Design patterns and Fortran 90/95

Arjen Markus[1]
WL|Delft Hydraulics
The Netherlands

What are design patterns?

In the literature on object oriented programming (OO), *design patterns* are a very popular subject. Apart from any hype that may be connected to the concept, they are supposed to help you look at a programming problem and come up with a robust design for its solution. The reason design patterns work is not that they are something new, but instead that they are time-honoured, well-developed solutions.

I will not repeat the story about architectural design patterns and Christopher Alexander who recognised their potential. Instead I will try to explain how these (software) design patterns can be used in setting up Fortran 90/95 programs, despite the "fact" that Fortran 90/95 lacks certain OO features, such as inheritance and polymorphism. It may not be stressed in all OO literature, but design patterns help you find solutions that do not necessarily involve inheritance or polymorphism (*cf.* Shalloway and Trott, 2002).

Design patterns come by fancy names such as the Adapter pattern or the Decorations pattern and explaining what they are and how to use them is best done via a few examples.

Two course exercises

During the preparation of an in-house course on Fortran 90/95, I wrote down two exercises to get people to start thinking about the language and about ways to make their programs more widely useable:

Exercise 1:
Set up a library of (basic) numerical methods to solve a system of ordinary differential equations. The methods must at least include the one-step method by Euler. The functions of this library take as one of their arguments the name of a function that will compute and return the derivatives of each dependent variable at the given time and state. In other words:

The system:

$$\begin{aligned} d\underline{x}/dt &= \underline{f}(t,\underline{x}) \\ \underline{x}(0) &= \underline{x_0} \end{aligned}$$

is to be solved via:

$$\underline{x_{k+1}} = \underline{x_k} + dt * \underline{f}(t,\underline{x_k})$$

with given initial condition $\underline{x_0}$

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

The function `f` is to be implemented as a function that returns an array of derivatives (this requirement makes it impossible to implement this using FORTRAN 77 features only).

Exercise 2:
Devise a program that will read data from a file (one number per line, to keep it simple) and gather all these data for later processing. The processing step involves printing some statistical parameters: mean, minimum, maximum and a (cumulative) histogram of the number of data in ten equally-sized intervals between the minimum and maximum.

Notes:
- We do not know in advance the range of the data (they are within the range of ordinary reals, but that is all we know) nor the number of data.
- The program is to be written in such a way that we can later isolate the various tasks and put them in a library for general use.

Now, I wanted to come up with elegant solutions of the above problems to show off the modern features of Fortran. I also received a few questions about these exercises and I started to think some more about them. At this time (October 2005) a discussion on the Fortran newsgroup (comp.lang.fortran) caught my eye and I suddenly realised what I was looking for. The book *Design Patterns Explained* by Shalloway and Trott (2002) did the rest.

The Façade pattern

One of the simplest patterns is the *Façade pattern*. You typically encounter it when you have to use a complex or inconvenient library. Let us have a look at *exercise 1*. In the description only Euler's method is mentioned, but suppose we expand it to include Heun's method and one of the methods by Runge and Kutta too (these are simple enough, the actual routines can have same interface as the original one). Here is a skeleton version of a module that implements them:

```
! ode_methods.f90
!    Module to solve systems of ordinary differential equations
!    Note:
!    Each function
!
module ode_methods
   implicit none
contains
function euler_step( time, x, func, deltt ) result(newx)
   real, intent(in)                :: time
   real, intent(in),dimension(:) :: x
   real, intent(in)                :: deltt
   real, dimension(size(x))        :: newx

   interface
      function func( time, x ) result(dx)
         real, intent(in)                :: time
         real, intent(in),dimension(:) :: x
         real, dimension(size(x))        :: dx
      end function func
   end interface

   newx = x + deltt * func(time,x)
end function euler_step

function heun_step( ... )
   ...
end function heun_step
```

```
function runge_kutta_step( ... )
   ...
end function runge_kutta_step

end module ode_methods
```

For any one who has had some training in numerical analysis this will be a small, easy-to-use library:

```
module myfunc
   implicit none
contains

function func( time, x ) result(dx)
   real, intent(in)               :: time
   real, intent(in),dimension(:) :: x
   real, dimension(size(x))      :: dx

   dx(1) = 0.1 * ( 1.0-x(1) ) - 0.2 * x(2)
   dx(2) = -0.05 * x(2)
end function func

end module myfunc


program solve_ode
   use ode_methods
   use myfunc

   real, dimension(2) :: x
   real               :: time
   real               :: deltt

   x(1)  = 1.0
   x(2)  = 0.5
   time  = 0.0
   deltt = 0.1

   do while ( time .lt. 10.0 )
      write(*,'(3f10.4)') time, x

      x    = euler_step(time, x, func, deltt ) ! Or any other
      time = time + deltt
   enddo
end program solve_ode
```

But what if your customers do not want to be bothered picking the right method? What if they ask you to provide an even easier-to-use library? Do you throw away the superfluous methods or do you use another solution? *It is time for the Façade pattern*.

