Bitter, Rick et al "Object-Oriented Programming in LabVIEW"
LabVIEW Advanced Programming Techinques
Boca Raton: CRC Press LLC,2001

10 Object-Oriented Programming in LabVIEW

This chapter applies a different programming paradigm to G: Object-Oriented Programming (OOP). New languages like Java and its use on the Internet have created a lot of interest in this programming paradigm. This chapter explains the concepts that make object-oriented programming work, and applies them to programming in LabVIEW.

This chapter begins with definitions of objects and classes. These are the fundamental building blocks of OOP. Key definitions that define OOP are then presented which give a foundation for programmers to view applications in terms of their constituent objects.

Once the basics of OOP are described, the first stage of objects is presented--object analysis. Fundamentally, the beginning of the design is to identify the objects of the system. Section 10.4 discusses Object Design, the process by which methods and properties are specified. The interaction of objects is also defined in the design phase. The third and last phase is the Object Programming phase. This is where the code to implement the methods and properties is performed.

This type of structuring seems foreign or even backward to many programmers with experience in structured languages such as LabVIEW. Object-oriented is how programming is currently being taught to computer science and engineering students around the world. A significant amount of effort has been put into the design of a process to produce high-quality software. This section introduces this type of philosophy to LabVIEW graphical programming.

Object-oriented design is supported by a number of languages, including C++ and Java. This book tries to refrain from using rules used specifically by any particular language. The concept of object-oriented coding brings some powerful new design tools, which will be of use to the LabVIEW developer. The concept of the VI has already taught LabVIEW programmers to develop applications modularly. This chapter will expand on modular software development.

This chapter discusses the basic methodology of object coding, and also discusses a development process to use. Many LabVIEW programmers have backgrounds in science and engineering disciplines other than software engineering. The world of software engineering has placed significant emphasis into developing basic design processes for large software projects. The intent of the process is to improve

©2001 CRC Press LLC

software quality and reduce the amount of time it takes to produce the final product. Team development environments are also addressed in this methodology.

As stated in the previous paragraph, this chapter only provides a primer on object design methodology. There are numerous books on this topic, and readers who decide to use this methodology may want to consult additional resources.

10.1 WHAT IS OBJECT-ORIENTED?

Object-oriented is a design methodology. In short, object-oriented programming revolves around a simple perspective: divide the elements of a programming problem into components. This section defines the three key properties of object-oriented: encapsulation, inheritance, and polymorphism. These three properties are used to resolve a number of problems that have been experienced with structured languages such as C.

It will be shown that LabVIEW is not an object-oriented language. This is a limitation to how much object-oriented programming that can be done in LabVIEW, but the paradigm is highly useful and it will be demonstrated that many benefits of object-oriented design can be used successfully in LabVIEW. This chapter will develop a simple representation for classes and objects that can be used in LabVIEW application development.

10.1.1 THE CLASS

Before we can explain the properties of an object-oriented environment, the basic definition of an object must be explained. The core of object-oriented environments is the "class." Many programmers not familiar with object-oriented programming might think the terms "class" and "object" are interchangeable. They are not. A "class" is the core definition of some entity in a program. Classes that might exist in LabVIEW applications include test instrument classes, signal classes, or even digital filters. When performing object programming, the class is a definition or template for the objects. You create objects when programming; the objects are created from their class template. A simple example of a class/object relationship is that a book is a class; similarly, *LabVIEW Advanced Programming Techniques* is an object of the type "book." Your library does not have any book classes on its shelves; rather, it has many instances of book classes. An object is often referred to as an instance of the class. We will provide a lot more information on classes and objects later in this chapter. For now, a simple definition of classes and objects is required to properly define the principles of object-oriented languages.

A class object has a list of actions or tasks it performs. The tasks objects perform are referred to as "methods." A method is basically a function that is owned by the class object. Generally speaking, a method for a class can only be called by an instance of the class, an object. Methods will be discussed in more detail in Section 10.2.1.

The object must also have internal data to manipulate. Data that are specified in the class template are referred to as "properties." Methods and properties should be familiar terms now; we heard about both of those items in Chapter 7, ActiveX. Active X is built on object-oriented principals, and uses the terminology extensively.

©2001 CRC Press LLC

Experienced C++ programmers know the static keyword can be used to work around the restriction that objects must exist to use methods and properties. The implementation of objects and classes in this chapter will not strictly follow any particular implementations in languages. We will follow the basic guidelines spelled out in many object-oriented books. Rules regarding objects and classes in languages like C++ and Java are implementations of object-oriented theory. When developing objects for non-object-oriented languages, it will be helpful to not strictly model the objects after any particular implementation.

LabVIEW does not have a built-in class object. Some programmers might suspect that a cluster would be a class template. A cluster is similar to a structure in C. It does not directly support methods or properties, and is therefore not a class object. We will use clusters in the development of class objects in this chapter. One major problem with clusters is that data is not protected from access, which leads us to our next object-oriented principal, encapsulation.

10.1.2 Encapsulation

Encapsulation, or data hiding, is the ability for an object to prevent manipulation of its data by external agents in unknown ways. Global variables in languages like C and LabVIEW have caused numerous problems in very large-scale applications. Troubleshooting applications with many global variables that are altered and used by many different functions is difficult, at best. Object-programming prevents and resolves this problem by encapsulating data. Data that is encapsulated and otherwise inaccessible to outside functions is referred to as "private data." Data that is accessible to external functions is referred to as "public data."

The object-oriented solution to the problem of excessive access to data is to make most data private to objects. The object itself may only alter private data. To modify data private to an object, you must call a function, referred to as a method, that the object has declared public (available to other objects). The solution that is provided is that private data may only be altered by known methods. The object that owns the data is "aware" that the data is being altered. The public function may change other internal data in response to the function call. Figure 10.1 demonstrates the concept of encapsulated data.

Any object may alter data that is declared public. This is potentially dangerous programming and is generally avoided by many programmers. Since public data may be altered at any time by any object, the variable is nearly as unprotected as a global variable. It cannot be stressed enough that defensive programming is a valuable technique when larger scale applications are being written. One goal of this section is to convince programmers that global data is dangerous. If you choose not to pursue object-oriented techniques, you should at least gather a few ideas on how to limit access to and understand the danger of global data.

A language that does not support some method for encapsulation is not object-oriented. Although LabVIEW itself is not object-oriented, objects can be developed to support encapsulation. Encapsulation is extremely useful in large-scale LabVIEW applications, particularly when an application is being developed in a team environment. Global data should be considered hazardous in team environments. It is often

©2001 CRC Press LLC

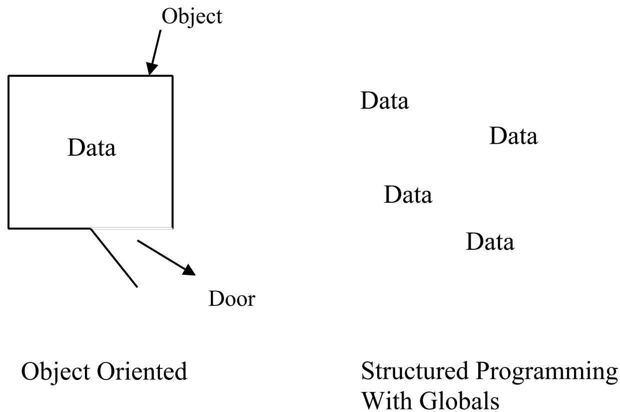

FIGURE 10.1

difficult to know which team member's code has accessed global variables. In addition to having multiple points where the data is altered, it can be difficult to know the reason for altering the data. Using good descriptions of logic (DOL) has minimized many problems associated with globals. Using encapsulation, programmers would have to change the variable through a subVI; this subVI would alter variables in a known fashion, every time. For debugging purposes, the subVI could also be programmed to remember the call chain of subVIs that called it.

Encapsulation encourages defensive programming. This is an important mindset when developing large-scale applications, or when a team develops applications. Application variables should be divided up into groups that own and control the objects. A small degree of paranoia is applied, and the result is usually an easier to maintain, higher quality application. Global variables have been the bane of many college professors for years. This mindset is important in languages like C and C++; LabVIEW is another environment that should approach globals with a healthy degree of paranoia.

10.1.3 Aggregation

Objects can be related to each other in one of two relationships: "is a" and "has a." A "has a" relationship is called "aggregation." For example, "a computer has a CD-ROM drive" is an aggregated relationship. The computer is not specialized by the CD-ROM, and the CD-ROM is not interchangeable with the computer itself. Aggregation is a fundamental relationship in object design. We will see later in the chapter that an aggregated object is a property of the owning object.

Aggregation is a useful mechanism to develop complex objects. In an object diagram, boxes represent classes, and aggregation is shown as an arrow connecting the two objects. The relationship between the computer and CD-ROM is shown in Figure 10.2.

©2001 CRC Press LLC

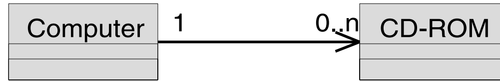

FIGURE 10.2

10.1.4 INHERITANCE

"Inheritance" is the ability for one class to specialize another. A simple example of inheritance is a software engineer is a specialization of an engineer. An engineer is a specialization of an employee. Figure 10.3 shows a diagram that demonstrates the hierarchy of classes that are derived from an employee class. Common in object-oriented introductions is the "is a" relationship. A class inherits from another if it is a specialization or is a type of the superclass. This is a fundamental question that needs to be asked when considering if one class is a specialization of another. Examples of this relationship are engineer "is a" employee, and power supply "is a" GPIB instrument.

When one class inherits from another, the definition of the class is transferred to the lower class. The class that is inherited from is the "superclass" and the inheriting class is the "subclass." For example, consider a class employee. An engineer "is a" employee (please excuse the bad grammar, but its difficult to be grammatically correct and illustrate an "is a" relationship!). This means that the definition of an employee is used by and expanded by the engineer class. An engineer is a specialization of employee. Other specializations may be vice-president, human resources personnel, and managers. Everything that defines an employee will be used by the subclasses. All employees have a salary; therefore, engineers have salaries. The salary is a property of employee and is used by all subclasses of employee.

All employees leave at the end of the day, some later than others. The function of leaving is common, and is a function that all employee subclasses must use. This method is directly inherited, the same leave function may be used by engineers, vice-presidents, and marketing subclasses.

All employees work. Engineers perform different jobs than human resource employees. A definition in the employee class should exist because all employees do some kind of work, but the specifics of the function vary by class. This type of function is part of the employee specification of the employee class, but must be done differently in each of the subclasses. In C++ this is referred to as a "pure virtual function." When a class has a pure virtual function, it is referred to as an "abstract class." Abstract classes cannot be created; only their subclasses may be created. This is not a limitation. In this example, you do not hire employees, you hire specific types of employees.

There is another manner in which classes acquire functions. If employee has a method for taking breaks, and normal breaks are 15 minutes, then most subclasses will inherit a function from employee that lets them take a 15-minute break. Vice-presidents take half-hour breaks. The solution to implementing this method is to have a pure virtual method in employee, and have each subclass implement the break function. Object programming has virtual functions. The employee class will have

©2001 CRC Press LLC

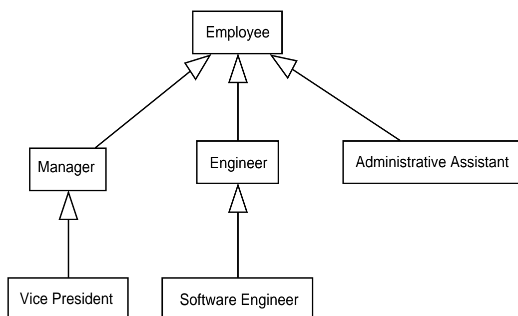

FIGURE 10.3

a 15-minute break function declared virtual. When using subclasses such as engineer and the break function is called, it will go to its superclass and execute the break function. The vice-president class will have its own version of the break function. When the vice-president class calls the break function, it will use its own version. This allows for you to write a single function that many of the subclasses will use in one place. The few functions that need to use a customized version can without forcing a rewrite of the same code in multiple places.

Inheritance is one of the most important aspects of object-oriented programming. If a language cannot support inheritance, it is not object-oriented. LabVIEW is not an object-oriented language, but we will explore how many of the benefits of this programming paradigm can be supported in LabVIEW.

