A Gentle Introduction to Object-Oriented Software Principles

Maj. Christopher Bohn, Ph.D., and John Reisner
Air Force Institute of Technology

For many of today's software professionals, the concepts behind object-oriented (OO) software principles are as familiar as the tips of their fingers; for others, this is not the case. If you are a structured-programming developer who is beginning to suspect that OO is not simply the flavor-of-the-decade, or if you are the program manager of a software-intensive system who has never written code, you may be wondering: Just what is this OO paradigm? This article will help you find that answer.

The notion that an object is *an instantiated member of a class* is a comfortable and familiar one to Generation X programmers who were educated in the 1990s and to programmers with Ada95, C++, or Java experience. However, to those who have backgrounds replete with Pascal, COBOL, or FORTRAN skills, the OO paradigm can be intimidating, confusing, or even exasperating. If you examine OO code, the constructs seem familiar enough: declarations, loops, functions, and procedures. So, what about OO development is so vastly different, unique enough that it even has been designated as a new paradigm?

In a nutshell, the OO approach:

> ... encapsulates data with corresponding operations and employs polymorphic mechanisms such as class inheritance. With data encapsulation, object classes can be conveniently reused, modified, tested, and extended. [1]

This quote, while a concise summary of the paradigm, is laden with OO-speak, providing little help to the curious neophyte.

This article explains several key aspects of OO software – abstraction, encapsulation, information hiding, decentralized problem solving, inheritance, polymorphism, and reuse – with the aim of presenting a clear overview and summary of the unique strengths of object-oriented software development.

Abstraction

Abstraction is a term used to describe putting certain parts of a system in a black box; that is, to ignore the implementation details of that subsystem. We frequently use abstraction in everyday life: We start our automobiles without thinking about ignition systems or internal combustion, we send documents to printers without fully understanding spoolers and queues in the network.

With abstraction, system developers do not need to think about systems in their full complexity. It would be the rare automobile designer who could simultaneously keep track of details about the exhaust manifold and the seat belt buckle; similarly, software developers need to *abstract-away* details of some parts of the system while focusing on others. Also, with abstraction, code can be reused without full knowledge of the reused component. Instead, we can focus on an abstract mental model of the component's expected behavior, which we will refer to as its *cover story* or *contract*. As long as reused code maintains the cover story, and as long as it does not violate its contract, it does not matter how its task is accomplished. Again, consider the automobile designer: Subsystems that rely on the correct operation of an engine can be designed to work regardless of whether ignition timing is accomplished with a mechanical distributor or by using an electronic timing mechanism. So long as the timing works correctly, the cover story is not broken, and the system will work.

Telling programmers that they *do not need to know* implementation details might make them uneasy. Should they not be concerned about reliability, supportability, and efficiency? Absolutely. Abstraction is not used to integrate shoddy code into a system under the guise of reuse. Rather, we do not *need to know* the details because the contracts already provide sufficient information about the expected object behavior. The end goal is to have a software architecture that facilitates both substitution and reuse. This also helps the system evolve with minimal *collateral damage* from changes (*i.e.* we can change one part of the system without having to change others, an important and recurring theme in OO).

Encapsulation

Reuse is nothing new – code reuse libraries have been around for decades. But in OO, we reuse more than algorithms; we reuse classes.

In OO, classes and objects are closely related; an *object* is a particular member (or *instance*) of a *class*. Classes are a kind of template, providing a means to define both the *state* and *behavior* of an object.

For example, a car's *state* might be described as *engine running, traveling east at 45 mph, with a total of 150 miles traveled*. We might expect the possible *behaviors* of a car to include turning the engine on, turning the engine off, moving at some speed and direction, changing the speed and direction, and retrieving the total distance traveled.

Once the object's state and behavior are described, we can then describe a *class* of cars. This is done by identifying a car's *attributes*, as well as the methods (sometimes called the *operations*) that a car can perform. An example of this is shown in Figure 1.

In an OO software system, an Automobile class is likely to look quite a bit different than the one shown in Figure 1. After all, Defense Advanced Research Projects Agency (DARPA) Grand Challenge [2] aside, computers do not change the direction and speed of a car; drivers do. However, consider a hypothetical information system for a state's bureau of motor vehicles. Such a system might also model a class for automobiles, but it would look more like the one depicted in Figure 2.

