

# USE OF THE MEDIATOR DESIGN PATTERN IN THE LaSRS++ FRAMEWORK

Kevin Cunningham\*

Unisys Corporation  
NASA Langley Research Center  
MS 169  
Hampton, VA 23681

## Abstract

Software design patterns are effective, efficient, established solutions to common software design problems. The Mediator Design Pattern is a particularly simple design pattern. It is used extensively in the NASA Langley Standard Real-Time Simulation in C++ (LaSRS++) Framework. The Mediator Design Pattern acts as an information "broker" between the components of a system. The use of the Mediator Design Pattern is a simple means by which re-usability, simplicity and testability of flight simulation software can be obtained.

## Introduction

The goals of maintainability, extensibility and reliability are certainly common to all software development efforts. The ability to achieve these goals grows more critical as the scope and intended lifespan of software increase. The use of design patterns and object-oriented analysis, design and programming techniques to meet these goals is well established.<sup>1-4</sup>

The NASA Langley Standard Real-Time Simulation in C++ (LaSRS++) Framework<sup>5</sup> is a large scale collection of classes which are used to build various simulations. The framework software is intended to have a long lifespan, outliving any one research simulation project which it is used to develop. With this in mind, care was taken

to design the software using patterns and object-oriented techniques.

There is much commonality in the way certain software components interact. The relationships between components result in patterns which are repeated over and over regardless of the system being modeled. Establishing efficient and effective mechanisms for these relationships is a problem which requires careful attention. Design patterns are evolved, yet simple solutions to these common problems in software development.<sup>2</sup> By using "tried and true" design patterns, a software developer is spared the time and expense of iterating to an efficient solution to the problem.

The Mediator Design Pattern is a common software design pattern that is used extensively in the LaSRS++ Framework. The Mediator is responsible for passing data between, and managing the behavior of, its aggregate components. It is a simple, yet evolved solution to several common software design problems. The use of the Mediator Design Pattern simplifies component interfaces and eliminates component interdependencies, leading to a more simple design. This results in enhanced re-usability and increased testability. The final product is software with superior reliability, maintainability and extensibility.

---

\*Senior Member, AIAA

Copyright ©1999 by the author. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

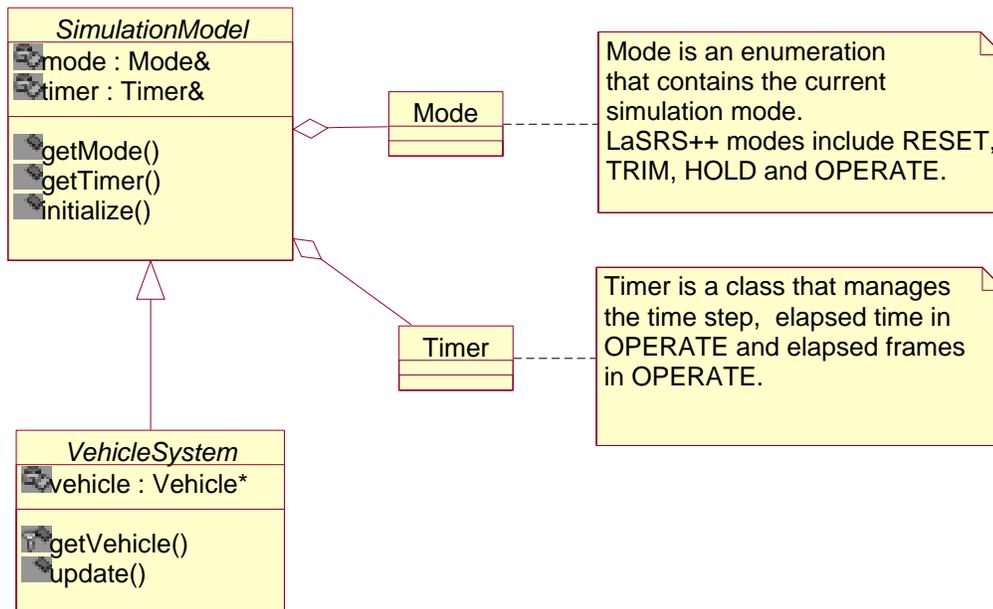


Figure 1: The Vehicle System

### Fundamentals

Figure 1 is a class diagram<sup>6</sup> of the Vehicle System hierarchy. In the C++ language, a *class* is a software developer’s fundamental building block. A single class represents a single concept or physical entity. It contains functions, which define behavior, as well as data. A particular instance of a class is called an *object*. Objects of various class types can be used to create complex software in a straight forward manner.

