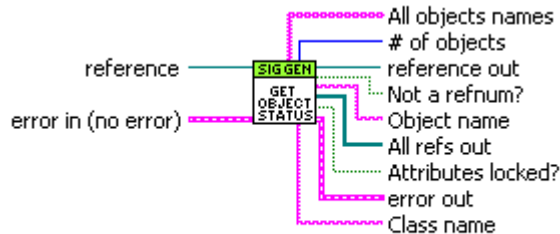
However, there is a trick to enforce all virtual declared methods to be implemented in a sub class. In the “Class implementation” case of all virtual methods (except for the destructor) simply generate an error²⁴. This way an error occurs each time a base class method is executed that does not have a sub class implementation. To ensure that an object is not instantiated from the abstract base class, the new private support method *GetObjectStatus*²⁵ may be used in the abstract base class constructor to check that the reference inside the constructor does not belong to the abstract base class itself and if so generate an error.

²⁴ In C++ these methods are called *abstract methods* in an abstract class.

²⁵ See chapter 9. It is possible to get the class name of the sub class the reference belongs to.

9 Object status

In GOOP 2.0 there is a new support method called *GetObjectStatus*. It returns the status²⁶ of the object. Figure 23 shows an example of the *GetObjectStatus* support method. It looks the same for all classes (base classes and sub classes).



SigGen GetObjectStatus.vi

Figure 23. An example of the private support method *GetObjectStatus*. It returns status about the object and also about other objects in the class.

The method may be used to test if a reference is valid²⁷ and also retrieve the class name of the reference. If the *GetObjectStatus* is used in a base class method that is inherited, it retrieves the name of the sub class the reference belongs to. This is useful when creating abstract classes as described in chapter 8.5. Be careful, because if a distribution is made as described in chapter 12.1 and prefixing is used, this will actually change the class name to *include* the prefix.

This method may also be used to retrieve information about other objects in the class. All names of the objects are listed and their corresponding object references. The number of objects instantiated of the class is returned as well.

If needed, the *GetObjectStatus* may be used as a public method. This is easy to change in the GOOP Wizard.

10 Inspector

For each class there is a class unique debugging tool, the *Inspector*. The inspector exists for GOOP 1.0 also but looks a little bit different in GOOP 2.0. Figure 24 shows the Inspector. All objects instantiated of the class are shown in the object list. The selected objects' attribute values are shown on the front panel. The values of the class attributes are also shown on the front panel to the right.

If the *ObjectAttributes.ctl* and *ClassAttributes.ctl* are large, they will probably “mess” up the Inspector. This is easily fixed; simply open the inspector in the GOOP Wizard as a regular method²⁸ (this way it will not auto start) and place the *ObjectAttributes.ctl* and *ClassAttributes.ctl* wherever you want. The inspector VI diagram is not locked, so you may perform any change needed.

²⁶ Compare with the status operation defined for the native LabVIEW™ Notification, Semaphore, Rendezvous and Queue.

²⁷ If the reference in is not valid, there will NOT be an error. Only the “Not a refnum?” will be true.

²⁸ The inspector is unique for each class and is stored as a private support method.

Due to performance aspects in inheritance implementation some features that existed in GOOP 1.0 is no longer supported in the GOOP 2.0 inspector. The statistics are not present and the “debug with single object” is not supported.

At the front panel of the inspector, there is an *object reference indicator*. The value²⁹ of this is depending on the selected object in the list. By right clicking on the object reference and selecting “Copy data” you actually copy the reference value into the clipboard. Now, by opening the front panel on any method of the class, including inherited method, you may paste this reference value (pointing out the selected object) by right-clicking on the input reference and selecting Data operations->Paste data. Execute the method with the run button in the tool bar. This way, methods may be debugged³⁰ as regular VIs.

Notice if a method that is being debugged, as mentioned above, is a *reentrant* virtual method that will call a sub class implementation, and this implementation in turn uses the *same* base class method, that is being debugged as a sub-VI³¹, this will cause an error code 1003 (VI is not executable)³². However, this situation is avoided by using the sub class method directly for debugging instead of the virtual base class method.

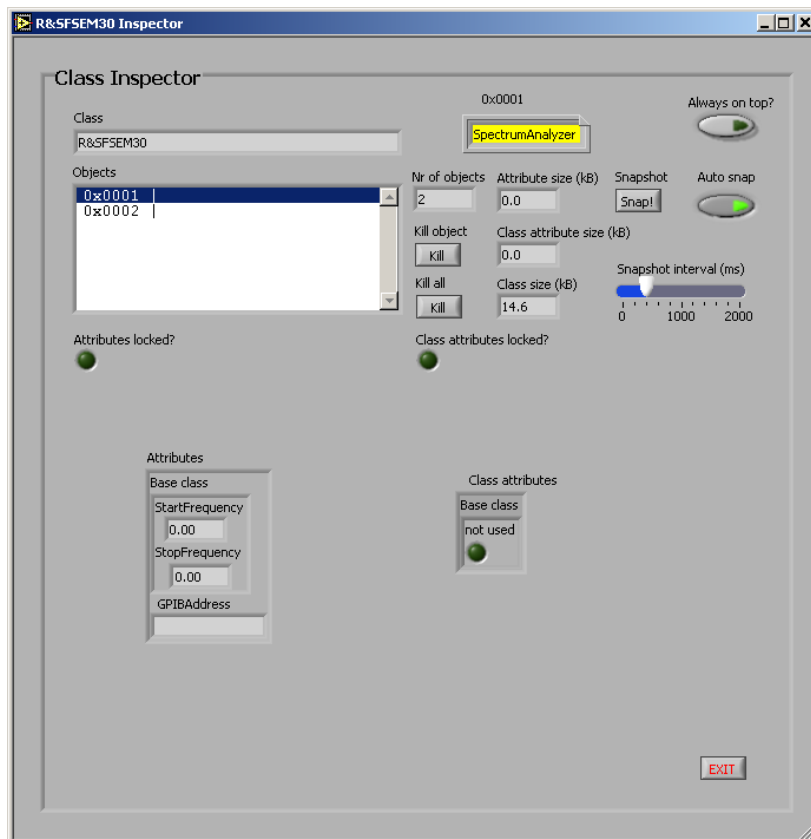


Figure 24. An example of the class inspector. When debugging many classes at the same time, the *Objects In Memory* tool gives a much better overview, where all objects from all classes are shown in one panel. See Figure 2.

²⁹ An object reference actually holds a value even if it does not look like it on the front panel. It is a “hidden” I32 value. The “hidden” value is shown above the reference as hex number in the Inspector.

³⁰ This debugging feature is supported in GOOP 1.0 as well.

³¹ See chapter 8.2.2 Base class implementation in a sub class implementation.

³² LabVIEW™ will actually consider a reentrant VI that runs as a top-level application non-reentrant.

11 Error handling

In GOOP 2.0, errors occur if classes are used incorrectly. The errors that may occur are found in Table 1. The codes that have been used are LabVIEW™ standard error codes (mostly code 1, An input parameter is invalid). This way, GOOP 2.0 does not conflict with any user written error codes.

<i>Code</i>	<i>Description</i>	<i>Cause</i>
1	Object reference is invalid	An invalid reference has been passed to a method. Usually this occurs if a sub class method is used incorrectly together with another subclass, inherited from the <i>same</i> base class. The two sub classes are from two different branches in the class hierarchy and will not work together. See Figure 25.
1	Object already exists	Trying to create two objects within the same class with identical names.
1	Object not found	Trying to look up an existing object that does not exist.
1	Reference in is invalid	An incorrect use of the input reference on a constructor.
1	Attributes not locked	Trying to set attributes when they are not locked.
7	Dynamic method not found	Trying to preload a virtual method in a sub class constructor that is not implemented in the same sub class.
1003	VI is not executable	A sub class implementation of a virtual method is not executable when called dynamically (the application may be executable, but dynamic methods are loaded at runtime). This error may also occur when debugging a single virtual method if the method is reentrant and the sub class implementation uses the debugged reentrant method as sub VI.
1031	Connector pane mismatch	The connector pane of a sub class implementation is not identical to the virtual base class connector pane. Remember that a required input in the base class must also be required in a sub class implementation.
1035	Operation is invalid for this type of VI	A dynamically called method in a sub class is LabVIEW™ Polymorphic, which is not allowed for virtual methods.

