Object-Oriented Programming in Fortran95: Patterns and Process

by

Henry J. Gardner

Computer Science, FEIT
Australian National University
Canberra ACT 0200 Australia

and

Viktor K. Decyk

Department of Physics and Astronomy
University of California, Los Angeles
Los Angeles, CA 90095-1547 USA

Abstract

The Fortran95 programming language, which is widely used by computational scientists, can be used to emulate object-oriented programming constructs such as classes. We show, in the context of a real-world, computational-science example, how the language also lends itself to expressing some well-known, object-oriented design patterns. We have found these patterns to be genuinely useful in our application and contend that a wider dissemination of programming prescriptions such as these may have a significant influence on the way in which scientific software is constructed.
Introduction

We contend that the lack of acceptance of object-oriented programming amongst computational scientists is because the area has not been “pitched” in the appropriate way for the target audience. Computational scientists are smart, innovative people with a healthy workload and a hunger for success. They are used to wading through books on algorithms and optimization in order to write simulation software because they can see the benefit: their codes work and they run faster after all the effort. But the rewards for adopting object-oriented programming are often further down the track: when a code has become large and complex enough to warrant some serious attention to its organization. At these later stages, a computational scientist might be tempted to follow clear prescriptions for organizing, or refactoring, their software. In analogy with the situation for algorithms, one could imagine a computational scientist wading through books on possible software architectures to select one which would be appropriate for a particular application. Object-orientation has the potential to supply just these sorts of prescriptions but, at the present time, it does not do so in a context that computational scientist can easily understand.

Object-oriented design patterns provide guidance for the organization, maintenance and reuse of software. To many software professionals outside of computational science, this important field is represented by the seminal work by Gamma et a. [1]. But this book is, itself, not an easy read for an OO novice: it contains a large amount of distinctive terminology and some patterns are described with so many caveats that it can be difficult to find a clear programming prescription amongst them. Added to this, the principal languages considered, C++ and Smalltalk, are not languages of choice for computational scientists.

In this paper, we describe an attempt that we have made to try to implement some important design patterns in a language which is widely used by computational scientists, Fortran95. The patterns that we have been used have been found to be genuinely useful in the organization of a real-world, high-performance-computing application for particle simulation.

Objects and Design Patterns in Fortran95

Object-orientation, as it is commonly taught, involves a lengthy apprenticeship in the principles of encapsulation, inheritance and polymorphism and the mysterious way in which these principles are meant to work together to produce reusable software. Object-oriented design patterns are a reasonably advanced topic which involve some “unlearning” of these basic principles. In particular, one of the two basic principles of patterns is [1]:

“Favor composition over inheritance”

That is, patterns are primarily to do with the encapsulation and management of software subsystems and most of them have little to do with inheritance. Indeed, the major place of inheritance is through the use of interfaces to implement patterns as in the other fundamental principle:

“Program to interfaces rather than implementations”
In a recent paper [2] we set out to determine whether some of the classic patterns could be realistically implemented in Fortran95. Ultimately, our idea is that object-orientation can be pitched to computational scientists in a way similar to that shown in the figure: Scientists start out by solving problems in a problem-solving, procedural language such as Fortran95. As part of learning such a language, techniques for encapsulation are emphasized. For example, in Fortran95 the “module” construct can be used to group data close to functions and subroutines which operate on that data and this emulation of a class has been central to object-based programming in this language [3]. Thus, modules not only have the fundamental purpose of dividing up a code into easily-maintainable chunks, but they can also have the desirable effect of raising the level of abstraction when solving scientific problems.

As scientific codes start to get large enough, some believable prescriptions can be followed to progressively improve the software organization. These prescriptions will be adaptations of well-known design patterns in OO languages, but they are really object-based rather than object-oriented.

The application that we considered can be thought of as part of a Fortran95-based framework for supporting Particle-In-Cell (PIC) codes in plasma physics [4]. Such codes integrate, self-consistently, the trajectories of many millions of particles in the electromagnetic fields created by the particles themselves. The purpose of the frameworks is to provide easy to use, trusted, components that are useful for building different kinds of PIC codes. For example, the codes can be electrostatic or electromagnetic, they can solve for the full set of Maxwell's equations or a subset, and they can be tuned for different computer architectures including massively parallel supercomputers.
Patterns

In Reference [2] we motivate the introduction of patterns into our particle simulation program by imagining an evolutionary coding environment: A scientist starts out by writing an electrostatic simulation code. Later he or she chooses to extend this code by including electromagnetic particles and their consequent field contributions. In the next stage, varying boundary conditions are introduced and, finally, the entire code is wrapped up and used for a completely different application than the one which was first envisioned (as an accelerator-modeling code rather than a generic particle simulation code). These various modifications are implemented by following versions of the factory, abstract factory, strategy and facade design patterns.

The details of all of the pattern implementations are described in Reference [2]. In the following, we will briefly describe how these implementations work. Part of the implementation of the factory pattern is shown in the Appendix.

A factory pattern encapsulates the construction of one of several, related classes. In our example, we considered a situation where different particles in a particle simulation code respond to different kinds of forces. In the past, scientists would have created different versions of their code for different kinds of problems. The factory pattern allows one to build a single code which can handle various kinds of particles. The way in which it works can be seen in the Appendix. An initial es_particles_class, which also defines a particles type, is converted into an es_electromagnetic_class by the addition of new methods to define the particle push and charge deposit for electromagnetic particles (all of these methods delegate to legacy Fortran77 code). The generic particles_class module has links to both of the specific particles classes and encapsulates the decision-making as to which type of particle to create and manipulate. The main program, also shown in the Appendix, is entirely without any decision-making logic and the choice of which particle to create is only dependent on the value of a parameter ("emforce").

