Development of Polymorphic GSI
(How to Add a New Observation Type)

J. Guo and R. Todling, NASA/GMAO
Mar. 15, 2017, at NASA/GMAO

- Recent Object-Oriented Changes
- Implementing a New Observation Type
- Upcoming Developments

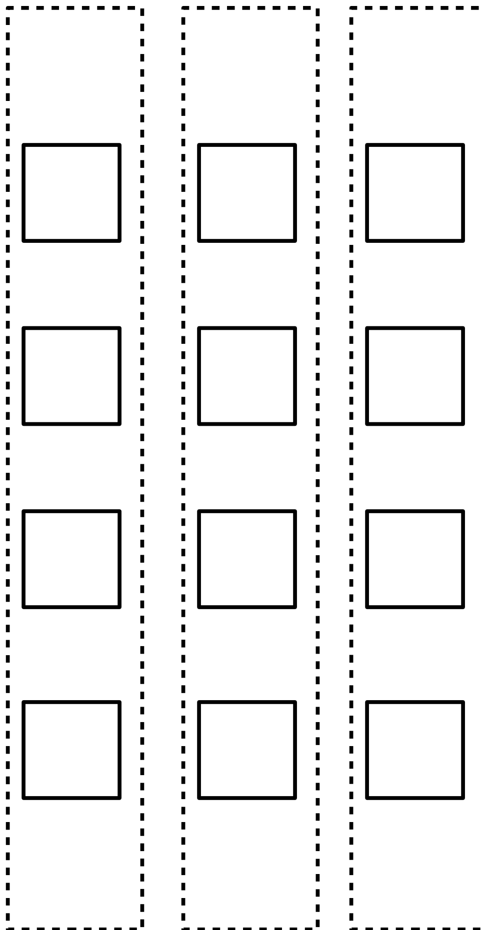
Recent Object-Oriented Changes

Original Procedural Code Structure

obsmod

```
Module obsmod
...

Type t_ob_type
...
Type t_ob_head
...

Type w_ob_type
...
Type w_ob_head
...

Type q_ob_type
...
Type q_ob_head
...

Type rad_ob_type
...
Type rad_ob_head
...
Contains
...
End module obsmod
```

setupX/intX/stpX/etc.



write_obsdiags

```
Subroutine &
write_obsdiags()
...; Contains
Subroutine &
write_thead()
...
End subroutine

Subroutine &
write_whead()
...
End subroutine

Subroutine &
write_qhead()
...
End subroutine

Subroutine &
write_radhead()
...
End subroutine
...

End subroutine
```

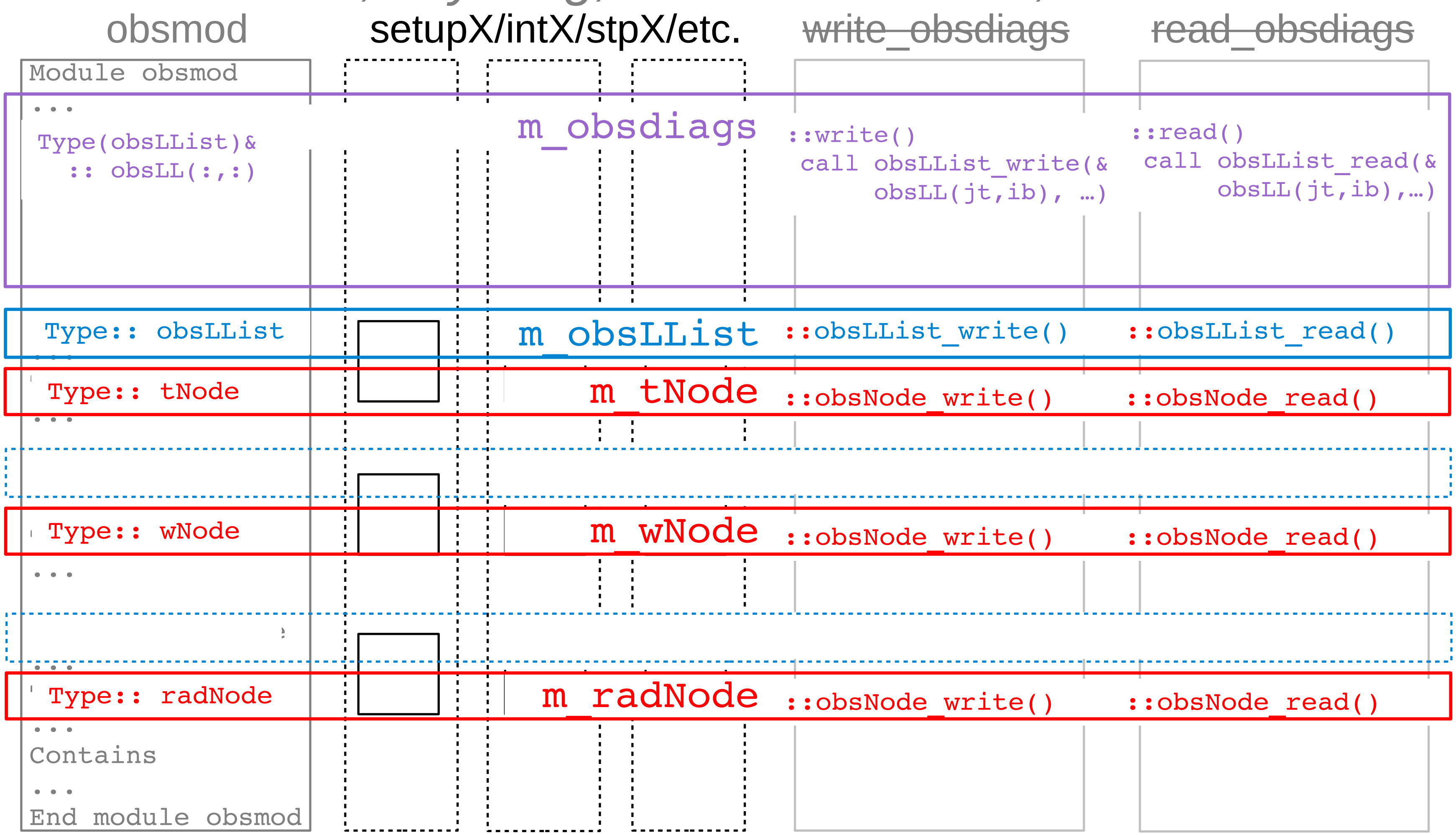
read_obsdiags

```
Subroutine &
read_obsdiags()
...; Contains
Subroutine &
read_thead()
...
End subroutine

Subroutine &
read_whead()
...
End subroutine

Subroutine &
read_qhead()
...
End subroutine

Subroutine &
read_radhead()
...
End subroutine
...

End subroutine
```

