Object-Oriented Development

GRADY BOOCH, MEMBER, IEEE

Abstract—**Object-oriented development is a partial-lifecycle software development method in which the decomposition of a system is based upon the concept of an object. This method is fundamentally different from traditional functional approaches to design and serves to help manage the complexity of massive software-intensive systems. The paper examines the process of object-oriented development as well as the influences upon this approach from advances in abstraction mechanisms, programming languages, and hardware. The concept of an object is central to object-oriented development and so the properties of an object are discussed in detail. The paper concludes with an examination of the mapping of object-oriented techniques to Ada® using a design case study**

Index Terms—**Abstract data type, Ada, object, object-oriented development, software development method.**

I. Introduction

RENTSCH predicts that "object-oriented programming will be in the 1980's what structured programming was in the 1970's" [1]. Simply stated, *object-oriented development* is an approach to software design in which the decomposition of a system is based upon the concept of an object. An *object* is an entity whose behavior is characterized by the actions that it suffers and that it requires of other objects.

Object-oriented development is fundamentally different from traditional functional methods, for which the primary criteria for decomposition is that each module in the system represents a major step in the overall process. The differences between these approaches becomes clear if we consider the class of languages for which they are best suited.

The proper use of languages like Ada and Smalltalk requires a different approach to design than the approach one typically takes with languages such as Fortran, Cobol, C, and even Pascal. Well-structured systems developed with these older languages tend to consist of collections of subprograms (or their equivalent), mainly because that is structurally the only major building block available. Thus, these languages are best suited to functional decomposition techniques, which concentrate upon the algorithmic abstractions. But as Guttag observes, "unfortunately, the nature of the abstractions that may be conveniently achieved through the use of subroutines is limited. Subroutines, while well suited to the description of abstract events (operations), are not particularly well suited to the description of abstract objects. This is a serious drawback" [2].

Languages like Ada also provide the subprogram as an elementary building block. However, Ada additionally offers the package and task as major structural elements. The package gives us a facility for extending the language by creating new objects and classes of objects, and the task gives us a means to naturally express concurrent objects and activities. We can further extend the expressive power of both subprograms and packages by making them generic. Together, these facilities help us to better build abstractions of the problem space by permitting a more balanced treatment between the nouns (objects) and verbs (operations) that exist in our model of reality.

Of course, one can certainly develop Ada systems with the same methods as for these more traditional languages, but that approach neither exploits the power of Ada nor helps to manage the complexity of the problem space.

In general, functional development methods suffer from several fundamental limitations. Such methods

- do not effectively address data abstraction and information hiding;
- are generally inadequate for problem domains with natural concurrency;
- are often not responsive to changes in the problem space.

With an object-oriented approach, we strive to mitigate these problems.

Before we get too detailed, let us consider alternate designs for a simple real-time system using functional and object-oriented techniques.

A cruise-control system exists to maintain the speed of a car, even over varying terrain [3]. In Fig. 1 we see the block diagram of the hardware for such a system. There are several inputs:

• System on/off	If on, denotes that the cruise-control system should maintain the car speed.
• Engine on/off	If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
• Pulses from wheel	A pulse is sent for every revolution of the wheel.
• Accelerator	Indication of how far the accelerator has been pressed.

Manuscript received June 17, 1985.
The author is with Rational, Mountain View, CA 94043.
IEEE Log Number 8405735.
®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

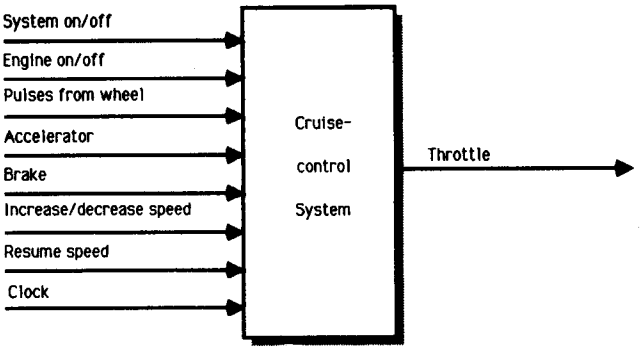


Fig. 1. Cruise-control system hardware block diagram.

- Brake — On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
- Increase/Decrease Speed — Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
- Resume — Resume the last maintained speed; only applicable if the cruise-control system is on.
- Clock — Timing pulse every millisecond.

There is one output from the system:

- Throttle — Digital value for the engine throttle setting.

How might we approach the design of the software for the cruise control system? Using either functional or object-oriented approaches, we might start by creating a data flow diagram of the system, to capture our model of the problem space. In Fig. 2, we have provided such a diagram, using the notation by Gane and Sarson [4].

With a functional method, we would continue our design by creating a structure chart. In Fig. 3, we have used the techniques of Yourdon and Constantine [5] to decompose the system into modules that denote the major functions in the overall process.

With an object-oriented approach, we proceed in an entirely different manner. Rather than factoring our system into modules that denote operations, we instead structure our system around the objects that exist in our model of reality. By extracting the objects from the data flow diagram, we generate the structure seen in Fig. 4. We will more fully explain the process and the meaning of the symbols used in the figure later. For the moment, simply recognize that the amorphous blobs denote objects and the directed lines denote dependencies among the objects.

Immediately, we can see that the object-oriented decomposition closely matches our model of reality. On the other hand, the functional decomposition is only achieved through a transformation of the problem space. This latter design is heavily influenced by the nature of the subpro-

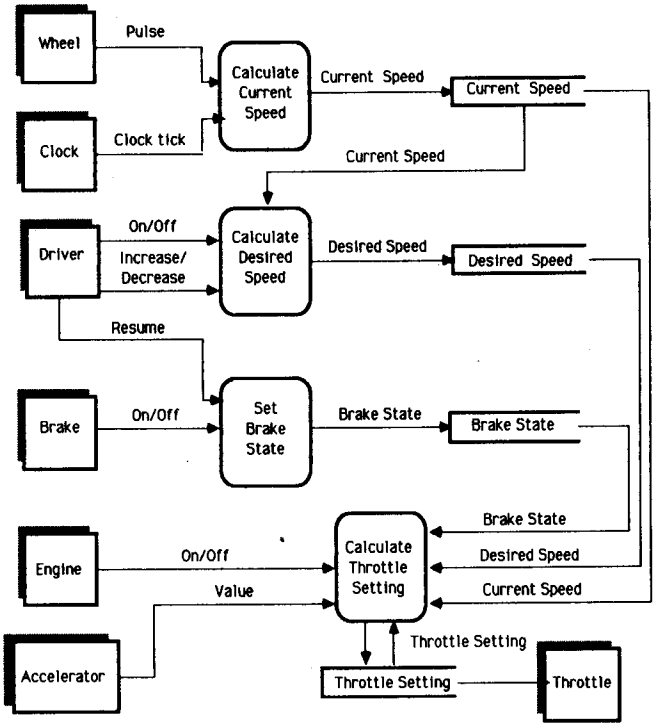


Fig. 2. Cruise-control system data flow diagram.

