Abstract

This paper discusses the concept, application, and usefulness of software design patterns in Fortran95 based scientific programming. Three patterns are discussed in detail, Factory (and Abstract Factory), Strategy, and Facade. It is concluded that such patterns are very useful in enabling increasingly complex scientific software development in Fortran95.
Introduction

As computers become more powerful, there is a growing desire to make scientific codes increasingly complex. In the computer science literature, researchers have discovered that similar solutions to programming complex problems have appeared in different contexts, and they have called such solutions “design patterns.” The seminal work by Gamma, Helm, Johnson, and Vlissides [1], identified 23 recurring design patterns. These patterns are intended to be language independent, and, indeed, a number of texts have appeared discussing their application to C++, Java, and other languages [2-3]. However, many of these patterns appeared in the context of graphical user interfaces and other non-scientific contexts, and are not obviously applicable to scientific programming. Furthermore, it is not obvious if they can be profitably used in non-object-oriented languages such as Fortran95. In this paper we will discuss several design patterns that have been useful for us in our goal of creating more ambitious Fortran95-based scientific programs. Some of the techniques used in implementing these design patterns are based on earlier work in emulating object-oriented concepts in Fortran95 [4-5]. In general, the patterns we have found most useful are those which encapsulate in one place, the variation and control of some important capability in the code.

Factory Pattern - Introduction

The Factory pattern encapsulates the construction of one of several, related classes.

As an example, we will consider a particle simulation code where different particles respond to different kinds of forces. In the past, scientists would have created different versions of their code for different kinds of problems. In this paper, we will discuss how the factory pattern allows one to build a single code which can handle various kinds of particles.

We will begin by creating a class for electrostatic particles, which respond to Coulomb forces only. The purpose of the class is to simplify handling variations, and its construction can be quite simple: it can be merely a front end to a complex legacy code. Classes in Fortran95 have the following structure: a type, followed by functions which operate on that type, and sometimes shared data [4]:

2
module es_particles_class

type particles
    integer :: npp
    real :: qm, qbm
end type

contains

! subroutine new_es_particles(this,qm,qbm)
! assign initial values to type
...
!
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

The type describes properties of particles, but does not actually contain particle
data, which are stored in normal (and familiar) Fortran95 arrays. In this example,
we store the particle’s charge (qm), charge to mass ratio (qbm), and the number of
valid particles in the array (npp), which might change in time. The constructor
of the class merely assigns initial values to the type, as follows:

subroutine new_es_particles(this,qm,qbm)
! assign initial values to type
implicit none
    type (particles) :: this
    real :: qm, qbm
    this%npp = 0
    this%qm = qm; this%qbm = qbm
end subroutine new_es_particles

Most of the functions provide a simple interface to some legacy code. For
example the initialization program 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
end subroutine initialize_es_particles

The push functions uses the electric field array (efield) and the time step to update the particle co-ordinates (also calling legacy code):

subroutine es_push(this,part,efield,dt)
! advance particles in time from electrostatic forces
  implicit none
  type (particles) :: this
  real, dimension(:,,:), pointer :: part
  real, dimension(:,,:,:,:), pointer :: efield
  real :: dt
! call legacy push subroutine
end subroutine es_push

The main iteration loop in the program consists of a charge deposit, a field solver, a particle push subroutine, and a boundary condition check. The main program then looks like the following.
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

Now let us consider electromagnetic 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. An electromagnetic particle class can be created by “using” the electrostatic class, which makes the electrostatic routines available to the electromagnetic class, and adding the new subroutines as follows (note that a new type is not needed for electromagnetic particles, since their properties are the same as for electrostatic particles):
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

The main iteration loop for the electromagnetic code looks like the following:

  program em_main
    ! main program for electromagnetic particles
    use em_particles_class
    ...
    ! loop over number of time steps
    do i = 1, nloop
      call em_current_deposit(electrons,part,current,dt)
      call charge_deposit(electrons,part,charge_density)
      ! omitted: solve for electromagnetic fields
      call em_push(electrons,part,efield,bfield,dt)
      call particle_manager(electrons,part)
    enddo