Table 1. Error codes used in GOOP 2.0.

Notice that the destructor will always execute even if there is an error in and the object will be deleted. This way memory leaks are avoided. A good idea is to use the new *Objects In Memory* debugging tool to trace non-deleted objects.

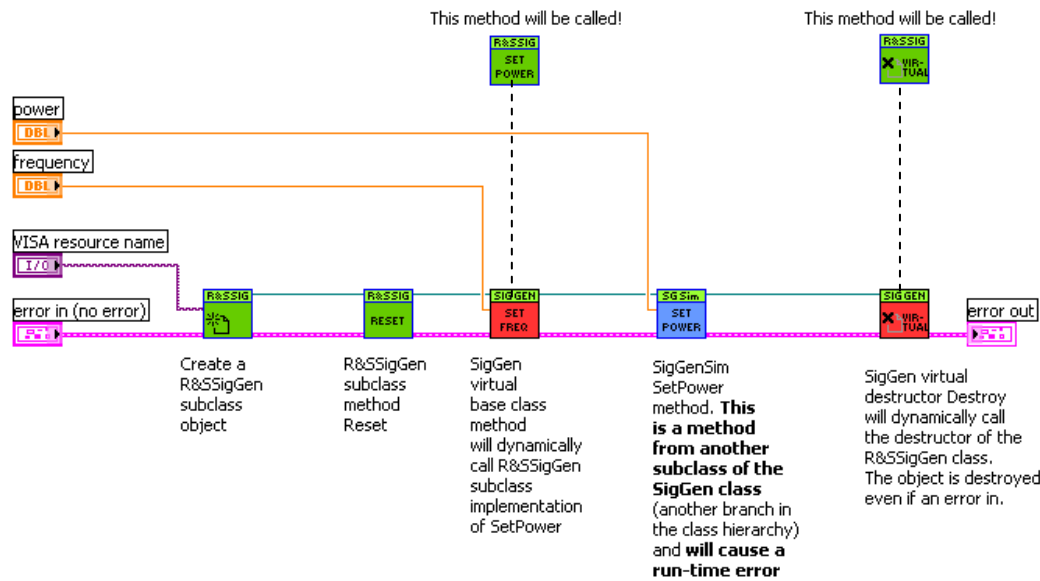


Figure 25. An example of an error situation where a method from another subclass (from a different branch in the class hierarchy) is used incorrectly. It is possible to wire them together because both of the R&SSigGen and SigGenSim sub classes has inherited from the same base class and therefore use the same ObjectReference.ctl. This will cause a runtime error in the SigGenSim that will notice that the reference is not valid for the SigGenSim class.

12 Distribution

When developing LabVIEW™ applications and there is a need to build a distribution to make a release of the application, usually one of the four strategies are used:

1. Just deliver the LabVIEW™ code as it is.
2. If the code is version controlled, release a version in the Source Code Control System³³ by applying labels.
3. Create a LabVIEW™ LLB with “Save with options...” containing all VIs needed in the application and maybe apply a password or remove diagrams.
4. Build Application (.exe) with the NI LabVIEW™ Application Builder.

There may also be other ways and combination of the above strategies. Normally, the distributed application has some kind of version attached to the release.

For example, the GOOP Wizard is not developed, as it looks when installed, in a large LLB with all VIs in one place. The source code is in a folder structure which is under Source Code Control (SCC). Each time a new release is performed a label is attached in the SCC and also the large LLB containing all needed VIs is created using the *Development Distribution* explained in chapter 12.1. This shows that two of the strategies are used when releasing a new version of the Wizard.

Now, there is one problem by just performing the last two strategies, using the “Save as...” and Build Application. The problem is that GOOP 2.0 uses VI Server for

³³ ClearCase and SourceSafe are two examples of Source Code Control tools.

implementing the dynamic binding. Anyone that has tried to use the “Save as...” or Application Builder in LabVIEW™ knows that dynamically called VIs by a VI server needs to be included in the application manually.

Just building an application that contains a lot of virtual method that will call sub class methods will actually result in the sub class implementation *not being included* in the application. However, there is a solution to this problem. The new tool **Development Distribution will automatically find and add dynamically called sub class method (and their sub-VIs) to the distributed application.** The Development Distribution tool is included in the GOOP Wizard 3.0, but may also be run as a separate product. It consists of two parts:

- Development distribution – performs a “Save with options...” including all dynamically called method and their sub-VIs.
- Create Build Script – creates a build script (.bld file) where all dynamically called methods are added as “Dynamic VIs”.

There is one condition for the tool to work properly and find all dynamically called methods. **At least one VI from each sub class must be included in the LabVIEW™ source code.** Normally this is no problem, because the constructor method, Create, must be used somewhere to create the class and is mostly always used somewhere in the code. The constructor of a sub class may be the only sub class method actually used in the application if only virtual base class methods are used.

12.1 Development distribution

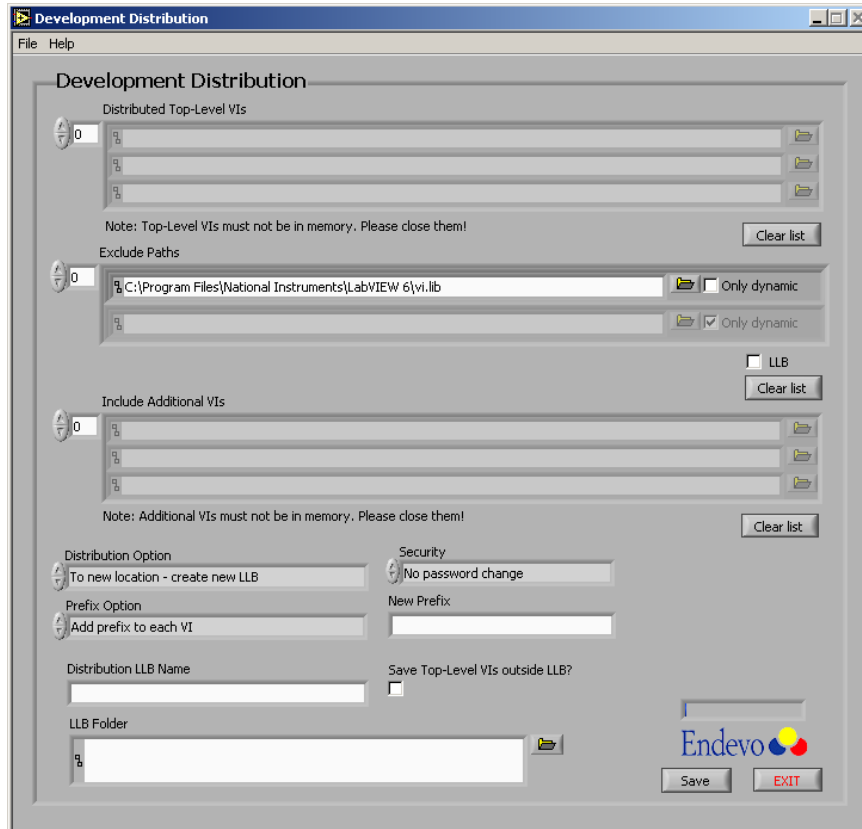


Figure 26. The front panel of the Development Distribution tool.

Figure 26 shows the front panel of the Development Distribution tool. This tool will perform the same thing as a standard LabVIEW™ “Save with option...” and save all VIs into a large LLB with an optional password. This tool is a little bit more advanced and include all dynamically called subclass methods and their sub VIs as well. If any of the dynamically called VI’s contain more dynamic calls, these will be added as well. A complete recursive search through the entire VI hierarchy (including dynamic called VIs) will be performed to find all possible dynamic calls.

