Limitations of Current Implementation of Object Oriented Programming in LabVIEW 8.20 and a Proposal for an Alternative Implementation

v. 1.2

Tomi Maila
Tomi.Maila@helsinki.fi

University of Helsinki

August 16, 2006

Introduction

Object-oriented programming is a programming paradigm that has proven to be an excellent tool to build modular and easily maintainable software and software components. Object-oriented programming has been extremely successful in the area of large-scale application design, which has been one of the weak areas of LabVIEW until now. The paradigm of object-oriented programming has been under heavy research since the introduction of the concept in1960's. The object oriented programming, as we currently know it, is a result of a 40-year long evolution. During the evolution, the paradigm of object-oriented programming has developed into a number of proven concepts, which all leading object-oriented programming languages share.

Object oriented programming implementation in LabVIEW 8.20 doesn't share many of the above mentioned proven concepts; instead many of these proven concepts are left out of the implementation leaving the current implementation practically handicapped. This wouldn't be a major problem if the paradigm of object oriented programming would allow adding these features later on and still keeping the LabVIEW backwards compatible with the earlier implementations of the object-oriented programming. This, however, is not the case. The object oriented programming concepts left out of the LabVIEW 8.20, are so central to the object-oriented programming paradigm, that the missing features cannot be added later on still keeping the backward compatibility.

In this document we first give an analysis of the most important problems with the current implementation. After pointing out the key problems with the current implementation of object-oriented programming in LabVIEW, I propose an alternative implementation in the second half of this document that should overcome many of the above mentioned problems.

Contents

1 Document License

2 Limitations in the LabVIEW Implementation of Object-Oriented Programming

2.1 Initialization, Cleanup and Dataflow

2.1.1 Constructors

LabVIEW doesn't allow software developers to create non-default constructors for the classes.

In object-oriented programming paradigm objects abstract both the functionality and the properties of an entity. In object-oriented programming paradigm objects should never, not even transiently, be in a state that doesn't represent a valid entity. To ensure this, objects often need to be initialized to a proper state. For object initialization object-oriented programming languages use the concept of class constructors. A class constructor may either be a class developer defined function or a default constructor if class doesn't need any special initialization. A class constructor is always called when an object that class type is created. If object creation for some reason fails, an error is generated. There may be multiple constructors per class, each of which creates a valid object of that class type.

In LabVIEW 8.20 implementation of object-oriented programming only default constructor is available. The default constructor only reserves memory for the object and sets member variables to their default values. LabVIEW constructors cannot interact with other objects or other environment which is often required for class initialization. Since non-default constructors are not available in LabVIEW, objects can be initialized only after creation by calling a class method.

There are several problems with the LabVIEW implementation of object initialization. First of all, class user can create an object that isn't properly initialized. A common problem in many programming languages like C is that these languages don't require variable initialization. Software developers are often in rush and easily forget initialization. Class user can then pass the non-initialized object to functions and methods that expect the classes to be initialized. This may cause unnecessary bugs which could be easily avoided by non-default constructors.

A central concept of object-oriented programming is hiding the implementation; the class user doesn't need to know how the class is implemented in order to use it. In the early implementations of a class, initialization may not be needed. In later implementations some initialization tasks may be required for the class objects to function properly. Most object-oriented programming languages always call a non-default constructor when it is present and in such a way ensure that initialization is taken care of when the class developer decides to change the implementation of the class to include initialization.

In LabVIEW it is up to the class user to decide if he will call the initialization method. Since initialization wasn't required in the early versions of the class, the user may have written a program that doesn't or even cannot call any initialization method for the object. Later on, when the class implementation is changed so that initialization is required, the class user must modify each of his programs using the class to call the initialization method. This is against the central concept of hiding the implementation in object-oriented programming paradigm.

A constructor call at the position where the object is being created makes the code more readable than a separate call to initialization method. It's more logical to create a red car than to create a car in general and then later on initialize it to be red.

A constructor of most object-oriented languages cannot be called twice. In LabVIEW, an initialization method can be called as many times as the user of the class wishes. This may also be prone to errors. In early implementations of the initialization method, this may not cause problems, but changes in the initialization method once again breaks the code and forces the user of the class to modify all of his code not to use double initialization.

In most object-oriented programming languages the compiler takes care that the parent class and member classes are always properly initialized before the class itself is initialized. In LabVIEW the class developer may easily forget to initialize the parent class and member classes which will lead to numerous bugs.

A constructor in most object-oriented programming languages is different from other class methods in such a way that constructors can never be inherited from an ancestor class. In LabVIEW the initialization method is always inherited from an ancestor class. This causes severe problems because one cannot change the initialization interface (front panel and connector pane) from that of an ancestor class.

Examples

Database class

Let DatabaseClass be a class that represents a database. To be able to connect to the database, the object of the DatabaseClass needs the following information: database server address, username and password.

In major object-oriented programming languages to create a DatabaseClass object, user must always pass the server address, username and password to the class constructor. If the connection to the database server fails, an object is not created and an error is generated.

In LabVIEW the DatabaseClass object cannot be initialized when created. This allows user to create a DatabaseClass even when there is no database available at all. The object creation always succeeds, meaning that even though the database is not available, an error is not generated. To initialize the object, user can call a separate initialize class method after the object has already been created. Since this is not mandatory task for the class user, the user may not call the initialize method and the class is not initialized. So user can create a DatabaseClass object which doesn't even represent a database. This is in contradiction with the object-oriented programming paradigm in which an object always represents an entity.

Polygon and square graphic object classes

One of the corner stones of object-oriented programming is reusability which is compromised in LabVIEW implementation of object oriented programming. Let's assume that there is a graphics library with a PolygonClass to create and draw polygons on a screen. The PolygonClass has an Init method, which takes an array of corner points defining the polygon as a parameter.

A class developer would like to create a SquareClass. He decides to use the PolyhonClass as a starting point. He would like to create an initialization function that takes the co-ordinates of the top-left corner and the side length of the square as input parameters. However he cannot use the Init as the name for the initialization method, since the interface of the SquareClass initialization is different from the interface of the PolygonClass initialization. He creates a method called Init2 to initialize the SquareClass objects. LabVIEW requires the function interface to stay the same tough the descendent classes.

The class user is accustomed to call Init method to initialize the class since it's the method name most National Instrument classes use for initialization. This method however doesn't initialize the square correctly but rather initialize a polygon. Now the functions taking squares as input parameters as well as square member functions may fail because of improper initialization.

2.1.2 Destructors

LabVIEW doesn't allow users to create class destructors.

In addition to proper initialization, a proper cleanup is a central concept in object-oriented programming paradigm. Clean-up is not only about cleaning the object from the system memory; it may contain multiple class developer defined steps. Unlike most object-oriented programming languages, LabVIEW doesn't force class cleanup. As with initialization, class developer may create a clean up method for the class, but in the end it's up to the user to call this clean-up method.

In most object-oriented programming languages classes have a special method called destructor which is always automatically called when the class is no longer needed. The destructor method takes care of the class clean-up. The class-developer can define steps that it takes to clean-up the object. The compiler takes care of calling parent class and all the member class destructors in proper order in the clean-up process.

Traditionally LabVIEW has been famous for freeing programmers from thinking of clean-up routines. It automatically takes care of freeing memory when a wire buffer is no longer needed. It would be natural that the class users didn't have to think of the clean-up routines themselves; instead the class developer defines what kind of clean-up the object needs and then LabVIEW would automatically take care of the clean-up when the object is no longer needed.

This however is not the case. LabVIEW doesn't allow automatic cleanup of objects unlike other object-oriented programming languages. The class user must call a clean-up method when he no longer needs the class to advice LabVIEW to take care of the clean-up. It would be extremely easy for the class user to forget proper clean-up. This will lead to numerous hard-to-find bugs in LabVIEW code.

Examples

File I/O class

Consider a FileIOClass which is used for writing data to a file. When an object is initialized a file is created for writing data. To speedup the writing process, the data is stored in a buffer and written only when the buffer is full. The class has a method AppendData which appends data to the buffer and if buffer is full flushes the buffer content to the file. The class also has a cleanup method which should be called in the end to write the data still remaining in the buffer to the file, and closing the file properly.

The class user uses the class to write data to the files. Since calling the cleanup method is not obligatory, he forgets to call the cleanup method in his software. The beginning of the file is written correctly but the last bytes from the file are missing. Because the file format is quite simple, the file looks perfectly ok, it just doesn't contain all the data there should be. The user of the class doesn't notice the problem and keeps using the class incorrectly for years. Later on he notices that he has been using the class incorrectly and that all the data files he has from several years are of no use.

2.1.3 Copy-constructors

LabVIEW doesn't have a concept of copy-constructors

Object-oriented programming languages ensure that when ever a copy of an object is created, it's properly initialized. Just copying the memory content of the object is not always enough. Sometimes the class developer must be able to define some proprietary task that must be performed when the object is copied. This is implemented with class developer defined copy-constructor in most object-oriented programming languages. If a copy-constructor for a class exists, it's always called when a copy of a object of that class type is made.

LabVIEW doesn't have the concept of copy-constructor at all. Being a data flow language, it's extremely easy to create copies of the objects in LabVIEW by branching the wire. These copies may not (or even cannot) however be properly initialized. As a user experience point-of-view the user expects that a copy of an object is created when a wire is branched; you do get a copy of all the other data types when you branch a wire, why wouldn't you get a valid copy of a class when you branch the wire.

If a copy of the class is not made when wire is branched is also problematic from other than user experience point-of-view. If an object wire is branched and a method modifying the object is called on one of the branches, then branches may no longer be equal but not different either. In this way two object which both represents "alternative futures" of the same object are created. This is definitely not what the programmer expects.

