From OOP to Nuts

Object-oriented programming techniques have been slow to become popular for embedded systems development. Maybe the problem isn’t OOP, but C++. Here we see how Ada helps make OOP highly effective.

A key part of successful object-oriented programming is knowing when not to use it. The examples provided by Thomas Niemann in his article “Nuts to OOP!” (August 1999, p. 16) were not proof that object-oriented programming (OOP) is flawed. Rather, they were perfect examples of when not to use objects.

For those who may not have read the original piece, let me take you back to an article that suggested the increasing use of object-oriented programming by embedded developers is unwarranted. To support his position, the author described an example C++ program in which a DEVICE class provided the interface to an unnamed hardware device with two 16-bit registers. Each of these registers was abstracted with a template class, called Register, that allowed various operations (=, ^, &, |, << and >>) to be performed upon them. The DEVICE class did nothing but give these two registers names.

As Michael Barr wrote in his counterpoint to the original article: “Abstractions are only useful when they hide something.” Mr. Niemann’s example hid nothing of the underlying behavior and, in addition, took just as many lines of code to control the “abstracted” device as it would have taken to control the device directly. The real purpose of his DEVICE abstraction should have been to get away from reading and writing registers. It’s the results of manipulating the registers that should be abstracted. In the process, the raw reads and writes will be hidden.

A proper example

For the last eight years, I have worked as an embedded software engineer, primarily developing software in Ada. Mr. Niemann says “Nuts to OOP!” but OOP isn’t really his problem. What he ought to say is “Nuts to C++!” Ignoring the weaknesses and overcomplications of his example for a moment, most of the issues that he describes as flaws of OOP are really problems with the C++ language.

Before responding to Mr. Niemann’s many criticisms of OOP, I’d like to show you a proper use of objects to abstract the functionality of a hardware device. We begin with the definition and implementation of a Register class in Ada95, which are shown together in Listing 1.

To create and use a register within your application code, you need only write:

```ada
My_Register : Register.Object;
Register.Create (My_Register, SomeAddress);--set the register
```
Over the years, I have found Ada code to be readable, reusable, and maintainable. Although these same code features can be achieved in other languages, Ada95 establishes a higher minimum standard in each of these areas.

The object-oriented principles of encapsulation, abstraction, modularity, and hierarchy lay the foundation for good programming. Ada95 takes advantage of these principles and enforces them:

- **Encapsulation.** Encapsulation is what keeps the application code from knowing or needing to know the implementation details of the class. This means that when you change an address or a way of doing things, you don’t have to change every piece of source code that depends on the subprograms that are in your interface. Encapsulation in the above example is implemented by declaring the type Light.Object as private. No client can then access the values in that record directly.

- **Abstraction.** When I woke up this morning, I tied my shoes. Long ago, I learned how to make my hands work together to tie my shoes. Before that, I learned how to make my shoes work at all. The idea that I don’t have to think about anything but tying my shoes is abstraction. In the above example, we can think of turning the LED on and off. We don’t need to think of setting values in a register.

- **Modularity.** Modularity is the ability to break a problem or its solution into smaller pieces. The classes in object-oriented programming languages allows us to do this. In Ada95, we specifically use packages and can go one step further and use child packages.

- **Hierarchy.** Hierarchy is the idea of arranging elements of a problem into a treelike structure. For example, a digital display output may be built using several LEDs. In this case, you could combine them...
together and pass characters to the digital display. This would in turn send the appropriate on/off commands to the LEDs on the display. These again could be combined to provide a multi-character display, to which you could pass a string as the argument. Hierarchy may also include inheritance, which involves adding functionality and state to a class and basing it on another class. For instance, we have an LED that can be turned on and off. We may want to manage an LED that can change colors. This would involve providing a method to change the color along with a value in the state to hold the color of the light. This colored light could simply inherit the rest of its behavior (on and off) from the Light class.

Myths and facts

Finally, I would like to take an opportunity to discuss some of the OOP myths and facts presented by Mr. Niemann. First, what were called myths:

- **Objects are needed to protect data.** In order to truly protect data in C++, you must use the class mechanism. In Ada, however, the declaration of a private type allows for the protection of data without a class. Unfortunately, in C++ the encapsulation mechanism and the inheritance mechanism are one and the same. If you would like to declare data as private, you must put that data inside a class. In Ada 95, the encapsulation can take place regardless of whether or not you use inheritance. Encapsulation is implemented in Ada95 by declaring types in the private portion of a package. Inheritance is implemented using **tagged** types. These mechanisms may be used individually or together to provide an appropriate solution.

- **Objects are needed to group data and procedures.** Again, if you are going to protect data and group it with pro-

---

**LISTING 1 A Register class in Ada95**

```ada
package Register is
    type Object is private;
    function Create (The_Address : in System.Address) return Object;
    procedure Write (This : in out Object;
                     The_Value : in Unsignedshort);
    procedure Read (This : in out Object;
                    The_Value : out Unsignedshort);
private
    type Object is record
        The_Location : System.Address;
    end record;
end Register;

package body Register is
    function Create (The_Address : in System.Address) return Object is
        This : Object;
    begin
        This.The_Location := The_Address;
        return This;
    end Create;

    procedure Write (This : in out Object;
                    The_Value : in Unsignedshort) is
        Register : Unsignedshort;
        for Register'Address use This.The_Location;
    begin
        Register := The_Value;
        end Write;

    procedure Read (This : in out Object;
                   The_Value : out Unsignedshort) is
        Register : Unsignedshort;
        for Register'Address use This.The_Location;
    begin
        The_Value := Register;
        end Read;
end Register;
```
In C++, you must use a class. You can, of course, put some procedures together with some data that they manipulate, but the only way to ensure that the data manipulation is always happening within the proper code is to use a class. In Ada95, you have several options. Data and procedures can be grouped with or without using the inheritance mechanism.

