Object-Oriented Design Patterns in Fortran 90/95

Viktor K. Decyk[a] and Henry J. Gardner[b]

[a] *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
email: decyk@physics.ucla.edu

[b] *Department of Computer Science, College of Engineering and Computer Science, Australian National University, Canberra, ACT 0200, Australia*
email: Henry.Gardner@anu.edu.au

Abstract

This paper discusses the concept, application, and usefulness of software design patterns for scientific programming in Fortran 90/95. An example from the discipline of object-oriented design patterns, that of a game based on navigation through a maze, is used to describe how some important patterns can be implemented in Fortran 90/95 and how the progressive introduction of design patterns can usefully restructure Fortran software as it evolves. This example is complemented by a discussion of how design patterns have been used in a real-life simulation of Particle-in-Cell plasma physics. The following patterns are mentioned in this paper: Factory, Strategy, Template, Abstract Factory and Facade.

Key words:
object-oriented, design patterns, Fortran
PACS: 89.80, 07.05.Wr

---

1 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 (see, for example, [2,3]). But the treatment of design patterns in Fortran 90/95 is quite lacking by comparison to these languages.

Fortran 90/95 is a non-object-oriented language but it contains a language construct, the `module`, which can be used to emulate "classes". The resulting "object-based" style of programming has been recommended by some authors [4–8] as a way of enhancing the *encapsulation* and *reuse* of Fortran 90/95 code "components".

In the following two sections we review the basic concepts behind object-based programming in Fortran 90/95 and introduce the running example of the maze game. This example is used to illustrate the implementation of three important design patterns, Template, Strategy and Factory, in Secs. 4 and 5. The way in which a design patterns approach has been used to construct a framework for particle-in-cell simulation is described in Sec. 6 and the paper concludes with Sec. 7.

2 Classes in Fortran 90/95

In the following sections we will illustrate how some object-oriented (OO) design patterns can be usefully emulated in Fortran 90/95. As a vehicle for this discussion, we will use a running example which has been adapted from [1]. This is a game program which creates a two-dimensional maze through which a player needs to navigate in order to find an exit. The various versions of the software that we discuss will be available through the CPC software archive.

In our discussions, we will refer to listings of sample Fortran 90/95 code as well as to some simple diagrams. The first example, in Listing 1 and Fig. 1, is an abridged Fortran 90/95 *class* which models the concept of a room in our maze. The class defines a type, `Room`, which contains an integer field for the room number and an array of logical variables which determine whether this room

has walls in the north, south, east and west directions. The class also defines 4 constants to denote these directions of the compass. It has a *constructor* which initializes the values of variables inside the `Room` type (and which would also allocate space for allocatable variables). Good object-oriented programming practice recommends that data be kept private and that public procedures be written to access or modify that data: the fields of the `Room` type are *private* but the procedures used to access or to modify those fields are *public*. (In performance-critical procedures, direct data access is sometimes favored for reasons of program optimization. We will comment further on information hiding between classes throughout this paper.)

The emulation of classes in Fortran 90/95 has been discussed in [4–8]. Classes are meant to define a type together with procedures which operate on data of that type. Objects of that type are constructed using constructor procedures and destructed (deallocated), if necessary, by destructor procedures. In Fortran 90/95, the selection of the correct procedure for a given type needs to be made by passing an object of the required type. By convention, we give this object the name `this` and make it the first argument in the argument list.

Listing 1. A class describing a room in a maze

```
module Room_class
  implicit none
  public
  ! Room -- definition
  type Room
     private
     ! room number
     integer :: roomNumber
     ! walls in directions NSEW?
     logical, dimension(4) :: walls
  end type Room
  ! parameters
  integer, parameter :: NORTH=1, SOUTH=2, EAST=3, WEST=4
contains
  ! - constructor
  subroutine new_Room(this, roomNumber, N, S, E, W)
    type(Room) :: this
    integer, intent(in) :: roomNumber
    logical, intent(in) :: N, S, E, W
    ! set the values of the Room type
    ...
    ! (Call random_number to find walls.
    ! Either a wall is read in as true or
    ! a coin is flipped to see if it will be true
    ! or false.)
    ...
```

```
  end subroutine new_Room

  function getRoomNumber(this) result(res)
    ! accessor procedure for private data
    type(Room) :: this
    integer :: res
    res = this%roomnumber
  end function getRoomNumber

  function getWalls(this) result(wallList)
    ! accessor procedure for private data
    type(Room), intent(in) :: this
    logical, dimension(4) :: wallList
    ! return current list
    wallList = this%walls
  end function getWalls
end module Room_class
```

Software engineers use diagrammatic notations, the most famous of which is the Unified Modeling Language [9]. We shall use a simplified version of one of these UML notations, called `class diagrams`, to illustrate the various versions of our maze game and the use of patterns in it. UML represents classes as boxes with compartments which list the type components, other module data (for example, static data and constants) and module procedures. A UML diagram for the `Room_class` might look like Fig. 1.