Rather than come up with a new library (based on the code of the old one, but slightly modified), you simply shield off all the methods they do not need to know or have access to:

```
module ode_solve
   use ode_methods, only: next_step => euler_step
end module ode_solve
```

This is the module your customers will use. It gives limited access to the original library. If needed, you could add a wrapper function to this new module which would, say, decide whether the time-step is small enough and if not split the one step into a number of smaller steps, thereby releasing your customer from the responsibility of supplying that himself.

The program becomes more general (it refers to a generic method, instead of Euler's):

```
program solve_ode
   use ode_solve
   use myfunc

   …

   do while ( time .lt. 10.0 )
      write(*,'(3f10.4)') time, x

      x    = next_step(time, x, func, deltt )
      time = time + deltt
   enddo
end program solve_ode
```

The *Façade pattern* is also an excellent way to describe a Fortran 90/95 wrapper around libraries like LAPACK to get rid of arguments that indicate the array sizes or pass work space to the routines.

The Adapter pattern

A solution to the second exercise is:

```
! stat1.f90 --
!    Straightforward solution to exercise 2
!
program stat1
   implicit none

   integer, parameter          :: increm = 10
   integer                     :: number
   real, dimension(:), pointer :: data1
   real, dimension(:), pointer :: data2

   integer                     :: ierr
   real                        :: value
   real                        :: vmin
   real                        :: vmax
   real                        :: vmean

   !
   ! Initialise the array, open the file and store all
   ! the data
   !
   number = 0
   allocate( data1(1:number) )

   open( 10, file = 'stat1.inp' )

   do
      read( 10, *, iostat=ierr ) value
      if ( ierr .ne. 0 ) exit

      if ( number .ge. size(data1) ) then
```

```
         allocate( data2(number+increm) )
         data2(1:number) = data1(1:number)
         deallocate( data1 )
         data1 => data2
      endif

      number         = number + 1
      data1(number) = value
   enddo

   close( 10 )

   !
   ! Print the statistics
   !
   if ( number .gt. 0 ) then
      vmin  = minval( data1(1:number) )
      vmax  = maxval( data1(1:number) )
      vmean = sum( data1(1:number) ) / number
      write(*,*) 'Mean:    ', vmean
      write(*,*) 'Minimum: ', vmin
      write(*,*) 'Maximum: ', vmax
      call histogram( data1(1:number), vmin, vmax )
   else
      write(*,*) 'No data in the file!'
   endif
contains
subroutine histogram( array, vmin, vmax )
   real, dimension(:) :: array
   real               :: vmin
   real               :: vmax

   integer            :: i
   real               :: bound

   write(*,*) 'Upper limit - number of data'
   do i = 1,10
      bound = vmin + i * (vmax-vmin)/10.0
      write(*,*) bound, count( array .le. bound )
   enddo
end subroutine histogram

end program stat1
```

It uses the functions from the library that the Fortran 90/95 standards define, rather than contain its own code for them, but it does fail as an elegant solution in two aspects:

- The reallocation of the array holding the data is left to the using program.
- Reallocation requires extra memory. You could use a linked list of blocks of memory instead, but then you are not able to use the standard functions.

What would be nice is a library that accommodates for both types of storage: the simple *dynamically allocated array* which will cause overhead when it is being filled, but which is otherwise easy to use and the *linked list of memory blocks* which does not have the overhead, but which presents more difficulty when accessing the individual data.

This is the type of problem that is suited for the *Adapter* pattern: provide a uniform interface for disparate objects with the same or nearly the same functionality.

We can design the solution in at least two ways:

- Take advantage of our knowledge of the underlying storage mechanism. This leads to an implementation that is fast (we access the data directly) but that requires tuning to each type of data storage we care to include.
- Define a uniform interface to store and access individual data items. The statistical module can then be written essentially independent of the underlying storage. The only drawback: it may be slower, as it does not "know" anything about the properties of the storage mechanism and therefore relies on getting individual items or storing individual items via functions and subroutines.

