Introducing Object-Oriented Concepts into GSI

Enhancements are now being made to the Grid-point Statistical Interpolation (GSI) data assimilation system to expand its capabilities and open the way for broadening the scope of its applications. These represent a starting point for the so-called *GSI refactoring*, which is to take shape as part of the Joint Effort for Data-assimilation Integration (JEDI) project coordinated by Thomas Auligné and Yannick Trémolet of the JCSDA.

Our initial contribution amounts to introducing object-oriented concepts to: (1) improve the GSI handling and expandability of the various observation types associated with the forward observation operators (FOO), (2) generalize the interface to the background (guess) states interpolations needed by the FOO, and (3) provide aframe work to allow for addition of user-specific subcomponents without need of changes to the actual GSI software. These implementations follow a bottom-up strategy so the refactored software never loses operability, thus remaining ready for use all along in currently supported GSI applications. A top-down approach to develop a Unified Forward Operator is concurrently underway, benefiting from contributions from many other collaborators; this, however, is not part of the present discussion.

Concepts of object-oriented (OO) programming have been around for a while and are supported by languages such as C++, Java, and many others. In general, OO programming enforces three main concepts: (1) abstraction; (2) encapsulation; and (3) polymorphism. From a scientific developer's point of view, *abstraction* allows for high-level code components to look simple, mimicking generic algorithmic concepts, similar to using mathematical symbols in equations; *encapsulation* allows the symbols to be defined in full details with separate expressions; and *polymorphism* allows for use of a single abstract entity with generic interfaces to represent specific entities of different types, while simultaneously managing encapsulated implementation differences between types as extensions. Fortran, the programming language of most codes used by our community, and in particular GSI, has slowly incorporated OO functionalities: Fortran 90 has supported abstraction and encapsulation since its initials tages. More recently, Fortran 2003 and 2008 bring in polymorphism, thus expanding the OO capabilities in the language.

The path we chose to introduce advanced programming concepts in GSI represents a mild version of refactoring, where, to a large extent, the familiar code remains recognizable. What follows provides examples of how modifications are being made to GSI's observation handling, guess interpolators, and hooks to user-specific components.

Extensible Observation Types

At the time of this writing, the official release of GSI incorporates a polymorphic observation operator. At its initial stage, polymorphism has been introduced without strict encapsulation (see below). This has been done to allow users to familiarize themselves with the code changes

before further changes take place, which will involve considerable shuffling of software. The original GSI implementation of the ability to handle multiple observation types can be labeled *procedural*, since the code is grouped by functionality instead of datatypes. This is illustrated in Figure 1, where *obsmod* controls initialization, finalization, and referencing to all observation types, whereas its "methods" (not treated as such in original code) are placed elsewhere and remain separated from other methods of the same type: the nonlinear observation operators (setupX), their linearized counter parts (intX), their conjugate-gradient step calculations (stpX), and I/O-related procedures are placed independently from the type they relate to, namely, *X_ob_type*.
[Fig 1 near here please]

In an OO world, these procedures should be thought of as methods of a particular type (object). Each self-contained type controls its own methods. For example, the nonlinear and linearized opēratōrs related to the radiance observation type should be among the methods controlled by the radiance type. There is also an *abstract* observation type that represents any concrete observation type under consideration of a generic algorithm implementation. A generic implementation then dispatches to concrete type-bound-procedures on the fly, at run-time. A schematic illustration of a modular encapsulation of the methods of each type into separate components is seen in Figure 2. Notice the horizontal arrangement of the blocks in this figure compared the vertical arrangement of those in Figure 1.
[Fig 2 near here please]

This particular restructuring of the observation types reveals the similarities among different types. More important, modularization of the types allows for simplification of the writing of many of them by exploiting their similarities and letting the compiler create the contents of similar required types on the fly. This is done in Fortran by defining a type as an extension of another existing type. Figure 3 provides code snippets illustrating this case. Examination of the code reveals that the methods associated with observation types pm2.5 (top) and pm10 (bottom), which are aerosol-like observations related with different particle sizes, are largely the same. Therefore, the type for pm10 (bottom right) can simply extend the type for pm2.5, with minor differences between the two being accommodated by creating exceptions within the context of pm10 (not shown).
[Fig 3 near here please]

Interface to Guess Interpolation
Use of polymorphism concepts can also be applied to design a flexible (general) handling of the guess (background) interpolations required by the observation simulator (part of the FOO that converts the background to the observable). Just as in the rewrite of the observation operators in modular OO-like framework, the guess interpolations can be handled (1) by development of an interface layer to virtually support generic (abstract) GSI guess state interpolation applications, and (2) by recognizing that every interpolator can be implemented as an extension of corresponding abstract interpolators.