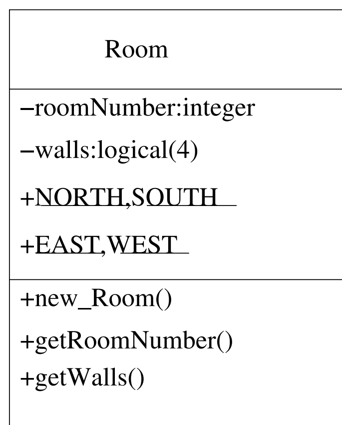

Fig. 1. UML diagram for the `Room_class`. Private data and methods are flagged with "-". Public data and methods are flagged with "+". Static data and constants are underlined. Where shown, data types are given after the variable names. Full argument lists for procedures are not shown.

3 The Maze Game: `mazev1`

In this section we present our simple programming example as if it were a larger software development project.

Imagine that you set out to write a Fortran 90/95 program for a maze game. Good software engineering practice would dictate that you would start this exercise by defining the *requirements* of your system, that you would follow this phase by an *analysis* of these requirements before moving on to *design* and, subsequently, *implement* and *test* your software. Standard software process methodologies recognize that there are feedback loops between (at least adjacent) development phases.

The *requirements* for our system might read as follows:

(1) The program will create two-dimensional maze structures of arbitrary row and column dimensions.
(2) Each room in the maze can have between zero and four walls. Movement between rooms can be made providing it is unimpeded by a wall.
(3) A player starts in a particular room and needs to find the exit room.
(4) A player can quit at any time, and lose the game, or can find the exit room and win the game.

An *analysis* of these requirements would flesh out important issues and it could specify a sequence of steps which the program would walk through from the perspective of a player (called a *use case*). As an example, Requirement 4 raises the issue of whether it is always possible for a player to find the exit room or whether this might be blocked by walls. Another example is that Requirement 2 raises the issue of whether walls can be one-way (when moving from room `i` to room `j` but not when moving from room `j` to room `i`).

Procedural software design can start by taking the requirements and use cases and expressing these as pseudocode or flow-charts and progressively refining these models until coding of the algorithms can begin. Object-oriented design starts by considering the data-structures needed in the software and then attempts to locate the procedures in the same classes as the data on which they operate. There are two major data entities in our example which can be candidates for classes:

- the maze
- rooms in the maze

Our requirements also mention another data entity, walls. But walls are most properly modelled as fields within rooms rather than classes in their own right. Discovering classes and allocating entities to fields within these classes is part

of the art of object-oriented software analysis and design.

In the previous section, we gave an example of how a room class would look for this simple maze example. The `Maze_class` would contain a data-structure made up of rooms and it would enable movement from room to room. Rooms might be labeled by numbers. One of the rooms might be the "exit" room and another might be the "current" room in a particular game. This class may also return the array of walls about the current room. Figure 2 shows a UML diagram for such a `Maze_class` module.

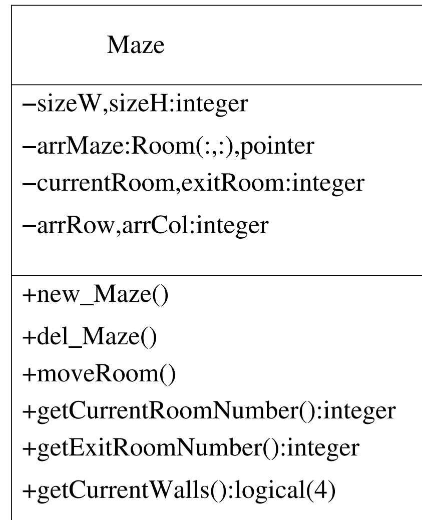

Fig. 2. UML diagram for the `Maze_class`.

An interesting question is where to locate the *game logic* of our program. The procedures making up the game logic will handle user input and oversee the movement between rooms during the course of the game. It is most clearly associated with the data entity of the maze, but it could be decoupled from the `Maze_class` and placed in a `MazeGame_class` of its own. We will assume that our designer has chosen to do this, and we will subsequently show that this was a wise decision to make. Although it involves some extra typing at this stage, we have chosen to define a `MazeGame` type which contains only a `Maze` data element. The UML for this class is shown in Fig. 3.