There is also a possibility to use a regular folder instead of an LLB and also to prefix each VI in the distribution. Read all about the Development Distribution tool in ref [6] Development Distribution – White Paper. Available at www.symbio.com.

12.2 Building executables

The same problem with dynamically called VIs occurs when using the NI LabVIEW™ Application Builder. All dynamically called sub class methods must be added as “Dynamic VIs”, which might be quite a job to trace and add.

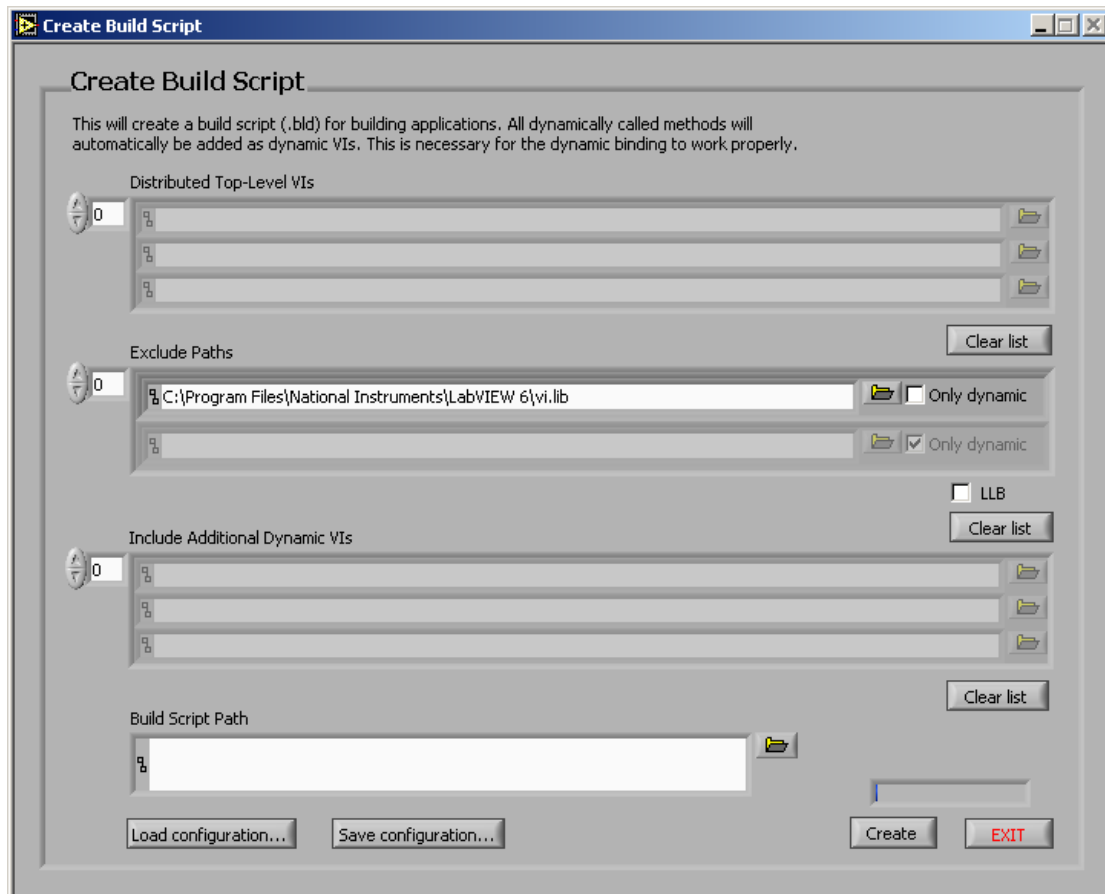


Figure 27. The front panel of the *Create Build Script* tool that is part of the *Development Distribution* tool.

However, the solution is to use the new tool, *Create Build Script*, which is part of the Development Distribution tool. This will create a build script (.bld) that is loaded in the NI LabVIEW™ Application Builder. All dynamically called sub class methods

are automatically added as “Dynamic VIs”. Read all about the Create Build Script tool in ref [6] Development Distribution – White Paper. Available at www.endevo.se.

13 Performance issues, common mistakes and pitfalls

When developing LabVIEW™ applications based on GOOP there are some aspects that should be considered for the code to be efficient.

13.1 Performance

There are a number of aspects for improving performance in GOOP2. Considering the following aspects and try to avoid these situations if possible.

- **Avoid having large complex data structures as attributes.**
- **Avoid having attributes that are “growing”**, for example an array that gets more and more elements. This will force LabVIEW™ to reallocate memory each time the attributes should be written
- **The number of objects will affect performance.** More than a couple of hundred objects will slow down performance radically.
- **Avoid reading and writing attributes inside loops.** If it is possible to read the attributes first and then write them when done, this will improve performance. If the strategy can be used depends on if the updated attributes are needed elsewhere in the application, read in parallel as the values are updated. Figure 28 and Figure 29 shows an example.
- **The class depth, number of ancestors, will affect access time to the attributes.** For example, if the access time to a root class attribute is x ms, a sub class will have the access time of $2x$ ms. If this class is inherited as well, there will be an access time of $3x$ ms for the sub-subclass and so on.
- **Do not make a method virtual, unless you really need to.** Especially in subclasses, usually it is enough if the base class method is virtually declared. All virtual methods have to perform a check if a sub class implementation exists. However, the GOOP 2.0 uses a caching algorithm to store information if sub class methods exists or not and also stores VI references to be reused after the first call to a dynamic method. This way performance for virtual methods are maximized, but will always be slower than non-virtual methods.
- **Use preloading of the redefined subclass methods in the constructor.** This will cache information that the sub class implementations exist and also load them to memory. This is a tremendous performance improvement.

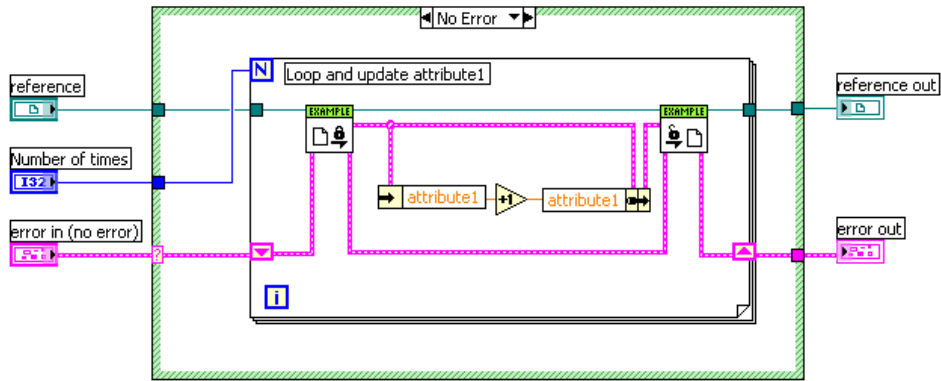


Figure 28. This “dummy” example shows a very inefficient and unnecessary updating of attributes. There is a loop updating an attribute over and over again. If the attributes are not read in parallel somewhere else in the execution this is an unnecessary writing of the attribute.

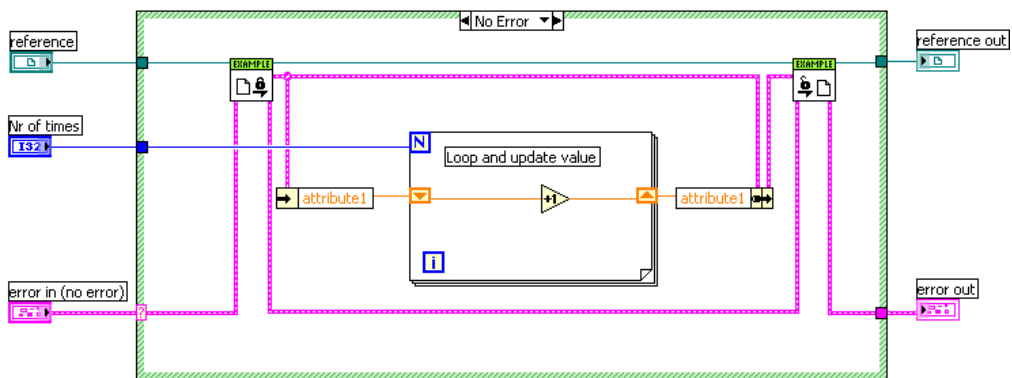


Figure 29. A much faster implementation of the same method as shown in Figure 28. The loop actually does nothing, but it illustrates the principle, avoiding updating attributes in a loop if it is not absolutely necessary.