- **Objects are needed when implementing large programs.** The idea of implementing a very large system (in today’s terms) without taking advantage of encapsulation, abstraction, hierarchy, and modularity is absurd. The advantages you gain from using object-oriented technology are strong enough to consider their use.

- **Objects are easier to maintain.** I agree that if I were developing a small piece of code that had no other code depending on it, then making it a class would complicate matters. But if I have any code (other modules) that depends on the functionality I am producing, the entire effort is much easier to manage with object-oriented technology. When speaking of maintaining an individual class, yes the work is harder. But when it comes to maintaining an entire system, the work becomes easier.

- **The switch statement is bad.** Here, I agree with Mr. Niemann, there is nothing wrong with C++’s switch statement (or the corresponding “case” statement in Ada95). However, I’ve never even heard of such an OOP myth before. In fact, it should be clear that when developing real-time systems, you can always know how long a switch or a case statement will take in the worst case. If you are using dynamic run-time dispatching through inheritance, it is much harder to determine what will happen ahead of time.

According to Mr. Niemann, these are the facts about OOP:

- **Object-oriented programming requires more time, up front, doing design work.** We plan before we program. But it’s not OOP that requires more planning. It’s the whole idea of writing software correctly. Analysis and design are also supposed to be used in other programming paradigms, they just often aren’t.

- **Object-oriented programming requires more time to code a solution.** In describing why this is an issue, Mr. Niemann points out that it takes time to understand polymorphism and inheritance. Neither polymorphism nor inheritance are requirements for writing object-oriented programs. If a programmer designs an effective solution, the idea of polymorphism can be built in as appropriate. If you make every solution use inheritance and polymorphism it may become more complex. But why do that? I suggest using these features of the languages only when appropriate.

- **Object-oriented programming results in a more complex solution.** When Mr. Niemann recommends this, I need to ask what is meant by a more complex solution. Harder to use? Harder to understand? I’ve been around procedural programs enough to know that it would certainly become a much simpler world when changes that I make do not impact the entire application. Coding in an object-oriented language, whether it be Ada95 or any other, limits the impact that changes made to one part of the code will have on the rest of the system.

- **Object-oriented programming results in code that is more prone to error.** Mr. Niemann makes this claim based on the fact that he has proven object-oriented programming to be more complex with his example. If you are only talking about a 100-line program, it’s more complex to develop using classes in an object-oriented language. But on a grand scale, object-oriented programming reduces complexity and localizes and contains errors.

---

**LISTING 2** The definition of the Light class

```ada
with System;
with Opsys;
package Light is
  type Object is tagged private;
  function Create (Theaddress : System.Address;
                   Themask  : Opsys.Unsignedshort) return Object;
  procedure Turn_On (This : in out Object);
  procedure Turn_Off (This : in out Object);
private
  type Object is tagged
    record
      Address : System.Address;
      Mask   : Opsys.Unsignedshort;
      Is_On  : Boolean := False;
    end record;
end Light;
```
Object-oriented programming results in increased maintenance costs. Software is always hard to maintain. It is especially hard to understand and see the impacts of the code you are changing. At least in the object-oriented paradigm, you can reduce the coupling between modules, so that changes do not have such a dramatic impact on the entire application. Those who maintain legacy systems often take six months or more to come to a good understanding of even a small system. That’s a high maintenance cost that is independent of the programming paradigm. If you have built your system using object-oriented principles, at least the maintainers will be able to see the scope and potential impact of the changes they need to make before they make them.

C++ may be flawed and an object-oriented programming language. But does that mean we ought to abandon object-oriented programming altogether? Certainly not. There are many object-oriented programming languages to choose from. In particular, Ada95 does an excellent job of addressing the concerns of the embedded systems programming world, while maintaining an object-oriented capability. Let’s not give up on object oriented technology simply because we can’t easily encapsulate registers in C++.

Chad Bremmon works for Armstrong Consulting in the areas of object-oriented e-development and business process engineering. Chad was previously an Air Force Officer at the Pentagon for five years, after which he worked as a software engineering specialist for Rational Software. You can e-mail him at chadb@armstrongconsulting.com.

Resources
For more information about Ada, go to www.acm.org/sigada or www.adapower.com.

**Listing 3** The implementation of a Light class

```ada
with Opsys;
use type Opsys.Unsignedshort;

package body Light is

  function Create (Theaddress : System.Address;
                   Themask : Opsys.Unsignedshort)
    return Object is
    This : Object;
    begin
      This.Address := Theaddress;
      This.Mask := Themask;
      This.Is_On := False;
      return This;
    end Create;

  procedure Turn_Off (This : in out Object) is
    begin
      if This.Is_On then
        Register := Register xor This.Mask;
      end if;
      end Turn_Off;

  procedure Turn_On (This : in out Object) is
    begin
      if not This.Is_On then
        Register := Register xor This.Mask;
      end if;
      end Turn_On;

end Light;
```

**Listing 4** An example of using a Light class

```ada
with Light;
with System;
procedure Lightdriver is
  My_Light : Light.Object;

begin
  My_Light := Light.Create (16#FF5E#, 16#40#);
  Light.Turn_On (My_Light);
  Light.Turn_Off (My_Light);
end Lightdriver;
```