Design patterns and Fortran 2003

Arjen Markus[1]
WL|Delft Hydraulics
The Netherlands

Introduction

Design patterns have been popular for quite some time in the context of C++ and Java and other object-oriented languages. While you can use them in a programming language like Fortran 95 which does not support classes or inheritance as named language features, as I demonstrated in a previous article (Markus, 2006), some patterns require additional work to be translated into Fortran with these named features.

The Fortran 2003 standard provides several facilities common to object-oriented programming and so should be more convenient in the implementation of these patterns. In this article I want to investigate three different patterns and demonstrate how we can indeed benefit from Fortran 2003.

Note: not all code fragments have not been tested, as there is at the moment of writing no compiler yet that supports the complete standard. While writing them the book by Metcalf, Reid and Cohen has been of great value (Metcalf et al. 2004).

Reverse communication

The first pattern I will talk about is known as reverse communication (Dongarra et al., 1995), as this is a design pattern that seems typical for a numerical context. Here is a skeleton version:

```
      call setup_calculation( ... )
loop: do
          call calculate_result( ..., x, y, task )

          !
          ! Handle the specific task
          !
          select case( task )
              case( 'calculate-x' )
                  ... calculate the "x" to be passed to the next round
              case( 'calculate-y' )
                  ... calculate the "y" to be passed to the next round
              case( 'done' )
                  exit loop
              case default
                  write(*,*) 'Severe programming error - unknown task: ', task
                  stop
          end select
      enddo loop
```

The motivation for this pattern is:

- It must be possible to customise parts of the computation. In the context of LAPACK for instance, different methods should be available to compute vector and matrix norms, depending on what is suitable for the particular problem.

[1] E-mail: arjen.markus@wldelft.nl

- While it is possible to define interfaces for routines that the user must supply, this forces a particular interface and data structure and it can become cumbersome when there are more than a few such tasks.

Let us make this more concrete (Metcalf et al. give a very similar example, op. cit. section 16.9). A function to numerically compute the integral of some mathematical function might look like this:

```
real function integral( start, stop, nsteps, f )
    real, intent(in)    :: start, stop
    integer, intent(in) :: nsteps
    interface
        real function f(x)
            real, intent(in) :: x
        end function f
    end interface

    real :: stepsize, x

    stepsize = (stop-start) / nsteps
    integral = 0.0
    do i = 1,nsteps
       x        = start + (i+0.5)*stepsize
       integral = integral + stepsize * f(x)
    enddo
end function integral
```

The interface to the mathematical function is fixed. We can not easily parameterize the function, for instance:

$$f(x) = \exp(-ax)/(1 + bx^2)$$

where a and b are parameters that can be changed during the program's execution. One workaround for this problem is to store them in a module, but this is not threadsafe:

```
    module parametrised_functions
        real :: a, b

    contains
    real function f(x)
        real :: x

        f = exp(-a*x)/(1.0 + b*x**2)
    end function f
    end module
```

Another solution to this is to use reverse communication:

```
    !
    ! Client code
    !
    start  =   0.0
    stop   =  10.0
    nsteps = 100
    call init_integral( intdata, start, stop, nsteps )

    do
        call compute_integral( intdata, value, x, task )
```

```
        select case( task )
            case( 'value' )
                value = f(x,a,b) ! We are free to chose what happens here.
                                 ! As long as we can pass the "right" value
                                 ! to the integration routine
            case( 'result' )
                integral = value
                exit
        end select
    enddo
```

and the general routines become:

```
module integration_service
    implicit none
    !
    ! Server code
    !
    type integration_data
        real              :: start, stop, stepsize, integral
        integer           :: nsteps, stp
        character(len=10) :: task
    end type integration_data

contains

subroutine init_integral( intdata, start, stop, nsteps )
    type(integration_data), intent(out) :: intdata
    real, intent(in)                    :: start, stop
    integer, intent(in)                 :: nsteps

    intdata%start    = start
    intdata%stop     = stop
    intdata%nsteps   = nsteps

    intdata%task     = 'init'
    intdata%stp      = 0
    intdata%integral = 0.0
    intdata%stepsize = (stop-start)/nsteps
end subroutine init_integral

subroutine compute_integral( intdata, value, x, task )
    type(integration_data), intent(inout) :: intdata
    real, intent(inout)                   :: value
    real, intent(out)                     :: x
    character(len=*), intent(out)         :: task

    select case ( intdata%task )
        case ( 'init' )
            x    = intdata%start + 0.5 * intdata%stepsize
            task = 'value'
            intdata%task = 'value'
        case ( 'value' )
            intdata%integral = intdata%integral + value

            if ( intdata%stp < intdata%nsteps ) then
                x = intdata%start + (intdata%stp + 0.5) * intdata%stepsize
                task = 'value'
            else
                value = intdata%integral
                task  = 'result'
            endif
    end select
```

```
end subroutine compute_integral

end module
```