13.2 Common mistakes and pitfalls

There are some common mistakes that all GOOP programmers will make sooner or later.

13.2.1 Deadlock situations

There are a couple of deadlock situation that might occur when using GOOP. The most common deadlock situation is shown in Figure 30. A modifying method locks the attributes, and then calls a private method that also wants to modify the attributes. When the private method tries to lock the attributes, it will be forced to wait because the attributes are already locked by the calling method. The private method will actually end up waiting for it self to finish executing! This will of course never occur and the deadlock situation is a fact.

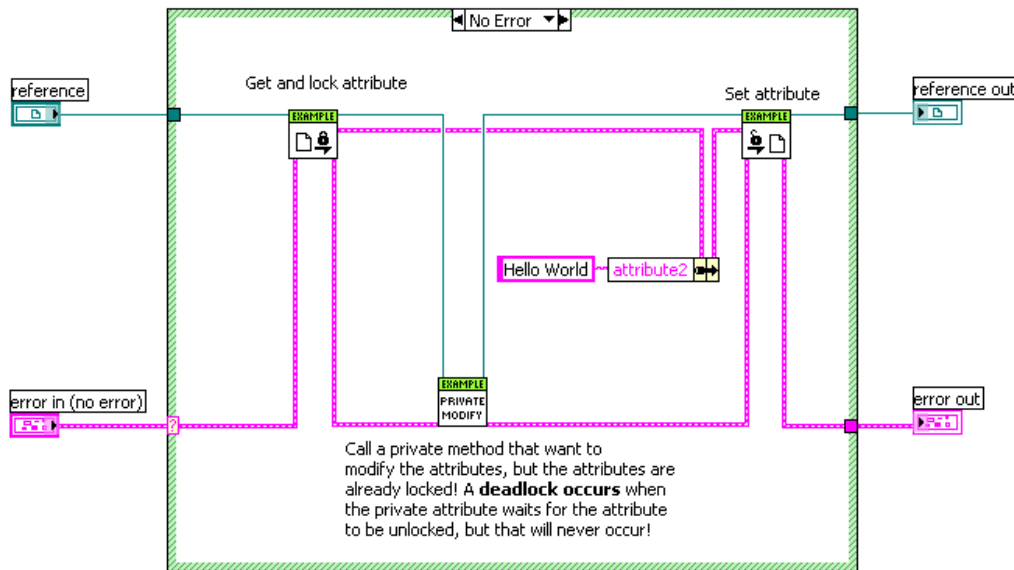


Figure 30. An example of a common deadlock situation. A modifying method locks the attribute and then calls a private method that also tries to modify the attribute. The private method will end up waiting for the attributes to be unlocked which of course never will occur.

Another deadlock situation that is common is when working with a virtual method that calls a subclass implementation, which uses *the same base class method VI as a sub-VI*. The virtual method must be reentrant and this is explained in detail in chapter 8.2.2.

If the virtual method is not reentrant, it will actually end up waiting for it self to finish executing (the Call By Reference Node is actually still executing until the sub class implementation is finished). When the sub class implementation wants to run the virtual method as a sub-VI, it will notice that it is already executing and will wait for it to finish and that will never occur! If the virtual method is reentrant, the sub class implementation will not have any problem executing the virtual base class method as a sub-VI.

13.2.2 Virtual methods do not work

When virtual methods do not work as expected there is a number of checkpoints that might be considered:

- The connector pane of the base class method and sub class implementation must be identical. Remember that a required input in the base class must also be required in a sub class implementation.
- The method name of a subclass implementation must be identical to the base class method.
- The controls and indicators in virtual methods must be wired to the Call By Reference Node in the “Call Dynamic Method” case.

13.2.3 Other common mistakes

One of the common mistakes that new users of GOOP do is actually forgetting to destroy the objects. Make sure that the *Destroy* methods are executed even if an error occurs in the code somewhere. The destructor will delete the object even if there is an error in, but the destructor might be placed in a sub-VI that will not execute if there is

an error. Use the *Objects In Memory* tool to make sure no objects remain when the application finishes execution. Kill all objects and trace the memory leak in the code.

When building an application with GOOP, do not “overuse” creating classes. GOOP is not a native part of LabVIEW™ and many GOOP classes result in a lot of sub-VIs. There is nothing wrong with creating collections of VIs³⁴ that are not GOOP classes and use them together with GOOP classes. If you cannot find any attributes when designing a class, consider if it is really necessary to use a class.

14 System requirements

- LabVIEW™ 6.1 (or later)
- GOOP Wizard 3.0 requires LabVIEW™ Professional Development System or LabVIEW™ Full Development System
- GOOP 2.0 runs on all LabVIEW™ 6.1 Development Systems

This means that it is possible to develop applications that will run on the LabVIEW™ Base Package using GOOP 2.0, but requires LabVIEW™ Professional Development System or LabVIEW™ Full Development System while developing the application with the GOOP Wizard 3.0.

15 Example

This example will demonstrate how inheritance of attributes and methods works, how to implement virtual methods and how an “abstract” class is created.

15.1 The challenge

The challenge is to build a system using a *generic* spectrum analyzer in the code; this system must work together with many different spectrum analyzers of different vendors. New kinds of spectrum analyzers must also be easy to add in the future. In our example, the spectrum analyzer in the lab is not always available and occupied by others. Therefore we also want to use a simulated spectrum analyzer when developing. This way we may test the code that uses the spectrum analyzer and see it behaves correctly.

For now, it is enough to support only the R&S FSEM30 spectrum analyzer. However, this spectrum analyzer needs to be calibrated, but sometimes we do not need calibration, so we must make this system flexible.

15.2 The design solution

With a pen and a piece of paper we quickly come up with a design solution shown in Figure 31 as an UML class diagram with the class hierarchy.

³⁴ This is also called *Function Libraries*.

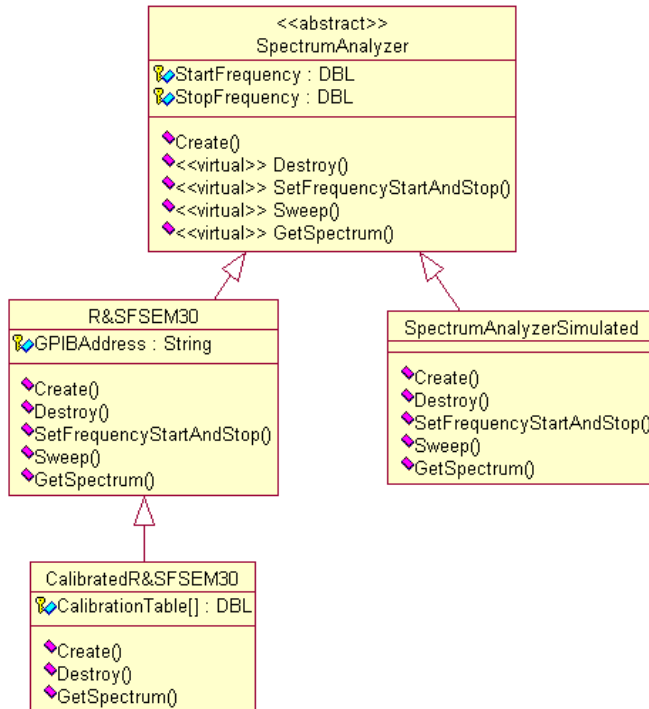


Figure 31. The class hierarchy used in the example. The root base class at the top is an abstract Spectrum Analyzer class. It has two sub classes, one real implementation the R&S FSEM30 (which is a Rohde&Schwartz™ spectrum analyzer) and the other, a simulated Spectrum Analyzer.

