Understanding Object-Oriented Programming Concepts

Ray Klump, *Member, IEEE*

***Abstract*—Most power engineering students these days have had at least a casual introduction to computer programming concepts. Specifically, they have become functionally fluent in one or more programming languages and can use their knowledge of basic syntax to write simple programs that perform some desired task. Some students may also have had the opportunity to learn basic software design concepts, such as the benefits of top-down design and the practice and virtues of programming in a modular fashion using a structured programming approach. However, fewer students have been exposed to the theory and practice of object-oriented software development. Understanding object-oriented programming concepts requires that the student undergo a paradigm shift. The student must move from thinking about modeling systems and problems in terms of the *actions* that must be performed to thinking about the *objects* that must interact with each other to perform those actions. Successfully navigating this change in thought is not a trivial undertaking, but the rewards of doing so can be tremendous. This presentation provides a brief introduction to the concepts and benefits of the object-oriented approach and explains why power engineering students may benefit greatly from a more formal introduction to the topic.**

***Index Terms*—Object Oriented Programming, Object Oriented Methods, Software Reusability, Power System Modeling, Power Engineering Education**

I. INTRODUCTION

Object-oriented programming, or OOP, is a software development philosophy that has revolutionized the art and practice of writing computer applications. Regardless of the target discipline, programmers who work on sizable, multi-faceted software usually approach such problems from an OOP mindset. The OOP approach borrows the ideas of modularity, smooth flow of control, and procedure-based implementation of subtasks from its ideological and equally revolutionary predecessor, structured programming. However, OOP supplements these ideas with an emphasis on conceptualizing systems in terms of the components that must work together to achieve the required functionality and behaviors. In other words, whereas a structured programmer strives to implement a system by coding the system's tasks as a set of procedures and functions, the OOP developer implements a system by modeling its components. Modeling components means creating objects that exhibit certain characteristics and knowledge and perform specific tasks using those characteristics and knowledge. Expressed another way, OOP developers build systems from building blocks that possess both data and methods for working with that data. These building blocks are called objects.

Developers of technical software can benefit profoundly from using an OOP approach. They can create software that offers far greater flexibility and extensibility than what they could achieve using a purely procedure-based design. Moreover, they can incorporate components developed by other programmers to add functionality to their own projects, freeing them from having to implement features that have already been realized. In fact, OOP offers an unprecedented degree of software reusability, a tremendous advantage for those that must model large, diverse engineering systems.

Power system engineers are quite familiar with modeling large, diverse engineering systems. The number of different components that comprise a system, coupled with the great variety of models, attributes, and functions for those components, makes power system analysis a prime candidate for OOP-based software development. This argument holds for managing and performing calculations on power system data, but it is particularly relevant when it comes to presenting that data to the user. Effective communication of power system characteristics and trends to the user requires providing poignant visualization tools accessible through a convenient graphical user interface. The only practical way to fashion such technology is to take an OOP approach to software development. Thus, it is crucial that power engineering students be formally exposed to the ideas, implementation, and benefits of object-oriented design.

This discussion provides a "crash course" in these ideas in the hopes of spurring a more formal classroom presentation of the topic for power engineering students. It outlines the concepts that underlie OOP, identifies the primary advantages of the approach, and recommends ways to introduce OOP to power engineering students.

II. CORNERSTONES OF OBJECT-ORIENTED PROGRAMMING

The central ideas of OOP are encapsulation, inheritance, information hiding, data abstraction, and polymorphism. Each of these concepts helps give the object-oriented strategy considerable advantages over the more traditional structured programming approach.

Ray Klump is with PowerWorld Corporation, Urbana, IL 60544 USA (e-mail: ray@powerworld.com).

A. Encapsulation

Designing an object-oriented model involves defining a set of classes. A *class* is a template from which *objects* are created. The template, or blueprint, provided by a class specifies a set of *data* and *methods* that all objects created according to its specifications will contain. In other words, the class definition identifies the characteristics and behaviors of objects created from it. The term *encapsulation* simply describes the class's function of bundling data and methods to work with that data into a single data structure. This differs from the role of the familiar *record* data structure, which contains data fields but no prescriptions for retrieving, setting, or using the data.

Students often confuse the terms *class* and *object*. It is important to distinguish the class, which should be viewed as the template, blueprint, or prescription, from the object, which is the specific "widget" built according to that template, blueprint, or prescription. For example, **FordEscort** is a class of automobile; my1988GrayFordEscortThatNeedsNewTires is a specific *object* built according to that class blueprint.