An object is an *instance* of a class. When we create a new instance, the attributes of that particular object will take on specific values. For example, if someone bought a brand-new Lexus as a gift for their spouse (like on those holiday commercials), that would necessitate a trip to the license bureau, where a friendly clerk would *instantiate a new object* of the Automobile class type (actually, the clerk would do this unknowingly – thanks to abstraction – merely by clicking on a New Registration icon). For this instance, Make = Lexus, Model = GS300, Year = 2007, Color = Silver, etc. The attributes of a class are analogous to a *record* in languages such as Pascal, while the methods of a class are implemented as procedures and functions. Collectively, the values of the attributes define the object's *state*. Essentially, the attributes and methods jointly implement the class' contract: They specify what information the object contains, as well as its expected behavior.

OO systems place the data (the attributes) and the operations that use that data (the methods) together. Moreover, the object's state can only be changed by using the operations defined in the class. Organizing software systems in a modular way is not new [3], but this *encapsulation* of attributes and methods – that is, the grouping of data and behavior in the same entity, as opposed to them being dispersed throughout the system – is a key differentiator between the structured and object paradigms.

Information Hiding

As a rule, when we define a class, we specify that the methods are *public* while the attributes are *private*; that is, objects can call the methods of another object, but no other object can have direct access to the values of its attributes. This means that the only way to read or change the state of an object (*i.e.*, to read or change an attribute's value) is through the class' methods.

In our Figure 2 example, we can assume that the Register method will assign initial values to all of the attributes (the first six likely entered by the clerk, while the seventh might be automatically generated with a calendar function, e.g., *one year from today's date*). The renew( ) method will only affect one attribute, plateExpirationDate. If some other scenario requires other read or write access to the variables, then methods to perform these reads or writes would need to be added to the class. For example, we might want to add a changeColor( ) method, so that if an auto owner repainted his car, that change could be reflected in the system without having to re-register the vehicle. However, there would not be a changeYear( ) method, as the model-year of an automobile would not change.

Using the methods to access the attributes means that the information is *hidden* by a *public interface*, (*i.e.*, the data are hidden behind the public methods of the class). If the information were not hidden, then another object could directly access and change an Automobile object's year, either accidentally or by design causing that object to break its cover story (the modified contract states that Automobiles can be registered, renewed, and re-painted, but not artificially aged). Returning for a moment to the real-world automobile (Figure 1), anyone who suspects that a screwdriver has been used to tamper with a car's odometer can sympathize with the perils of direct access to data that should be protected.

In the realm of OO, we can assume that any part of a component we depend upon will always behave as expected, and that no other part of the system will cause our part to violate a contract. Encapsulation and information hiding keep both of these assumptions safe.

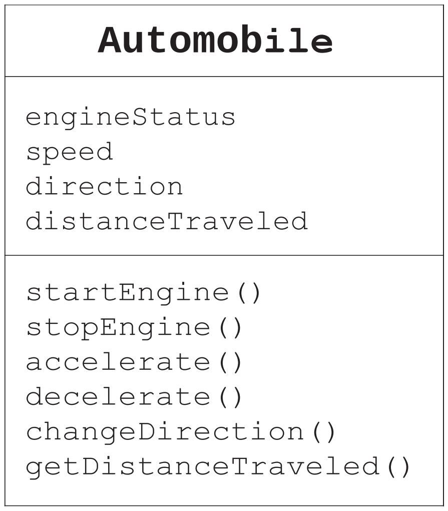
Decentralized Problem Solving

Thus far, our examples have been limited to a single class (i.e., the Automobile class). In a small-scale system, encapsulated objects are a nifty idea, but the full benefits of these assumptions are not realized until a system with many interacting classes is designed.

OO software uses the concept of *message* passing to create an architecture in which objects request services from and provide services for each other. When you want an object to do something, you *invoke one of its methods* by *passing a message*. This is very similar to making a procedure or function call. However, there are two notable differences, one being physical (the called code resides in a different and separately compiled module from the calling code); while the other is conceptual (the method invoked is part of a class' predefined public interface). These differences allow us to design systems that have interacting classes of objects with well-defined responsibilities. In a vehicle registration system, such an architecture may not yield much of benefits, but in other applications (such as a windows-based operating system, or a suite of gaming software), the potential benefits are quite high.