We use an *abstract* base class called *SpectrumAnalyzer* with all methods declared *virtual*. This class describes the *interface* of the spectrum analyzer. The interface is the set of public methods that the spectrum analyzer exports. This way we may develop an application using a spectrum analyzer without actually knowing what kind of spectrum analyzer is being used. This is a good example of *generic* programming.

In this example we only support one model, but if more models should be supported, just make another subclass from the root base class, *SpectrumAnalyzer*. The simulation of the spectrum analyzer is implemented as a sub class, just as if it were another spectrum analyzer model.

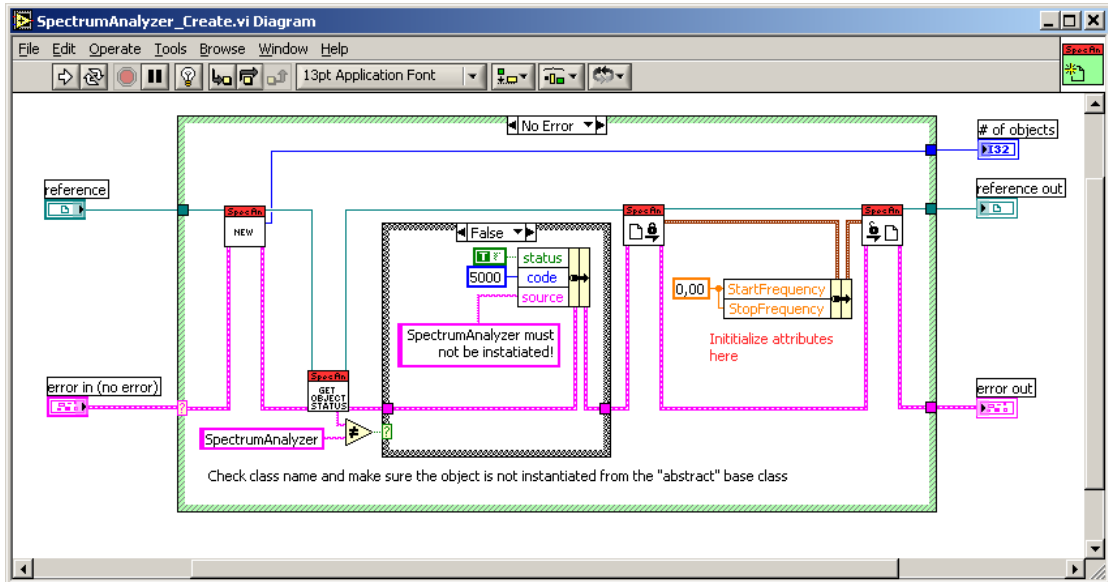
To support calibration of the R&SFSEM we create a sub class of the R&SFSEM30 class, redefine the *GetSpectrum* methods and add the calibration routine. However, we do not want to reimplement how the R&SFSEM30 retrieves the spectrum, so we *use the R&SFSEM30 implementation in the sub class implementation*.

15.3 The implementation

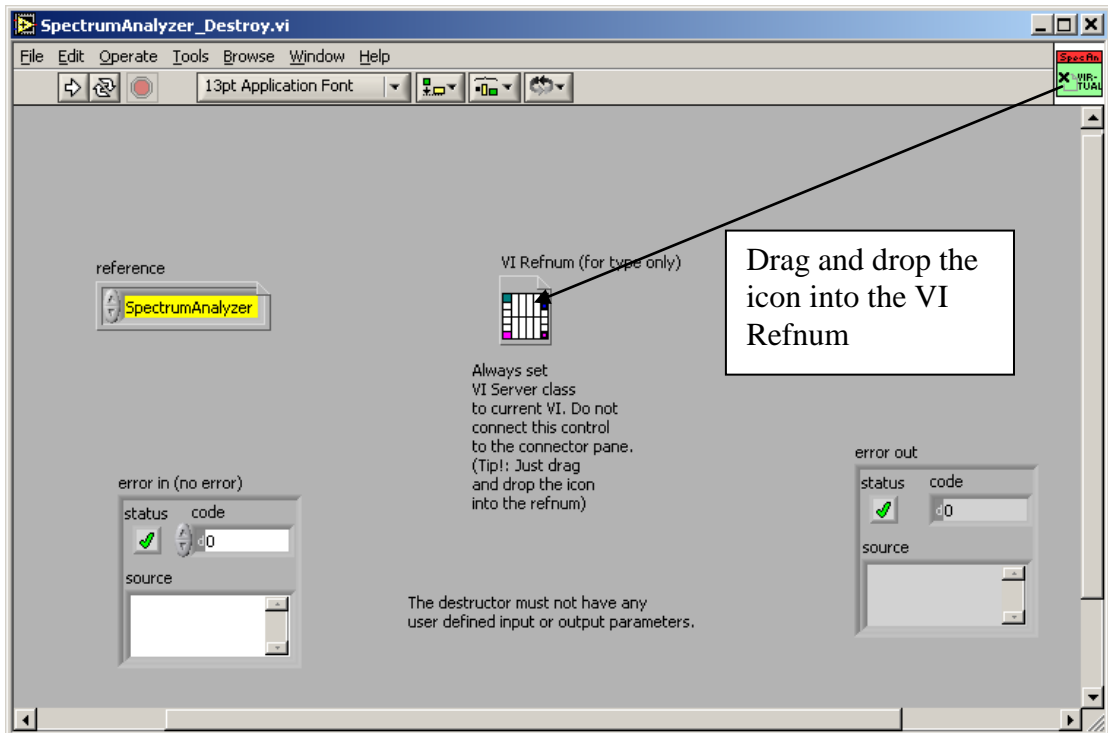
Now we are ready to implement the system. Notice that the GPIB communication is not totally correct and is not working in reality. It is simplified for the example.

First we create the base class. GOOP2.0 does not support abstract classes but we are going to perform a small trick to make it behave almost as an abstract class.

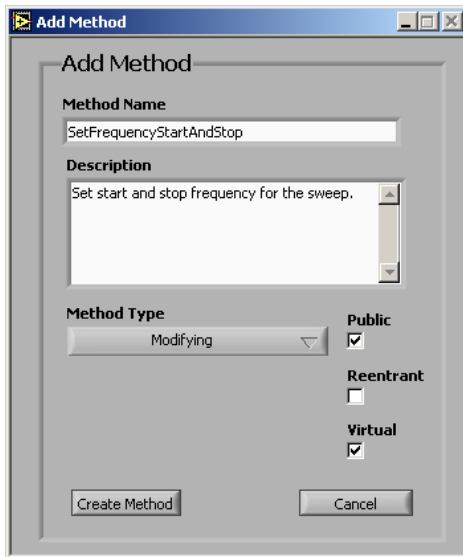
1. Create a standard class called *SpectrumAnalyzer* with the Class Generator in GOOP Wizard 3.0. Select *New class* in the file menu. Follow the instructions. Make sure that the *Virtual Destructor* is set to TRUE. Use the attributes from the design in Figure 31.
2. When the class creation is finish, open the constructor, the *Create* method and implement it. The result is shown below.