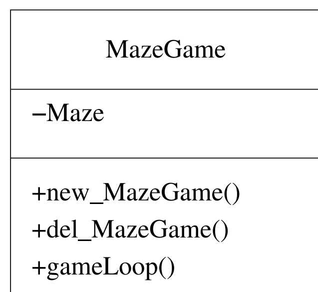

Fig. 3. UML diagram for the `MazeGame_class`.

Our system just needs one other class to initialize the maze and to repeatedly call the game loop until a termination condition is obtained. This *client* class is the main program shown in Listing 2.

Listing 2. The main program for the maze game

```
program MazeGame_Test
  use MazeGame_class
  implicit none
  type(MazeGame) :: game
  logical :: bEndGame=.false., bWonGame
  call new_MazeGame(game, 3, 4)
  do while (bEndGame .eqv. .false.)
     bEndGame = gameLoop(game, bWonGame)
  end do
  print *, 'Game Over!'
  if (bWonGame .eqv. .true.) then
     print *, 'Congratulations, you won!'
  else
     print *, 'Bad luck, you lost!'
  end if
  call del_MazeGame(game)
end program MazeGame_Test
```

A full UML class diagram for the system has open arrows between classes which are *associated* with one another by virtue of having a Fortran 90/95 `use` statement. For example, our main program `MazeGame_Test`, has an association with `MazeGame_class` which can be seen from Fig. 4 (as well as from Listing 2). The "multiplicity indicators" on the classes `Maze_class` and `Room_class` represent how many objects of one class type on one side of the association arrow will be associated with one object of the type on the other side. By default, no multiplicity indicators implies a one-to-one association.

The implication of an association chain, such as that shown between `MazeGame_Test` and `Room` in Fig. 4, is that all public entities of `Room` are available to `MazeGame_Test`. However, the global access modifiers, and the information-hiding and renaming facilities provided by the Fortran 90/95 `use` statement, complicate this picture. For example the two statements near the beginning of `MazeGame_class`:

```
  private
  public :: MazeGame, new_MazeGame, del_MazeGame, gameLoop
```

imply that only those public entities which are explicitly listed in this statement can be accessed from `MazeGame_Test`. Specifically, the only access by `MazeGame_Test` to entities in `Maze_class` and `Room_class` are those controlled by `MazeGame_class`; by-passing `MazeGame_class` to directly manipulate mazes

and rooms is not allowed. We have represented this "privatizing" by placing the symbol "#" (traditionally the "protected" symbol in UML) above the association link of Fig. 4. Fortran 90/95 is very good at privatizing things and this form of information hiding is a very good thing for object-oriented software.

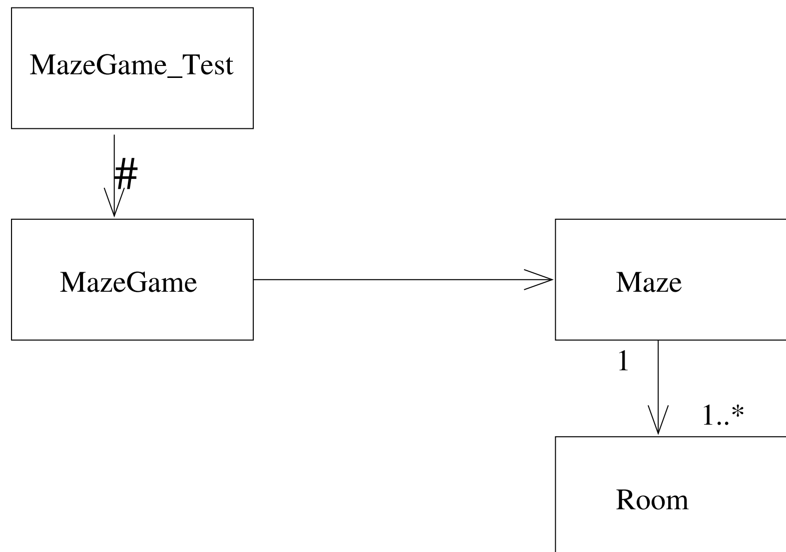

Fig. 4. Class diagram for `mazev1`.

4 The Template and Strategy Patterns: `mazev2`

With all of the attention to encapsulation and information hiding described in the previous section, our `mazev1` program represents good software engineering practice for the implementation of a single maze game. Design patterns will now be introduced for the first time to enable the program structure to be generalized to support different maze games.

