IS OBJECT-ORIENTED DESIGN SOUND SIMULATOR SOFTWARE ENGINEERING?

David C. Gross and Lynn D. Stuckey, Jr.
Boeing Defense & Space Group
Huntsville, Alabama

ABSTRACT

Every advance of software technology has led to the discovery of new barriers. The introduction of a new software technology expands the domain of solvable problems while revealing an undiscovered country of unsolvable problems. Generally, the advances have been revolutions which changed the paradigms defining common practice. Each revolution has threatened the existing order while it offered new power. The effect of this phenomenon has shaped the history of software technology into a series of searches for a "magic bullet". What is the new approach which will break the latest barrier down? There are problems with such a mind-set. One is that we come to expect paradigm shifts based on the calendar, and not necessarily on real progress. How do we decide if a technique is a real advance or a passing fancy? Another is that we used to integrate new philosophies into our paradigms, where now we adopt complete new paradigms, and discard what has gone before.

The current candidate for the holy grail is object-orientation. Since simulation is a software technology consumer, we view the possibilities of object-orientation with interest and concern. What problems will an object-orientation approach help us resolve and how? This paper presents a software engineering examination of the effect of object-orientation on simulation software. We review the fundamentals of an object-orientation. We expand on this understanding by discussing a contrived example of simulation software. Having defined the object-oriented methodology, we review the goals and principles which define software engineering, as a basis for our evaluation. Finally, we analyze the effects of an object-orientation for simulation software and draw conclusions about its utility.

ABOUT THE AUTHORS

David C. Gross is a software systems engineer with the Missiles & Space Division of the Boeing Defense & Space Group. He has worked in all phases of simulation and training systems from requirements development through delivery. He is currently involved in basic research for software engineering and applied research for system simulation. Mr. Gross has a Bachelor of Science in Computer Science/Engineering from Auburn University and is working toward a Master of Operations Research at the University of Alabama at Huntsville. His thesis investigates the utility of high level languages such as C++ and Ada for simulation.

Lynn D. Stuckey, Jr. is a software systems engineer with the Missiles & Space Division of the Boeing Defense & Space Group. He has been responsible for software design, code, test, and integration on several Boeing simulation projects. He is currently involved in research and development activities dealing with software development for weapon and threat simulators. Mr. Stuckey holds a Bachelor of Science degree in Electrical Engineering from the University of Alabama, in Huntsville and holds a Master of Systems Engineering from the University of Alabama, in Huntsville. His thesis presents a systems engineering approach to software development.

IS OBJECT-ORIENTED DESIGN SOUND SIMULATOR SOFTWARE ENGINEERING?

By David C. Gross and Lynn D. Stuckey, Jr.
Boeing Defense & Space Group
Huntsville, Alabama

INTRODUCTION

Object-Oriented is one of the hottest topics in software because it is touted as the latest solution to the "software crisis". We all know that we are in a crisis because of the flood of articles telling us so. The difficulty, like many of our country's social ills, is that while there is much consternation about the crisis and no shortage of proposed ideas, there is very little progress toward implementing a solution. On the other hand, companies and individuals are getting rich selling silver bullets to slay the beast. In the case of software, specialists are rushing countless papers, books, and products to take advantage of this heavy interest. Bhaskar has observed that the phrase object-oriented, "has been bandied about with carefree abandon with much the same reverence accorded 'motherhood', 'apple pie', and 'structured programming'".[1] The presumed motivation for the rush to object-oriented is improved productivity. However, little of the discussion has revealed the supposed improvements, arguing that methodology is good simply because it is object-oriented, and any methodology change is good.