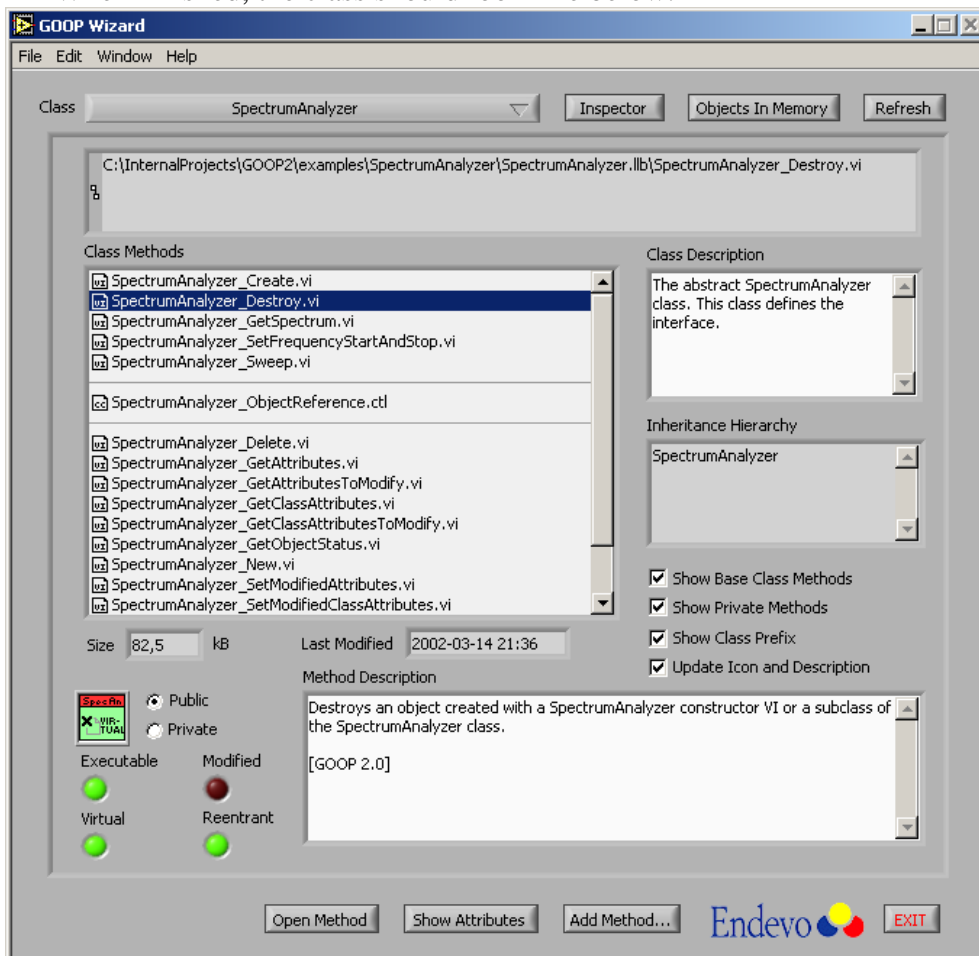
3. Open the virtual destructor method, *Destroy*, and notice that the VI Refnum control must be updated directly. Just drag and drop the destructor icon into the VI Refnum control on the front panel. Remove the GetAttributes in the "Class implementation" case.



- Now, choose *Add Method* either from the menu or press the button at the front panel of the GOOP Wizard.



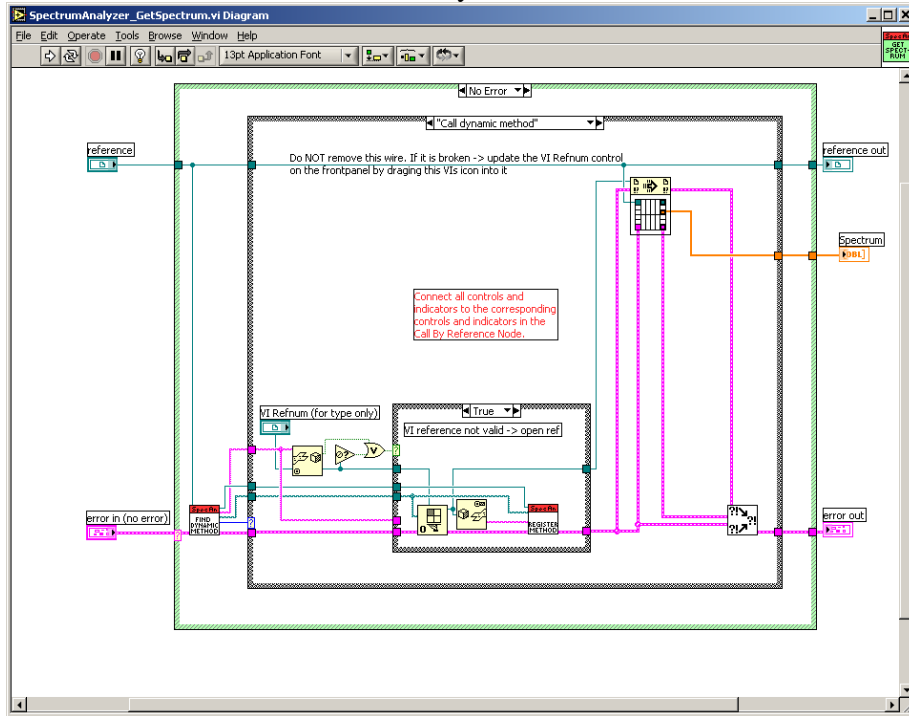
- Create the method of the class one by one, make sure *Virtual* is set to TRUE. When finished, the class should look like below.



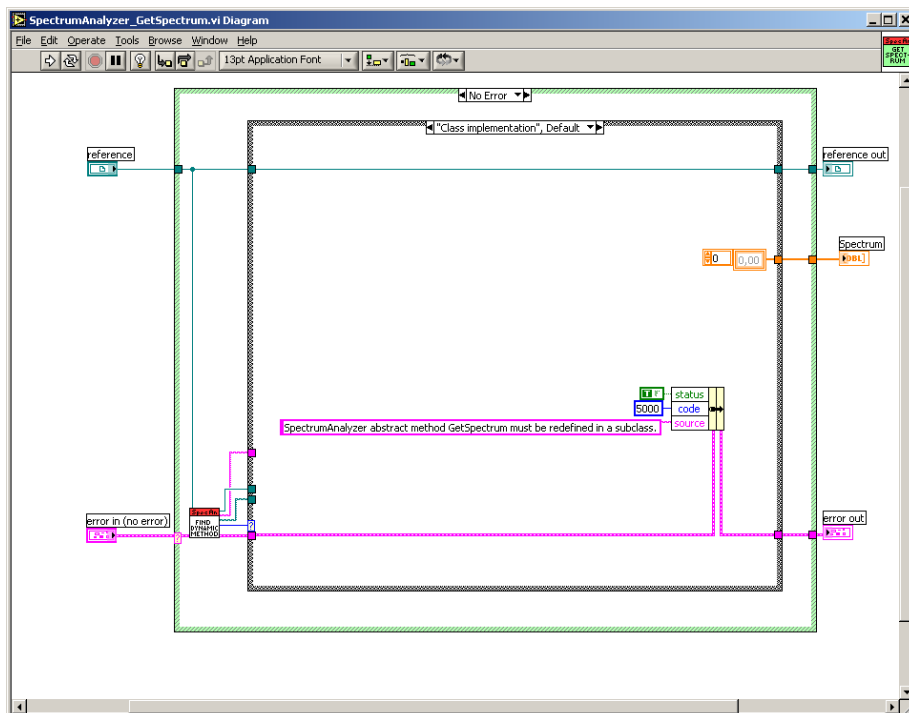
- Now, implement all methods by adding suitable controls and indicators as described in the design in Figure 31. Attach them to the connector pane.

Update the VI Refnum control by dragging and dropping the icon of the method onto it as described in step 3.