Without the use of the factory design pattern, a program where one could select at run time which type of particles to use, might first read an input variable which controls the type:

  integer :: emforce
  write (*,'(a)',advance='no') 'Enter emforce value: '
  read (*,*) emforce

Then one could use a select case or if-then-else construct to choose the appropriate routine to execute:
program generic_main
! main program for various kinds of particles
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
    !
    ! loop over number of time steps
    do i = 1, nloop
      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

The widespread use of select case or if statements makes the code more difficult to read and also imposes on the user of the code the necessity to keep track of all the choices when modifying the code. The factory design patterns assigns the responsibility for making the appropriate choices to its own class, so that the programmer need not be further concerned about it. The full listing of all these classes are in the Appendices A and B.
Factory Pattern - Implementation

The factory pattern is implemented by creating a new generic particle class called \texttt{particles\_class}, which will encapsulate the decision about which type of particle executes which type of subroutine. Before we can implement this, however, we must first make some minor adjustments to the previous classes. First, we add a component to the \texttt{particles} type in the \texttt{es\_particles\_class} called \texttt{emforce} to indicate type of particle, as well as a new parameter, \texttt{ELECTROSTATIC}, as follows:

```fortran
module es_particles_class

! type particles
  integer :: emforce, npp
  real :: qm, qbm
end type
!
integer, parameter :: ELECTROSTATIC = 0
!
contains

```

In the \texttt{em\_particles\_class} class, we just add another constant, \texttt{ELECTROMAGNETIC}:

```fortran
module em_particles_class

! integer, parameter :: ELECTROMAGNETIC = 1
!
contains

```

The new generic, \texttt{particles\_class} uses both previous classes, and adds a generic constructor which sets the \texttt{emforce} component:
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

In addition the new class adds two new subroutines that select which push routine
to use, and whether the current deposit is performed. The new push subroutine
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

The current deposit is similar. The new main loop now has the case statements
omitted, and the same program can be used with either type of particle. The
constructor stores the emforce parameter internally, and the new class will make
all the appropriate decisions.
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

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 the case statements inside a special class, particles.

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 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 full listing of all the factory pattern is in the Appendix C.
Abstract Factory Pattern

The abstract factory design pattern coordinates the creation of families of factories which need to work together. In order to illustrate this, we need to have at least one other factory. In the previous example, we created a factory to encapsulate the variation of different kinds of particles. We omitted the discussion of the field solvers. In fact, electrostatic and electromagnetic particles need different kinds of fields. 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 before, this variation of fields can also be encapsulated in a field factory.

The process is very similar to the creation of the particle factory. We have two kinds of fields, electrostatic (longitudinal) fields, and electromagnetic (transverse) fields, and we create classes for each of them. In the electrostatic field class, \texttt{es\_field}, we define a field type, which contains the dimensions of the fields, as well as a parameter, \texttt{emf}, which defines the kind of field we need:

\begin{verbatim}
type field
  integer :: emf
  integer :: nx, ny, nz
end type
\end{verbatim}

In addition, the \texttt{es\_field} class contains initialization functions, \texttt{ffts} and a \texttt{solve\_poisson} function. The electromagnetic field class, \texttt{em\_field}, contains addition and copy functions and an \texttt{update\_maxwell} function. Finally, a generic fields factory is created to encapsulate the type of decision about which type of field executes which type of subroutine. It contains a generic constructor which sets the \texttt{emf} component. It also contains a \texttt{solve\_efield} function to calculate the electrostatic field and an \texttt{update\_fields} function to calculate the electromagnetic fields. The \texttt{field\_factory} is listed in Appendices D, E, and F.

