From OOA to C++: The Missing Link

Stephen J. Mellor

Project Technology, Inc.
10940 Bigge Street
San Leandro, CA 94577-1123
(510)567-0255

Abstract

In 1912, one Charles Dawson discovered a skull fragment that was accepted as evidence of the long-sought missing link between apes and humans. In 1953, Piltdown Man as the fossil was known, was shown to be a forgery, a combination of a 600-year-old human skull and a modern orangutan. Sadly, this tale is redolent of today's methods for making the transition from object-oriented analysis to code: two related, but fundamentally disjoint parts are stuck together to make what seems to be a rational progression from one step to another.

As a solution to this problem, this paper will show how to build a detailed and specific mapping from an object-oriented analysis into C++ code. The code will be built using a revolutionary technique called Recursive Design, a part of the Shlaer-Mellor Method. The approach is based on the construction of an architecture that defines the rules for the organization of data, control and algorithm, and then a translation, via the architecture, from the application directly to C++.

One of the many advantages claimed for object-oriented analysis and design is that the transition between the analysis and design is significantly smoother than for other methods. There are two main reasons for this claim.

First, in most object-oriented methods, the notation for analysis and design activities is the same. A single notation helps alleviate the discontinuity between the two activities. On the other hand, in structured analysis and design there are two notations: the data flow diagram for analysis and the structure chart for design. There is then a need to say the same thing twice, once in analysis notation and once in design notation.

The second reason for the claim is that the structure of an object-oriented design and implementation can be the same as for the analysis. The argument runs as follows. The object-oriented analyst abstracts objects on the basis of the real-world subject matter under analysis. The resulting objects tightly connect data and function to act as a unit. As in the real world, each of the objects is distinct from the other objects and loosely coupled. The designer then turns the objects in the analysis directly into the objects in the design. Any change in the real world leads to a corresponding change in the analysis, which, in turn, leads to a corresponding change in the design. The size and cost of making the changes are as low as can be, because the analysis models represent the real world exactly,

as the design models reflect the analysis exactly. Again, this contrasts with the structured, functional, view in which functions are primary, and changes in data structure wreak havoc on all the functions that touch that data.

Design by Elaboration

But, it isn't quite that simple. In the degenerate case, as suggested by the above, the design is exactly the same as the analysis. If this were really true there is no work for the designer to do! In a real project, we are never that lucky. It is typical to add some detail to the analysis and, perhaps, to reorganize the models to improve efficiency. Most object-oriented methods define the design process as a process of refinement, or *elaboration*, of the analysis models to yield the design. This step can be very large: perhaps twice the size of the analysis.

Now there are two ways to interpret this understanding of the design process. One possibility is that the job of making a design involves real work, and perhaps the transition between analysis and design is not as smooth as suggested. A second possibility is that the analyst must take the implementation environment into account during analysis, and abstract objects on the basis of data structures selected for efficient access, or on the basis of threads for efficient execution, or on a preconceived aggregation of elements or primitives into objects. This second option implies that the two difficult jobs of analysis and design are done together, but on two different models at different levels of detail.

Design by Translation

In contrast, the Shlaer-Mellor Method employs an approach to object-oriented analysis and design that separates analysis from design on the basis of subject matter, and not on level of detail. The application-independent subject matter of design is called the software architecture domain. The software architecture domain proclaims and enforces the policies regarding data, control and algorithm in the system as a whole, and, when built, it acts as a design environment into which the application is embedded using a *translation* mechanism.

The software architecture domain provides, *inter alia*, an execution engine for the OOA formalism, data organization and access facilities, and tasking protocols, both inter- and intra-processor, as required. These *mechanisms* stand alone, though they are often driven by data from the application. In addition, the software architecture domain must define translation rules for the use of these mechanisms. These translation rules are realized as *archetypes* that act as a kind of template on which replacements are made. For example, we may require that every object in the analysis become a class in the implementation. This may be written `class <object> { ... }`. The intent here is that `<object>` will be replaced by the name of an object from the application analysis to yield a C++ code fragment.

There are many possible architectures, and for a given application some subset of these architectures may be appropriate. For the purposes of this presentation, we shall describe at a high level, a simple architecture that has a very direct correspondence to the formalism of OOA. The architecture can be characterized as a *single task, asynchronous, event-based, object-oriented, direct instance data implementation, internally-managed class-based linked list with passive iterators* architecture. Put into English, the architecture will yield a single task; it will execute threads asynchronously; it will treat events as the unit of control; it will make use of encapsulation, inheritance, and polymorphism; the data for each instance will be stored directly with no modification or optimization; it will employ linked lists to track multiple instances of a class managing each list within each class; and it will use passive iterators to execute operations on the several instances of a class. For comparison, the architecture described in *Object Lifecycles* [1], Chapter 9, is a single task, synchronous, event-based, object-oriented, direct instance data implementation, internally-managed class-based linked list with passive iterators architecture: just one character different. The first architecture is asynchronous: it executes each event one at a time, independent of the thread in which the event occurs. The second architecture is synchronous: it executes a single thread to completion.