10.1.5 Polymorphism

Polymorphism is the ability for objects to behave appropriately. This stems from the use of pointers and references in languages like C++ and Java (Java does not support pointers). It is possible in C++ to have a pointer to an employee class, and have the object pointed to be an engineer class. When the work method of the pointer is called, the engineer's work method is used. This is polymorphism; this property is useful in large systems where collections of objects of different type are used.

LabVIEW does not support inheritance, and cannot support polymorphism. We will show later in this chapter how many of the benefits of object-oriented programming can be used in LabVIEW, despite its lack of support for object-oriented programming. Polymorphism will not be used in our object implementation in this chapter. It is possible to develop an object implementation that would support inheritance and polymorphism, but we will not pursue it in this chapter.

10.2 OBJECTS AND CLASSES

The concept of OOP revolves around the idea of looking at a programming problem in terms of the components that make up the system. This is a natural perspective in applications involving simulation, test instruments, and data acquisition (DAQ). When writing a test application, each instrument is an object in the system along

©2001 CRC Press LLC

with the device under test (DUT). When performing simulations, each element being simulated can be considered one or more classes. Recall from Section 10.1.1 that an instance of a class is an object. Each instrument in a test rack is an instance of a test instrument class or subclass.

10.2.1 Methods

Methods are functions; these functions belong to the class. In LabVIEW, methods will be written as subVIs in our implementation. The term "method" is not indigenous to object-oriented software, but recall from Chapter 7, ActiveX, that ActiveX controls use methods. Methods may be encapsulated into a class. Methods are considered private when only an instance of the class may use the method.

Methods that are public are available for any other object to call. Public methods allow the rest of the program to instruct an object to perform an action. Examples of public methods that objects should support are Get and Set functions. Get and Set functions allow an external object to get a copy of an internal variable, or ask the object to change one of its internal variables. The Get functions will return a copy of the internal data; this would prevent an external object from accidentally altering the variable, causing problems for the owning object later. Public methods define the interface that an object exposes to other elements of the program. The use of defensive programming is taken to individual objects in object-oriented programming. Only public methods may be invoked, which allows objects to protect internal data and methods.

Only the object that owns itself may call private methods. These types of functions are used to manipulate internal data in a manner that could be dangerous to software quality if any object could alter the internal data. As an example of using objects in a simulation system, consider a LabVIEW application used to simulate a cellular phone network. A class phone has methods to register to the system, make call, and hang up. These methods are public so the program can tell phone objects to perform those actions. Each method, in turn, calls a Transmit method that sends data specific to registration, call setup, or call teardown. The information for each type of message is stored in the specific methods and is passed to the Transmit function. The Transmit function is private to the object; it is undesirable for any other part of the program to tell a phone to transmit arbitrary information. Only specific message types will be sent by the phones. The transmit method may be a common use function internal to the class.

10.2.1.1 Special Method — Constructor

Every class requires two special methods. The first is the Constructor. The Constructor is called whenever an instance of the class is created. The purpose of the Constructor is to properly initialize a new object. Constructors can effectively do nothing, or can be very elaborate functions. As an example, a test instrument class for GPIB instruments would have to know their GPIB address. The application may also need to know which GPIB board they are being used on. When a test instrument object is instantiated, this information is passed to the function in the Constructor.

©2001 CRC Press LLC

This allows for the test instrument object to be initialized when it is created, requiring no additional configuration on the part of the programmer. Constructors are useful when uninitialized objects can cause problems. For example, if a test instrument object ends up with default GPIB address of 0 and you send a message to this instrument, it goes back to the system controller. In Section 10.7.1 we will implement Constructor functions in LabVIEW.

The Constructor method is something that cannot be done with simple clusters. Clusters can have default values, but a VI to wrap around the cluster to provide initialization will be necessary. The Constructor function in LabVIEW will be discussed in Section 10.7. Initialization will allow an object to put internal data into a known state before the object becomes used. Default values could be used for primitive data types such as integers and strings, but what if the object contains data that is not a primitive type, such as a VISA handle, TCP handle, or another object? Constructors allow us to set all internal data into a known state.

10.2.1.2 Special Method — Destructor

The purpose of the Destructor is the opposite of the Constructor. This is the second special method of all classes. When an object is deleted, this function gets called to perform cleanup operations such as freeing heap memory. LabVIEW programmers are not concerned with heap memory, but there are cases when LabVIEW objects will want to have a Destructor function. For instance, if when an object is destroyed it is desirable to write information on this object to a log file. If a TCP conversation were encapsulated into a class object, the class Destructor may be responsible for closing the TCP connection and destroying the handle.

In languages such as C++, it is possible to have an object that does not have a defined Constructor or Destructor. The compiler actually provides default functions for objects that do not define their own Constructor and Destructor. Our implementation does not have a compiler that will graciously provide functions that we are too lazy to write on our own. The object implementation presented later in this chapter requires Constructors for all classes, but Destructors will be optional. This is not usually considered good programming practice in object-oriented programming, but our implementation will not support the full features of OOP.

10.2.2 Properties

Properties are the object-oriented name for variables. The variables that are part of a class belong to that class. Properties can be primitive types such as Booleans, or can be complex types such as other classes. Encapsulating a class inside of another class is "aggregation." We will discuss aggregation later in this chapter. An example of a class with class properties is a bookshelf. The bookshelf itself is a class with an integer property representing the number of shelves. If the shelf were defined to have a single book on the "A" shelf, then a property to describe the book would be necessary. The description of the book is defined as a class with its own properties, such as number of pages.

Properties defined for a class need to have some relevance to the problem to be solved. If your shelf class had a color constant to represent the color of the shelf,

©2001 CRC Press LLC

this information should be used somewhere in the program. Future considerations are acceptable; for instance, we do not use the color information now, but the next revision will definitely need it. If extra properties are primitive types, such as Booleans, there will not be any significant problems. When extra properties are complex types or use resources such as TCP conversations, performance issues could be created because of the extra resources the classes use in the system.

The values of an object's properties make the object unique. All objects of a class have the same methods and property types. Differentiation between objects can only be done with the property values. An example of this would be a generic GPIB instrument class. This class may have properties such as GPIB board and GPIB address. The values of board and address make different GPIB instruments unique. All GPIB objects would have the same methods and property types (address and board number). The value of the address and board make the particular GPIB object unique.

Most properties are private to the class. This means that the class may only modify the member variables (properties) itself. This is another measure of defensive programming. Global data has caused countless headaches for C programmers, and encapsulation is one solution to preventing this problem in object-oriented applications. The implementation for objects in this chapter will effectively make all properties private. This means that we will have to supply methods for modifying data from outside the class.

10.3 OBJECT ANALYSIS

Object analysis is the first stage in an object-oriented design process. The objects that comprise the system are identified. The object analysis phase is the shortest phase, but is the most important. Practical experience has shown us that when the object analysis is done well, many mistakes made in the design and program phases have reduced impacts. Good selection of objects will make the design phase easier. Your ability to visualize how objects interact will help you define the needed properties and methods.

When selecting objects, every significant aspect of an application must be represented in one of the objects. Caution must be exercised to not make too many or negligible-purpose objects. For example, when attempting to perform an object analysis on a test application using GPIB instruments, an object to represent the GPIB cables will not be very useful. Since none of the GPIB interfaces need to know about the cables, encapsulating a cable description into an object will not be of any benefit to the program. No effort is being made at this point to implement the objects; a basic understanding of which objects are necessary is all that should be needed. The design and interactions should be specified well in advance of coding. Having the design finalized allows for easier tracking of scheduling and software metric collection. This also eliminates the possibility of "feature creep," when the definition of what a function is supposed to do keeps getting changed and expanded. Feature creep will drag out schedules, increase the probability of software defects, and can lead to spaghetti code. Unfortunately, spaghetti code written in LabVIEW does have a resemblance to noodles.

©2001 CRC Press LLC

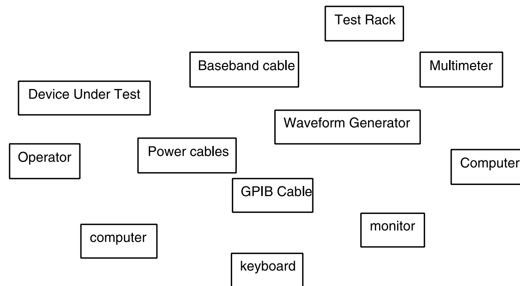

FIGURE 10.4

Example 1:

This example attempts to clarify the purpose of the object analysis on the design of an application to control an Automated Test Equipment (ATE) rack. This example will start out with a relatively simple object analysis. We will be testing Meaningless Objects in Example (MOIE). Table 10.1 identifies the equipment used in the rack. The MOIE has two signal lines, one input line and one output line.

Solution 1:

We will attempt to make everything available an object. Figure 10.4 shows a model of the objects that exist in the rack. This model is not very useful; there are far too many objects from which to build a solution.

The objects for the GPIB cables are not necessary since the test application will not perform any operations directly with the cables. GPIB read and write operations will be performed, but the application does not need to have internal representations for each and every cable in the system. The objects for the input voltages to the test instruments also make for thorough accounting for everything in the system. Nevertheless, if the software is not going to use an object directly, there is no need to account for it in the design.

We are not using objects to represent GPIB cables, but there are times when a software representation of a cable is desirable. If you are working with cables for signal transmission or RF/microwave use, calibration factors may be necessary. An object would be useful because you can encapsulate the losses as a function of cable length, current supplied, or any other relevant factors.

Solution 2:

The easiest object analysis to perform is to declare the three items in the test rack to be their own objects. The simple diagram in Figure 10.5 shows the object relation. This could be an acceptable design, but it is clearly not sophisticated. When performing object analysis, look for common ground between objects in the system. If two or more objects have common functions, then look to make a superclass that has these methods or properties, and have the objects derive from them. This is code reuse in its best form, you only need to write the code once.

©2001 CRC Press LLC

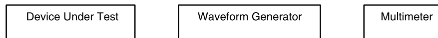

FIGURE 10.5

TABLE 10.1
Object Analysis Example #1

Equipment	Purpose
Multimeter	Measure DC bias on output signal line to Device under Test(DUT)
Arbitary Waveform Generator	Generate test signal stimuli. The signal is generated on the input signal line to the DUT.
Device under Test	The Meaningless Object in Example (MOIE).

Solution 3:

Working from Solution 1 with the three objects (the DUT, waveform generator, and multimeter objects), consider the two GPIB instruments. Both obviously read and write on the GPIB bus, there is common ground between the instruments. A GPIB instrument class could be put into the model, and the meter and waveform generator could inherit the read and write functions of the superclass. Figure 10.6 shows the model with the new superclass.

Which is the best solution, 2 or 3? The answer to that question is — it depends. Personal preferences will come into play. In this example, GPIB control is not necessarily a function that needs to be encapsulated in an object. LabVIEW functions easily supply GPIB functions, and there may be little benefit to abstracting the control to a superclass. The GPIB instrument object needs to be implemented, and this may become more work for the programmer.

Example 2:

This example will perform an object analysis on generic test instruments. We will be using the results of this example throughout the chapter. The analysis here will be the focus of a design example in the next section. We have a need to design a hierarchy of classes to support the development of test instrument drivers. An object analysis starting this vaguely needs to start with a base class named "instrument." This base class defines the basic properties and methods that all test instruments would have. Base classes such as this typically have a few properties and methods, and that is about it. Making common ground can be difficult.

A common property would be an address. This address could be used for the GPIB primary address, or a COM port number for a serial instrument. If we define an address to be a 32-bit number, then it could also be the IP address for Ethernet instruments (should they come into existence soon). This is really the only required property in the abstract base class because it is the only common variable to the major communications protocols. For example, we would not want a baud rate property because TCP/IP or GPIB instruments would not use it. Information on the

©2001 CRC Press LLC

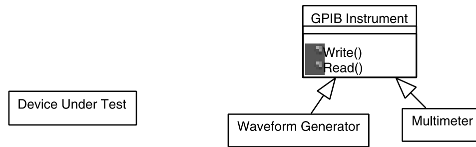

FIGURE 10.6

physical cable lengths is not necessary because only high-speed GPIB has any need for this type of information.