Two important relationships, *inheritance* and *containment*,<sup>7,8</sup> are illustrated in figure 1. Inheritance is also called an “is a” relationship (an F/A-18 is an airplane). It is the means by which the detail can be built up in a layered fashion. Containment is called a “has a” relationship (an F/A-18 has a flight control system). Containment occurs when one object instantiates (or contains a reference to) another object.

Figure 1 illustrates that the Simulation Model class contains Mode and Timer objects by reference. The

variables which represent the references are `mode` and `timer`. The class also contains functions which a client can call to obtain access to the mode and timer objects. Additionally, the class contains a function defining behavior at initialization. The Vehicle System class inherits from the Simulation Model class. The vehicle pointer in Vehicle System is a pointer to the parent vehicle which contains the Vehicle System object.

A *software framework* is a collection of components which are used to build a variety products. In the case of the LaSRS++ framework, the products are simulations. One of the goals for the LaSRS++ framework is to foster good design at the aircraft model level via design choices implemented at the framework level. The Vehicle System concept is one example of this.

The design of the LaSRS++ framework is such that classes which are Vehicle Systems act as mediators. These mediator classes are intended to manage communication between the vehicle itself and its systems’ components. The mediator is responsible for passing

data into its aggregates, instructing the aggregates to perform their calculations, and extracting any output. The Vehicle System is the framework's design pattern for decoupling a model from the vehicle.

### Framework Systems

The Vehicle System class shown in figure 1 is an *abstract class*. Abstract classes are not designed to support the creation of objects, therefore there can be no instances of an abstract class. They can only be inherited. Abstract classes serve as a portal to system extension, providing the form which other classes are to follow. When inheritance relationships are chained together additional detail is added with each layer. The layer below the Vehicle System class provides common interfaces which in turn will be inherited.

The LaSRS++ framework provides support for a

number of systems:

- Aerodynamic System
- Caution And Warning System
- Control System
- Fuel System
- Hydraulic System
- Landing Gear System
- Navigation System
- Propulsion System
- Weapon System

The class diagram for the some systems in the framework is shown in figure 2.

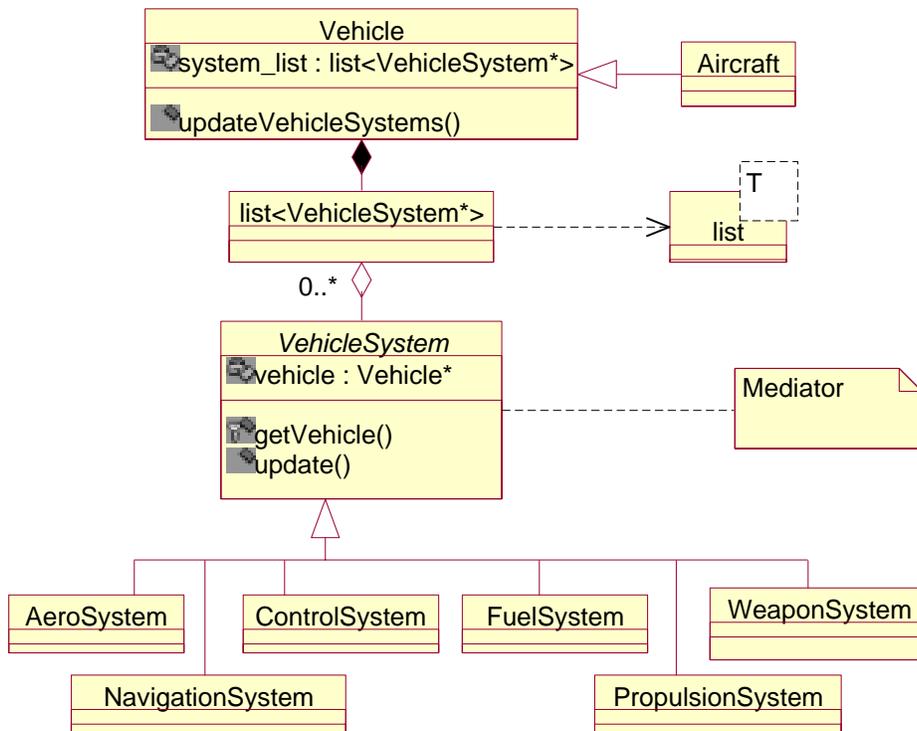


Figure 2: Systems in the Framework

Figure 2 shows the relationship between the Vehicle class and the Vehicle Systems class. A Vehicle has a linked list<sup>9</sup> of Vehicle Systems. There are no restrictions on the number of systems that can go on the list. As a specific aircraft instantiates its systems, the objects need only be added to the list. This registration is a simple matter of a single call to a function which is inherited from the Vehicle object. This makes it possible to update all the systems in a vehicle by simply sending one instruction to update whatever is on the list.