It is important that the correct field type be used for a given particle type. An abstract factory class can provide the proper co-ordination. As a minimum it can merely provide a function which calculates the proper field type for a given particle type, as follows:
function get_emfield(emforce) result(emf)
  integer :: emforce, emf
  select case(emforce)
    case (ELECTROSTATIC)
      emf = E_STATIC
    case (ELECTROMAGNETIC:RELATIVISTIC)
      emf = E_MAGNETIC
  end select
end function

The proper field type is then passed to the appropriate constructor. Alternatively, the abstract factory can encapsulate the constructors for both factories. The new main program, including field solver, now looks like the following:

! select appropriate field
  emf = get_emfield(emforce)
!
  ...
  call new_field(emfield,emf,nx,ny,nz)
  call initialize_scalar_field(emfield,charge_density)
  call initialize_fields(emfield,current,efield,bfield,exyz,bxyz)
!
! loop over number of time steps
  do i = 1, nloop
    call current_deposit(electrons,part,current,dt)
    call charge_deposit(electrons,part,charge_density)
    ! solve electrostatic field
    call solve_efield(emfield,charge_density,efield)
    ! solve electromagnetic field
    call update_fields(emfield,current,efield,bfield,exyz,bxyz)
    call push_particles(electrons,part,efield,bfield,dt)
    call particle_manager(electrons,part)
  enddo

The important feature is that the programmer no longer has to worry explicitly about making sure the correct fields are used with the correct particles.
Strategy Pattern

It is often the case that we use different algorithms in scientific computing in different situations, or we want to compare the behavior of different algorithms to help us decide on the best one. The strategy pattern is designed to encapsulate the choice of algorithm and make the 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. This boundary condition can be implemented by making use of sine and cosine expansions. For example, in this case the charge density can be represented as a Fourier sine series, as follows:

\[
\rho(x, y) = \sum \rho_{nm} \sin(n\pi x/L_x) \sin(m\pi y/L_y)
\]

The potential is then given by:

\[
\Phi(x, y) = \sum \Phi_{nm} \sin(n\pi x/L_x) \sin(m\pi y/L_y), \text{ where}
\]

\[
\Phi_{nm} = 4\pi \rho_{nm} / ((n\pi/L_x)^2 + (m\pi/L_y)^2)
\]

The electric field, found by differentiation of the potential, is then an expansion of various combinations of sines and cosines. We can implement such a solver by creating a d_field factory. It is very similar to the periodic field factory. The d_field factory would define new transforms to replace the fft, namely, fст, for a fast sine transform, and fсст and fсст for different combinations of fast sine and cosine transforms appropriate for the electric and magnetic field expansions, respectively. In addition, we need a new solve_d_poisson function and a new update_d_maxwell to replace the periodic versions of these functions. These functions are defined in a new d_es_field class and d_em_field class. The remaining functions defined in the periodic version of these classes are used unchanged. Finally, all these functions are used to define new solve_efield and update_fields functions in the new d_field factory.
If both field factories use the same names for the solvers, namely `solve_efield` and `update_fields`, then we could choose which solver to use by just changing the name of the `use` statement in the main program, either `use fields_class` for the periodic case or `use d_fields_class` for the Dirichlet case. This works because the two functions which differ in each factory have the same function names and arguments. This is useful if we do not wish to have both versions available at the same time, for example, to reduce the size of unused code. It is even possible to place the two different `use` statements into one line modules, and select which one to use by having two different targets in a make file.

Alternatively, we can create a class which will makes the choice for us, using the strategy pattern. To do this, we add a new item, `psolve`, to the fields type definition in the `es_field` class:

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

We will call this new class, the `fields_strategy` class. First of all it contains a new constructor for setting this component:

```fortran
module fields_strategy_class
  ! ...
  ! integer, parameter :: PERIODIC = 1, DIRICHLET = 2
  ! contains
  ! subroutine new_general_field(this,emf,psolve,nx,ny,nz)
  ! assign initial values to type
  implicit none
  type (field) :: this
  integer :: emf, psolve, nx, ny, nz
  call new_field(this,emf,nx,ny,nz)
  this%psolve = psolve
  end subroutine new_general_field

It should also contain a function which will choose the appropriate solver to call. However, we have a name conflict if the both factories use the same name for the solver. Fortunately, Fortran95 has a renaming facility we can use to locally
rename the functions when one module uses another. In the example here, we rename the `solve_efield` function in the `fields` class to `p_solve_efield`, and the `solve_efield` function in the `d_fields` class is renamed `d_solve_efield`:

```fortran
module fields_strategy_class
!
use fields_class, p_solve_efield => solve_efield, &
    p_update_fields => update_fields
use d_fields_class, d_solve_efield => solve_efield, &
    d_update_fields => update_fields

Then we can refer to the different names when we create a new `solve_efield` function which implements the strategy pattern:

```fortran
subroutine solve_efield(this,charge_density,efield)
! solve electrostatic field
implicit none
type (field) :: this
real, dimension(:,,:,:,:), pointer :: charge_density
real, dimension(:,,:,:,:), pointer :: efield
if (this%pserve==PERIODIC) then
    call p_solve_efield(this,charge_density,efield)
else if (this%pserve==DIRICHLET) then
    call d_solve_efield(this,charge_density,efield)
endif
end subroutine solve_efield
```

To implement the strategy pattern, the only change to the main code is the addition of a new constructor. The complete listing of the Strategy class in in Appendix G. Note that as in the factory pattern, we have encapsulated what varies (here the algorithm), and separated the decision making from the implementation.
Facade Pattern

One way that scientific computing is becoming more ambitious is by creating larger models by using previously written codes as subsystems or components. These subsystems generally provide very specific inputs and outputs to the larger model. 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. We will take as our example the code showing the particle factory. Instead of a main program, one creates a module, and the declaration section becomes static data in the module, as follows:

```fortran
module plasma_class
! facade interface to replace main program
use particles_class
implicit none
integer :: idimp = 6, npp = 32768, nx = 32, ny = 32, nz = 32
integer :: emforce = ELECTROMAGNETIC
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
save
!
contains
```

All of the code between the declaration and the main iteration loop becomes a constructor, as follows:

```fortran
subroutine new_plasma()
call new_particles(electrons,emforce,qm,qbm)
call initialize_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
end subroutine new_plasma
```

16
The code inside the iteration loop becomes an update function:

```fortran
subroutine update_plasma
  call current_deposit(electrons,part,current,dt)
  call charge_deposit(electrons,part,charge_density)
! omitted: solve for efield, bfield
  call push_particles(electrons,part,efield,bfield,dt)
  call particle_manager(electrons,part)
end subroutine update_plasma
```

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 and the main code it does nothing more than control the do loop. 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, as follows:

```fortran
type plasma
  real, dimension(:,,:), pointer:: part
end type
```

One moves the `part` array from the `plasma` class to the accelerator program, and the `new_plasma` function sets the `part` component of the `plasma` type to point to that array. All the functions in the class now have a `plasma` type argument to pass the `part` data to the individual subroutines. The allocation and initialization routines for the array `part` are removed from the `es_particles`
class, since the accelerator program is now performing that function. The main program now looks 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. The new plasma class is listed in Appendix H. 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.

In this example, we have assumed that there will be only one plasma component. If multiple plasma components are expected, one can encapsulate all the data in the plasma class in this plasma type, as follows:

```fortran
type plasma
  integer :: idimp, npp, nx, ny, nz, emforce
  real :: qm, qbm, dt
  type (particles) :: electrons
  real, dimension(:,,:), pointer:: part
  real, dimension(:,,:,,:), pointer :: charge_density
  real, dimension(:,,:,,:), pointer :: efield
end type
```

Multiple instances of the plasma can then be created. One might also wish to create a destructor function to deallocate plasma components. The facade pattern is a powerful way to build complex codes based on previously written codes.
Conclusion

