

# A GENERIC OBJECT-ORIENTED IMPLEMENTATION FOR FLIGHT CONTROL SYSTEMS

Patricia C. Glaab, Michael M. Madden\*

Unisys Corporation  
NASA Langley Research Center  
Mail Stop 125B  
Hampton, VA 23681

## Abstract

This paper presents a design for a generic flight control system (FCS) architecture that breaks the control system into a coupled interaction of laws and devices with a standardized method for execution. Laws generally are computational components that generate commands as outputs, and any number of laws can be isolated and registered on a list in any order for execution. Control devices are code components that receive the command inputs and use additional computations to generate device outputs, such as the servoactuator positions for the control of surfaces. Any number of devices are allowable for a given flight control system and are registered in list format for execution. This method allows for both simplistic FCS implementations and highly complex control systems without changing the architectural requirements of the high level executive.

By separating the laws from the devices, special case handling required for control law bypassing (such as that required for direct-drive surface testing and linear analysis) is easily handled at the execution level. No special code support is required internal to the laws.

## Introduction

This paper presents a method for structuring flight control system code as part of an object-oriented simulation framework. The design presented was developed at NASA Langley Research Center by

Unisys Corporation for use within the Langley Standard Real-Time Simulation in C++ (LaSRS++). The goals for the original design included several objectives. The first objective was to develop a common software heirarchy that could effectively handle FCS requirements for different styles of aircraft. The parent classes for the elements would reside in the simulation framework. Ideally, this structure would simplify installation of new FCS models being added while providing a common execution style that only required one documentation effort.

A second objective was to allow special FCS processing as part of the common executive. Specifically, direct manipulation of surface positions was required for linear analysis and open-loop checkcase matching. This design was developed to allow this type of operation without embedding special handling instructions in the aircraft-specific FCS code.

To achieve the solution, an architecture was developed that reduces a given FCS to a collection of *laws* and *devices* and treats them as separate components. Once this is done, the common elements in seemingly unique control systems becomes apparent. The flexibility of the system is maintained through the use of variable length lists to control the laws and devices. This method effectively accommodates aircraft with varying ranges of complexity and sizes of models.

## Design Objectives

Aircraft that differ in both form and function necessarily have unique technical requirements for their flight controls. The differences between a military fighter aircraft FCS and that of a commercial

---

Copyright ©1998 by the authors. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

transport at first inspection may seem more profound than the commonality. Yet, in order to satisfy the design goal, a methodology that serviced each style effectively was required.

In this paper, the implementations of an F16A and a Boeing 757 aircraft are used to demonstrate the application of the final design to two very different aircraft models. The F16A simulation model uses a low-speed subset of that aircraft's control laws. The FCS outputs one signal for aileron, stabilator, leading edge flap, speedbrake, and rudder as required to interface to the supplied aerodynamics model. The 757 simulation model uses a full-envelope FCS model and supplies surface deflections for left and right ailerons, elevators, flaps, and slats, a single rudder and stabilizer, and twelve spoiler deflections as required for its aerodynamics model. The 757 uses a complex servoactuator model with hinge moments, hydraulic system options, and variable rate and position limits. The F16A uses a simple first-order servo calculation with fixed rate and position limiting. The F16A computes its commands to surfaces using only a longitudinal, lateral, and directional control law. The 757 also uses laws for the three axes, and additionally uses a high lift, a spoiler, and a yaw damper control law. Yet both are very effectively modeled using the architecture described here.

The approach used in the design of this FCS architecture was to ignore the specific type of surfaces (or devices) and the intricacies of generating the specific commands to drive them, and to reduce the level of complexity to a group of *laws* that act upon a group of *devices*. In this way, the software is modeled similar to the actual aircraft. In an actual aircraft, some combination of pilot inputs and synthesized signals from a flight control computer are fed to the control surface servos. The actual deflection that results from the command may depend on the current aircraft states or environmental conditions, depending on the complexity of the simulation model. The derivation of commands and the surface movements can be treated as separate entities.

Once this separation of laws and devices is made, the similarities between very different styles of FCS become quite apparent.

## Laws

A control law is the set of computations that provide a command as an output. The internal computations may represent flight control computer modeling,

mechanical linkages, etc. Inputs to control laws are generally pilot inputs from the cockpit and aircraft state variables. The outputs are the commands to the devices.