At this stage, the discussion that follows is part of a prototype under test at NASA's Global Modeling and Assimilation Office (NASA/GMAO). The snippets of code shown below are not final, and consideration is taking place with feedback from other members of the JEDI team to establish consensus on the approach presented here. Nevertheless, discussions have been rather positive and favorable toward what follows, so we feel confident that only minor adjustments will be necessary.

A key component to the present design of an abstract interface is the realization that things like the grid, resolution, partition, choice of variables, and specific partition information related to the state vector should not appear in the interface. This makes the interface rather generic. The only things exchanged between the calling programs and the underlying interfaces are physical quantities, such as variables, 2D or 3D spatial locations, time of observations, and in certain cases, the processor identifiers. A code snippet of possible calls from within a general (setupXYZ) nonlinear GSI operator is given in Figure 4. The abstraction layer is designed to honor all current GSI use-cases and functionalities. Very few assumptions are needed about the guess-state for the interpolations, so the approach here leaves sufficient room for expansion of functionality.

[Fig 4 near here please]

User-Specific Subcomponents

There are multiple examples in GSI where its internal components maybe specific to a particular application. One of the simplest examples is the placement of timers. A particular user, or group of users, might prefer using their own timer utility library to assess performance of internal subcomponents of the code. GSI is presently enabled with calls to start, end, and summarize timings obtained by user-provided timing library. In its default use by the National Centers for Environmental Prediction Environmental Modeling Center (NCEP/EMC), a timing library is not typically provided when the executable of GSI is created; others, such as NASA/GMAO, may provide one or another library that automatically supplies timing information after each execution of GSI.

To avoid unnecessary code localization, presently, a default do-nothing stub timer is provided in GSI through a set of Fortran implicit interfaces, so that without GSI code changes, each user application can choose to swap at the linkage stage with a functioning timer. The same concept is also used by several GSI stubs as a crude approach of managing complex user-specific extensions, including a stub of tangent-linear and adjoint models for 4DVar, and a stub of ensemble background fields, etc. With true polymorphism, stub-swapping by users can be avoided by bringing minor enhancements to the current stub mechanism, where user-specified stub replacements can be implemented and configured as formal extensions.

In Summary

The bottom-up, initial steps toward refactoring of GSI discussed above are expected to facilitate maintainability, extensibility, and scalability of the software. The approach turns GSI into a code

that is as object-oriented as any Fortran code can be at this stage in the language support to OO concepts. Abstraction and encapsulation are expected to be the main contributors to enhanced maintainability. The polymorphic implementation of some of its main functionalities will contribute to facilitate extensibility. An easier-to-maintain and easier-to-extend software will facilitate identification of computational bottlenecks and consequent improvement in software scalability.

Acknowledgements: We would like to thank the GSI Committee for supporting the introduction of the concepts discussed here in the GSI software. We thank Thomas Aulign´e for his energetic enthusiasm to refactor GSI using modern software concepts. And we thank Yannick Tr´emolet for discussions related to the implementation of a generalized interface to guess-interpolators.

Jing Guo[1] and Ricardo Todling, NASA Global Modeling and Assimilation Office
[1]Additional Affiliation: Science Systems and Applications, Inc., Lanham, MD