Recent Object-Oriented Changes

Procedural vs. Object-Oriented

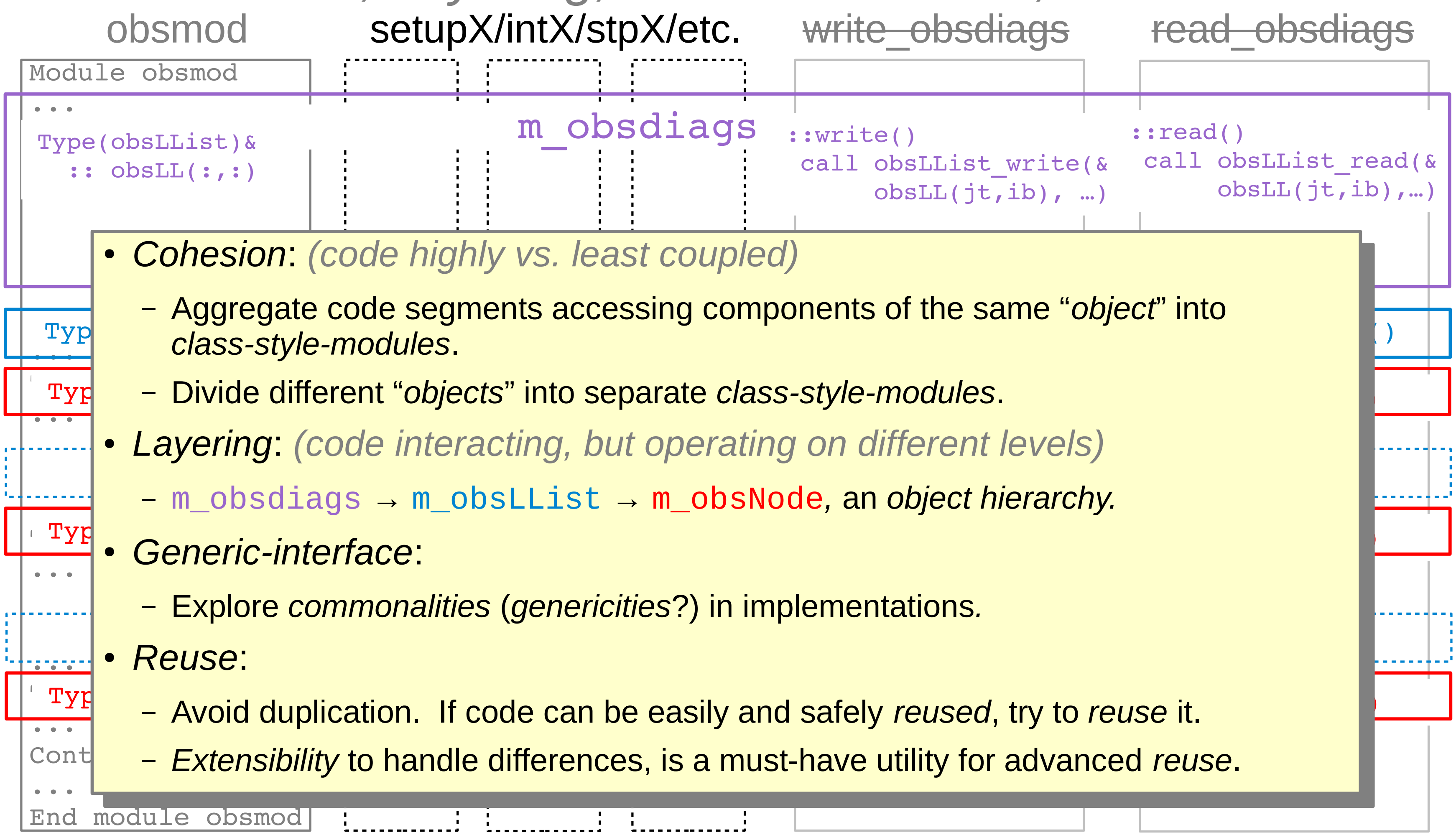
– Cohesion, Layering, Generic-interface, and Reuse –