A disciplined object-oriented approach involves developing a program by designing its component classes and specifying the manner in which those classes will interact. This strategy differs from the standard structured approach. The traditional structured approach views a system in terms of the tasks it performs and thus strives to code the functions and procedures that implement those tasks. The building blocks of such programs are its *functions*, and the programmer develops them by focusing on the *verbs* of the problem statement [1]. An object-oriented designer focuses on the types of *objects* that need to interact to perform a function. In other words, there is a heavy emphasis on the *nouns* of the problem statement, which the designer translates into the program's *classes*. Objects created according to these classes perform their tasks using the *methods* packaged within them. If one considers that the physical world consists not of independent actions but rather of *actors* that perform actions, it becomes clear that developing programs from an object-oriented perspective more closely parallels real physical systems. Software classes, like physical entities, encapsulate both data (characteristics) and methods (behaviors).

B. Inheritance

Classes can extend the definitions provided by other classes. Specifically, they can inherit the data and methods of another class. For example, suppose class B extends, or inherits from, class A. Then, class A is called the *superclass*, and class B is called the *subclass*. An object created according to the subclass definition has all the characteristics prescribed by the superclass definition, but it also can supplement these data with additional attributes unique to itself. Likewise, an instance of class B has all the methods, or behaviors, prescribed by class A, but it also can add its own unique behaviors. In other words, the class B definition includes both the methods of class A and additional methods used to perform different tasks. Furthermore, although the subclass inherits all the behaviors of the superclass, it can implement the behaviors in different ways. In other words, the methods of class B can *override* the same-named methods of class A to perform an identical task in a different way.

For example, consider a system for which the designer has defined a class called **Generator**. A generator is a machine that converts mechanical energy to electrical energy. However, a few different types, or classes, of electric power generators currently find use. For example, both combustion turbines and coal-fired units *are* generators in that they *inherit* the ability to generate electricity from mechanical energy. Suppose, then, that the developer defines two classes, **CombustionTurbine** and **CoalFired**, that extend, or inherit from, class *Generator*. Objects of all three classes might share attributes such as the amount of power they are generating, the maximum and minimum amount of power they can generate, their fuel and operating costs, physical location, and plant name. They may also each have a method called produceElectricity whose job is to provide an electrical output. However, the definition for class **CombustionTurbine**'s produceElectricity method will differ from that of class **CoalFired**, even though both inherit the ability to produce electricity from their common superclass, **Generator**. Both subclasses actually *are* types of generators, and so both know what it means to produce electricity. However, they do so in different ways. Again, this sort of relationship, in which types of objects exhibit similar sets of behaviors but act in different ways, very closely parallels physical reality.

C. Data Abstraction

A basic tenet of the structured programming approach is that programs should be constructed in a *top-down* manner. The top-down approach, also called *stepwise refinement*, suggests developing algorithms in a series of steps that move from a very general statement of the desired behavior to an increasingly specific view of the task's implementation details. The top-down approach also finds application in OOP circles, but in a slightly different guise. In developing inheritance hierarchies, object-oriented developers factor out the most rudimentary shared attributes and behaviors and ascribe them to a root superclass, which may be considered the ultimate superclass for the class hierarchy. This ultimate superclass provides the most abstract interpretation of what it means for an object to be of a certain category. In fact, most object-oriented languages allow programmers to define such superclasses explicitly as being *abstract*. One cannot create specific objects from abstract classes, because they are too vague to serve as a blueprint for something tangible. For example, consider the previous example involving the **FordEscort** class. Its superclass might be the more abstract class **Automobile**. Class **Automobile** is far too vague to create a definitive representation of it. However, **FordEscort**, which itself *is* (in other words, inherits data and methods from) an automobile, has a realizable blueprint from which one can create specific objects. Similarly, depending on the level of detail being targeted in developing the model, class **Generator** may be too vague to create an object from it, because it is unclear how a specific instance of **Generator** will produce electricity. **Generator** is thus an abstract class. More specific descendants of **Generator** define how they produce electricity and thus allow the programmer to create objects from them. This practice of creating a design from general definitions of common attributes and behaviors to more specific classes that add characteristics and specific implementations is called *data abstraction*. Data abstraction helps the developer categorize components of systems according to their form and function and thus unveils a composite picture of how the different actors in a system interrelate. Moreover, data abstraction helps a system take more complete advantage of one of the most powerful features of object-oriented design, polymorphism.