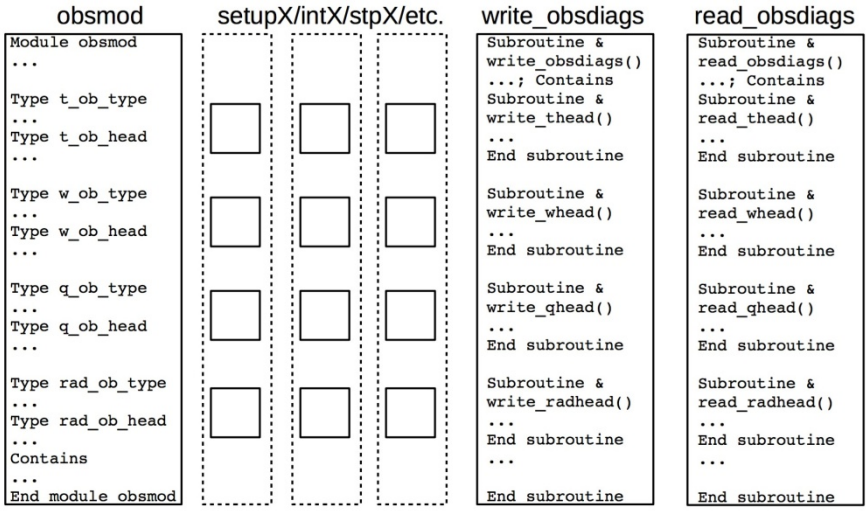


Figure 1. Schematic view of original *procedural* coding of components related to the observation operators in GSI. Left-most rectangle (obsmod): initialization, finalization and general management of various observation types; middle rectangles: non-linear (setupX) and linearized observation operators (intX), and conjugate-gradient step calculations (stpX) for each observation type; two right-most rectangles: writes and reads of the observation types. Note the vertical (procedural) organization separates an observation type from its methods.

