Graphical Object-Oriented Programming In LabVIEW

Jörgen Jehander
Endevo
www.endevo.se

Introduction

A problem today in too many LabVIEW programs is that they are expensive to maintain. Fixing bugs or adding new functionality to an existing program is often difficult and time-consuming. There are two main reasons:

1. The programs are often designed in a traditional top-down design where a program is composed of a hierarchical tree of subVIs. During the design, not enough consideration is given to how the subVIs should reflect the problem domain. Many programs consist only of the GUI layer – the application layer is forgotten. In many cases, a subVI is created only to downsize the code in the higher-level VI.
2. Application state information is implemented with global variables, which results in complicated data dependencies and race conditions.

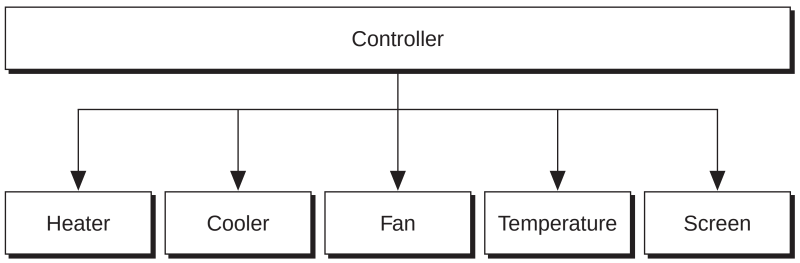
The following case study highlights these problems and suggests how they can be addressed and solved.

Case Study – A Climate Control System in LabVIEW

Description

The embedded version of LabVIEW, LabVIEW RT, was chosen as the platform for a climate control system. The climate control system consists of the following parts:

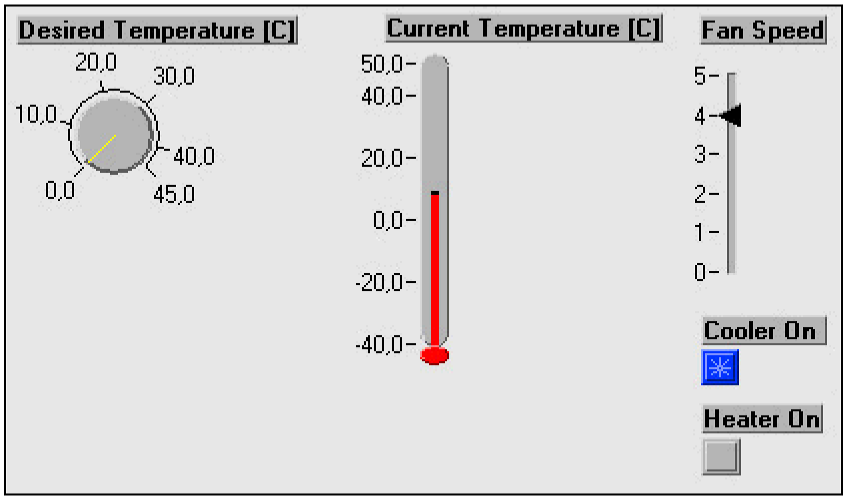
- Pentium-based data acquisition board with integrated analog and digital I/O
- A 9 in. VGA flat screen
- Heater
- Cooler
- Fan
- Temperature Sensor

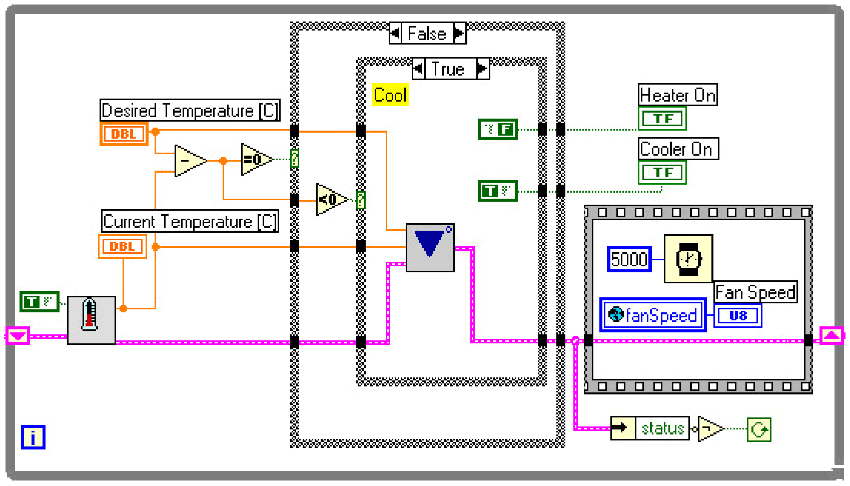


The heater and cooler are both turned on and off with digital signals. Analog signals are used to inform the heater or the cooler how many degrees the air needs to be heated or cooled, respectively. The fan has five different speeds controlled by digital signals. From the temperature gauge the current temperature is sent from the car by an analog signal. A user controls the system by the user interface provided by the LabVIEW program. The first release of the program looked like this.

LabVIEW Solution

The picture below represents the graphical user interface (front panel) of the main VI (Climate Controller.vi) panel. A knob in the upper left corner controls the desired temperature. On the right side, current system information is displayed.





As seen from the block diagram above, both the handling of the GUI and the application logic is managed in the same VI. Handling of the GUI is done by the code that manages validation of user inputs and displays information to the user. The application logic is the collection of different rules that control the application.

In the example of the climate control system, the application logic consists of rules that decide when to start heating or cooling. Having the **GUI handling** and **application logic** in the same VI establishes hard dependencies between these two tasks. Unfortunately, many LabVIEW programs are structured this way because applications are often developed by starting with the GUI and adding application logic as needed. This strong coupling between the user interface and the application logic makes it difficult to maintain the code and causes the scalability (incorporation of new features or distribution of work) of the application to become expensive. Additionally, strong coupling between VIs makes it often impossible to reuse many pieces of the application in other projects.