Suppose we wanted to have several options for an algorithm such as the game logic of the maze game. The *Template* design pattern is used to encapsulate the invariant part of a particular algorithm in one class and varying parts in other classes. The *Strategy* design pattern controls the switching between the varying parts of the algorithm. These two patterns are commonly used together.

In the maze game the game logic can been split into two parts. The first part contains the main loop which solicits input from a user ("Where do you want to go?") and determines the status of the game (whether the exit room has been found). The actual movement within the maze might vary with the particular game logic and this can be encapsulated in a procedure, `nextMove()`, which is implemented in different ways in different classes. In this version of the

software, we illustrate this by implementing the original game logic as well as a new game which does not have walls about the rooms. The Template pattern is illustrated in Fig. 5 which shows the splitting of the game loop between the original MazeGame_class and the new StandardGameComponent_class.

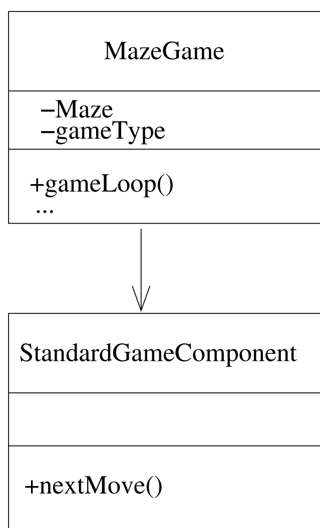

Fig. 5. Illustration of a Template pattern for the maze game logic.

We implement the switching between different movement strategies in the MazeGame_class. A flag, gameType, needs to be added to the MazeGame type and a case statement needs to be added to the gameLoop() procedure to select the appropriate version of nextMove(). The flag will be passed through from the main program in a modified MazeGame_class constructor. Associations will be maintained between MazeGame_class and the two component classes (StandardGameComponent_class and NoWallsGameComponent_class). Because the procedure name nextMove is used in each of the component classes, the Fortran 90/95 renaming facility is used to avoid name conflicts within MazeGame_class. Listing 3 shows part of the revised MazeGame_class module. The possible values of the gameType flag are represented as the constants STANDARD and NOWALLS which are defined in the mazev2 specification statements. These new constants have been made public but all of the other entities in the program are still hidden from the main program as before.

Figure 6 shows the class diagram for this version of the program. In comparison with mazev1, the old MazeGame_class has been articulated into a hierarchy of classes.

In this restructuring to mazev2 almost all of the changes have been localized to MazeGame_class and the game component classes. Room_class and Maze_class are identical to those in mazev1. Only the call to the constructor needed to be modified in the main MazeGame_Test program.

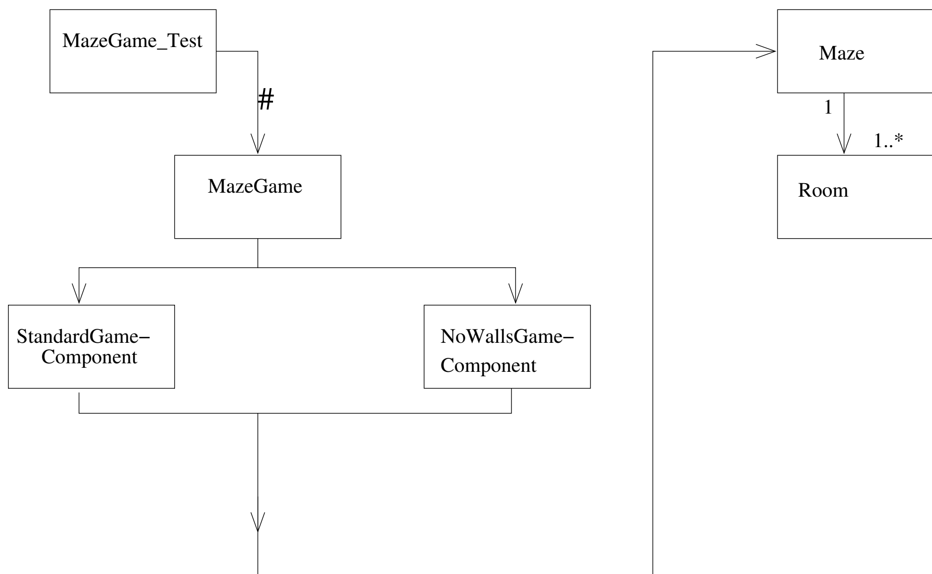

Fig. 6. Class diagram for `mazev2`.