Instead of this weird behavior, a copy of the object should be made each time a wire is branched. If a copy constructor for the class exists, LabVIEW should call the copy constructor at each wire branching point to create a copy of an object.

Examples

File I/O class

We use the FileIOClass in our example once again. Consider a FileIOClass which is used for writing data to a file. When an object is initialized a file is created for writing data. The file reference is stored as a private data member of the class. The class also has an integer data member which points to the current position in the file which is incremented each time the data is written to the file.

When the user braches a FileIOClass object wire, a memory duplicate of the class will be made by LabVIEW. Both of these memory duplicates have the same file reference and same current position member variable at the moment of branching. The class user now calls a class method on one of the two braches which deletes some data from the end of the file and decreases the current position marker. The two braches now have different object which both refer to the same file. The object which called the method correctly points to the end of the file. On the other hand, the object of the other branch no longer is a valid object since the position marker points past the end of the file. There is no way to

avoid these kinds of situations in current implementation of LabVIEW object oriented programming.

2.1.4 Object references

LabVIEW doesn't allow creating references to objects and nor does it allow passing the same object to multiple block diagram nodes either.

LabVIEW is a dataflow programming language in which data buffers are almost always referenced by value. This is an important property of the language and allows highly efficient and concurrent code generation. LabVIEW objects are always referenced by value, and by reference objects are not allowed. Referencing objects by value is strongly encouraged as long as objects don't refer to real objects in outside world like files, windows, data-acquisition devices and so on. Referencing objects by value allows extending the highly efficient and concurrent code generation to object oriented programming. Serious problem arises however when there is a real-world counterpart for the LabVIEW objects.

There are two ways to create objects in most object-oriented programming languages; either statically creating them or dynamically creating references to the objects. There is a need for both of these ways.

Dynamic creation allows class user to define the lifetime of an object. The clean-up doesn't occur automatically, but the user must take care of the clean-up by calling a specific destruction routine when the object is no longer needed. In C++ for example dynamic objects are created with *new* command and destroyed with *delete* command.

Static creation of an object on the other hand is more limited but also easier to use. The compiler takes care of object creation and destruction. When copy of an object reference refers to the same object, copy of an object creates a new object of the same class. In this way the class user can choose between making copies of the object and passing the same object to multiple modules. Most conventional object-oriented programming languages also have a method to create a reference from a class and vice versa.

LabVIEW only has one representation for the classes; the static one. Objects are always referenced by value. This causes major problems for many reasons. First of all, the same object cannot be passed to multiple methods by branching the wire. Instead a (possibly invalid) copy of the object is always created as discussed in the chapter 2.1.3. Second, if the object may reserve huge amount of memory and other system resources. Making copy of the object may not be wise. Instead there should be a way to pass the same object to different places as a reference. Third LabVIEW uses object-model for front-panel objects which are passed as reference. This causes confusion among the users since there are two different kinds of objects which have distinct programming models. Fourth there should be a way to dynamically create and destroy objects, the process for which object references are more natural than objects themselves.

If copy-constructors are implemented as we suggest in the chapter 2.1.3, the object references are needed to avoid creating a totally distinct copies of objects when wires are branched. User could then select between creating a copy of the object itself and creating copy of the object reference, which points to the same object.

Examples

Data storage class

Consider a class object which holds a large data array in a private data element. Dataflow programming requires making braches of the class wire. Each of the braches makes a copy of the large data array consuming memory and taking time to make a memory copy. The memory would get easily fragmented and LabVIEW would run out of memory. This could be avoided by allowing references to objects which could be passed to various places instead of the objects.

File I/O class

We refer to our FileIOClass once again. In chapter 2.1.3 we presented a problem which may occur if an object wire is branched and an invalid copy of the object is made because there is no copy-constructor. Using class references instead of classes allow class users to avoid such problems in the cases when the same class needs to be referred in multiple places.

2.2 Visibility and Access Scope

2.2.1 Protected and Public Data Members

LabVIEW doesn't have protected and public data members

Even though main concept in object oriented programming is hiding the implementation, there are numerous cases where there are strong reasons not to hide all of the class data members from all of the other objects. For this reason most of the object-oriented programming languages have protected and public level data members in addition to the private level data members.

In LabVIEW all of the class data members are private and there is no way of creating protected or public data members. The only way to access the data members outside of the class is by calling on of the class methods. This can be a major disadvantage in many situations. First of all, LabVIEW is often used in high performance or real time environments. In these situations one has to choose between performance and hiding the implementation. Calling a method to retrieve a variable value may limit the performance significantly. Second, LabVIEW is an ideal language for fast prototyping. There are cases in which creating class methods to access each member variable just makes the

programming a simple and straight forward things much too complicated and time consuming.

Examples

Multiple virtual machines sharing a register

Consider a simple virtual machine which contains a registry to store values and a interpreter to interpret the code. The virtual machine could have multiple different instruction sets. Each instruction set is represented by a separate class. All of the virtual machines share a common registry, so it would be wise to create a parent class for the registry. All of the different instruction sets would then inherit from the registry class.

Virtual machines only perform simple operations such as computing +,-,*,/. The time it takes to perform a single operation is significantly less than the time it takes to call a member function to retrieve and store registry values. If protected level variables could be used the performance of the virtual machines could be increased maybe ten fold or even more. In this case there is even no reason to hide the implementation, the registry doesn't need to hide the implementation from the virtual machines. It's only the virtual machines who need to hide the implementation from the users of the virtual machines.

2.2.2 Class level data member variables (static variables)

LabVIEW doesn't allow creating class level data member variables.

Most object-oriented programming languages allow creating class level (static) member variables. These variables are visible to and shared between all objects of the same class. These variables are kind of class level counterpart of LabVIEW global variables. Class level variables are commonly used in object-oriented programming. Compared to global variables they allow easy way to share information between objects of the same class type and unlike global variables still hiding the implementation. LabVIEW doesn't have the concept of class level data member variables.

Examples

Object counter

A common programming problem is to keep count on the number of objects. An object-oriented solution is to create an object counter based on class level data members. The member variable is initialized to zero at program start-up. Each object created then increments the object counter during initialization and decrements the counter during destruction. At each moment the number of class objects can then be found out by reading the class level variable.

Counting objects in LabVIEW may be possible currently by using a shift register in a private method of the class. The shift register however is not re-initialized when main VI is rerun. Therefore the shift register trick only works for the first time the VI is ran. Class

level variables are also conceptually easier to understand and faster to access than member methods with shift register.

Concurrent file I/O

Another example is acquiring data from multiple devices and writing all the data to a single file. The data acquisition is taken care of object instances of data acquisition class; one object instance is created for each data acquisition device. The writing of data to the single file is taken care of a single instance of another class which takes care of concurrent requests using semaphores. The data acquisition class objects share a common instance of the file i/o class, which conceptually represents a single file.

As in the object counter example, the same functionality may be achieved with a private method of the data acquisition class. The private method should contain a shift register containing the file I/O object responsible for the file I/O. Now however the file should be properly closed when the main VI exits. Since shift register value is never disposed, a destructor cannot be called for a shift register value when the main VI exits. Class level object however could be disposed when the main VI exits and classes are destroyed (if destructors will be implemented as proposed in chapter 0).

2.2.3 Application Level Objects

LabVIEW doesn't allow creating global application level objects

Most object-oriented programming languages allow creating global instances of class i.e. global objects. These global objects are initialized when the main level VI is started and destructed when the main level VI exits.

LabVIEW has a concept of global variables, but they currently cannot be used as container for global level objects since the global variables don't have a concept of passing initialization parameters to the class constructor. Class default parameters cannot be used, since the initialization requirement may change from application to application and still the class default parameter values stay the same.

Ability to create application level object instances may be practical if the whole application needs to share a common resource.

Examples

Virtual private network (VPN) connection

Let's assume that an application needs to connect the company network trough virtual private network connection. A class is created to take care of the VPN connection. The same VPN connection is shared by all the objects in the application. To allow all the objects share the same connection, the VPN module responsible for the connection is

created as a application level object and as such is visible to all the objects in the application.

As in the previous examples, the same functionality may be achieved with a shift register embedded inside a application level VI. This doesn't however allow properly closing the connection when the application exits. A more elegant solution would be an application level object that would be initialized on main VI start and destructed on main VI exit.

2.2.4 Network Level Objects

LabVIEW doesn't allow creating network level objects

LabVIEW being especially designed for distributed systems it would be natural to extend the concept of application level global objects to network level global objects. Network level global objects would allow accessing a single object instance anywhere from the network as long as the connection security is properly taken care of.

A natural way to initialize a network level object is to read the initialization information from a INI file on a application server start-up and cleaning the object up on application server exit. Of course dynamic object creation must also be allowed.

Examples

Data acquisition database

A global data acquisition database could be implemented by creating a network level object that is initialized based on parameters in an INI file on LabVIEW application server start-up. The application server is started as a Windows service. Starting the application server then also creates a database object managing the database and remote connections to the database. Data acquisition devices call the methods of the network level object to store data to the database. Data-analysis tools call the methods of the network level object to retrieve data from the database.

2.2.5 Friend Classes

LabVIEW doesn't have allow creation of friend classes

Many object-oriented programming languages have a concept of friend classes. A firend class can access private data members and private methods of another class as if they were its own. The friend classes can be very practical to build objects that can monitor other objects. This is especially valuable during the development phase, when one needs to debug the software but one also don't want to add extra debugging functionality to the classes to be release for production. Friend classes can also be practical in other kind of system monitoring applications.