In this paper we have tried to show the significance of several software design patterns for scientific programming. We have also tried to show how to implement them in Fortran95. Although Fortran95 is not an object-oriented language, most of the patterns can be implemented quite naturally. One reason for this is that most design patterns use object composition instead of inheritance. Since Fortran95 does not natively support inheritance, this is fortunate. The major difference between the implementation of these patterns in Fortran95 and in an object-oriented language is the explicit appearance of if-then-else constructs to implement polymorphism. However, by separating the polymorphism from the implementation, this becomes easily manageable. In fact, for many scientists, it is preferable to have such choices explicit and obvious than to have them implicit and hidden. Notice that we have implemented all the examples as skeleton code, which actually delegate work to some legacy code. This separation of object design from low level implementation allows one to rapidly design and redesign such patterns until the desired program is achieved.
References:

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


Acknowledgments

The work of V.K.D. was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. It was also supported in part by the U. S. Department of Energy, under the SCIDAC program.
module es_particles_class
!
! electrostatic particles
!
implicit none
!
private
!
public :: ELECTROSTATIC
!
public :: particles, new_es_particles, initialize_es_particles
!
public :: charge_deposit, es_push, particle_manager
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
subroutine es_push(this,part,efield,dt)
! advance particles in time from electrostatic forces
implicit none
  type (particles) :: this
  real, dimension(:,,:), pointer :: part
  real, dimension(:,,:,:,:), pointer :: efield
  real :: dt
write (*,*) 'call es_push'
end subroutine es_push

! subroutine particle_manager(this,part)
! handle boundary conditions
implicit none
  type (particles) :: this
  real, dimension(:,,:), pointer :: part
write (*,*) 'call particle_manager'
end subroutine particle_manager

! end module es_particles_class
Appendix B: Electromagnetic Particle Class

module em_particles_class
!
 electromagentic particles
!
  use es_particles_class
 implicit none
 private
 public :: ELECTROMAGNETIC
 public :: particles, new_es_particles, initialize_es_particles
 public :: charge_deposit, em_current_deposit, em_push
 public :: particle_manager
!
 integer, parameter :: ELECTROMAGNETIC = 1
!
 contains
!
 subroutine em_current_deposit(this,part,cu,dt)
 ! deposit particle current onto mesh
 implicit none
 type (particles) :: this
 real, dimension(:,,:), pointer :: part
 real, dimension(:,,:,:,:), pointer :: cu
 real :: dt
 write (*,*) 'call em_current_deposit'
 end subroutine em_current_deposit
!
 subroutine em_push(this,part,efield,bfield,dt)
 ! advance particles in time from electromagnetic forces
 implicit none
 type (particles) :: this
 real, dimension(:,,:), pointer :: part
 real, dimension(:,,:,:,:), pointer :: efield, bfield
 real :: dt
 write (*,*) 'call em_push'
 end subroutine em_push
!
 end module em_particles_class
Appendix C: Generic Particle Class

module particles_class
!
generic particles
!
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
module es_field_class
!
! electrostatic field
!
implicit none
private
public :: E_STATIC
public :: field, new_es_field
public :: initialize_scalar_field, initialize_vector_field
public :: scalar_fft, vector_fft, solve_poisson
!
  type field
     integer :: emf, psolve
     integer :: nx, ny, nz
  end type
!
  integer, parameter :: E_STATIC = 0
!
contains
!
subroutine new_es_field(this,nx,ny,nz)
!
! assign initial values to type
implicit none
  type (field) :: this
  integer :: nx, ny, nz
  this%emf = E_STATIC
  this%nx = nx; this%ny = ny; this%nz = nz
  write (*,*) 'calling new_es_field'
end subroutine new_es_field
!
subroutine initialize_scalar_field(this,q)
!
! allocate and initialize scalar field
implicit none
  type (field) :: this
  real, dimension(:,,:), pointer :: q
  allocate(q(this%nx,this%ny,this%nz))
  write (*,*), 'calling initialize_scalar_field'
end subroutine initialize_scalar_field
!
subroutine initialize_vector_field(this,fxyz)
!
! allocate and initialize scalar field
implicit none
  type (field) :: this
  real, dimension(:,,:,:), pointer :: fxyz
  allocate(fxyz(3,this%nx,this%ny,this%nz))
  write (*,*), 'calling initialize_vector_field'
end subroutine initialize_vector_field
! subroutine scalar_fft(this,q,isign)
! perform fft on scalar field
  implicit none
  type (field) :: this
  real, dimension(:,,:,:), pointer :: q
  integer :: isign
  write (*,*) 'calling scalar_fft'
end subroutine scalar_fft
!
subroutine vector_fft(this,fxyz,isign)
! perform fft on vector field
  implicit none
  type (field) :: this
  real, dimension(:,,:,:,:), pointer :: fxyz
  integer :: isign
  write (*,*) 'calling vector_fft'
end subroutine vector_fft
!
subroutine solve_poisson(this,q,fxyz)
! solve for longitudinal electric field
  implicit none
  type (field) :: this
  real, dimension(:,,:), pointer :: q
  real, dimension(:,,:,:,:), pointer :: fxyz
  write (*,*) 'calling solve_poisson'
end subroutine solve_poisson
!
end module es_field_class
Appendix E: Periodic Electromagnetic Field Class