Perhaps more importantly, the object paradigm allows us to create decentralized solutions to computing problems. Decentralization permits us to solve complex problems that we could not solve with a centralized solution. For example, consider the post office. If the clerk behind the counter was responsible for delivering every incoming piece of mail, then the postal service would be very inefficient indeed. Instead, the post office employs classes of people and machines with well-defined responsibilities – clerks, sorters, handlers, deliverers – each doing their part to deliver around 100 billion pieces of mail annually [4].

Figure 1: *A Class of Automobiles, Defined By State (Four Attributes) and Behavior (Six Methods)*



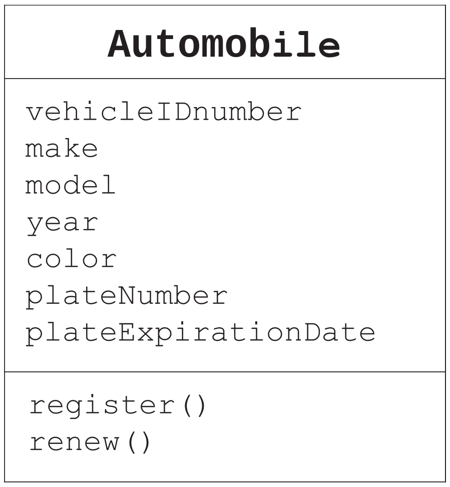
Inheritance

In the course of developing OO software, we often find that we need a class of objects that *specializes* an existing class. For example, a Leased Automobile is a special type of automobile. Besides the make, model, and other attributes that every automobile has, a leased automobile has another attribute: the lessor. Another special type of Automobile is a state-owned automobile. Through a mechanism called *inheritance*, we can formalize this relationship. We call Automobile the *parent class* or *superclass*, while LeasedAutomobile and StateAutomobile are known as *child classes* or *subclasses*. This is sometimes called the *is-a* relationship: a LeasedAutomobile is-a kind of Automobile. Everything that is true about a parent class is also true about its child classes; the special types of automobiles have all the same attributes and operations as a standard automobile. Plus, they have additional attributes and/or operations. For example, a Leased-Automobile also has a Lessor attribute, and a StateAutomobile has a scheduleMaintenance( ) operation. The attributes and operations in Automobile need not be reproduced in its subclasses; they are *inherited*.

Another situation is known as *generalization*. For example, boats also need to be registered. If we discover that there are many similarities between boats and automobiles, we can create a new class, Vehicle, and make Automobile and Boat subclasses of Vehicle. The commonalities among Automobile and Boat are then moved into the Vehicle superclass, possibly renaming attributes that represent the same concept by different names.

Inheritance leads to improved reuse and maintainability. Because subclasses inherit all

Figure 2: *Another Automobile Class, One More Likely to be Found in an Information System*