Examples

Printing debug information

Consider you are having troubles finding out bugs you have in your class. Using friend class you can create a class monitor that gives you real time information what is going on inside you class during runtime. This would be more practical than modifying your to be production code.

Storing class state to disk

Consider you need to sometimes save the states of all your classes to disk to be able to retrieve the current execution state later on. This may be practical if you need fast disaster recovery system in your data acquisition production environment. You could write a public save method for each of your classes. Public methods would allow anybody to call this method. You don't want to give the class user the opportunity to save the states of individual objects. Rather the saving of the system state should be done in background.

If friend classes were implemented in LabVIEW you could create a class that would be able to access the and save the states of all of the objects in memory automatically every now and then.

2.3 Inheritance and Interfaces

2.3.1 Abstract Classes and Interfaces

LabVIEW doesn't have concepts of abstract classes or interfaces

Abstract classes and interfaces are very central concepts in object-oriented programming paradigm. Abstract classes allow you to create classes which define the properties of the descendent classes but can from which a object can never be created. Similarly interfaces allow programmer to create methods that only contain the front panel components and connector pane but have no block diagram and as such cannot be called. Both abstract classes and interfaces are used by inheriting the class.

A descendent class of an abstract class contains all the properties of an abstract class but it can extend the properties of the abstract class and also objects can be derived from it. An interface can be implemented in a descendent class by creating a method with the same front panel and connector pane but with also a block diagram.

In some object oriented programming languages a whole class can work as an interface. These kinds of interfaces are often used in creating abstraction layers for different kinds of services and not having to allow inheritance from multiple classes. The object-oriented languages that don't have a concept of interfaces often have to support inheritance of multiple classes.

LabVIEW has no concepts of abstract classes or interfaces.

Examples

A musical instrument class

Consider a class for musical instruments. Since all of the musical instruments can be played the musical instrument class contains a play method. Since musical instrument as an abstraction cannot be actually played, the play method is made as an interface method. Since musical instrument class contains an interface method, the class it self is abstract and no objects of the class can be created.

A piano class is derived from a musical instruments class. Since piano is a real instrument, it can also be played. Therefore we implement the play interface for the piano class to play an instrumental piano piece. For a violin class we implement the same play interface but now it's programmed to play an instrumental violin piece.

In LabVIEW the class user can create an object from a musical instrument class and even call the play method of the class. The abstract classes help class users not mistakenly creating objects from the class which is meant only as an parent class for a certain group of objects.

An interface for data acquisition instrument

Consider you have multiple kinds of data acquisition instruments. You want to hide the implementation of the data acquisition instruments behind a common interface. You create an interface class which has the methods read data and write data.

Now consider you want to use data file to simulate data acquisition instrument. The read data method should read data from the file and write data method should write data to the file. Since you already have a File class, you want to inherit your interface class from the file class.

So you create a class called File acquisition and inherit from file class. You then define that the File acquisition class implements the data acquisition instrument interface. You add the methods read data and write data to read data to take care of the file IO.

LabVIEW doesn't allow this kind of functionality.

3 Proposal for Alternative Implementation of Object-Oriented Programming in LabVIEW

In current chapter we propose an alternative model for implementing object-oriented programming model in LabVIEW.

3.1 Initialization and Cleanup

3.1.1 Constructors

All LabVIEW class objects must always be initialized automatically so that invalid objects can exist only when an error in the initialization occurs. We propose each LabVIEW class has zero or more class developer defined constructors. If there are no class developer defined constructors, LabVIEW generates a default constructor for the class (Figure 1).

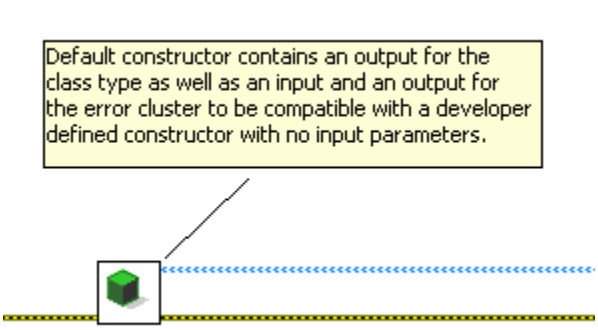

Figure 1 LabVIEW generates a default constructor if class developer has not defined any non-default constructors.

Constructors are always naturally polymorphic (Figure 2). There is not a separate polymorphic VI for the constructors but the polymorphism is built in to the lvclass file. This ensures that class developer can add new constructors as needed so that old programs using the class won't break up.

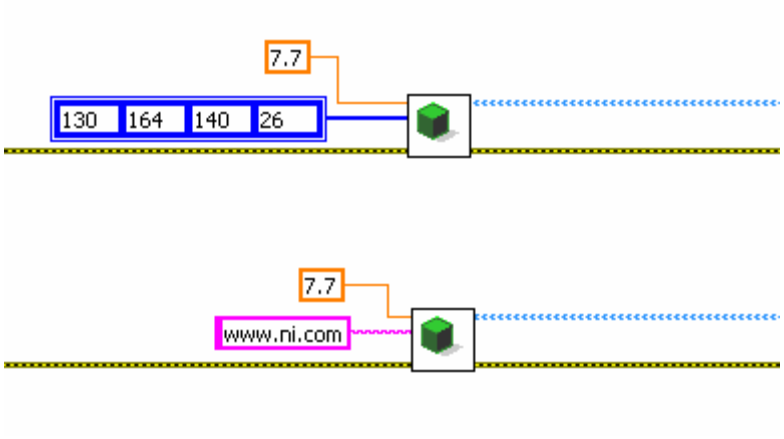

Figure 2 Constructors are always polymorphic. There is always one or more constructors for each class.

Creating objects

An object of a specific class is created by placing that class (lvclass file) on the block diagram. Placing the class on the block diagram inserts on of the class constructors on the block diagram. If class has no class developer defined constructors, a LabVIEW generated constructor is placed on the block diagram instead. The class user can select which of the many constructors to use either by selecting the appropriate constructor i) from the context menu of the constructor node (Figure 3), ii) by the type of the wire connected to one of the inputs of the constructor or iii) by selecting from the drop down menu below the constructor node.

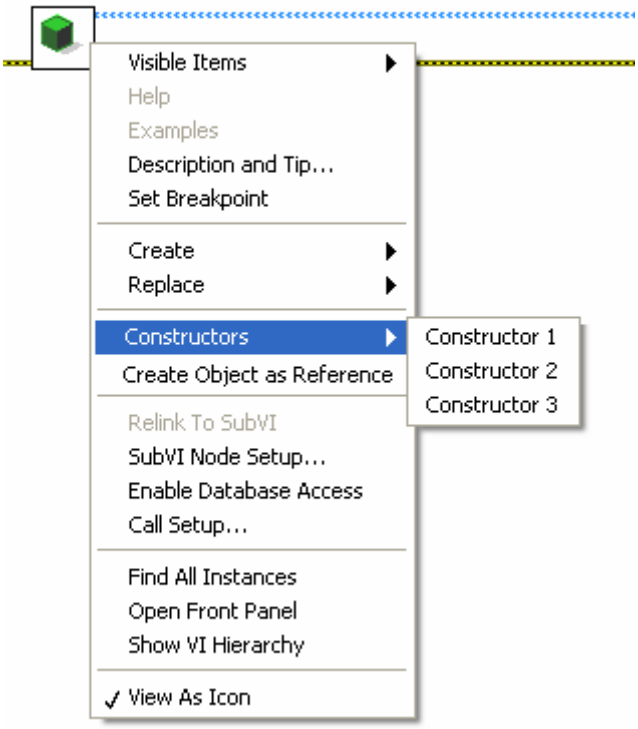

Figure 3 - Selecting one of the many constructors for a class to be created from the context menu.

If a class is placed on the front panel or one of the clusters or non-empty arrays on the front panel contains one or more classes, the VI can't be independently run, but must be called as a subVI. Each of the above mentioned front panel objects must be connected via the connector pane to avoid existence of non-initialized classes.

Creating object references

Each object is created as a wire of the object type by default. However, class user may select to create an object as an object reference instead. Class user can select from constructor node context menu on the block diagram if the object should be created as a reference (Figure 4). Additional constructors are not needed to create objects as

reference. LabVIEW will automatically create an object of the proper type using the same constructors.

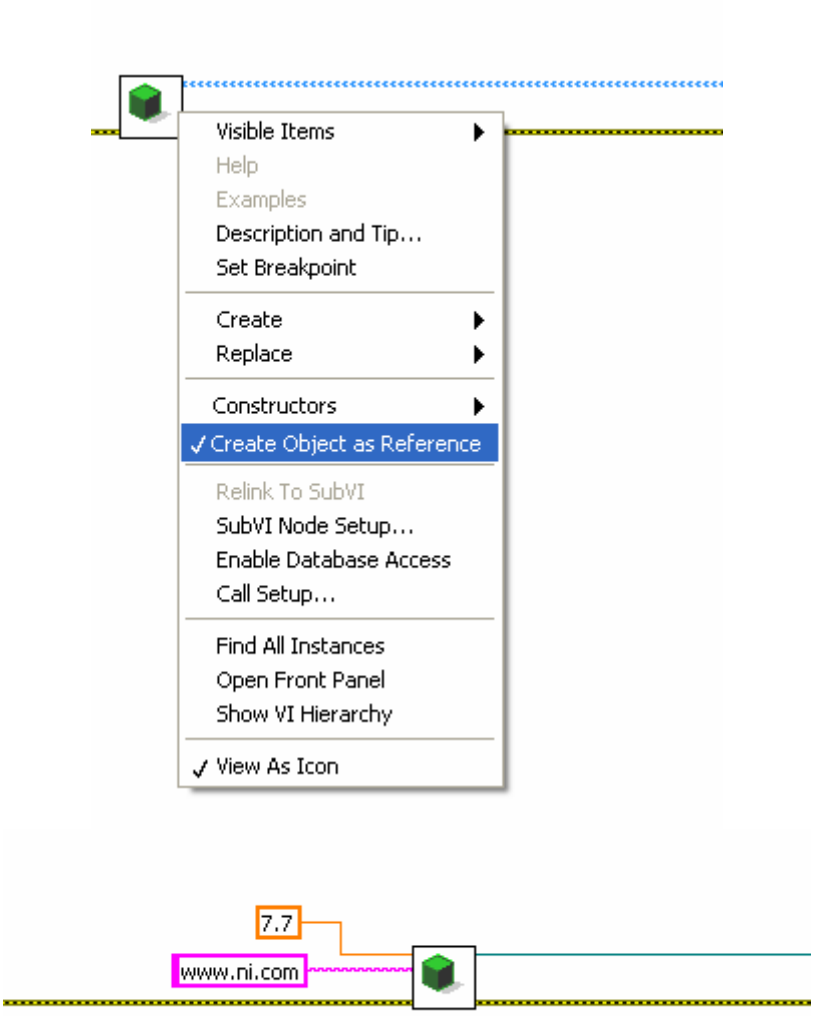

Figure 4 Creating objects as reference. The class user can select from the context menu if the object should be created as a reference.