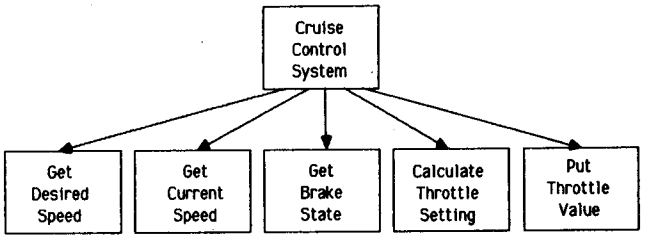


Fig. 3. Functional decomposition.

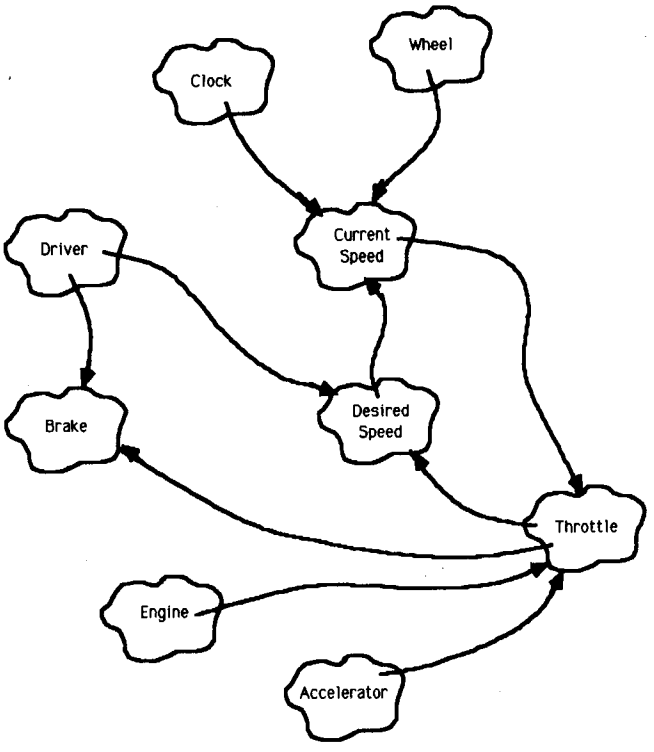


Fig. 4. Object-oriented decomposition.

gram and so emphasizes only the algorithmic abstractions that exist. Hence, we can conclude that a functional decomposition is imperative in nature: it concentrates upon the major actions of a system and is silent on the issue of the agents that perform or suffer these actions.

The advantages of the object-oriented decomposition are also evident when we consider the effect of change (and change will happen to any useful piece of software). One side-effect of the functional decomposition is that all interesting data end up being global to the entire system, so that any change in representation tends to affect all subordinate modules. Alternately, in the object-oriented approach, the effect of changing the representation of an object tends to be much more localized. For example, suppose that we originally chose to represent car speed as an integer value denoting the number of wheel revolutions per some time unit (which would not be an unreasonable design decision). Suppose that we are now told to add a digital display that indicates the current speed in miles per hour. In the functional decomposition, we might be forced to modify every part of the system that deals with the representation of speed, as well as to add another major module at the highest level of the system to manage the display. However, in the object-oriented decomposition, such a change directly affects only two objects (current speed and desired speed) and would require the addition of one more object (the display) that directly parallels our modification of reality.

Regarding an even more fundamental change, suppose that we chose to implement our cruise-control system using two microcomputers, one for managing the current and desired speeds and the second to manage the throttle. To map the functional decomposition to this target architecture requires that we split the system design at the highest level. For the object-oriented approach, we need make no modification at this level of the design to take advantage of the physical concurrency.

II. Object-Oriented Development

Let us examine the process of object-oriented development more closely. Since we are dealing with a philosophy of design, we should first recognize the fundamental criteria for decomposing a system using object-oriented techniques:

Each module in the system denotes an object or class of objects from the problem space.

Abstraction and information hiding form the foundation of all object-oriented development [6], [7]. As Shaw reports, "an abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others" [8]. Information hiding, as first promoted by Parnas, goes on to suggest that we should decompose systems based upon the principle of hiding design decisions about our abstractions [9].

Abstraction and information hiding are actually quite natural activities. We employ abstraction daily and tend to develop models of reality by identifying the objects and operations that exist at each level of interaction. Thus, when driving a car, we consider the accelerator, gauges, steering wheel, and brake (among other objects) as well as the operations we can perform upon them and the effect of those operations. When repairing an automobile engine, we consider objects at a lower level of abstraction, such as the fuel pump, carburetor, and distributor.

Similarly, a program that implements a model of reality (as all of them should) may be viewed as a set of objects that interact with one another. We will study the precise nature of objects in the following section, but next, let us examine how object-oriented development proceeds. The major steps in this method are as follows:

- Identify the objects and their attributes.
- Identify the operations suffered by and required of each object.
- Establish the visibility of each object in relation to other objects.
- Establish the interface of each object.
- Implement each object.

These steps are evolved from an approach first proposed by Abbott [10].

The first step, *identify the objects and their attributes*, involves the recognition of the major actors, agents, and servers in the problem space plus their role in our model of reality. In the cruise-control system, we identified concrete objects such as the accelerator, throttle, and engine and abstract objects such as speed. Typically, the objects we identify in this step derive from the nouns we use in describing the problem space. We may also find that there are several objects of interest that are similar. In such a situation, we should establish a class of objects of which there are many instances. For example, in a multiple-window user interface, we may identify distinct windows (such as a help window, message window, and command window) that share similar characteristics; each object may be considered an instance of some window class.