module em_field_class
! electromagnetic field
!
use es_field_class
implicit none
private
public :: E_STATIC, E_MAGNETIC
public :: field, new_es_field
public :: initialize_scalar_field, initialize_vector_field
public :: scalar_fft, vector_fft, solve_poisson
public :: new_em_field, update_maxwell, add_fields, copy_fields
!
integer, parameter :: E_MAGNETIC = 1
!
contains
!
subroutine new_em_field(this,nx,ny,nz)
! assign initial values to type
implicit none
type (field) :: this
integer :: nx, ny, nz
call new_es_field(this,nx,ny,nz)
this%emf = E_MAGNETIC
write (*,*) "calling new_em_field"
end subroutine new_em_field
!
subroutine update_maxwell(this,exyz,bxyz)
! solve maxwell's equation for transverse efield, bfield
implicit none
type (field) :: this
real, dimension(:,:,:,:), pointer :: exyz, bxyz
write (*,*) 'calling update_maxwell'
end subroutine update_maxwell
!
subroutine add_fields(this,fxyz,exyz)
! add longitudinal and transverse fields
implicit none
type (field) :: this
real, dimension(:,:,:,:), pointer :: fxyz, exyz
write (*,*) 'calling add_fields'
end subroutine add_fields
!
subroutine copy_fields(this,fxyz,bxyz)
! copy transverse fields
implicit none
type (field) :: this
real, dimension(:,:,:,:), pointer :: fxyz, bxyz
write (*,*) 'calling copy_fields'
end subroutine copy_fields
end module em_field_class
Appendix F: Generic Field Class

! module fields_class
! abstract fields
!
use es_field_class
use em_field_class
implicit none
private
public :: E_STATIC, E_MAGNETIC
public :: field, new_field, initialize_scalar_field
public :: initialize_fields, solve_efield, update_fields
!
contains
!
subroutine new_field(this,emf,nx,ny,nz)
! assign initial values to type
implicit none
type (field) :: this
integer :: emf, nx, ny, nz
call new_es_field(this,nx,ny,nz)
this%emf = emf
write (*,*) 'done new_field'
end subroutine new_field
!
subroutine initialize_fields(this,cu,efield,bfield,exyz,bxyz)
! initialize electromagnetic fields
implicit none
type (field) :: this
real, dimension(:,:,:,:), pointer :: cu,efield,bfield,exyz,bxyz
call initialize_vector_field(this,efield)
select case (this%emf)
case (E_MAGNETIC)
call initialize_vector_field(this,cu)
call initialize_vector_field(this,bfield)
call initialize_vector_field(this,exyz)
call initialize_vector_field(this,bxyz)
case default
nullify(cu,bfield,exyz,bxyz)
end select
write (*,*) 'done initialize_fields'
end subroutine initialize_fields
subroutine solve_efield(this, charge_density, efield)
! solve electrostatic field
implicit none
type (field) :: this
real, dimension(:,:,:,:), pointer :: charge_density
real, dimension(:,:,:,:), pointer :: efield
call scalar_fft(this,charge_density,-1)
call solve_poisson(this,charge_density,efield)
write (*,*) 'done solve_efield'
end subroutine solve_efield