Recent Object-Oriented Changes

Procedural vs. Object-Oriented

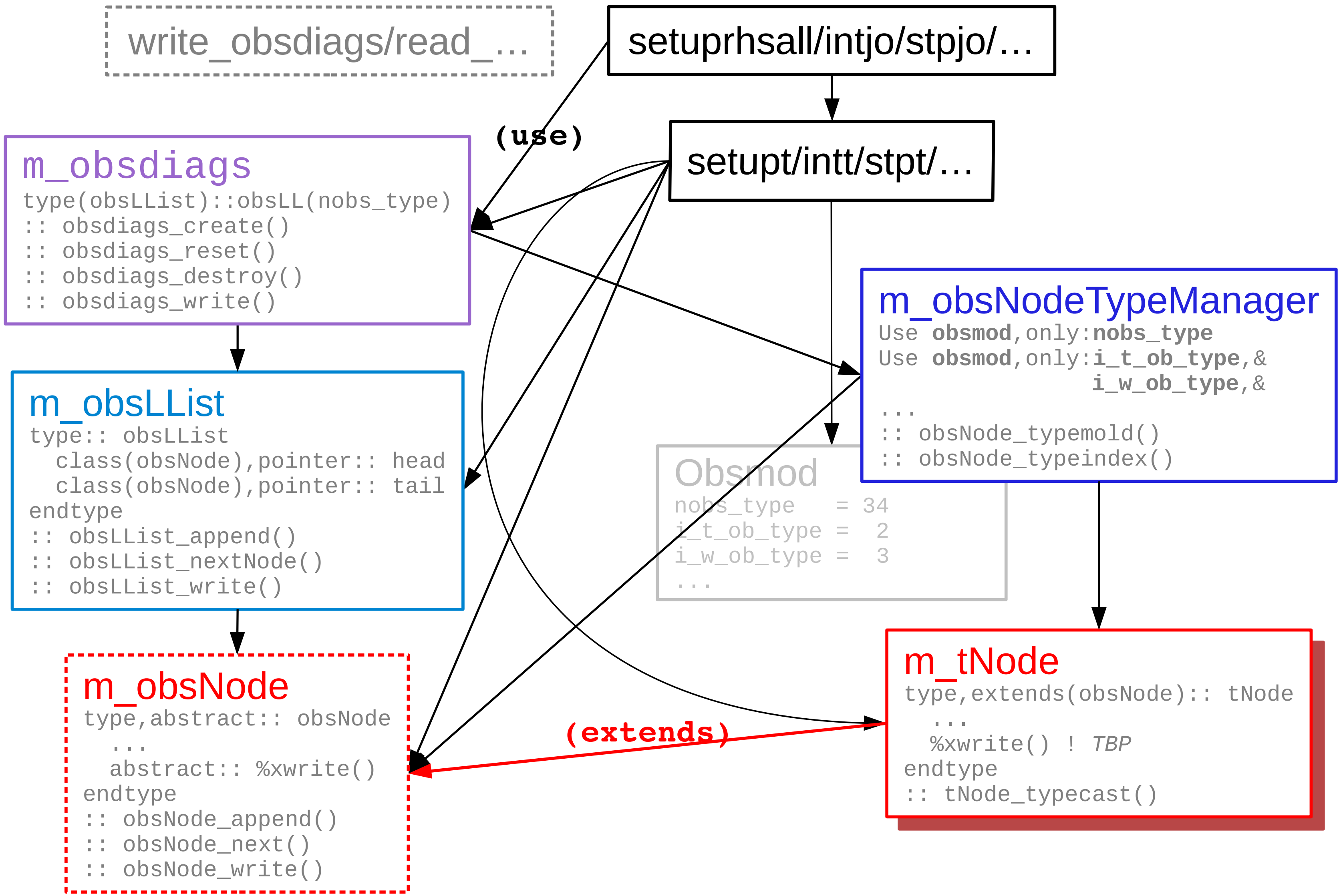
– Cohesion, Layering, Generic-interface, and Reuse –



- *Cohesion*: *(code highly vs. least coupled)*
  - Aggregate code segments accessing components of the same "*object*" into *class-style-modules*.
  - Divide different "*objects*" into separate *class-style-modules*.
- *Layering*: *(code interacting, but operating on different levels)*
  - `m_obsdiags` → `m_obsLList` → `m_obsNode`, an *object hierarchy.*
- *Generic-interface*:
  - Explore *commonalities* (*genericities*?) in implementations*.*
- *Reuse*:
  - Avoid duplication. If code can be easily and safely *reused*, try to *reuse* it.
  - *Extensibility* to handle differences, is a must-have utility for advanced *reuse*.