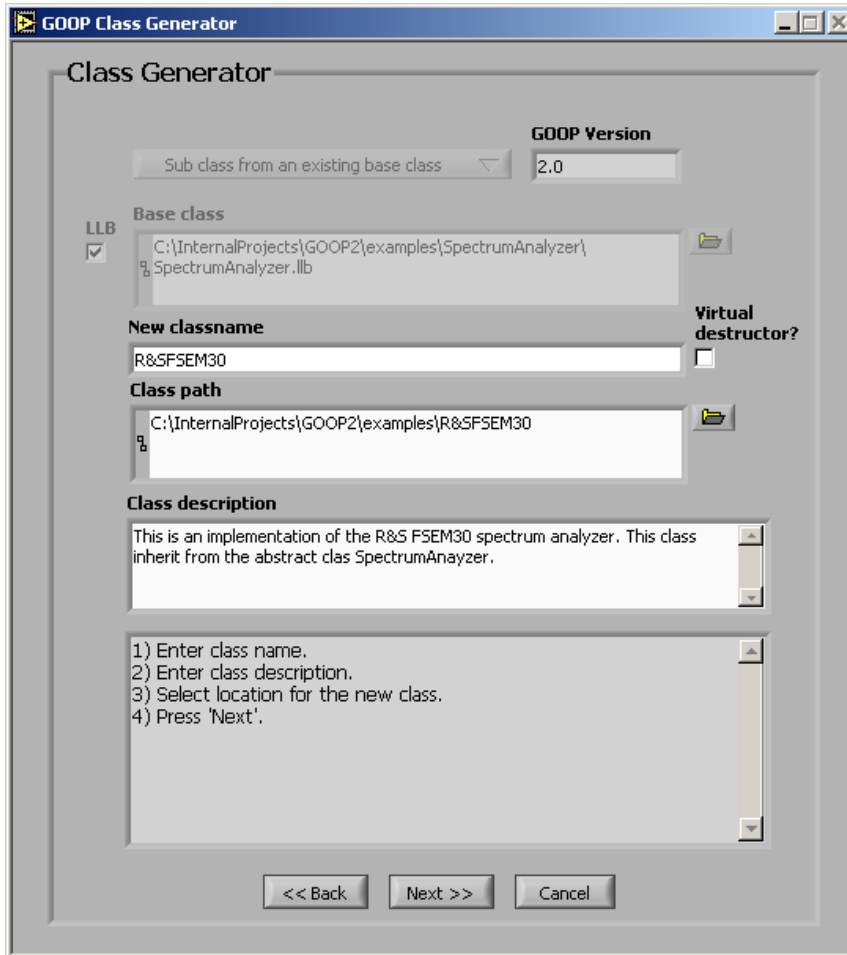
The Call By Reference Node in the “Call Dynamic Method” case will now adapt to the connector pane of the same method. Wire all corresponding controls and indicator to the Call By Reference Node.



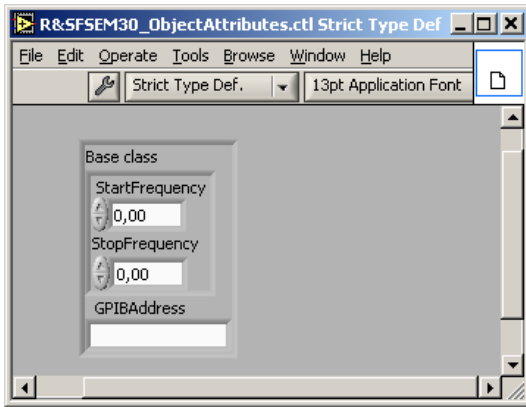
7. In the other case, “Class implementation”, simply generate an error and wire constants to the indicators. This case must never execute in an abstract class. This method must be redefined in a sub class.



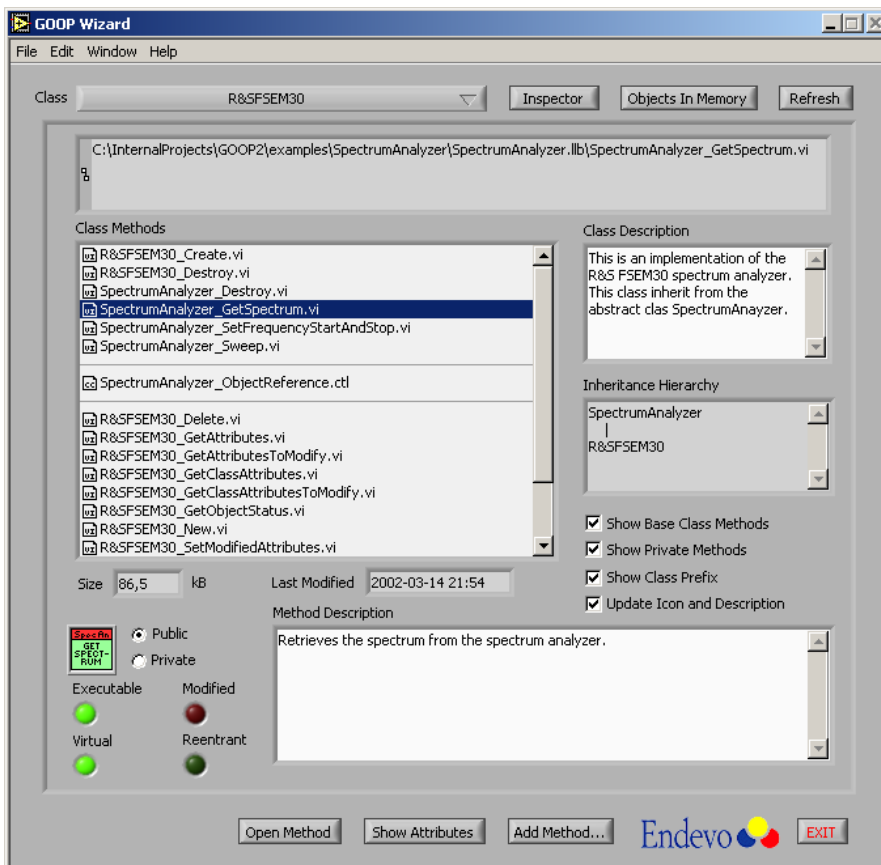
8. When the abstract base class is finished it is time to create the *R&SFSEM30* sub class. Choose *New Class* again in the GOOP Wizard 3.0. Choose class type “Sub class from an existing base class”. Now, the base class to be used must be pointed out. Browse and select the *SpectrumAnalyzer* class. Do not select a virtual destructor.