Scalability

The functionality of the application should not be constrained by the interface it presents to the user. If it becomes necessary for an application to be changed to increase functionality or performance, it should impact the existing GUI as little as possible.

One way of increasing the performance of an application is to distribute its work among several machines. For instance, data acquisition might be performed on one computer, computationally intensive analysis and processing on another, while yet another computer archives results in a database. The whole system might need to be controlled or monitored from a number of other computers. Often the monitoring GUI can be run on a less powerful computer than the main parts of the application. A measurement server and several remote GUI clients running inside a Web browser is a good example of how this concept can be applied in a test and measurement system.

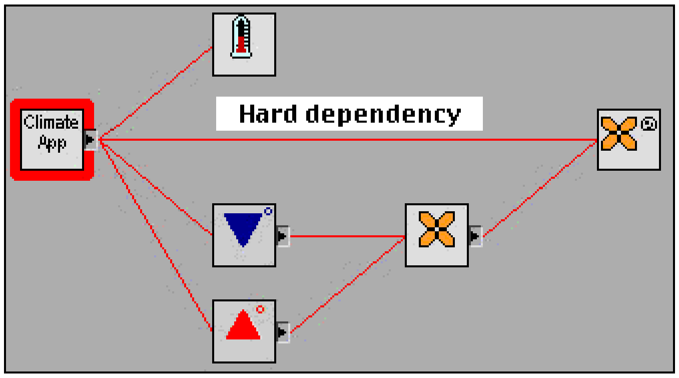
The key factor for being able to distribute a system among several machines is to isolate key parts of the application from one another and to decouple the GUI handling from the application logic. A poor initial design where the GUI handling and the application logic are intertwined will require a major redesign when additional needs arise. By isolating the individual parts of the application, on the other hand, it can evolve gracefully over time.

Maintainability

As the application logic or the GUI grows more complicated, the code becomes even harder to understand and modify if both the GUI handling and the application logic are in the same VI or intertwined throughout the VI hierarchy. Making changes to GUI handling (to fix a bug, for example) should not affect the application logic or vice versa.

Often the change cycles for the GUI and the application logic are different – in some cases you may even need multiple GUIs for the same application logic. If the code of both the GUI and the application logic is located in the same VI, they cannot be separately changed according to their own change cycle. Similarly, multiple GUIs would not be able to reuse the application logic if it is part of their top-level VI.

In the block diagram code below, a global variable is used to pass the current fan speed, as calculated by the application logic, to GUI. Using this global variable directly in the GUI-handling code creates a hard dependency by coupling the implementation of the GUI to the internal implementation of the Fan VIs. If, at some point in the future, the Fan VIs need to change the representation of this global variable from integer to string, it will cause the GUI to have to change its code as well. As a matter of fact, all users of this global variable would have to be modified to support the representation change.



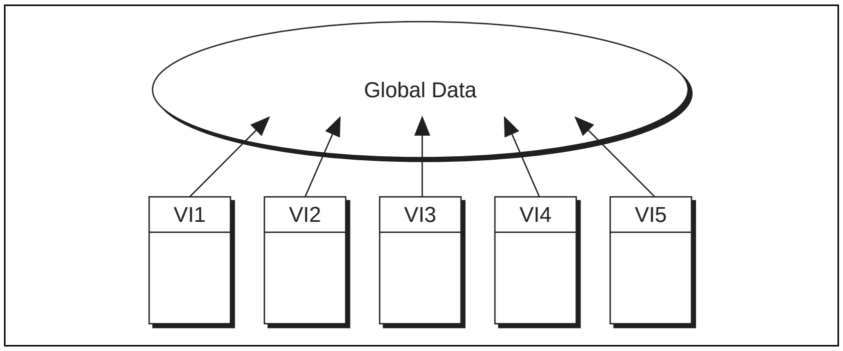
Reusability

Using a global variable to store the state information of the Fan makes it difficult to reuse the Fan VIs within the same application. If another fan were required for the Climate Control system, a separate global variable and corresponding set of VIs would have to be created.

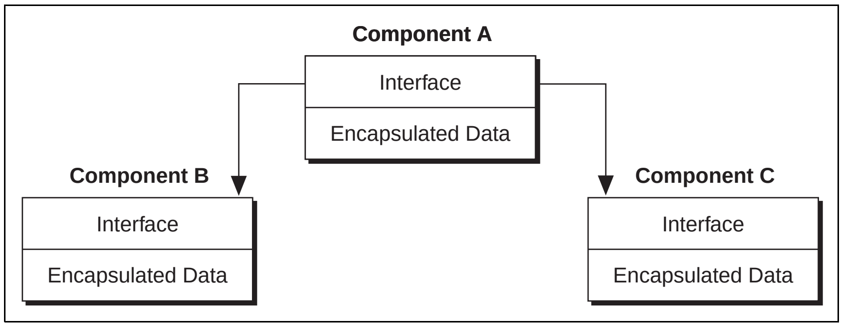
Although the Fan VIs, along with the global variable, could be used in separate applications, the same maintainability issues would cause problems in these applications if the representation of the global variable were changed.

Monolithic LabVIEW Programs

The picture below illustrates a LabVIEW program in which state information is stored in global variables accessed by five VIs. Hard dependencies exist between the VIs and the global variables. If any one of the VIs needs to change the representation of the global variables, it may require the other VIs to be modified because of the change.



Because of these dependencies, this **monolithic** program is very "static" and difficult to maintain and expand. A better way to design an application is to divide a monolithic program into separate pieces or components. Building component-based programs is similar to using Lego bricks, which fit together nicely.