All instruments will likely need some type of read and write methods. These methods would be "pure virtual." Pure virtual means that every subclass must support these methods, and the base class will supply no implementation, just the requirement that subclasses have these methods. Pure virtual methods allow each of the base classes to supply custom functionality to the read and write method, which is inherently different for a serial-, GPIB-, or TCP/IP-based instrument. By defining read and write methods, we have effectively standardized the interface for the objects. This is essentially what the VISA library did for LabVIEW; we are repeating this effort in an object-oriented fashion.

The subclasses of instruments for this example will be serial, GPIB, and IP instruments. Obviously, IP is not a common communication protocol for the test industry, but its representation in the object diagram allows for easy future expansion. The GPIB class will only cover the IEEE 488 standard (we will make 488.2 instruments a subclass of GPIB). The following paragraphs will identify the properties and methods that are required for each of the subclasses.

Serial instruments need to implement the read and write methods of the instrument class. Their COM port information will be stored in the address property, which is inherited from instrument. Specific to serial instruments are baud rates and flow control information. These properties will be made private to serial instruments. A Connect method will be supplied for serial instruments. This method will allow for the object to initialize the serial port settings and send a string message if desired by the programmer. The Connect method will not be required in the base class instrument because some instrument subclasses do not require a connection or initialization routine to begin operation—namely GPIB instruments.

GPIB instruments require a GPIB board and have an optional second address. These two properties are really the only additional items for a GPIB instrument. GPIB instruments do not require a Connect method to configure their communications port. This object must supply Read and Write methods because they derive from "instrument." Other than read, write, board number, and secondary address, there is little work that needs to be done for GPIB instruments.

As we mentioned previously, we intend to have an IEEE 488.2 class derived from the GPIB instrument class. Functionally, this class will add the ability to send the required commands (such as *RST). In addition to read and write, the required commands are the only members that need to be added to the class. Alert readers

©2001 CRC Press LLC

will have noticed that we have not added support for high-speed GPIB instruments. Not to worry, this class makes an appearance in the exercises at the end of this chapter.

IP instruments are going to be another abstract base class. This consideration is being made because there are two possible protocols that could be used beneath IP, namely UDP and TCP. We know that both UDP- and TCP-based instruments would require a port number in addition to the address. This is a property that is common to both subclasses, and it should be defined at the higher-level base class. Again, IP will require that UDP and TCP instruments support read and write methods. An additional pair of properties would also be helpful: destination port and address. These properties can again be added to the IP instrument class.

To wrap up the currently fictitious IP instruments branch of the object tree, UDP and TCP instruments need an initialization function. We will call UDP's initialization method "initialize," and TCP's method will be called "connect." We are making a differentiation here because UDP does not maintain a connection and TCP does. TCP instruments must also support a disconnect method. We did not include one in the serial instrument class because, in general, a serial port can effectively just go away. TCP sockets, on the other hand, should be cleaned up when finished with because they will tie up resources on a machine. The object diagram of this example can be seen in Figure 10.7. Resulting from this object analysis is a hierarchy describing the instrument types. This hierarchy can be used as the base for deriving classes for specific types of instruments. For example, a Bitter-2970 power supply may be a particular serial instrument. Serial Instrument would be the base class for this power supply, and its class could be put into the hierarchy beneath the serial instrument. All properties — COM port, methods, read and write — would be supported by the Bitter-2970 power supply, and you would not need to do a significant amount of work to implement the functionality.

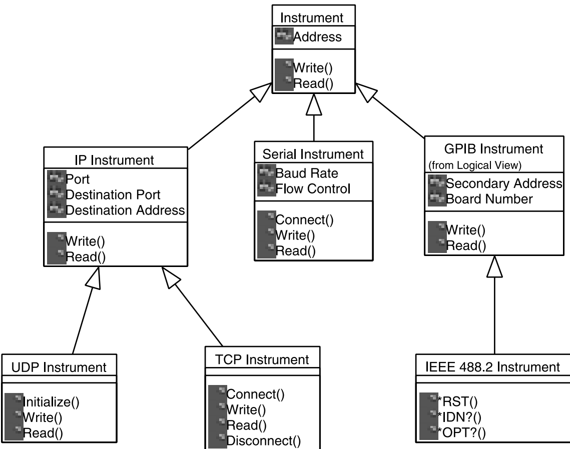

FIGURE 10.7

©2001 CRC Press LLC

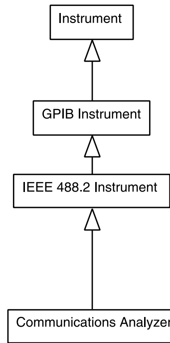

FIGURE 10.8

Example 3:

This example is going to be a bit more detailed than the previous one. We are going to perform the object analysis for the testing of a Communications Analyzer. This is a compound test instrument that has the functionality of an RF analyzer, RF generator, Audio Analyzer, and Audio Generator. The instrument has a list of other functions that it is capable of doing, but for the purposes of this example we will only consider the functions mentioned.

The first step in the object analysis is to determine what the significant objects are. The RF generator, AF generator, RF analyzer, and AF analyzer are fairly obvious. The HP8920 is compliant with IEEE 488.2, so it has GPIB control with a couple of standard instrument commands. Since the instrument is GPIB-based, we will start with abstract classes for instrument and GPIB instrument. A GPIB instrument is a specialization of an instrument. Further specification results in the 488.2 GPIB Instrument subclass. Figure 10.8 shows how our hierarchy is progressing. Some communication analyzers have additional capabilities, including spectrum analyzers, oscilloscopes, and power supplies for devices under test. These objects can also be added to the hierarchy of the design.

An HP-8920 is a communications test set. There are a number of communications analyzers available on the market, for example, the Anritzu 8920 and Motorola 2600 are competitive communications analyzers. All of the instruments have common subsystems, namely the RF analyzers and generators, and the AF analyzers and generators. The preceding sentences suggest that having an HP-8920 as a subclass of a 488.2 instrument is not the best placement. There is a communications analyzer subclass of 488.2 instrument. There may also be other subclasses of the 488.2 instrument, such as power supplies. Figure 10.9 shows the expanding hierarchy.

All of the frequency generators have common elements, such as the frequency at which the generator is operating. The RF generator is going to use a much higher frequency, but it is still generating a signal at a specified frequency. This suggests that a generator superclass may be desirable. Our communications analyzer is going to have several embedded classes. Recall that embedded classes are referred to as

©2001 CRC Press LLC

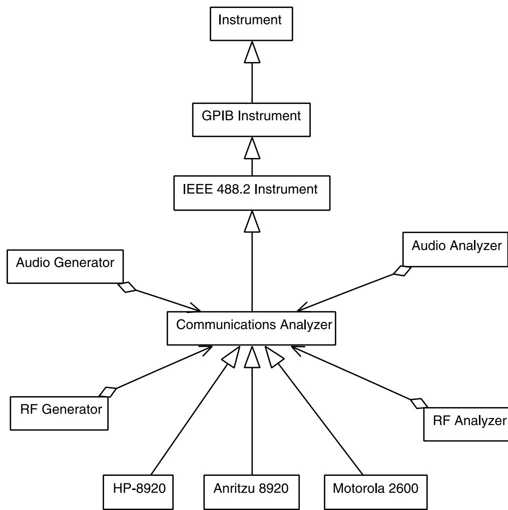

FIGURE 10.9

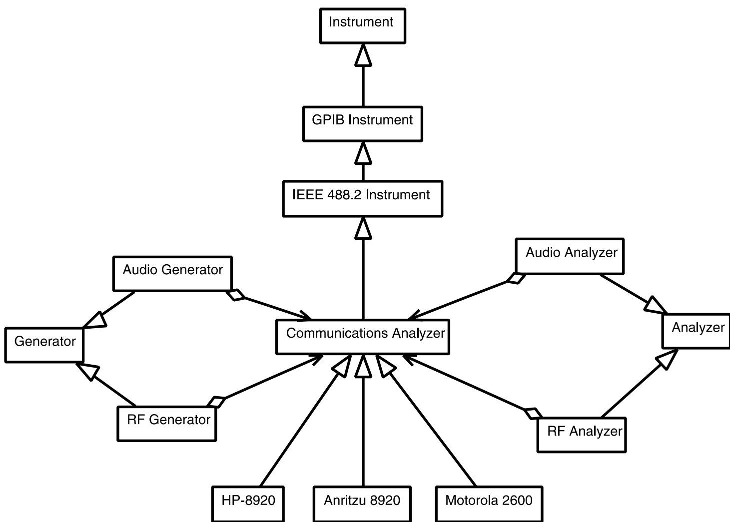

FIGURE 10.10

aggregated classes in object-oriented terminology. We have a new base class to start working with, namely Generator. The AF and RF analyzers have similar starting points, and their analysis is left as an exercise for the reader. Figure 10.10 shows a breakdown of the Analyzer, Generator, and Instrument classes. The dotted line connecting the communications analyzer and the generator/analyzers symbolizes

©2001 CRC Press LLC

aggregation, encapsulating one class into another. We have a basic design on the core objects. Each class should have a paragraph or so description of the purpose for their design. This would complete an object analysis on this project.

The last objects, oscilloscope, spectrum analyzer, and power supply, are independent objects. There are no base objects necessary to simplify their description. The power supply properties and methods should be fairly easy to decide. The oscilloscope and spectrum analyzer are significantly more complex and we are going to leave their analysis and design to the reader.

Object analysis is the first step in object programming, but it is arguably the most important. Good selection of objects makes understanding their interrelations much easier. Now that the basic objects in this system are defined, it is time to focus our attention on how the objects interrelate. The process of analysis and design is iterative; the spiral design model allows you to select your objects, design the objects, and program them. Once you complete one cycle, you can start over and tweak the object list, the function list, and the resulting code.

10.4 OBJECT DESIGN

Object design is where the properties, methods, and interactions of classes are defined. Like object analysis, this phase is pure paperwork. The idea is to have a clearly designed application before code is written. Schedules for coding can be developed with better precision because definitions of functions are well defined. It is much more difficult to estimate code schedules when VIs are defined ad hoc. Coordination of medium to large-scale applications requires project management in addition to a solid game plan. Object designs are well suited for project management processes. Chapter 4, Application Architecture, goes through the details of the waterfall and spiral design models.

A full application design forces architects and programmers to put a significant amount of thought into how the application will be built in the startup phase of a project. Mistakes commonly made during the analysis phase are to neglect or miss application details such as exception handling. Focus at this point of an application should revolve around the interaction of the objects. Implementing individual functions should not be performed unless there are concerns about feasibility.

It is not acceptable to write half the application at this phase of the design unless you are concerned about whether or not a concept is possible. For example, if you were designing a distributed application and had concerns about the speed of DCOM, by all means build a prototype to validate the speed of DCOM. Prototyping is acceptable at this phase because it is much easier to change design decisions when you have not written 100 VIs.

In this section we will explain identifying methods and properties for a class. Once the methods and properties are defined, their interactions can then be considered. The actual implementation of the specified methods and properties will not be considered until the next section. It is important to go through the three distinct phases of development. Once this phase of design is completed, an application's "innards" will be very well defined. The programmers will know what each object is for, how it should function, and what its interactions are with other objects in the

©2001 CRC Press LLC

system. The idea is to make programming as trivial an exercise as possible. Conceptually, programmers will not have to spend time thinking about interactions of objects because we have already done this. The last phase of the design, programming, should revolve around implementing objects on a method-by-method basis. This allows programmers to focus on the small tasks at hand.

Before we begin a full discussion of object design techniques, we need to expand our class understanding: Sections 10.4.1 and 10.4.2 introduce two concepts to classes. The Container class is useful for storing information that will not be used often. Section 10.4.2 discusses the Abstract class. Abstract classes are useful for defining classes and methods that are used to define interfaces that subclasses must support.

10.4.1 Container Classes

Most, but not all, objects in a system perform actions. In languages such as C++ it is often desirable to encapsulate a group of properties in a class to improve code readability. These container classes are similar in purpose to C structures (also available in C++) and clusters in LabVIEW. Container classes have an advantage over structures and clusters, known as "constructors." The constructor can provide a guaranteed initialization of the container object's properties. Knowing from Chapter 6, Exception Handling, that one of the most common problems seen are configuration errors, the object constructor of a container class helps prevent many configuration errors.

Container classes need to support Set and Get functions in addition to any constructors that are needed. If a class performs other actions, then it is not a container class. Container classes should be considered when large amounts of configuration information are needed for objects. Putting this information into an embedded class will improve the readability of the code because the property list will contain a nested class instead of the 20 properties it contains.