Listing 3. The first part of the revised `MazeGame_class` in mazev2

```
module MazeGame_class
  use StandardGameComponent_class, standardNextMove => nextMove
  use NoWallsGameComponent_class, nowallsNextMove => nextMove
  implicit none
  private
  public :: MazeGame, new_MazeGame, del_MazeGame, gameLoop
  public :: STANDARD, NOWALLS
  type MazeGame
     private
     type(Maze) :: maze
     integer :: gameType
  end type MazeGame
  integer, parameter :: STANDARD=0,NOWALLS=1
contains
  ...
  function gameLoop(this, wonGame) result(res)
  ...
          select case(this%gameType)
          case(STANDARD)
             call standardNextMove(this%maze,iDirection)
          case(NOWALLS)
             call nowallsNextMove(this%maze,iDirection)
          end select
   ...
   end function gameLoop
end module MazeGame_class
```

The idea of restructuring code to split up algorithms into varying and non-

varying parts is already familiar to Fortran programmers. In traditional Fortran programming, this restructuring is carried out using multiple functions and subroutines but without any additional structure. The Template and Strategy pattern locate the varying and non-varying parts of algorithms in identifiable classes. This makes the restructuring more formal and evident and helps in reasoning about programming complex algorithms.

5 A Factory Pattern: `mazev3`

We now consider a case where the maze game is further extended to have rooms in the maze with different properties. Specifically, we will introduce a `TreasureRoom_class` module which will contain a number of treasures which must be collected by entering each room. A treasure room is a room with additional properties and in object-oriented languages this relationship would be modeled by inheritance.

Factory patterns encapsulate the creation of objects within an inheritance hierarchy. In the case of our new version of the maze game, the software will need to be able to create mazes made up of ordinary rooms or of "treasure rooms". We achieve this by defining a `GenericRoom_class` class which has associations to both the conventional `Room_class` and the new `TreasureRoom_class` and which creates the correct object according to the value of a flag.

Before describing the details of the Factory pattern itself, we now take the opportunity to discuss how to emulate inheritance in Fortran 90/95.

5.1 A new method of emulating inheritance in Fortran 90/95

Inheritance specializes the behavior of a class by adding data and functionality to it. A parent class in an inheritance chain might model a general concept like a *Room*. A child of this class might add entities such as data representing *treasure* and procedures to add, retrieve and count this treasure.

Fortran 90/95 can emulate inheritance by delegating from the child class to its parent. There are several ways in which this delegation can be implemented, but they all share characteristics of a simple "structural" [1] design pattern: a client class calls procedures on one class, the child, in blissful ignorance of the fact that a number of these calls are being delegated up to the parent.

A detailed discussion of earlier models for implementing inheritance in Fortran 90/95 can be found in [4,8]. The present implementation, found in `mazev3`,

is shown in Fig. 7. The extra information needed to specialize a `Room` to a `TreasureRoom` has been placed in a new `ExtendRoom_class` module and this class provides methods to read and write this private data. We made three changes to the `Room_class`. The first was to add a new element to the `Room` type which is a pointer reference to `ExtendRoom_class`. The second was to allocate this pointer in the `newRoom()` constructor. The third was to add accessor procedures to obtain pointers to the `ExtendRoom` object and to the `Room` object itself. In `TreasureRoom_class` we have a constructor which delegates to `Room_class` and to `ExtendRoom_class`. The association between `Room_class` and `ExtendRoom_class` only uses the `ExtendRoom` type so that exisiting code in `Room_class` is ignorant of the details of `ExtendRoom`.

This implementation of inheritance differs from a simple-minded association between `TreasureRoom_class` and `Room_class` in order to let objects of both classes have the same type (`Room`) while preserving privacy. If the types were different, then the names of every function which is shared would need to be rewritten to explicitly delegate to the parent class.

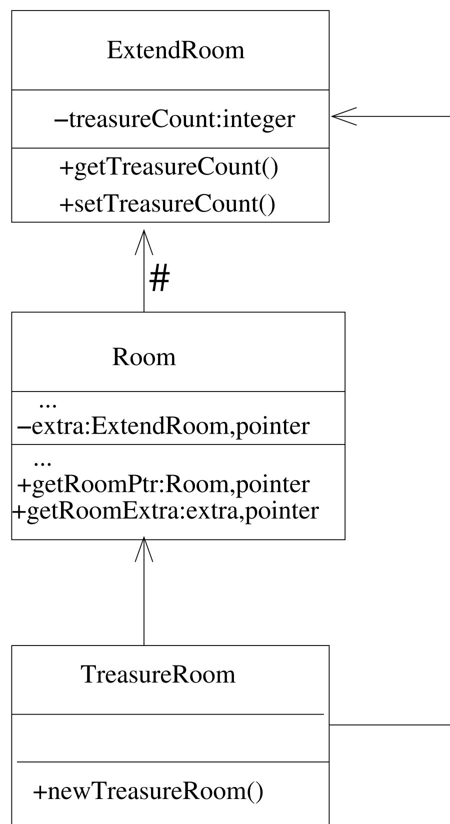

Fig. 7. Implementation of an inheritance hierarchy which includes `TreasureRoom` and `Room`.