A component can be thought of as a mini-application that has a specific interface describing its functionality. Each component encapsulates its state information, hiding the implementation from the user of the component. This encapsulation removes the hard dependencies that arise when global variables are directly accessed throughout an application. A component can provide access to its state information through functions or methods that will remain the same even if the internal representation of the state data is changed. Because users of a component access its information through methods, any change to the representation of the state information requires modification of only the access method instead of modification of all VIs that use the component.

Components can be designed to reflect the problem domain of an application. The parts that constitute the application problem will be the same as the components of which the program is comprised. When components are designed to reflect the problem domain of an application, the readability, maintainability and quality of the system is improved.

Using a component-based architecture, you can evolve your system over time as new components replace older components or add functionality. Your application will be scalable and easier to maintain.

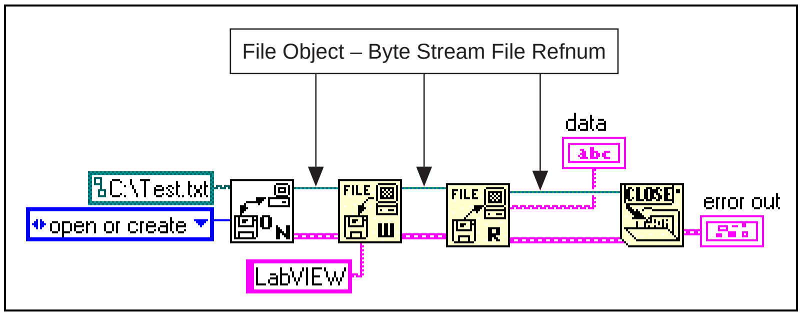
Existing Components in LabVIEW

You can look how file I/O, networking and ActiveX automation are used in LabVIEW to see how components are applied.

In LabVIEW, the following four functions are used to access files:

- **Open File** – Creates a file object and opens the file
- **Write File** – Writes data to the file using the file object created
- **Read File** – Reads data from the file using the file object
- **Close File** – Closes the file and deletes file object

A simple program that writes the text “LabVIEW” to a file and then reads it back is shown in the following picture.



Open File opens the given file and returns a unique reference associated with the file opened. This reference is then used in the **Read File** and **Write File** functions for reading and writing to the file. Finally the reference is passed to the **Close File** function to close the file. The **Read**, **Write** and **Close** functions require the reference in order to know which file to operate on. Each file opened has some state information associated with it, such as the size of the file and the read-offset into the file. The file indirectly references this data (similarly to a point). Because the file reference encapsulates this internal state information, you can access this data only through the file I/O functions.

Looking at the file I/O functionality as a component, we would view the four functions as the methods that represent the interface to the LabVIEW File component. Access to the state information of the File component is possible only through these methods. The state information used by the methods is identified by references, referring to File objects (instances of the File component data). We use the term *object* to mean *instance of a component*.

Graphical Object-Oriented Programming, GOOP

Classes

Parts from which an object-oriented system is built are expressed as classes. In the climate control system we can identify the following pieces:

- Pentium-based data acquisition board with integrated analog and digital I/O
- A 9 in. VGA flat screen
- Heater
- Cooler
- Fan
- Temperature Sensor

You can improve the readability of your program by using classes that express the same entities as the ones identified in your application because the individual parts of the problem description are directly reflected in the program.

The class describes the two aspects of each entity:

1. Public methods defining its behavior and responsibilities – what can be done with it?
2. Private data defining state information that is required by the public methods.

Looking at the Fan entity will help us illustrate the class concept.

Class name: Fan

Public methods:

Set Speed

Get Speed

Private data:

Speed

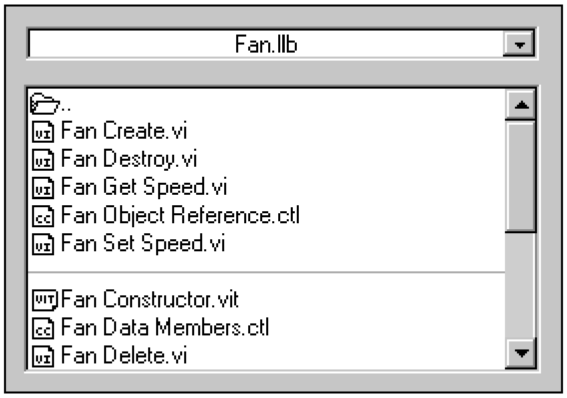
With the two methods, you can set the speed of the fan and find out the current speed. The Fan remembers the current state information (speed setting) in the *speed* private data.

The terms public and private denotes the access rights. The public methods are accessible to users of the class while the private data is not. Because users of the class can manipulate the Fan object only through the public methods, they will not be affected if the representation of the private data is changed.

A C++ declaration of the fan class may look like this:

```
class Fan {
        public:
                Fan()
                ~Fan()
                unsigned char getSpeed();
                void setSpeed(unsigned char speed);
        private:
                unsigned char speed;
};
```

The same class expressed in LabVIEW looks like this:

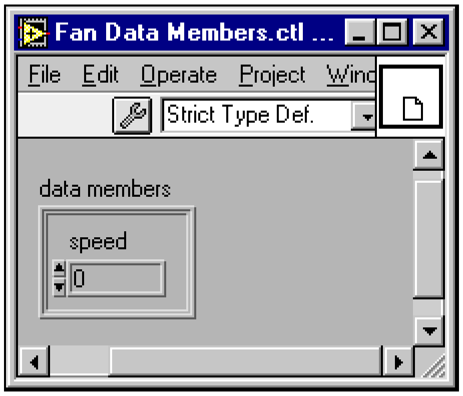


Fan.llb

..
Fan Create.vi
Fan Destroy.vi
Fan Get Speed.vi
Fan Object Reference.ctl
Fan Set Speed.vi

Fan Constructor.vit
Fan Data Members.ctl
Fan Delete.vi

An LLB contains all the VIs that implement a class. Top-level VIs (appearing above the line) represent the public methods, including **Get Speed** and **Set Speed** described above. Beside the public methods, the object reference for the fan class, **Fan Object Reference**, is also a top VI. The private class data member *speed* is defined inside a cluster and stored in the **Strict Typedef** control `Fan Data Members.ctl`.