We shall describe here only event handling mechanisms and data implementation and access. While this is incomplete, it is sufficient to illustrate the concepts required to support a real link between OOA and C++.

Event Handling Mechanisms

The mechanisms of the example architecture include a `Finite State Model` class, a `Transition` class, an `Event` class, an `Active Instance` class and some others not required for this example. The `Finite State Model` and the `Transition` classes take on the responsibility of maintaining, and executing at run time, the state transition tables that may be associated with each object in the application analysis. The state transition tables are treated as data on which these classes operate. The `Event` class is used to create, store and delete event instances. Each time an action generates an event, it calls the constructor for the `Event` class and returns. The main loop of the task then accesses the next event (perhaps the same one that has just been created, perhaps another) and then sends the event to the destination class. This is achieved by a class `Active Instance`, which is a base class of all the application classes that have state models. It defines a virtual function, `Do Event`, that takes on the job of taking an event, finding the current state of the object instance, and causing a transition to occur using `Finite State Model` and `Transition`. The main loop calls `Do Event` that is provided by each application class inheriting from `Active Instance`.

Data Access

The data organization scheme in this architecture is defined to be direct, so there is one private data member for each attribute defined on the object information model. Data access to instances will be implemented as instance-based public member functions, up to

two for each private data member of the object. There may be one write accessor that takes a value and stores it, and there may be a read accessor that returns the value of the specified attribute. Each of these accessor functions is required only if there is at least one other object that requires it.

Some classes require operations where the handle of the instance is not known, for example, Find a Dog weighing more than 30 pounds, or Find all Collies. By contrast with the instance-based queries, we call these operations *class-based queries.* This architecture uses linked lists to link together all the instances of a class to enable these class-based operations. In addition, the `head` pointer to the first instance is stored as class data within the class, and all other list management is handled within the class. The same scheme is used for all classes that have class-based queries acting on them.

A *passive iterator* is a single class-based function that finds an instance, or a set of instances, according to some specified criteria. The two example class-based operations above, if implemented as functions in the `Dog` class, are passive iterators. The client of the iterator simply calls the function; the iterator contains a loop that iterates through all the instances. (A passive iterator contrasts with an *active iterator*, which is a set of functions, say `reset` and `next`, that iterate through each instance of the set one by one. The control of the iteration is in the client, and the iterator maintains an implied cursor inside the server class.) All class-based queries found in the OOA of the application can be implemented using the passive iterator. There are some important details to manage here, such as how to return a set of instances, and how to handle nested queries such as Find all the Dogs weighing more than 30 pounds whose owners live in Berkeley, but schemes can be found to manage these in a regular manner.

All classes in the system can use the same approach to data access. Each application object that has only instance-based queries can be implemented directly with no link to other instances of the class. Each application object that has class-based queries must link the instances together. Each class-based operation is implemented as a passive iterator. The underlying logic required to build each passive iterator is the same in all cases.

Combinations

So far, we have shown that there is a way to send and receive events, that data can be implemented directly, and that it can be accessed. Real implementations combine these concepts in several ways.

In this architecture, a *passive object* is an object that does not do anything of its own volition, but has data that can be accessed by other objects. The only logic in a passive object is data access code. A passive object may have instance-based operations, or it may have class-based operations, or both. There is a pattern, an archetype, that can be laid out for each passive object.

An *active object* is an object that has a state machine associated with each instance. The mechanisms for the state machine traversal have already been tested. There is a pattern, an archetype, that can be laid out for each active object.

An *assigner object* is an object that has a single state machine associated with the set of instances. (This type of object is used in OOA to manage contention.) Each such assigner object inherits from `Active Instance` to provide for the ability to traverse a state machine. Some objects may have two state models, one for a single instance, and one for the set of all instances. The same class inherits from `Active Instance` twice. There is a pattern, an archetype, that can be laid out for each assigner object.

The C++ Code

The mechanisms are written in C++.

The archetypes for passive, active, and assigner objects are written in C++ extended to allow for replacement of elements from the application so that `class <object> { .. }` will yield `class oven{ ..}`, `class light { .. }`, `class tube { .. }` etc., assuming that `oven`, `light` and `tube` are objects in the analysis. The intent here is that `<object>` will be replaced by the name of an object from the application analysis to generate a C++ code fragment.

The combination of the mechanisms and the archetypes and replacements from the application produce the final system.

Where's the Design?

The example architecture described above is a particularly direct translation of the analysis into the design. There is sufficient detail in the analysis and the architecture to yield a complete system.

However, the architecture need not be direct at all. We have defined architectures that vary all of the adjectives used to describe the example architecture, including periodic architectures used in certain kinds of real-time systems, multitasking architectures, and thread-based architectures. The task of the designer is to define the architecture and the translation from the analysis to the implementation via that architecture. This separation of concerns leaves analysis and design as distinct concepts, solving different problems, but with a clear, defined, link between the two activities.

[1] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, 1992 Prentice Hall, Englewood Cliffs, NJ.