A methodology is a collection of cooperating methods, guided by a philosophy. A method is a disciplined process for producing products. Many of the so-called software methodologies, bandied about in the marketplace of ideas, are not methodologies at all. They either (1) do not contain recommendations about how to actually do things or (2) they are too detailed to be useful. A methodology should spell out general steps to follow. It should be specific enough to give guidance but be general enough to apply to most software applications. It should not to be taken as a step-by-step way to develop entire systems. All-encompassing recipes for how to get the work done simply do not exist

The software industry has long concentrated on the evaluation of code as the only key to quality. However, the other industries have learned at the hands of the Japanese that you can not "inspect-in quality", you must "build-in quality". By definition, we have to understand the process by which a product is made before we can improve it. We give lip service to understanding, and then improving, software methodologies. Why? The problem is that introduction of a real methodology into the production of software would make it a controlled engineering process instead of a glorious dark room hack. The focus needs to shift to evaluating the software *process*, not just the software *product*. It is not clear that object-orientation is the methodology improvement to achieve this end. We must evaluate it against identified measures of effectiveness.

Software engineering provides the measures of effectiveness we require to evaluate methodologies. It is an appropriate basis for evaluation since it is an attitude toward software technology, not a specific technique.

REVIEW OF OBJECT-ORIENTATION

Object-orientation is an attitude (or mindset) to large scale software development. Object-oriented methods exist for every phase of the software development lifecycle, hence the interrelated terms of Object-Oriented *Analysis* (OOA), Object-Oriented *Design* (OOD), and Object-Oriented *Programming* (OOP). Adding to the general confusion, OOD is often used to mean the entire scope of object-oriented methods. Despite the breadth of scope for object-oriented methods, there are a set of concepts and techniques that define object-oriented. It is application of these techniques that makes a thing object-oriented, not the use of particular languages or tools.

Even so, the discussion will occasionally require examples which we will provide in the C++ programming language. C++ is particularly relevant to training system simulation for three reasons. First, the increasing popularity of C++ has provoked the United States Air Force into studying the lifecycle effects of C++ versus Ada. Second, some military simulation customers are considering requiring C++ for a project's programming language. Finally, the object orientation of C++ is an overlay of an existing language, namely C. The introduction of object orientation into Ada will be much the same in Ada 9X.

Objects

The term *object* means just what it always has -- something intelligible or perceptible by the mind. As such, in a piece of software, an object will be the focus of attention, thought, feeling, and effort. The items designated as objects in the software should have some clear relationship to the "real" world, that is, they should be the items which an expert in the field (but not necessarily a software expert) would expect to find. For example, if the software is to model an elephant, we would choose the first layer of objects as legs, ears, nose, tongue, skin, etc. This is in contrast to the classical design approach based only on functional requirements.

Nevertheless, the fundamental implementation of each object will be procedural; an object-orientation does not spare us the burden of telling the computer how and what to do. Algorithms in object-oriented design do not *look* much different than other designs. Much of the change is in selection of names, and the way functions are used. Historical work in computer science has addressed this fundamental problem -- how do we get the computer to do what we want? It is the success of these efforts which has led to a new problem, not how to tell the computer what to do, but how to coordinate the different pieces of software to make a unified whole?

Techniques

Object-oriented designs employ certain techniques. Unfortunately, there is no definitive set of techniques. The set discussed here are generally accepted. Object-oriented techniques fall into two groups; (1) the fundamental definition of objects and (2) the manipulation of those objects. Object definition applies the technique of encapsulation. Objects are manipulated through the techniques of inheritance, message passing, and dynamic binding.

The first technique, *encapsulation*, is the concept that the details of an implementation are hidden from its users. Borrowing from our elephant example again, although we suspect that its internal processes are extremely complicated we do not understand them, nor do we need to. Monitoring and controlling its internal processes are the elephant's problem -- unless of course we are the zoo keeper. Encapsulation separates users from owners, ensuring that users will interface with the object only at the intended interface points. Figure 1 illustrates the concept of encapsulation. In this figure, the shaded area within the object represents hidden (encapsulated) details which are isolated from external access. Manipulation of the object is possible only through the external members which are represented by the unshaded area of the object.