The lifespan of the object created by reference is fully controlled by the class user, with the exception that on application exit all objects are properly disposed. The lifespan of the normal objects is controlled by LabVIEW as described later.

Creating Constructors

Class developer can create non-default constructors by right clicking on the class icon on the project browser and selecting *New* → *Constructor*. Constructors have a special icon on the project browser. Each constructor is created to a separate *Constructors* folder (Figure 5). *Constructors* folder is always present in each class even when no non-default constructors exist. The class developer doesn't need to add constructors to a polymorphic VI, instead lvclass file acts as a kind of polymorphic VI instance for the constructors.

When class developer creates a new constructor, LabVIEW generates a special constructor VI, with special block diagram and front panel properties.

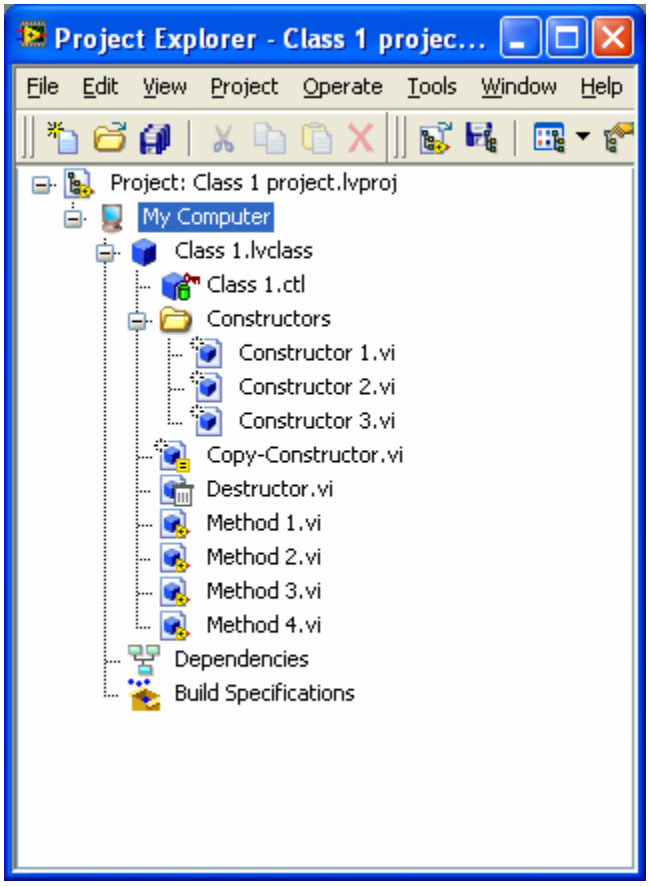

Figure 5 Constructors are created to a separate *Constructors* folder.

Constructor Block Diagram and Front Panel

Constructors have a special block diagram and front panel properties. Each constructor has three fixed connectors: an object out indicator and error in and error out clusters. These controls and indicators cannot be removed or disconnected. Constructors may have any number of class developer defined inputs but no class developer defined outputs.

On the block diagram each constructor has a class constant of the class type. Class constant member variables are initialized to the values defined in the class control. However parent class and member classes are not initialized and must always be initialized by calling appropriate constructors in the constructor block diagram. Constructor fails to compile if the wire to Class out indicator contains any non-initialized objects; only exception is zero-sized arrays containing some class instances.

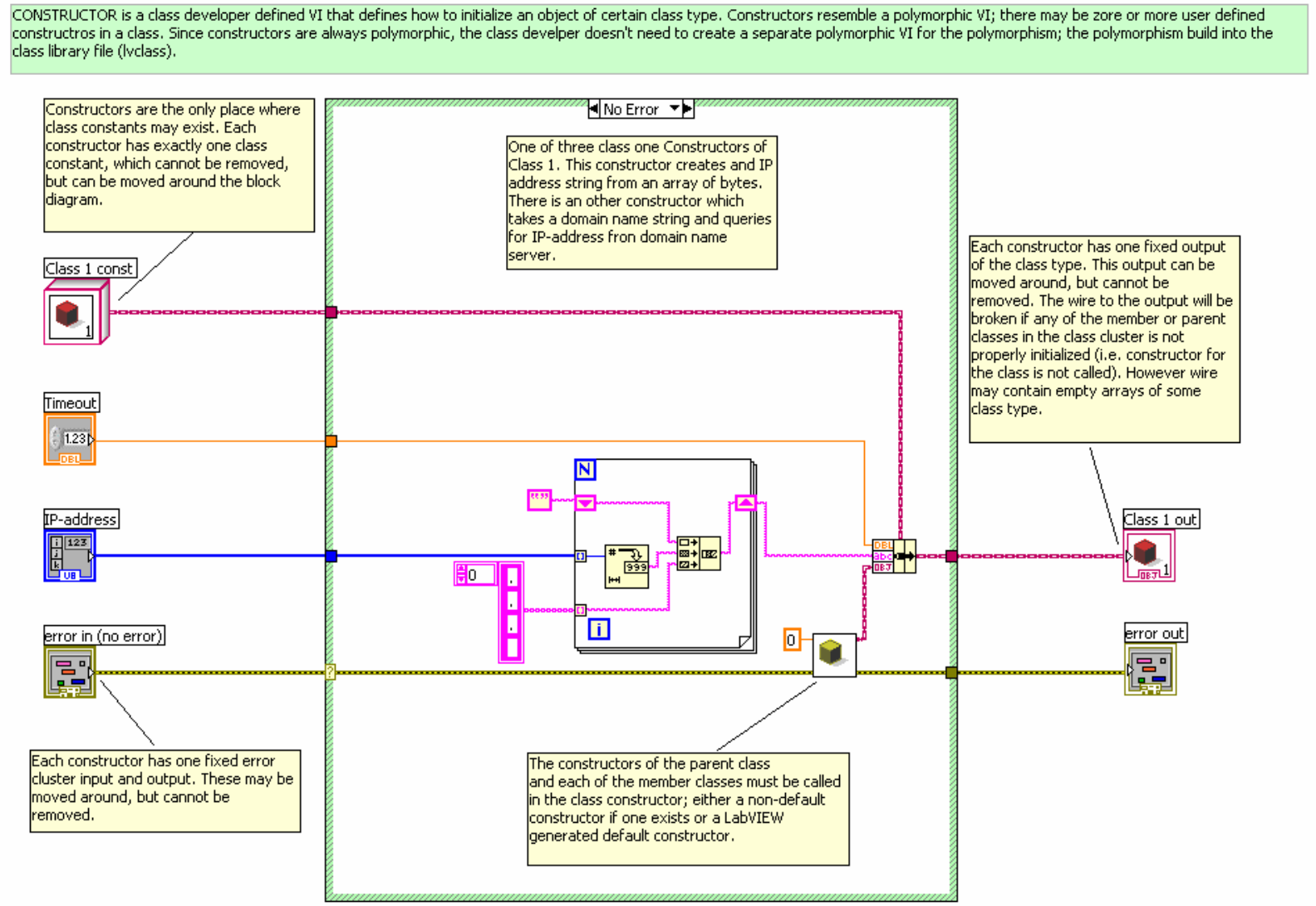

Figure 6 Constructor block diagram. Constructors have a fixed class constant (top-left), fixed class output (middle right), and fixed error in and error out clusters. Parent class constructor must be called in the constructor block diagram. Also all member classes should be initialized in the block diagram.

3.1.2 Copy-Constructors and Creating Copies of Objects

When ever a copy of an object is made, LabVIEW ensures that copies are properly initialized and as such valid objects. Since all pure memory copy doesn't always ensure proper initialization a concept of copy-constructors is introduced.

Creating Copy-Constructors

Class developer may create a single copy-constructor for the class. The copy-constructor is created the same way as constructors are created; by selecting *New* → *Copy Constructor* from the class context menu in the project browser. Copy-constructor has a special icon in the project browser (Figure 5).

Copy-Constructors have a special block diagram and front panel; there is an input of the class reference type for the object to be copied and an output of the class type for the copy created. There is also an input and an output for the error cluster (Figure 7). The error cluster is required since copying the class may always fail. No additional inputs or outputs are allowed.