10.4.2 Abstract Classes

Some classes exist to force a structure on subclasses. Abstract classes define methods that all subclasses use, but cannot be implemented the same way. An example of an abstract class is an automobile class. A sport utility has a different implementation for Drive than a subcompact car. The automobile class requires that all subclasses define a Drive method. The actual implementation is left up to the subclass. This type of method is referred to as "pure virtual." Pure virtual methods are useful for verifying that a group of subclasses must support common interfaces. When a class defines one or more pure virtual methods, it is an abstract class. Abstract classes may not be instantiated by themselves. An abstract class has at least one method that has no code supporting it.

An abstract class available to the LabVIEW programmer is the Instrument abstract class we designed in Section 10.3. This class requires several methods be supported. Methods to read and write should be required of all instrument derivatives, but serial instruments have different write implementations than GPIB instruments.

©2001 CRC Press LLC

Therefore, the Write method should be specified as a pure virtual function in the instrument base class. A pure virtual function is defined, but not implemented. This means that we have said this class has a function, but we will not tell you how it works. When a pure virtual function is defined in a class, the class cannot be instantiated into an object. A subclass may be instantiated, but the subclass must also provide implementation for the method.

Continuing on the instrument base class, consider adding a Connect method. Future TCP-based instruments would require a Connect method. Considering future expansions of a class is a good design practice, but a Connect method is not required by all subclasses. Some serial devices may have connect routines, but some will not. GPIB instruments do not require connections to be established. Connections to the VISA subsystem could be done in the constructor. The conclusion is that a Connect method should not be defined in the instrument base class. This method may become a required interface in a subclass, for example, the Connect method can be defined in a TCP Instrument subclass.

The highest class defined in many hierarchies is often an abstract class. Again, the main purpose of the top class in a hierarchy is to define the methods that subclasses must support. Using abstract classes also defines the scope of a class hierarchy. A common mistake made by many object designers is to have too many classes. The base class in a hierarchy defines the scope of that particular class tree. This reduces the possibility of introducing too many classes into a design.

To expand on the object analysis begun in the previous section, consider Example 2. We have the base class "Instrument." All instruments have a primary address, regardless of the communications protocol they use. For a GPIB instrument, the address is the primary address. For a serial instrument, the COM port number can be the primary address. Leaving room for the future possibility of Ethernet and USB instruments, the address property will be a 32-bit number and will be used by all instruments. A 32-bit number was chosen because IP addresses are actually four 8-bit numbers. These 8-bit numbers can be stored in a single 32-bit number. This is actually what the String to IP function does in LabVIEW. Because the address is common to all major subsystems, we will define it as a property of the Instrument base class.

We have identified one property that is common to all instruments; now on to common methods. We know that all instruments must read and write. The Read and Write functions will be different for each type of instrument, therefore, the Instrument class must have two pure virtual methods, Read and Write. Languages like C++ use strong type checking, which means you must also define the arguments to the function and the return types. These arguments and return types must match in the subclass. The good news for us is that we are not required to follow this rule. All instruments so far must read and write strings and possess an address. This seems like a good starting point to the instrument architecture. Figure 10.11 shows a Rational Rose drawing of the Instrument class.

The subclass of Instrument that we will design now is the GPIB instrument subtype. Here we are forced with a decision, which properties does a GPIB instrument require? Earlier, we decided to use the VISA subsystem for communications. VISA will handle the communications for our GPIB instruments in this example

©2001 CRC Press LLC

FIGURE 10.11

also. The property that a GPIB instrument class requires is a VISA handle. To generate the handle, the primary address, secondary address, and GPIB board must be given in the constructor. The GPIB Instrument class now has one required constructor. Support for read and write must be provided. Read and Write functions are abstract in the Instrument base class, so we must provide the functionality in this class. These methods and properties pretty much encapsulate the functionality of most GPIB (IEEE 488) instruments.

Some instruments, namely IEEE 488.2-compliant instruments have standard commands, such as Reset (*RST), Identity (*IDN?), and Options (*OPT?). Literally, IEEE 488.2 instruments are a subclass of GPIB instruments, and they will be in this object design. The 13 standard commands will be encapsulated as functions in the 488.2 Instrument subclass. The problem that we are developing is that we are in the second phase of the programming, object design. This class should have been thought of during the object analysis. When following a waterfall design methodology, we should stop performing the design and return to the analysis phase. After a moment of thought, several new features for the 488.2 subclass could become useful. For example, the Identification (*IDN?) command could be used in the constructor. This would allow the application to validate that the instrument on the GPIB bus is, in fact, a 488.2 instrument. If the instrument did not respond, or responds incorrectly to the *IDN? command, an error could be generated by the constructor. This type of functionality could be very useful in an application, better user validation without adding a lot to the program.

Now that methods required of Instrument, GPIB Instrument, and IEEE 488.2 Instrument have been defined, it is time to make use of them. Any instruments built using objects will be subtypes of either IEEE 488.2 or GPIB Instrument. If the physical instrument complies to the 488.2 standard, it will be a subclass of the 488.2 Instrument class. In the event we are working with an older, noncompliant instrument, then it will descend directly from GPIB Instrument. This will allow us to make sure that 488.2 commands will never be sent to a noncompliant instrument.

As far as defensive programming goes, we have made excellent progress in defending the communications ports. Each instrument class encapsulates the communications port information and does not directly give access to any of the external code. This will make it impossible for arbitrary commands to be sent to any of the instruments. We will limit the commands that are sent to instruments to invoke the objects issue. Assuming the objects are written correctly, correct commands can only be sent on the communications lines.

Another set of methods that you can define for classes is operators. If you had an application that used vector objects, how would you perform addition? An

©2001 CRC Press LLC

Addition operator that accepts two vector objects can be written. It is possible to specify operators for all the arithmetic operators such as Addition, Subtraction, Multiplication, and Division. Also, Equality operators can be defined as methods that a class can support. It may be possible to use LabVIEW's Equality operator to directly compare the string handles to our objects, but there are some instances where comparing the flattened strings may not yield the results desired.

We have just identified a number of methods that will exist in the Instrument class hierarchy. To help visualize the interaction among these objects, we will use an interaction diagram. Software design tools such as Rational Rose, Microsoft Visual Modeler, and Software through Pictures provide tools to graphically depict the interaction among classes.

Since the implementation of classes that we will be using later in this chapter does not support inheritance, we will aggregate the superclasses. The interaction diagram will capture the list of VI calls necessary to accomplish communication between the objects. In an interaction diagram, a class is a box at the top of the diagram. A vertical line going down the page denotes use of that class. Arrows between class objects descend down the vertical lines indicating the order in which the calls are made. Interaction diagrams allow programmers to understand the intended use of classes, methods, and their interactions. As part of an object design, interaction diagrams provide a lot of information that class hierarchies cannot.

If we had an application that created an IEEE 488.2 Instrument object and wanted to send an *RST to the physical instrument, the call chain would be fairly simple. At the upper left of the interaction diagram, an object we have not defined appears — Application. This "class" is a placeholder we are using to indicate that a method call is happening from outside an object. Next in line appears IEEE 488.2 Instrument. The arrow connecting the two classes indicates that we are invoking the *RST method. When the *RST method is called, a string *RST will be created by the object and sent to the GPIB Write function. This encapsulation allows us to control what strings will appear on the GPIB bus. In other words, what we have done is defined a set of function calls that will only allow commands to be sent that are valid. This is effectively an Application Programming Interface (API) that we are defining through this method. The diagram for this interaction appears in Figure 10.12.

This diagram is just one of possibly hundreds for a large-scale application. Interaction diagrams do not clearly define what the methods are expected to do, however; a description of logic (DOL) is required. In addition to interaction diagrams for every possible sequence of function calls, a written description of each call is required. In the DOL should appear function inputs and outputs and a reasonable description of what the function is expected to accomplish. DOLs can be simple, as in the case of the *RST command:

*RST-
inputs: error cluster, string handle for object
outputs: error cluster
This function sends a string, *RST on the GPIB bus to the instrument defined in the string handle.

©2001 CRC Press LLC

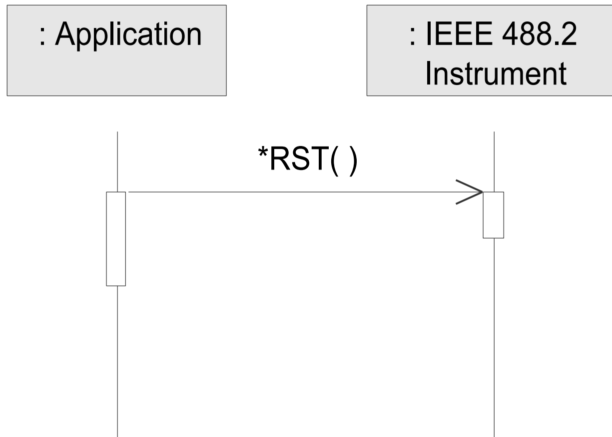

FIGURE 10.12

For complex functions, the description may become a bit more complicated. Remember that the interaction diagrams are there to help. In a DOL you do not need to spell out an entire call sequence, that is what the interaction diagrams are there for. The DOL and interaction diagrams should leave no ambiguity for the programmers. Together, both pieces of information should minimize the thought process needed to program this particular sequence of events.

The class hierarchy, interaction diagrams, and DOL provide a complete picture for programmers to follow. The interaction diagrams provide graphical direction for programmers to follow in the next phase of development, object programming. Thus far we have managed to define what objects exist in the system, and what methods and properties the objects have. Object interactions are now defined, and each method and property has a paragraph or so describing what it is expected to do. This leaves the small matter of programming the methods and objects. This is not a phase of development for taking shortcuts. Leaving design details out of this phase will cause ambiguities in the programming phase. Any issues or possible interactions that are not covered in the design phase will also cause problems in the programming phase. Software defects become a strong possibility when the design is not complete. Some programmers may neglect any interactions they do not see, and others may resolve the issue on their own. This could well cause "undocumented features," which can cause well-documented complaints from customers. When the software model is complete and up to date it is an excellent resource for how an application behaves. In languages like C++ when an application can have over 100 source code files, it is often easier to look at interaction diagrams and get an idea of where a problem is. Surfing through thousands of lines of source code can be tedious and cause programmers to forget the big picture of the application.

©2001 CRC Press LLC

If you do not choose to follow an object-oriented software design methodology, a number of concepts in this chapter still directly apply to your code development. Start with a VI hierarchy — determine what pile of VIs will be necessary to accomplish these tasks. Then write out the interaction diagrams for the possible sequences of events. When writing out interaction diagrams, be sure to include paths that occur when exception handling is in process. Once the interactions are defined, write out descriptions of logic for each of the VIs, and *voila*! All that is left is to code the individual VIs. Writing the code may not be as easy as just described, but much of the thought process as to how the application should work has been decided. All you have to do is code to the plan.

10.5 OBJECT PROGRAMMING

This section concludes our discussion of the basic process of developing object-oriented code. This is the last phase of the programming, and should be fairly easy to do. You already know what every needed object is, and you know what all the methods are supposed to do. The challenge is to keep committed to the waterfall model design process. Waterfall design was discussed in Chapter 4, Application Structure. Object programming works well in large-scale applications, and programming technique should be coordinated with a process model for control of development and scheduling.

Once the object analysis and design are done, effort should be made to stick to the design and schedule. In the event that a defect is identified in the design of an application, work on the particular function should be halted. The design should be reviewed with a list of possible solutions. It is important in a large-scale application to understand the effects a design change can have on the entire application. Making design changes at will in the programming phase can cause issues with integration of the application's objects or subsystems. Software quality can become degraded if the impact of design changes is not well understood. If objects' instances are used throughout a program, it should be clearly understood what impact a design change can have on all areas of the application.

In a general sense, there are two techniques that can be followed in assembling an object-oriented application. The first technique is to write one object at a time. The advantage of this technique is that programmers can be dispersed to write and test individual objects. Once each object is written and tested, the objects are integrated into an application. Assuming that all the objects were written and properly tested, the integration should be fairly simple. If this technique is being used, programmers must follow the object design definitions precisely; failure to do so will cause problems during integration.