Suppose we wanted to add a third type of particle to model relativistic electromagnetic particles. Relativistic particles need a new component in the particles type, the speed of light, as well as a new push and current deposit subroutine. These two subroutines, as well as a new emforce value, RELATIVISTIC, would be incorporated into a new relativistic class. Two new lines would be added in the generic particles class to the push and current deposit subroutines to allow the selection of relativistic particles. In addition, the constructor would have a new optional argument, the speed of light. Except for the additional argument in the constructor, the main loop would not change at all.

The abstract-factory design pattern coordinates the creation of families of factories which need to work together. In order to illustrate this, we considered the field solvers needed for each particle type: Electrostatic particles need to solve a Poisson equation to obtain an electric field. Electromagnetic particles need to solve the full Maxwell equation for both the electric and magnetic fields. In the same way as the straight factory pattern, this variation of fields can be encapsulated in a field factory.
The strategy pattern is designed to encapsulate the choice of an algorithm and make algorithms interchangeable. It does so by separating the selection of the algorithm from the implementation. In our particle simulation example, we had a field factory which implemented various field solvers using periodic boundary conditions. It is often the case that other boundary conditions are needed. One example is the Dirichlet boundary condition, which physically represents a conductor and which might be implemented by making use of sine and cosine expansions.

We can create a class which will makes the choice for us as to which boundary conditions to impose using the strategy pattern. To do this, we add a new item, psolve, to the field type definition (in the es_field class which is not described in detail here):

```fortran
  type field
    integer :: emf, psolve
    integer :: nx, ny, nz
  end type
```

We can call this new class the fields_strategy class and we can include a subroutine which will determine which solver to call. In this case, Fortran95's renaming facility can be used to rename functions or subroutines when one module “uses” another so that name conflicts involving the “solve” subroutine can be avoided. When implementing the strategy pattern, the only change to the main code is the addition of a new constructor.

The facade pattern provides a simplified, high level interface to a subsystem. It is used when one needs to use only a part of a subsystem, or to access a subsystem in a special way. If only one instance of a subsystem will be needed, one can turn a scientific code into a subsystem in two steps. In Reference [2] we take as our example the code showing the particle factory. Instead of a main program, we create a module, and the declaration section becomes static data in the module. To complete this 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
```

In this first step, the plasma component is completely encapsulated. In a more realistic case, some information has to flow between the plasma component and the main program. For example, suppose the main code models high energy accelerators, and the authors wish to add a plasma component to it, perhaps due to mutual interactions between the particles. The main code might provide the particle data, and the plasma component might update the particle co-ordinates. 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:

```fortran
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 code 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.

**Conclusion**

We have summarized an approach that we have taken in Reference [2] to introduce design patterns into computational science software written in Fortran95. Although we have not been able to describe all of their details in this paper, the 4 patterns discussed have been genuinely useful in managing the architecture of a particle-in-cell plasma simulation code. We have also envisioned a simple, but realistic, process of software development whereby patterns are used to progressively refactor and enhance a code as requirements expand over time.

Although it is possible to step through most of the classic 23 patterns [1] and to come up with natural equivalents in Fortran95 the nature of the language is that some of these are “more natural” than others. Still, we have found the patterns described here to be genuinely useful in organizing a PIC framework. In the future we expect to report on implementations of other design patterns in Fortran95.
References:

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, [Addison-Wesley, Reading, MA, 1995].


Appendix: Generic Particles Class and Generic Main Program

Note that, in the following listings, the type definition for particles is contained in the es_particles_class. Both the es_particles_class and the em_particles_class call legacy Fortran77 code to perform the particle push and charge deposit.

```fortran
module particles_class
! generic particles factory class
!
use es_particles_class
use em_particles_class
implicit none
private
public :: ELECTROSTATIC, ELECTROMAGNETIC
public :: particles
public :: new_particles, initialize_particles,
charge_deposit
public :: current_deposit, push_particles, particle_manager
!
contains
!
subroutine new_particles(this,emforce,qm,qbm)
implicit none
type (particles) :: this
integer :: emforce
real :: qm, qbm
call new_es_particles(this,qm,qbm)
this%emforce = emforce
write (*,*) 'done new_particles'
end subroutine new_particles
!
subroutine push_particles(this,part,fxyz,bxyz,dt)
! advance particles in time
implicit none
type (particles) :: this
real, dimension(:,:,,:), pointer :: part
real, dimension(:,:,:,:), pointer :: fxyz, bxyz
real :: dt
select case(this%emforce)
case (ELECTROSTATIC)
 call es_push(this,part,fxyz,dt)
case (ELECTROMAGNETIC)
 call em_push(this,part,fxyz,bxyz,dt)
end select
write (*,*) 'done push_particles'
end subroutine push_particles
!
```

subroutine current_deposit(this,part,cu,dt)
! deposit current
  implicit none
  type (particles) :: this
  real, dimension(:,,:), pointer :: part
  real, dimension(:,,:,:,:), pointer :: cu
  real :: dt
  if (this%emforce==ELECTROMAGNETIC) then
    call em_current_deposit(this,part,cu,dt)
    write (*,*) 'done current_deposit'
  endif
end subroutine current_deposit

end module particles_class

program main
  ! main program for various kinds of particles
  use particles_class
  ...
  call new_particles(electrons,emforce,qm,qbm)
  ...
  ! loop over number of time steps
  do i = 1, nloop
    call current_deposit(electrons,part,current,dt)
    call charge_deposit(electrons,part,charge_density)
    ! omitted: solve for electrostatic or electromagnetic fields
    !
    call push_particles(electrons,part,efield,bfield,dt)
    call particle_manager(electrons,part)
  enddo
  ! end program