The class user defines in the block diagram how the copy of an object should be made. The new object is usually initialized either by i) calling one of the class constructors or by ii) dereferencing the input object reference in which case a copy of all the member variables and the parent instance is created or iii) both calling one of the class constructors and copying some of the member variables and classes to the newly initialized class (Figure 7). LabVIEW will call copy-constructor of the parent class and each member class when ever required.

Creating Copies of Objects

Creating a copy of a native LabVIEW type only requires memory copy of the object. However for objects this is not always enough. If class developer has created a copy-constructor for the class, it will always be called when a copy of an object of the class type is made. Since creating a copy may always fail, the error must somehow penetrate to the VI making the object copy. There are two possibilities for the error penetration: i) either one passes the error cluster from the VI to the copy-constructor and back or ii) one implements the concept of exceptions in LabVIEW and copy-constructor throws an exception if copy-construction fails.

If LabVIEW will support exceptions, the call to copy-constructor is straight-forward. When ever LabVIEW makes a copy of an object, a copy constructor is automatically called. The call won't be visible to the class user in any way. If the copy-construction fails, the copy-constructor should throw an exception that can be catch in the VI creating the copy. This is a highly preferred way of implementing copy constructor calls. The programs remain clear and the copy-constructors are automatically called.

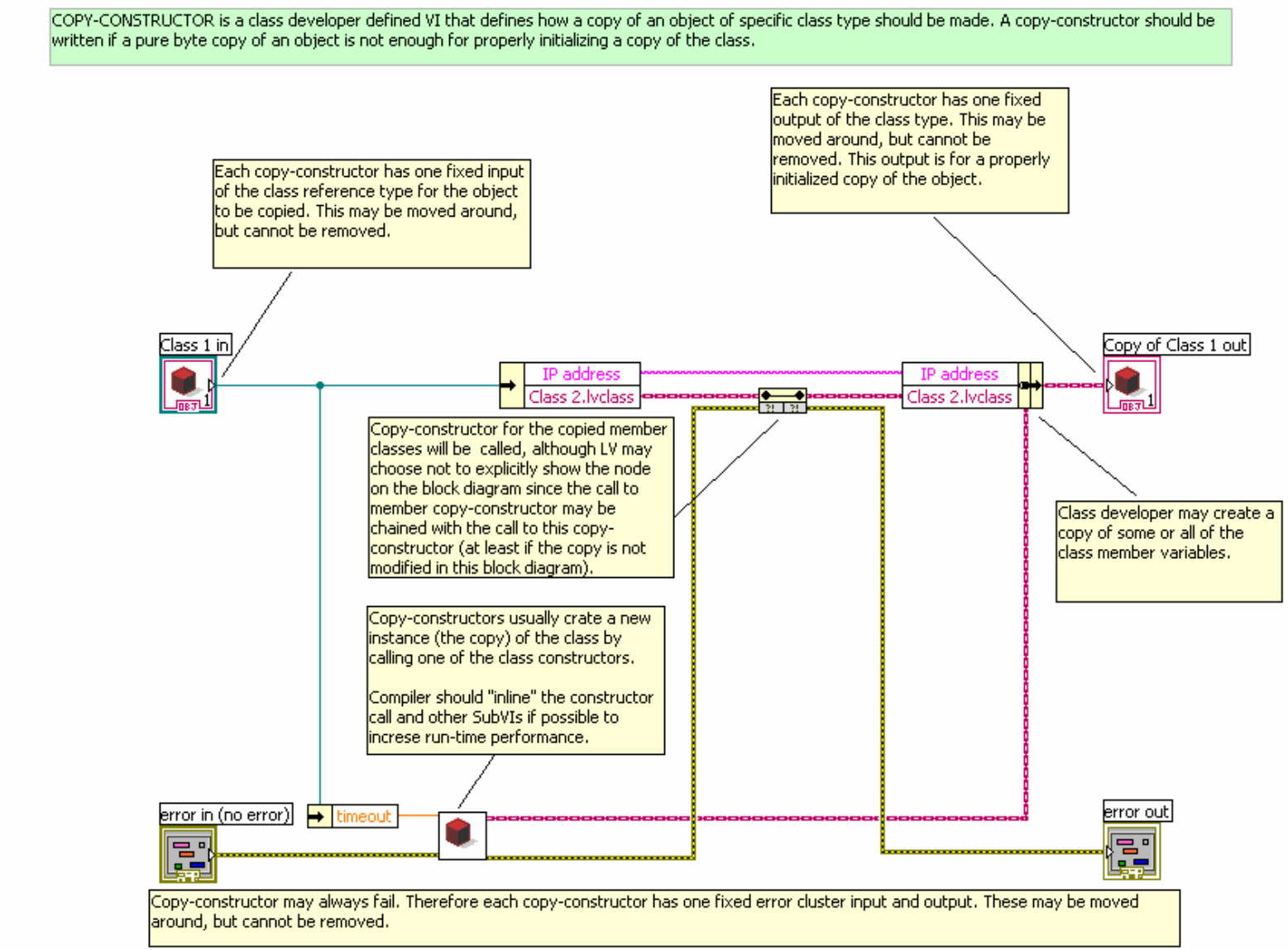

Figure 7 Copy-constructor block diagram. Copy-constructors allow class developers to properly initialize the copy of an object. The object is passed to the copy-constructor as an object reference.

If exceptions will not be supported, which is a shame since exceptions are and important for dealing with complicated errors and exceptional situations, the error cluster must be passed between the copy-constructor and the VI making the copy. We propose that in this case a new special *copy object* node will be added to LabVIEW. The new *copy object* node will look similar to *Split Signals* node (Figure 8). There will be one input for the object to be copied and one or more outputs for the newly created copies. There will also be an error cluster passing through the node.

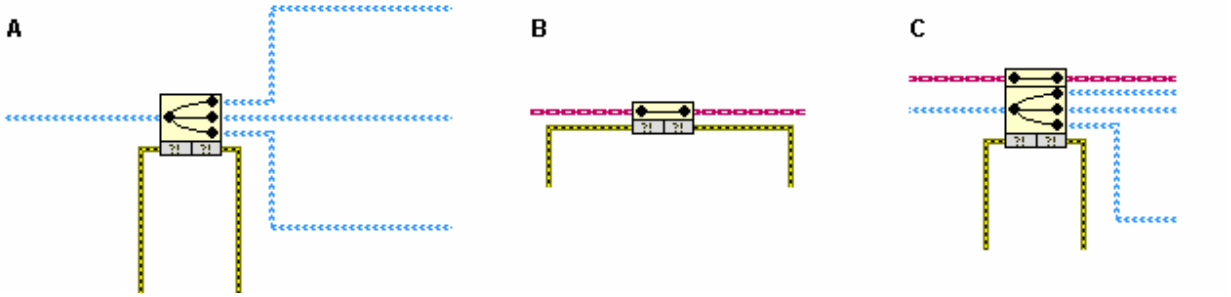

Figure 8 *Copy object* node resembles of *Split Signals* node. It has always a single input terminal of a class type and one (B) or more (A) output terminals of the class type. There is also input and output for the error cluster. LabVIEW may choose to combine multiple nodes of even different object types into a sigle node (C).

When ever a copy is made of an object that requires a call to copy-constructor, LabVIEW will place a *copy node* on the block diagram automatically and resize it to match the number of copies required (Figure 9). *Copy node* is not a traditional SubVI. It's rather a notation for the process in which LabVIEW will call all the required copy-constructors to copy the specified objects. LabVIEW may choose to combine copying of multiple objects into a single *copy node* to save space on the block diagram (Figure 8C). *Copy node* will also be created if a copy is made of a compound type i.e. cluster or array containing objects requiring copy-construction. The error cluster won't be automatically connected but the class user must take care of connecting the error cluster if needed. The *copy node* may be freely moved, but it cannot be removed. LabVIEW will automatically remove the node when no longer needed. The user may extend the *copy node* to make more copies of the object.

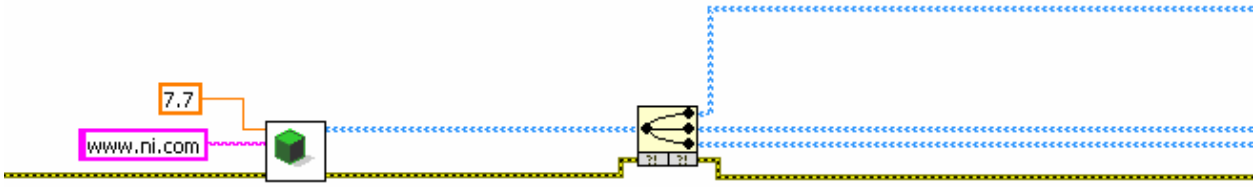

Figure 9 LabVIEW automatically creates a *copy node* when needed.

Often the copy of an object is passed to a SubVI. If the SubVI contains connectors for error in and out, then it is not necessary to place *copy node* on the block diagram of the main VI. The copy-constructor call may be chained with the call to SubVI. If an error occurs in the copy-construction, the error can be passed to the SubVI. The *copy node* is therefore necessary on if the copy is passed to a node without error cluster input and output.

Copy node and copy-construction call is not necessary if a wire branch is created but the copy is used only for reading. Then the LabVIEW runtime can schedule the reading operations prior to any modification to the object buffer.

Creating Copies of Object References

If user creates a copy of an object reference, the object is not copied; only the reference is copied (Figure 10). The same object reference may then be passed to more than one place at a time. All the copies of an object reference refer to the same object.

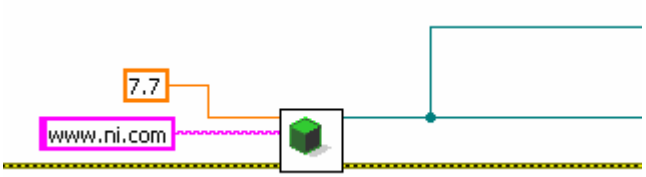

Figure 10 Creating a copy of an object reference does not call *copy node*.