Components

A LabVIEW component can be made up of one or more classes. The interface to a component is defined by the public methods of one of the classes from which it is implemented.

Objects

Ordinary data types and objects are similar in that they share identity and state.

The following C++ code declares two integer variables whose identity are `int1` and `int2` and creates two fan objects who are identified by `heaterFan` and `coolerFan`.

```
int int1, int2;
Fan *heaterFan = new Fan();
Fan *coolerFan = new Fan();
```

The major difference between an ordinary integer variable and an object is that an object has a *behavior*, which is defined by the public methods in the class from which it was created. Invoking these methods modifies the internal state of the object. The following C++ code sets the values of `int1` and `int2` to `1` and `5`, respectively. Invoking the `setSpeed` method will set the speed state of the Fan objects to `3` and `5`, respectively.

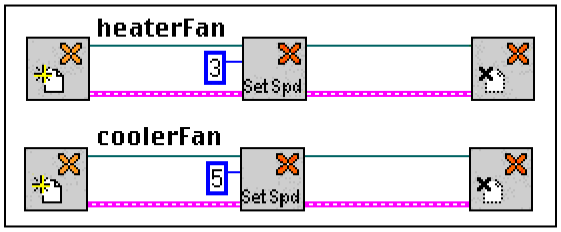
```
int1 = 1;
int2 = 5;

heaterFan->setSpeed(3);
coolerFan->setSpeed(5);
```

Because each object that was created has its own copy of the private data (speed), the method calls above will result in the state variable `speed` being `3` for the `heaterFan` object and `5` for the `coolerFan` object.

Finally, the objects are deleted when they are no longer needed, as shown in the following C++ code:

```
delete heaterFan;
delete coolerFan;
```

The following picture shows how the equivalent code creation, assignment and deletion for the Fan objects would look when implemented in LabVIEW.



The constructor method **Create Fan** creates a new fan object and returns a reference (of type **Fan Object Reference**) that represents it. The reference is passed to the **Set Speed** method to set the speed of the fan. Finally, the fan objects are deleted by calling the **Destroy Fan** destructor method. As is customary in LabVIEW, error information is passed between the VIs using the error cluster.

The usage of the Fan VIs closely resembles the usage of built-in LabVIEW components, such as the file example we saw before. In both cases you have a collection of functions or VIs that use a reference to pass object information between them.

Summary

LabVIEW contains several built-in reference-based components, such as File I/O, TCP/IP Networking and ActiveX Automation. Using the GOOP approach described in this document, you can take advantage of the benefits of object-based design and development. Adding the **class** and **object** concepts from other object-oriented languages to LabVIEW, you can build your own components.

By implementing classes, you can take advantage of the design and development practices used by developers who use standard OO languages. You will also be able to use standard OO analysis and design tools, such as Rational Rose and Microsoft Visual Modeler and apply principles described in literature about OO design and development.

The component-based approach to developing application augments the traditional data-flow programming used in LabVIEW applications. You will develop large applications by identifying and designing components that implement distinct parts of the functionality of the application. Today, components are widely used in ActiveX automation and other technologies. Having your application based on components should make it easier to interface with these technologies as LabVIEW evolves in the future.

Let’s take a look at how the case study appears in LabVIEW when it becomes component-based and implemented with GOOP.

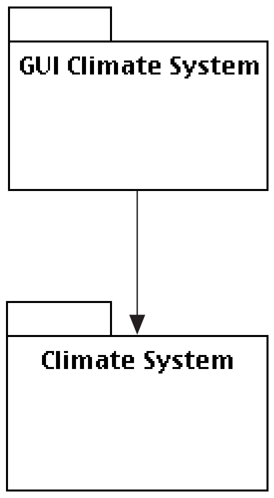
Case Study – A Component Based Climate Control System in LabVIEW

Components

The first step in component-based design is to identify individual pieces of the problem domain. In this example, a climate control system is the problem domain. The description of the climate control system identifies the heater, cooler, fan, and temperature gauge. These physical entities are natural candidates for classes because they are a natural part of the problem domain. Because these classes by themselves have no knowledge about how a climate control system works, an "intelligent" control class is needed as well. This "Climate Controller" will contain the application rules for our climate control system.

Because we want to establish a clear separation between the GUI handling and the application logic, we divide the system into two components:

- GUI Climate System
- Climate System



The **GUI Climate System** component will contain all program logic needed for validating user inputs and transfer the information to the **Climate System** component. The **GUI Climate System** will also present information from the **Climate System** component to the user.

All application logic for the climate system will be contained inside the Climate System component.

The next section describes the class from which the Climate System component is comprised. The relationship between the classes is represented in Universal Modeling Language (UML), a notation system used in object-oriented analysis and design.