The aspects that make reverse communication awkward to use (both from the client's point of view and from the point of view of the programmer of the general routines) are:

- Part of the logic of the underlying algorithm is exposed. The programmer who wants to use the services of the above routines must cooperate with this logic.
- The general routines need to have a way to store their state between calls. This is relatively easy though tedious for iterations, but almost impossible for recursive algorithms.

In Fortran 2003 we have type-bound procedures and we can extend types. So the fixed interface we had in the first version of the integration function can be made flexible:

```
module integration_1d

    !
    ! Define an object type with a procedure "eval" for evaluating the function
    !
    type, abstract :: function_object
        !
        ! No data components for this abstract type
        !
    contains
        procedure(eval_function), deferred :: eval
    end type function_object

    !
    ! Interface for the function "eval"
    !
    abstract interface
        real function eval_function( data, x )
            import   :: function_object
            type(function_object), intent(in) :: data
            real, intent(in)                  :: x
        end function eval_function
    end interface

contains

real function integral( start, stop, nsteps, f )
    real, intent(in)    :: start, stop
    integer, intent(in) :: nsteps

    !
    ! Use an "object" with type-bound procedure "eval"
    !
    type(function_object) :: f

    real :: stepsize, x

    stepsize = (stop-start) / nsteps
    integral = 0.0
    do i = 1,nsteps
       x        = start + (i+0.5)*stepsize
       integral = integral + stepsize * f%eval(x)
    enddo
end function integral
```

```
end module integration_1d
```

Now the user code needs to define a "function object" that conforms to the interface for such objects:

```
module functions

    use integration_1d

    type, extends(function_object) :: myfunc
        real :: a, b
    contains
        procedure :: eval => eval_exp
    end type myfunc

contains

real function eval_exp( data, x )
    class(myfunc), intent(in) :: data
    real, intent(in)          :: x

    eval_exp = exp(-a*x)/(1.0 + b * x**2)
end function eval_exp

end module functions

program test_functions

    use functions

    type(myfunc) :: f

    f = myfunc(1.0,2.0)

    write(*,*) 'Integral (0...1): ', integral(0.0, 1.0, 100, f )

end program
```

So, with a type-bound procedure we can evaluate the function we need with any parameters it requires without the integration routines having to "know" anything about these parameters.

Does this mean that reverse communication is an obsolete design? Probably not: reverse communication can be regarded as a particular implementation of a client-server system and it can also be regarded as an extension of an event-driven (or message-driven) approach: the client starts the service it requires and reacts to the messages – whether the service routines are fed the results of these reactions (as in the example above) is not relevant to the structure of the client code. In fact, when processing, say, XML files, this kind of structure is very common: a general routine reads the XML file and invokes directly or indirectly routines that do the actual processing of the contents.[2]

There is also the question of scalability: how many data are needed to properly handle the task? For instance, in a numerical computation of the flow of water through a network of pipes or open canals, one such task might be to regularly check if the solution is physically acceptable (mass-balance errors) or acceptable from an engineering point of view (maximum flow velocity,

[2] Another way of looking at reverse communication is that it provides a means to use co-routines, even if there is no direct support in the language for such co-operating tasks.

"negative" pressures indicating cavitation). Flexible methods might require a lot of extra information that will become part of the data passed to the general solver.

The Observer pattern

An alternative solution is found in the Observer pattern: instead of prescribing what tasks are to be done, the service simply offers a facility to register routines or data objects with type-bound procedures that will be invoked whenever a particular event occurs.