Recent Object-Oriented Changes

In a Module Hierarchy View



Implementing a New Observation Type

This is NOT

- It is NOT about injecting a new *source of observations*,
  - neither new `read_obs()` routine, nor new guess-vector or control-vector variable.
- It is NOT about some new algorithm.
  - Algorithms are not the subject.
- It is NOT about layers of wrappers to shoehorn your code into a framework.
- It is NOT about aspects of Fortran that most of us are fairly familiar with.
- It is NOT a finalized implementation solution.

This is

- It is about steps to add a new *observation type* under the new code structure,
  - a GSI internal data structure, supporting observation operators.
- It is about code *restructuring*,
  - with algorithms, styles, even debris preserved.
- It is about exploring natural hierarchy and genericity in existing implementation,
  - connecting abstractions directly to variables, to improve maintainability and extensibility.
- It is about applying advanced Fortran features in precision,
  - most from F90s, and some from F2Ks.
- It is an incremental, bottom-up development, through refactoring.
  - It is imperfect, but solid and important.
  - Examples, experience, expertise, engineering requirements? We will learn together.

Implementing a New Observation Type

Before and After

Before

1. Add your new `type(anew_ob_type)`, in module `obsmod.F90`.
2. Then
   - Add enumerator `i_anew_ob_type=35`, in `obsmod`;
   - Increase count `nobs_type`, in `obsmod`.
3. Add corresponding declarations, `allocate()`, and `deallocate()` operations, in `obsmod`.
4. Add a new entry, in `setupyobs.f90`.
5. Create a new `setupanew.f90`; Add its call to `setuprhsall.f90`.
6. Create a new `intanew.f90`; Add its call to `intjo.f90`.
7. Create a new `stpanew.f90`; Add its call to `stpjo.f90`.
8. Add new I/O routines to `read_obsdiags.F90` and `write_obsdiags.F90`.

This implementation

1. Add a new module `m_anewNode.F90` for your new `type(anewNode)`; Complete required module interfaces and *type-bound-procedures*.
2. Then
   - Add enumerator `i_anew_ob_type=35`, in `obsmod`;
   - Increase count `nobs_type`, in `obsmod`;
   - Support it in `m_obsNodeTypeManager.F90`.
3. Add alias `anewhead` in `m_obsdiags.F90`.
4. Add a new entry, in `m_obsHeadBundle.F90`.
5. Create a new `setupanew.f90`; Add its call to `setuprhsall.f90`.
6. Create a new `intanew.f90`; Add its call to `intjo.f90`.
7. Create a new `stpanew.f90`; Add its call to `stpjo.f90`.
8. Nothing.

- Little has been changed in the major steps of adding `anew_ob_type` (`anewNode`), *except the I/O part*.

Implementing a New Observation Type
Before and After

<u>*Before*</u>

1. Add your new `type(anew_ob_type)`, in module `obsmod.F90`.
2. Then
   - Add enumerator `i_anew_ob_type=35`, in `obsmod`;
   - Increase count `nobs_type`, in `obsmod`.
3. Add corresponding declarations, `allocate()`, and `deallocate()` operations, in `obsmod`.
4. Add a new entry, in `setupyobs.f90`.
5. Create a new `setupanew.f90`; Add its call to `setuprhsall.f90`.
6. Create a new `intanew.f90`; Add its call to `intjo.f90`.
7. Create a new `stpanew.f90`; Add its call to `stpjo.f90`.
8. Add new I/O routines to `read_obsdiags.F90` and `write_obsdiags.F90`.

<u>*This implementation*</u>

1. Add a new module `m_anewNode.F90` for your new `type(anewNode)`; Complete required module interfaces and *type-bound-procedures*.
2. Then
   - Add enumerator `i_anew_ob_type=35`, in `obsmod`;
   - Increase count `nobs_type`, in `obsmod`;
   - Support it in `m_obsNodeTypeManager.F90`.
3. Add alias `anewhead` in `m_obsdiags.F90`.
4. Add a new entry, in `m_obsHeadBundle.F90`.
5. Create a new `setupanew.f90`; Add its call to `setuprhsall.f90`.
6. Create a new `intanew.f90`; Add its call to `intjo.f90`.
7. Create a new `stpanew.f90`; Add its call to `stpjo.f90`.
8. Nothing.

- Little has been changed in the major steps of adding `anew_ob_type` (`anewNode`), <u>*except the I/O part*</u>.
- So developers won't have to worry about their on going works in `setup`/`int`/`stp` routines <u>*for now*</u>.