Classes

The **Climate System** component is comprised of five classes:

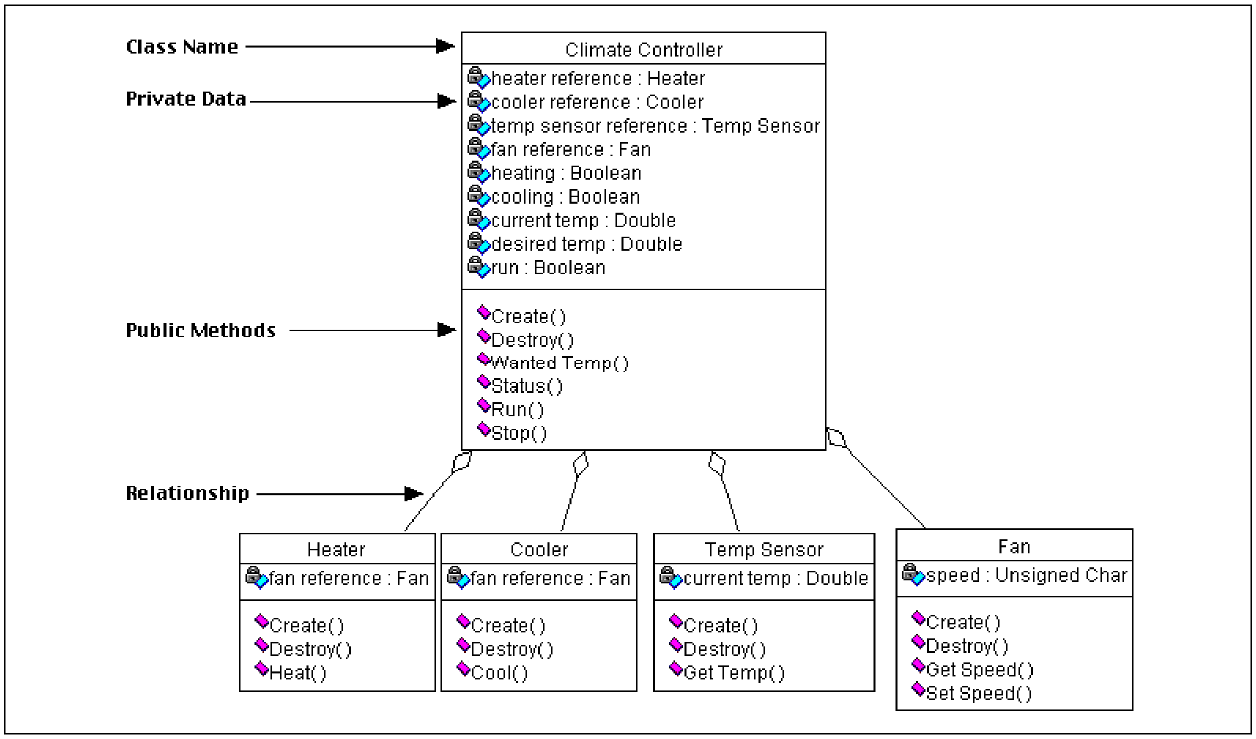
- Climate Controller
- Fan
- Heater
- Cooler
- Temp Sensor

In addition to the constructor and destructor, the Climate Controller class contains the following four methods that constitute the **interface** for the **Climate System** component.

• Wanted Temp	<< Sets the desired temperature >>
• Status	<< Returns current climate process information >>
• Run	<< Starts and executes the climate process >>
• Stop	<< Stops the climate process >>

The **GUI Climate System** component uses these methods to interact with the **Climate System**.

The following UML class diagram describes the classes that comprise the **Climate System** component and the relationships between these classes. Every rectangular box represents a class and the lines between them define the type of relationship.



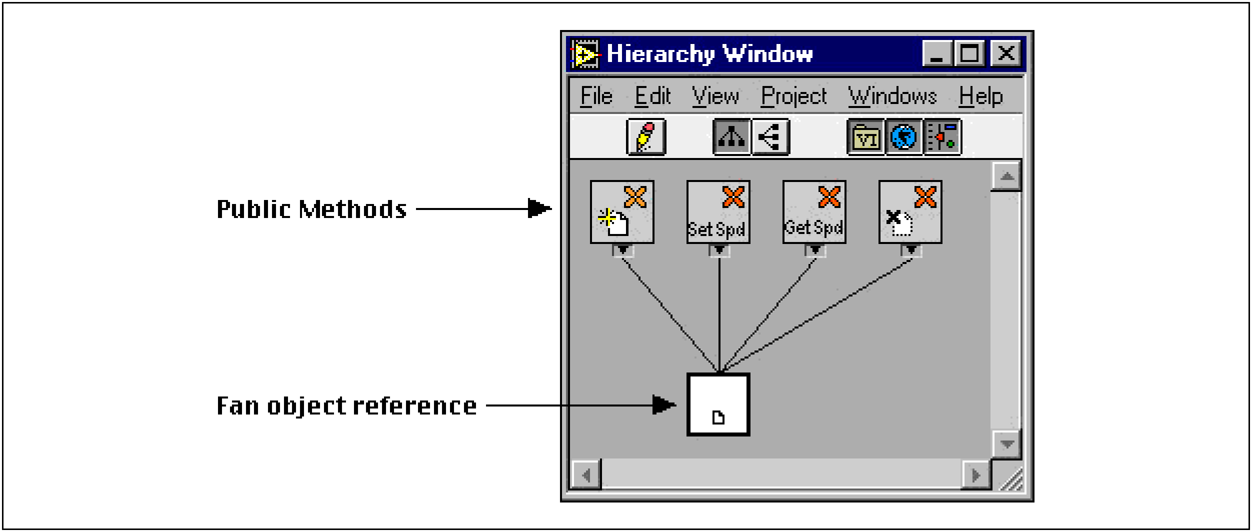
GOOP Mechanics

We implement the climate system controlled by building the five classes described in the previous section. Because LabVIEW has currently no direct support for classes and objects, they will have to be constructed in regular VIs. You can use the **LabVIEW GOOP Wizard** to automatically generate VIs that a class is composed of and that manage the instances of class data for you.

Class

As described before, we will use a LabVIEW LLB to store the VIs that implement a class. VIs that represent public methods will be marked as top-level in the LLB. All other (non top-level) VIs implement either private methods or other support functionality and should not be directly called from users of this class.