5.2 Factory pattern implementation

Our Factory pattern is encapsulated in a new module GenericRoom_class which maintains associations with both Room_class and TreasureRoom_class. Objects of either of these classes are created in the GenericRoom_class constructor according to the value of a flag. No other coding is needed in GenericRoom_class apart from defining constants which represent legal values of the room type flag. The situation is shown in Fig. 8.

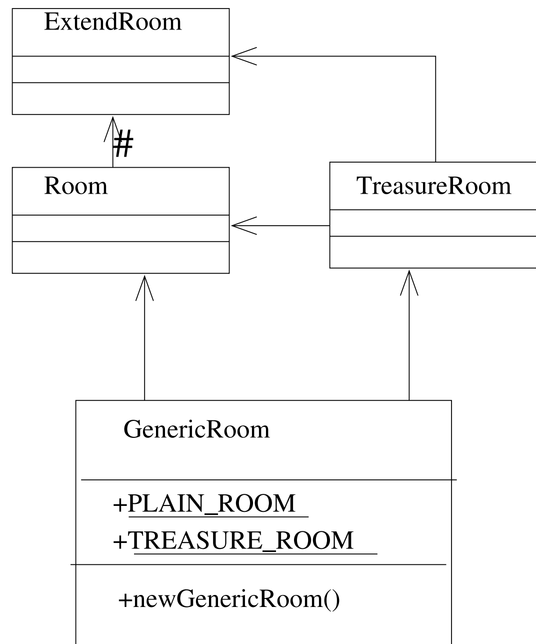

Fig. 8. The GenericRoom_class maintains links to both room classes and creates one or the other according to the value of a flag.

5.3 Modifications to the rest of the maze game

The Maze_class needs to be modified slightly to link to GenericRoom_class, rather than directly to Room_class. It also needs to include a procedure to count treasure over the entire maze. There is one additional procedure, collectTreasure(), which can be called to remove a treasure item from a particular room. This procedure delegates to setTreasureCount() in ExtendRoom_class.

A further modification to Maze_class is that its default access modifier has now been made private. Only a well-defined set of procedures and types has been denoted public. This is an additional "privatization" to that introduced by the MazeGame_class in Sec. 3. It makes the room subsystem in our program safe from modification by the other, game, subsystem. These privatizations of

subsystems are like the Facade design pattern which will be discussed in Sec. 6. A class diagram for the modified MazeGame_class is shown in Fig. 9.

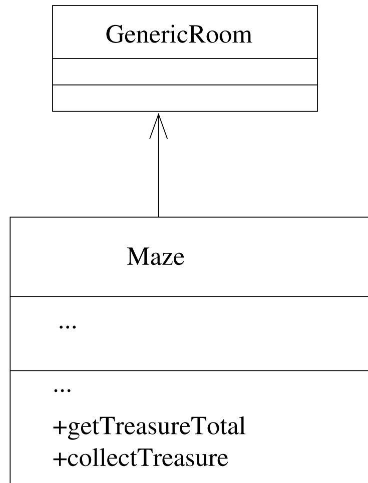

Fig. 9. The revised Maze_class used in version 3 of the maze game.

The other modifications to the software are quite straightforward. There is a new game component, TreasureGameComponent_class, whose nextMove() procedure is identical to that in StandardGameComponent_class with the addition of one line to collect treasure. We have also created one additional function in each game component to test for specialized game termination conditions. (The treasure game needs to test for the presence of treasure before terminating a game and, in general, future game logics may have their own termination conditions.) The MazeGame_class needs to have a new constant for the new game type and the case statement to select the next move needs to have an additional option. The test for termination in MazeGame_class now calls the termination test functions within the game components. Finally, the MazeGame_class constructor selects the room type based on the game type and passes a room type flag down to the Maze_class constructor. No significant changes are needed for the main MazeGameTest_class. A class diagram for the complete system is shown in Fig. 11.