Control laws may be loosely or tightly coupled, depending on the aircraft model. In a very simplistic configuration (such as a general aviation aircraft without any autopilot), the laws may be simply the interaction of mechanical connections that transform pilot command inputs into commanded force or deflection outputs. In a more complex model, a stability augmentation system or a flight control computer may generate or modify commands.

This FCS class architecture design cares neither about the number of laws used nor the internal complexity. The parent *ControlLaw* class, which resides in the general framework, defines methods required by client aircraft-specific control law classes which inherit from it. By maintaining a variable length list of laws, *ControlSystem* simply operates upon each law registered to it at the referenced location. In the C++ implementation, the aircraft-specific FCS instantiates the laws required and registers a pointer for each law during construction.

*ControlSystem* contains a pure virtual function called *execute()* which must be specifically defined by each client control law. Making the method pure virtual insures definition of the method within each client. During FCS processing, the control system executive steps through its list of registered laws and simply calls the *execute()* method for each. Varying number of control laws are accommodated for different aircraft models with the same high level controller code.

## Devices

A device is a software representation of what would be hardware on an actual airplane. Surfaces are the primary type of device used, but a device can be any aircraft component that augments the flight. Side-thrusters would be another device example. Inputs to devices are usually a command signal and state variables or environmental variables that effect the device's movement. Most modern devices use some type of servoactuators to achieve output deflections.

The flexibility for number and complexity of devices is maintained similarly by a variable length list maintained by the parent *ControlSystem* class. In the C++ implementation, the aircraft-specific FCS instantiates the number and type of devices it requires for its model and registers a pointer for each to the

device list at construction time. The *ControlDevice* parent class, from which all control devices inherit, contains a pure virtual method called *drive()* which must be defined by each client device class. During execution, the *ControlSystem* executive steps through its list of devices and calls the *drive()* method for each.

### Modeling of Common Behaviors

The computational behaviors common to FCS code structure and execution are also contained within the parent *ControlSystem* class. Every FCS must provide inputs to its laws, execute the laws, provide inputs to its devices, and drive them. In the standard execution, these are done by the *ControlSystem* class. The *ControlSystem*, from which the aircraft-specific FCS inherits, defines the pure virtual methods *directInputsToControlLaws()* and *directInputsToControlDevices()*. These must be

redefined by each airplane to fit its own unique requirements.

Additionally, the methods *evaluateControlLaws()* and *driveControlDevices()* are defined within the parent *ControlSystem* class and contain the code to step through the list of laws and devices. These need not be refined at the aircraft-specific level. They are, however, virtual methods that may be redefined if some unusual execution is required.

One other method is provided as virtual by the *ControlSystem* class for use at the client's discretion. The *compositeDeviceCalculations()* method is provided to compute combinations of device outputs required for other systems in the simulation. For example, in the 757 aircraft, this method is used to compute *slat\_average* and *elevator\_differential* which are exported to the aerodynamics model and the data recording model.