To create a real copy of an object one has a reference for, *copy node* can be manually placed on the block diagram (Figure 11). On call to manually placed copy-node LabVIEW automatically dereferences the object reference, creates a copy of an object by calling the appropriate copy-constructors if necessary, and returns a reference to the copied object. Manually placed *copy node* can be removed normally.

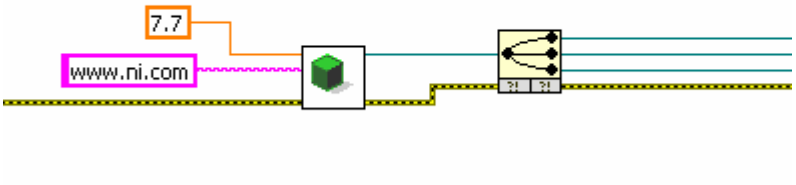

Figure 11 One can create a copy of an object one has a reference for by manually placing *copy node* to the block diagram.

3.1.3 Destructors and Cleanup of Objects

When an object needs to be disposed, LabVIEW will call the object destructor; either a class developer defined one or LabVIEW generated default destructor. The object destructor takes care of required cleanup procedures like flushing buffers, closing files and network connections etc. After destructor calls LabVIEW will release the allocated memory.

The destructors of the member objects and the destructor of the parent class will be automatically called after the destructor of the object has exited. The order of destructor calls is important so that the member objects and the parent class part of the object are valid throughout the life time of the object. The object can be considered disposed only after its destructor has finished running.

Object Lifespan

A destructor is called when object is no longer used. For a normal object, the object will be disposed when the wire terminates (Figure 12). The disposal and destructor call may not be immediate, but LabVIEW may select the best appropriate moment for disposal of an object.

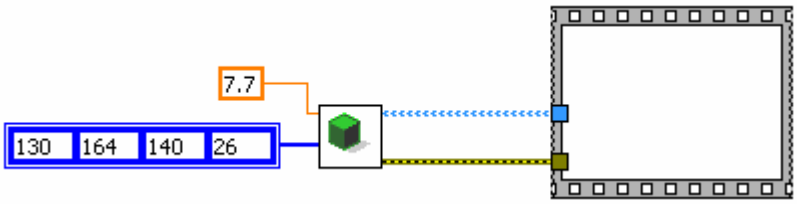

Figure 12 Object is disposed after the wire terminates.

There are some occasions when immediate destructor call is mandatory. For example the remote connection may need to be terminated immediately for security reasons. For this purpose the user can request LabVIEW to dispose an object immediately by placing a *dispose node* on the block diagram. Dispose node always calls the class destructor immediately (Figure 13).

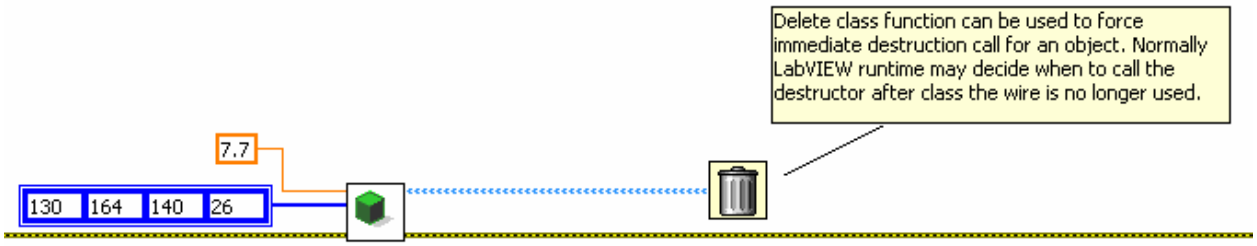

Figure 13 Dispose node can be used to force LabVIEW to immediately call class destructor.

Object references are not disposed automatically. Instead the user must manually define when the objects should be disposed. The use must ensure that the object is no longer used in any other place when requesting an object reference disposal. Object references are disposed with the same *dispose node* that is used for immediate disposal of normal objects. Object references are disposed immediately unless the user selects delayed disposal from the context menu of the *dispose node*.

After all running VIs have exited, the remaining object references will be automatically disposed and the appropriate destructors will be called by LabVIEW runtime engine.

Creating Destructors

Class developer may create a single destructor for the class. The destructor is created the same way as constructors are created; by selecting *New* → *Destructor* from the class context menu in the project browser. Destructor has a special icon in the project browser (Figure 5).

Like constructors and copy-constructors, destructors are special kinds of VIs. Destructors only have a single input of class reference type for the object to be disposed and no outputs. Since destructors may never fail, there is no error cluster input or output in the destructors. The destructor block diagram is illustrated in Figure 14.

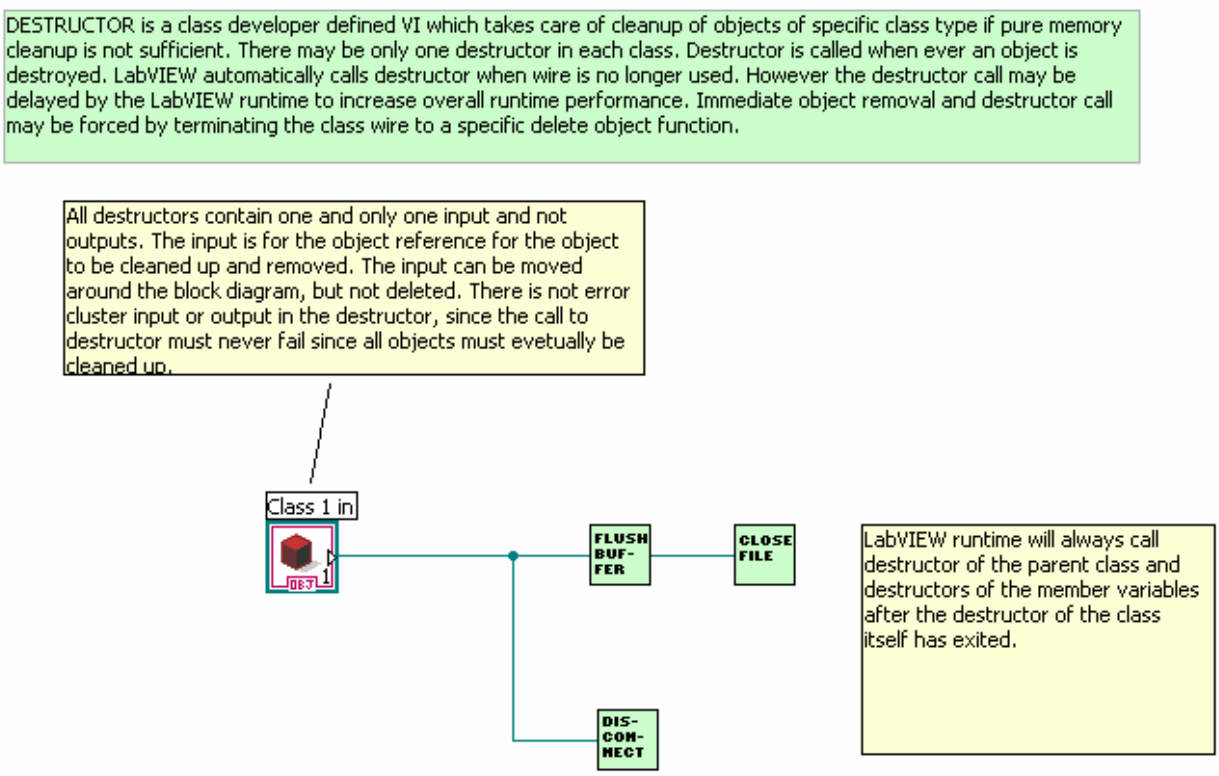

Figure 14 Destructor will take care of cleanup of objects. Each destructor contains an object reference input for the object to be disposed an no other front panel components connected to connector pane.

3.2 Object Oriented Programming Model

3.2.1 Object References

As discussed in section 3.1, there are two ways to use objects on the block diagrams; object wires and object reference wires. To complete the programming model LabVIEW should contain nodes for creating object references from objects and dereferencing object references back to objects (Figure 15). The object reference wires are strongly typed. They always represent objects of only one type, so there is no need to explicitly specify the class type when dereferencing an object reference.

Figure 15 Creating object references with *reference node* and dereferencing object references with a *dereference node.*

When object wire is branched, a copy of an object is made by calling copy-constructor if necessary. When object reference wire is branched, all the wire branches refer to the same object (Figure 16).

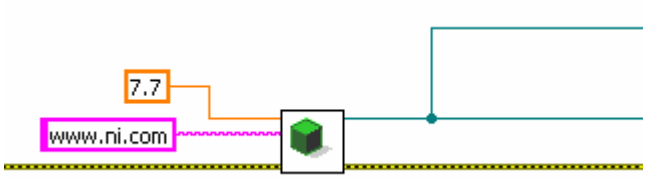

Figure 16 Copy of a reference refers to the same object as the original object.

However when a reference wire is branched so that there are multiple wires referring to the same object and later dereferenced, a copy of the original buffers needs to be made since in dataflow model two buffers can never be the same object (Figure 17).

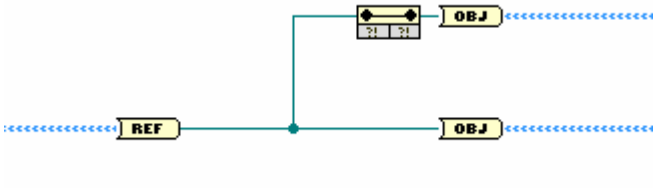

Figure 17 When there are multiple references to the same object and one of the references is dereferenced then a copy of the object is created and a copy-constructor is called when ever necessary.