6 Design Patterns in a Particle In Cell simulation framework

In this section, we briefly describe how the design patterns introduced above have been applied in a living scientific simulation code. The application is a framework for Particle in Cell (PIC) plasma simulation. PIC plasma codes integrate the self-consistent equations of motion of a large number of charged particles in electromagnetic fields. Their basic structure is to calculate the density of charge, and possibly current, on a fixed grid (by calling "deposit" procedures). Maxwell's equations, or a subset thereof, are solved on this grid

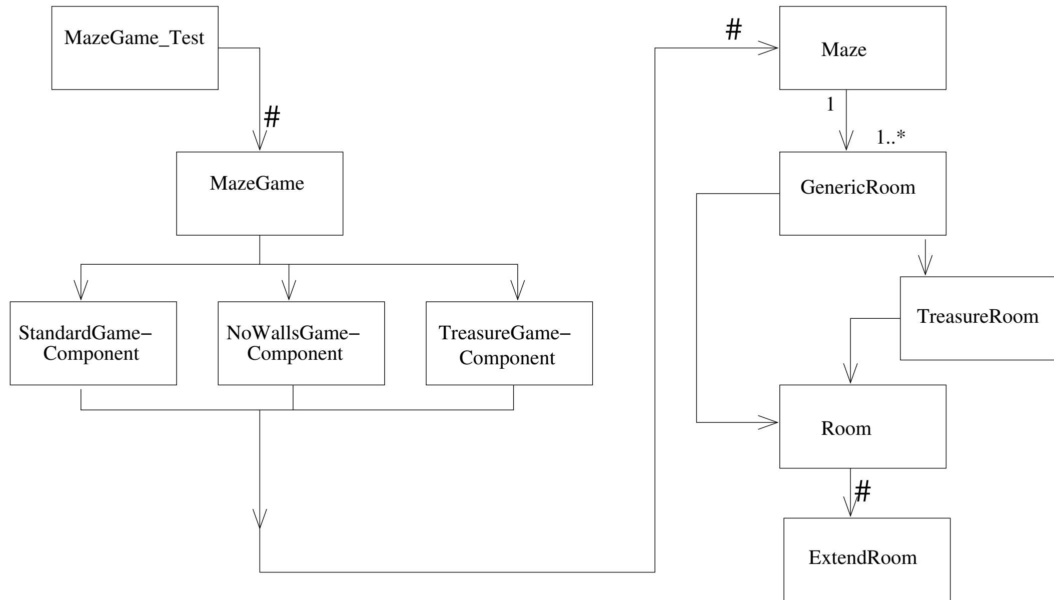

Fig. 10. Class diagram for `mazev3`.

and the forces on all particles are calculated using Newton's Law and the Lorentz force. Particle motion is advanced (by calling "push" procedures) 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: an *electrostatic* code models particles that respond to Coulomb forces only; *electromagnetic* codes deal with particles which interact to both electric and magnetic fields; *relativistic* codes deal with relativistic effects; *multispecies* codes can model the interactions of several different types of ions as well as electrons in the plasmas. Many of these PIC code types have a need to treat differing boundary conditions and differing solution techniques for the electromagnetic fields. A framework for PIC models of this type needs to allow all of these submodels to be generated and for common code to be maintained and reused between them.

The starting point for our PIC framework is a simple Fortran 90/95 class for electrostatic particles which respond to Coulomb forces only. 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`. Most of the subroutines included in this module provide a simple interface to legacy code:

Listing 4. Module for the electrostatic particles class in the PIC framework

```
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
    ... (includes calls to legacy code)

    subroutine charge_deposit(this,part,q)
! deposit particle charge onto mesh
    ... (includes calls to legacy code)

    subroutine es_push(this,part,fxyz,dt)
! advance particles in time from forces
    ... (includes calls to legacy code)

    subroutine particle_manager(this,part)
! handle boundary conditions
    ... (includes calls to legacy code)

    end module es_particles_class
```

Electromagnetic particles respond to both electric and magnetic forces. Their push procedure is different, and there needs to be 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 is created by association with the electrostatic class in a structural pattern as a substitute for inheritance:

Listing 5. Module for the electromagnetic particles class.

```
    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 Factory pattern can be used to create particles of the the correct type. In common with the Factory pattern described in Sec. 5, a “generic particle” class is constructed which creates storage for particles of the relevant type and which

then ensures that the correct type of push and current deposit subroutines are chosen for a given particle type. The implementation details of this pattern are described in reference [10], but we note that this implementation uses the conventional modeling of inheritance in Fortran 90/95 rather than the trick described in Sec. 5.

In the PIC framework, is it now necessary to create the correct type of fields corresponding to the particles: electrostatic particles need to solve a Poisson equation to obtain an electric field but electromagnetic particles need to solve the full Maxwell equations for both the electric and magnetic fields. This variation of fields can also be encapsulated in a field factory. The creation of the correct type of field and for the correct type of particle is an example of the coordination of families of factory patterns which is an example of an *Abstract Factory* design pattern.

The Strategy pattern is used in the PIC framework to encapsulate algorithms for solving for the electromagnetic fields under different boundary conditions (periodic, Dirichlet and Neumann). The implementation of this pattern is discussed further in reference [11].

Finally, the *Facade pattern* has been used to wrap up a subsystem of the PIC framework for modelling the plasma response to particles which are accelerated in a particle accelerator. The idea of the Facade pattern is to provide a simplified, high level interface to a subsystem. If one starts with a well-written Fortran 90/95 code, it is possible to turn it into a subsystem with a facade in three steps: Instead of a main program, one creates a module, and the declaration section becomes static data in that module. Active code in the old main program gets wrapped up into constructors and procedures. Finally, some objects of data will need to be passed between the new main program and the old subsystem. These data types will need to be defined inside the subsystem and the code needed to pass the data will need to be written.

7 Conclusions

In this paper we have discussed how some object-oriented design patterns can usefully be incorporated into software written in Fortran 90/95. The design patterns that we have mentioned, Factory, Strategy, Template, Abstract Factory and Facade, are all concerned with the encapsulation and reuse of *subsystems* of Fortran 90/95 programs. If there is one message from this paper, it is that Fortran 90/95 programmers can profitably direct their design focus to encapsulation at the subsystem level. As long as subsystems are clearly encapsulated through privatization, then minor, object-oriented "sins", such as letting some data entities be public inside a subsystem, can be tolerated if

they are needed for performance reasons.

Other work on design patterns in Fortran 90/95 has recently appeared in the literature. In addition to those references by ourselves discussed in the body of this paper, we note that [12,13] contain a discussion of implementation details of some patterns in Fortran 90/95. Clearly there is much scope for this discussion to continue as the scientific programming community engages in better software engineering practice to manage growing complexity. Similarly, the advent of Fortran 2003 will encourage scientific programmers to consider how patterns might best be incorporated into that language. We note that well-written Fortran 90/95 code which uses design patterns to encapsulate subsystems should be more easily reused a mixed Fortran 2003/90/95 framework than one which is not.

8 Acknowledgements

The authors express their appreciation for a reviewer's recommendation that the maze game would make a good study for illustrating design patterns in Fortran 90/95. They also greatly appreciate the work done by Joseph Antony on the maze game simulation software.

Viktor Decyk's work was performed, in part, at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA. Decyk was also partly supported by the US Department of Energy, under the SCIDAC program.

Henry Gardner wishes to acknowledge the support of the Education, Outreach and Training program of the Australian Partnership for Advanced Computing (APAC).

References

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995, ISBN 0201633612.

[2] Shalloway, A. and Trott, J. R., *Design Patterns Explained*, Addison-Wesley, 2002, ISBN 0201715945.

[3] Metsker, S. J., *Design Patterns Java Workbook*, Addison-Wesley, 2002, ISBN 0201743973.

[4] Decyk, V. K., Norton, C. D., and Szymanski, B. K., Scientific Programming **6** (1997) 363.

[5] Gray, M. G. and Roberts, R. M., Computers in Physics **11** (1997) 355.

[6] Machiels, L. and Deville, M. O., ACM Transactions on Mathematical Software **23** (1997) 32.

[7] Decyk, V. K., Norton, C. D., and Szymanski, B. K., Computer Physics Communications **115** (1998) 9.

[8] Decyk, V. K. and Norton, C. D., Scientific Programming **12** (2004) 45.

[9] Object Management Group, (2007), http://www.uml.org/. Last accessed 28 March 2007.

[10] Decyk, V. K. and Gardner, H. J., A factory pattern in fortran 95, in *Proceedings of the International Conference on Computational Science, ICCS2007, in Lecture Notes in Computer Science*, pages 576–583, Springer Verlag, 2007.

[11] Norton, C. D., Decyk, V. K., Szymanski, B. K., and Gardner, H., Scientific Programming **14** (2007), In press.

[12] Markus, A., SIGPLAN Fortran Forum **25** (2006) 13.

[13] Gardner, H. J. and Decyk, V. K., SIGPLAN Fortran Forum **25** (2006) 8.