Implementing a New Observation Type
Before and After

Before

1. Add your new `type(anew_ob_type)`, in module `obsmod.F90`.
2. Then
   - Add enumerator `i_anew_ob_type=35`, in `obsmod`;
   - Increase count `nobs_type`, in `obsmod`.
3. Add corresponding declarations, `allocate()`, and `deallocate()` operations, in `obsmod`.
4. Add a new entry, in `setupyobs.f90`.
5. Create a new `setupanew.f90`; Add its call to `setuprhsall.f90`.
6. Create a new `intanew.f90`; Add its call to `intjo.f90`.
7. Create a new `stpanew.f90`; Add its call to `stpjo.f90`.
8. Add new I/O routines to `read_obsdiags.F90` and `write_obsdiags.F90`.

This implementation

1. Add a new module `m_anewNode.F90` for your new `type(anewNode)`; Complete required module interfaces and *type-bound-procedures*.
2. Then
   - Add enumerator `i_anew_ob_type`~~=35, in obsmod;~~
     ~~Increase count nobs_type, in obsmod;~~
     and support it, in `m_obsNodeTypeManager.F90`.
3. ~~Add alias anewhead in m_obsdiags.F90.~~
4. ~~Add a new entry, in m_obsHeadBundle.F90.~~
5. Create a new `setupanew.f90`; Add its call to `setuprhsall.f90`.
6. Create a new `intanew.f90`; Add its call to `intjo.f90`.
7. Create a new `stpanew.f90`; Add its call to `stpjo.f90`.
8. Nothing.

- Little has been changed in the major steps of adding `anew_ob_type` (`anewNode`), *except the I/O part*.
- So developers won't have to worry about their on going works in `setup/int/stp` routines *for now*.
- With additional cleaning up, some transitional solutions for smoother code evolution *will be removed*.

Implementing a New Observation Type

Code Snippet: `m_obsNode.F90`, “*the Base Type*”

```
module m_obsNode ! The base type
  use [...]
  implicit none; private ! Except

!---- (1) abstract data type definition with TBPs ---------
public:: obsNode ! data structure
  type, abstract:: obsNode
    class(obsNode),pointer:: llpoint => NULL()
    Logical     :: luse      ! data's local "ownership"
    real(r_kind):: elat,elon ! obs. lat-lon in degrees
    [...]
  contains
      !---- TBPs must be defined by each extension ------
    procedure(intrfc_mytype_),deferred:: mytype ! type name
    procedure(intrfc_setHop_),deferred:: setHop ! re-constr
    procedure(intrfc_xwrite_),deferred:: xwrite ! write ext
    [...]
      !---- TBPs may be redefined by extensions ---------
    procedure,nopass:: header_read  => obsHeader_read_
    procedure,nopass:: header_write => obsHeader_write_
    [...]
  end type obsNode

!---- (2) public interfaces for a polymorphic node --------
public:: obsNode_next     ! next => obsNode_next(here)
public:: obsNode_append   ! call obsNode_append(here,next)
public:: obsNode_islocal  ! if(obsNode_islocal(here)) then..
public:: obsNode_isluse   ! if(obsNode_isluse(here)) then..
public:: obsNode_setluse  ! call obsNode_setluse(here)
public:: obsNode_read     ! call obsNode_read(here,lu,...)
public:: obsNode_write    ! call obsNode_write(here,lu,...)

  interface obsNode_next  ; module procedure next_   ; end..
  interface obsNode_append; module procedure append_; end..
  [...]
```

```
!---- (3) abstract-interfaces defining defered TBPs -------
[...]
  abstract interface
    subroutine intrfc_xwrite_(aNode,junit,jstat)
      use kinds,only: i_kind
      import:: obsNode
      implicit none
      class(obsNode), intent(in):: aNode
      integer(kind=i_kind), intent(in ):: junit
      integer(kind=i_kind), intent(out):: jstat
    end subroutine intrfc_xwrite_
  end interface

contains
!---- (4) implementations of all actual prodecures --------
[...]
  subroutine obsHeader_write_(junit,mobs,jwrite,jstat)
    use kinds,only: i_kind
    implicit none
    integer(i_kind),intent(in ):: junit  ! output unit
    integer(i_kind),intent(in ):: mobs   ! record count
    integer(i_kind),intent(in ):: jwrite ! obstype enum
    integer(i_kind),intent(out):: jstat  ! iostat
    write(junit,iostat=jstat) mobs,jwrite
  end subroutine obsHeader_write_
[...]
  function next_(aNode) result(here_) ! => aNode%llpoint
    implicit none
    class(obsNode),pointer:: here_
    class(obsNode),target,intent(in):: aNode ! non-Null
    here_ => aNode%llpoint
  end function next_
[...]
end module m_obsNode
```