3.2.2 Passing Objects to Class Methods, SubVIs and Other Nodes

The object references can be used in identical way as objects themselves. All nodes that primarily take an object as an input can take an object reference instead. This includes the cluster bundle and unbundled nodes. LabVIEW dereferences the object reference before it's passed to the node. Similarly all nodes that take an object reference as an input can take an object as an input instead. LabVIEW automatically creates a reference of an object and passes it to the node (Figure 18).

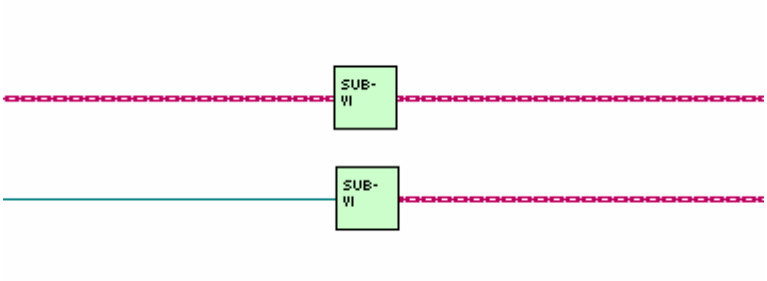

Figure 18 All nodes can accept both the class type and class reference type objects as inputs. LabVIEW automatically transforms between the types.

There is an exception to the above mentioned behavior when passing objects or object references to class methods. The class out indicator is automatically adapted to the same type as the wire connected to the class in connector of the method (Figure 19).

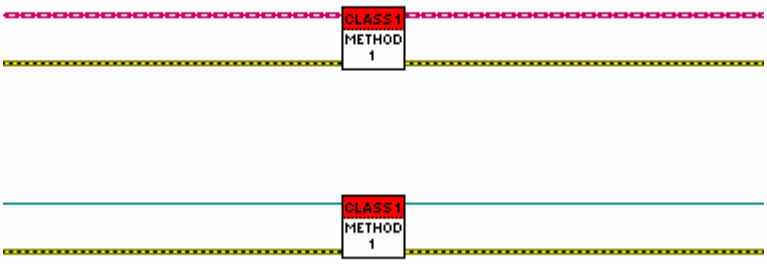

Figure 19 Passing objects to methods. The method can accept both the class type and the class reference type objects. The class out connector is adapted to the type of input wire.

3.2.3 Creating Class Methods

Unlike in the LabVIEW 8.20, the class methods *Class in* and *Class out* connectors are of type class reference, not of class type (Figure 20). Class reference is more natural and safer way to refer to one specific object instance inside class methods. If objects were used instead of object references, it would be too for the class developer to create a copy of the object inside the class method. The class method could therefore return a copy of the original object, not the original object. This is definitely not the intension. Since all nodes accept the class reference wire, the programming with the class reference objects is as easy as programming with class objects.

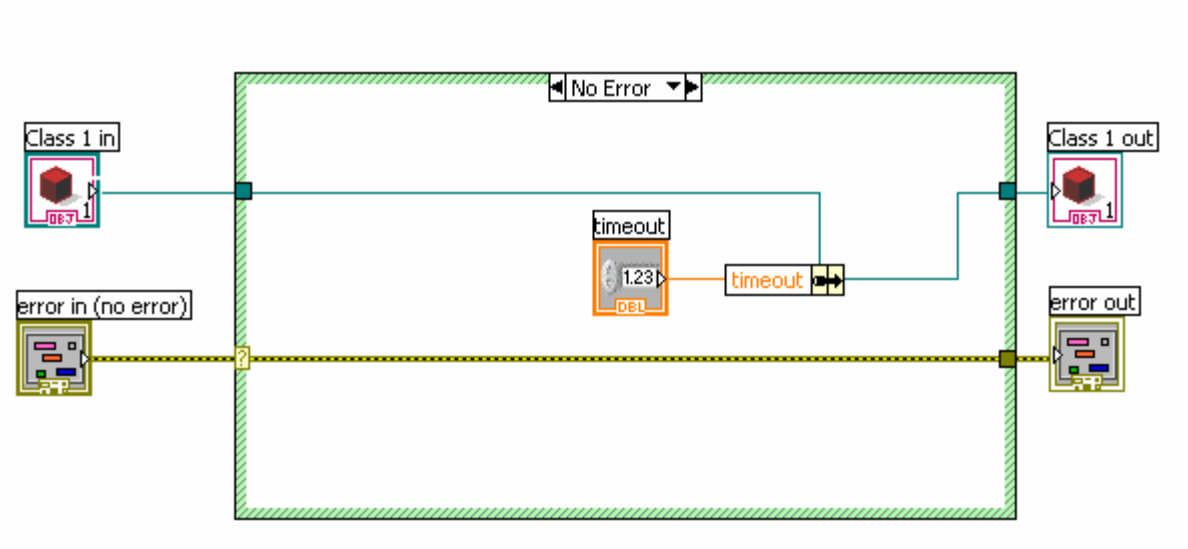

Figure 20 Class methods use object references, not class type itself. This ensures that the object is not copied when a method is called.

3.2.4 Typecasting

In Object-Oriented programming there is a need to cast an object to more generic and sometimes even to more specific class. To perform this, the class can be dragged and dropped inside *more specific* or *more generic* typecast node (Figure 21). For the more generic typecast there is also a context menu available in the node, since all the parent classes are always known. Connecting objects to *target class* input of the typecast nodes is not always practical since placing a constructor on the block diagram creates an object instance. Compiler could however not call constructors or copy-constructors for wires connected only to *target class* input of *more generic* or *more specific* nodes.

Figure 21 Casting a class to more generic or more specific can be performed by dragging appropriate class to the more generic or more specific node respectively.

3.2.5 Placing Methods and Property Nodes on Block Diagram

To make the programming with object easier in LabVIEW, the class methods as well as ancestor class methods can easily be placed on block diagram from context menu of the object or object reference wire (Figure 22). Likewise property nodes for object member variables can be created in similar manner. Property node can be created only to member variables for which the block diagram has access privileges to.

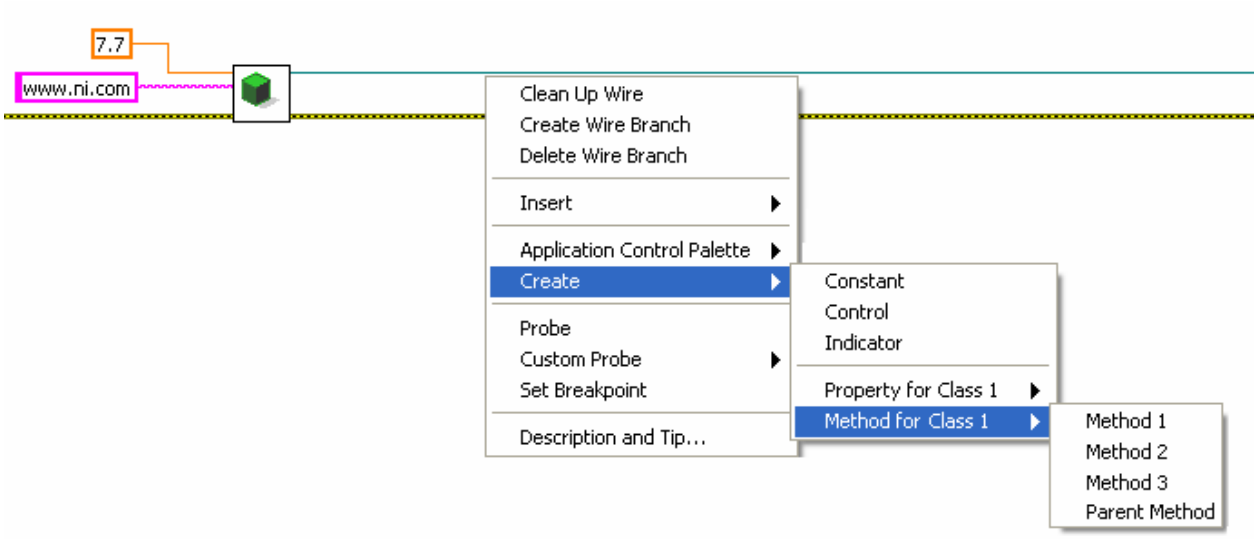

Figure 22 Property nodes for public class properties as well as methods can easily be created by selecting create from the context menu of the wire or node output terminal.

3.2.6 Member Variables

Unlike in LabVIEW 8.20, the member variables should have private, protected and public access scopes. The access scope for each variable in the class type control is selected from the context menu of each variable (Figure 23). Only top-level variables can have access scope; variables inside clusters and arrays inherit the scope of the cluster or array respectively. Public member variables are accessed trough property nodes.