The next step, *identify the operations suffered by and required of each object*, serves to characterize the behavior of each object or class of objects. Here, we establish the static semantics of the object by determining the operations that may be meaningfully performed on the object or by the object. It is also at this time that we establish the dynamic behavior of each object by identifying the constraints upon time or space that must be observed. For example, we might specify that there is a time ordering of operations that must be followed. In the case of the multiple-window system, we should permit the operations of open, close, move, and size upon a window object and require that the window be open before any other operation be performed. Similarly, we may constrain the maximum and minimum size of a particular window.

Clearly, the operations suffered by an object define the activity of an object when acted upon by other objects. Why must we also concern ourselves with the operations required of an object? The answer is that identifying such operations lets us decouple objects from one another. For example, in the multiple-window system we might assume the existence of some terminal object and require the op-

erations of Move_Cursor and Put. As we will see later, languages such as Ada provide a generic mechanism that can express these requirements. The result is that we can derive objects that are inherently reusable because they are not dependent upon any specific objects, but rather depend only upon other classes of objects.

In the third step, to *establish visibility of each object in relation to other objects*, we identify the static dependencies among objects and classes of objects (in other words, what objects see and are seen by a given object). The purpose of this step is to capture the topology of objects from our model of reality.

Next, to *establish the interface of each object*, we produce a module specification, using some suitable notation (in our case, Ada). This captures the static semantics of each object or class of objects that we established in a previous step. This specification also serves as a contract between the clients of an object and the object itself. Put another way, the interface forms the boundary between the outside view and the inside view of an object.

The fifth and final step, *implement each object*, involves choosing a suitable representation for each object or class of objects and implementing the interface from the previous step. This may involve either decomposition or composition. Occasionally an object will be found to consist of several subordinate objects and in this case we repeat our method to further decompose the object. More often, an object will be implemented by composition; the object is implemented by building on top of existing lower-level objects or classes of objects. As a system is prototyped, the developer may chose to defer the implementation of all objects until some later time and just rely upon the specification of the objects (with suitably stubbed implementations) to experiment with the architecture and behavior of a system. Similarly, the developer may chose to try several alternate representations over the life of the object, in order to experiment with the behavior of various implementations.

We must point out that object-oriented development is a partial-lifecycle method; it focuses upon the design and implementation stages of software development. As Abbott observes, "although the steps we follow in formalizing the strategy may appear mechanical, it is not an automatic procedure... [it] requires a great deal of real world knowledge and intuitive understanding of the problem" [11]. It is therefore necessary to couple object-oriented development with appropriate requirements and analysis methods in order to help create our model of reality. We have found Jackson Structured Development (JSD) to be a promising match [12] and recently, there has been interest in maping requirements analysis techniques such as SREM to object-oriented development [13].

Systems designed in an object-oriented manner tend to exhibit characteristics quite different than those designed with more tranditional functional approaches. As Fig. 5 illustrates, large object-oriented systems tend to be built in layers of abstraction, where each layer denotes a collection of objects and classes of objects with restricted visibility to other layers; we call such a collection of objects a *subsytem*. Furthermore, the components that form a subsystem tend to be structurally flat (like we saw in Fig. 4), rather than being stricly hierarchical and deeply nested.

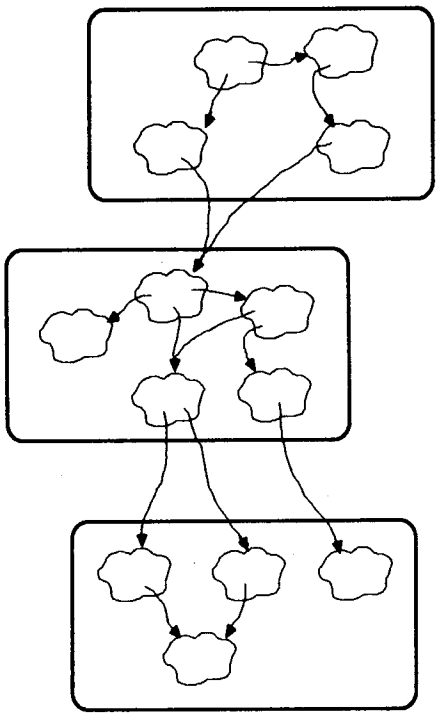


Fig. 5. Canonical structure of large object-oriented systems.

It is also the case that the global flow of control in an object-oriented system is quite different from that of a functionally decomposed system. In the latter case, there tends to be a single thread of control that follows the hierarchical lines of decomposition. In the case of an object-oriented system, because objects may be independent and autonomous, we typically cannot identify a central thread of control. Rather, there may be many threads active simultaneously throughout a system. This model is actually not a bad one, for it more often reflects our abstraction of reality. We should add that the subprogram call profile of an object-oriented system typically exhibits deeply nested calls; the implementation of an object operation most often involves invoking operations upon other objects.

There are many benefits to be derived from an object-oriented approach. As Buzzard notes, "there are two major goals in developing object-based software. The first is to reduce the total life-cycle software cost by increasing programmer productivity and reducing maintenance costs. The second goal is to implement software systems that resist both accidental and malicious corruption attempts" [14]. Giving empirical evidence that supports these points, a study by Boehm-Davis notes that "the completeness, complexity, and design time data would seem to suggest that there is an advantage to generating program solutions using...object-oriented methods" [15]. Regarding the maintainability of object-oriented systems, Meyer observers that "apart from its elegance, such modular, object-

oriented programming yields software products on which modifications and extensions are much easier to perform than with programs structured in a more conventional, procedure-oriented fashion" [16]. In general, understandability and maintainability are enhanced due to the fact that objects and their related operations are localized.

Perhaps the most important benefit of developing systems using object-oriented techniques is that this approach gives us a mechanism to formalize our model of reality. As Borgida notes, "the chief advantage of object-oriented frameworks is that they make possible a direct and natural correspondence between the world and its model" [17]. This even applies to problems containing natural concurrency, for as the Boehm-Davis study reports, "the object-oriented method seemed to produce better solutions for [a problem] which involved real-time processing" [18].