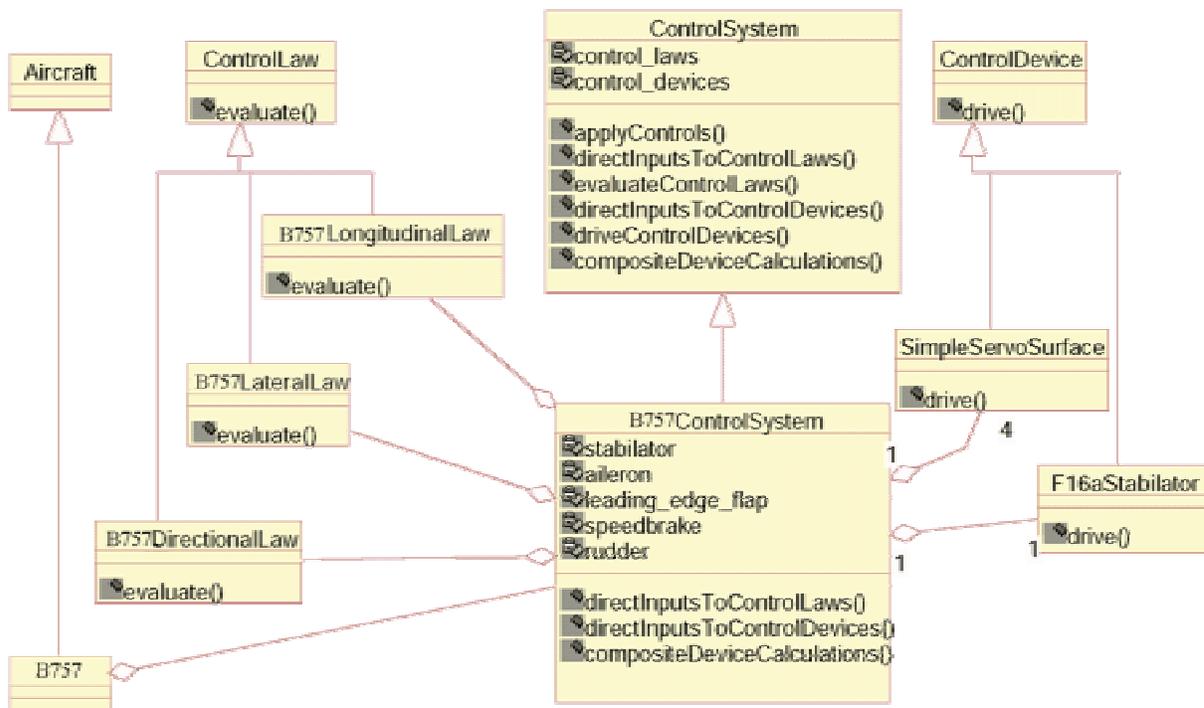


Figure 1- 757 FCS Class Structure

### Execution Options

Since this methodology was developed to support special processing requirements for linear analysis and open loop checkcase matching, alternate processing flexibility was built into the architecture

at the parent class level. A flag is provided by the *ControlSystem* class called *open\_loop*. When the *open\_loop* flag is set true, an alternate method of execution is followed. During open-loop processing, execution is defined in its entirety by the client aircraft control system via a virtual method called

*useTrimCommandsAsResponse()*. Within the client's version of this method, commands or deflections can be directly assigned as necessary for special handling. The executive section is simply defined in *useTrimCommandsAsResponse()*, and the *open\_loop* flag set true.

### Inheritance Structure

Figures 1 and 2 show the inheritance structure for the 757 commercial transport and the F16A military fighter within the generic FCS architecture. The top-most boxes on each diagram represent the framework classes: *ControlSystem*, *ControlDevice*, *ControlLaw* and *Aircraft*. Here the *B757* executive class, which inherits from *Aircraft*, has 21 control surfaces and 5

control laws to generate its model. The *F16A*, by comparison, only uses three control laws and five control surfaces to define its FCS model. The pure virtual methods, *evaluate()* and *drive()*, initially defined in the *ControlLaw* and *ControlDevice* classes respectively, are redefined in the aircraft-specific classes which inherit from them. Pure virtual classes *directInputsToControlDevices()* and *directInputsToControlLaws()*, initially defined in *ControlSystem*, are redefined for the aircraft-specific control system classes. Since each aircraft requires combined surface outputs, each redefines the *compositeDeviceCalculations()* method. Neither, however, defines *evaluateControlLaws()* or *driveControlDevices()* which are handled entirely by the parent class, *ControlSystem*.