The second technique in writing an object-oriented application is to write enough code to define the interfaces for each of the objects. This minimally functional set of objects is then integrated into the application. The advantage of this technique is to have the skeleton of the entire application together, which minimizes integration problems. Once the skeleton is together, programmers can fill in the methods and internal functionality of each of the objects. Before embarking on this path, define

©2001 CRC Press LLC

which need to be partially functional and which need to be fully functional. External interfaces such as GPIB handles may need to be fully functional, while report generation code can only be functional enough to compile and generate empty reports.

10.6 DEVELOPING OBJECTS IN LABVIEW

This section begins to apply the previous information to programming in LabVIEW. Our object representations will use clusters as storage containers for the properties of an object. SubVIs will be used as methods. We will develop VIs to function as constructors and destructors, and to perform operations such as comparison. In addition, methods identified in the object design will be implemented.

Cluster type definitions will be used as containers, but we will only present strings to the programmer. Strings will be used as a handle to the object. Clusters will be used for methods because this prevents programmers from "cheating." The idea here is to encapsulate the data and prevent access as a global variable. Programmers will need to use the Get/Set or other defined methods for access to object "innards." This satisfies the requirement that data be encapsulated. This may seem like a long-winded approach, but it is possible to add functionality to the Set methods that log the VI call chain every time this method is invoked. Knowing which VI modified which variable at which time can be a tremendous benefit when you need to perform some emergency debugging. This is generally not possible if you have a set of global variables that are modified in dozens of locations in the application.

This implementation will not provide direct support for inheritance. Inheritance would require that the flattened string be recognizable as one of a list of possible clusters. Having a pile of clusters to support this functionality is possible, but the parent class should not need to maintain a list of possible child classes. In languages like C++, a virtual function table is used "under the hood" to recognize which methods should be called. Unfortunately, we do not have this luxury. Identifying which method should be called when a virtual method is invoked would be a significant undertaking. This book has a page count limit, and we would certainly exceed it by explaining the logistics of object identification. We will simulate inheritance through aggregation. Aggregation is the ability for one object to contain another as a property. Having the parent class be a property of the child class will simulate inheritance. This technique was used with Visual Basic 4.0 and current versions; Visual Basic does not directly support inheritance. Polymorphism cannot be supported directly because inheritance cannot be directly supported. This is a limitation on how extensive our support for objects can be.

The object analysis has shown us which classes are necessary for an application, and the object design has identified the properties and methods each object has. The first step in implementing the object design is to define the classes. Since LabVIEW is not an object-oriented language, it does not have an internal object representation. The implementation we are developing in this section is not unique. Once you understand the underlying principles behind object-oriented design, you are free to design your own object representations.

©2001 CRC Press LLC

10.6.1 Properties

All objects have the same properties and methods. What makes objects of the same type unique are the values of the properties. We will implement our objects with separated properties and methods. The methods must be subVIs, and the class templates for properties will be type definitions using clusters. The cluster is the only primitive data type that can encapsulate a variety of other primitive types, such as integers. Class definitions will be encapsulated inside clusters. The internal representation for properties is clusters, and the external representation will actually be strings. Since it is impossible for a programmer to access a member variable without invoking a Get or Set method, our implementation's properties are always private members of the class.

When you program a class definition, the cluster should be saved as a type definition. We will use this type definition internally for all the object's methods. The type definition makes it convenient for us to place the definition in class methods. The cluster definition will only be used in internal methods to the class; programmers may not see this cluster in code that is not in direct control of the class. External to the object, the cluster type definition will be flattened to a string. There are a number of reasons why this representation is desirable, several of which will be presented in the next section.

Some readers may argue that we should just be passing the cluster itself around rather than flattening it into a string. The problem with passing the class data around as a cluster is that it provides temptation to other programmers to not use the class methods when altering internal data. Flattening this data to a string makes it difficult, although not impossible, for programmers to cheat. Granted, a programmer can always unflatten the cluster from a string and cheat anyway, but at some point we are going to have to make a decision to be reasonable. Flattening the cluster to a string provides a reasonable amount of protection for the internal data of the object.

10.6.2 Constructors

An object design may determine that several constructors will be necessary. The purpose of the constructor is to provide object initialization, and it may be desirable to perform initialization in several different manners. Our class implementation will require that each object have at least one available constructor. The constructor will be responsible for flattening the Typedef cluster into a string as an external handle to the object. Object-oriented languages such as C++ do not require absolutely that each class have a constructor. Like Miranda rights, in C++, if you do not have a constructor, the compiler will appoint one for you. We are supplying an object implementation for LabVIEW, and our first rule is that all objects must have at least one constructor.

A simple example for a class and its constructor is a point. A point has two properties, an x and y coordinate. Figure 10.13 shows the front panel for a constructor function for a point class. The required two inputs, the x and y coordinates are supplied, and a flattened string representing the points is returned. The code diagram is shown in Figure 10.14. A simple Build Cluster function was used. For more

©2001 CRC Press LLC

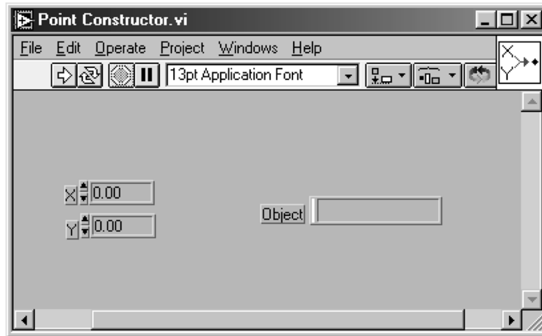

FIGURE 10.13

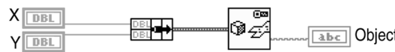

FIGURE 10.14

complicated functions, you want to build a cluster and save it as a control. This control can be placed on the front panel and not assigned as a connector. This would facilitate the cluster sizing without giving up the internals to external entities.

Another object we will develop using the point is the circle. A circle has a radius and an origin. The circle object will have the properties of a point for origin and a double-precision number for its radius. The cluster control for a circle is shown in Figure 10.15. The circle object constructor, like the point constructor, uses floating-point numbers for radius, *x*, and *y*. The front panel is shown in Figure 10.16, and the code diagram is shown in Figure 10.17.

The front panel uses type definitions for the contents of the clusters. Point coordinates are fed into the point constructor and the string input is fed into the circle's cluster definition. The clusters are used only as inputs to the Bundle function. Next, the cluster is then flattened and returned as a string. The circle uses a nested class as one of its properties.

Now, say that a programmer has the radius and the point itself in a string. Another constructor can be built using the string point object and the radius as inputs. The block diagram for this VI is shown in Figure 10.18. Note that the first thing the constructor does is validate the point object. The error clusters are included in this VI; it is a trivial matter to include them in the other constructors. The circle object demonstrates where multiple constructors are desirable. Functions that have the same purpose, but use different arguments can be called "overloaded." In C++ and Java, a function name can be used many different times as long as the argument list is different for each instance of the function. The function name and argument list is the function's signature in C++; this does not include the return type. As long as different signatures are used, function name reuse is valid. LabVIEW does not have this restriction if desired multiple constructors with the same input list can be used.

©2001 CRC Press LLC

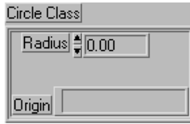

FIGURE 10.15

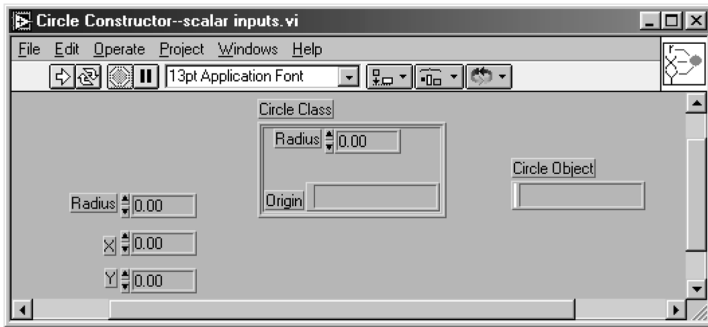

FIGURE 10.16

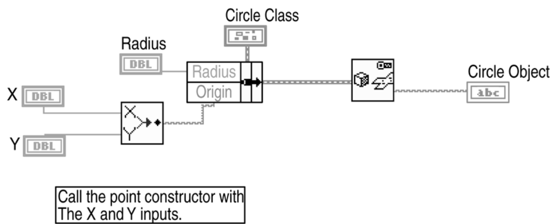

FIGURE 10.17

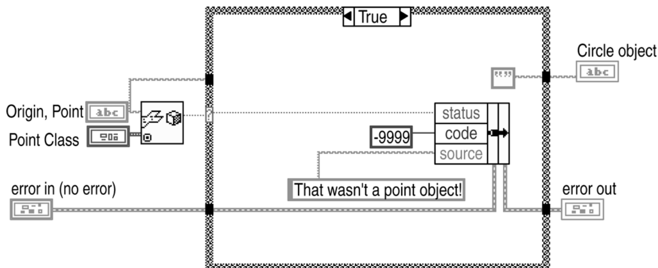

FIGURE 10.18

©2001 CRC Press LLC

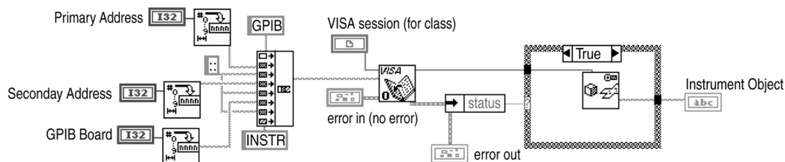

FIGURE 10.19

Example 10.6.1

Develop a constructor for a GPIB instrument class. All GPIB instruments have primary addresses, secondary addresses, and GPIB boards they are assigned to. All instrument communications will be performed through the VISA subsystem.

Solution:

First, we need to consider the design of the object. The problem statement made it obvious that addresses and GPIB boards are to be used by the object. The statement also says that the communications will be performed via VISA calls. The cluster really needs just one item, a VISA handle. The inputs of addresses and board number can be formatted into a VISA descriptor. The VISA Open VI will open the connection and return the needed VISA handle. Figure 10.19 shows the constructor. Not shown is the front panel, in which the only notable features are the limitations placed on the GPIB address: the primary and secondary addresses must be between 1 and 31. If the secondary address is zero, then we will use the primary address as the secondary address. In addition, the GPIB board number must be between 0 and 7. Since this VI will perform communications outside LabVIEW and into the VISA subsystem, error clusters are used to indicate the success or failure of the operation. If VISA Open returns an error, then an empty string is returned as the object in addition to setting the error cluster. Our implementation of objects will return empty strings when objects cannot be created.

10.6.3 Destructors

Destructors in our object implementation only need to exist when some activity for closing the object needs to be done. Any objects used to encapsulate TCP, UDP, VISA, ActiveX (automation), or synchronization objects should destroy those connections or conversations when the object is destroyed. If the object's string is allowed to go out of scope without calling the destructor, LabVIEW's engine will free up the memory from the no-longer-used string. The information holding the references to the open handles will not be freed up. Over long periods of time, this will cause a memory leak and degrade LabVIEW's performance.

Classes such as the point and circle do not require a destructor. The information they hold will be freed up by the system when they are eliminated; no additional handles need to be closed. This is actually consistent with other programming languages such as C++ and Java. Programmers need to implement destructors only when some functionally needs to be added.

©2001 CRC Press LLC

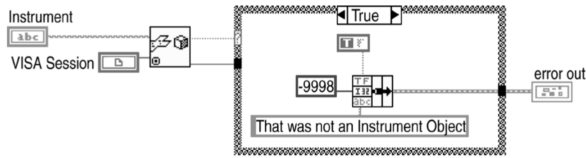

FIGURE 10.20

Example 10.6.2

Implement a destructor for the GPIB class object created in Example 10.6.1.

Solution:

All that needs to be done for this example is to close the VISA handle. However, when we recover the cluster from the object string, we will verify that it is a legitimate instance of the object. The destructor code diagram is shown in Figure 10.20.

The significance of destructors is important for objects that communicate with the outside world. For internally used objects, such as our point and circle, or for objects simulating things like signals, no destructor is necessary. Unlike constructors, there is only one destructor for an object. In our implementation it is possible to construct multiple destructors, but this should be avoided. It will be obvious in the object design what items need to be closed out; this should all be done in one point. The act of destroying an object will require the use of only one destructor. Having multiple destructors will serve to confuse programmers more than it will help clarify the code.