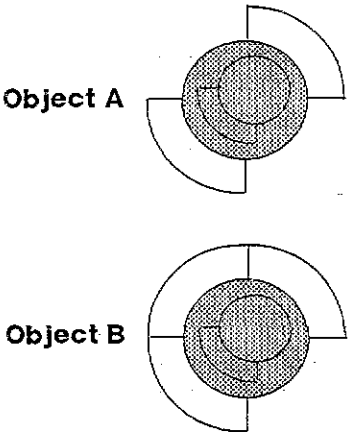

Figure 1

Technique: Encapsulation

The second technique, *inheritance*, exists to better support reusability. The notion of inheritance is that there is no such thing as an absolutely new start. Most new programs are really expansions or developments based on existing implementations. Again, if we are attempting to develop an elephant, and we already have built a generic mammal, would not this be a good place

to begin? The subsystems that an elephant shares with all mammals are already designed, built, and tested, and therefore all we need to build is the elephant unique subsystems. Multiple inheritance is a special kind of inheritance, which permits selection of which attributes and operations the new object wishes to inherit.

There is some debate as to what features constitute "real" inheritance. This is to some extent a conflict between the motivation of inheritance (to avoid redeveloping the wheel) and its implementation. Coad levies the following requirements for real inheritance: (1) share data structures (and instances of them) above, (2) share methods (and static instances of them) above, (3) ability to add new data structures, (4) add, extend, or override methods.[3] Figure 2 illustrates the concept of inheritance. Objects B, C, and D have inherited operation I. Object D can inherit either operation II-A (from object B) or operation II-B (from object C).

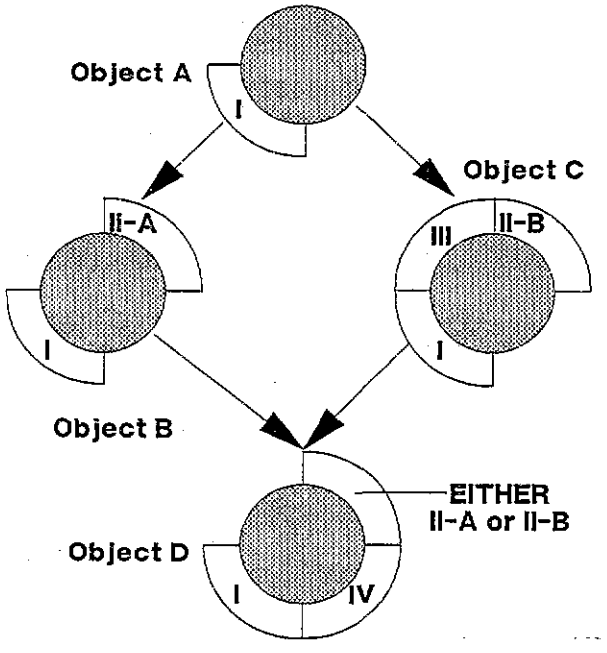

Figure 2

Technique: Inheritance

The third technique, *message passing*, addresses the problem of communication between objects. In a pure object-oriented approach, the message is a selection of one of the manipulations that an object knows how to perform. Less pure implementations, such as C++, permit variable data passed to objects as parameters. The thrust of this is that objects are independent actors, which do not depend on the state information of other objects. Figure 3 illustrates message passing.

