CONTROL SYSTEM DESIGN USING LABVIEW OBJECT ORIENTED PROGRAMMING

D. Beck, H. Brand, GSI, Planckstraße 1, D-64291 Darmstadt, Germany

Abstract

In 2006, LabVIEW Object Oriented Programming (LVOOP) became available as a new feature with LabVIEW version 8.20. This work accomplishes a design study in order to investigate the use of LVOOP to control system development. With LVOOP, the way of object-oriented programming must be reconsidered, since this approach reveals quite a few differences compared to conventional object-oriented programming.

INTRODUCTION

Concept of "Dataflow"

Compared to text based languages, the graphical programming language LabVIEW reveals a fundamental difference which is the so-called paradigm of "Dataflow" as indicated in Fig. 1.

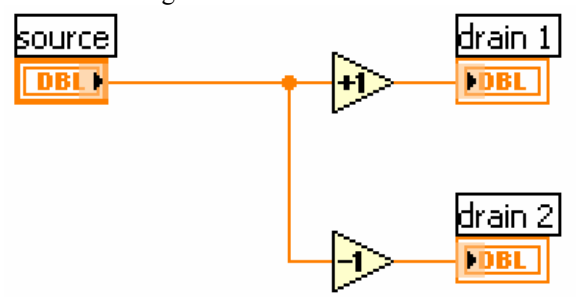

Figure 1: Concept of "Dataflow".

Data flow through a wire from source to drain. There are different types of wires for the respective data types (integer, double, string, etc.). As shown in Fig 1., a source can be a so-called "control". The emerging data can be a copy of the default value of that control or it can be a value that has been passed to the control, in case when the code shown in Fig. 1 has been called by some other piece of code. As an elementary consequence, the developer of the code above has no control when and where memory is allocated for the data flowing through the wire. The well known concept of variables does not apply to LabVIEW. Classical variables do not exist.

In Fig. 1, one source of data is shown together with two drains of data that are connected by a so-called "wire fork". To be consistent with dataflow, wires use "by-value" syntax. When a wire forks, its value is duplicated. On the one hand, duplicating data requires more memory. On the other hand, the "by-value" syntax implies the value on each individual wire to be independent from all other wires. Each wire together with the operations along that particular wire represents an own thread. By this, the concept of dataflow implies a natural parallelism and is inherently thread-safe due to the "by-value" concept.

Concept of LVOOP

Object oriented programming with LabVIEW extends the concept of "Dataflow" to the concept of "Objectflow" [1]. A class in LVOOP consists merely of a user defined data type together with methods that can be applied to values of that data type. Once could say that LVOOP allows the developer to create object oriented wires. Object orientation in LVOOP means the following.

1. Simple Inheritance. Neither multiple inheritance nor interfaces as in Java.
2. Strict encapsulation. Data of a class are always private. Public or protected data do not exist.

As in other object oriented languages, a derived class may overload (naming convention of LVOOP: "override") an abstract method of its base class. However, the override method must have exactly the same input and output parameters as the respective method of the parent class.

The concept of dataflow requires objects to flow through their "class wires" from source to drain, like the flow of data in Fig. 1. This has three fundamental consequences.

1. Objects contain only data and no active code. Agents do not exist.
2. LabVIEW does not have classical variables. For the same reason, LVOOP has no equivalence to the concept of a *constructor* and a *destructor*. There are neither constructors nor destructors.
3. Objects can only be accessed "by value" and never "by reference".

The fundamental differences between LVOOP and conventional object oriented languages prevent a straight-forward *implementation* of design patterns that are based on the idea of objects as entities [2]. However, many of those design patterns are useful for designing control systems, and need to be *reinvented* with respect to the dataflow paradigm.

DESIGN PATTERNS

Reference Pattern

The same object can not be used by two independent wires. A wire fork, as shown in Fig. 1, just creates an identical copy of the object flowing from the source to the two drains. In order to profit from the inherent parallelism in LabVIEW, the reference pattern is required.

The reference pattern makes use of one of the event mechanisms that is part of LabVIEW. A so called *message queue* is a shared resource that is identified via a reference. It provides a buffer allowing a sender to feed data into the queue from one thread. A receiver waits in a

second thread to obtain data from the same queue. Since LVOOP objects are only data, they can flow from a source into the message queue and can be retrieved somewhere else. This situation is depicted in Fig. 2.