In the example of computing the flow of water in a network of pipes the event could be the completion of a single step in the integration over time. Here are some fragments of code that accomplish this:

- In the module containing the definitions of the solver we have the following types, one to serve as a "parent" class for the actual observer objects and the other to hold all the interesting data on the solution:

```
type, abstract :: observer
contains
    procedure(check_result), deferred, pass(obs) :: check
end type observer

type solution_data
    class(observer), dimension(:), allocatable :: observers
    ... ! Fields defining the solution
end type solution_data
```

- We can define an abstract interface to the checking routine and an abstract type that serves as a "parent" class for the actual observer objects:

```
abstract interface
    logical function check_result( obs, result_data )
        import :: observer, solution_data
        class(observer)     :: obs
        type(solution_data) :: result_data ! Type, not class
    end function check_result
end interface
```

  Descendents of the class "observer" must define a procedure "check" with the interface "check_result". What is done in this procedure is completely up to the particular implementation.
- The solver module contains at least procedures like below to allow the registration of the specific observer objects and the evaluation of the solution by these procedures:

```
subroutine add_observer( solution, obs )
    type(solution_data) :: solution
    class(observer)     :: obs

    solution%observers = (/ solution%observers, obs /)
end subroutine add_observer

subroutine solve( solution, ... )
    type(solution_data) :: solution

    ... ! Preliminaries

    do while ( time < time_end + 0.5*time_step )
        call solution%solve_one_step( ... )
```

```
            !
            ! This is the moment in the computation to
            ! see if the solution still obeys our
            ! requirements
            !
            acceptable = .true.
            do i = 1,size(solution%observers)
                acceptable = acceptable .and. &
                    solution%observers(i)%check( solution )
            enddo
            if ( .not. acceptable ) exit

            time = time + time_step

        enddo
        ...
    end subroutine solve
```

Via the subroutine we register data objects with the solution object that can check the solution according to some specific criteria. The solution object (or the associated procedures) contain no details whatsoever of these checks, except that the data object must be of a particular lineage – its type must be an extension of "observer".

(Note the use of the automatic reallocation feature in the subroutine "add_observer": this makes extending arrays really easy!)

In the example we have only one interesting type of event: the completion of a single time step. But we can easily extend this to any number of event types, each associated with its own type-bound procedure.

The purpose of the "Observer" pattern is to ensure a loose coupling between two parts of a program that need to interact. Ideally there should be no feedback from the observing data object to the calling solution object (or its "solve" procedure), for instance by setting a flag, like in the above or by changing fields in the "solution_data" structure. Such two-way communication can be established, though, by making the solution object an observer of the observing data object. You will need to avoid the circular module dependency by putting those parts in an extra layer, but that way a loose coupling is preserved.

The Template Method pattern

Because Fortran 2003 supports polymorphic variables – variables whose type is dynamic, rather than static – we can implement the Template Method pattern in an elegant way.[3] This design pattern is intended to solve the situation where the overall structure of an algorithm and some of its parts are general and other parts are specific:

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Redefine the steps in an algorithm without changing the algorithm's structure. (cited from Gamma et al. 1995)

Instead of subclasses we can also use different procedures that are then used within the algorithm. For example, the algorithm to solve a system of ordinary differential equations (ODE) is pretty much fixed (a loop over time, updating the solution and offering the possibility for outputting the

[3] Note that the list of observers in the previous example is a polymorphic variable as well.

intermediate results), but there are numerous methods to actually compute the solution for the next time.