Implementing a New Observation Type

Code Snippet: obsNode → pm2_5Node

```
module m_obsNode !  The base type
! abstract: class-module of any generic obs.
  use kinds, only: i_kind,r_kind
  implicit none
  private
  public:: obsNode        ! data structure
!---- node operations
  public:: obsNode_next   ! nextNode => obsNode_next(here)
  public:: obsNode_append ! call obsNode_append(here,next)
[...]
  type,abstract:: obsNode
    class(obsNode),pointer:: llpoint=>NULL() ! %next
    logical         :: luse ! local "ownership"
    real(r_kind)    :: time ! obs. time offset in sec.
    real(r_kind)    :: elat ! latitude in degree
    real(r_kind)    :: elon ! longitude in degree
    integer(i_kind)::  idv ! device ID, "is" in "do is=1,"
    integer(i_kind)::  iob ! initial obs. ID, "ioid"

  contains  !---- type-bound-procedures -------------------
    procedure(intrfc_mytype_),deferred:: mytype  ! typename

    procedure(intrfc_setHop_),deferred:: setHop  ! ij, wij
    procedure(intrfc_xread_ ),deferred:: xread   ! read ..
    procedure(intrfc_xwrite_),deferred:: xwrite  ! write ..
    procedure(intrfc_isval..),deferred:: isvalid ! validate
    procedure(intrfc_gettl..),deferred:: gettlddp ! dotprod
            !----------- overridable TBP -----------------
    procedure,nopass:: header_read => obsHeader_read_
    procedure,nopass:: header_write=> obsHeader_write_
    procedure::             init   => init_
    Procedure::             clean => clean_
  end type obsNode
contains; [...]
end module m_obsNode
```

```
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
```

Implementing a New Observation Type

Code Snippet: pm2_5Node → pm10Node

```
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
```

Implementing a New Observation Type

Code Snippet: Generic and Type-bound Interfaces

```
module m_pm2_5Node !  Generic interfaces, not type-bound
[...]
  public:: pm2_5Node_typecast ! a named invocation
      ! my_pm2_5 => pm2_5Node_typecast(polyNode)  ! or
      ! my_pm2_5 => pm2_5Node_typecast(polyNode,my_pm2_5)
      interface pm2_5Node_typecast
        module procedure typecast______, &
                         typecast_mold_
      end interface

  public:: typecast           ! For generic invocation
      ! my_pm2_5 => typecast(polyNode,my_pm2_5)
      interface typecast
        module procedure typecast_mold_
      end interface
[...]
contains
function typecast______(aNode) result(ptr_)
  type(pm2_5Node), pointer:: ptr_
  class(obsNode) , pointer,intent(in):: aNode
  ptr_ => null(mold=ptr_)
  select type(aNode)
  type is(pm2_5Node)
    ptr_ => aNode
  end select
end function typecast______

function typecast_mold_(aNode,mold) result(ptr_)
  type(pm2_5Node), pointer:: ptr_
  class(obsNode) , pointer, intent(in):: aNode
  type(pm2_5Node),          intent(in):: mold
  Ptr_ => typecast______(aNode)
end function typecast_mold_
[...]
end module m_pm2_5Node
```

```
module m_pm2_5Node !  Type-bound interfaces
[...]
  public:: pm2_5Node          ! data structure
  type,extends(obsNode):: pm2_5Node
    [...]
  contains  !---- type-bound-procedures ------------------
    procedure::   mytype       ! print*,ob%mytype()
    procedure::   setHop       ! call ob%setHop()
    [...]
  end type pm2_5Node
[...]
contains
function mytype(self,about)
  character(len=:),allocatable:: mytype
  class(pm2_5Node), intent(in):: self
  character(len=*),optional,intent(in):: about

  mytype="[pm2_5Node]"
  if(.not.present(about)) return

  select case(about)
  case('cobstype')
    mytype="in-situ pm2_5 obs"
  [...]
end function mytype

subroutine setHop(self)
  use m_cvgridLookup,only: getw => cvgridLookup_getiw
  class(pm2_5Node),intent(inout):: self
  call getw(self%elat,self%elon,self%ij(1:4),self%wij(1:4))
  self%ij (5:8) = self%ij(1:4)
  self%wij(5:8) = 0.
end subroutine setHop
[...]
end module m_pm2_5Node
```

Implementing a New Observation Type

Code Snippet: `obsmod` → `m_obsNodeTypeManager`

```
module obsmod ! Now
! abstract:
  use kinds , only: i_kind
  implicit none
  private
  public:: nobs_type
  public::    i_ps_ob_type
  public::     i_t_ob_type
  public::     i_w_ob_type
  public::     i_q_ob_type
  public::   i_spd_ob_type
  [...]
  public:: i_pm2_5_ob_type
  [...]
  public::  i_pm10_ob_type
  [...]
  public::  i_anew_ob_type
[...]
! Declare types
  integer(i_kind),parameter::    i_ps_ob_type= 1
  integer(i_kind),parameter::     i_t_ob_type= 2
  integer(i_kind),parameter::     i_w_ob_type= 3
  integer(i_kind),parameter::     i_q_ob_type= 4
  integer(i_kind),parameter::   i_spd_ob_type= 5
  [...]
  integer(i_kind),parameter:: i_pm2_5_ob_type=21
  [...]
  integer(i_kind),parameter::  i_pm10_ob_type=33
  [...]
  integer(i_kind),parameter::  i_anew_ob_type=35

  integer(i_kind),parameter:: nobs_type=35 ! all ob.types

[...]
end module obsmod
```

```
module m_obsNodeTypeManager ! Now
! abstract: a one-stop-shop managing all obs-types
  use obsmod, only: nobs_type
  [...]
  use obsmod, only: iobsType_anew => i_anew_ob_type
  use m_anewNode, only: anewNode
  [...]
  implicit none
  private
  public :: nobs_type
  public :: obsNode_typeMold
    interface obsNode_typeMold; module procedure &
      index2vmold_; end interface
  public :: obsNode_typeIndex
    interface obsNode_typeIndex; module procedure &
      vmold2index_; end interface
[...]
  type(anewNode),target,save:: anew_mold
[...]

contains
function vmold2index_(mold) result(index_)
  integer:: index_
  class(obsNode),target,intent(in):: mold
  select type(mold)
  type is(anewNode); index_=iobsType_anew
[...]
function index2vmold_(i_obType) result(obsmold_)
  class(obsNode),pointer:: obsmold_
  integer,intent(in):: i_obType
  obsmold_ => null()
  select case(i_obType)
  case(iobsType_anew); obsmold_ => anew_mold
[...]
end module m_obsNodeTypeManager
```