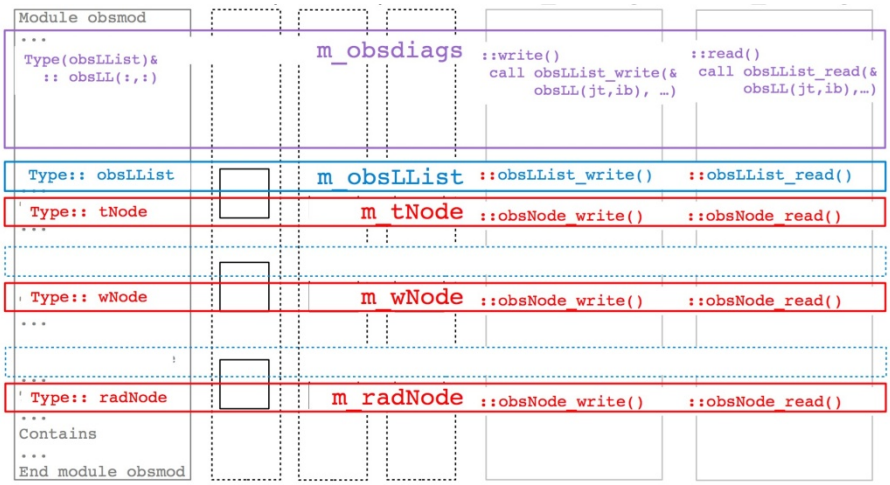


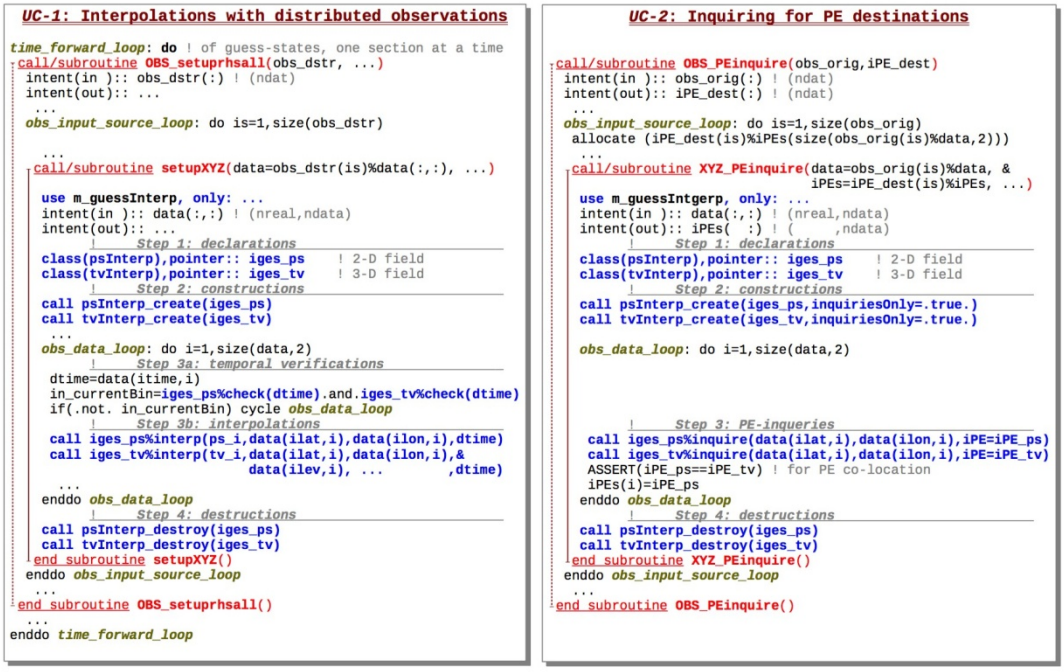
Figure 2. Schematic view of current *polymorphic* implementation related to the observation operators in GSI. In this new construct, each given observation type (e.g., the temperature operator tNode) controls its own operations: initialization, finalization, read, write (highlighted in colored horizontal blocks), with additional procedures (setupX, intX, stpX) to be encapsulated soon in the future.

```
module m_pm2_5Node ! (1) start-from-scratch approach
! abstract: class-module of in-situ pm-2.5
  use obsmod, only: obs_diag
  use kinds , only: i_kind,r_kind
  use m_obsNode, only: obsNode
  implicit none
  private
  public:: pm2_5Node          ! data structure
  public:: pm2_5Node_typecast ! cast obsNode as pm2_5Node
[...]
  type,extends(obsNode):: pm2_5Node
    type(obs_diag),pointer:: diags => NULL()
    real(r_kind)   :: res     ! residual
    real(r_kind)   :: err2    ! obs error squared
    real(r_kind)   :: raterr2 ! ratio of obs error squared
    real(r_kind)   :: wij(8)  ! grid interpolation weights
    integer(i_kind):: ij(8)   ! grid references
    real   (r_kind):: dlev    ! vertical grid index

  contains  !---- type-bound-procedures ------------------
    procedure::   mytype      ! implemented here

    procedure::   setHop      ! implemented here
    procedure::    xread      ! implemented here
    procedure::   xwrite      ! implemented here
    procedure::  isvalid      ! implemented here
    procedure:: gettlddp      ! implemented here

    ! procedure, nopass:: header_read     ! from obsNode
    ! procedure, nopass:: header_write    ! from obsNode
    ! procedure::         init            ! from obsNode
    ! procedure::         clean           ! from obsNode
  end type pm2_5Node
contains; [...]
end module m_pm2_5Node
```

```
module m_pm10Node ! (2) copy-then-edit approach, from pm2_5
! abstract: class-module of in-situ pm-10
  use obsmod, only: obs_diag
  use kinds , only: i_kind,r_kind
  use m_obsNode, only: obsNode
  implicit none
  private
  public:: pm10Node          ! data structure
  public:: pm10Node_typecast ! cast obsNode as pm10Node
[...]
  type,extends(obsNode):: pm10Node
    type(obs_diag),pointer:: diags => NULL()
    real(r_kind)   :: res     ! residual
    real(r_kind)   :: err2    ! obs error squared
    real(r_kind)   :: raterr2 ! ratio of obs error squared
    real(r_kind)   :: wij(8)  ! grid interpolation weights
    integer(i_kind):: ij(8)   ! grid references
    real   (r_kind):: dlev    ! vertical grid index

  contains  !---- type-bound-procedures ------------------
    procedure::   mytype      ! implemented here

    procedure::   setHop      ! implemented here
    procedure::    xread      ! implemented here
    procedure::   xwrite      ! implemented here
    procedure::  isvalid      ! implemented here
    procedure:: gettlddp      ! implemented here

    ! procedure, nopass:: header_read     ! from obsNode
    ! procedure, nopass:: header_write    ! from obsNode
    ! procedure:: init                    ! from obsNode
    ! procedure:: clean                   ! from obsNode
  end type pm10Node
contains; [...]
end module m_pm10Node
```

```
module m_pm10Node ! (3) type-extends approach, extends(pm2_5)
! abstract: class-module of in-situ pm-10


  use m_pm2_5Node, only: pm2_5Node
  implicit none
  private
  public:: pm10Node          ! data structure
  public:: pm10Node_typecast ! cast obsNode as pm10Node
[...]
  type,extends(pm2_5Node):: pm10Node









  contains  !---- type-bound-procedures ------------------
    procedure::   mytype      ! implemented here

    ! procedure::   setHop      ! from pm2_5Node
    ! procedure::    xread      ! from pm2_5Node
    ! procedure::   xwrite      ! from pm2_5Node
    ! procedure::  isvalid      ! from pm2_5Node
    ! procedure:: gettlddp      ! from pm2_5Node

    ! procedure, nopass:: header_read     ! from obsNode
    ! procedure, nopass:: header_write    ! from obsNode
    ! procedure:: init                    ! from obsNode
    ! procedure:: clean                   ! from obsNode
  end type pm10Node
contains; [...]
end module m_pm10Node
```

Figure 3. Within OO concepts, one can recognize similarities among different components, thus allowing for code reuse, improvement for readability, and easier management of differences for *extensibility*. As an example, the figure compares the type controlling observations from the abstract to pm2.5 (top), and then to pm10 (bottom). Blue-colored headers of three panels highlight different approaches to create a new type. Exploitation of their similarities means simplification of, say, pm10 by turning it into an extension of pm2.5 (bottom right); in other words, using pm2.5 as a template created on the fly by the compiler.