subroutine update_fields(this, cu, efield, bfield, exyz, bxyz)
! solve electromagnetic field
implicit none
type (field) :: this
real, dimension(:,:,:,:), pointer :: cu, efield, bfield, exyz, bxyz
select case (this%emf)
case (E_MAGNETIC)
call vector_fft(this, cu, -1)
call update_maxwell(this, exyz, bxyz)
call add_fields(this, efield, exyz)
call copy_fields(this, bfield, bxyz)
call vector_fft(this, bfield, 1)
end select
call vector_fft(this, efield, 1)
write (*,*) 'done update_fields'
end subroutine update_fields

end module fields_class
Appendix G: Field Strategy Class

module fields_strategy_class
! fields with various boundaries
!
  use fields_class, p_solve_efield => solve_efield, &
  &p_update_fields => update_fields
  use d_fields_class, d_solve_efield => solve_efield, &
  &d_update_fields => update_fields
  implicit none
private
public :: E_STATIC, E_MAGNETIC, PERIODIC, DIRICHLET
public :: initialize_scalar_field, initialize_vector_field
public :: field, new_field, initialize_fields
public :: new_general_field, solve_efield, update_fields
!
  integer, parameter :: PERIODIC = 1, DIRICHLET = 2
!
contains
!
  subroutine new_general_field(this,emf,psolve,nx,ny,nz)
  ! assign initial values to type
  implicit none
  type (field) :: this
  integer :: emf, psolve, nx, ny, nz
  call new_field(this,emf,nx,ny,nz)
  this%psolve = psolve
  end subroutine new_general_field
!
  subroutine solve_efield(this,charge_density,efield)
  ! solve electrostatic field
  implicit none
  type (field) :: this
  real, dimension(:,:,,:), pointer :: charge_density
  real, dimension(:,:,,:), pointer :: efield
  if (this%psolve==PERIODIC) then
    call p_solve_efield(this,charge_density,efield)
  else if (this%psolve==DIRICHLET) then
    call d_solve_efield(this,charge_density,efield)
  endif
  end subroutine solve_efield
subroutine update_fields(this, cu, efield, bfield, exyz, bxyz)
! solve electromagnetic field
implicit none

! type (field) :: this
real, dimension(:,:,:,:), pointer :: cu, efield, bfield, exyz, bxyz

if (this%psolve==PERIODIC) then
  call p_update_fields(this, cu, efield, bfield, exyz, bxyz)
else if (this%psolve==DIRICHLET) then
  call d_update_fields(this, cu, efield, bfield, exyz, bxyz)
endif
end subroutine update_fields

end module fields_strategy_class
Appendix H: Facade Particle Class

module plasma_class
! facade interface to replace main program
use particles_class
implicit none
!
type plasma
  real, dimension(:,:,), pointer:: part
end type
!
integer :: nx = 32, ny = 32, nz = 32
integer :: emforce = ELECTROMAGNETIC
real :: qm = 1.0, qbm = 1.0, dt = 0.2
type (particles) :: electrons
real, dimension(:,:,,:), pointer :: charge_density
real, dimension(:,:,,:), pointer :: efield, bfield, current
!
contains
!
subroutine new_plasma(this,part)
type (plasma) :: this
real, dimension(:,:,), pointer :: part
call new_particles(electrons,emforce,qm,qbm)
this%part => part
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
end subroutine new_plasma
!
subroutine update_plasma(this)
type (plasma) :: this
call current_deposit(electrons,this%part,current,dt)
call charge_deposit(electrons,this%part,charge_density)
! omitted: solve for efield, bfield
call push_particles(electrons,this%part,efield,bfield,dt)
call particle_manager(electrons,this%part)
end subroutine update_plasma
!
end module plasma_class