A Factory Pattern in Fortran 95

Viktor K. Decyk[1] and Henry J. Gardner[2]

[1] Department of Physics and Astronomy, University of California
Los Angeles, CA, 90095-1547, USA
decyk@physics.ucla.edu
[2] Computer Science, FEIT, CECS, Australian National University
Canberra, ACT 0200, Australia
Henry.Gardner@anu.edu.au

Abstract. This paper discusses the concept and application of software design patterns in Fortran 95-based scientific programming. An example of a factory pattern is given together with a context in Particle in Cell plasma simulation.

Keywords: Fortran, design patterns, object oriented, Particle in Cell.

1 Introduction

Object-oriented (OO) design patterns are an aspect of modern software engineering which is widely accepted as best practice in commercial software development. The seminal work on this subject is the book by Gamma, Helm, Johnson, and Vlissides[1], which identifies 23 recurring design patterns together with example contexts and a discussion of their strengths and disadvantages. Although many other design patterns texts have appeared subsequently most of these have had a focus on commercial applications and graphical user interfaces. In the scientific computing and parallel programming communities, there has been some discussion of design patterns for scientific applications in C++ and Java [1] as well as the recent book by Mattson et al[2]. But, until now, the treatment of Fortran seems to have been relegated to how legacy code might be integrated into frameworks written in other languages rather than how patterns might be used in Fortran itself. A discussion of Fortran 95 as a serious implementation language for design patterns has only just begun to appear[3,4].

Fortran 95 programs can be written in an "object-based" fashion using the `module` construct to define the fundamental object-oriented concept of a *class* [5,6,7,8,9,10]. But inheritance is not a feature of Fortran 95, so a fully OO approach is not possible. Instead module "composition", where one module has an association, or link, to another with the keyword `uses`, is used to emulate inheritance. According to Gamma et al.[1], one of the fundamental principles of OO design patterns is to "favor composition over inheritance" so this motivates

[1] For example, in conference series such as the International Symposium on Object-Oriented Computing in Parallel Environments (ISCOPE).

Y. Shi et al. (Eds.): ICCS 2007, Part I, LNCS 4487, pp. 583–590, 2007.

an exploration of how the essential nature of design patterns might be captured in programs written in Fortran 95.

In this paper, we describe one important pattern together with its computational science context: It has been used to build part of a large software framework for Particle in Cell (PIC) plasma simulation written in Fortran. (For comparison, the design and implementation of an object-oriented Particle in Cell simulation in Java is given in [11].)

2 An Object-Based Electrostatic Particle Simulation

Particle in Cell plasma codes[12] integrate the self-consistent equations of motion of a large number (up to billions!) of charged particles in electromagnetic fields. Their basic structure is to calculate the density of charge, and possibly current, on a fixed grid. Maxwell's equations, or a subset thereof, are solved on this grid and the forces on all particles are calculated using Newton's Law and the Lorentz force. Particle motion is advanced and new densities are calculated at the next time step.

It is a common practice for scientists to build a set of PIC models to study plasma phenomena at differing levels of complexity. At the basic level, an *electrostatic* code models particles that respond to Coulomb forces only. This is then extended to treat *electromagnetic* particles which correspond to both electric and magnetic fields. As the details of the physics are refined, the models can incorporate relativistic effects, differing boundary conditions, differing field solutions, multispecies plasmas and so on. A framework for building PIC models would allow all of these submodels to be generated and for common code to be maintained and reused between them.

We start by creating a Fortran 95 class for electrostatic particles (which respond to Coulomb forces only). This class uses the Fortran 95 `module` to wrap up and reuse legacy subroutines written in Fortran 77. It has the following structure: a type, followed by functions which operate on that type, and, perhaps, shared data[6]. The type declaration describes properties of particles, but it does not actually contain the particle position and velocity data which are stored elsewhere in normal Fortran arrays and are passed to the class in the subroutine argument "`part`". The type stores a particle's charge, `qm`, charge to mass ratio, `qbm`, and the number of particles of that type, `npp`:

```
   module es_particles_class
   type particles
      integer :: npp
      real :: qm, qbm
   end type
   contains
   subroutine new_es_particles(this,qm,qbm)
!  'this' is of type 'particles'
!  set this%npp, this%qm, this%qbm
   ...
   subroutine initialize_es_particles(this,part,idimp,npp)
```

```
! initialize particle positions and velocities
   ...
   subroutine charge_deposit(this,part,q)
! deposit particle charge onto mesh
   ...
   subroutine es_push(this,part,fxyz,dt)
! advance particles in time from forces
   ...
   subroutine particle_manager(this,part)
! handle boundary conditions
   ...
   end module es_particles_class
```

