Comments on "Design patterns and Fortran 90/95" by Arjen Markus

Henry J. Gardner
Computer Science, FEIT, CECS
Australian National University
Canberra ACT 0200 Australia
(Henry.Gardner@anu.edu.au)

and

Viktor K. Decyk
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099 USA, and
Department of Physics and Astronomy
University of California, Los Angeles
Los Angeles, CA 90095-1547 USA

The article by Markus [1] discusses the subject of object-oriented (OO) design patterns and describes ways in which three well-known patterns might be implemented in Fortran 90/95. It appears that the idea of object-oriented design patterns in Fortran might just be coming of age and it is a very appropriate subject for discussion in a Forum such as this.

We have been developing our own implementations of design patterns in Fortran 90/95 for some time and have one paper under review [2] and another being presented at the 6th Workshop on Parallel/High-Performance Object Oriented Scientific Computing (POOSC) in Nantes, France in July 2006 [3]. Our application area is particle-in-cell simulations in plasma physics, but the results that we have obtained will, we hope, be of interest to Fortran 90/95 programmers in general.

Our own starting point with design patterns in Fortran 90/95 stems from the literature on object-based programming in that language [4-10]. Several authors have observed that the obvious unit of encapsulation, the Fortran 90/95 module, can be thought of as an object-based "class": it can support a type which defines the stored data together with subprograms which can operate on that data. With some discipline and ingenuity, these authors have shown how to emulate the notions of object construction and destruction, inheritance and polymorphism. These emulations were actually implemented using "object composition". For example, one "child" module might include another "parent" module in order to simulate inheritance. As Markus correctly points out, most of the famous design patterns are based on object composition and the authors of [4-10] have, frequently, been using design patterns as a way of simulating OO behaviour.

Because the subject of OO design patterns in Fortran 90/95 is so fresh, there is ample scope for them to be presented in different ways by different authors. Whilst appreciating the clarity and relevance of Markus' expositions, we would like to make some points of criticism which we believe are important to a further discussion of this topic. The points we make refer to the ideas expressed in the book by Gamma et al. [11] which is considered to be the fundamental exposition of the topic of OO design patterns.
Granularity and the Facade Pattern

Our first point is the granularity of the concerns involved with design patterns. If the fundamental unit of encapsulation is the Fortran 90/95 module, then design patterns relate to how a collection of modules interact to achieve some end. Their focus is on groups of modules or subsystems of modules. Their primary goal is the encapsulation and reuse of some clearly-defined activity in a broader software system. Thus, for example, the Facade pattern is concerned with providing well-defined entry points to a whole collection of modules and not just one module as in Markus’ example [1]. Furthermore, the Facade pattern is often used to restrict the possible linkages between a particular software subsystem and other subsystems. Thus, it is most common to find that ONE facade, or only a small number of facade modules, replaces a large number of possible module includes. In contrast, Markus’ example had one, library module which contained a large number of ode solvers and he then, potentially, employed a large number of facade modules, each of which was tasked with exposing a single entry point to this library module. In his example, each of the library subprograms is still available if the appropriate facade module is used. The example has one module with many facades rather than having many modules with one facade. It is an example of selective information-hiding rather than a true packaging of a subsystem with a well-defined entry point.

In our own work, we describe how the Facade pattern can be used to package up a whole Fortran90/95 program and reuse it as a subsystem. In [2] we take a plasma simulation code and explain how it can be used to model the plasma component of a much larger simulation - for example a simulation of a particle accelerator which includes plasma interactions. Instead of a main program, we create a module, and the declaration section becomes static data in that module. To complete the transformation, all of the code between the declaration and the main iteration loop becomes a constructor and the code inside the iteration loop becomes an update function. The main program now has encapsulated everything except the iteration loop:

```fortran
program facade
  use plasma_class
  integer :: i, nloop = 1
  call new_plasma()
  ! loop over number of time steps
  do i = 1, nloop
    call update_plasma()
  enddo
end program
```

After this step, the plasma component has been completely encapsulated. In a more realistic case, some information has to flow between the plasma component and the main program. For example, the main code might provide the particle data, and the plasma component might update the particle coordinates. In that case, one might add a type definition to the plasma_class which contains the data which must be communicated. The main program for the accelerator simulator might now look like:
program accelerator

use plasma_class
integer :: i, idimp = 6, npp = 32768, nloop = 1
real, dimension(:,,:), pointer :: part
type (plasma) :: plasma_component

allocate(part(idimp,npp))
call new_plasma(plasma_component,part)
! loop over number of time steps
do i = 1, nloop
    ! other processing by accelerator code can be done here.
    call update_plasma(plasma_component)
enddo
end program

Now the accelerator program controls the particle data. Additional information that needs to be passed back and forth between the main program and the plasma component can be added to the type one at a time. For example, the size of the grid (nx,ny,nz) can be input to the plasma component or a velocity-distribution diagnostic might be returned by the plasma component.

Object Composition and the Adapter Pattern

If the Facade pattern is the primary mechanism for encapsulating subsystems, then the Adapter pattern is the primary means of reusing them. The notion is that, often, supplied software components do not match exactly to what is expected by the client software. No problem! The Adapter pattern consists of a special adapter class (module) which implements the subprogram signatures expected by the client software by delegation to a supplied subsystem. In the case of the Adapter pattern, it can turn out that the supplied subsystem is as small as a single class or class hierarchy. In fact, library classes often have small cosmetic differences to those expected by client software and it is a straightforward exercise to construct an Adapter to bring the two together.

In [1], Markus provides an example which has two different implementations of a data structure but which both expose the same subprogram signatures to client classes and he says that this is an Adapter pattern. In fact, this is closer to being an emulation of the traditional object-oriented idea of an "interface". In OO circles, it is considered to be good form to "program to interfaces rather than implementations". In fact, this is "Rule Number 2" of OO design patterns according to Gamma et al [11]. ("Rule number 1" is to "favour object composition over inheritance".) There are many OO data-structure libraries which expose a single interface to client classes but which contain many implementation classes. The Java collections framework is one example of such a library.

Conclusion

We believe that design patterns offer great hope to Fortran 90/95 programmers and would urge others to participate in a fuller discussion of their possible implementations and benefits
References