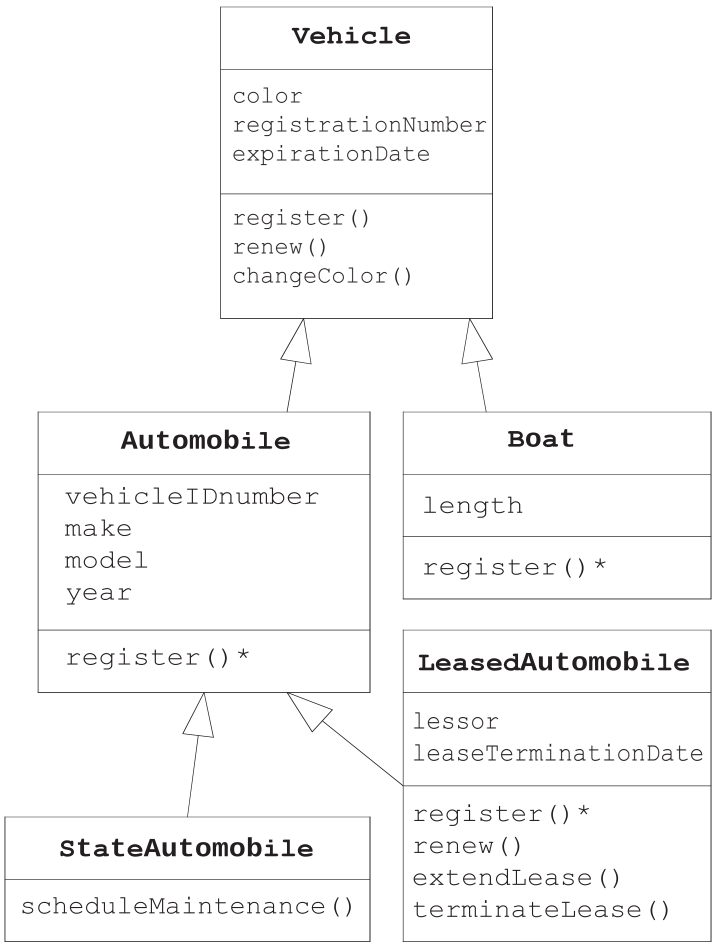


Figure 3: *An Inheritance Hierarchy of the Vehicle Class*

the attributes and operations of their superclass, they automatically get all the debugged functionality that is already in their parent class. When creating a child class, we need only concentrate on the ways in which the new class is different from its parent.

Maintainability improves because when a change is made to a parent class, then the subclasses (and *their* subclasses) automatically get the changes, too. Make the change in one place, and the change is propagated through the entire inheritance hierarchy. Another way maintainability is improved is through isolation of the subclasses. If a change needs to be made for only one specialized class, then those changes can be made without any risk of introducing new errors into other classes. This increased maintainability is even more beneficial as code evolves over time.

Polymorphism

Because everything that is true about a parent class is true about its child classes, instances of the child classes will maintain their parent's cover story as their own. They will have a richer cover story, to be sure, but they should not break the cover story they inherited. To put it another way, their contract is an amendment of their parent class' contract: They may promise a little more, but they will also honor their parent's contract.

This is the beginning, but not the end, of the concept known as *polymorphism*. Recall that one of the purposes of abstraction is to support the substitution of one component with another that honors the first component's contract. Polymorphism is a special form of substitution, in which the component to be used may not be known until the program is running.

There is more to polymorphism than simply substituting one class with another that has a few extra attributes and methods. When we create a subclass, we can *override* some of the methods in the superclass. That is, the method has the same name and parameters as the method in the parent class, but actual code for the method is different, providing the behavior appropriate for the particular subclass. Consider the inheritance tree in Figure 3; we have marked with an asterisk the operations that are overridden in the subclasses. The register( ) operation has been overridden by Auto=mobile, Boat, and LeasedAutomobile so that the clerk would be prompted to enter values for the additional attributes. StateAutomobile, though, reuses the register( ) operation provided by its parent class. Similarly, all classes except LeasedAutomobile reuse the renew( ) operation provided by Vehicle, since they do not need any special behavior. LeasedAutomobile, however, overrides renew( ), since the lessor may need to be consulted.

When the software expecting a Vehicle object is provided an object whose specific class is somewhere in Vehicle's inheritance hierarchy, the operations appropriate to that object will be invoked automatically (the details of how this happens are well beyond the scope of this paper). For example, if an Automobile object is registered, the code found in the register( ) method of the Automobile class is executed. However, when a LeasedAutomobile object is registered, then the register( ) code executed is the overriding method found in Leased Automobile. This selection of the correct method to be executed occurs automatically, without any embedded if-then-else or case logic required; therein lies the power of polymorphism.

Tying It All Together

The concepts presented in this article are often difficult to grasp, especially in a single sitting. Moreover, people who are just beginning to understand these concepts often find that they can follow an OO designer's line of reasoning, but have a more difficult time when it is their turn to try it out. Some authors have said that it takes six to nine months of experience to become proficient in fully exploiting OO techniques [5]. Further, the impact of adopting an OO mindset is not limited only to design; indeed, it has implications throughout the development life cycle, from requirements analysis to design, to implementation and test, and on through maintenance. Volumes could be (and have been) written on each of these topics. To become more familiar with the breadth and depth of OO development, we recommend that you investigate educational opportunities from a reputable continuing education provider, and that you find a software engineer with mature OO skills to serve as a mentor as you grow into increasingly larger, increasingly critical endeavors.◆

Additional Reading

If you are interested in learning how the OO approach influences the software life cycle, you may want to start with these books. Alistair Cockburn's *Writing Effective Use Case*s can help you turn your functional requirements into use cases, which will drive the rest of your project's development; Doug Rosenberg's *Use Case Driven Object Modeling with UML: A Practical Approach* does a fine job of describing the activities you would use to begin your design from those use cases. Implementation issues can be language-dependent, but Steve McConnell's *Code Complete* and Timothy Budd's *An Introduction to Object-Oriented Programming* address the issues as language-neutral as possible. Paul Jorgensen's *Software Testing: A Craftsman's Approach* is a good book for testing in general, and the last five chapters specifically cover object-oriented testing. Similarly, Thomas Pigoski's *Practical Software Maintenance* covers the breadth of software maintenance, with a chapter dedicated to OO's impact. If you're already familiar with OO concepts, you may want to look at some more advanced books, such as Ken Pugh's *Prefactoring*, Joshua Kerievsky's *Refactoring to Patterns*, and David Astel's *Test-Driven Development: A Practical Guide*.

