



What is GOOP?

A Demonstration of Basic GOOP Principles

by David Hoadley, PhD

A different way to think about LabVIEW™ programming is gaining momentum among developers. This article describes the benefits of graphical object-oriented (OO) programming – GOOP.

The prevailing design paradigm in the LabVIEW community is top-down, sometimes called functional *decomposition*. Decomposition refers to the process of reducing the whole programming task into a series of easily implemented components.

An object-oriented (OO) program is decomposed into a collaboration between subcomponents called *objects*. Objects are named subprograms with attributes or properties (data) and methods (functions). Method calls provide the interface between objects. An object's interface can contain public (usable by any component that can identify the object) or private (usable by only other similar objects) methods. A *class* describes a collection of similar objects, or the data type that defines all of the attributes and methods of the member objects. One or more objects can be created from a class definition.

Think of objects as a means to divide the functionality of a system into logical and meaningful components, rather than just abstract components of an application. The closer those

components mimic the real behavior of the system, while still providing a model from which an application can be crafted, the more power the OO design paradigm has.

Benefits of Object-Oriented Design

The most significant benefits to OO designs are increases in scalability, reusability, ease of implementation, and lowered maintenance costs.

For more background information about the benefits of object-oriented LabVIEW programming, see Robert Buhrke's "G++ with ObjectVIEW" article (*LTR Volume 9, Number 3*) and Application Note 143 available from www.ni.com.

Scalability and Maintenance

A typical non-object-oriented program tends to have a complex relationship structure among its subcomponents. In a small application, this does not present much of a problem. As applications grow, the number of subprograms increases directly, but the number of interconnections between components tends to grow at a much faster rate, as shown in *Figure 1*. Eventually, changes to the application can result in so many unexpected side effects that a complete rewrite becomes a preferable option to integrating new features. OO applications are far more scalable due to the loose associations between code modules. *Figure 2* is