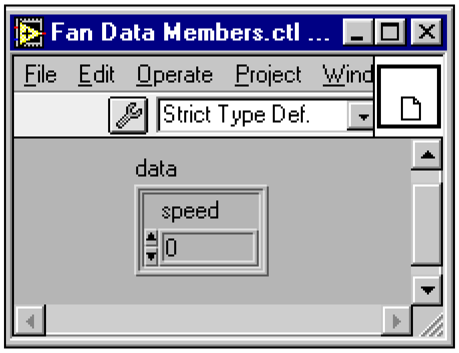
The following is the picture of the VIs that represent the public interface of the Fan class implemented in LabVIEW.



Class Private Data

The private data members for the class are defined inside a cluster stored as a typedef control.

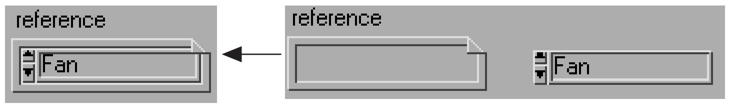
The Fan class contains the private data member **speed**.



Object Reference

The object reference is used to identify individual instances of objects. The object reference consists of an enumeration control, containing the name of the class, inside a Data Log file refnum. Using an enumeration prevents the user from inadvertently wiring an object reference from one class to a method from another class.

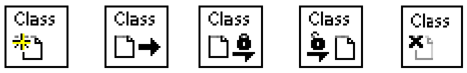
The following is a picture of the Fan Reference.



Object Data Methods

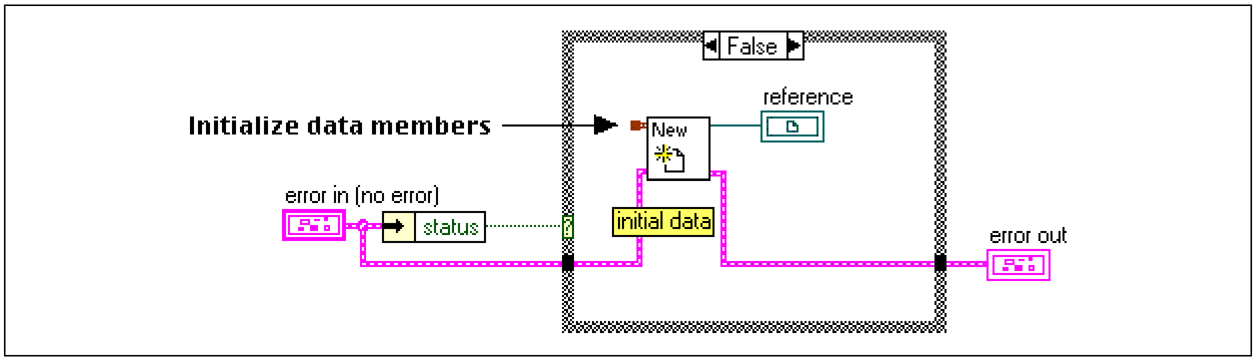
The public interface to your class is comprised of public methods (top-level VIs in your class LLB). A method VI takes an object reference input in order to be able to access the private data of the object on which it operates.

When you use the **LabVIEW GOOP Wizard** to create a class LLB, it generates the following five helper functions so you can create objects and access their data. You use these functions in the methods that you write.

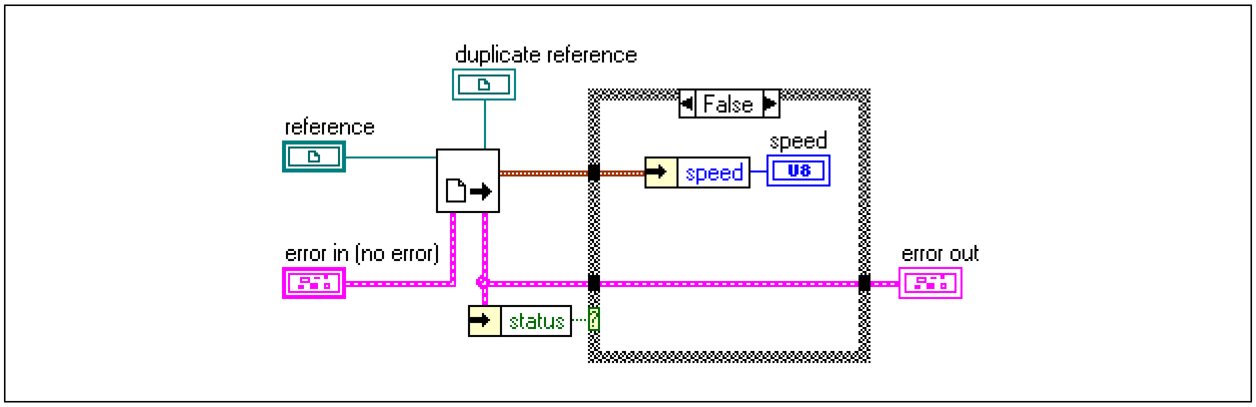


1. **Create an object**—Allocates space for the object's private data (described by the private data typedef control) and returns a reference to this data. You use this function in the constructor method of your class.
2. **Get an object's data**—Returns the state data associated with the object's reference. You use this function in methods that need to read the current state of your object.
3. **Get an object's data and lock it for further modifications**—Returns the state data associated with the object's reference and locks the object so that other methods cannot modify its state. You use this function (along with the following one) in methods that need to modify the object's state information. After you have computed the object's state information, use the following function to assign the data to the object and to unlock the object.
4. **Set an object's data**—Assigns the state data associated with an object's reference and releases the lock placed by the previous function. You can use this function only after you have first locked the object with the previous function. After this function has been executed, the object is ready for a new modification transaction.
5. **Destroy an object**—Releases the spaces associated with an object reference. You use this function in the destructor method of your class. After this function has been called, the object reference becomes invalid.

