Octopus: Coding paradigm

Martin Lüders

Octopus Advanced Course 2023, MPSD Hamburg
Programming paradigms

Programming languages can be classified as:

- **Imperative**: instruct the machine how to change its state
  - Procedural: (old) Fortran, C, etc.
  - Object-oriented: C++, new Fortran, Python, Java, etc.

- **Declarative**: everything is defined as a function: given input leads to result. No side effects.
  - Functional: Mathematica, Lisp, etc.

Note: most programming languages do not exclusively fall into one category.
Procedural programming

- based on procedure call
- procedures combine a series of computational steps
- can be called from everywhere in the program
- change the state of the computer (side effects)
- focus on variables, data structures and procedures
- usually, separation between data and procedures
- Examples: Fortran, ALGOL, COBOL, BASIC, C
Object Orientated Programming (OOP)

- focus on 'objects'
- objects contain the data and methods acting on them
- objects interact with other objects
- objects are instances of classes
- idea: hide internals of an object from the outside world
- Examples: C++, Java, Python, R, modern Fortran, ...
Object Orientated Programming (OOP) in a nutshell

class: 'compound data type' with data specific methods

object: A concrete instance of a class that has properties and methods

encapsulation: Bundles data and methods that operate on the data, hiding internal complexities.

inheritance: Classes can be constructed from a parent, adding or modifying properties or methods

polymorphism: Ability of objects to take on different forms. Allows one interface to be used for a general class of actions.
Object Orientated Programming (OOP): Encapsulation

- Data and also methods can be 'private'
- Access through well defined interface
- Internals can be changed without affecting the rest of the code
Object Orientated Programming (OOP): Inheritance

- Derive new classes from existing ones, extending their functionality
- Example:
  - Parent: classical particle:
    - data: mass, position, velocity
    - methods: accelerate(Force, dt)
  - Child: charged classical particle: only add charge, keep the others
Object Oriented Programming (OOP): Polymorphism

- Same method can execute different behaviour
- Child classes can override methods of parent
- Routines/functions can accept parent or child classes as arguments
- Examples: Propagators
  - abstract propagator: defines method propagate(dt)
  - different implementations, derived from abstract propagator, e.g., verlet, rk4
  - system contains a propagator prop, can be instantiated as verlet, or rk4
  - some function of system calls: prop->propagate(dt)
  - this will work for all propagators defined for that system
Object oriented design (OOD) principles

- Object oriented programming allows for very structured code.
- Important to get the structure right
- Common design principles:
  - **DRY**  don’t repeat yourself
    avoid code duplication
  - **KISS** keep it simple, stupid
    don’t make it more complicated than necessary
  - **SOLID** five more detailed rules, see next slide.
**OOD: SOLID**

- **Single responsibility principle (SRP):**
  A class should have only one responsibility. This makes the code easier to understand and maintain.

- **Open-closed principle (OCP):**
  A class should be open for extension but closed for modification. New features should be added to the class without modifying the existing code.

- **Liskov substitution principle (LSP):**
  Objects of a subclass should be substitutable for objects of their superclass. A subclass should be able to do everything that its superclass can do.

- **Interface segregation principle (ISP):**
  A client should not be forced to depend on methods that it does not use. A interfaces should be split into smaller, more specific interfaces.

- **Dependency inversion principle (DIP):**
  High-level modules should not depend on low-level modules. Instead, both should depend on abstractions. The code should depend on interfaces, not concrete implementations.
OOD: design patterns

For many situations, clear patterns have emerged.

- https://refactoring.guru: Overview of design patterns

- Scientific Software Design: The object oriented way (There is a copy in my office)
Object Orientation in Fortran

Some differences to 'proper' OOP languages (e.g. C++):

- no multiple inheritance
- less sophisticated polymorphism
- no templates! (that’s a pain)

Some further reading:

- FortranWiki
- PGI introduction
Object Orientation in Fortran

• A type is a "class". It can be declared as abstract, or extend another type.

• Polymorphism:
  • Declare an argument as class(..) instead of type(...).
  • In order to access specific behaviour, we need a select type construct.

• Abstract type:
  can not be instantiated

• Deferred binding:
  not implemented in the abstract class; only define the interface for child classes

• Non-overridable:
  methods cannot be overloaded by child classes.
Object Orientation in Fortran: Example linked list

Example: linked_list.F90

- Definition of the class type linked_list_t
- Inheritance: integer_linked_list_t
- Unlimited polymorphism: class(*)
- select type construct:
Object Orientation in Fortran: Example linked list

```fortran
type :: linked_list_t
  private
  integer, public :: size = 0
  class(list_node_t), pointer :: first_node => null()
  class(list_node_t), pointer :: last_node => null()
contains
  procedure :: add_node => linked_list_add_node
  procedure :: add_ptr => linked_list_add_node_ptr
  ...
  generic :: assignment(=) => copy
  procedure :: empty => linked_list_empty
  final :: linked_list_finalize
end type linked_list_t

type :: list_node_t
  private
  class(*), pointer :: value => null()
  type(list_node_t), pointer :: next_node => null()
contains
  procedure :: get => list_node_get
  procedure :: next => list_node_next
  procedure :: set_next => list_node_set_next
  final :: list_node_finalize
end type list_node_t

interface list_node_t
  procedure list_node_constructor
end interface list_node_t
```