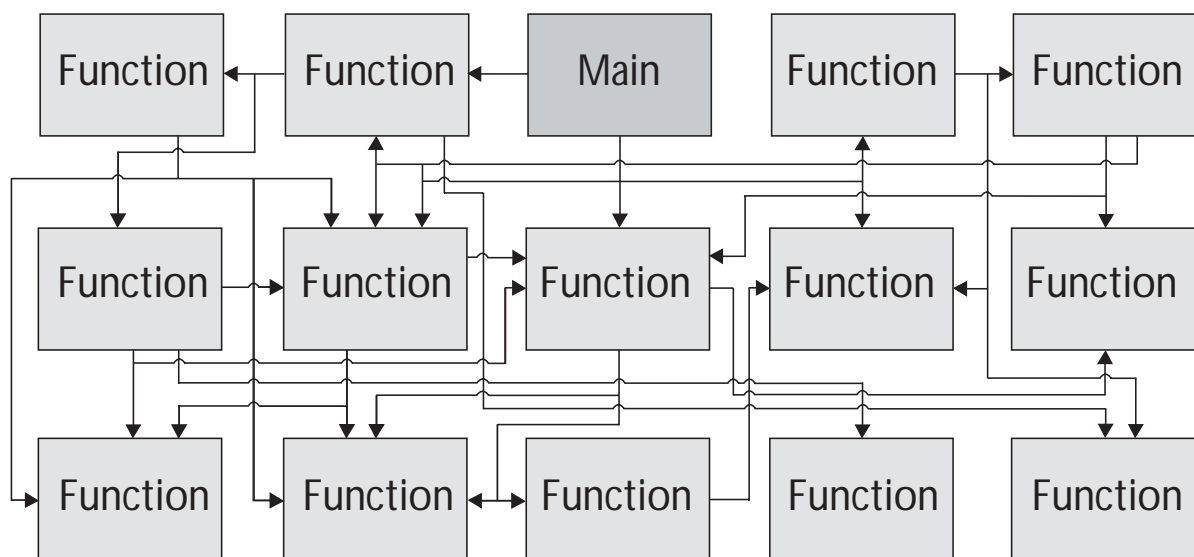


Figure 1: Functionally Decomposed Design

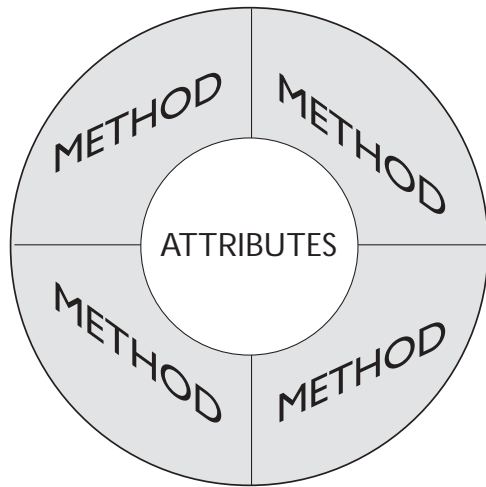


Figure 2: Object Within an Object Oriented Design

a schematic representation of an object. Its core is comprised of a private, encapsulated data space and private methods. The interface from the module to any other object is represented by a series of public method calls that provide the interface to the object, enforcing controlled associations. This series of programming interfaces limits the scope of interdependencies and provides for a reliable application growth mechanism.

Design layers can reduce the number of interdependencies among components. For example, consider an application

where a number of different instruments are used to make measurements. A good approach is to isolate the data analysis or logging layer from the instrumentation layer. The instrumentation layer can provide a standard mechanism for accessing different types of hardware. The intermediate layer protects the upper and lower levels from interdependencies.

Another limitation of functionally decomposed programs becomes apparent, however, when we look at the way data is managed in a large layered application, as shown in *Figure 3*. Even in a good design, it can be difficult to avoid global data store components.

OO code provides an inherent mechanism for data management. The identity of an object always refers to a particular unique data set – the attributes of the named object. See *Figure 4* for a schematic representation of an OO application design.

Finally, a good OO design makes it easier to train a new developer on the functionality of the program. Once an overview of the object collaborations is presented, only the deep details of any class that needs modification must be understood by the additional developer in order for one to become productive.

Code Reuse

Data dependencies from application layers to global data spaces limit the reuse potential of code. Mining an application for reusable components can be a daunting exercise due to the interdependencies. Reuse is easier with OO code because of