III. The Properties of an Object

The notion of an object plays the central role in object-oriented systems, but actually, the concept is not a new one. Indeed, as MacLenna reports, "programming is object-oriented mathematics" [19]. Lately, we have observed a confluence of object-oriented work from many elements of computer science. Levy suggests that the following events have influenced object-oriented development [20]:

- advances in computer architecture, including capability systems and hardware support for operating systems concepts;
- advances in programming languages, as demonstrated in Simula, Pascal, Smalltalk, CLU, and Ada;
- advances in programming method, including modularization and information hiding and monitors.

We would add to this list the work on abstraction mechanisms by various researchers.

Perhaps the first person to formally identify the importance of composing systems in levels of abstraction was Dijkstra [21]. Parnas later introduced the concept of information hiding [9] which, as we will discuss later, is central to the nature of an object. In the 1970's, a number of researchers, most notably Liskov, Guttag, and Shaw, pioneered the development of abstract data type mechanisms [22]–[24]. The late 1970's and early 1980's also saw the application of a number of software development methods (such as JSD) that were declarative rather than imperative in nature.

The greatest influence upon object-oriented development derives from a small number of programming languages. SIMULA 67 first introduced the class as a language mechanism for encapsulating data, but, as Rentsch reports, "the Smalltalk programming system carried the object-oriented paradigm to a smoother model." Indeed, "the explicit awareness of the idea, including the term object-oriented, came from the Smalltalk effort" [1]. Other object-oriented languages such as Ada and Clascal followed the more tranditional path of SIMULA, but in the early 1980's we also saw a number of languages merge the concepts of Lisp and Smalltalk; thus evolved languages such as Flavors and LOOPS. It is also clear that Lisp alone may be effectively used to apply object-oriented techniques [25]. More recently, there has been work to add Smalltalk constructs to C, resulting in a language named Objective-C [26]. Languages such as Smalltalk have collectively been called actor languages, since they emphasize the role of entities as actors, agents, and servers in the structure of the real world [27].

Interestingly, the concept of an object has precedence in hardware. Work with tagged architectures and capability-based systems has led to a number of implementations that we can classify as object-oriented. For example, Myers reports on two object-oriented architectures, SWARD and the Intel 432 [28]. The IBM System 38 is also regarded as an object-oriented architecture [29].

Every source we have introduced presents a slightly different view of object-oriented systems, but from this background we can extract the common properties of the concept. Thus, we may define an object as an entity that:

- has state;
- is characterized by the actions that it suffers and that it requires of other objects;
- is an instance of some (possibly anonymous) class;
- is denoted by a name;
- has restricted visibility of and by other objects;
- may be viewed either by its specification or by its implementation.

The first and second points are the most important: an object is something that exists in time and space and may be affected by the activity of other objects. The state of an object denotes its value plus the objects denoted by this value. For example, thinking back to the multiple-window system we discussed in the first section, the state of a window might include its size as well as the image displayed in the window (which is also an object). Because of the existence of state, objects are not input/output mappings as are procedures or functions. For this reason, we distinguish objects from mere processes, which are input/output mappings.

From Smalltalk, we get the notion of a method, which denotes the response by an object to a message from another object. The activity of one method may pass messages that invoke the methods of other objects. Abstract data types deal with operations is a related way. Liskov suggests that such operations be divided "into two groups: those which do not cause a state change but allow some processes. Whereas a aspect of the state to be observed...and those which cause a change of state" [30]. In practice, we have encountered one other useful class of operations, the iterator, which permits us to visit all subcomponents of an object. The concept of an iterator was formalized in the language Alphard [31]. For example, given an instance of a terminal screen, we may wish to visit all the windows visible on the screen.

Together, we may classify these operations as follows:

- *Constructor:* An operation that alters the state of an object.

- *Selector:* An operation that evaluates the current object state.
- *Iterator:* An operation that permits all parts of an object to be visited.

To enhance the reusability of an object or class of objects, these operations should be primitive. A primitive operation is one that may be implemented efficiently only if it has access to the underlying representation of the object. In this sense, the specification of an object or class of objects should define "the object, the whole object, and nothing but the object."

We may classify an object as an actor, agent, or server, depending upon how it relates to surrounding objects. An actor object is one that suffers no operations but only operates upon other objects. At the other extreme, a server is one that only suffers operations but may not operate upon other objects. An agent is an object that serves to perform some operation on the behalf of another object and in turn may operate upon another object.

Another important characteristic of objects is that each object is a unique instance of some class. Put another way, a class denotes a set of similar but unique objects. A class serves to factor the common properties of a set of objects and specify the behavior of all instances. For example, we may have a class named Window from which we create several instances, or objects. It is important to distinguish between an object and its class: operations are defined for the class, but operations only have an effect upon the object.

Of course, and this gets a little complicated, one can treat a class as an object (forming a *metaclass*), with operations such as creating an instance of the class. This strange loop in the definition is not only academically interesting, but also permits some very elegant programs.

The term *class* comes from SIMULA 67 and Smalltalk; in other languages, we speak of the *type* of an object. Also from Smalltalk, we get the concept of inheritance, which permits a hierarchy of classes. In this sense, all objects are an instance of a class, which is a subIcass of another class (and so on). For example, given an object, its class may be Text_Window, which is in turn a subclass of the more general class Window. An object is said to inherit the methods of this chain of classes. Thus, all objects of the class Text_Window have the same operations as defined by the class Window (and we may also add operations, modify existing operations, and hide operations from the superclass).

Now, and this is an area of much emotional debate, we suggest that inheritance is an important, but not necessary, concept. On a continuum of "object-orientedness," development without inheritance still constitutes object-oriented development. On the other hand, object-oriented development is more than just programming with abstract data types, although abstract data types certainly serve as an important influence; indeed, we can characterize the behavior of most objects using the mechanisms of abstract data types. Whereas development with abstract data types tends to deal with passive objects (that is, *agents* and *servers*), object-oriented development also concerns itself with objects that act without stimulus from other objects (we call such objects *actors*). Another difference between programming with abstract data types and object-oriented development is that, in both cases, we concern ourselves with the operations suffered by an object, but in the latter case, we also concern ourselves with the operations that an object requires of other objects. As we have mentioned, the purpose of this view is to decouple the dependencies of objects, especially when coupled with a language mechanism such as Ada generic units.

Another way to view the relationship between object-oriented development and programming with abstract data types is that object-oriented development builds on the concepts of the latter, but also serves as a method that exposes the interesting objects and classes of objects from our abstraction of reality.