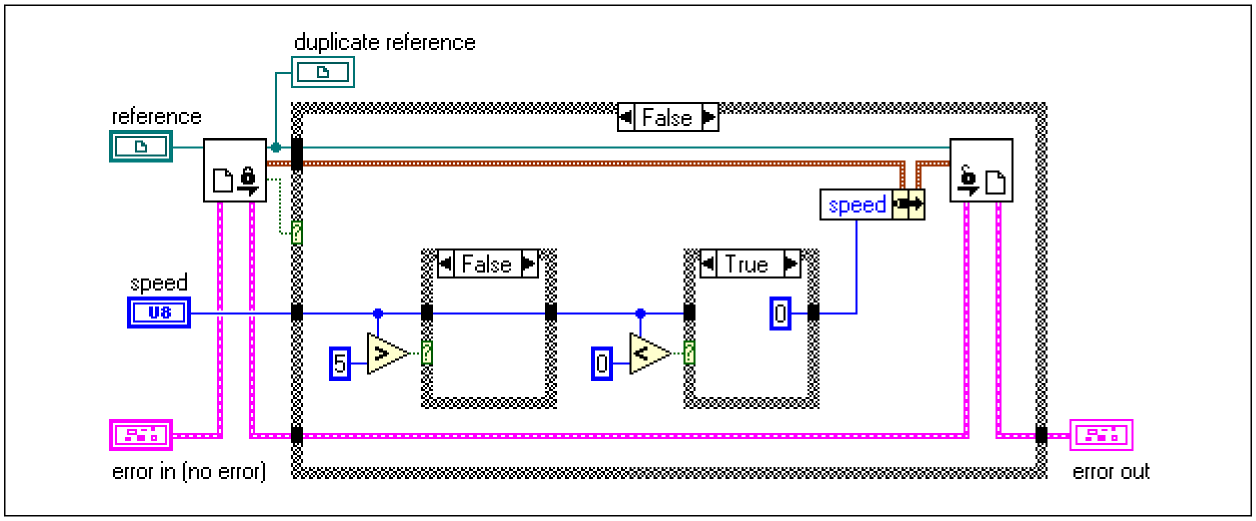
The **Fan Create** constructor method uses the helper VI (1) **Fan New** to allocate data for the fan object and to return its reference. If the data member **speed** needed to be initialized, it would have been connected to the **initial data** input of this VI.



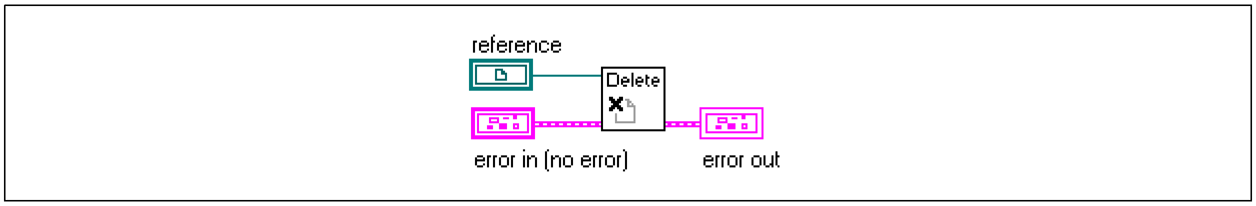
The **Get Speed** method uses the helper VI (2) **Fan Get Data** to return the state information of the Fan object.



The **Set Speed** method uses the helper VIs (3) **Fan Get Data to Modify** and (4) **Fan Set Modified data** to change value of the Fan data member **speed**.

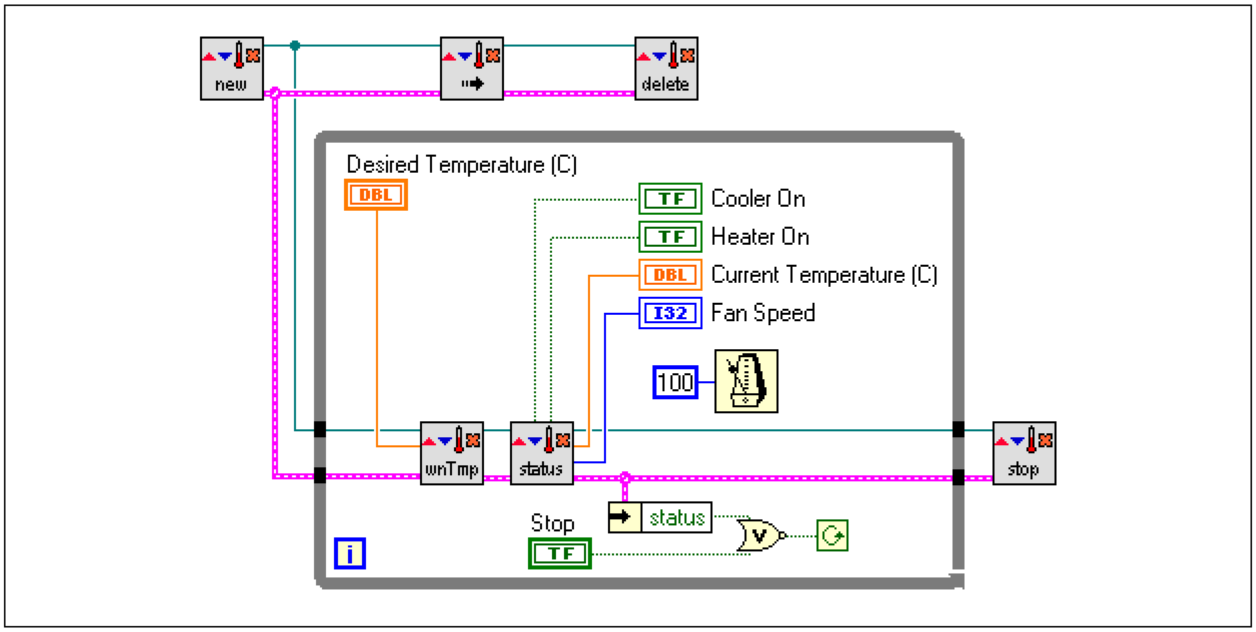


The **Fan Destroy** destructor method uses the helper VI (5) **Fan Delete** to release the memory storing the Fan object's state information.