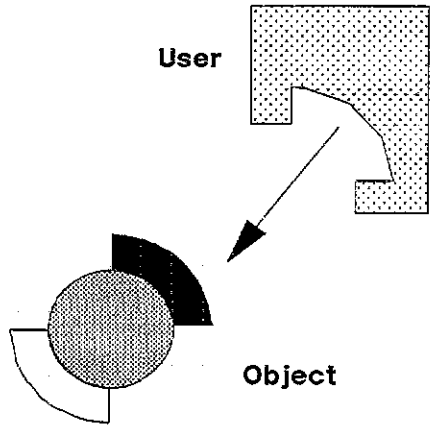

Figure 3

Technique: Message Passing

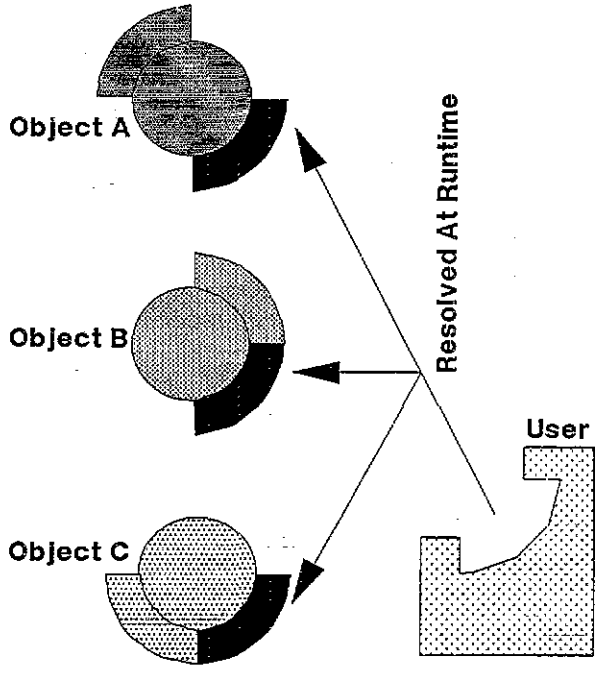

Figure 4

Technique: Dynamic Binding

Figure 4 illustrates the final technique, *dynamic binding*. It is sometimes called the most powerful feature of object-oriented programming. Dynamic binding delays the resolution of calls to

specific names until run-time. The opposite strategy, static binding, is demonstrated in languages such as Ada which resolve everything possible at compilation. Dynamic binding permits the design of polymorphic objects. Objects are *polymorphic* when calls to a single member method results in different operations based on the class of the caller. The dark shading on Figure 4 indicates an object member which fulfills the user's requirement -- however, the actual object that fulfills the requirement is resolved at runtime.

CONTRIVED EXAMPLE

At this point, the concepts of object-orientation may seem overwhelming. A walk-through a brief example should illuminate the differences and similarities between an object-orientation and a more classical design approach. The example is rather simplistic, but a discussion of the development of this software in terms of an object-oriented approach should be beneficial.

The example chosen is the generation of random variants. Simulations frequently model effects according to random processes. Most simulations require the capability to easily generate random numbers. In addition, the models may require a variety of probability distributions (such as the uniform, the normal, and the exponential distributions). Our contrived example is the design of a collection of software to provide this capability for random effects modeling.

How would an object-oriented approach differ from the classic approach? The fundamental algorithms for computing the specific random variate will not differ at all; the differences arise in the partitioning of the design into components.

An object-oriented approach proceeds from a different mindset. The first step is to identify the objects involved in the system. Object identification is not a trivial task. Most textbook examples deal with systems with a fairly obvious set of objects such as traffic (cars, pedestrians, roads, lights, etc) or graphics (pixels, circles, polygons, etc). But what are the objects in a system of random variates? Random numbers do not have correlations in the real world; we can not feel, taste, or hear them. When this is the case (and it is in a surprising number of cases), it may be helpful to imagine fictional machines which are fulfilling the system requirement. In our contrived example then, we can imagine a room of machines, each generating a different random variate. Next, we identify the information each machine must maintain to do its job, such as the random number generator seed, the last variate, and so forth. In addition, we must identify the processes the machine must do to perform its job. In the pure object-oriented approach, we must treat the machine as an individual actor, capable of receiving and carrying out an assignment without assistance. Just as in the classic approach, we may identify parts of the machines that can be grouped into a new machine. The information required by the machine will become the data members of the object. The processes performed by the machine will become the member functions of the object. And finally, the machines used in common will become the inherited classes of the object.