In some cases, the class of an object may be anonymous. Here, the object does have a class but its class is not visible. The implication is that there may be only one object of the class (since there is no class name from which instances may be declared). Practically, we implement such objects as abstract state machines instead of instances of a class.

Another important consideration of any object-oriented system is the treatment of names. The rule is simple: objects are unique instances of a class, and names only serve to denote objects. As Liskov observes, "variables are just the names used in a program to refer to objects" [32]. Thus, an object may be denoted by one name (the typical case) or by several names. In the latter situation, we have an alias such that operation upon an object through one name has the side-effect of altering the object denoted by all the aliases. For example, we may have several variables in our window system that denote the same window object; operating upon an object through one name (such as destroying the window) has the effect of altering the object denoted by all other names. This one object/many name paradigm is a natural consequence of the notion of an object, but depending upon the manner of support offered by the underlying language, is the source of many logical errors. The key concept to remember is that supplying a name to an constructor does not necessarily alter the value of the name, but instead, alters the object denoted by the name.

The names of objects should have a restricted scope. Thus, in designing a system, we concern ourselves with what objects see and are seen by another object or class of objects. This in fact is the purpose of one of the steps in our method, that of establishing the visibility among objects. In the worst case, all objects can see one another, and so there is the potential of unrestricted action. It is better that we restrict the visibility among objects, so as to limit the number of objects we must deal with to understand any part of the system and also to limit the scope of change.

Finally, every object has two parts, and so may be viewed from two different ways: there is an outside view and an inside view. Whereas the outside view of an object serves to capture the abstract behavior of the object, the inside view indicates how that behavior is implemented. Thus, by seeing only the outside view, one object can interact with another without knowing how the other is represented or implemented. When designing a system, we first concern ourselves with the outside view.

The outside view of an object or class of objects is its specification. The specification captures all of the static and (as much as possible) dynamic semantics of the object. In the specification of a class of objects, we export a number of things to the rest of the system, including the name of the class and the operations defined for objects of the class. Ideally, our implementation language enforces this specification, and prevents us from violating the properties of the specification.

Whereas the outside view of an object is that which is visible to other objects, the inside view is the implementation and so is hidden from the outside. In the body of an object or object class, we must chose one of many possible representations that implements the behavior of the specification. Again, if the language permits it, we may replace the implementation of an object or class of objects without any other part of the system being affected. The benefits of this facility should be clear: not only does this enforce our abstractions and hence help manage the complexity of the problem space, but by localizing the design decisions made about an object, we reduce the scope of change upon the system.

IV. Ada and Object-Oriented Development

Clearly, some languages are better suited than others to the application of object-oriented development; the major issue is how well a particular language can embody and enforce the properties of an object. Smalltalk and its immediate relatives provide the best match with these concepts, but it is also the case that languages such as Ada may be applied in an object-oriented fashion. Specifically, in Ada:

- Classes of objects are denoted by packages that export private or limited private types.
- Objects are denoted by instances of private or limited private types or as packages that serve as abstract state machines.
- Object state resides either with a declared object (for instances of private or limited private types) or in the body of a package (in the case of an abstract state machine).
- Operations are implemented as subprograms exported from a package specification; generic formal subprogram parameters serve to specify the operations required by an object.
- Variables serve as names of objects; aliases are permitted for an object.
- Visibility is statically defined through unit context clauses.

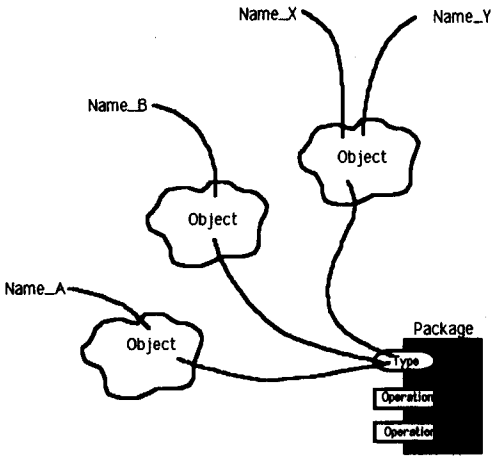


Fig. 6. Names, objects, and classes.

- Separate compilation of package specification and body support the two views of an object.
- Tasks and task types may be used to denote actor objects and classes of objects.

Fig. 6 illustrates the interaction of these points.

It is also the case that we can provide a form of inheritance using derived types. Thus, we could define a class of objects in a package that exports a nonprivate type, and then build on top of this class by deriving from the first type. The derivation inherits all the operations from the parent type. Because we have used an unencapsulated type (a type that is not private or limited private), we may add new operations, replace existing operations, and hide operations from the parent class. However, we must realize that there is a tradeoff between safety and flexibility. By using an unencapsulated type, we avoid much of the protection offered by Ada's strong typing mechanism. Smalltalk favors the side of flexibility; we prefer the safety offered by Ada, especially when applied to massive software-intensive systems.

Earlier, we used a few simple symbols to represent the design of the cruise-control system. It should come as no surprise that some people can grasp the essence of a design just by reading package specifications, while others are more effective if they are first given a graphical representation of the system architecture; we fall into the latter category. Since neither structure charts nor data flow diagrams capture the interesting properties of an object, we offer the set of symbols in Fig. 7, evolved from our earlier work [33]. We have found them to be an effective design notation that also serve to directly map from data flow diagrams to Ada implementations.

As Fig. 7 represents, these symbols are connected by directed lines. If we draw a line from object A to object B, this denotes that A depends upon the resources of B in some way. In the case of Ada units, we must make a distinction regarding the parts of a unit that exhibit these dependencies. For example, if the specification of package X depends upon Y, we start the directed line from the colorless part of the symbol for X; if the body of X depends

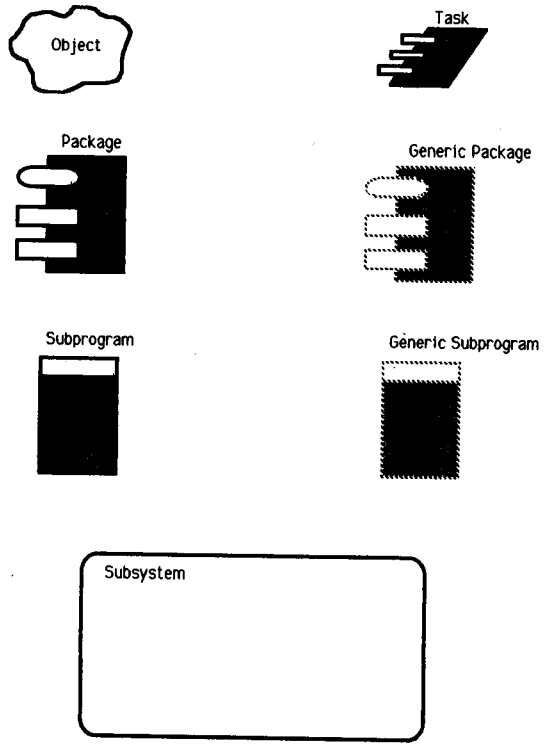


Fig. 7. Symbols for object-oriented design.

upon Y, we start the directed line from the shaded part of X.