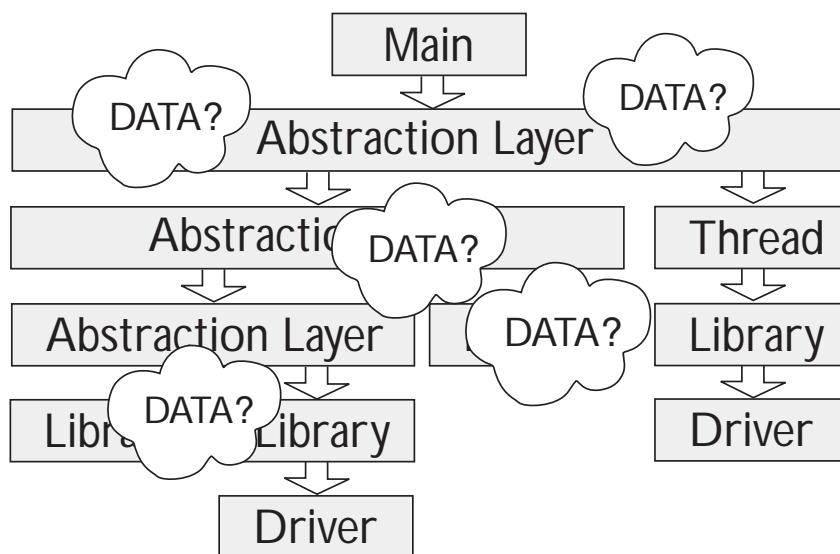


Figure 3: Large Layered Application Without OO Design

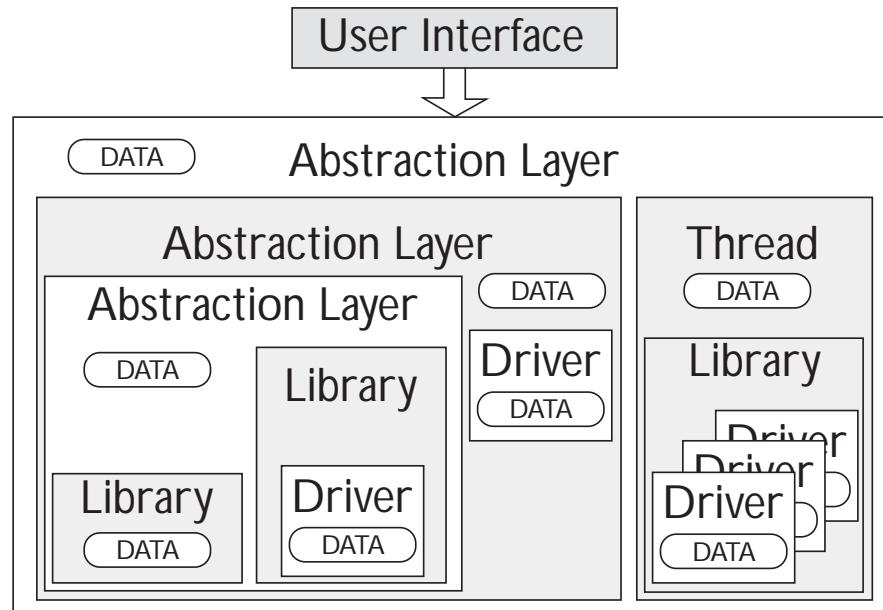


Figure 4: Large Application with OO Design

the loose associations. The independent data sets provide a means to quickly add support for additional objects. Finally, a well-defined public method interface for an object means that it is easy to determine the functionality of a reusable component, limiting developer training requirements. Modifications to the object's data set occur in a class-defined way, limiting the risk of misuse. Reducing the understanding of what happens deep within the private members of the class makes it more likely for code to be shared among a team of developers or to still retain some value to the original developer many projects from now.

Ease of Development

A daunting prospect in a large project is that there may be no single developer who understands all of the functionality required by the system. Often it is difficult to divide tasks along functional lines for multiple developers. For example, giving someone responsibility for a file abstraction layer means that that developer may have to interact with numerous other teams – those creating a recipe editor for the process control, those managing user authentication, those needing to store default values for GUI entries, the data acquisition team, and so on.

By carving up an application into classes, a natural division of labor arises. An OO design for an application specifies each class' attributes, method calls, and relationships to other classes. It then becomes simpler for developers to

work together developing classes for drivers, user interfaces, and other logical components which are more independent of one another than subprograms in a functionally decomposed application.

Think of objects as a means to divide the functionality of a system into logical and meaningful components, rather than just abstract components of an application.

Another benefit is that each class can be debugged with simple test programs that exercise each method. Often no more than five VIs are needed: Create, SetData, MethodA, GetData, and Destroy. The inevitable integration issues are minimized by the thorough testing of smaller modules now possible throughout the application hierarchy, however. This holds true as long as the method interfaces and requirements were appropriately determined and adhered to by the developers.



OO Programming in LabVIEW

Object-oriented tools for LabVIEW are readily available. In fact, some features of the LabVIEW Functions palette are classes.

The LabVIEW File I/O palette VIs represent a class definition for file objects. When an object is created (opened), attributes such as the file path, data size, and access rights are initialized. A pointer to the current location in the file is also set to its first byte. Later in a program, methods such as read and write file provide functionality and update attribute values. All that is needed to perform these methods are method-specific parameters and the identity (refnum) of the object. Finally, when processing is complete, the object is not needed and is destroyed. There is never any doubt which object is the target of any method call, as long as we keep the identities straight in our program. Another example is the Queues palette under the Synchronization tools.