Figure 5 provides a graphical representation of the "uniformVariate" class, which illustrates the class members available. Not explicit on this

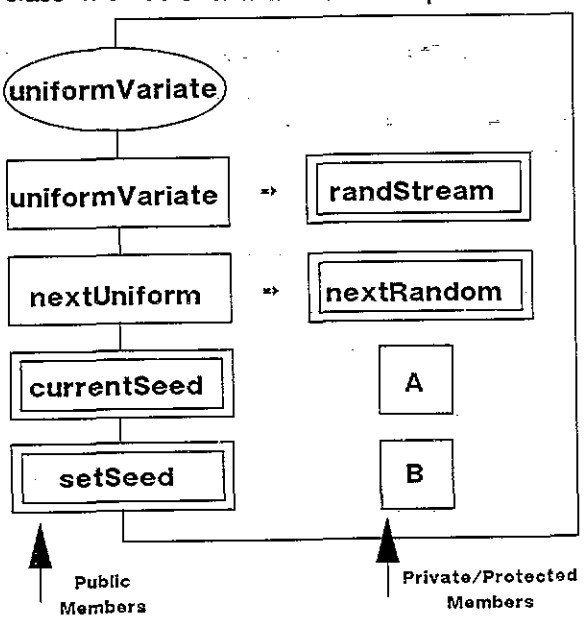

Figure 5

Class Definition in Contrived Example

drawing is the fact the uniformVariate inherits its members from a more fundamental class, "randStream". The double box indicates members which are inherited. The figure

distinguishes between public members, with external access, and private/protected members. Public class members may be publically inherited; protected members must be at most protected. There is a private member of the base class randStream, namely the data member stream, however it has no visibility at the class level illustrated. Even so, the visible members of randStream may access the private members of randStream. The remainder of the class definitions and elaborations to meet the requirements of our contrived example are not shown, but continue in a like manner.

REVIEW OF SOFTWARE ENGINEERING

Goals

The first step in evaluating software is to define the goals we seek to achieve. In our evaluation, we have adopted the four goals of software engineering, that have become the standard for software evaluation. These are maintainability, efficiency, reliability, and understandability. Each of the goals is important, and there is a tension that exists between them. Complete victory for one goal frequently defeats another. The goals can only be accomplished through good design. They cannot be efficiently added later.

Maintainability is measured in two ways: (1) the degree of difficulty in continuing to use the software in the face of changing equipment, requirements, and personnel over the life of the project, and (2) the ability to implement a controlled change in the software without adversely affecting the rest of the system. Maintainable software is desirable because the operational cost of software can be significantly greater than the development cost.

Efficiency is defined as the fact or quality of being efficient; competency in performance; the ratio of work done or energy developed by a machine or engine, etc., to the energy supplied to it. In the software world, efficiency breaks down into two basic concepts: execution speed and memory usage.

Reliability is the probability that a system will perform in a satisfactory manner for a given period of time under specified operating conditions. In software, reliability means to operate in a manner that satisfies the window of normal operation without intervention. This satisfaction includes the ability to handle degraded operation and a graceful handling of error conditions.

Understandability as a software goal is one of the easiest to implement, but is most often ignored. It is the quality of the map from the software's model of reality to the real world. The software is a diagram of the requirements, design and implementation.

Principles

The principles of software engineering are *qualitative characteristics* within the software that help measure progress toward the goals. Just as the goals, the principles matter in every phase of the lifecycle. *Abstraction* deals with how one views the system. The essence of abstraction is to extract essential properties while omitting nonessential details. *Information Hiding* is the characteristic in software that certain details that should not affect other parts of a system are made inaccessible. Information hiding therefore conceals how an object or operation is implemented. *Modularity* is purposeful structuring of the physical architecture of the software system. Modularity deals with how the structure of an object can make the attainment of some purpose easier. *Localization* helps to create modules that are loosely coupled to the outside world and cohesively strong internally. It is concerned with physical proximity of software components. *Uniformity* simply means that the software utilizes a consistent notation and is free of any unnecessary differences. *Confirmability* implies that the system is designed so that it can be readily tested. It increases the credibility of the design by building validation into the code.

Evaluating Methodologies

Software has long been the darkside of engineering. It required gurus and wizards to develop software that worked. This was acceptable when the bulk of software was written in assembly language and relegated to small microprocessors in an insignificant part of a system. However, as software languages and

computers advanced, our software methodologies have not. A methodology must be judged as to its ability to infuse the software engineering principles into the software process and the resulting software product. The evaluation must be focused on the *big* picture. The lack of an overall systems mindset has been a failing in methodologies from the onset of software development.