V. Design Case Study

Let us apply the object-oriented method to one more problem, adapted from the study of Boehm-Davis [15].

There exists a collection of free-floating buoys that provide navigation and weather data to air and ship traffic at sea. The buoys collect air and water temperature, wind speed, and location data through a variety of sensors. Each buoy may have a different number of wind and temperature sensors and may be modified to support other types of sensors in the future. Each buoy is also equipped with a radio transmitter (to broadcast weather and location information as well as an SOS message) and a radio receiver (to receive requests from passing vessels). Some buoys are equipped with a red light, which may be activated by a passing vessel during sea-search operations. If a sailor is able to reach the buoy, he or she may flip a switch on the side of the buoy to initiate an SOS broadcast. Software for each buoy must:

- maintain current wind, temperature, and location information; wind speed readings are taken every 30 seconds, temperature readings every 10 seconds and location every 10 seconds; wind and temperature values are kept as a running average.
- broadcast current wind, temperature, and location information every 60 seconds.
- broadcast wind, temperature, and location information from the past 24 hours in response to requests from passing vessels; this takes priority over the periodic broadcast.

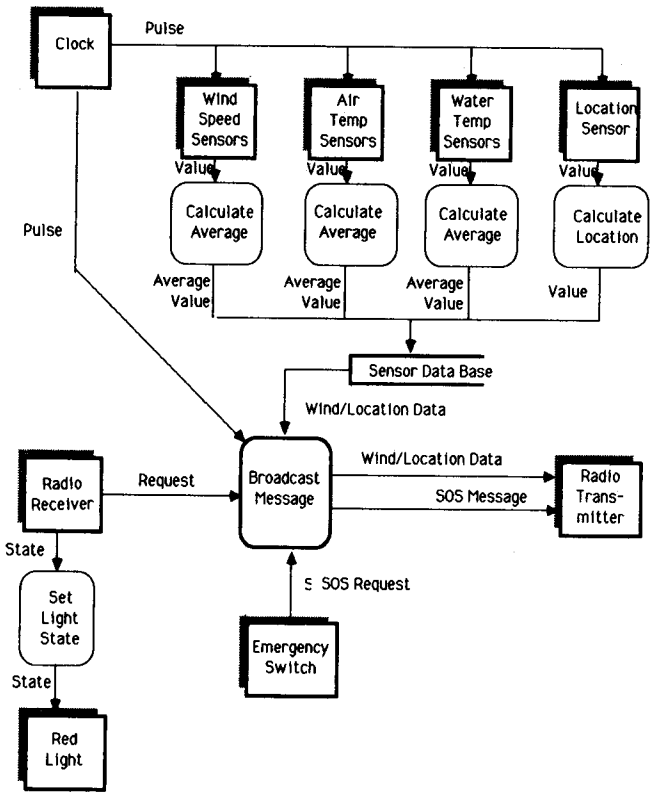


Fig. 8. Host at sea buoy data flow diagram.

- activate or deactivate the red light based upon a request from a passing vessel.
- continuously broadcast an SOS signal after a sailor engages the emergency switch; this signal takes priority over all other broadcasts and continues until reset by a passing vessel.

To formalize our model of reality, we begin by devising a data flow diagram for this system, as illustrated in Fig. 8. The design proceeds by first identifying the objects and their attributes. Drawing from this level of the data flow diagram, we include all sources and destinations of data as well as all data stores. In general, data flows have a transitory state; we will typically not treat them as objects, but rather just as instances of a simple type. Additionally, wherever there is a major process that transforms a data flow, we will allocate that process to an object that serves as the agent for that action. Thus, our objects of interest at this level of decomposition include the following:

• Clock	Provides the stimulus for periodic actions.
• Wind Speed Sensors	Maintains a running average of wind speed.
• Air Temperature Sensors	Maintains a running average of air temperature.
• Water Temperature Sensors	Maintains a running average of water temperature.

• Location Sensor	Maintains the current buoy location.
• Sensor Database	Serves to store weather and location history.
• Radio Receiver	Provides a channel for requests from passing vessels.
• Radio Transmitter	Provides a channel for broadcast of weather and location reports as well as SOS messages.
• Emergency Switch	Provides the stimulus for the SOS signal.
• Red Light	Controls the activity of the emergency light.
• Message Switch	Serves to generate and arbitrate various broadcast messages.

Next, we consider the operations suffered by and required of each object. We will take a first cut by simply listing the operations that characterize fundamental behavior. First, we identify the operations suffered by each object from within the system; these operations roughly parallel the state change caused by a data flow into an object:

• Clock	None
• Wind Speed Sensors	Take Sample
• Air Temperature Sensors	Take Sample
• Water Temperature Sensors	Take Sample
• Location Sensor	Take Sample
• Sensor Database	Put Value
	Get Value
• Radio Receiver	None
• Radio Transmitter	Broadcast Weather/Location Report
	Broadcast SOS
• Emergency Switch	None
• Red Light	Set State
• Message Switch	Request History Report
	Request Periodic Report
	Request SOS

Notice that for the Sensor Database, we have the operation Get Value which seems to go against the data flow implied by all other operations. In practice, we will encounter some objects that are passive in nature, especially those that denote data stores. Whereas Get Value does not change the state of the object, it returns a value of the state of the object. Since a passive object such as the Sensor Database cannot know when a value is needed, we must supply this operation to permit the state to be retrieved by another object.