The system specific classes shown in figure 2, which inherit from Vehicle System, establish the common interface that various simulations will use. When a specific vehicle simulation is being developed, the details specific to that simulation model are added to the interface. All systems of that type share the same interface, only the encapsulated details will vary.

### A Closer Look At A System

Figure 3 is a more detailed class diagram. It shows how a mediator class fits into an architecture to decouple one part of a design from another. This means that a class representing a system being modeled does not require direct knowledge of the aircraft of which it is a part. The model is not dependent on the vehicle to perform its function.

Figure 4 is an object interaction diagram. The interactions depicted in that diagram are for a greatly simplified aerodynamic model. Enough detail is provided to illustrate how a typical mediator is used to decouple classes.

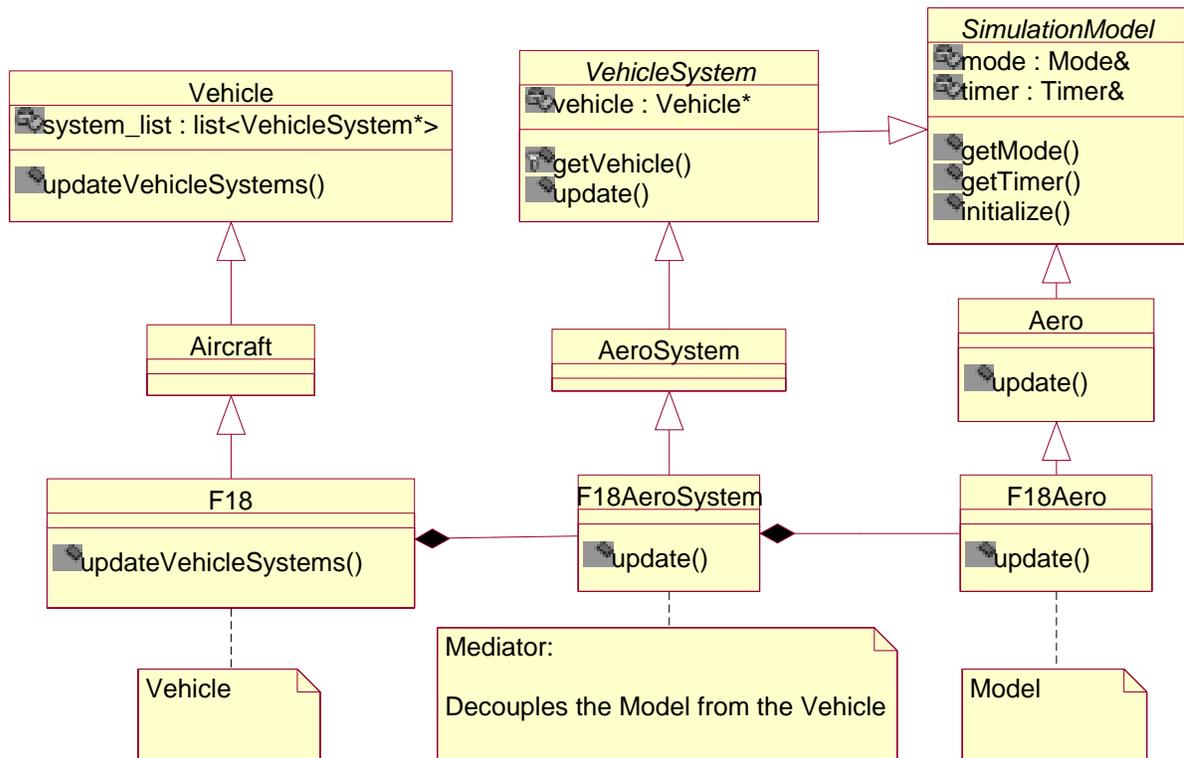


Figure 3: An Aerodynamic System Mediator

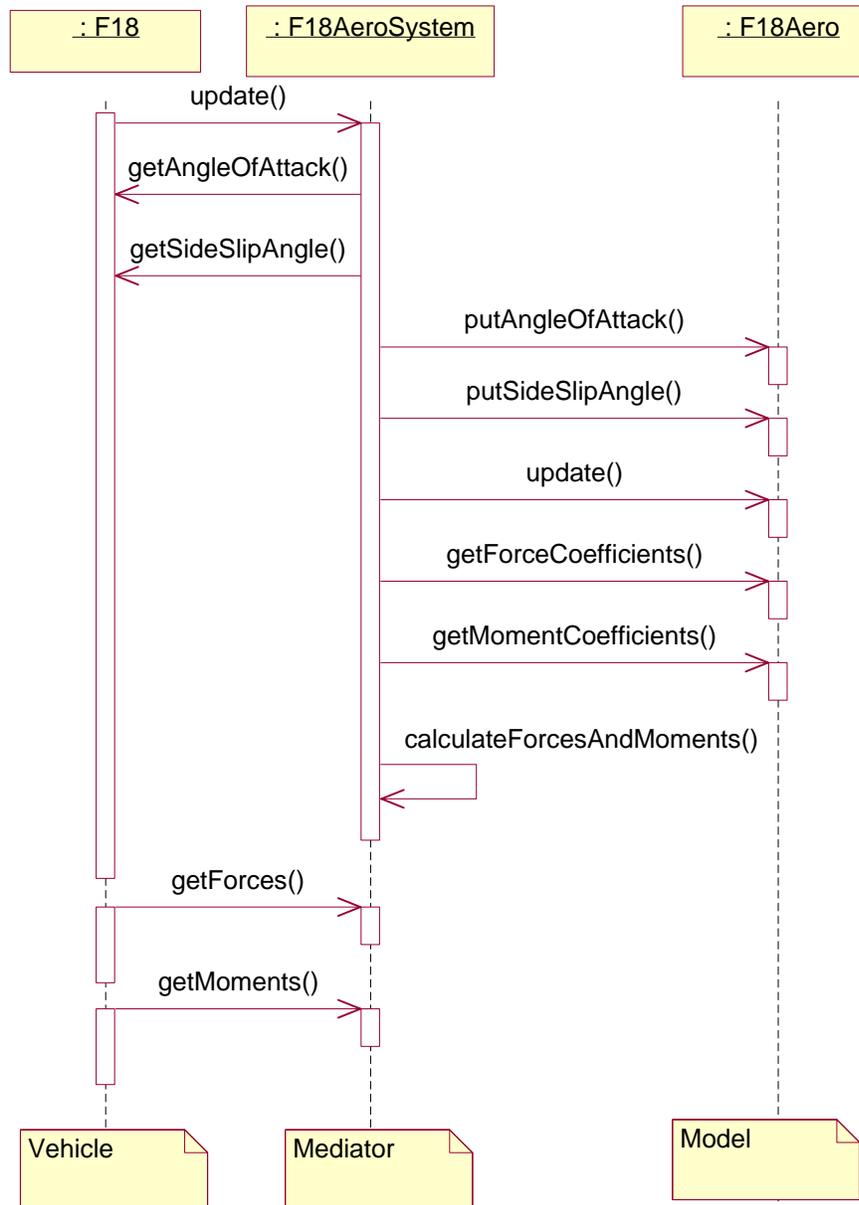


Figure 4: Aerodynamic System Object Interaction Diagram

Figure 4 shows that the interactions begin with the vehicle calling the update function in the mediator. The instruction (or message) for the Aerodynamic System to update itself triggers a series of events. The mediator calls functions in the vehicle to obtain independent data required for table lookups in the aerodynamic model.

After the necessary data has been obtained by the mediator, it is passed into the model. This occurs by calling publicly accessible inlined mutator functions which are defined in the model class. Mutator functions are the mechanism by which an object's internal, private data may be set. Next, the mediator instructs the aerody-

dynamic model to update itself. After the coefficients are calculated and totaled, the results are retrieved when the mediator calls the inlined accessor functions. Finally, the mediator calls a function defined within the mediator class which converts the coefficients into dimensional forces and moments.

The use of the mediator pattern allows the model class to remain blissfully ignorant of the vehicle's details. The mediator now contains the details required to manage the communication between the two objects. Complexity is shifted out of the model and into the mediator. In addition to decoupling components, the overall system is simplified by this design choice.

### Major Benefits

Two major benefits of the decoupling provided by the use of the mediator pattern are increased testability and increased re-usability. Testing is fundamental to ensuring software validity and reliability. Prior to integration testing, it is desirable to test all of a model's components in an isolated environment. The key element to this type of unit testing is the ability to decouple the model from the rest of the simulation. Effort can then be put into developing highly effective test scenarios, rather than conjuring-up unique ways to isolate the test process from the simulation environment.