10.6.4 Methods

In this section we will begin implementing methods that objects will use. The two distinct classifications of methods will be Public and Private methods. Protected methods will not be supported; their implementation will be much more difficult. A stronger class design in the future may allow for protected interfaces. Private and public methods will be handled separately because their interfaces will be different.

Our object implementation will use methods to interface with the outside application exclusively. Setting or retrieving the values of properties will be handled through methods called Set and Get. This is how ActiveX is implemented. Each property that you have read and/or write access to is handled through a function call.

10.6.4.1 Public Methods

Public methods take a reference to the object in the form of a string. This prevents outside code from modifying private properties without a defined interface. A special example of a Public function is the constructor. The constructor takes inputs and returns a string reference to the object. This is a public method because it was called from outside the object and internal properties were configured by a method the object supports.

©2001 CRC Press LLC

Class designs may have many public methods, or just a few. Object design tools like Rational Rose generate code that, by default, will include Get and Set functions for access to object properties. We will be using Get and Set functions to alter properties of our objects. The Get/Set functions will form the protective interface for member variables. Set methods will allow us to control or force rules on setting internal data members. This gives us the ability to perform a lot of intelligent coding for the object users. From our previous example, a For GPIB Instrument, we could have a Set GPIB Address method as public. This method allows programmers to allow the object to determine if its GPIB address has been changed during execution. If this is the case, the object could be made capable of refusing to make the change and generate an error, or of closing its current VISA session and creating a new one. Public methods enable the object to make intelligent decisions regarding its actions and how its private data is handled. This is one of the strengths of encapsulation. When designing objects, consideration can be made as to how the Get and Set interfaces will operate. Intelligent objects require less work of programmers who use them because many sanity-checking details can be put into the code, freeing object users from trivial tasks.

10.6.4.2 Private Methods

The major distinction between public and private methods in our LabVIEW implementation is that private methods may use the type definition cluster as input. Private methods are considered internal to the class and may directly impact the private properties of the class. The extra step of passing a string reference is unnecessary; basically, a private method is considered trustable to the class. When using this object implementation in a LabVIEW project, a private method should always appear as a subVI to a public method VI. This will enable you to verify that defensive programming techniques are being followed. As a quick check, the VI hierarchy can be examined to verify that private methods are only called as subVIs of public methods. Public methods serve as a gateway to the private methods; private methods are generally used to simplify reading the code stored in public methods.

By definition, private methods may only be invoked by the object itself. When can an object call an internal method? Public methods may call private methods. When a public method is executing, it may execute private methods. This may seem to be a waste, but it really is not. This is a good defensive programming practice. The public method is an interface to the external program, and the private methods will be used to accomplish tasks that need to be performed.

As an example of a private method that an object would use, consider implementing a collection of VIs to send e-mail using Simple Mail Transfer Protocol (SMTP). To send mail using SMTP, a TCP connection must be established to a server. The Read and Write TCP functions are not something you would want a user of the SMTP mail object to directly have access to. Implementing the SMTP protocol is done internally to the SMTP object. Again, this is defensive programming. Not allowing generic access to the TCP connection means that the SMTP object has complete control over the connection, and that no other elements of code can write data to the mail server that has not been properly formatted.

©2001 CRC Press LLC

Simple objects may not require private methods to accomplish their jobs. A simple object such as vector does not require a private method to determine its magnitude. In our implementation, private methods are only necessary if they simplify code readability of a public method. There are a number of places where this is desirable. In our GPIB Instrument class, public methods would store the strings needed to send a given command. The common ground between all the public methods would be the need to send a string through the GPIB bus. A Write VI can be made as a private method so you do not need to place GPIB write commands and addressing information in each of the VIs. This can be done in the single private method and dropped into public methods.

10.7 EXAMPLE, DEVELOPING INSTRUMENT DRIVERS

This section will develop several instrument drivers to illustrate the benefits of object modeling in LabVIEW. This technique works very well with SCPI instruments. SCPI command sets are modularized and easily broken down into class templates. Object-based instruments can be considered alternatives to standard instrument drivers and IVI Instruments. Our first example will concentrate on power supplies, specifically one of the IVI-based instruments.

A simple object model for a power supply would have the core definition of a power supply. It would be fair to assume that GPIB controllers will control a vast majority of power supplies. It would make sense to reuse the core GPIB Instrument class to control GPIB behavior. This design will use a one-class-fits-all approach. In a true object-oriented implementation, it would make sense to have a power supply abstract base class that defines voltage and current limit properties. In addition to the current limit property, we will be supplying a read only property: current. This will allow users to read the current draw on the supply as a property. Since we are adding object implementations to a non-object-oriented language, many of the advantages of abstract base classes do not apply. Instead, we will define the methods and properties that are used by the vast majority of power supplies and implement them with the addition of a model property. This technique will work well for simple instruments, but complex instruments such as oscilloscopes would be extremely difficult. Multiple combinations of commands that individual manufacturer's scopes would require to perform common tasks would be an involving project, which it is for the IVI Foundation.

The purpose of the Model property is to allow users to select from an enumerated list of power supplies that this class supports. Internally, each time a voltage or current is set, a Select VI will be used to choose the appropriate GPIB command for the particular instrument model. From a coding standpoint, this is not the most efficient method to use to implement an instrument driver, but the concepts of the objects are made clear.

The first step in this object design is to determine which properties go inside the cluster Typedef for this class. Obvious choices are values for current limit and voltage. This will allow the power supply driver to provide support for caching of current limit and voltage. No values will be retained for the last current measurement made; we do not want to cache that type of information because current draw is subject to

©2001 CRC Press LLC

FIGURE 10.21

large changes and we do not want to feed old information back to a user. Other values that need to be retained in this class are the strings for the parent classes, GPIB Instrument, and IEEE 488.2 Instrument. The cluster definition is shown in Figure 10.21. The constructor for power supply will call the constructor for IEEE 488.2 Instrument. We need to retain a handle for this object, which will take the GPIB address information as an argument. The GPIB addressing information does not need to be retained in the Typedef for power supply. The IEEE 488.2 Instrument will, in turn, call the constructor for GPIB Instrument, which takes the GPIB address information to its final destination, the VISA handle it is used to generate.

The Constructor VI for the power supply takes several inputs: the GPIB board number, GPIB primary address, GPIB secondary address, and the power supply model. The power supply constructor will then call the constructor for the IEEE 488.2 Instrument. Since we have a higher base class, the IEEE 488.2 Instrument will call the constructor for the GPIB Instrument class. This class will initialize the VISA session and return a string burying this information from users and lower classes. This defensive programming restricts anyone from using this handle in a mechanism that is not directly supported by the base class. Constructor functions for each of the three classes in the chain are shown in Figure 10.22.

The next items that need to be developed are the Get and Set VIs for the properties' voltage and current. Get and Set methods will be supplied for voltage and current limits, but only a Get command needs to be supplied for the current draw property. Current draw is actually a measurement, but from an object design standpoint it is easily considered to be a property. Set Voltage is shown in Figure 10.23. The Get Current Draw method is shown in Figure 10.24.

We are not supplying Get or Set functions for the GPIB address or instrument model information. It is not likely that a user will switch out power supplies or change addressing information at run-time, so we will not support this type of operation. It is possible to supply a Set Address method that would close off the current VISA handle and create a new GPIB Instrument class to reflect the changes. This sounds like an interesting exercise for the reader, and it appears in the exercises at the end of this chapter.

One issue that we face is the lack of inheritance in our classes. The code to support inherited methods is somewhat bulky and limiting. The GPIB Write commands need to propagate from power supply to IEEE 488.2 Instrument to GPIB Instrument. This is a fairly large call chain for an act of writing a string to a GPIB bus. The wrapper VIs do not take a significant amount of time to write, but larger

©2001 CRC Press LLC

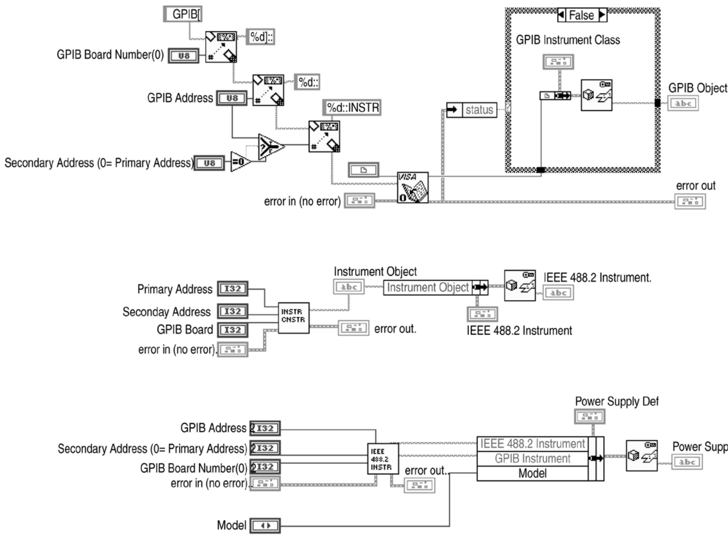

FIGURE 10.22

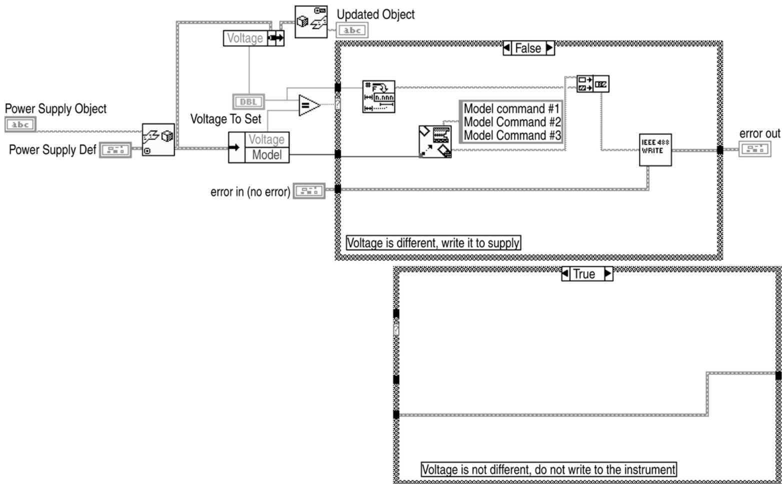

FIGURE 10.23

instruments with several hundred commands could require several hundred wrappers, which would take measurable engineering time to write and is a legitimate problem. In true object-oriented languages, the inherited functions are handled in a more elegant fashion.

The two interaction diagrams presented in Figures 10.25 and 10.26 show the sequence of VI calls needed to set the voltage of the power supply. The first diagram

©2001 CRC Press LLC

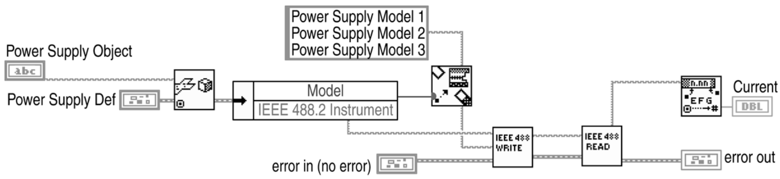

FIGURE 10.24

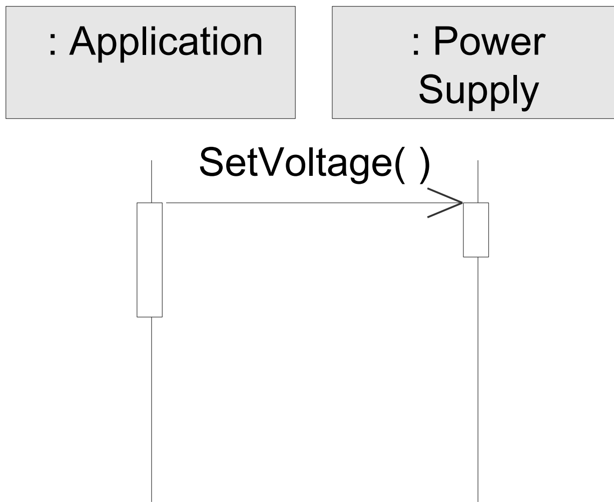

FIGURE 10.25

shows what happens when the Voltage Set method gives a value equal to the currently-cached voltage. The function simply returns without issuing the command. The second diagram shows that the Voltage Set method had an argument other than the currently-cached command and triggers the writing of a command to the GPIB instrument class.