Actually we can combine these two approaches and I will do so by using the first approach for the dynamic array and the second for the linked list.

However we design the libraries for storage and statistics, the main program will be almost the same:[2]

```
! stat_main.f90 --
!    General main program
!
program stat_main
   use statistics

   implicit none

   real              :: value
   integer           :: ierr
   integer           :: number
   type(STAT_ARRAY) :: data
   !
   ! Or alternatively:
   !    type(STAT_LIST)        :: data
   !

   call stat_init( data )

   open( 10, file = 'stat1.inp' )

   do
      read( 10, *, iostat=ierr ) value
      if ( ierr .ne. 0 ) exit

      call stat_add( data, value )
   enddo

   close( 10 )

   !
   ! Print the statistics
   !
   number = stat_count( data )
   if ( number .gt. 0 ) then
      write(*,*) 'Mean:    ', stat_mean( data )
      write(*,*) 'Minimum: ', stat_minimum( data )
      write(*,*) 'Maximum: ', stat_maximum( data )
      call histogram
   else
```

[2] The histogram subroutine could of course be moved into the library too. I have not done it because it prints directly to the screen and for a library you probably want control over that. So that would have generated unnecessary details.

```
      write(*,*) 'No data in the file!'
   endif
contains
subroutine histogram

   real               :: vmin
   real               :: vmax

   integer            :: i
   real               :: bound

   vmin = stat_minimum( data )
   vmax = stat_maximum( data )

   write(*,*) 'Upper limit - number of data'
   do i = 1,10
      bound = vmin + i * (vmax-vmin)/10.0
      write(*,*) bound, stat_count( data, bound )
   enddo
end subroutine histogram

end program stat_mean
```

The only choice made in the program is the type of storage via the derived types STAT_ARRAY and STAT_LIST.

To illustrate a kind of code inheritance that is possible in Fortran despite the lack of a formal inheritance feature, I start with the linked list.

The linked list

We want to separate the storage mechanism from the statistical routines, so we define two modules. The one for the linked list looks like this:[3]

```
! module for storing blocks of data in a linked list
!
module block_lists
   integer, parameter, private :: blocksize = 10

   type STORE_LIST
      integer                    :: number
      real, dimension(:), pointer :: data
      type(STORE_LIST), pointer   :: next
   end type STORE_LIST

contains
subroutine store_init( store )
   type(STORE_LIST), intent(inout)     :: store

   store%number = 0
   allocate( store%data(1:blocksize) )
   nullify( store%next )

end subroutine store_init

subroutine store_add( store, value )
   type(STORE_LIST), intent(inout), target :: store
```

[3] I apologize for the amount of code in this part - I wanted to show something that actually works instead of just skeleton code.

```
   type(STORE_LIST), pointer               :: current
   type(STORE_LIST), pointer               :: new

   current => store

   do while ( associated(current%next) )
      current => current%next
   enddo

   if ( current%number .ge. blocksize ) then
      allocate( new )
      call store_init( new )
      current%next => new
      current => new
   endif

   current%number = current%number + 1
   current%data(current%number) = value

end subroutine store_add

real function store_value( store, idx )
   type(STORE_LIST), intent(in), target :: store
   integer, intent(in)                  :: idx

   type(STORE_LIST), pointer            :: current
   type(STORE_LIST), pointer            :: new
   integer                              :: idxn

   current => store
   idxn    =  idx

   do while ( idxn .gt. blocksize )
      if ( associated(current%next) ) then
         current => current%next
         idxn    =  idxn - blocksize
      else
         store_value = 0.0 ! Actually an error!
         return
      endif
   enddo

   store_value = current%data(idxn)

end function store_value

integer function store_count( store )
   type(STORE_LIST), intent(in), target :: store

   type(STORE_LIST), pointer            :: current

   current => store
   store_count = 0

   do while ( associated(current%next) )
      store_count = store_count + current%number
      current => current%next
   enddo

end function store_count

end module block_lists
```

The statistics module can be written in an almost generic way:

```
module stat_lists
   use block_lists, DATA_STORE => STORE_LIST

   private :: store_init, store_add, store_value, store_count

!
! Include the actual code
!
include 'stat_generic.f90'

end module stat_lists
```