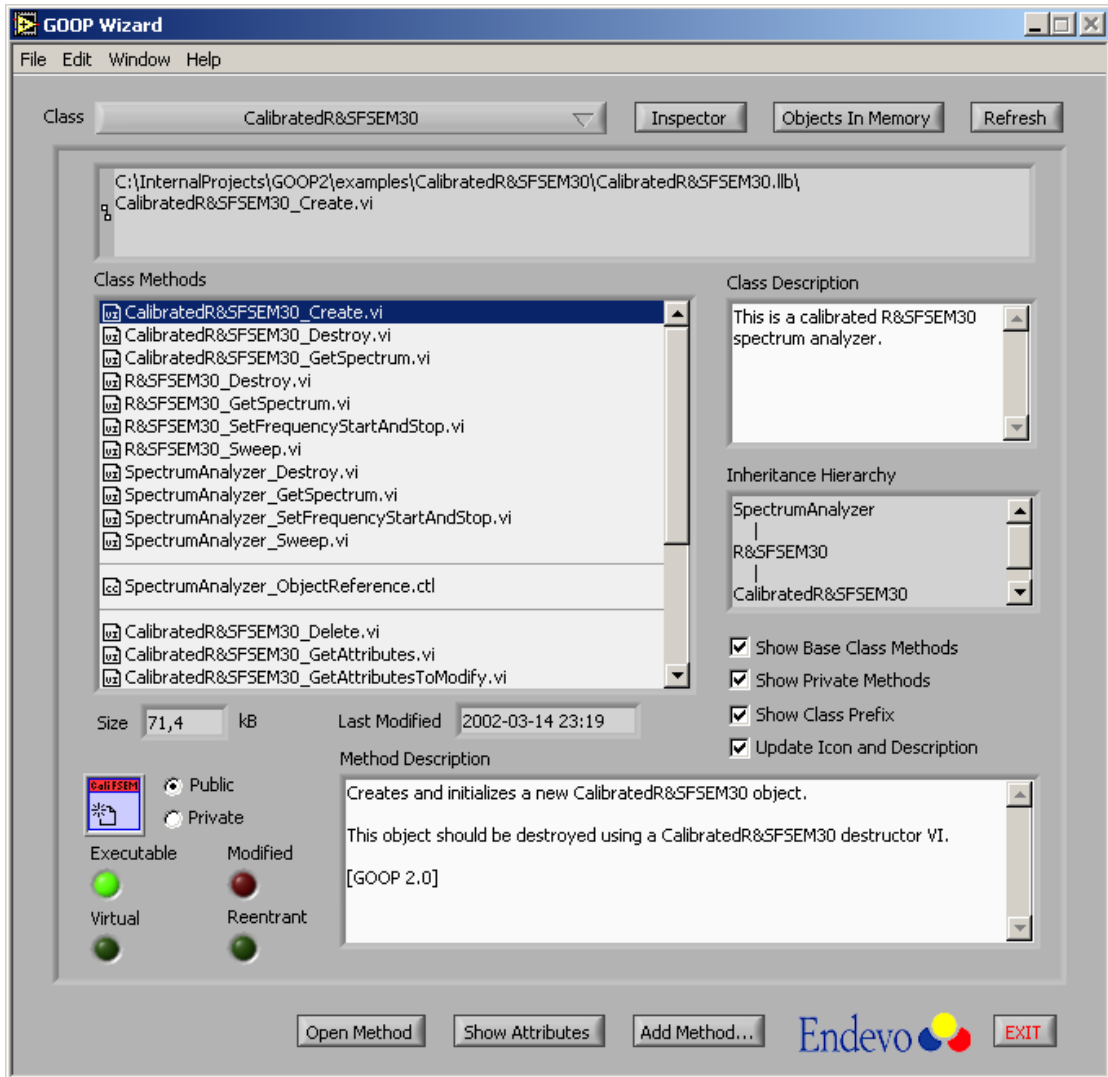
9. When the ObjectAttributes.ctl and ClassAttributes.ctl pops-up, just close the ClassAttributes.ctl. We are not using class attributes. Notice that the SpectrumAnalyzer_ObjectAttributes.ctl is an element called “Base class” in the R&SFSEM30_ObjectAttributes.ctl. Replace the dummy attribute Child attribute 3 with “GPIBAddress” as stated in the design from Figure 31. You may also notice the ObjectReference.ctl never pop-up. A sub class always uses the base class’ reference.



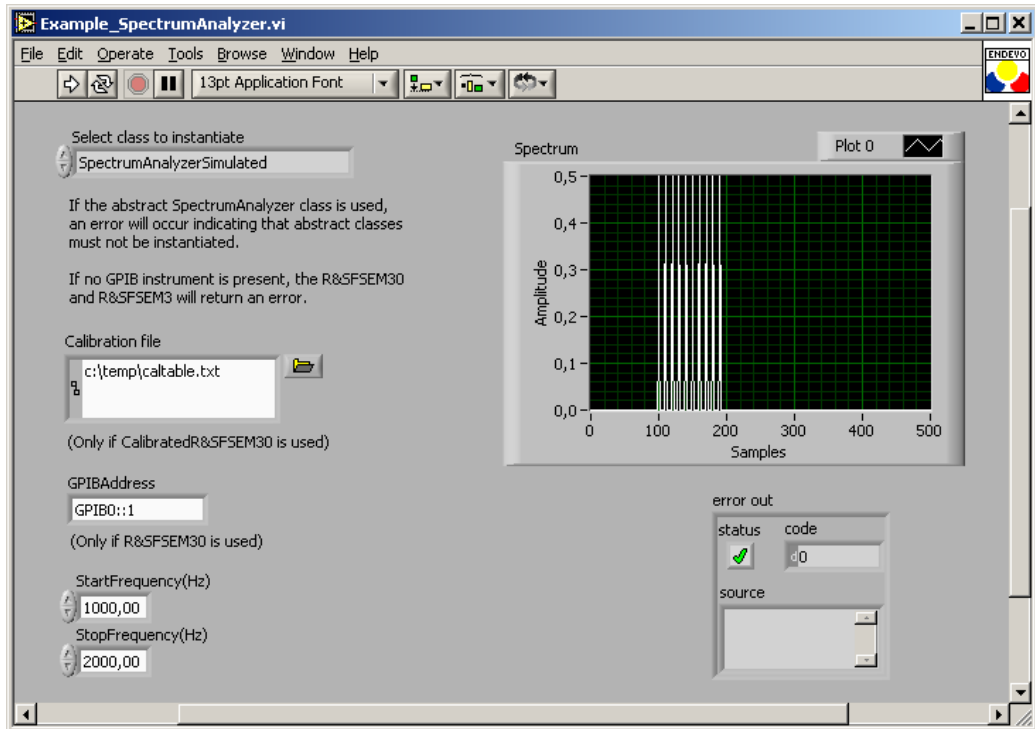
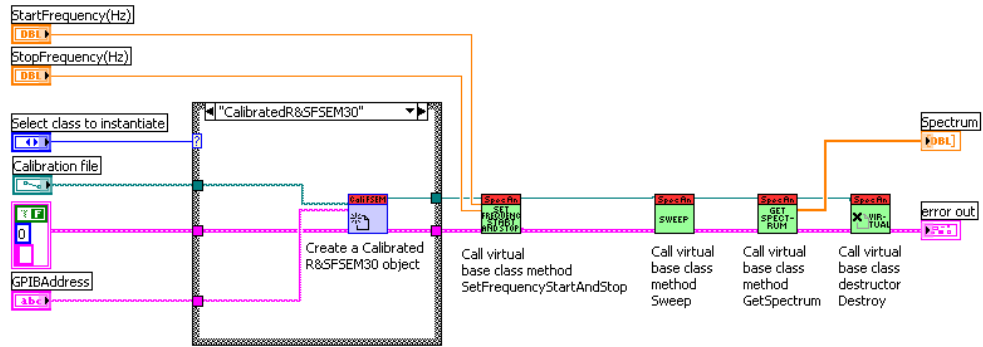
10. After the class has been generated. The class looks like below. Notice the inheritance hierarchy and that all inherited methods are also shown in the method list.



11. Open and implement the constructor and destructor just as we did for the SpectrumAnalyzer class. All the constructor should do is to update the GPIB address and also preload the methods we are going to implement in the sub class. The destructor does not do anything.



17. Now, it is time to also implement the simulated spectrum analyzer, *SpectrumAnalyzerSimulated* class. Repeat the same procedure as for the R&SFSEM30 class, when creating the class. Select the SpectrumAnalyzer as base class.
18. Implement all methods as “dummy” methods that actually do nothing, but they have to exist or else the base class implementation will be executed and that is forbidden for an abstract class and will cause an error. Let the GetSpectrum method return a simulated spectrum consisting of a number of sine tones.
19. When all classes are implemented, it is time to test the classes. Let’s build a small test VI. It consists of a case structure, one case for each of the constructors of the classes we created, which chooses the class to instantiate. All other methods used are virtually declared methods of the SpectrumAnalyzer abstract base class.



20. Choose the class to instantiate on the front panel and run the test VI. Above is the result when the SpectrumAnalyzerSimulated class was used and the simulated sine tone spectrum is shown on the front panel. The R&SFSEM30 and CalibratedR&SFSEM30 will probably cause a GPIB error (if not FSEM is connected of course!). If the abstract class is instantiated there will always be an error indicating that an abstract class never must be instantiated.

The only class that may be executed is the simulated spectrum analyzer. This illustrates the power of inheritance and how flexible it is to add new instruments and also simulations that make it possible to develop (and debug) a system without actually having access to, or knowledge of the instruments that will actually be used later. Below you can see the result of trying to run the system using the SpectrumAnalyzer class and an R&SFSEM30 class.