D. Polymorphism

Polymorphism may be considered the "great enabler" of OOP. Polymorphism enables programmers to manipulate subclass objects using superclass references. This advantage may seem insignificant until one considers that most physical systems consist of *collections* of objects. A power system, for example, consists of a collection of components that may, in a very general sense, be considered objects of some abstract class called **PowerSystemObject**. Looked at more closely, each object can be said to hail from a more specific subclass of **PowerSystemObject**. There are **Bus** objects, **Generator** objects, **TransmissionLine** objects, **Load** objects, and so on. Each of these subclasses may actually have their own subclasses descend from them. All **PowerSystemObject** instances share common attributes and methods. They each have an identifier by which system operators reference them. They each have a geographic location. They each have a status. They each can report their identifier, geographic location, and status to other objects that request such information from them, but the manner in which they do so likely will differ among different classes. For example, suppose the method that reports a **PowerSystemObject**'s identifier is called getIdentifier. When a **Bus** object communicates its status to another object, it might add the string "Bus" before its identifier, whereas a **Load** object might add the string "Load". Thus, the definition of getIdentifier for **Bus** objects differs from the definition of getIdentifier for **Load** objects. Interestingly, however, if one iterates through this collection using a generic **PowerSystemObject** reference and calls getIdentifier from the context of this reference, the computer will locate and invoke the version of getIdentifier that is specific to the actual type of the object being considered.

For example, consider this snipet of Java code that implements precisely the scenario described above:

```
PSObject psObj[] = new PSObject[2];
PSObject p;
Bus b = new Bus();
Load l = new Load();
psObj[0] = b;
psObj[1] = l;
for (int i = 0; i < psObj.length; i++)
  System.out.println(psObj[0].getIdentifier());
```

The lone instruction of the *for* loop will call a different version of getIdentifier during each of the two passes through the loop. When i = 0, psObj[i] is **Bus** object, and so the getIdentifier method of **Bus** will be called. However, when i = 1, psObj[i] is a **Load** object, which means that the getIdentifier method of **Load** will be called. The polymorphic nature of OOP languages provides this behavior automatically.

Without polymorphism, the programmer would have to employ *if* statements or *switch* logic to test the type of the currently active object. This alternative would clearly be difficult to maintain as the size and diversity of the modeled system changes. Polymorphism allows the developer to process objects of related classes in a very general way by focusing on class similarities. Polymorphism accomodates implementation differences by automatically dispatching commands to the appropriate methods at execution time. This allows code to be far more flexible and extensible.

E. Information Hiding

Classes encapsulate data and methods into a single data structure. Subclasses inherit these data and methods from superclasses, and subclasses and superclasses can thus be said to be related to each other. Suppose class A and class B have been defined to have entirely different inheritance trees; in other words, class A and B are not related to each other through inheritance. An object of class A can access the data and methods of class B, and vice-versa. However, it may be a good idea to limit such outside access. An object of class A may not know the proper way to set the value of a particular attribute of an object of class B, for example, and may thus try to set it to an improper value. For this reason, object-oriented designs usually prescribe class data to be kept private and to be accessed only through publicly accessible methods. The publicly accessible methods of a class establish the interface by which objects of a class interact with the outside world. Non-related objects have no access to the object's private fields and methods, but they can communicate using the public methods that comprise the interface.

This public-versus-private scope concept also mirrors physical reality quite well. Engineering analyses of interconnected systems tend to treat even very complex components as "black boxes" that connect to the rest of the system only at specific terminals. Information hiding provides object-oriented developers a particularly convenient way to connect "black boxes" into their own software systems. This makes the use of third-party software components, even those developed in a different programming language, both possible and safe. It also allows modelers of physical systems to substitute and test different models within the same software platform with minimal impact on peripheral code. Here again, the OOP approach graces software systems with tremendous extensibility and flexibility.

III. Advantages of Object-Oriented design

The previous section has already identified the key benefits of object-oriented software design. This section places these advantages into two categories: heightened extensibility through closer modeling of physical systems, and unprecedented code reusability.

A. Closer Modeling of the Physical World

Object-oriented development models systems in terms of *actors* rather than isolated *actions*. Newton's First Law of Motion predicts that objects at rest remain at rest unless *acted upon by another object*. Object-oriented design respects this behavior. Objects, which possess certain characteristics referred to as their *data*, use their set of behaviors, referred to as their *methods*, to act upon other objects, thus causing some change of state. In structured, function-centric programming, the action that causes the state change is like a *deus ex machina*, appearing without antecedent, introduced simply by virtual of the instruction stored in a computer register. Object-oriented software models clarify the initiator of the action and thus more clearly show the relationship between system components. This makes object-oriented models easier to understand, document, and maintain.

One of the human intellect's greatest gifts is the ability to summarize. A primary tool of summarization is the ability to categorize groups of related objects together according to their shared attributes and behaviors. Once categorized, same-classed objects can be collected together in some easily accessible location and used more conveniently to perform some task. OOP's principles of inheritance and polymorphism enable an identical process. Groups of similar classes are related through inheritance. They may be stored in a collection such as an array or vector and referenced using a generic handle that can refer to objects of all such classes. Each object stored in this collection knows how to perform its tasks in its own unique way and will do so reliably when called upon.