In the next example, we consider a more complicated instrument. This example will introduce another concept in programming, "Friend classes." Friend classes are the scourge of true object-oriented purists, but they present shortcuts to solving some of our implementation problems.

10.7.1 Complex Instrument designs

In this example we will implement the communications analyzer we performed the object analysis on in Section 10.3. Now that we understand how our object implementation works in LabVIEW, we can review some of the analysis decisions and move forward with a usable driver.

©2001 CRC Press LLC

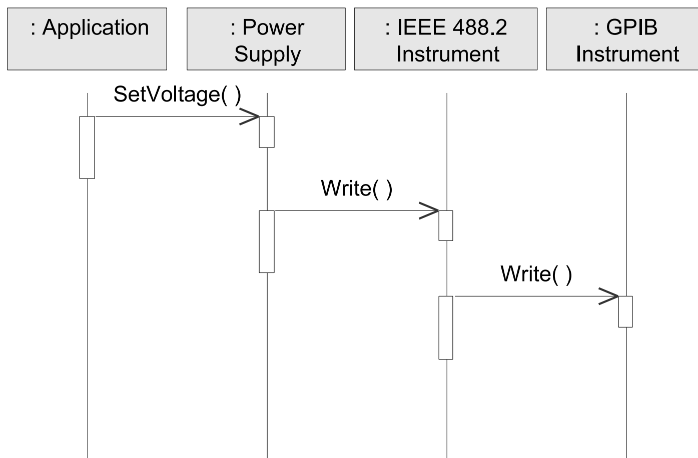

FIGURE 10.26

Some limitations arose in the power supply example we did previously, and we will need to address these limitations. First, power supplies are generally simple instruments on the order of 20 GPIB commands. Modern communications analyzers can have over 300 instrument commands and a dizzying array of screens to select before commands can be executed. The one driver model for a group of instruments will not be easily implemented; in fact, the IVI Foundation has not addressed complex test instruments at this time. The matrix of command combinations to perform similar tasks across different manufacturer's instruments would be difficult to design, and an amazing amount of work to implement. In other words, standardizing complex instruments is a significant undertaking, and it is not one that the industry has completely addressed. It is also not a topic we should address.

The object analysis presented in Section 10.3 works very well in object-oriented languages. Our implementation does have some problems with inheritance, and we will have to simplify the design to make implementation easier. Abstract classes for the analyzers and generators have a single property and a handful of pure virtual methods. Since virtual methods are not usable in our implementation, we will remove them. We will still make use of the IEEE 488.2 class and the GPIB Instrument class. The object hierarchy is greatly simplified. The communications analyzer descends from IEEE 488.2 Instrument, which is a natural choice. The generators, RF and AF, appear on the left side of the communications analyzer. This is an arbitrary choice since there is no special significance to this location. The analyzers appear on the right. The arrow connecting the communications analyzer to the component objects denotes aggregation. Recall that aggregated classes are properties of the owning class; they do not exist without the owning class. This is a natural extension of the object model; this model reflects the real configuration of the instrument. At this point we can claim a milestone in the development: the object analysis is now complete.

©2001 CRC Press LLC

Communications Analyzer

RF Generator
RF Analyzer
AF Generator
AF Analyzer
IEEE 488.2 Instrument

GetRfGenerator()
GetRfAnalyzer()
GetAfGenerator()
GetAfAnalzyer()
GetIEEE_488_2_Instrument()

FIGURE 10.27

Hewlett Packard's HP-8920A communications analyzer will be the instrument that is implemented in this design. We are not endorsing any manufacturer's instruments in this book; its standard instrument driver appears on the LabVIEW CDs and allows us to easily compare the object-based driver to the standard driver.

The next difficulty that we need to address is the interaction between the aggregated components and the communications analyzer. This example will implement a relatively small subset of the HP-8920's entire command set. This command set is approximately 300 commands. Each component object has between 20 and 50 commands. A significant number of wrapper VIs need to be implemented since the aggregated components are private data members. This author is not about to write 300 VIs to prove an example and is not expecting the readers to do the same. We will need to find a better mechanism to expose the aggregate component's functionality to outside the object. The solution to this particular problem is actually simple. Since the analyzers and generators are properties of the communications analyzer, we can use Get methods to give programmers direct access to the components. This does not violate any of our object-oriented principals, and eliminates the need for several hundred meaningless VIs. Figure 10.27 shows the list of the properties and methods the communications analyzer has. Properties and methods with a lock symbol next to them are private members and may only be accessed directly by the class itself.

Now that we have a basic design down for the core object, it is time to examine a small matter of implementation. Since we will give programmers access to the analyzers and generators, it stands to reason that the programmers will set properties and expect the components to send GPIB commands to the physical instrument. Reexamining the class hierarchy shows us that we do not have the ability to access the GPIB Instrument class directly from the generator and analyzers because they do not derive from GPIB Instrument. This presents a problem that needs to be resolved before we proceed with the design.

We require that the component objects be capable of accessing the GPIB Read and Write methods of the communications analyzer. It does not make sense to have each of the components derive from the communications analyzer — this does not satisfy the "is a" relationship requirement for subclasses. In other words, the phrase, "an RF generator 'is a' communications analyzer," is not true. Having the component objects derive from the communications analyzer is not a solution to this problem.

We can give the component objects a property for the communications analyzer. Then they can invoke its GPIB Write method. This solution does not realize the

©2001 CRC Press LLC

class hierarchy we developed in the object analysis phase, either. It is possible to go back and change the class hierarchy, but now we do not satisfy the “has a” requirement for aggregated components. An audio generator “has a” communications analyzer is not a true statement and, therefore, having a property of a communications analyzer does not make much sense.

Since none of our solutions thus far make it in our given framework, we will need to expand it somewhat. An RF analyzer is a component in a communications analyzer and it does have access to the onboard controller (more or less). When performing an object analysis and design, it often helps to have the objects interact and behave as the “real” objects do. For the communications analyzer to give access to the GPIB Read and Write methods would violate the encapsulation rule, but it does make sense as it models reality. The keyword for this type of solution in C++ is called “friend.” The friend keyword allows another class access to private methods and properties. We have not discussed it until this point because it is a technique that should be used sparingly. Encapsulation is an important concept and choosing to violate it should be justified. Our previous solutions to GPIB Read and Write access did not make much sense in the context of the problem, but this solution fits. In short, we are going to cheat and we have a legitimate reason for doing so. The moral to this example is to understand when violating programming rules makes sense. Keywords such as friend are included in languages like C++ because there are times when it is reasonable to deviate from design methodologies.

Now that we have the mechanism for access to the GPIB board well understood, we can begin implementation of the methods and properties of the component objects. We shall begin with the audio generator. Obvious properties for the audio generator are Frequency, Enabled, and Output Location. We will assume that users of this object will only be interested in generating sinusoids at baseband. Enabled indicates whether or not the generator will be active. Output Location will indicate where the generator will be directing its signal. Choices we will consider are the AM and FM modulators and Audio Out (front panel jacks to direct the signal to an external box). The last property that we need to add to the component is one for GPIB Instrument. Since the communications analyzer derives from IEEE 488.2 Instrument, we do not have direct access to the GPIB Instrument base class. It will suffice to have the IEEE 488.2 Instrument as a property.

Methods for the audio generator will be primarily Get and Set methods for the Frequency, Enabled, and Output Location properties. Programmers have a need to change and obtain these properties. Since this component borrowed the information regarding the GPIB bus, it is not appropriate to provide a Get method to IEEE 488.2 Instrument. A Set method for IEEE 488.2 Instrument does not make much sense either. The audio generator is a component of a communications analyzer and cannot be removed and transferred at will. The IEEE 488.2 Instrument property must be specified at creation as an argument for the constructor. The complete list of properties and methods for the audio generator appear in Figure 10.28.

The two last issues to consider for the audio generator are the abilities to cache and validate its property values. This is certainly possible and desirable. We will require the Set methods to examine the supplied value against that last value supplied. We also need to consider that the values supplied to amplitude are relative to where

©2001 CRC Press LLC

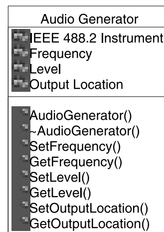

FIGURE 10.28

the generator is routing its output. A voltage is used for the audio out jacks, where FM deviation is used when the signal is applied to the FM modulator. When a user supplies an amplitude value, we need to validate that it is in a range the instrument is capable of supporting. This is not a mission-critical, show-stopping issue; the instrument will limit itself and generate an error, but this error will not be propagated back to the user's code. It makes more sense to have the object validate the inputs and generate an error when inputs are outside of acceptable ranges. RF generator will be subjected to similar requirements.

Last is the destructor for the function. Most of the properties are primitive, floating-point numbers and do not require a destructor. The IEEE 488.2 property has an embedded handle to a VISA session. It would make sense for the control to clean this up before it exits. In this case it is undesirable for the component objects to clean up the VISA handle. A total of five components have access to this property, and only one of them should be capable of destroying it. If we destroy the VISA handle in this object, it will be impossible to signal the other components that the GPIB interface is no longer available. Since this component has "borrowed" access to the GPIB object, it stands to reason that only the communications analyzer object should have the ability to close out the session. Therefore, this object does not need a destructor; everything including the GPIB object property can simply be released without a problem. We must be certain, however, that the destructor for the communications analyzer terminates the VISA session.

The audio and RF generators have a fair amount in common, so it is appropriate to design the RF generator next. The RF generator will need to have an IEEE 488.2 property that is assigned at creation, like the audio generator. In addition, the RF generator will require a Frequency, Amplitude, and Output Port property. Without rewriting the justification and discussion for the audio generator, we will present the object design for the RF generator in Figure 10.29.

Next on the list will be the audio analyzer. This component will not have any cached properties. The Get methods will be returning measurements gathered by the instrument. Caching these types of properties could yield very inaccurate results. The IEEE 488.2 Instrument will be a property that does not have a Get or Set method; it must be specified in the constructor. The audio analyzer will need a property that identifies where the measurement should be taken from. Measurements can be made from the AM and FM demodulator in addition to the audio input jacks. Measurements

©2001 CRC Press LLC

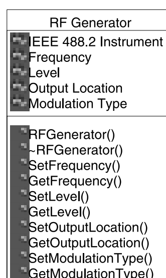

FIGURE 10.29

that are desirable are Distortion, SINAD, and Signal Level (Voltage). The Distortion and SINAD measurements will be made only from the demodulator, while the Signal Level measurement can only be made at the audio input jacks. These are requirements for this object.

The object should change the measurement location property when a user requests a measurement that is not appropriate for the current setting. For example, if a user requests the Signal Level property when the measurement location is pointing at the FM demodulator, the location property should be changed to Audio In before the measurement is performed. Measurement location can and should be a cached property. Since measurement location is the only cached property, it is the only one of the measurement properties that should be included in this class. SINAD, Distortion, and Signal Level will not be stored internally, and there is no reason why they need to appear in the cluster Typedef. The list of properties and methods for this class appears in Figure 10.30.

The analysis and design for the RF analyzer will be very similar to the audio analyzer. All measurement-related properties will not be cached, and the IEEE 488.2 method will not be available to the user through Get and Set methods. The properties that users will be interested in are Power Level, *x*, and *y*. Object Analysis is pretty much complete. The logic regarding the design of this component follows directly from the preceding three objects. The class diagram appears in Figure 10.31.

Now that we have a grasp on what properties and methods the objects have available, it is time to define their interactions. A short series of interaction diagrams will complete the design of the communications analyzer. First, let's consider construction of this object. The constructor will need to create the four component objects in addition to the IEEE 488.2 Instrument. Component constructors will be simple; they do not need to call other subVIs to execute. The IEEE 488.2 Instrument needs to call the constructor for GPIB Instrument. Sequentially, the communications analyzer constructor will need to call the IEEE 488.2 constructor first; this property is handed to the component objects. The IEEE 488.2 constructor will call GPIB Instrument's constructor, and, lastly, the other four component constructors' can be called in any order. The sequence diagram for this chain of events is presented in Figure 10.32.