Second, we must identify the operations required of each

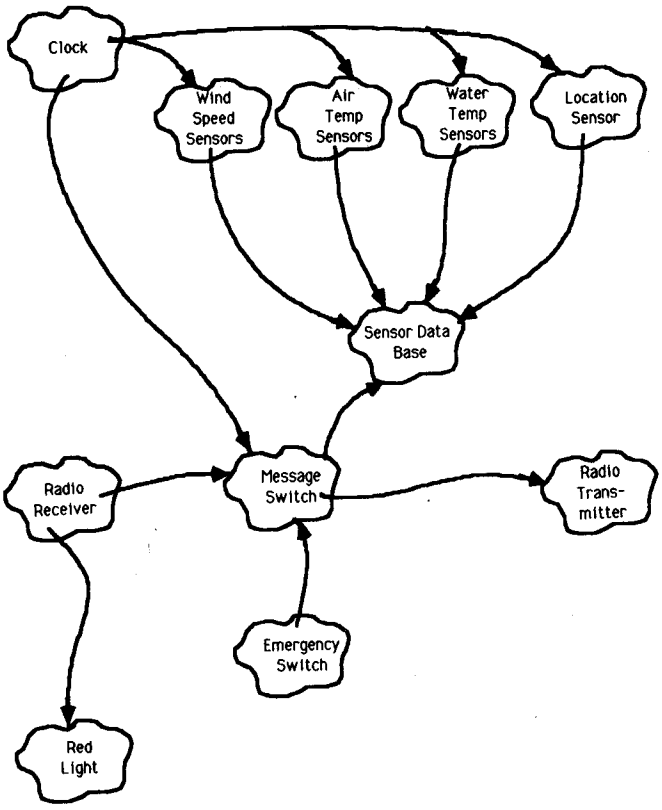


Fig. 9. Host at sea buoy objects.

object; these operations roughly parallel the action of a data flow from an object:

• Clock	Force Sample
	Force Periodic Report
• Wind Speed Sensors	Put Value
• Air Temperature Sensors	Put Value
• Water Temperature Sensors	Put Value
• Location Sensor	Put Value
• Sensor Database	None
• Radio Receiver	Force History Report
	Set Light State
• Radio Transmitter	None
• Emergency Switch	Force SOS
• Red Light	None
• Message Switch	Send Weather/Location Report
	Send SOS

Notice that we have a balance between the operations suffered by and required of all objects. For each operation suffered by an object, we have some other object or set of objects that requires that action.

This analysis leads us directly to the next two steps, establishing the visibility of each object in relation to other objects and its interface. Using the symbols we introduced earlier, we may start by indicating the dependencies among objects, as denoted in Fig. 9. In general, the dependencies follow the direction of the operations required of each object.

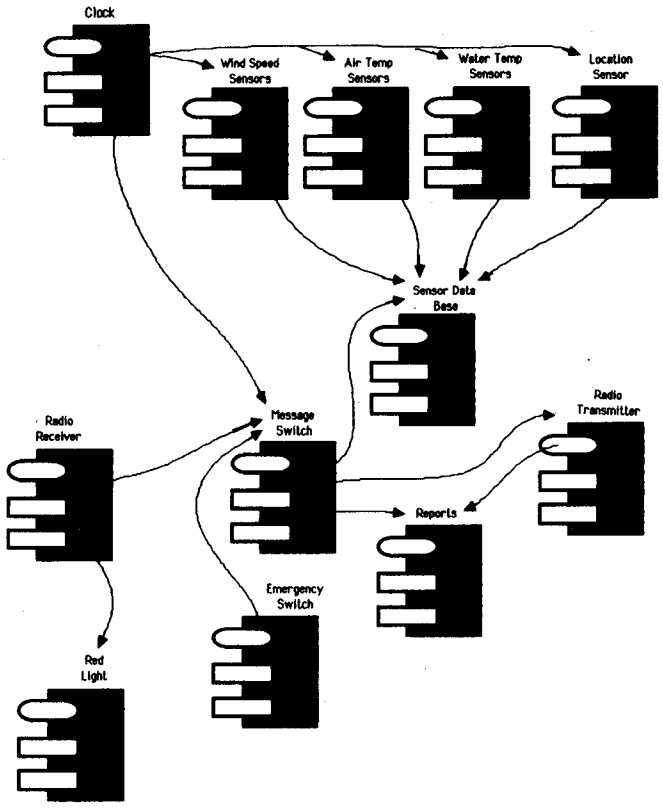


Fig. 10. Host at sea buoy objects.

In the previous section, we noted the correspondence between Ada units and objects. Hence, we may transform the design in Fig. 9 to an Ada representation. This transformation is simple: we denote each object or class of objects as a package, and for all but the most primitive data flows, we also provide a package that exports the type of the data flow, made visible to both the source and the destination of the flow. Fig. 10 illustrates this design after the transformation. Notice that there is a one-to-one correspondence between the objects in Fig. 9 with the packages in Fig. 10. We have only introduced one new package, Reports, which provides types that denote messages broadcast from the system.

Continuing our object-oriented development, we would next write the Ada specification for every package and then implement each unit. For example, we might write the specification of the Air Temperature Sensors as:

```
generic
   type Value is digits < >;
   with procedure Put_Value (The_Value: in Value);
package Air_Temperature_Sensors is

   type Sensor is limited private;

   procedure Take_Sample (The_Sensor: in out
      Sensor);

private
   type Sensor is . . .
end Air_Temperature_Sensors;
```

In this package, we export a limited private type (so as to provide a class of sensors) as well as one operation (Take_Sample). We also import one operation (Put_Value), that each sensor requires of the Sensor Database.

There are some interesting generalities we can draw from the design in Fig. 10. Notice that each package that denotes a sensor has the same set of dependencies and roughly the same set of operations that characterize its behavior. Therefore, it would be possible for us to factor out the similarities among these objects, produce one generic Sensor package and then treat each sensor object as an instance of this component. Furthermore, if we already have a simple data base package, we might adapt it to provide the Sensor Database instead of creating a new one for this application. Finally, if we are careful, we could write the Radio Transmitter and Radio Receiver packages such that they could be applied in other problems that use similar equipment.

In all these cases, we have identified the need for a reusable software component. Indeed, we find that there is a basic relationship between reusable software components and object-oriented development:

Reusable software components tend to be objects or classes of objects.