UC-1: Interpolations with distributed observations

```
time_forward_loop: do ! of guess-states, one section at a time
 call/subroutine OBS_setuprhsall(obs_dstr, ...)
  intent(in ):: obs_dstr(:) ! (ndat)
  intent(out):: ...
   ...
  obs_input_source_loop: do is=1,size(obs_dstr)

    ...
   call/subroutine setupXYZ(data=obs_dstr(is)%data(:,:), ...)

    use m_guessInterp, only: ...
    intent(in ):: data(:,:) ! (nreal,ndata)
    intent(out):: ...
          !     Step 1: declarations
    class(psInterp),pointer:: iges_ps    ! 2-D field
    class(tvInterp),pointer:: iges_tv    ! 3-D field
          !     Step 2: constructions
    call psInterp_create(iges_ps)
    call tvInterp_create(iges_tv)
     ...
    obs_data_loop: do i=1,size(data,2)
          !     Step 3a: temporal verifications
     dtime=data(itime,i)
     in_currentBin=iges_ps%check(dtime).and.iges_tv%check(dtime)
     if(.not. in_currentBin) cycle obs_data_loop
          !     Step 3b: interpolations
     call iges_ps%interp(ps_i,data(ilat,i),data(ilon,i),dtime)
     call iges_tv%interp(tv_i,data(ilat,i),data(ilon,i),&
                         data(ilev,i), ...        ,dtime)
      ...
    enddo obs_data_loop
          !     Step 4: destructions
    call psInterp_destroy(iges_ps)
    call tvInterp_destroy(iges_tv)
   end subroutine setupXYZ()
  enddo obs_input_source_loop
   ...
 end subroutine OBS_setuprhsall()
  ...
enddo time_forward_loop
```

UC-2: Inquiring for PE destinations

```
call/subroutine OBS_PEinquire(obs_orig,iPE_dest)
 intent(in ):: obs_orig(:) ! (ndat)
 intent(out):: iPE_dest(:) ! (ndat)
  ...
 obs_input_source_loop: do is=1,size(obs_orig)
  allocate (iPE_dest(is)%iPEs(size(obs_orig(is)%data,2)))
   ...
 call/subroutine XYZ_PEinquire(data=obs_orig(is)%data, &
                               iPEs=iPE_dest(is)%iPEs, ...)
  use m_guessIntgerp, only: ...
  intent(in ):: data(:,:) ! (nreal,ndata)
  intent(out):: iPEs(  :) ! (      ,ndata)
        !     Step 1: declarations
  class(psInterp),pointer:: iges_ps    ! 2-D field
  class(tvInterp),pointer:: iges_tv    ! 3-D field
        !     Step 2: constructions
  call psInterp_create(iges_ps,inquiriesOnly=.true.)
  call tvInterp_create(iges_tv,inquiriesOnly=.true.)

  obs_data_loop: do i=1,size(data,2)




        !     Step 3: PE-inqueries
   call iges_ps%inquire(data(ilat,i),data(ilon,i),iPE=iPE_ps)
   call iges_tv%inquire(data(ilat,i),data(ilon,i),iPE=iPE_tv)
   ASSERT(iPE_ps==iPE_tv) ! for PE co-location
   iPEs(i)=iPE_ps
  enddo obs_data_loop
        !     Step 4: destructions
  call psInterp_destroy(iges_ps)
  call tvInterp_destroy(iges_tv)
 end subroutine XYZ_PEinquire()
 enddo obs_input_source_loop
  ...
end subroutine OBS_PEinquire()
```

Figure 4. With abstraction *m_guessInterp*, (UC-1; left) is a GSI use-case of guess interpolators with distributed observations, while (UC-2; right) is a GSI use-case of inquiring for processor destinations with respect to guess interpolators. Both use-cases are schematically shown in the code snippets of planned implementations. These are very close to the current GSI implementation, but generic. Blue-colored statements highlight complete life cycles of individual interpolators created by m_guessInterp, with few assumptions about concrete implementation of specific interpolators.