Most of the subroutines provide a simple interface to some legacy code. For example, the initialization subroutine assigns initial positions and velocities to the particle array, `part`:

```
   subroutine initialize_es_particles(this,part,idimp,npp)
! initialize positions and velocities
   implicit none
   type (particles) :: this
   real, dimension(:,:), pointer :: part
   integer :: idimp, npp
   allocate(part(idimp,npp))
   this%npp = npp
! call legacy initialization subroutine for part
   ...
```

The iteration loop in the main program consists of a charge deposit, a field solver, a particle push subroutine, and a boundary condition check:

```
   program es_main
! main program for electrostatic particles
   use es_particles_class
   implicit none
   integer :: i, idimp = 6, npp = 32768, nx = 32, ny = 32, nz = 32
   integer :: nloop = 1
   real :: qm = 1.0, qbm = 1.0, dt = 0.2
   type (particles) :: electrons
   real, dimension(:,:), pointer:: part
   real, dimension(:,:,:), pointer :: charge_density
   real, dimension(:,:,:,:), pointer :: efield
! initialization
   call new_es_particles(electrons,qm,qbm)
   call initialize_es_particles(electrons,part,idimp,npp)
   allocate(charge_density(nx,ny,nz),efield(3,nx,ny,nz))
! main loop over number of time steps
   do i = 1, nloop
      call charge_deposit(electrons,part,charge_density)
! omitted: solve for electrostatic fields
      call es_push(electrons,part,efield,dt)
```

```
      call particle_manager(electrons,part)
   enddo
!
   end program es_main
```

3 Extension to Electromagnetic Particles

Now let us consider particles which respond to both electric and magnetic forces. The push is different, and there is a current deposit in addition to the charge deposit. But the initialization, charge deposit, and particle manager are the same as in the electrostatic class and they can be reused. An electromagnetic particle class can be created by "using" the electrostatic class and adding the new subroutines as follows. (This "using" is an example of using object composition in a place where inheritance might be employed in an OO language.)

```
   module em_particles_class
   use es_particles_class
   contains
   subroutine em_current_deposit(this,part,cu,dt)
! deposit particle current onto mesh
   ...
   subroutine em_push(this,part,fxyz,bxyz,dt)
! advance particles in time from electromagnetic forces
   ...
   end module em_particles_class
```

A program where one could select which type of particles to use might first read a flag, `emforce`, and then use this flag to choose the appropriate routine to execute:

```
   program generic_main
   use es_particles_class
   use em_particles_class
   integer, parameter :: ELECTROSTATIC = 0, ELECTROMAGNETIC = 1
   ...
   call new_es_particles(electrons,qm,qbm)
   call initialize_es_particles(electrons,part,idimp,npp)
   allocate(charge_density(nx,ny,nz),efield(3,nx,ny,nz))
   if (emforce==ELECTROMAGNETIC) then
      allocate(current(3,nx,ny,nz),bfield(3,nx,ny,nz))
   endif
   do i = 1, nloop !loop over number of time steps
      if (emforce==ELECTROMAGNETIC) then
         call em_current_deposit(electrons,part,current,dt)
      endif
      call charge_deposit(electrons,part,charge_density)
! omitted: solve for electrostatic or electromagnetic fields
      select case (emforce)
      case (ELECTROSTATIC)
         call es_push(electrons,part,efield,dt)
      case (ELECTROMAGNETIC)
```

```
        call em_push(electrons,part,efield,bfield,dt)
    end select
    call particle_manager(electrons,part)
enddo
end program generic_main
```

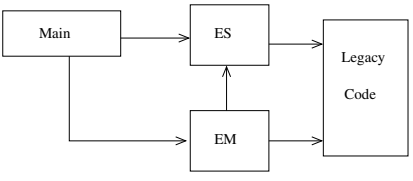

Fig. 1. An electrostatic/electromagnetic particle simulation which reuses `es_particles_class` (denoted by "ES")

The design of this program is shown schematically in Fig. 1 where, with exception of the legacy code, the boxes represent modules and the open arrows represent "uses" associations between modules. This design is disciplined and it reuses much of the previous, electrostatic, code, but the widespread use of `select case` or `if` statements can make the main program difficult to read and also necessitates keeping track of all the different choices if the code should be extended further.