Via the renaming facilities that Fortran offers we have managed to free the source code for the statistical routines from any (textual) reference to the underlying storage mechanism, so the actual code can be put in a separate file that can be included whenever convenient:

```
! stat_generic.f90 --
! Statistical module, using list of blocks
!
!
! Generic part
!
   type STAT_DATA
      integer          :: number
      real             :: vsum
      real             :: vmin
      real             :: vmax
      type(DATA_STORE) :: data
   end type

contains

subroutine stat_init( store )
   type(STAT_DATA), intent(inout) :: store

   store%number = 0
   store%vsum   = 0.0
   store%vmin   = 0.0
   store%vmax   = 0.0
   call store_init( store%data )
end subroutine stat_init

subroutine stat_add( store, value )
   type(STAT_DATA), intent(inout) :: store
   real, intent(in)               :: value

   store%vsum   = store%vsum + value
   if ( store%number .eq. 0 ) then
      store%vmin   = value
      store%vmax   = value
   else
      store%vmin   = min( store%vmin, value )
      store%vmax   = max( store%vmax, value )
   endif

   store%number = store%number + 1

   call store_add( store%data, value )
end subroutine stat_add
```

```
real function stat_minimum( store )
   type(STAT_DATA), intent(inout) :: store

   stat_minimum = store%vmin
end function stat_minimum

real function stat_maximum( store )
   type(STAT_DATA), intent(inout) :: store

   stat_maximum = store%vmax
end function stat_maximum

real function stat_mean( store )
   type(STAT_DATA), intent(inout) :: store

   stat_mean = 0.0
   if ( store%number .gt. 0 ) then
      stat_mean = store%vsum / store%number
   endif
end function stat_mean

real function stat_count( store, upper )
   type(STAT_DATA), intent(inout) :: store
   real, optional                 :: upper

   integer                        :: i

   if ( .not. present(upper) ) then
      stat_count = store%number
   else
      stat_count = 0
      do i = 1,store%number
         if ( store_value(store%data,i) .le. upper ) then
            stat_count = stat_count + 1
         endif
      enddo
   endif
end function stat_count
!
! End of generic part
```

The dynamic array

For the dynamically allocatable array we do not keep a complete separation between the storage and the statistical source code, because the counting function can be more efficiently written. We do want to take advantage of the generic code shown in the previous section. First we define the module for this type of storage:

```
! module for storing data in a "dynamic array"
!
module dyn_arrays
   implicit none
   integer, parameter, private :: increment = 10

   type DYN_ARRAY
      integer                       :: number
      real, dimension(:), pointer :: data
   end type DYN_ARRAY

contains
subroutine store_init( store )
```

```
   type(DYN_ARRAY), intent(inout)    :: store

   store%number = 0
   allocate( store%data(1:increment) )

end subroutine store_init

subroutine store_add( store, value )
   type(DYN_ARRAY), intent(inout) :: store
   real, intent(in)               :: value

   real, dimension(:), pointer    :: new_data
   integer                        :: old_size

   old_size = size( store%data )
   if ( store%number .ge. old_size ) then
      allocate( new_data(1:old_size+increment) )
      new_data(1:old_size) = store%data
      deallocate( store%data )
      store%data => new_data
   endif

   store%number = store%number + 1
   store%data(store%number) = value

end subroutine store_add

real function store_value( store, idx )
   type(DYN_ARRAY), intent(in) :: store
   integer, intent(in)         :: idx

   if ( idx .le. 0 .or. idx .gt. store%number ) then
      store_value = 0.0 ! Actually an error
   else
      store_value = store%data(idx)
   endif

end function store_value

integer function store_count( store )
   type(DYN_ARRAY), intent(in) :: store

   store_count = size( store%data )

end function store_count

end module dyn_arrays
```

Now the statistical module can be written as:

```
module stat_arrays
   use dyn_arrays, DATA_STORE => DYN_ARRAY

   private :: store_init, store_add, store_value, store_count

!
! Include the actual code
!
include 'stat_generic.f90'

!
! Overwrite the function stat_count:
```

```
! We have a more efficient way
!
integer function stat_count_array( store, upper )
   type(DATA_STORE), intent(in) :: store
   real, optional               :: upper

   if ( present(upper) ) then
      stat_count_array = count( store%data .le. upper )
   else
      stat_count_array = size( store%data )
   endif
end function stat_count_array

end module stat_arrays
```

We have re-implemented a single function in this module to be able to use more efficient data access but we rely on the generic code for a number of others – a kind of inheritance or perhaps the word *delegation* is more appropriate for this flavour of object oriented programming.