Implementing a New Observation Type

Code Snippet: `obsmod` → `m_obsNodeTypeManager`, *as Enum?*

```
module obsmod ! Later
! abstract:
  use kinds , only: i_kind
  implicit none
  private
! public:: nobs_type
! public::    i_ps_ob_type
! public::     i_t_ob_type
! public::     i_w_ob_type
! public::     i_q_ob_type
! public::   i_spd_ob_type
  [...]
! public:: i_pm2_5_ob_type
  [...]
! public::  i_pm10_ob_type
  [...]
! public::  i_anew_ob_type
[...]
! Declare types
! integer(i_kind),parameter::    i_ps_ob_type= 1
! integer(i_kind),parameter::     i_t_ob_type= 2
! integer(i_kind),parameter::     i_w_ob_type= 3
! integer(i_kind),parameter::     i_q_ob_type= 4
! integer(i_kind),parameter::   i_spd_ob_type= 5
  [...]
! integer(i_kind),parameter:: i_pm2_5_ob_type=21
  [...]
! integer(i_kind),parameter::  i_pm10_ob_type=33
  [...]
! integer(i_kind),parameter::  i_anew_ob_type=35

! integer(i_kind),parameter:: nobs_type=35 ! no. obs.types

[...]
end module obsmod
```

```
module m_obsNodeTypeManager ! Later
! abstract: a one-stop-shop managing all obs-types
  [...]
  use m_anewNode, only: anewNode
  implicit none
  private
  public:: nobs_type
  public:: obsType_lbound,obsType_ubound

  public:: obsNode_typemold
  public:: obsNode_typeindex
[...]
  public:: iobsType_anew
[...]
  enum, bind(C)
    enumerator:: floor_ = 0

    enumerator:: iobsType_ps ! ps
    enumerator:: iobsType_t  ! upper air t_virtual
    enumerator:: iobsType_w  ! upper air wind (u,v)
    [...]
    enumerator:: iobsType_anew  ! a-new obs. type

    enumerator:: ceiling_
  end enum

  integer(i_kind),parameter:: nobs_type=ceiling_-floor_-1
  integer(i_kind),parameter:: obsType_lbound=floor_+1
  integer(i_kind),parameter:: obsType_ubound=ceiling_-1
  integer(i_kind),parameter:: obsType_ikind=kind(floor_)
[...]
  type(anewNode),target,save:: anew_mold
contains
[...]
end module m_obsNodeTypeManager
```

Upcoming Developments

- Cleaning and tuning
  - Reconcile changes by the recent merge into GSI trunk.
  - Improve in details and precision, of earlier implementations of this work.
  - Reduce some obs-types as extensions of other obs-types.
  - Divide labor of `obsmod` further, into each obs-type and a type-manager.
  - Remove transitional aliases, `yobs` or `obsHeadBundle`, `thead`, `ttail`, …
- Restructuring of more complicated code
  - Fold `type::obs_diag` into `type::obsNode`.
  - Restructure `int()` and `stp()`,
    - Layer out linked-list operations,
    - Implement linear operators, `%TLop()` and `%ADop()`;
    - Array-ize to reduce procedure-call, type-casting, and de-referencing.

Upcoming Developments
and More

- Restructuring of `setup()`
  - Layer out linked-list operations.
  - Single obs. in `setup()` interacts with several *objects*
    - Do <u>*info-table*</u> look-up;
    - Estimate the *guess* → <u>*polymorphic-guess-interpolator*</u>, plus the non-linear operator;
    - Quality-control;
    - Construct the tangent-linear operator, and
    - Store it to its own *observation-type* (*i.e.* `type(someNode)`).
  - "*There is more than one way to skin a cat.*"
- More algorithm possibilities
  - regional-grid support of the re-configurable split-observer mode,
  - in-memory `alltoallv()` of polymorphic obs-types, for on-demand re-distribution to fit different grid partitions,
  - …