4 A Factory Pattern

The essential idea of a factory pattern is to encapsulate "creational" logic inside a dedicated class, or collection of classes. We propose that a factory pattern can be incorporated into the above example by creating a "generic particle" class which will create storage for particles of the relevant type and will then ensure that the correct type of push and current deposit subroutines are chosen for a given particle type. This can be done by reusing almost all of the earlier version of the software without modification - save for the addition of the flag, `emforce`, into the `particles` type. The first part of this new class would read:

```
module particles_class
use es_particles_class
use em_particles_class
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
end subroutine new_particles
...
```

Within particles_class, the particle push routine looks like:

```
   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
```

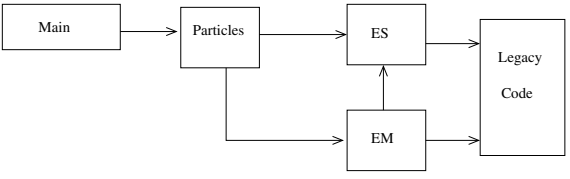

Fig. 2. Representation of the Fortran 95 factory pattern described in the text

The main loop of our refactored program now has the if and select case statements omitted and the decision making has been delegated to the particles_class module. The listing follows and the block diagram is shown in Fig. 2.

```
   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
```

How much work would it be to add a third type of particle? For example, suppose we wanted to create relativistic, electromagnetic particles. Relativistic

particles need a new component in the particles type, the speed of light, as well as a different push and current deposit subroutine. These two subroutines, as well as a new `emforce` value, `RELATIVISTIC`, would be incorporated into a new, *rel_particles_class* 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.

5 Discussion

Figure 3 shows a conventional, object-oriented factory pattern. A *client* has an association with an instance of a *factory* class which is responsible for creating an object from an inheritance hierarchy of target objects. The factory returns a handle on the desired object which, thereafter, calls methods directly on that object. The target object often implements an *interface* which is defined by the top of an inheritance hierarchy. The pattern shown in Fig. 2 differs from the conventional OO factory pattern because of the lack of inheritance, and the lack of conventional, OO interfaces, in Fortran 95. In our Fortran 95 pattern, the `particles` class is responsible for creating the object of the desired type and also for funneling calls to the correct subroutines for that particular type after it has been created. Still, the present pattern can be recommended to Fortran 95 programmers because it encapsulates and reuses significant portions of code and it manages these away from the main program logic.

In general, the rule for design patterns is to encapsulate what varies. We did this first by writing a general main program which had the `if` and `select case` statements explicit. We then encapsulated these statements inside a special class, `particles`. We have thus demonstrated a simple programming pattern together how it might be used in a believable process of iterative software development. In the complete framework, this process has been extended to model relativistic, multispecies plasmas with varying boundary conditions as well as with varying parallel-programming models.

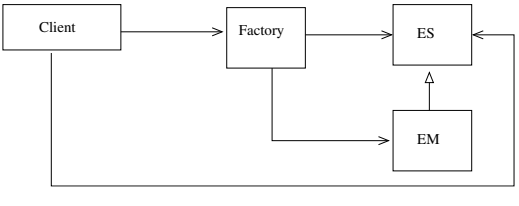

Fig. 3. Representation of a more conventional, object-oriented factory pattern

Acknowledgments

Viktor Decyk acknowledges the support of the US Department of Energy under the SCIDAC program. Henry Gardner acknowledges the support of the Australian Partnership for Advanced Computing (APAC) National Facility and the APAC Education, Outreach and Training program. Further details of this pattern and other patterns in Fortran 95, as well as an introduction to object-based programming in Fortran 95, will be available on the APAC-EOT website (`www.apac.edu.au`).

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley (1995) ISBN 0201633612.
2. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison-Wesley (2005) ISBN 0321228111.
3. Markus, A.: Design patterns and fortran 90/95. SIGPLAN Fortran Forum **25** (2006) 13–29
4. Gardner, H.J., Decyk, V.K.: Comments on the arjen markus article: Design patterns and fortran. SIGPLAN Fortran Forum **25** (2006) 8–11
5. Gray, M.G., Roberts, R.M.: Object-based programming in fortran 90. Computers in Physics **11** (1997) 355–361
6. Decyk, V.K., Norton, C.D., Szymanski, B.K.: How to express c++ concepts in fortran 90. Scientific Programming **6** (1997) 363–390
7. Decyk, V.K., Norton, C.D., Szymanski, B.K.: Expressing object-oriented concepts in Fortran 90. ACM Fortran Forum **16** (1997) 13–18
8. Cary, J.R., Shasharina, S.G., Cummings, J.C., Reynders, J.V., Hinkler, P.J.: Comparison of c++ and fortran 90 for object-oriented scientific programming. Computer Physics Communications **105** (1997) 20–36
9. Machiels, L., Deville, M.O.: Fortran 90: An entry to object-oriented programming for the solution of partial differential equations. ACM Transactions on Mathematical Software **23** (1997) 32–49
10. Decyk, V.K., Norton, C.D., Szymanski, B.K.: How to support inheritance and runtime polymorphism in fortran 90. Computer Physics Communications **115** (1998) 9–17
11. Markidis, S., Lapenta, G., VanderHeyden, W.: Parsek: Object oriented particle-in-cell implementation and performance issues. In: Proceedings of joint ACM-ISCOPE conference on Java Grande, Seattle, Washington, USA, 3-5 November, 2002, ACM, New York (2002) 141–147
12. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. Institute of Physics Publishing (1991) ISBN 0750301171.