In the code example below, we use different methods, like Euler, Heun, to name classical methods, in a general set up:

```
module ode_solvers
    implicit none
    type ode_solver
        real :: time, timestep, begin, end
        real, dimension(:), allocatable :: state
        procedure(eval_deriv), pointer :: f
        procedure(solve_step), pointer :: solve
    contains
        procedure(next_step), pointer  :: next_step
        procedure(initial), pointer    :: initial_values
    end type ode_solver

    type ode_method
        procedure(next), pointer :: solve
    end type

    integer, parameter :: ODE_EULER = 1
    integer, parameter :: ODE_HEUN  = 2
    integer, parameter :: ODE_RUNGE_KUTTA = 3
    ... (other methods)

    abstract interface
        !
        ! Subroutine for storing the initial values
        !
        subroutine initial( solver, initial_data )
            type(ode_solver)        :: solver
            real, dimension(:)      :: initial_data
        end subroutine initial

        !
        ! Function to compute the derivative
        !
        function eval_deriv( x, t )
            real, dimension(:)       :: x
            real, dimension(size(x)) :: eval_deriv ! Derivative is array of
                                                   ! same size as x
            real                     :: t
        end function eval_deriv

        !
        ! Interface function to hide the internal workings
        !
        function next(solver, time, time_step)
            type(ode_solver)   :: solver
            real               :: time
            real               :: time_step
            real, dimension(size(ode_solver%state)) :: next_step
        end function next_step

        !
        ! Function to actually compute the new values
        !
        function solve_step(solver, time, time_step)
            type(ode_solver)   :: solver
            real               :: time
```

```
            real               :: time_step
            real, dimension(size(ode_solver%state)) :: next_step
        end function next_step
    end interface

    !
    ! Declaration of the functions
    !
    procedure(initial) :: initial_values
    procedure(next)    :: next_step

    procedure(solve_step) :: solve_euler
    procedure(solve_step) :: solve_heun
    procedure(solve_step) :: solve_runge_kutta
    procedure(solve_step) :: ...

    !
    ! Definition of the specific methods for
    ! solving the ODE
    !
    type(ode_method), dimension(10) :: solvers = &
        (/ ode_method(solve_euler), ode_method(solve_heun), &
           ode_method(solve_runge_kutta), ... /)

contains

!
! General code
!

subroutine initial_values( solver, initial_data )
    type(ode_solver)   :: solver
    real, dimension(:) :: initial_data

    solver%state = initial_data
end subroutine initial_values

type(ode_solver) function new_solver( ode_type, rhs )
    integer, intent(in) :: ode_type
    procedure(f)        :: rhs

    new_solver%f = rhs
    if ( ode_type >= 1 .and. ode_type <= ... ) then
        new_solver%next_step = solvers(ode_type)
    else
        new_solver%next_step = solvers(1) ! Use Euler's method by default
    endif
end function

function next_step( solver, time, time_step )
    type(ode_solver)   :: solver
    real               :: time
    real               :: time_step
    real, dimension(size(solver%state)) :: next_step

    next_step = solver%solve( time, time_step )
    solver%state = next_step
end function next_step
```

The specific methods for solving the equations numerically can be defined as follows:

```
!
! Specific methods
```

```
!

function solve_euler( solver, time, time_step )
    type(ode_solver)   :: solver
    real               :: time
    real               :: time_step
    real, dimension(size(solver%state)) :: solve_euler

    solve_euler = solver%state + time_step * solver%f( solver%state, time )

end function solve_euler

function solve_heun( solver, time, time_step )
    type(ode_solver)   :: solver
    real               :: time
    real               :: time_step
    real, dimension(size(solver%state)) :: solve_euler
    real, dimension(size(solver%state)) :: v_predictor
    real, dimension(size(solver%state)) :: v_corrector

    v_predictor = solver%state + time_step * solver%f( solver%state, time )
    v_corrector = solver%state + time_step * &
                      solver%f( v_predictor, time+time_step )

    solve_heun = ( v_predictor + v_corrector ) / 2.0

end function solve_heun

... (all the other methods)

end module ode_solvers
```

The function “new_solver” fills a variable of type “ode_solver” that contains all the information and pointers to specific methods to solve a system of ordinary differential equations:

```
module some_ode
    !
    ! We use module variables here, out of laziness
    ! Better would be to use a derived type with all necessary
    ! data and procedure pointers
    !
    real :: a     ! Force/displacement
    real :: b     ! Dampening coefficient

contains

function dampened_oscillator( x, t )
    real, dimension(:)       :: x
    real, dimension(size(x)) :: dampened_oscillator
    real                     :: t

    dampened_oscillator(1) = x(2)                  ! Velocity
    dampened_oscillator(2) = -a * x(1) -b * x(2)   ! Force

end function dampened_oscillator

end module some_ode

program solve_some_ode
    use ode_solvers
    use some_ode
    ...
```

```
        solver = new_solver( ODE_HEUN, dampened_oscillator )

        call solver%initial_values( (/ 0.0, 1.0 /) )

        time = 0.0
        do while( time < 10.0 )
            state = solver%next_step( time, deltt )
            call print_state( state )
            time  = time + deltt
        enddo

    end program
```

So, as required by the Template pattern, we have a number of fixed procedures ("new_solver", "initial_values", "next_step") that are used within a fixed algorithm and a number of alternatives to be used inside it. By selecting a different constant in the construction of the solver, we select whatever method we think is suitable – nothing else in the program is influenced.

One thing should be noted: we can not check at compile time if the number of equations (as seen in the result of the function "f") is equal to the number of state variables (as defined in the call to "initial_values"). We need a runtime check for that – for instance in the subroutine "next_step", or "initial_values".