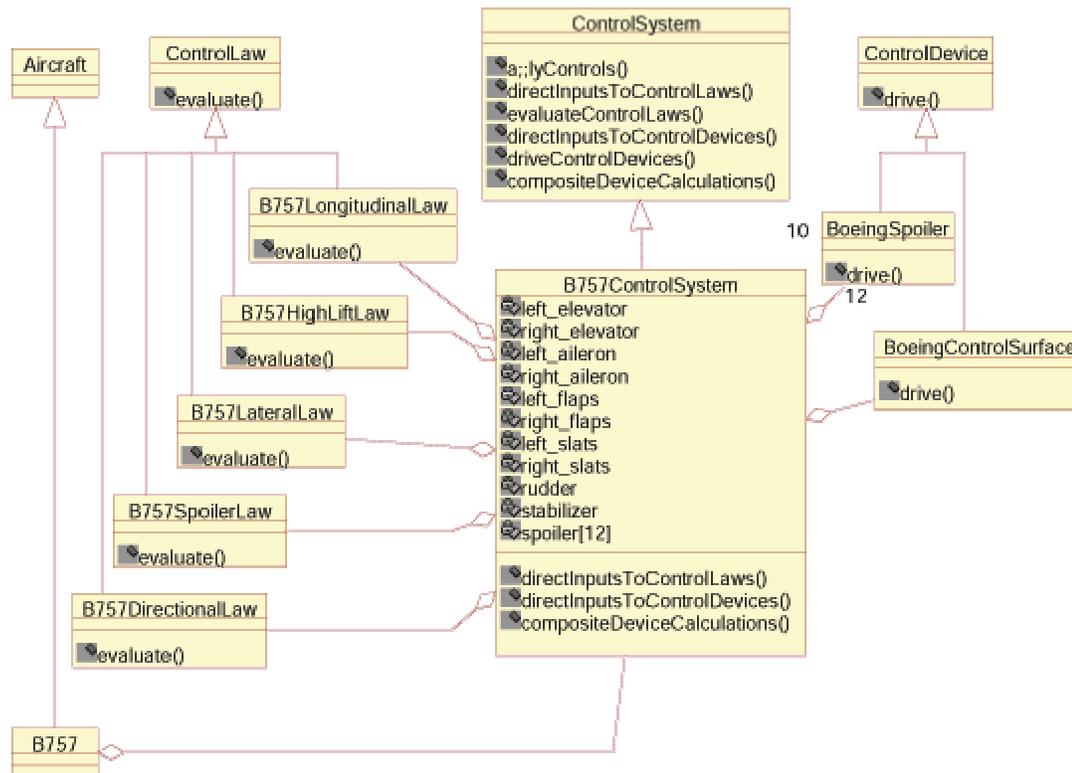


Figure 2 - F16a FCS Class Structure

## Object Interaction

Figure 3 and 4 show the interaction of the objects during closed-loop operation for the Boeing 757 and F16A aircraft simulations. At construction time, the client control system instantiates the number and type of laws and devices it requires and registers a pointer to each to the list of laws contained in *ControlSystem*. The order of registration dictates the order of execution used later for each law and device by the *ControlSystem* class. The client aircraft, as part of its

execution process, tells *ControlSystem* to *applyControls()*. The *applyControls()* method within *ControlSystem* calls *directInputsToControlLaws()*, a pure virtual method defined at the client control system level. The *directInputsToControlLaws()* method in the client control system sets inputs required for each control law that it uses. This input processing usually involves gathering state variables from other parts of the client aircraft and sending them to the passive control law objects.