Component-Based LabVIEW Code

The panel (GUI) of the top-level VI looks the same as in the original implementation of the temperature control system. The following picture shows the new diagram of the top-level VI.



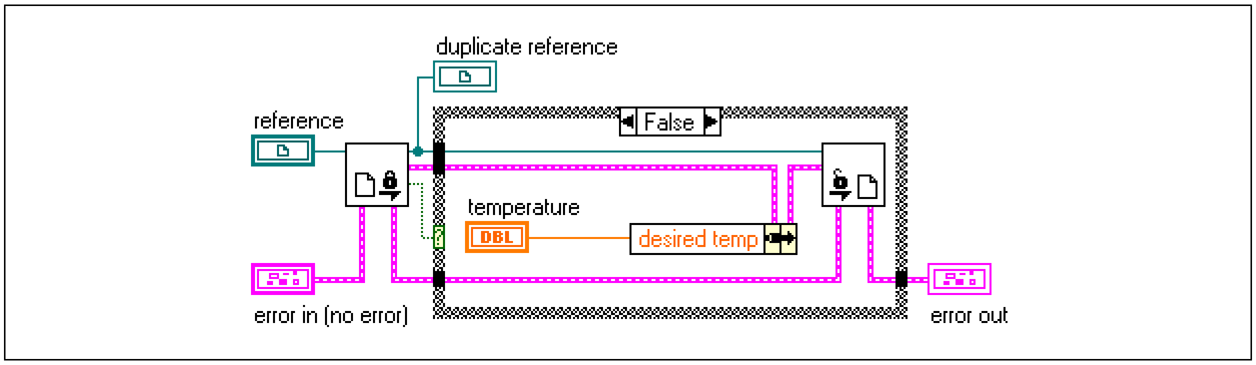
Now that the GUI handling VI is implemented on top of the **Climate System** component, the diagram uses the component's methods to affect and query the status of the system. Because the component encapsulates the sub pieces from which it is composed (fan, heater, and temperature probe), they do not appear directly on this diagram and the GUI VI is decoupled from their implementation. The GUI VI will not have to be modified if the implementation of the **Climate System** component changes.

Let's take a closer look at what happens in the GUI diagram code above.

The **Create** method (top left) is called to create a Climate Controller object. It produces a reference that is then passed to the **Run** method (top center). The **Run** method executes the controller logic in a loop, comparing the current temperature (retrieved from the temp probe) to the target temperature (stored in the **desired temp** private data member) and accordingly changing the settings of the heater, cooler, and fan.

The reference to the Climate Controller object is also passed to the loop that handles the GUI. Inside this loop, the desired temperature is passed to the climate controller using the **Wanted Temp** method, and the current status of the controller is queried using the **Status** method. When the user hits the stop button, the **Stop** method is executed which will terminate the execution of the **Run** method and the Climate Controller object is destroyed by the **Destroy** method (top right).

The following picture shows the diagram of the **Wanted Temp** method. This method simply modifies the private data member **desired temp**, which is being read inside the **Run** method.



Summary

We can easily see the two main benefits derived from breaking the Climate Control system into components.

By isolating the implementation of the Climate Control component from the GUI handling VI, we are free to change its internal implementation without having to worry about how the GUI might be affected. This encapsulation solves the problems associated with the usage of global variables as seen in the initial implementation.

Similarly, by decoupling the GUI handling from the application logic, we can change parts of the GUI without having to worry about the rest of the application. For example, in the original implementation, the update rate of the GUI was directly tied to the behavior of the temperature controller, because both were inside the same loop. By moving the controller loop inside the **Run** method, we can increase or decrease the update rate of the GUI loop without affecting the controller behavior. We could even completely remove the GUI handling loop, and the controller would remain unaffected.

Summary

This document describes how you can use graphical object-oriented programming to address some of the **maintainability**, **scalability**, and **reuse** problems associated with application development in LabVIEW.

Maintainability

The case study shows how hard dependencies within a LabVIEW program can make it difficult to maintain. In an intertwined or monolithic application, a change to one part of the program can often directly affect other parts.

By designing an application as a collection of components, the dependencies between the different parts of the system are reduced to the interfaces presented by the components. When components are used only through their public interface, internal implementation changes inside a component will not affect other parts of the application.

By carefully selecting components that mirror the problem domain of the application, future maintenance work on the application becomes less expensive because the code will remain readable and understandable.

Scalability

Decoupling the different pieces of an application makes it easier to scale the different parts for different needs.

In the example of the climate control system, we could easily add more controllers by making additional calls to the **Climate Controller Create** and **Run** methods. Alternatively, we could have multiple GUIs display the status of a single climate controller at the same time.

Reusability

Each object contains its own instance data, so invoking methods on one object does not affect other objects of the same class. Unlike when global variables are used to store instance data, no new VIs need to be written and saved to disk in order to support multiple objects. All that is required is another call to the constructor method of the class.

Because the internal representation is encapsulated from the users of a component or class, changes to this internal representation will not directly affect the users of this class. Thus, you can reuse the same component or class in many applications without having to worry about future incompatibility.

Final Thoughts

Designing and implementing a class is not a trivial task. It requires familiarity with some concepts that may be new for a LabVIEW programmer. However, you can use tools, such as the **LabVIEW GOOP Wizard**, that reduce the time it takes to implement a class.

Once you have created and tested your classes, they become as easy to use and reuse as the existing components that are shipped with LabVIEW.

341852A-01 Oct99