©2001 CRC Press LLC

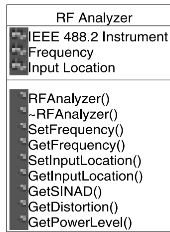

FIGURE 10.30

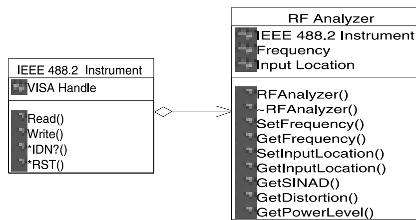

FIGURE 10.31

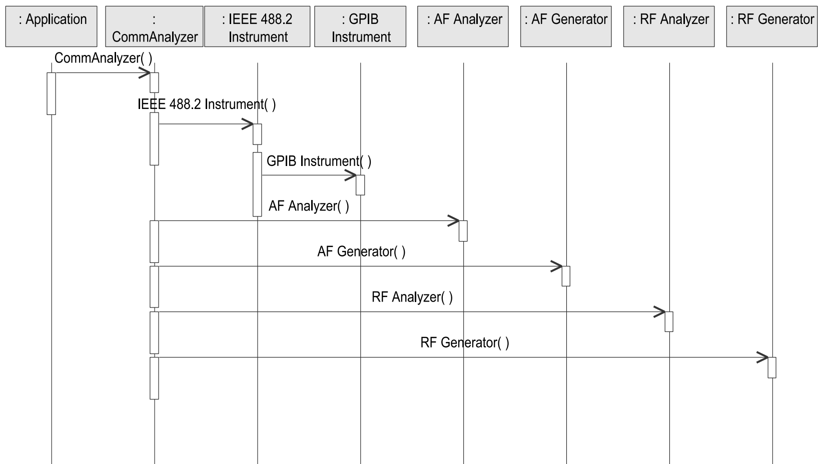

FIGURE 10.32

©2001 CRC Press LLC

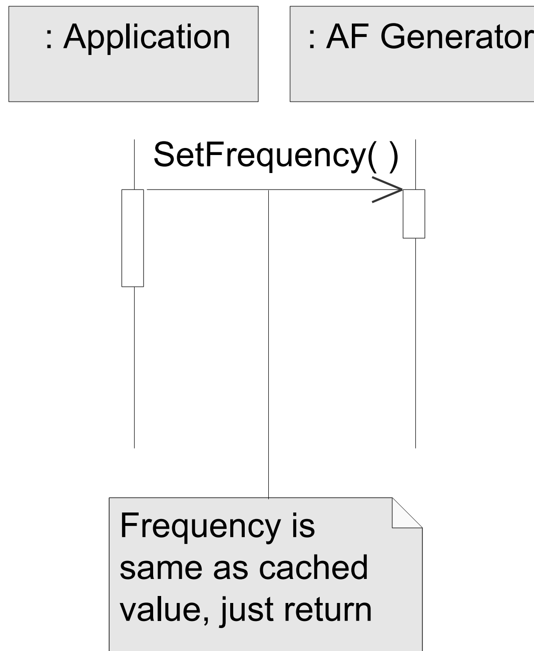

FIGURE 10.33

Figure 10.33 and 10.34 depict how the audio generator should behave when the audio generator's Set Frequency method is invoked. First, we will consider the call chain that should result when the new frequency value is equal to the old value. The method should determine that the property value should not be altered and the function should simply return. Figure 10.33 shows this interaction. Secondly, when the property has changed and we need to send a command to the physical instrument, a nontrivial sequence develops. First is the "outside" object placeholder. Set Frequency will then call IEEE 488.2's Write method to send the string. IEEE 488.2 will, in turn, call GPIB Instrument to send the string. This is a fairly straightforward call stack, but now that we have a description of the logic and sequence diagram there is absolutely no ambiguity for the programming. Figure 10.34 shows the second call sequence. Most of the call sequences for the other methods and properties follow similar procedures.

Now that each class, property, and method have been identified and specified in detail, it is time to complete this example by actually writing the code. Before we start writing the code, an e-mail should be sent to the project manager indicating that another milestone has been achieved. Writing code should be fun. Having a well-documented design will allow programmers to implement the design without having numerous design details creep up on them. Looking at the sequence diagrams, it would appear that we should start writing the constructors for the component objects first. Communication analyzer's constructor requires that these VIs be avail-

©2001 CRC Press LLC

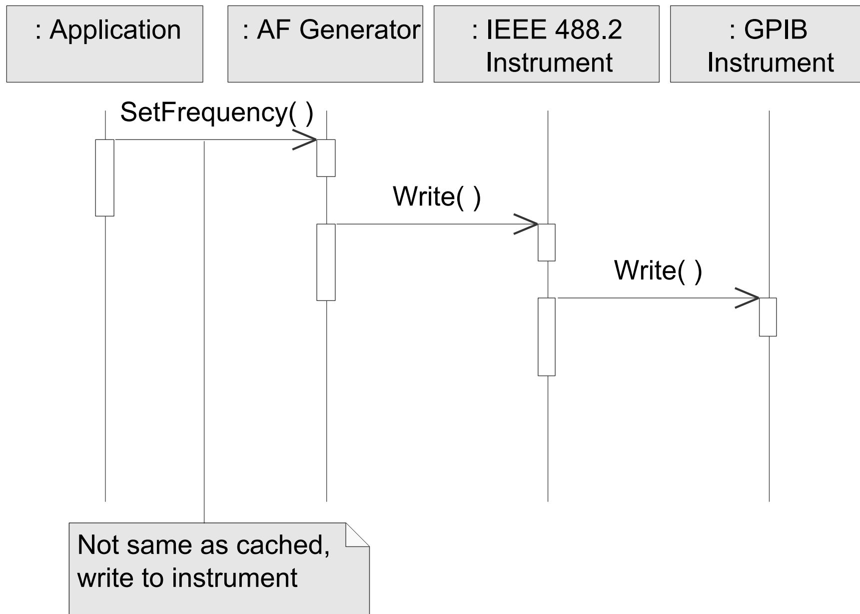

FIGURE 10.34

able, and we have already written and reused the IEEE 488.2 and GPIB Instrument objects. Audio generator's constructor diagram appears in Figure 10.35. The other constructors will be left to the exercises at the end of the chapter. Using skeleton VIs for these three constructors, we are ready to write the communication analyzer's constructor. The error cluster is used to force order of execution and realize the sequence diagram we presented in Figure 10.32. So far, this coding appears to be fairly easy; we are simply following the plan.

In implementing some of the functions for the analyzers and generators, we will encounter a problem. The Get and Set methods require that a particular screen be active before the instrument will accept the command. We left this detail out of the previous analysis and design to illustrate how to handle design issues in the programming phase. If you are collecting software metrics regarding defects, this would be a design defect. Coding should be stopped at this point and solutions should be proposed and considered. Not to worry, we will have two alternatives.

One possible solution is to send the command changing the instrument to the appropriate screen in advance of issuing the measurement command. This is plausible if the extra data on the GPIB bus will not slow the application down considerably. If GPIB bus throughput is not a limiting factor for application speed, this is the easiest fix to put in. Changes need to be made to the description of logic and that is about it. We can finish the coding with this solution.

Another alternative is to include a screen property in the communications analyzer object. This property can cache information regarding which screen is active.

©2001 CRC Press LLC

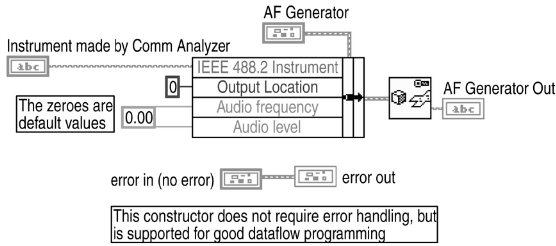

FIGURE 10.35

If this is the case, programmers can Get/Set Screen before invoking the methods of the component objects. This will require us to go back and change all the sequence diagrams, the property list of communications analyzer, and document that users are responsible for the status of the screen.

The last alternative is to have communications analyzer define a global variable that all the component objects can access to check the status of the physical instrument's screen. This solution is going to be tossed out because it is a "flagrant technical foul" according to the object-oriented programming paradigm. Global data such as this can be accessed by anyone in any location of the program. Defensive programming is compromised and there is no way for the component objects to validate that this variable accurately reflects the state of the physical instrument. Having considered these three alternatives, we will go with the first and easiest solution. Measurement settling times will be orders of magnitude longer than the amount of time to send the screen change command. The transit time for the screen command is close to negligible for many applications. With this simple solution we present Figure 10.36, the code diagram for setting the audio generator's frequency. Remaining properties are implemented in a similar fashion.

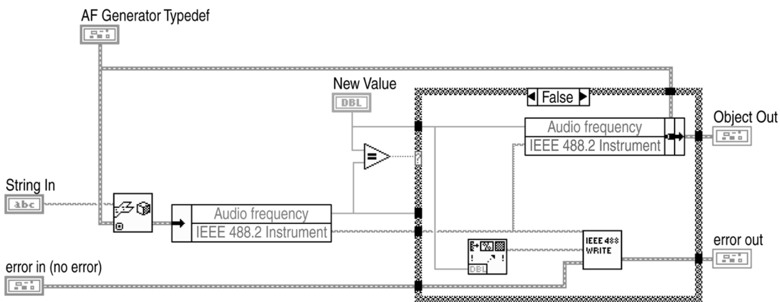

FIGURE 10.36

©2001 CRC Press LLC

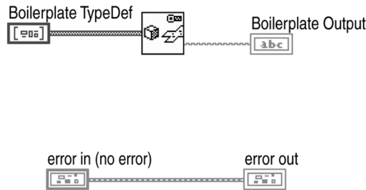

FIGURE 10.37

10.8 OBJECT TEMPLATE

Many of the VIs that need to be implemented for an object design need to be custom written, but we can have a few standard templates to simplify the work we need to do. Additionally, objects we have written will begin to comprise a "trusted code base." Like other collections of VIs, such as instrument drivers, a library of VIs will allow us to gain some reduction in development time by reuse. The template VIs built in this section are available on the companion CD.

The template for constructors needs to be kept simple. Since each constructor will take different inputs and use a different Typedef cluster, all we can do for the boilerplate is get the Flatten to String and Output terminals wired. Each constructor should return a string for the object and an error cluster. Simple objects such as our geometric shapes may not have any significant error information to return, but complex classes that encapsulate TCP conversations, Active X refnums, or VISA handles would want to use the error cluster information. Figure 10.37 shows the template constructor.

Get properties will be the next template we set up. We know that a string will be handed into the control. It will unflatten the string into the Typedef cluster and then access the property. This operation can be put into a VI template. It will not be executable, since we do not know what the Typedef cluster will be in advance, but a majority of the "grunt work" can be performed in advance. Since return types can be primitive or complex data types, we will build several templates for the more common types. Figure 10.38 shows the Get template for string values. This template will be useful for returning aggregated objects in addition to generic strings. Error clusters are used to allow us to validate the string reference to the object.

Set property templates will work very much like Get templates. Since we do not have advance information about what the cluster type definition will be, these VIs will not be executable until we insert the correct type definition. Again, several templates will be designed for common data types. Error clusters are used to validate that the string handed into the template is the correct flattened cluster.

©2001 CRC Press LLC

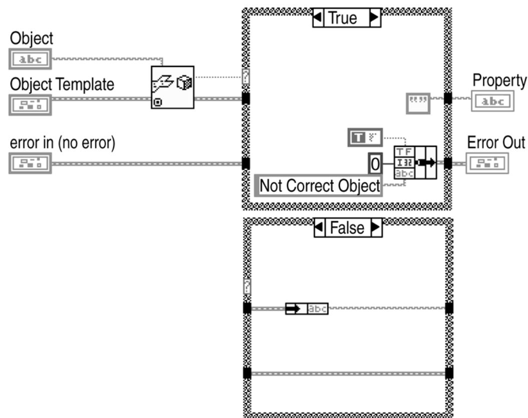

FIGURE 10.38

10.9 EXERCISES

1. An object-based application for employee costs needs to be developed. Relevant cost items are salary, health insurance, dental insurance, and computer lease costs.
2. What properties should be included for a signal simulation object? The application will be used for a mathematical analysis of sinusoidal signals.
3. Construct an object-based instrument driver for a triple-output power supply based on the example power supply in Section 10.7.

BIBLIOGRAPHY

Bertand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1998.

©2001 CRC Press LLC