Looking at the structure of these classes, we begin to see how to develop OO code in LabVIEW. Methods are VIs that operate on the objects. Objects are identified by reference, such as a name or a refnum. The preferred method to provide this identity in LabVIEW has become to define a new refnum data type for each class by placing an enumeration control inside a datalog refnum control. LabVIEW will provide a broken run arrow if refnums from different classes are mis-wired to method VIs. The value of the refnum becomes our unique identifier for each object. Finally, there is an object repository that stores the attributes for all of the objects in a class.

Demonstration

For a hands-on demonstration of an object-oriented LabVIEW program, refer to the CD Jukebox.vi example on this issue's LTR Resource CD.

Two classes collaborate to implement this system: CD and CD Player. In this example, a class is realized as the contents of a directory of VIs and controls. Let us look at a functional overview of each main type of file for the CD class as an example.

CD Class

The subdirectory */CD* contains the public members of the class. The Create and Destroy VIs manage the lifetime of the objects, and the other methods modify object data to provide functionality. Typically, all attributes of the object are private. All public VIs require access to components in the */CD/private* directory.

The CD class manages attributes such as the CD title (the object name), the names of all tracks on the disc, and the

position of the CD in the jukebox in the object repository. The application maintains a CD database by creating objects whenever the user selects **Load a CD** from the front panel and deleting them when **Eject** is used. Accessing a CD to gain information is accomplished by public method VIs in the CD directory. The CD Jukebox.vi application needs only to maintain an array of CD objects on its own block diagram.

CD Player Class

CD Player is a class with a many-to-many association with the CD class. The attributes of a CD Player are Title (a reference to a CD object), Track Number, Elapsed Time(s), Mode, and a VI reference called GUI.

The Create and Destroy VIs implement the typical constructor/destructor pair. One interesting thing to note is that we are creating not only a CD Player object, but we also create a GUI cloned from a template VI (see "Cloning Front Panels" by Allen C. Smith in *LTR Volume 6, Number 3*) in the CD Player/private directory: CD Player_GUI.vit. Because data space and user interfaces are created on demand for this class, the only limitation to one CD Player is our CD Jukebox application.

CD Jukebox Application

The first thing to note is how thin this top level application is, as shown in *Figure 5*. The CD Jukebox.vi is little more than an event handling state machine with some shift registers storing references to objects. All functionality is achieved by using public methods in the CD and CD Player classes.

On program start, one CD Player is created and a clone GUI is displayed by the CD Player class. To run the demo, first populate an array of CD references by entering data for the Title, Tracks, and Track Length controls. Then, select Load a CD. If the CD title is unique, a new CD object is created with the CD_Create.vi method. Its attributes are initialized to the values specified on the CD Jukebox front panel, and its refnum is added to the array of CD references in the appropriate shift register on the CD Jukebox.vi block diagram. Once one or more CDs have been created, you can select a title and track in the list boxes and control the CD player appropriately.

Tools

The LabVIEW classes used here were developed from the VISTA GOOP Templates (formerly Class Generator) using Endevo's GOOP Inheritance Toolkit (GIT). To provide a transparent implementation of GOOP for the article (all source code including the object repository is available for your review), advanced features of the GIT were replaced by using external templates. The GIT features are described in the "Inheritance in GOOP" article by Mattias Ericsson and