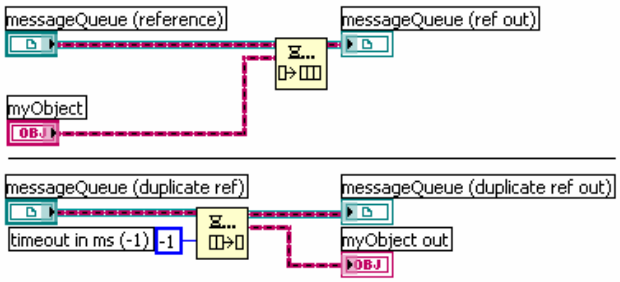

Figure 2: Idea of a referenced object Top: Insert an object to a message queue. Bottom: Retrieve an object from a message queue.

In case there is no second thread retrieving the object immediately, the object will remain in the queue until it is retrieved or the program terminates.

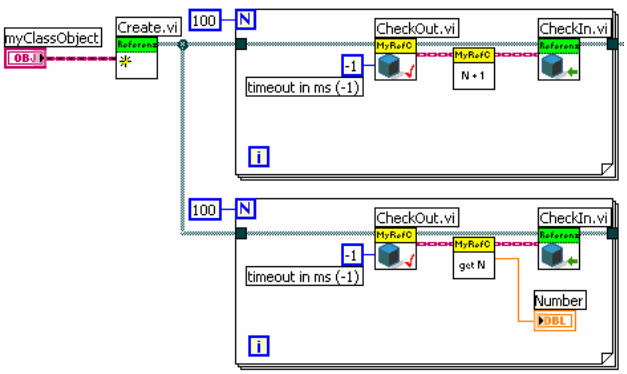

Figure 3: Usage of the reference pattern.

Fig. 3 depicts an example for using a *reference* class that is part of the reference pattern. The *reference* class provides *create*, *checkout* and *checkin* methods. An object flows into the *create* method where a *reference* object is created and the object is inserted into a message queue (see Fig. 2, top). The reference object contains and protects the reference of the message queue as private data. The wire fork in Fig. 3 duplicates the *reference* object, but not the original object which is in the message queue. Two identical copies of the *reference* object are now used in two threads represented by two loops. Each time the loop iterates, the respective thread uses the *checkout* method and must wait, until the object is available in the message queue. It performs an action on the object and re-inserts the object into the queue using the *checkin* method. To conclude, the reference pattern allows using one object in different threads and provides mutual exclusion.

Factory Pattern

The factory pattern allows for creating and initializing objects of any class at runtime. An overview on the factory pattern is depicted in Fig. 4.

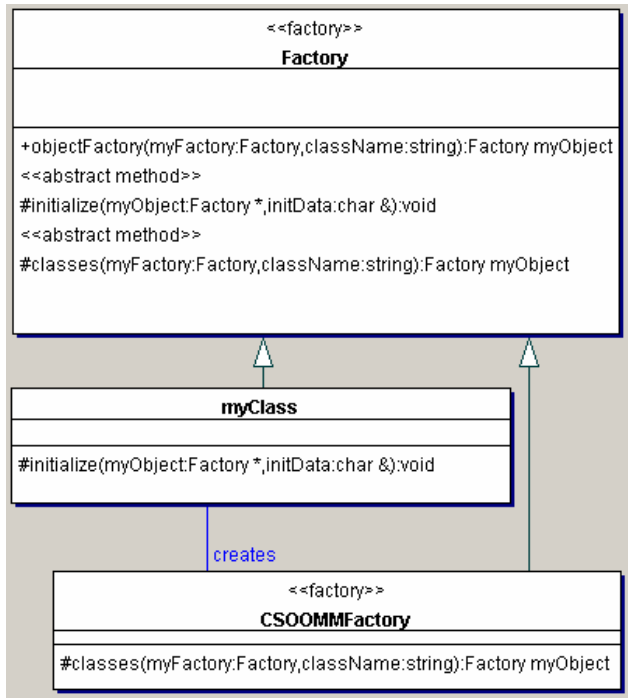

Figure 4: Factory pattern.

Shown are three classes, the *factory* class, an application specific factory *CSOOMMFactory* and one example of a class *myClass*. For producing objects, *myClass* must be derived from the *factory* class and must implement the override method *initialize*. The *CSOOMMfactory* class is also derived from the *factory* class and need to implement the override method *classes* which contains an object constant for each class (here: *myClass*) to be produced.

DESIGN STUDY *CSOOMM*

This design study aims at setting up a simple, distributed, event-driven and object oriented control system by combining LVOOP with DIM as communication layer.

Object Management

LVOOP does not provide an object management. Within *CSOOMM*, this is implemented by joining the reference pattern and the factory pattern. An *objectManager* library provides *_new* and *_delete* routines as well as routines to convert an object name into its reference and vice versa. An example for using the *_new* operator is shown in Fig. 5.

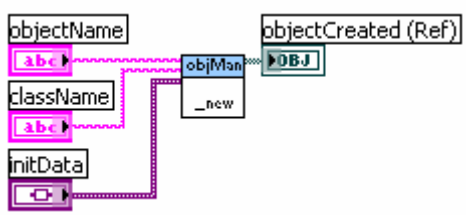

Figure 5: Creating a referenced object.