EVALUATION

Once object-orientation is understood and the criteria for evaluation are defined, it is now time to evaluate if object-oriented design is sound simulator software engineering. Our experience with software engineering techniques derives from our work in various real-time systems, ranging from flight crew trainers to desktop scenario simulators to embedded weapon computer systems. The evaluation following arises from our experimentation with Ada and C++ in these environments. The discussion raises the most general effect of an object-oriented approach, and briefly touches on specifics in C++.

Before we launch into the structured evaluation, there is one general point that needs to be made. The most important driver of the quality of simulation software is the quality of the simulation "map" to the real world. In fact, system simulation is simply the construction of useful models of real-world systems. These models cannot be useful if the map between them and reality is obtuse or non-intuitive. The ability of non-software specialists to grasp and comprehend the map determines the level of credibility of the simulation. As such, an object-orientation would seem to be the ideal approach for design of simulation software. After all, the model's map is the transformation of real-world objects into software objects. And this is in fact true, at least in the case of the underlying philosophy of the object-orientation approach. However, some of the techniques are less useful.

Software Engineering Principles

Figure 6 illustrates how the object-oriented techniques encourage application of the various stated software engineering principles.

Software Engineering Principle	Object Oriented Technique: Encapsulation	Inheritance	Message Passing	Dynamic Binding
Abstraction	X	X		
Information Hiding	X			
Modularity	X		X	
Localization	X			
Uniformity			X	
Confirmability	X	X		

Figure 6
Principles Versus Techniques

The encapsulation technique is strongly aligned with the software engineering principles. In C++, for example, each class member (data or function) can be declared at three layers of access: *private, protected, or public.* Private items are limited within the class, protected members are visible within the class and inheritors of the class, and public members are visible within the class, to inheritors, and to declarers of objects of the class. This directly promotes the principles of abstraction and information hiding. The presence of encapsulation provides a vehicle for implementing the modularity principle's call for varying cohesion within the software architecture. Encapsulation tends to promote standalone testable units for confirmability.

Inheritance is something of a mixed bag. Inheritance addresses the call of abstraction for decomposition in levels, by building a defined hierarchy of objects. An object which inherits depends on another, simpler object. However, languages such as C++ provide "escape clauses", such as C++'s "friend". It permits non-members to access private and protected members across class boundaries. This is a direct violation of the abstraction, modularity, and information hiding principles. In its favor, inheritance promotes comfirmability when a class inherits a previously proved and tested simpler class. However, it can dramatically increase confirmability problems when unwanted functions are inherited. All public members of the subclass are inherited and accessible, even if the superclass is unaware they exist. This is particularly worrisome in the case of virtual functions (see discussion of dynamic binding below). The

capability for multiple inheritance (inheritance from multiple subclasses) further complicates the problem of tracking unforeseen inheritance.

Message passing promotes a fully uniform approach to object interface. By definition, it will also promote loose coupling between object layers in a hierarchy which works toward modularity. Message passing enforces the central *object-oriented assumption that objects are* independent actors, capable of completing any assignment. In its purest form, message passing discourages the transmission of data in a message simply relays an assignment to the object. However, simulation objects emulate a world of tremendous complexity and interaction. In fact, analysis of object interaction is frequently the point of the simulation. Message passing discourages object interaction. At low levels in the architecture, objects require access to large amounts of data created by other objects (if they are to incorporate a single design decision). At high levels, the number of different assignments which an object might perform can make the pos*sible messages grow exponentially, if they must* be unique.