Combining the modules

At this point we have two statistical modules, now we need a way to glue them together, so that the using program can simply select the right implementation via the type of storage it uses. The overall module imports the routines from the two specific modules under a specific name and then unifies them under a generic name:

```
! Overall statistics module
!
module statistics
   use stat_lists, only: STAT_LIST => STAT_DATA,
                         stat_init_list    => stat_init,
                         stat_add_list     => stat_add,
                         stat_count_list   => stat_count,
                         stat_mean_list    => stat_mean,
                         stat_minimum_list => stat_minimum,
                         stat_maximum_list => stat_maximum

   use stat_arrays, only: STAT_ARRAY         => STAT_DATA,
                          stat_init_array    => stat_init,
                          stat_add_array     => stat_array,
                          stat_count_array,
                          stat_mean_array    => stat_mean,
                          stat_minimum_array => stat_minimum,
                          stat_maximum_array => stat_maximum

   !
   ! Provide uniform interfaces to these specific routines
   !
   interface stat_init
      module procedure stat_init_list
      module procedure stat_init_array
   end interface

   interface stat_add
      module procedure stat_add_list
      module procedure stat_add_array
   end interface

   interface stat_mean
      module procedure stat_mean_list
      module procedure stat_mean_array
```

```
   end interface

   interface stat_minimum
      module procedure stat_minimum_list
      module procedure stat_minimum_array
   end interface

   interface stat_maximum
      module procedure stat_maximum_list
      module procedure stat_maximum_array
   end interface
end module

   interface stat_count
      module procedure stat_count_list
      module procedure stat_count_array
   end interface
end module
```

All this renaming may seem inelegant at first, but it is in fact not much different from the typical class definition in C++ or the use of interfaces in Java. There you separate the interface information from the actual implementation via two different source files. In the example above we define in a separate module how the two modules fit together.

Let us recapitulate what we have:

- A general module for accessing the statistical functions and data structures
- Two specific modules for dealing with two different types of storage
- A generic source file that can be used for an implementation of yet another type of storage
- A main program that can select which implementation to use by changing a single declaration

Setting up the infrastructure may have seemed a lot of work, but adding a third type of storage will now be very simple.

The Singleton pattern

A third pattern I want to discuss is the *Singleton* pattern: a way to let all parts of a program have access to a single set of data while guaranteeing that there is only one such set. The *Singleton* pattern is useful for instance for configuration data – they will be required throughout the program and you do not want to pass a reference around in all argument lists.

Let us have a look at the following code fragment (the skeleton definition of a module that manipulates a list of key-value pairs, also known as a dictionary):

```
module dictionary
   implicit none
   private            ! Make sure everything is hidden, unless we make it
                      ! publically accessible

   integer, parameter :: value_length = 120
   type KEY_VALUE
      character(len=40)  :: key
      character(len=value_length) :: value
   end type KEY_VALUE

   type DICT
      private
      type(KEY_VALUE), dimension(:), pointer :: list
   end type DICT
```

```
   type(DICT), private :: dict_data   ! We always want this one to be private

   !
   ! Public functions/subroutines
   !
   public :: dict_load
   public :: dict_set
   public :: dict_get

contains

subroutine dict_load( filename )
   …
end subroutine

subroutine dict_set( key, value )
   …
end subroutine

function dict_value( key ) result(value)
   character(len=*)            :: key
   character(len=value_length) :: value
   …
end function dict_value

end module dictionary
```

With this module it is possible to manipulate the dictionary variable "dict_data" and to get the values of the keys stored in that dictionary. It is *not* possible to create a new one - the DICT type is hidden and there is no routine that allows you to manipulate an arbitrary variable of the DICT derived type anyway.

This module constitutes a simple example of the *Singleton* pattern. Despite the precautions we took to hide almost everything, the code remains flexible: should you later decide that the program needs to be able to create new dictionaries and manipulate them, a few changes (given below in italics) will suffice to allow this in a backward compatible manner:

```
module dictionary
   implicit none
   private            ! Make sure everything is hidden, unless we make it
                      ! publically accessible

   integer, parameter :: value_length = 120
   type KEY_VALUE
      character(len=40)  :: key
      character(len=value_length) :: value
   end type KEY_VALUE

   type DICT
      private
      type(KEY_VALUE), dimension(:), pointer :: list
   end type DICT

   type(DICT), private :: dict_priv   ! Renamed for clarity

   !
   ! Public functions/subroutines
   !
   public :: DICT
```

```
   public :: dict_load
   public :: dict_set
   public :: dict_get

   interface dict_load
      module procedure dict_load_pub
      module procedure dict_load_priv
   end interface

   interface dict_set
      module procedure dict_set_pub
      module procedure dict_set_priv
   end interface

   interface dict_value
      module procedure dict_value_pub
      module procedure dict_value_priv
   end interface

contains

subroutine dict_load_priv( filename )
   call dict_load_pub( dict_priv, filename )
end subroutine

subroutine dict_load_pub( dict_data, filename )
   type(dict) :: dict_data
   !
   ! The original code of dict_load in the previous version, only now
   ! dict_data is a local variable, instead of a variable in the
   ! module.
   !
end subroutine

… (ditto for dict_set and dict_value)

end module dictionary
```

A program that needs the singleton approach continues to work, as the overloading of the routines makes sure you can use the hidden dictionary. We have also reused all of the code, while making this new version.