Given a rich set of reusable software components, our implementation would thus proceed via *composition* of these parts, rather than further *decomposition*.

VI. Conclusion

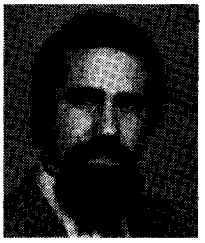
We must remember that object-oriented development requires certain facilities of the implementation language. In particular, we must have some mechanism to build new classes of objects (and ideally, a typing mechanism that serves to enforce our abstractions). It is also the case that object-oriented development is only a partial-lifecycle method and so must be coupled with compatible requirements and specification methods. We believe that object-oriented development is amenable to automated support; further research is necessary to consider the nature of such tools.

Perhaps the greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world. This leads to improved maintainability and understandability of systems whose complexity exceeds the intellectual capacity of a single developer or a team of developers.

References

[1] T. Rentsch, "Object-oriented programming." *SIGPLAN Notices*, vol. 17, no. 9, p. 51, Sept. 1982.
[2] J. Guttag, E. Horowitz, and D. Musser, *The Design of Data Type Specification* (Current Trends in Programming Methodology, Vol. 4). Englewood Cliffs, NJ: Prentice-Hall, 1978, p. 200.
[3] Adapted from an exercise provided by P. Ward at the Rocky Mountain Institute for Software Engineering, Aspen, CO, 1984.
[4] C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
[5] E. Yourdon and L. Constantine, *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[6] H. Levy, *Capability-Based Computer Systems*. Bedford, MA: Digital Press, 1984, p. 13.
[7] G. Curry and R. Ayers, "Experience with traits in the Xerox Star workstation," in *Proc. Workshop Reusability in Program.*, ITT Programming, Stratford, CT, 1983, p. 83.
[8] M. Shaw, "Abstraction techniques in modern programming languages," *IEEE Software*, vol. 1, no. 4, p. 10, Oct. 1984.
[9] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, Dec. 1972.
[10] R. Abbott, "Report on teaching Ada," Science Applications, Inc., Rep. SAI-81-312WA, Dec. 1980.
[11] ——, "Program design by informal English descriptions," *Commun. ACM*, vol. 26, no. 11, p. 884, Nov. 1983.
[12] M. Jackson, *System Development*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[13] M. Alford, "SREM at the age of eight: The distributed computing design system," *Computer*, vol. 18, no. 4, Apr. 1985.
[14] G. Buzzard and T. Mudge, "Object-based computing and the Ada programming language," *Computer*, vol. 18, no. 3, p. 12, 1985.
[15] D. Boehm-Davis and L. Ross, "Approaches to structuring the software development process," General Elec. Co., Rep. GEC/DIS/TR-84-B1V-1, Oct. 1984, p. 13.
[16] B. Meyer, "Towards a two-dimensional programming environment," in *Readings in Artificial Intelligence*. Palo Alto, CA: Tioga, 1981, p. 178.
[17] A. Borgida, S. Greenspan and J. Mylopoulos, "Knowledge representation as the basis for requirements specification," *Computer*, vol. 18, no. 4, p. 85, Apr. 1985.
[18] D. Boehm-Davis and L. Ross, "Approaches to structuring the software development process," General Elec. Co., Rep. GEC/DIS/TR-84-B1V-1, Oct. 1984, p. 14.
[19] B. MacLennan, "Values and objects in programming languages," *SIGPLAN Notices*, vol. 17, no. 12, p. 75, Dec. 1982.
[20] H. Levy, *Capability-Based Computer Systems*. Bedford, MA: Digital Press, 1984, p. 13.
[21] B. Liskov, "A design method for reliable software systems," in *Proc. Fall Joint Comput. Conf.*, AFIPS, 1972, p. 67.
[22] B. Liskov and S. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, Mar. 1975.
[23] J. Guttag, E. Horowitz, and D. Musser, *The Design of Data Type Specification* (Current Trends in Programming Methodology, Vol. 4). Englewood Cliffs, NJ: Prentice-Hall, 1978.
[24] M. Shaw, "Abstraction techniques in modern programming languages," *IEEE Software*, vol. 1, no. 4, Oct. 1984.
[25] A. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: M.I.T. Press, 1985.
[26] B. Cox, "Message/object programming: An evolutionary change in programming technology," *IEEE Software*, vol. 1, no. 1, Jan. 1984.
[27] "A symposium on actor languages," *Creative Comput.*, Oct. 1980.
[28] G. Myers, *Advances in Computer Architecture*. New York: Wiley, 1982.
[29] H. Deitel, *An Introduction to Operating Systems*. Reading, MA: Addison-Wesley, 1983, p. 456.
[30] B. Liskov and S. Zilles, *An Introduction to Formal Specifications of Data Abstractions* (Current Trends in Programming Methodology, Vol. 1). Englewood Cliffs, NJ: Prentice-Hall, 1977, p. 19.
[31] M. Shaw, W. Wulf, and R. London, "Abstraction and verification in Alphard: Iteration and generators," in *Alphard: Form and Content*. New York: Springer-Verlag, 1981.
[32] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Schiefler, and A. Snyder, *CLU Reference Manual*. New York: Sprinter-Verlag, 1981, p. 8.
[33] E. G. Booch, "Describing software design in Ada," *SIGPLAN Notices*, Sept. 1981.



Grady Booch (M'82) received the B.S. degree from the United States Air Force Academy, Colorado Springs, CO, in 1977 and the M.S.E.E. degree from the University of California at Santa Barbara in 1979.

He is Director of Software Engineering Projects at Rational, Inc. He has been actively involved in Ada research, implementation, and education since 1979. He has lectured on Ada and software development methods across the United States and in Europe, and has published over 30 technical articles. In addition, he has written *Software Engineering with Ada* (Benjamin/Cummings), and is currently working on another book dealing with reusable software components. He also is author of the column "Dear Ada" which appears regularly in *Ada Letters*. Previously, he was a faculty member in the Department of Computer Science at the United States Air Force Academy. In his other assignments, prior to leaving the Air Force in 1982, he was a project director and project engineer for several large real-time software developments for space and missile systems.

Mr. Booch is a member of the Association for Computing Machinery and the American Association for Artificial Intelligence. In 1983, he was given an award, for distinguished service to the Ada program, from the Undersecretary of Defense.