References

1. Lee, G. Object-Oriented GUI Application Development. Prentice-Hall, 1993.
2. DARPA. Grand Challenge <www.darpa.mil/grandchallenge/>.
3. Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules" Communications of the ACM 15.12 Dec. 1972: 1053-1058.
4. United States Postal Service. "2005 U.S. Postal Service Annual Report" <www.usps.com>.
5. Eckel, B. Thinking in Java. Prentice-Hall, 2002.

About the Authors

Maj. Christopher Bohn, Ph.D., is a software engineering course director at the Air Force Institute of Technology's (AFIT) School of Systems and Logistics where he teaches a series of distance-learning short courses. Over the past 13 years, Bohn has served in various Air Force operational and research assignments, including time as an engineer for the Air Force Research Laboratory's Collaborative Enterprise Environment. He is an Institute of Electrical and Electronics Engineers-certified software development professional. Bohn has a bachelor's degree in electrical engineering from Purdue University, a master's degree in computer engineering from AFIT, and a doctorate from Ohio State University.

AFIT/LS Research Park Campus
3100 Research BLVD
Kettering, OH 45420-4022
Phone: (937) 255-7777 ext. 3415
DSN: 785-7777 ext. 3415
Fax: (937) 656-4654
DSN: 986-4654
E-mail: christopher.bohn@afit.edu

John Reisner is the Director of Extension Services at the Air Force Institute of Technology's (AFIT) School of Engineering and Management. He retired from the Air Force in 2005 after serving 20 years as a software developer, systems analyst, and instructor of Software Engineering. He is an Institute of Electrical and Electronics Engineers-certified software development professional and has a bachelor degree from the University of Lowell and a master's degree from AFIT.

Air Force Institute of Technology
2950 Hobson WY
Wright-Patterson AFB, OH
45433-7765
Phone: (937) 255-3636 x7422
DSN: 785-3636
E-mail: john.reisner@afit.edu

More Online from CrossTalk

CrossTalk is pleased to bring you this additional article
with full text at <www.stsc.hill.af.mil/crosstalk/2006/10/index.html>.

What Science Fiction Authors Got Wrong, and Why We're Better Off For It

Maj Christopher Bohn, Ph.D.
Air Force Institute of Technology

Computers are commonplace today, and they have changed much of the way we go about our lives, saving time and money. Surprisingly, as much as classical science fiction authors were visionaries, they did not foresee the arrival of the digital computer and its influence on modern life.

Robert Heinlein once had a discussion with a professional astronomer in which the conversation turned to Heinlein's attention to detail. Heinlein recounted how, when writing Space Cadet (1948), he and his wife used yards of butcher paper over three days to calculate a particular orbit; the results were described in the story with one line of text, but it was necessary to drive the drama. When the astronomer wondered why Heinlein did not use a computer to make the calculation, he replied, *My dear boy, this was 1947.*

Of course, computers are commonplace today, and they have changed much of the way we go about our lives, saving time and money. Surprisingly, as much as Heinlein and other Golden Age science fiction authors were visionaries – Heinlein, for example, is credited with inventing the waterbed and tele-operated manipulators (waldoes), and with heavily influencing spacesuit design; none of them foresaw the coming of the digital computer and its influence on modern life.

So while computers are everywhere today, Heinlein did not have ready access to computers and neither did his characters. In *Starman Jones* (1953), faster-than-light travel is possible by entering hyperspace singularities on precise vectors. Computing course corrections to get the spaceship on the correct vector required astrogators to make rapid calculations making use of mathematical tables and a computer that amounted to little more than a 4-function calculator. Heinlein, a former naval officer, no doubt was drawing an analogy with mathematical tables classically used to navigate seaships (Interestingly, Charles Babbage was inspired to develop a mechanical computer in the 19th century while reviewing errata for a set of mathematical tables for celestial navigation).