- **type bound methods**
- **overloading operators**
- **unlimited polymorphic**
- **pointer**
- **finalizer**
- **constructor**

Octopus: coding paradigm
Object Orientation in Fortran: Example linked list

Example: linked_list.F90

- Inheritance: integer_list

```fortran
  type, extends(linked_list_t) :: integer_list_t
  private
  contains
    procedure :: add => integer_list_add_node
  end type integer_list_t
```

- select type construct:

```fortran
  function integer_iterator_get_next (this) result(value)
    class(integer_iterator_t), intent(inout) :: this
    integer :: value
    select type (ptr => this%get_next_ptr())
      type is (integer)
      value = ptr
    class default
      assert(.false.)
    end select
  end function integer_iterator_get_next
```
Object Orientation in Octopus

Where would we like to use OOP?

- Multisystem framework (more details later)
  - systems
  - interactions
  - propagators
- other 'objects':
  - Hamiltonians
  - operators (derivatives, ...)
  - grids, meshes
- low level objects:
  - mpi groups and communicators
  - linked lists
  - iterators
- This is work in progress...
Coding style

- Set of rules and guidelines for writing code
- Deals with indentation, white spaces, naming conventions, etc
- Makes the code easier to read and understand
- Ideally the code should read like plain English

**Bad**

```fortran
if (space%periodic_dim > 0) then
  ...
end if
```

**Good**

```fortran
if (space%is_periodic()) then
  ...
end if
```

- Helps avoiding some errors
Octopus coding standards

Some examples:

- Two space indentation
- No single letter variable names
- Module names end with `oct_m`, derived types with `_t`
- All functions should go inside modules.
- All modules must have `private` and `implicit none` statements
- Intents for subroutine arguments are mandatory
- ...

Preprocessor

- Changes the source before compilation
- We use the C preprocessor:
  - Standard
  - Widely available
  - Requires some tricks to work with Fortran code
  - Imposes (few) limitations on Fortran code
- Several macros generated when running configure script
- Conditional compilation:
  ```c
  #ifdef HAVE_MPI
  ...
  #else
  ...
  #endif
  ```
- Templating to generate same subroutine with different data types (float/complex/integers, scalar/array, etc)
General code structure

- Use of macros and the preprocessor:
  - `#ifdef ... ` `#endif`
    - conditional compilation
  - `SAFE_ALLOCATE()`
    - calls Fortran `allocate()`
    - returns error on failure
    - counts allocated memory for memory profiling
  - `PUSH_SUB()` `/` `POP_SUB()`
    - generates call stack for debugging
  - ”templating” (see later)
  - currently still many macros (e.g. types); most will be removed.
General code structure: Example

function energy_criterion_constructor(tol_abs, tol_rel, unit) &
  result(crit)

  FLOAT, intent(in) :: tol_abs
  FLOAT, intent(in) :: tol_rel
  type(unit_t), target, intent(in) :: unit
  class(energy_criterion_t), pointer :: crit

  PUSH_SUB(energy_criterion_constructor)

  SAFE_ALLOCATE(crit)

  crit%tol_abs = tol_abs
  crit%tol_rel = tol_rel
  crit%unit => unit
  crit%label = 'energy'

  POP_SUB(energy_criterion_constructor)
end function energy_criterion_constructor
General code structure: "Templating"

- Some object oriented languages (e.g. C++) provide templates: code which is independent of the data type.
- Fortran does not provide templates.
- Octopus uses macros as pseudo-templates:
  - macro X(...) prepends "type-prefix"
  - use macro R_TYPE as templated type in function definition.
- include functions as:

```fortran
#include "undef.F90"
#include "real.F90"
#include "my_function_inc.F90"

#include "undef.F90"
#include "complex.F90"
#include "my_function_inc.F90"
...```

Octopus: coding paradigm
Poor man’s templates

- define functions as:

  ```fortran
  function X(my_function)(arg1, arg2) result(res)
    R_TYPE, intent(in) :: arg1
    R_TYPE, intent(in) :: arg2
    R_TYPE, intent(out) :: res
  end function X(my_function)
  ```

- call functions as:

  ```fortran
  real(real64) :: da1, da2, dres
  complex(real64) :: za1, za2, zres

  dres = dmy_function(da1, da2)
  zres = zmy_function(za1, za2)
  ```
General code structure: "Templating"

- If signatures differ, combine in interface
  
  ```fortran
  interface my_function
    dmy_function, zmy_function
  end interface my_function
  ```

- We can’t always do that.
  Examples: some batch functions
  (numerical type is hidden inside class)