Dynamic binding delays decisions about linking calls to procedures at runtime. Some object-oriented articles feature it so prominently that you might think that it will also write the code for you. This (unfortunately) is not the case. In C++, dynamic binding is implemented through the inheritance of virtual functions. The notion is that many functions at different levels in the class hierarchy have the same conceptual purpose but different detailed *implementations. For example,* if you know how to move a stick is it not obvious how to move a table (which inherits stick)? The answer is no; although if the stick's move function is virtual, the table class will think it does know. This kind of blind inheritance violates modularity and information hiding. In truth, dynamic binding sidesteps all the software engineering principles.

Software Engineering Goals

Encapsulation is a step toward the goals of software engineering. It improves the quality of the simulation's map to reality, making software more maintainable. The ability to define the members of objects, and compare them against the real world increases understandability. Clever use of encapsulation works to mitigate some of the less desirable effects of other techniques.

Inheritance can make the creation of new software quick, particularly with small changes to existing class hierarchies. However, inheritance increases the difficulty of controlling changes and error propagation. Implicit inheritance decreases the *understandability* of the software to a engineer new to a class hierarchy. The inability in some object-oriented languages to *disinherit* means that classes will provide unwanted access to members.

Pure message passing (no data) is not a particularly efficient way to communicate between low level objects. Low level objects generally interchange data in order to reduce redundancy. Message passing becomes more practical for object interfacing as we move up the object hierarchy. However, it *can be extremely* inefficient as the power and scope of high level objects expands. Languages such as C++, which support parameter passing into member functions do not over-restrict object interfacing but neither do they support pure message passing. Message passing is a very reliable method for communication while it is inefficient.

Dynamic binding can result in unmaintainable, unreliable, un-understandable, and inefficient code. Blind inheritance of subclass virtual functions can permit an object to perform undesired operations. Delaying link decisions to runtime can be grossly inefficient in execution time. After all, the code to implement the operation had to *exist at some compile time, or it would not link* at run time either. The argument is made that dynamic binding reduces the need to recompile objects while editing code. On the other hand, languages like Ada attempt to encourage error correction early in the lifecycle, when they are cheap to fix. The tools to find and fix bugs are almost always better at development workstations than on simulator platforms. The cost of test time is always lower on a workstation than using a multi-million dollar simulation to test code. And the later an error is discovered, the more likely it is that the product will ship with the error because there was no time or money to correct it. Dynamic binding flies in the face of these

arguments. Dynamic binding is a cute feature looking for a problem that needs it.

Pitfalls

Some object-oriented implementations make messages simply requests for action by independent actor objects. However, many simulation objects need to communicate their internal states to other objects or to some central information controller. Consider a model of a surface-to-air missile site. If all it has is message passing for inter-object communication, how will it alert other objects (which it may not know exist) that it has activated its radar? This creates the problem of communicating state information between objects. One way to get around this problem would be a global common area -- which involves a well-known loss of control. Another is for each object to independently create the information it requires -- which is inefficient in terms of execution time and space. The only acceptable solution is parameter passing.

The unseen problem of message passing is that it hides the control of objects inside every object within the system that has visibility to the object's messages. How then can a object control objects in a system when it does not know they exist? A better approach is to separate the physics, interfaces, and controls of each object -- an approach which suggests parameter interfaces and a hierarchy of objects.

The typical instantiation of an object creates fresh copies of every member object -- including functions! C++ permits declaration of a static member functions to avoid this problem. Careless definition of object classes in a dynamic environment, such as weapon system simulation, can lead to the creation of thousands of new copies of objects. Notice that creation of an object creates all of its explicit members, and its implicit members through inheritance.

Software engineers using languages which support packages are used to seeing subprogram hierarchies captured in code. However, these hierarchies are a special subset of all possible hierarchies, in that they resemble "trees". Each node in such a hierarchy is the father of some set of children, which can in turn be fathers of new nodes. The child can not be the father of its father. In addition, such software engineers are used to dealing with hierarchies at the "root" (small end). Both of these rules of thumb are turned on their head in pure object-oriented designs. While package-oriented languages tend to grow "top down", class hierarchies tend to grow "bottom up". Through forward declaration of objects, virtual objects, multiple inheritance, and polymorphism, child nodes may in fact be the father of their father! This can have the effect of leading to a new kind of spaghetti code. Classic spaghetti code resulted from reckless jumping around within a code segment (goto). "Advanced" inheritance features provide the opportunity to jump aimlessly around class hierarchies. Is polymorphism the "goto" of object-oriented programming?