Connecting to DIM

The *dimProcessor* class is one of the base classes within *CSOOMM* and provides and encapsulates the interface to DIM [3, 4]. For ease of use, it is restricted to two data types namely arrays of double values and strings. The most important features of this class are

1. declaring and publishing DIM services,
2. declaring DIM commands,
3. create a periodic action timer,
4. an abstract method for handling received commands,
5. an abstract method for handling updated services, and
6. an abstract method for handling periodic actions.

The abstract methods of the *dimProcessor* class correspond to the call-back methods that can be used in the DIM environment.

Application Example

It is not possible to call LabVIEW code from other applications directly. Thus, the abstract methods of the *dimProcessor* class can not be used for direct call-back by the shared libraries of DIM. As objects in LVOOP are passive and provide no active threads, active LabVIEW code is required as an additional step and must be provided by the application program.

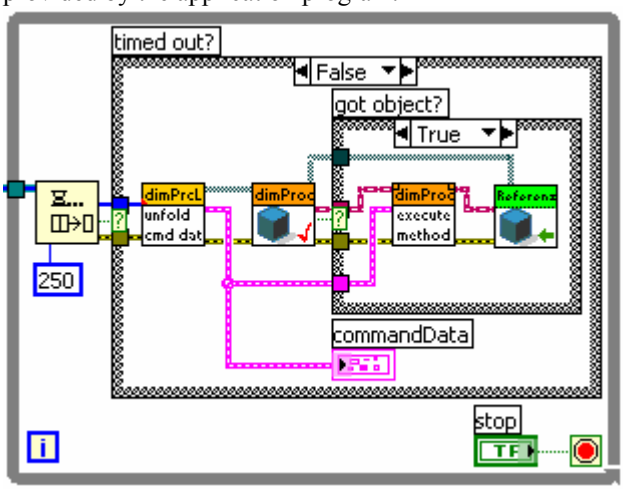

Figure 6: Example application for handling DIM commands.

Fig. 6 shows an example application as LabVIEW code that demonstrates the handling of DIM commands in a while-loop. The following is shown from left to right.

1. Wait, to receive a DIM command buffer via a message queue,
2. unpack the command buffer, get the respective *reference* object via the object management tools, unpack the command data,
3. *checkout* the object,
4. abstract method for handling commands of the *dimProcessor* class, this will call the respective override method of the class of the object, and
5. *checkin* the object.

Please note, that the checked-out object is typecasted to an object of the *dimProcessor* class. Thus, the code is class independent, as long as the class of the checked-out object has been derived from the *dimProcessor* class. The shown example demonstrates the re-usability and simplification of code, which results from using abstract methods provided by LVOOP.

STATUS AND CONCLUSION

A design study has been performed to successfully test the applicability of LVOOP to designing control systems. Testing of performance yielded the following results. Attribute data of classes are always private. However, the memory management of LVOOP is well done and accessing those data via class members can be accomplished quickly, even if the amount of data exceeds 10Mbyte. It is easily possible to instantiate more than 1000 device objects per PC, which corresponds to about 1000 physical devices per PC. When designing a control system, the use of abstract methods of base classes simplifies coding and greatly enhances reusability. As the only draw-back of LVOOP in LabVIEW version 8.20, there is no code sharing when executing the override methods to abstract methods. This restricts the usefulness of LVOOP in multi-threaded applications. However, this problem seems to be solved with the newer version of LabVIEW 8.5.

Although this design study demonstrates the usefulness of LVOOP successfully, this approach must be confirmed by application to a real control system like, as an example, the *CS*-framework [5].

REFERENCES

[1] Tutorium by National Instruments: "LabVIEW Object-Oriented Programming: The Decisions Behind the Design". http://zone.ni.com/devzone/cda/tut/p/id/3574.

[2] E. Gamma et al., Addison-Wesley (1995) ISBN 0-201-63361-2.

[3] C. Gaspar and M. Dönszelmann, "DIM - A Distributed Information Management System for the Delphi experiment at CERN", Proc. IEEE Eight Conference REAL TIME '93, Vancouver, Canada, 8.-11. June 1993.

[4] D. Beck et al., "LabVIEW-DIM Interface", Proc. "Virtuelle Instrumente in der Praxis 2005", VIP 2005, Fürstenfeldbruck, Germany, Editors R. Jamal and H. Jaschinski, ISBN 3-7785-2947-1 (2005) 20-26.

[5] D. Beck et al., Nucl. Instrum. Meth. A 527 (2004) 567-579, http://wiki.gsi.de/CSframework .