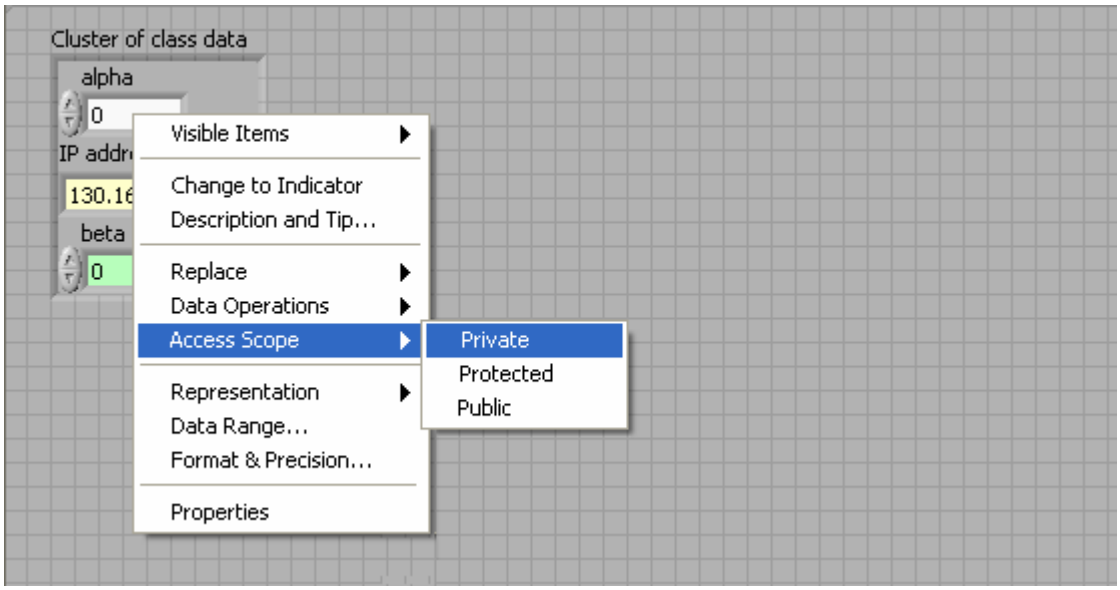

Figure 23 The access scope of member variables may be slected between private, protected and public access scopes. The default is private. The color of the variable indicates the access scope.

3.2.7 Concurrency

To avoid problems caused by simultaneous access to objects, some kind of concurrency control needs to be implemented in LabVIEW. The concurrency control would be implemented using a new *concurrency control diagram* (Figure 24). The diagram would guarantee that the objects under its control will not change outside the diagram until the diagram has exited. To avoid dead-locks there would be a timeout input in the diagram that would define the time for the diagram to wait for the objects to become available until it returns a timeout error. There would also be an input and an output for error cluster. The timeout error would be returned via the output error cluster.

A single *concurrency control diagram* can lock any number of objects or even arrays of objects. The objects to be locked are specified with special lock tunnels of the *concurrency control diagram.* The user can add new lock tunnels the same way as shift registers are added to loop diagrams.

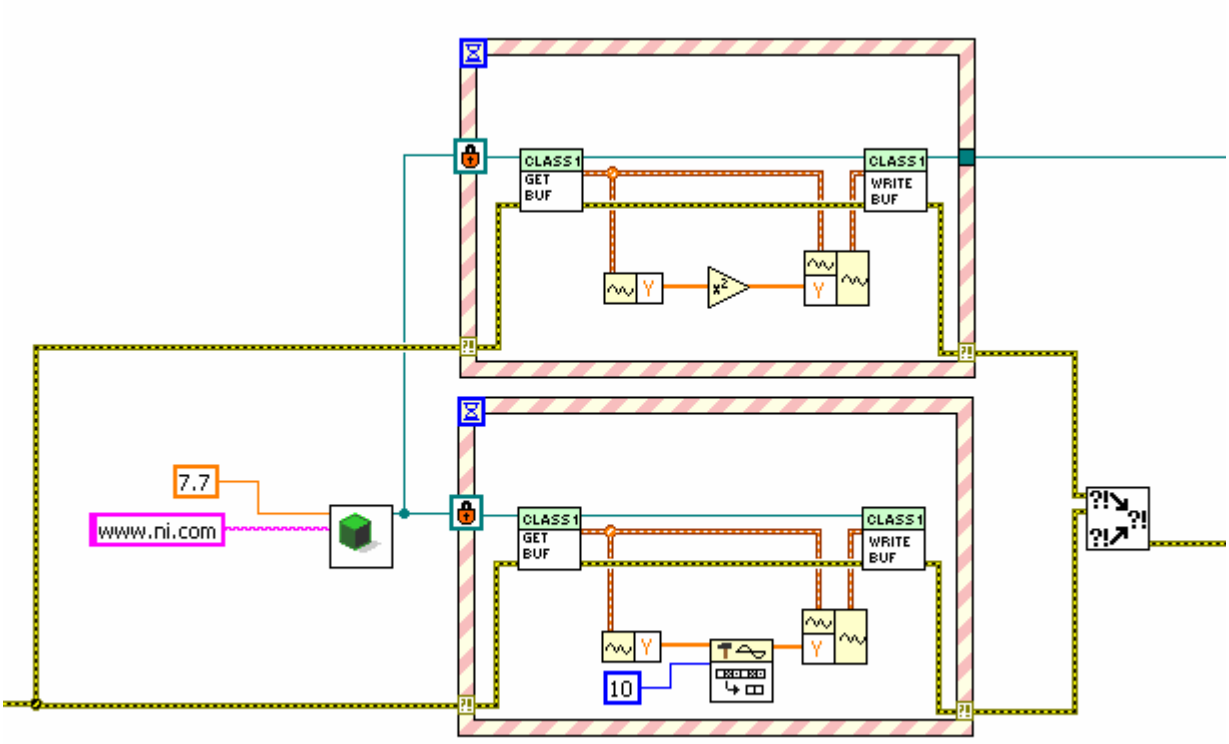

Figure 24 Locking an object using new concurrency control diagram. One of the diagram need to wait until the other diagram has exited.

The concurrency control diagrams could be nested one inside another even for the same object (Figure 25). In addition the nodes inside the diagram may also contain concurrency control diagrams for an already locked object. The inner *concurrency control diagrams* as well as *concurrency control diagrams* inside nodes on the outer *concurrency control diagram* will get a *subprivilege* to the object. *Subprivilege* guarantees that object is not modified outside the innermost *concurrency control diagram* until the diagram has exited. Thereafter other nodes and diagrams on the outer *concurrency control diagram* may gain an access to the object.

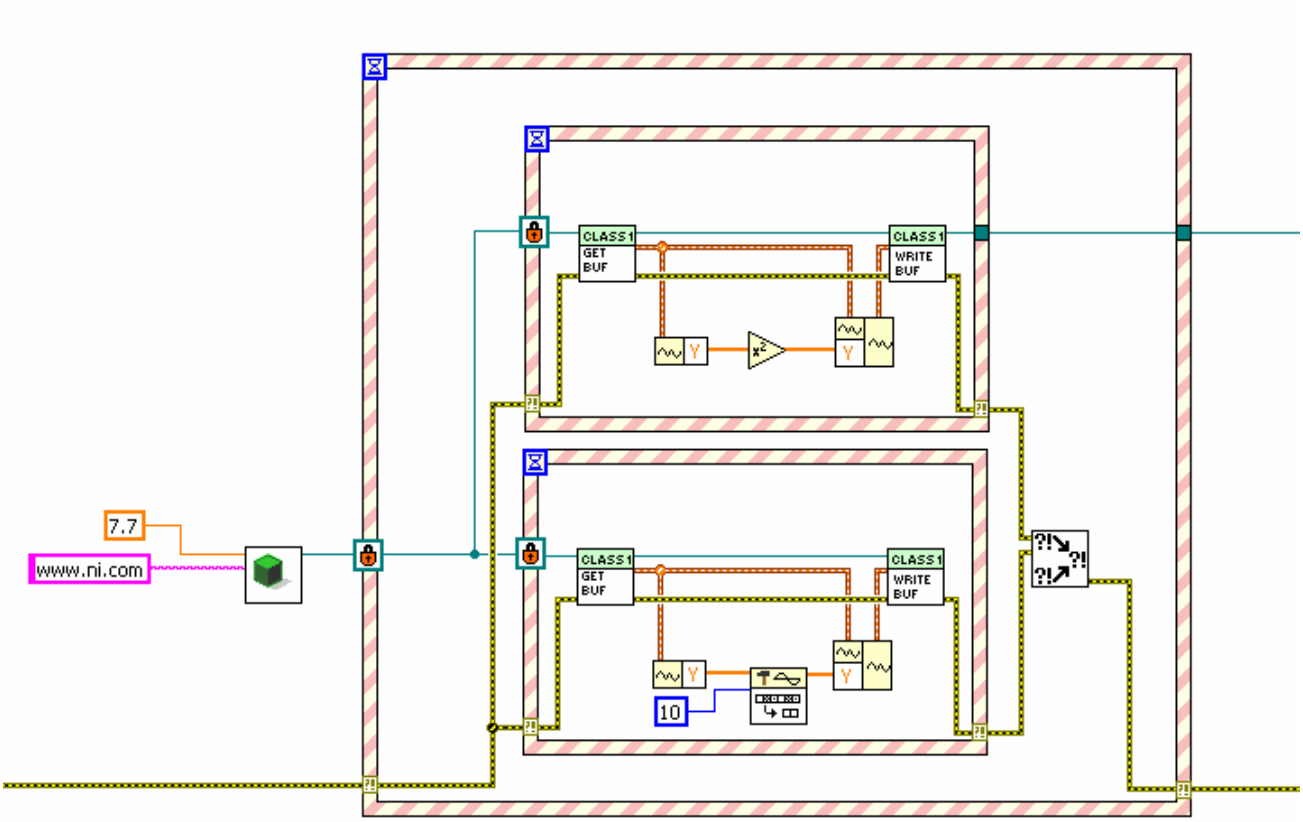

Figure 25 ***Concurrency control diagrams*** **can be nested one inside another. The outer diagram gets a privilege for the object for all of its nodes and diagrams. The inner** ***concurrency control diagrams*** **may get a subpriviledge over the other nodes and diagrams inside the outer** ***concurrency control diagram*****.**