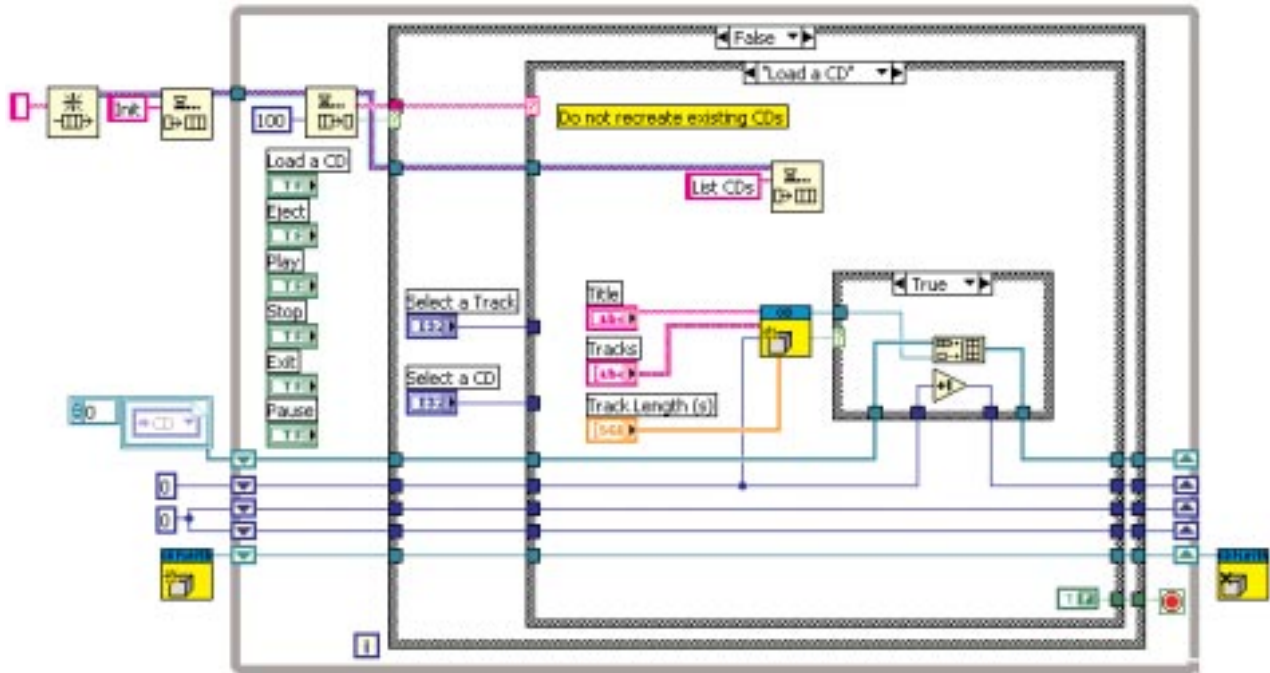


Figure 5: CD Jukebox VI

Jan Klasson on page 19 of this LTR issue. ObjectVIEW from Vogel Automatisierungstechnik GmbH and OpenGOOP, courtesy of the OpenG organization, are other GOOP toolkits. The NI™ LabVIEW GOOP Wizard, the precursor to the GIT, was developed collaboratively with Endevo.

Being Productive

While it is not simple to switch from functional, top-down design to OO techniques, training is available from VI Engineering's "System Design With GOOP" course. For more information visit viengineering.com or contact Wyatt Meek at wmeek@viengineering.com. Even though it takes practice and preparation to become a good OO designer, there are some simple guidelines that will help as you begin to build classes for LabVIEW projects. Once you are ready to propose a design, consider these questions:

- Do the classes and objects really represent the system well?
- Is the design modular, with numerous collaborating small classes rather than a few huge classes with most of the program's methods?
- Have you used sparingly classes which are just storage or basically implement a wrapper around a built-in data type (bool state with methods setstate, getstate)? If this is the case, you added a lot of overhead VIs in order to recreate a LabVIEW global variable.

Conclusion

For further information on GOOP, be sure to check www.zone.ni.com, OpenG.org, Endevo.se, and viengineering.com. Two good texts on OO analysis and design are *Object-Oriented Analysis and Design, 2nd Ed.* by Grady Booch, and *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd Ed.* by Craig Larman.

About the author:

David Hoadley, PhD is Chief Technology Officer at VI Engineering in Farmington Hills Michigan. Dr. Hoadley earned a B.S. in Astrophysics and Mathematics as well as a Masters Degree and Doctorate in Physics from Michigan State University. He has published numerous technical articles and conference papers, and contributed to the book "LabVIEW for Automotive, Telecommunications, Semiconductor, Biomedical and Other Applications" (Prentice Hall). David is also an instructor for VI Engineering's "System Design with GOOP" course, a Certified LabVIEW Developer and contributes his expertise to systems integrations projects for automotive, life science and aerospace applications.