If "time is money", then the ability to re-use software is gold. A component with many dependencies cannot be re-used. Inter-component dependencies tend to make a large system stiff, in-flexible and difficult to extend. When multi-component interaction and behavior are mediated, the design of a single component readily supports re-use by different models, systems, simulations and facilities.

### Lessons Learned

Figure 5 is the class diagram for the model of an aircraft's flight control law. It is a large and complex model. The figure illustrates the interdependencies that result when components of a flight control system share data through direct interaction. Each class is dependent

on all the other classes from which it needs data. This creates a tightly coupled system. Any new data or behavior added to one class affects all the other classes that depend on it. As a tightly coupled system grows, it tends to take on the characteristics of a monolithic class. This limits re-usability, hampers testability and significantly increases the time required to compile the software. In a worst case scenario, every class would be dependent on every other class. *Don't let this happen to you!*

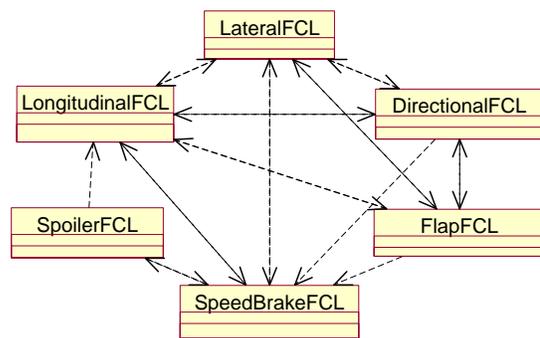


Figure 5: Class Dependencies Around

The solution to this common design problem is the Mediator Design Pattern. The use of the Mediator Design Pattern rids objects of their explicit dependencies. This greatly decouples and simplifies a system. Figure 6 illustrates the impact of a mediator on the system shown in Figure 5.

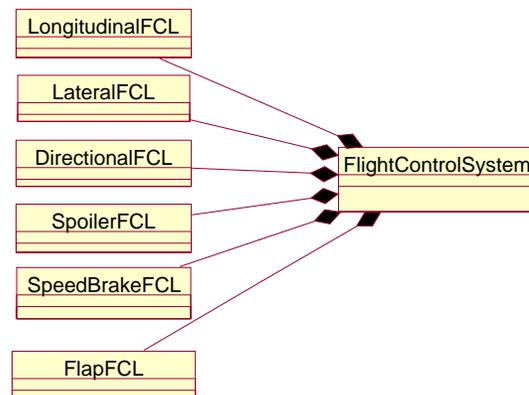


Figure 6: The Mediator Pattern To The Rescue

Figure 6 shows the more simple design. It is the superior design. The aggregate classes no longer depend on each other. In fact, they do not even depend on the mediator class which encapsulates them. This autonomy greatly increases re-usability. Testability is increased in that each class may be tested as a single unit, rather than testing the whole system at once. Finally, compilation times are reduced by the fact that a software change which only affects one class will only require re-compilation of that class, not an entire coupled system.

### Conclusions

The NASA Langley Standard Real-Time Simulation in C++ (LaSRS++) Framework and aircraft simulations developed within the framework have successfully used the Mediator Design Pattern. The Mediator Design Pattern is a simple, effective, efficient, established solution to a number of common software design problems. The use of the Mediator Design Pattern results in software that is more easily tested at the component level, thus promoting a higher quality, more reliable product. The Mediator Pattern provides object decoupling, minimizing component interdependencies. This results in more simple designs and software which is more maintainable and extensible. By minimizing system coupling, the Mediator Design Pattern facilitates software re-use at not only the model and simulation levels, but also at the inter-facility level.

### Bibliography

- [1] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, California, 1994.
- [2] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [3] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Massachusetts, 1996.
- [4] Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.
- [5] Richard A. Leslie, et al. LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft. Paper Number AIAA-98-4529, August, 1998.
- [6] Terry Quatrani. *Visual Modeling With Rational Rose and UML*. Addison Wesley, Reading, Massachusetts, 1998.
- [7] Bruce Eckel. *Thinking in C++*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.
- [8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 1997.
- [9] Atul Saini David R. Musser. *STL Tutorial and Reference Guid*. Addison-Wesley, Reading, Massachusetts, 1996.
- [10] Scott Meyers. *Effective C++*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.
- [11] Patricia Glaab, et al. A Method to Interface Auto-Generated Code into an Object-Oriented Simulation. Paper Number AIAA-98-4531, August, 1998.
- [12] P. Sean Kenney, et al. Using Abstraction to Isolate Hardware in an Object-Oriented Simulation. Paper Number AIAA-98-4533, August, 1998.
- [13] Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, Massachusetts, 1996.