Upcoming Developments
Other Considerations

- ***Incremental, bottom-up, and refactoring approach***
  - a working restructuring solution to support required use-cases,
  - frequent releases with assessable impacts, to support concurrent new developments,
  - to ensure continuity, priority, learning curve, and efficiency.
- ***Imperfections will be there along the way***
  - "*Given enough eyeballs, all bugs are shallow.*" So are imperfections.
  - Some forward looking change is better to be introduced incrementally.
  - "*Perfection is the enemy of progress.*"
- ***More efficient testing and repository process***
  - Critical to the productivity of the development.
  - Need portability and performance tests on other platforms.
  - Need feedback, discussions, as well as criticisms from other developers.
- ***From process efficiency for development to process stability for deployment***
  - *Multiple trunk-threads*: developing, integrated, then deployed, evolutionary and staged in phases.
  - *Requirements*: introduced incrementally, from innovative, portable, high performing, to stable.
  - *Maintenance*: defensive improvements, *passive* (fixes only), *preemptive*, or *progressive*.

Thank You!

- Object-Oriented Programming, “***has become recognized as the almost unique successful paradigm for creating complex software.***”

  – Numerical Recipes, 3rd Ed., 2007

- “***Existing languages – notably Fortran, which is arguably still primary language in HPC – proved remarkably adequate.***”

  “***… take existing HPC programs and have someone rewrite them in whatever way suited that individual, … the rewritten code was much more compact and readable than the original, but surprisingly, the ‘ideal’ programming language was basically Fortran.***”

  – The Ideal HPC Programming Language
  Vol. 53, No. 7, Commun. ACM, 2010

- “***Cutting-edge research still universally involves Fortran.***”

  “***Wherever you see giant simulations of the type that run for days on the world’s most massive super computers, you are likely to see Fortran code.***”

  “***These projects are just a few random examples from a large computational universe, but all use some version of Fortran as the main language.***”

  – Scientific Computing’s Future: Can any Coding Language Top a 1950s Behemoth?
  http://arstechnica.com/science, 2014

Appendix
Objectives and Status of this Work

- Support a customized background state grid, for GSI observers

1. **Decouple GSI control-vector grid from guess-state grid**
   - Temporally by-passed in the GSI *split-observer* mode.
2. **Re-configurable observation operators, with respect to GSI guess-state grid and control-vector grid separately**
   - Supported by the implementation of *GSI polymorphic observation types*, in standard Fortran 2003/2008 (Apr., 2016).
   - Merged with then current NCEP/EMC trunk releases, and committed back to the EMC repository for code preview (Jul., 2016).
   - Continuously merged, tested, integrated, and deployed (NASA/GMAO GEOS-DAS).
   - Currently merged up-to the latest NCEP/EMC trunk release *r86502*, and ran EMC *regression-test suite* successfully, on NASA/NCCS *discover* (in review).
3. **Generalize the interpolations of GSI guess state, to support other guess-state grid types**
   - A polymorphic approach in progress, but short of a common baseline to implement.

Appendix
Object-Oriented and Refactoring

- "**Object-Oriented**"?

  Abstraction: defining a named state, use-cases, with interfaces.

  Encapsulation: keeping implementation details private.

  Polymorphism: **"*providing a single interface to entities of different types*"**

  – Bjarne Stroustrup, C++ Glossary

- "**Refactoring**"?

  ***"Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.*

  *Its heart is a series of small behavior preserving transformations. Each transformation (called **a "refactoring"**) does little, but a sequence of transformations can produce a significant restructuring.**"***

  – from www.refactoring.com

Appendix
Benefits of Object-Oriented Implementation

- ***Maintainability***:
  - Benefits of ***abstraction*** and ***encapsulation***, far overweight appeared opaqueness
    - *Usecases-with-interfaces* – improve communication efficiency between code and developers;
    - *Interfaces-vs-implementation-details* – discipline, protect, guide, and ease developments;
    - *Proper object models* evolve slowly – minimize conflicts between concurrent rapidly-changing branches.
- ***Extensibility***:
  - Common traditional approaches for extensions are hard to maintain
    - *Extend-by-insertion* or *extend-by-copy-and-paste* – only seems simple and benign at first.
  - ***Polymorphism*** is a different and better approach
    - [FORTRAN] *hack-ish link-time dispatching* – with implicit interfaces and storage associations;
    - [Fortran-90s] *compilation-time dispatching* – with explicit (strong-typed and generic) interfaces;
    - [Fortran-2Ks] *run-time dispatching* – type-extension plus polymorphic entity, with type-bound-procedures.
- ***Scalability***:
  - In productivity, concurrent developments will be far less tangled.
  - In computational performance,
    - Easier to identify concurrency, threads, and even cache reuses, in good implementations;
    - Often easy to optimize overheads of small procedure-calls, e.g. through *inlining* and *arrayizing*.

Appendix
Polymorphic Guess-State Interpolator

- Assume a generic guess-state in some sub-domain partition, capable of “local” interpolations.
- Observations distribution will adapt to the partition of the guess-state.
- Generic interfaces for interpolations are independent of the guess-state grid definition and its partition.
- First implementation will be for the as-is GSI lat-lon grid.
- Then an extension implementation for GEOS cubed-grid.