Comparison with numerical libraries

The design patterns described above focus on the management of programming interfaces to entities in the program that is to be built. Implicit claims and in fact some explicit claims in the literature on design patterns are made that this kind of patterns is not possible in the realm of procedural programming and programming languages that do not support the object-oriented paradigm. In the main part of this article I have demonstrated that at least some canonical design patterns *can* be implemented in Fortran - in my opinion even in an elegant way.

The question arises whether design patterns are actually absent from software developed for solving numerical problems as that is the apparent niche for Fortran. Judging from a few references I would say that conscientious approaches to the design of such software abound, as much as they do in the literature on object-oriented design:

- Libraries such as BLAS are built in *layers*- BLAS routines fall into three categories that each use lower-level routines from a previous layer. Furthermore because of the different storage layouts (full matrices, band matrices and others) there are versions of most or all routines that specifically deal with these storage layouts: the interface is the same but the data objects that are passed are not. This has all the characteristics of the Façade and Adapter patterns (to hide the complexity of low-level routines and to provide a uniform interface to otherwise disparate objects).

- The Livermore solver for systems of ordinary differential equations is another example of deliberate design (*cf.* …) Behind a simple interface, it contains a vast collection of numerical methods. These methods are selected both by the user and automatically: the user selects the *class* of solution methods (which has consequences for the routines applied by the user) and the solver itself tries to optimise the numerical performance by selecting more or less accurate methods from the selected class.
  The documentation describes the design decisions as well as the internal working in some detail. From this description it is clear that the authors wanted to make a system that can be used, without changes to the code, for a wide variety of applications. With earlier libraries of this kind the user had to adapt the code to fit his/her requirements.
- In the context of solving systems of linear equations, Dongerra et al. (1995) describe a particular program design, called *reverse communication*. The basic idea is to provide flexibility in iteratively solving such equations that can not easily be achieved otherwise. In this design, user code provides specific functionality to an otherwise general solution procedure: a kind of client-server structure. They compare it to other designs, such as passing user-supplied functions to the general procedure.

In summary: the literature on numerical software contains several descriptions of general designs that have turned out useful in that context. While they are not called *design patterns*, they have all the characteristics. The main qualitative difference is that the implementation language, FORTRAN 77, offers less support to hide some of the details.

Conclusion

It may require a bit of imagination to see how *design patterns* can be applied in a Fortran program, but the main obstacle is one of terminology: *design patterns* are usually described in the context of "typical" object-oriented languages as C++ and Java. The terms that abound are: classes, inheritance, polymorphism and the like, terms which are not often used in the Fortran literature. Another handicap to understanding the role of *design patterns* is that they are often presented without the context in which they would be useful. I myself still have difficulty identifying the underlying *design pattern* in a piece of software, when there is no hint as to which has been used.

I hope this article will serve as a beginning for appreciating *design patterns*. I have tried to formulate the patterns in terms a Fortran programmer is familiar with and it should be possible to better understand the publications on *design patterns,* such as *Design Patterns Explained*, a book I highly recommend – it gave me the inspiration to write this article.

I wish to thank Richard Maine and Michael Metcalf for proof-reading an incomplete version of this article. Their comments and textual suggestions have helped to improve it.

References

C.J. Lawson, R.J. Hanson, D. Kincaid and F.T. Krogh (1979)
Basic Linear Algebra Subprograms for FORTRAN usage
ACM Trans. Math. Soft. volume 5 (1979), pp. 308-323
(See also: http://www.nag.co.uk/nagnews/nagping/np003a2.pdf)

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

K. Radhakrishnan and A.C. Hindmarsh (1993)
Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations
NASA, Reference Publication 1327, Lawrence Livermore National Laboratory
Report UCRL-ID-113855
http://gltrs.grc.nasa.gov/cgi-bin/GLTRS/browse.pl?2003/RP-1327.html

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