A further enhancement to the above could be to use a derived type with a type-bound procedure "f" instead of the function "f" directly. Just as with the first example (an alternative for reverse communication) we avoid module variables that way. But I will leave that as an exercise.

The Façade pattern revisited

In answer to my previous article, Henry Gardner and Viktor Decyk presented some of their own work on object-oriented programming in Fortran 95 (Gardner and Decyk, 2006). They demonstrated a different interpretation of the Façade pattern: one in which a complete program is encapsulated as a module with static data:

```
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
```

This example is expanded to include additional data for the plasma class, data that are relevant to both the plasma computations and the encompassing program.

If we take the object-oriented facilities of Fortran 2003, we can in fact expand this example in a completely different way: include different implementations of the plasma class. This could be relevant for investigating which formulations describe the underlying physics better:

```
module generic_plasma_class
    type, abstract :: plasma_data
        ... ! The relevant general data
```

```
    contains
        ... ! The relevant general routines/interfaces
    end type plasma_data

    abstract interface
        ! The interfaces
    end interface
contains
    ... ! The general routines
end module generic_plasma_class

!
! The specific classes we want for this computation
!
module plasma_class_mark_1
    use generic_plasma_class

    ! Actual implementation:

    type, extends(plasma_data) :: plasma_data_mark_1
        ...
    end type

end module plasma_class_mark_1

module plasma_class_mark_2
    use generic_plasma_class

    ! Actual implementation

end module plasma_class_mark_2

...

!
! This puts them all together
!
module plasma_class
    use plasma_class_mark_1
    use plasma_class_mark_2
    ...
contains
function new_plasma( which ) result(p)
    integer :: which
    class(plasma_data) :: p

    type(plasma_data_mark_1) :: p1
    type(plasma_data_mark_2) :: p2

    select case( which ) ::
        case(1) ::
            call p1%new_plasma   ! Initialise
            p = p1
        case(2) ::
            ...
    end select

end function new_plasma
end module plasma_class

program facade
    use plasma_class
    integer :: i, nloop = 100, which
```

```
    class(plasma_data) :: plasma

    plasma = new_plasma( which )

    ! loop over number of time steps

    do i = 1,nloop
        call plasma%update_plasma
    enddo
end program
```

Using the new class construct we can keep the main program ignorant of the specifics of the various implementations of the plasma computations. We only need to select the implementation at the beginning.

Conclusion

These examples show that Fortran 2003 offers features that help implement a large class of well-known design patterns. Sometimes the need to specify an interface in detail may make the code lengthier than seems necessary, but they make it possible for the compiler to check correct usage.

Gardner and Decyk indicate that object oriented techniques and object-oriented design patterns in particular are such a new subject in the context of Fortran that different interpretations by different authors are almost inevitable. I agree: modules are the Fortran means of packaging data and functionality, they are not as such designed for object-orientation. As they are the obvious means to achieve the isolation one requires in object-oriented programming, we must find ways to translate the concepts of object-oriented programming as found in the literature to Fortran.

This will not always be easy or perhaps even possible without sacrificing some aspect. But let us keep in mind that the various programming languages that are now claimed to be object-oriented use these concepts in different ways too.

I would like to thank Henry Gardner for reviewing the text and for supplying useful comments as well as several references to papers on this same subject.

References

V. K. Decyk and H. Gardner (2007)
A Factory Pattern in Fortran 95
Lecture Notes in Computer Science 4487, pp. 583-590, 2007

J. Dongarra, V. Eijkhout and A. Kalhan (1995)
Reverse Communication Interface for Linear Algebra Templates for Iterative Methods
http://www.netlib.org/lapack/lawnspdf/lawn99.pdf

E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995)
Design Patterns: Elements of Reusable Object-Oriented Software
Addison-Wesley, 1995

H. Gardner and V. Decyk (2006)
Comments on the Arjen Markus article: Design patterns and Fortran 90/95
ACM Fortran Forum, August 2006, 25, 2

A. Markus (2006)

Design patterns and Fortran 95
ACM Fortran Forum, April 2006, 25, 1

M. Metcalf, J. Reid and M. Cohen (2004)
Fortran 95/2003 explained
Oxford University Press, 2004

C. Norton, V.K. decyk, B.K. Szymanski and H. Gardner (2007)
The transition and adoption to modern programming concepts for scientific computing in Fortran
Scientific Programming, april 2007

A. Shalloway and J.R. Trott (2002)
Design Patterns Explained
A new perspective on object-oriented design
Addison-Wesley, 2002