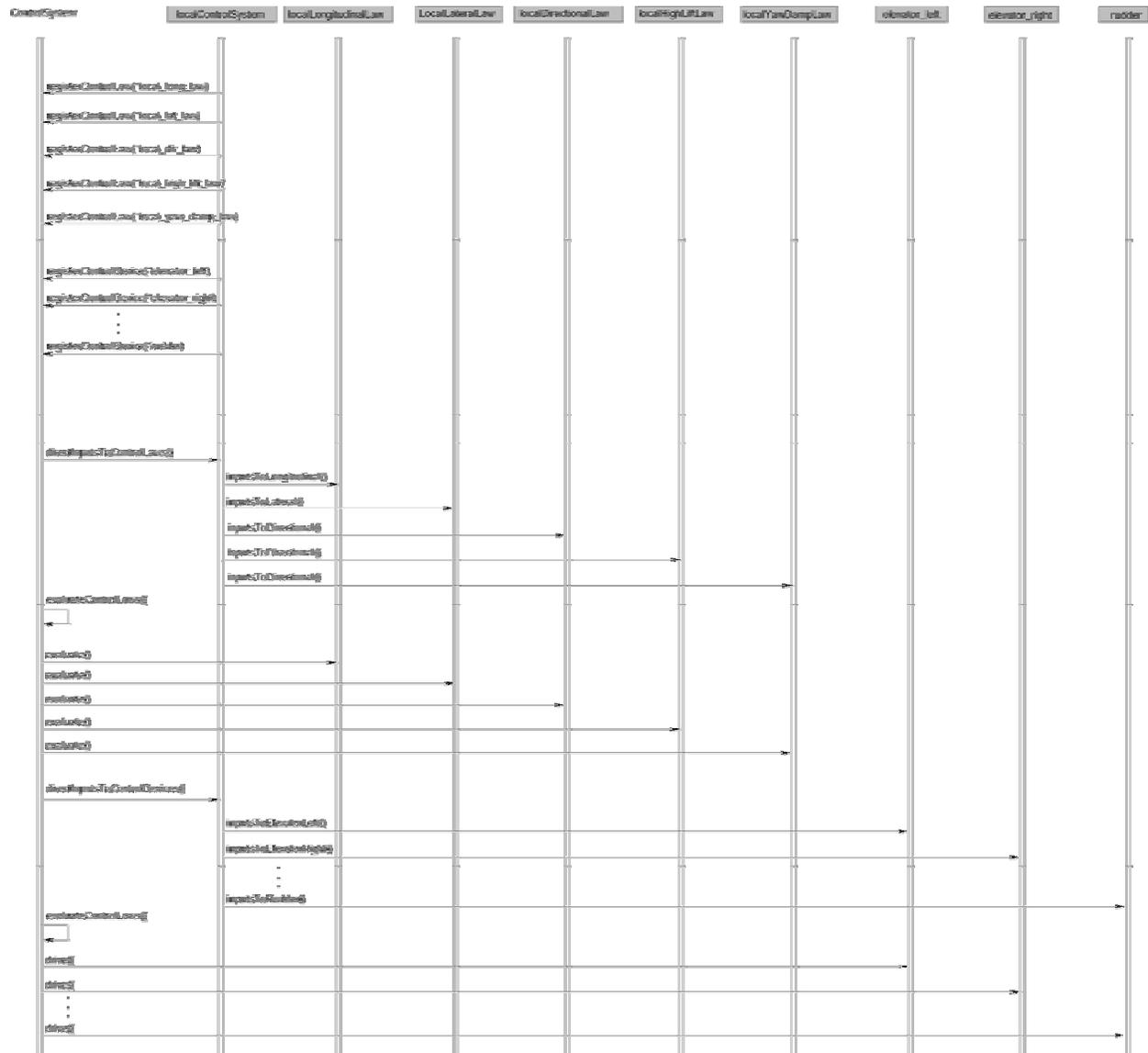


Figure 3 - 757 Object Interaction Diagram

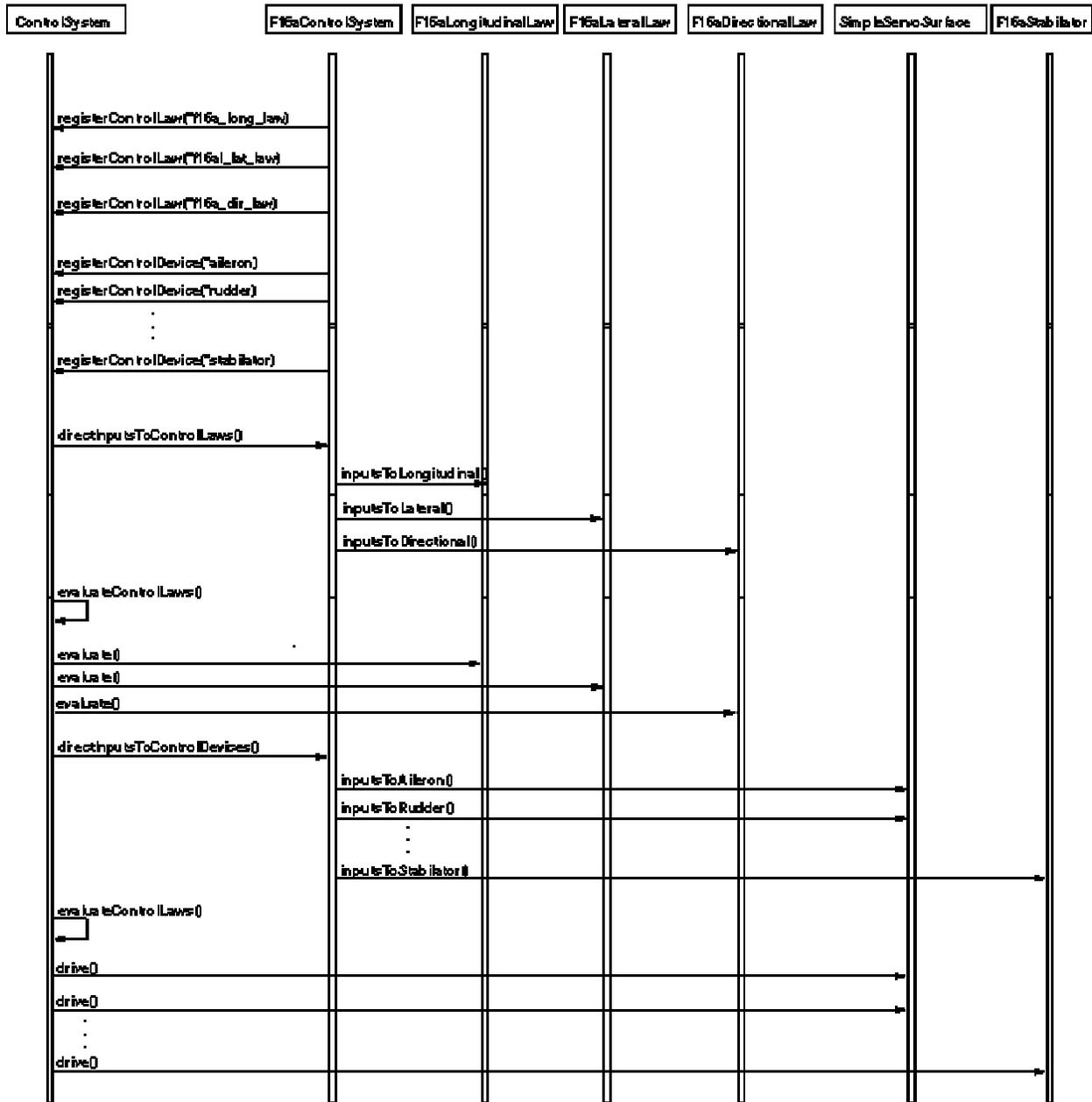


Figure 4 - F16A Object Interaction Diagram

The *applyControls()* method then calls *evaluateControlLaws()* which simply steps through its list of control laws and calls *evaluate()* for each. The *evaluate()* method is defined in the parent *ControlLaw* class as a pure virtual function, and must be redefined by each specific control law that inherits from it. All technical code required to compute the command output is contained in the *evaluate()*

method within the clients control laws or in local scope methods called by it.

Similar processing is then performed by the *ControlSystem* class for device calculations. The *directInputsToControlDevices()* method is defined at the client control system level and sets inputs required for each control device used. This input

processing usually involves gathering commands from the control laws and additional aircraft state variables from other simulation objects and updating the input data within each device object.

The *driveControlDevices()* method in *ControlSystem* is called which steps through the list of registered devices and instructs the *drive()* method to be executed for each one.

Finally, the *compositeDeviceCalculation()* method is called to compute required output combinations for other parts of the simulation.

Note that even though the number and type of laws and surfaces differ for the two aircraft, the execution methodology and the code structure for the classes remains the same.

## Conclusions

The initial design and testing phase for this flight control system architecture (as is usual for sophisticated and reliable object-oriented development), required a substantial effort on the part of the developer. This method was first used for the 757 simulation model. Before this architecture was installed, the 757 FCS code was encumbered with switch statements and branch “if” testing because of the extensive amount of special processing required for that particular project. The code was very fragile and had become almost indecipherable. After the 757 control system was revamped to fit this generic method, the code was highly robust and instantly comprehensible by anyone willing to familiarize themselves with the architecture.