The implementation of object details is (by definition) hidden. While this generally implements a principle of software engineering, the application of other object-oriented techniques can led to inter-object thrashing. The principle of message passing requires that threads of control pass to objects to complete the tasks. The object to which control is passed must complete the operation before returning control. The possibility of blind multiple inheritance may result in the original and the new object waiting for each other to complete a task. Depending on the architecture implementation, this can lead to a rapid exhaustion of the stack, or resource deadlock.

The pure object-oriented approach calls for objects to be independent, equal actors. However, this assumes that some object in the system has the ability to request operations from other objects, and to continue to do so for the duration of the simulation. Clearly, this is some implementation of an "executive" object. Coad [3] solves this by the creation of four different executives: problem domain, human interaction, task management, and data management. Given the intense nature of the typical simulation model, we would suggest a further subdivision of the problem domain component into "managers". An example would be a platform/projectile manager, an environment manager, a command manager, and a scenario manager. Notice that extension of this concept begins to look very much like the top down architecture well supported in Ada.

CONCLUSION

Of all the monsters that fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.[2]

Object-orientation is a specific way to decompose a system design, and it is radically disjoint with other approaches such as functional decomposition, data flow, etc. However, system decomposition is just one phase of the system lifecycle. The real measure of any methodology is its utility and effect throughout the lifecycle. Many object-oriented authors tout it as significantly reducing development time in the lifecycle. Perhaps, although this presupposes access and willingness by designers to use object hierarchies. We suspect the willingness is almost certainly lacking, given the readily apparent reluctance of software designers to participate in cooperative methods (structured programming, Ada, ...). Even if the designers come around, the real cost problem with software is not development, but system maintenance. Software maintenance consumes approximately eighty percent of the software system cost. As we have seen, there are real maintainability concerns with object-oriented methods.

The difficulties with object-orientation arise not so much from the motivation of the techniques as the specific implementation of them. We have seen this occur in languages before, for example with Ada tasking. The motivation for Ada tasking was noble, but the implementation did not support the determinism requirement for embedded applications. Likewise, the motivations behind the object-oriented techniques are noble, even if their implementation does not directly support embedded software's requirements.

The philosophy of object-orientation is valuable, but frequently covered up with an implementation. The best example of this is dynamic binding. This has become the holy grail of object-oriented languages, to the point where some references deny that a language, tool, or design can have value without it. Some object-oriented experts are pushing to require pure object-oriented languages to be purely virtual. But what is the benefit of dynamic binding? It delays resolution of names until run-time, which may reduce some design effort at the cost of substantial run-time penalties. Some claims for dynamic binding imply that it will generate code for you, but this is not the case. The central issue is design engineering versus ad hoc hacking.

Despite protestation to the contrary, object-orientation applies many techniques developed earlier such as structured programming, controlled interfaces, procedure control, and algorithmic modeling. Object-orientation actually attempts to define a philosophical framework for the application of these techniques to programming large problems. The object-orientation community has contrived a dichotomy between the "object-orientation" and "conventional" approaches. This presumes that either of these options are well-defined and widely practiced.

Object-orientation has to be integrated into software process and not taken as a silver-bullet that is used to the exclusion of all other methods and without regard to past experiences and lessons learned. Object-orientation has to be tailored. In the experience of the authors, this tailoring results in an object-focused approach. Object-focused means that the object-oriented philosophies are integrated into the software development process but not necessarily all the implementation/paradigm rules. Object-orientation becomes an integral enhancement to the software methodology, not a replacement.

REFERENCES

[1] G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.

[2] F. Brooks, "No Silver Bullet", *IEEE COMPUTER*, April 1987, p. 10.

[3] P. Coad, E. Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.

[4] D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development", *IEEE COMPUTER*, January 1992, p. 8.