This observation begs a related point: those using this collection of objects need not worry about *how* each object does what it is designed to do. Each object was built from a class blueprint that completely specifies the details of its behavior. The object doesn't need to share those details with those who call upon it; it merely needs to provide a service. The entity requiring the service delegates the role to the object without worrying about the details. Object-oriented design models an efficient boss-employee relationship; the boss doesn't micro-manage the employee's actions, and the employee, eschewing even an executive summary, fulfills its responsibility and provides nothing more than the result it was programmed to perform.

OOP provides the tools to model the cogs that comprise the machine. When a better cog comes along, the developer simply swaps the parts and restarts the machine. Software models thus become extremely versatile and customizable.

B. Reuse of Code and Interoperability

The fact that one class (the *subclass*) can inherit data and methods from another class (the *superclass*) creates tremendous opportunity for code reuse. Some methods of the subclass may be identical to those of the superclass; thus, no new code need be written. When a subclass method does override a superclass method, it is common for the subclass method to use the superclass's implementation and simply embellish it by performing additional tasks. Again, rather than recode the superclass implementation, most object-oriented languages simply allow the subclass to reference the superclass's implementation as part of its own implementation. In a large project, this represents a tremendous savings in the quantity of code that must be written.

The concept of a class as describing a machine cog or black box also means that software authors need not work in isolation. They can draw upon the work of other class authors to borrow functionality for their own projects. A sort of cottage industry has developed around the idea that software should be constructed as a network of interconnected components that communicate with each other along publicly defined interfaces. These interfaces consist of the components' publicly defined data and methods and provide the mechanism by which they can be plugged into a system and communicate with that system's objects. This notion of software development is the impetus behind such technologies Common Object Request Broker Architecture (CORBA), Sun's Java Servlets and Java Beans, and Microsoft Corporation's Common Object Model (COM), Distributed Common Object Model (DCOM), and .NET technologies. Many of these technologies are network-aware, enabling object-oriented developers to import functionality that resides on remote application servers into their own applications in real time. The encapsulation of data and methods and the ability to craft well-defined interfaces through exposing only certain data and methods makes this interoperability of resources possible.

IV. Classroom Recommendations

Much power engineering research, particularly research that focuses on systems analysis, is performed on PCs and workstations rather than lab benches. Thus, both undergraduate and graduate power engineering courses tend to devote significant time to teaching algorithms and numerical methods. This training provides vital preparation for students who may be asked to write software modules to model a particular power or mechanical system later in their careers. Usually, the training is delivered in a language-independent manner and focuses instead on how the algorithms and methods perform a particular calculation. This is as it should be. To force a particular language on students as they attempt to understand how the implicit trapezoidal method integrates systems of differential equations would only obscure the important lessons.

The tremendous benefits of OOP – reusability, extensibility, and interoperability – should not go unnoticed by power engineering students. Although much legacy code exists that predates the advent of OOP, these systems will eventually be replaced by software employing more modern approaches. Moreover, even deeply entrenched elder applications are now being ported to new hardware, where modern software middlemen typically provide an object-oriented communication interface to time-tested functions. Thus, power engineering students would benefit greatly from a formal discussion of OOP concepts.

However, the same restraint that is exercised when discussing algorithms and numerical methods should be exercised when discussing OOP. Specifically, the lessons should not require students to use a specific language or technology. Besides obfuscating the important concepts of OOP, dictating a language choice tempts obsolescence. This is particularly true in the fast-paced, mercurial world of software technology.

Instead, students should be asked to develop a suite of classes that together model a particular physical system. For example, a course on power systems analysis might pose an assignment where students are asked to create a class framework for modeling loads, buses, generators, areas, lines, shunts, areas, and zones. Their design should start from an abstract base class that encapsulates the data and methods common to all power system elements. Each subclass should then supplement and customize the attributes and behaviors of their ancestor classes to implement their own specific tasks and relationships. They should also be given a simple exercise in polymorphism in which they use pseudocode to demonstrate how objects created from their class framework can be manipulated in a very general way using superclass references. Such lessons should help students navigate, as well as appreciate, the shift from function-centric to object-centric coding.

V. References

[1] H. M. Deitel and P. J. Deitel, *Java How To Program*, 3rd ed. New York: Prentice Hall, 1999, p. 326.

VI. Biography

Ray Klump (M'1991) received his Ph.D. from the University of Illinois at Urbana-Champaign in October, 2000. He worked as a power systems engineer and as a software developer at Mid-America Interconnected Network (MAIN) between 1996 and 1998. He has worked as a power systems consultant and software developer at PowerWorld Corporation in Urbana, IL since 1998. His research interests include voltage stability assessment, computational techniques, and data visualization. Dr. Klump also teaches college courses in computer science, mathematics, and electrical engineering and considers teaching one of his favorite pastimes.