Since its inception, three other aircraft have used the architecture with extensive time saving both in testing and validation of the code and in the design review documentation requirements. Testing and coding for these aircraft was practically limited to the installation of their technical equations and inputs to them. Linear analysis capability was effectively free. The development effort was easily justified in this instance, if not absolutely required, for the 757 model to be maintainable through its projected life span. Smaller control systems that capitalized on the effort would probably not have expended the initial development effort unless a large amount of special processing was expected in that project’s life.

A common method of class structure and execution for a framework, however, is a large motivator. Installation and testing of control systems, previously handled exclusively by experienced simulation developers, is now effectively accomplished by

relatively new developers in the LaSRS++ framework. The conclusion is that the initial testing and development effort is highly recommended for several situations:

- when the complexity of one particular model compromises the code’s readability
- when many aircraft share a common framework
- when the responsibility for code maintenance will be handled by changing personnel over the project’s life span

## Bibliography

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.
- [2] Steve McConnell. Code Complete: A Practical Handbook of Software Construction. Microsoft Press, Redmond, Washington, 1993.
- [3] Terry Quatrani. Visual Modeling With Rational Rose and UML. Addison-Wesley, Reading, MA, 1998.
- [4] Scott Meyers. Effective C++. Addison-Wesley, Reading, Massachusetts, second edition, 1998.
- [5] Grady Booch. Object-Oriented Analysis and Design. Benjamin/Cummings, Redwood City, California, 1994.
- [6] R. Leslie, D. Geyer, K. Cunningham, M. Madden, P. Kenney, P. Glaab. LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft. AIAA-98-4529, Modeling and Simulation Technology Conference, Boston, MA, August 1998.